Contents

1 Getting Started
  1.1 Initial Set Up .......................................................... 3
  1.2 Ocean’s Programming Model ......................................... 9
  1.3 D-Wave Compute Resources ......................................... 16
  1.4 Examples .................................................................. 20
  1.5 Demonstrations and Jupyter Notebooks ......................... 72
  1.6 Additional Tutorials .................................................. 72

2 Concepts
  2.1 Binary Quadratic Models ............................................. 73
  2.2 Constraint Satisfaction ............................................... 74
  2.3 Hybrid .................................................................... 75
  2.4 Minor-Embedding ....................................................... 76
  2.5 QPU Topology ............................................................ 80
  2.6 Samplers and Composites ........................................... 82
  2.7 Solutions .................................................................. 83
  2.8 Glossary .................................................................. 84

3 How to Contribute
  3.1 Testing .................................................................... 89
  3.2 Submitting Changes ................................................... 89
  3.3 Coding Conventions ................................................... 90

4 Licenses
  4.1 dwave-ocean-sdk ....................................................... 93
  4.2 dwavebinarycsp ......................................................... 96
  4.3 dwave-cloud-client ................................................... 98
  4.4 dwave-hybrid ............................................................ 101
  4.5 dwave-neal ............................................................... 104
  4.6 dwave-networkx ......................................................... 106
  4.7 dwave-system ........................................................... 109
  4.8 dwave-tabu ............................................................... 112
  4.9 minorminer ............................................................... 114
  4.10 penaltymodel .......................................................... 116
  4.11 dwave-drivers ........................................................ 119
  4.12 dwave-inspector ....................................................... 123
Ocean software is a suite of tools D-Wave Systems provides on the D-Wave GitHub repository for solving hard problems with quantum computers.

- *Getting Started* shows how to install and begin using Ocean tools.
- *Concepts* defines and describes Ocean concepts and terminology.
New to Ocean? The following sections describe how to install Ocean tools, what they are and how they fit together, and give examples of using them to solve hard problems on a D-Wave quantum computer.

1.1 Initial Set Up

D-Wave’s Leap integrated development environment (IDE) is the fastest way to get started writing your quantum application or just learning to use Ocean tools. This cloud-based IDE—run in your browser—is available to all Leap accounts. It provides reusable/disposable workspaces (developer environments pre-configured with Ocean and other standard libraries as well as D-Wave extensions) for running code from your own GitHub repository or a collection of code examples you can then modify.

Alternatively, install the tools and configure for running problems on D-Wave remote compute resources, including quantum-classical hybrid solvers and the D-Wave quantum processing unit (QPU), or locally on your CPU.

1.1.1 Installing Ocean Tools

Ocean software is supported on the following operating systems:

- Linux
- Windows (tested on 64-bit Windows 8, 10)
- Mac (tested on mac OS X 10.13)

Ocean software requires a Python environment. Python versions 3.5 and higher are supported.

**Attention:** D-Wave’s Ocean software stopped supporting Python 2 at the end of 2019.

For information on why many in the Python development community are requiring Python 3, see the Python 3 statement.
This section explains how to install Ocean software, either the entire suite of tools or particular tools from the D-Wave GitHub repositories listed in the navigation bar.

Most Ocean tools require that you configure a solver on your system, which might be a D-Wave system or a classical sampler that runs on your local CPU.

**Python Virtual Environment**

It’s recommended that you work in a virtual environment on your local machine; depending on your operating system, you may need to first install Python and/or `virtualenv`.

1. **Download Python** describes how to install Python on your local machine for supported operating system.
   
   For Unix-based systems, which often have Python pre-installed, installation might be as simple as:
   
   ```bash
   sudo apt-get install python<version>
   ```
   
   **Attention:** For Windows systems, note that only 64-bit Python is supported.

2. **Install virtualenv** describes how to install the `virtualenv` tool for creating isolated Python environments on your local machine for supported operating system.

   For Unix-based systems, installing virtualenv is typically done with a command such as this or similar:
   
   ```bash
   sudo pip install virtualenv
   ```

3. Create a virtual environment for your Ocean work. For example, on Unix systems you might do:

   ```bash
   virtualenv ocean
   source ocean/bin/activate
   ```
   
   (On Windows operating system, activating a virtual environment might be done with the `Scripts\activate` command instead.)

   Your machine is now ready to install Ocean software.

**Install Ocean Software**

The simplest way to start is to install `dwave-ocean-sdk` for the full suite of Ocean tools.

- You can `pip install` the SDK inside your newly created virtual environment, typically with a command such as this or similar:

  ```bash
  pip install dwave-ocean-sdk
  ```

- Alternatively, you can clone `dwave-ocean-sdk` repo and install the SDK to your virtual environment; for example:

  ```bash
  git clone https://github.com/dwavesystems/dwave-ocean-sdk.git
cd dwave-ocean-sdk
python setup.py install
  ```

Note: To install a particular tool within the SDK only, follow the link to the GitHub repository for the tool, as listed in the navigation bar, and follow the installation instructions on the README file.
Set Up Your Environment

For a full and easy development experience it is recommended that before you start writing code, you complete the setup of your environment with two last steps:

- **Install Contributor Ocean Tools**
  Adds non-open-source tools such as the Problem Inspector.

- **Configuring Access to D-Wave Solvers**
  Sets defaults used for accessing D-Wave compute resources.

The dwave-ocean-sdk includes an *interactive CLI* that steps you through setup.

In the virtual environment you created as part of *Installing Ocean Tools*, run the `dwave setup` command. The output shown below includes the interactive prompts and placeholder replies for a full setup. The next sections explain the details.

```
$ dwave setup

Optionally install non-open-source packages and configure your environment.

Do you want to select non-open-source packages to install (y/n)? [y]:

D-Wave Drivers
These drivers enable some automated performance-tuning features.
This package is available under the 'D-Wave EULA' license.
The terms of the license are available online: https://docs.ocean.dwavesys.com/eula
Install (y/n)? [y]:
Installing: D-Wave Drivers
Successfully installed D-Wave Drivers.

D-Wave Problem Inspector
This tool visualizes problems submitted to the quantum computer and the results—returned.
This package is available under the 'D-Wave EULA' license.
The terms of the license are available online: https://docs.ocean.dwavesys.com/eula
Install (y/n)? [y]:
Installing: D-Wave Problem Inspector
Successfully installed D-Wave Problem Inspector.

Creating the D-Wave configuration file.
Configuration file not found; the default location is: /home/jane/.config/dwave/dwave.

Confirm configuration file path [/home/jane/.config/dwave/dwave.conf]:
Profile (create new) [prod]:
API endpoint URL [skip]:
Authentication token [skip]: ABC-1234567890abcdef1234567890abcdef
Default client class (qpu or sw) [qpu]:
Default solver [skip]:
Configuration saved.
```

Install Contributor Ocean Tools

The interactive `dwave setup` and `dwave install` commands of the the dwave-ocean-sdk step you through installation of non-open-source (“contrib”) tools.
If you did not install contributor packages with the `dwave setup` command in the Set Up Your Environment section, or want to add packages at a later time, you can use it again then or use the `dwave install` command.

```
$ dwave install --help
Usage: dwave install [OPTIONS] [PACKAGES]...

    Install optional non-open-source Ocean packages.

Options:
-1, --list       List available contrib (non-OSS) packages
-a, --all        Install all contrib (non-OSS) packages
-v, --verbose    Increase output verbosity
--help           Show this message and exit.
```

Both commands describe the tools and enable you to select which if any to install.

Most Ocean tools solve problems on a **solver**, which is a compute resource such as a D-Wave system or CPU, and might require that you configure a default solver. Configuring Access to D-Wave Solvers describes the next step of setting up your environment, how to configure your system to access D-Wave or other remote solvers.

### 1.1.2 Configuring Access to D-Wave Solvers

D-Wave’s Solver API (SAPI) provides access to **solvers**, remote compute resources for solving problems such as a D-Wave system or Leap’s quantum-classical hybrid solvers.

#### Interacting with SAPI

SAPI is an application layer built to provide resource discovery, permissions, and scheduling for D-Wave compute resources. The requisite information for problem submission through SAPI includes:

1. **API endpoint URL**
   
   A URL to the remote resources. By default, `https://cloud.dwavesys.com/sapi` is used to connect to resources provided by D-Wave’s Leap quantum cloud service, including D-Wave quantum computers.

2. **API Token**

   An authentication token used to authenticate the client session when you connect to the remote environment. Because tokens provide authentication, user names and passwords are not required in your code.

3. **Solver**

   A D-Wave resource to be used to solve your submitted problems; for example, a hybrid solver or a D-Wave 2000Q quantum computer.

You can find all the above information when you log in to your D-Wave account. For Leap users, select the Dashboard tab; for on-premises (Qubist) users, select the Solver API tab and the API Tokens menu item under your user name.

You save your SAPI configuration (URL, API token, etc) in a **D-Wave Cloud Client configuration file** that Ocean tools use unless overridden explicitly or with environment variables. Your configuration file can include one or more solvers.

**Note:** When you work in D-Wave’s Leap IDE, SAPI information such as your API token is pre-configured in the default workspace’s environment variables.
Creating a Configuration File

The simplest way to configure solver access is to use the *interactive CLI*, which is installed as part of the `dwave-ocean-sdk` installation.

If you did not already do so with the `dwave setup` command in the *Set Up Your Environment* section, or want to make changes at a later time, you can use the `dwave config` command.

```bash
$ dwave config --help
Usage: dwave config [OPTIONS] COMMAND [ARGS]...
Create, update or inspect cloud client configuration file(s).

Options:
   --help Show this message and exit.

Commands:
   create    Create and/or update cloud client configuration file.
   inspect   Inspect existing configuration/profile.
   ls        List configuration files detected (and/or examined paths).
```

Creating a configuration file using the `dwave config` is done as follows (the `dwave setup` command of the *Set Up Your Environment* section runs these same configuration steps):

1. In the virtual environment you created as part of *Installing Ocean Tools*, run the `dwave config create` command (the output shown below includes the interactive prompts and placeholder replies).

```bash
$ dwave config create
Configuration file not found; the default location is: /home/jane/.config/dwave/dwave.
  → conf
Confirm configuration file path [/home/jane/.config/dwave/dwave.conf]:
Profile (create new) [prod]:
API endpoint URL [skip]:
Authentication token [skip]: ABC-1234567890abcdef1234567890abcdef
Default client class (qpu or sw) [qpu]:
Default solver [skip]:
Configuration saved.
```

2. Enter the SAPI information (e.g. your API token) found as described in the section above. To get started, create a minimum configuration by accepting the command’s defaults (pressing Enter) for all prompts except the API token (Leap users) or API token and endpoint (on-premises users). You can in the future update the file if needed.

Alternatively, you can create and edit a *D-Wave Cloud Client configuration file* manually.

You can always set or override the solver, API token, and URL directly in your code or as local environment variables. For more information, see the examples in this document or *D-Wave Cloud Client*.

Verifying Your Configuration

You can test that your solver access is configured correctly with the *interactive CLI*.

1. In your virtual environment, run the `dwave ping` command (the output shown below is illustrative only).

```bash
$ dwave ping
Using endpoint: https://cloud.dwavesys.com/sapi
Using solver: My_DWAVE_2000Q
```

(continues on next page)
Wall clock time:
- Solver definition fetch: 2007.239 ms
- Problem submit and results fetch: 1033.931 ms
- Total: 3041.171 ms

QPU timing:
- total_real_time = 10493 us
- anneal_time_per_run = 20 us
- post_processing_overhead_time = 128 us
- qpu_anneal_time_per_sample = 20 us

# Snipped for brevity

2. Optionally, run the `dwave sample --random-problem` command to submit a random problem to a remote solver (the output shown below is illustrative only).

```bash
$ dwave sample --random-problem
Using endpoint: https://my.dwavesys.url/
Using solver: My_DWAVE_2000Q
Using qubit biases: {0: -1.0345257941434953, 1: -0.5795618633919246, 2: -0.9721956399428491, 3: 1.0598893924967594...
Using qubit couplings: {1634, 1638}: 0.721736584181423, (587, 590): 0.9611623181258304, (642, 64...
Number of samples: 1
Samples: [[1, 1, -1, -1, -1, -1, 1, -1, -1, -1, -1, -1, 1, -1, -1, -1, -1, -1, -1, -1, -1, 1, -1, 1, 1, -1, -1, -1, -1, -1, -1, -1, ...
Occurrences: [1]
Energies: [-2882.197791239335]
```

### Querying Available Solvers

**Note:** Leap accounts can see accessible solvers on the dashboard.

From your terminal, you can use the `interactive CLI` to see the available solvers, their parameters, and properties.

1. Run the `dwave solvers` command (the output shown below is illustrative only).

```bash
$ dwave solvers
Solver: DW_2000Q_33
Parameters:
  anneal_offsets: A list of anneal offsets for each working qubit (NaN if u...
  anneal_schedule: A piecewise linear annealing schedule specified by a list...
  annealing_time: A positive integer that sets the duration (in microsecond...<Output snipped for brevity>

Properties:
  anneal_offset_ranges: [[-0.18627387668142237, 0.09542224439071689], [-0.1836548...
  anneal_offset_step: 0.00456479499507194
  anneal_offset_step_phi0: 0.0002716837027763096
  annealing_time_range: [1, 150000]
```

(continues on next page)
Alternatively, from within your code or a Python interpreter you can query solvers available for a SAPI URL and API token using `dwave-cloud-client get_solvers()` function. For example, the code below queries available solvers for your default SAPI URL and a specified token.

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config(token='ABC-123456789123456789123456789') # doctest: +SKIP
>>> client.get_solvers() # doctest: +SKIP
[Solver(id='2000Q_ONLINE_SOLVER1'),
 UnstructuredSolver(id='hybrid_v1')]
```

Typically, once you have selected and configured a solver, your code queries its parameters and properties as attributes of the instantiated solver object. The code example below (with output snipped for brevity) sets a D-Wave system as the sampler, using the default SAPI configuration as set above, and queries its parameters.

```python
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler(solver={'gpu': True})
>>> sampler.parameters # doctest: +SKIP
{u'anneal_offsets': [u'parameters'],
 u'anneal_schedule': [u'parameters'],
 u'annealing_time': [u'parameters'],
 u'answer_mode': [u'parameters'],
 u'auto_scale': [u'parameters'], ...
```

Descriptions of D-Wave system parameters and properties are in the system documentation.

## 1.2 Ocean’s Programming Model

Learn Ocean software’s workflow for problem solving.

### 1.2.1 How a D-Wave System Solves Problems

This section explains some of the basics of how you can use D-Wave quantum computers to solve problems and how Ocean tools can help.

For quantum computing, as for classical, solving a problem requires that it be formulated in a way the computer and its software understand.

For example, if you want your laptop to calculate the area of a $1 coin, you might express the problem as an equation, \( A = \pi r^2 \), that you program as `math.pi*13.245**2` in your Python CLI. For a laptop with Python software, this formulation—a particular string of alphanumeric symbols—causes the manipulation of bits in a CPU and memory chips that produces the correct result.

The D-Wave system uses a quantum processing unit (QPU) to solve a binary quadratic model (BQM)\(^1\): given \( N \) variables \( x_1, \ldots, x_N \), where each variable \( x_i \) can have binary values 0 or 1, the system finds assignments of values that

---

\(^1\) The “native” forms of BQM programmed into a D-Wave system are the Ising model traditionally used in statistical mechanics and its computer-science equivalent, shown here, the QUBO.
minimize

\[ \sum_{i}^{N} q_{i} x_{i} + \sum_{i<j}^{N} q_{i,j} x_{i} x_{j} \]

where \( q_{i} \) and \( q_{i,j} \) are configurable (linear and quadratic) coefficients. To formulate a problem for the D-Wave system is to program \( q_{i} \) and \( q_{i,j} \) so that assignments of \( x_{1}, \ldots, x_{N} \) also represent solutions to the problem.

Ocean software can abstract away much of the mathematics and programming for some types of problems. At its heart is a binary quadratic model (BQM) class that together with other Ocean tools helps formulate various optimization problems. It also provides an API to binary quadratic samplers (the component used to minimize a BQM and therefore solve the original problem), such as the D-Wave system and classical algorithms you can run on your computer.

The following sections describe this problem-solving procedure in two steps (plus a third that may benefit some problems); see the Getting Started Examples section and System Documentation for further description.

1. **Formulate Your Problem for a Quantum Computer.**
2. **Sample the BQM on a Solver.**
3. **Improve the Solutions**, if needed, using advanced features.

![Diagram](image)

Fig. 1: Solution steps: (1) a problem known in “problem space” (a circuit of Boolean gates, a graph, a network, etc) is formulated as a BQM, mathematically or using Ocean functionality and (2) the BQM is sampled for solutions.

### 1.2.2 Formulate Your Problem for a Quantum Computer

There are different ways of mapping between a problem—chains of amino acids forming 3D structures of folded proteins, traffic in the streets of Beijing, circuits of binary gates—and a BQM to be solved (by sampling) with a D-Wave system or locally on your CPU/GPU.

For example, consider the problem of determining outputs of a Boolean logic circuit. In its original context (in “problem space”), the circuit might be described with input and output voltages, equations of its component resistors, transistors, etc, an equation of logic symbols, multiple or an aggregated truth table, and so on. You can choose to use Ocean software to formulate BQMs for binary gates directly in your code or mathematically formulate a BQM, and both can be done in different ways too; for example, a BQM for each gate or one BQM for all the circuit’s gates.

The following are two example formulations.
1. The **Boolean NOT Gate** example, takes a NOT gate represented symbolically as \(x_2 \iff \neg x_1\) and formulates it mathematically as the following BQM:

\[-x_1 - x_2 + 2x_1x_2\]

The table below shows that this BQM has lower values for valid states of the NOT gate (e.g., \(x_1 = 0, x_2 = 1\)) and higher for invalid states (e.g., \(x_1 = 0, x_2 = 0\)).

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>Valid?</th>
<th>BQM Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>No</td>
<td>1</td>
</tr>
</tbody>
</table>

2. Ocean’s *dwavebinarycsp* tool enables the following formulation of an AND gate as a BQM:

```python
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.and_gate(['x1', 'x2', 'y1']))  # add an AND gate
>>> bqm = dwavebinarycsp.stitch(csp)
```

The resultant BQM of this AND gate may look like this:

```python
>>> bqm  # doctest: +SKIP
BinaryQuadraticModel({'x1': 0.0, 'x2': 0.0, 'y1': 6.0},
... {( 'x2', 'x1'): 2.0, ('y1', 'x1'): -4.0, ('y1', 'x2'): -4.0},
... 0,
... 'BINARY')
```

The members of the two dicts are linear and quadratic coefficients, respectively, the third term is a constant offset associated with the model, and the fourth shows the variable types in this model are binary.

For more detailed information on the parts of Ocean programming model and how they work together, see *Ocean Software Stack*.

Once you have a BQM that represents your problem, you sample it for solutions. *Solving Problems by Sampling* explains how to submit your problem for solution.

### 1.2.3 Ocean Software Stack

The Ocean software stack provides a chain of tools that implements the steps needed to solve your problem on a CPU/GPU or a D-Wave system. As described in the *How a D-Wave System Solves Problems* section, these steps include formulating the problem in a way the quantum computer understands (as a *binary quadratic model*) and solving the formulated problem by submitting it to a D-Wave system or classical *sampler* (the component used to minimize a BQM and therefore solve the original problem).

It’s helpful to visualize the tool chain as layers of abstraction, each of which handles one part of the solution procedure.

**Abstraction Layers**

The *Ocean Software Stack* graphic above divides Ocean software and its context into the following layers of functionality:

---

1.2. Ocean’s Programming Model
Fig. 2: Ocean Software Stack
• Compute Resources
  The hardware on which the problem is solved. This might be a D-Wave quantum processor but it can also be
  the CPU of your laptop computer.

• Samplers
  Abstraction layer of the sampler functionality. Ocean tools implement several samplers that use the D-Wave
  system and classical compute resources. You can use the Ocean tools to customize a D-Wave sampler, create
  your own sampler, or use existing (classical) samplers to your code as you develop it.

• Sampler API
  Abstraction layer that represents the problem in a form that can access the selected sampler; for example, a di-
  mod binary quadratic model (BQM) class representing your problem wrapped in a minor-embedding composite
  that handles the mapping between your problem’s variables and the sampler’s graph.

• Methods
  Tools that help formulate a problem as binary quadratic models; for example dwave_networkx (repo) for graph-
  related problems.

• Application
  Original problem in its context (“problem space”); for example, circuit fault diagnosis attempts to identify failed
  logic gates during chip manufacturing.

**Problem-to-Solution Tool Chain**

As described in the *How a D-Wave System Solves Problems* section, problems can be posed in a variety of formulations;
the D-Wave system solves binary quadratic models. Ocean tools assist you in converting the problem from its original
form to a form native to the D-Wave system and sending the compatible problem for solving.

This section will familiarize you with the different tools and how you can fit them together to solve your problem.

**Bottom-Up Approach**

One approach to envisioning how you can map your problem-solving process to Ocean software is to start from the
bottom—the hardware doing the computations—and work your way up the Ocean stack to see the complete picture.
This section shows how you might map each stage of the process to a layer of the Ocean stack.

1. **Compute resource**
   You will likely use some combination of both local classical resources and a D-Wave system in your work with
   Ocean software. When would you use which?
   - CPU/GPU: for offline testing, small problems that can be solved exactly or heuristically in a reasonable
     amount of time.
   - QPU: hard problems or for learning how to use quantum resources to solve such problems.
   - Hybrid of both QPU and CPU/GPU: large, complex problems that need to run classically but may benefit
     from having some parts allocated to a quantum computer for solution.

2. **Sampler**
   Your sampler provides access to the compute resource that solves your problem.

   The table below shows some Ocean samplers and considerations for selecting one or another.
### Table 2: Ocean Samplers

<table>
<thead>
<tr>
<th>Computation &amp; Sampler</th>
<th>Usage</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical dimod ExactSampler()</td>
<td>Find all states for small (&lt;20 variables) problems.</td>
<td>For code-development testing.</td>
</tr>
<tr>
<td>Classical dimod RandomSampler()</td>
<td>Random sampler for testing.</td>
<td>For code-development testing.</td>
</tr>
<tr>
<td>Classical dimod SimulatedAnnealingSampler()</td>
<td>Simulated annealing sampler for testing.</td>
<td>For code-development testing.</td>
</tr>
<tr>
<td>Classical dwave-neal SimulatedAnnealingSampler()</td>
<td>Simulated annealing sampler.</td>
<td></td>
</tr>
<tr>
<td>Quantum dwave-system DWaveSampler()</td>
<td>Quick incorporation of the D-Wave system as a sampler.</td>
<td>Typically part of a composite that handles minor-embedding.</td>
</tr>
<tr>
<td>Quantum dwave-cloud-client Solver()</td>
<td>D-Wave system as a sampler.</td>
<td>For low-level control of problem submission.</td>
</tr>
<tr>
<td>Hybrid dwave-hybrid KerberosSampler()</td>
<td>dimod-compatible hybrid asynchronous decomposition sampler.</td>
<td>For problems of arbitrary structure and size.</td>
</tr>
<tr>
<td>Hybrid Leap's LeapHybridSampler()</td>
<td>Cloud-based quantum-classical hybrid sampler.</td>
<td>For problems of arbitrary structure and size, especially large problems.</td>
</tr>
<tr>
<td>dimod custom Write a custom sampler for special cases.</td>
<td>See examples in dimod.</td>
<td></td>
</tr>
</tbody>
</table>

### 3. Pre- and Post-Processing

Samplers can be composed of composite patterns that layer pre- and post-processing to binary quadratic programs without changing the underlying sampler.

The table below shows some Ocean composites and considerations for selecting one or another.

#### Table 3: Ocean Composites

<table>
<thead>
<tr>
<th>Tool &amp; Composite</th>
<th>Usage</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwave-system EmbeddingComposite()</td>
<td>Maps unstructured problems to a structured sampler.</td>
<td>Enables quick incorporation of the D-Wave system as a sampler by handling the minor-embedding to the QPU’s Chimera topology of qubits.</td>
</tr>
<tr>
<td>dwave-system FixedEmbeddingComposite()</td>
<td>Maps unstructured problems to a structured sampler.</td>
<td>Uses a pre-calculated minor-embedding for improved performance.</td>
</tr>
<tr>
<td>dwave-system TilingComposite()</td>
<td>Tiles small problems multiple times to a Chimera-structured sampler.</td>
<td>Enables parallel sampling for small problems.</td>
</tr>
<tr>
<td>dwave-system VirtualGraphComposite()</td>
<td>Uses the D-Wave virtual graph feature for improved minor-embedding.</td>
<td>Calibrates qubits in chains to compensate for the effects of biases and enables easy creation, optimization, use, and reuse of an embedding for a given working graph.</td>
</tr>
<tr>
<td>dimod SpinReversalTransformComposite()</td>
<td>Applies spin reversal transform preprocessing.</td>
<td>Improves QPU results by reducing the impact of possible analog and systematic errors.</td>
</tr>
<tr>
<td>dimod StructuredUnstructuredSampler()</td>
<td>Creates a structured composed sampler from an unstructured sampler.</td>
<td>Maps from a problem graph (e.g., a square graph) to a sampler’s graph.</td>
</tr>
</tbody>
</table>

---

1. This sampler is for low-level work on communicating with SAPI and is not a dimod sampler.
In addition to composites that provide pre- and post-processing, Ocean also provides stand-alone tools to handle complex or large problems. For example:

- **minorminer** for minor-embedding might be used to improve solutions by fine tuning parameters or incorporating problem knowledge into the embedding.
- **qbsolv** splits problems too large for the QPU into pieces solved either via a D-Wave system or a classical tabu solver.

4. **Map to a Supported Format**

Typically, you formulate your problem as a binary quadratic model (BQM), which you solve by submitting to the sampler (with its pre- and post-processing composite layers) you select based on the considerations listed above.

Ocean provides tools for formulating the BQM:

- **dwavebinarycsp** for constraint satisfaction problems with small constraints over binary variables. For example, many problems can be posed as satisfiability problems or with Boolean logic.
- **dwave_networkx** for implementing graph-theory algorithms of the D-Wave system. Many problems can be posed in a form of graphs—this tool handles the construction of BQMs for several standard graph algorithms such as maximum cut, cover, and coloring.

You might formulate a BQM mathematically; see *Boolean NOT Gate* for a mathematical formulation for a two-variable problem.

See the system documents for more information on techniques for formulating problems as BQMs.

5. **Formulate**

The first step in solving a problem is to express it in a mathematical formulation. For example, the *Map Coloring* problem is to assign a color to each region of a map such that any two regions sharing a border have different colors. To begin solving this problem on any computer, classical or quantum, it must be concretely defined; an intuitive approach, for the map problem, is to think of the regions as variables representing the possible set of colors, the values of which must be selected from some numerical scheme, such as natural numbers.

The selection function must express the problem’s constraints:

- Each region is assigned one color only, of $C$ possible colors.
- The color assigned to one region cannot be assigned to adjacent regions.

Now solving the problem means finding a permissible value for each of the variables.

When formulating a problem for the D-Wave system, bear in mind a few considerations:

- Mathematical formulations must use binary variables because the solution is implemented physically with qubits, and so must translate to spins $s_i \in \{1, +1\}$ or equivalent binary values $x_i \in \{0, 1\}$.
- Relationships between variables must be reducible to quadratic (e.g., a QUBO) because the problem’s parameters are represented by qubits’ weights and couplers’ strengths on a QPU.
- Formulations should be sparing in its number of variables because a QPU has a limited number of qubits and couplers.
- Alternative formulations may have different implications for performance.

Ocean demo applications, which formulate known problems, include:

- **Structural Imbalance**.
- **Circuit-Fault Diagnosis**.
Top-Down Approach

Another approach to envisioning how you can map your problem-solving process to Ocean software is to start from the top—your (possibly abstractly defined) problem—and work your way down the Ocean stack.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State the Problem</td>
<td>Define your problem concretely/mathematically; for example, as a constraint satisfaction problem or a graph problem.</td>
</tr>
<tr>
<td>Formulate as a BQM</td>
<td>Reformulate an integer problem to use binary variables, for example, or convert a nonquadratic (high-order) polynomial to a QUBO. Ocean’s <code>dwavebinarycsp</code> and <code>dwave_networkx</code> can be helpful for some problems.</td>
</tr>
<tr>
<td>Decompose</td>
<td>Allocate large problems to classical and quantum resources. Ocean’s <code>dwave-hybrid</code> provides a framework and building blocks to help you create hybrid workflows.</td>
</tr>
<tr>
<td>Embed</td>
<td>Consider whether your problem has repeated elements, such as logic gates, when deciding what tool to use to minor-embed your BQM on the QPU. You might start with fully automated embedding (using <code>EmbeddingComposite()</code> for example) and then seek performance improvements through <code>minnorminer</code>.</td>
</tr>
<tr>
<td>Configure the QPU</td>
<td>Use spin-reversal transforms to reduce errors, for example, or examine the annealing with reverse anneal. See the system documents for more information of features that improve performance.</td>
</tr>
</tbody>
</table>

1.3 D-Wave Compute Resources

Use Ocean’s `samplers` to solve problems on D-Wave’s compute resources (`solvers`) or locally on your CPU.

1.3.1 Solving Problems by Sampling

Having followed the steps of the Formulate Your Problem for a Quantum Computer section, you sample the BQM that now represents your problem for solutions. Ocean software provides quantum, classical, and quantum-classical hybrid `samplers` that run either remotely (for example, in D-Wave’s Leap environment) or locally on your CPU. These compute resources are known as `solvers`.

Note: Some classical samplers actually brute-force solve small problems rather than sample, and these are also referred to as “solvers”.

Sample the BQM on a Solver

Ocean’s `samplers` enable you to submit your problem to remote or local compute resources (`solvers`) of different types:

- **Hybrid Solvers** such as Leap’s `hybrid_v1` solver
- **Classical Solvers** such as `dimod.ExactSolver` for exact solutions to small problems
- **Quantum Solvers** such a D-Wave 2000Q system.

The example code below submits the BQM of the AND gate of the Formulate Your Problem for a Quantum Computer section.
to a Leap hybrid solver. In this case, `dwave-system`'s `dwave.system.LeapHybridSampler` is the Ocean sampler and the remote compute resource selected might be Leap hybrid solver `hybrid_v1`.

```python
>>> from dwave.system import LeapHybridSampler
>>> sampler = LeapHybridSampler()
# doctest: +SKIP
>>> answer = sampler.sample(bqm)  # doctest: +SKIP
>>> print(answer)                 # doctest: +SKIP
x1 x2 y1 energy num_oc.
0 0 1 0 0.0 1
['BINARY', 1 rows, 1 samples, 3 variables]
```

### Improve the Solutions

More complex problems than the ones shown above can benefit from some of the D-Wave system's advanced features and Ocean software's advanced tools.

When sampling directly on the D-Wave QPU, the mapping from problem variables to qubits, `minor-embedding`, can significantly affect performance. Ocean tools perform this mapping heuristically so simply rerunning a problem might improve results. Advanced users may customize the mapping by directly using the `minorminer` tool, setting a minor-embedding themselves, or using D-Wave’s problem-inspector tool.

For example, the `Boolean AND Gate` example submits the BQM representing an AND gate to a D-Wave system, which requires mapping the problem’s logical variables to qubits on the QPU. The code below invokes D-Wave’s problem-inspector tool to visualize the minor-embedding.

```python
>>> import dwave.inspector
>>> dwave.inspector.show(response)  # doctest: +SKIP
```

D-Wave systems offer features such as spin-reversal (gauge) transforms and anneal offsets, which reduce the impact of possible analog and systematic errors.

You can see the parameters and properties a sampler supports. For example, Ocean’s `dwave-system` lets you use the D-Wave’s `virtual graphs` feature to simplify minor-embedding. The following example maps a problem’s variables `x`, `y` to qubits 1, 5 and variable `z` to two qubits 0 and 4, and checks some features supported on the D-Wave system used as a sampler.

```python
>>> from dwave.system import DWaveSampler
>>> from dwave.system.composites import VirtualGraphComposite
>>> DWaveSampler(solver={'qpu': True}).properties['extended_j_range']
[-2.0, 1.0]
>>> embedding = {'x': {1}, 'y': {5}, 'z': {0, 4}}
>>> sampler = VirtualGraphComposite(DWaveSampler(solver={'qpu': True}), embedding)  # doctest: +SKIP
>>> sampler.parameters  # doctest: +SKIP
```

**Attention:** D-Wave’s `virtual graphs` feature can require many seconds of D-Wave system time to calibrate qubits to compensate for the effects of biases. If your account has limited D-Wave system access, consider using `FixedEmbeddingComposite()` instead.
Fig. 3: View of the logical and embedded problem rendered by Ocean’s problem inspector. The AND gate’s original BQM is represented on the left; its embedded representation, on the right, shows a two-qubit chain (qubits 176 and 180) for variable $x_2$. The tool is helpful in visualizing the quality of your embedding.

Note that the composed sampler (VirtualGraphComposite() in the last example) inherits properties from the child sampler (DWaveSampler() in that example).

See the resources under Additional Tutorials and the System Documentation for more information.

1.3.2 Hybrid Solvers

Quantum-classical hybrid is the use of both classical and quantum resources to solve problems, exploiting the complementary strengths that each provides. For an overview of, and motivation for, hybrid computing, see: Medium Article on Hybrid Computing.

Ocean software currently supports two types of hybrid solvers:

- **Leap’s Hybrid Solvers** are cloud-based hybrid compute resources.
- **dwave-hybrid Hybrid Solvers** are hybrid solvers developed in Ocean’s dwave-hybrid tool.

Leap’s Hybrid Solvers

D-Wave’s Leap quantum cloud service provides cloud-based hybrid solvers you can submit arbitrary BQMs to. These solvers, which implement state-of-the-art classical algorithms together with intelligent allocation of the quantum processing unit (QPU) to parts of the problem where it benefits most, are designed to accommodate even very large
problems. Leap’s solvers can relieve you of the burden of any current and future development and optimization of hybrid algorithms that best solve your problem.

*Structural Imbalance in a Social Network* is an example of submitting a problem for solution on a Leap hybrid solver.

**dwave-hybrid Hybrid Solvers**

*dwave-hybrid* provides you with a Python framework for building a variety of flexible hybrid workflows. These use quantum and classical resources together to find good solutions to your problem. For example, a hybrid workflow might use classical resources to find a problem’s hard core and send that to the QPU, or break a large problem into smaller pieces that can be solved on a QPU and then recombined.

The *dwave-hybrid* framework enables rapid development of experimental prototypes, which provide insight into expected performance of the productized versions. It provides reference samplers and workflows you can quickly plug into your application code. You can easily experiment with customizing workflows that best solve your problem. You can also develop your own hybrid components to optimize performance.

*Large Map Coloring* and *Problem With Many Variables* are examples of solving problems using *dwave-hybrid* samplers.

### 1.3.3 Classical Solvers

You might use a classical solver while developing your code or on a small version of your problem to verify your code. To solve a problem classically on your local machine, you configure a classical solver, either one of those included in the Ocean tools or your own.

#### Examples

Among several samplers provided in the *dimod* tool for testing your code locally, is the *ExactSolver()* that calculates the energy of all possible samples for a given problem. Such a sampler can solve a small three-variable problem like the AND gate of the *Formulate Your Problem for a Quantum Computer* section,

```python
>>> import dimod
>>> bqm = dimod.BinaryQuadraticModel({'x1': 0.0, 'x2': 0.0, 'y1': 6.0},
...    {('x2', 'x1'): 2.0, ('y1', 'x1'): -4.0, ('y1', 'x2'): -4.0},
...    0, 'BINARY')
```

as follows:

```python
>>> from dimod.reference.samplers import ExactSolver
>>> sampler = ExactSolver()
>>> response = sampler.sample(bqm)
>>> print(response) # doctest: +SKIP
['BINARY', 8 rows, 8 samples, 3 variables]
```
Note that the first four samples are the valid states of the AND gate and have lower values than the second four, which represent invalid states.

If you use a classical solver running locally on your CPU, a single sample might provide the optimal solution.

This example solves a two-variable problem using the *dwave_neal* simulated annealing sampler. For such a small problem, *num_reads=10* most likely finds the optimal solution.

```python
>>> import neal
>>> solver = neal.SimulatedAnnealingSampler()
>>> sampleset = solver.sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b'): -1}, num_reads=10)
>>> sampleset.first.sample['a'] == sampleset.first.sample['b'] == -1
True
```

### 1.3.4 Quantum Solvers

Ocean’s *dwave-system* tool enables you to use a D-Wave system as a sampler. In addition to *DWaveSampler()*, the tool provides a *EmbeddingComposite()* composite that maps unstructured problems to the graph structure of the selected sampler, a process known as *minor-embedding*. For the AND gate of the *Formulate Your Problem for a Quantum Computer* section,

```python
>>> import dimod
>>> bqm = dimod.BinaryQuadraticModel({'x1': 0.0, 'x2': 0.0, 'y1': 6.0},
... {('x2', 'x1'): 2.0, ('y1', 'x1'): -4.0, ('y1', 'x2'): -4.0},
... 0, 'BINARY')
```

the problem is defined on alphanumeric variables $x_1, x_2, y_1$, that must be mapped to the QPU’s numerically indexed qubits.

Because of the sampler’s probabilistic nature, you typically request multiple samples for a problem; this example sets *num_reads* to 1000.

```python
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> sampleset = sampler.sample(bqm, num_reads=1000)
>>> print(sampleset)
# doctest: +SKIP
x1 x2 y1 energy num_oc. chain_b.
0 1 0 0 0.0 321 0.0
1 1 1 1 0.0 97 0.0
2 0 0 0 0.0 375 0.0
3 0 1 0 0.0 206 0.0
4 1 0 1 2.0 1 0.333333
['BINARY'], 5 rows, 1000 samples, 3 variables]
```

Note that the first four samples are the valid states of the AND gate and have lower energy than invalid state $x_1 = 1, x_2 = 0, y_1 = 1$.

Once you have configured a *D-Wave Cloud Client configuration file* as described in the *Configuring Access to D-Wave Solvers* section, your default solver configuration is used when you submit a problem without explicitly overriding it.

Several of the examples in the *Getting Started* Example’s section show how to submit problems to D-Wave systems.

### 1.4 Examples

See how Ocean tools are used with these end-to-end examples.
Because many large, hard problems are best approached with quantum-classical hybrid solvers, a good place to start is with examples of the Beginner-Level Examples: Hybrid Computing section and then learn how to work directly on the quantum computer with examples of the Beginner-Level Examples: Using the QPU section.

1.4.1 Beginner-Level Examples: Hybrid Computing

Structural Imbalance in a Social Network

This example solves a structural-imbalance problem, similar to the Leap demo and Jupyter Notebook, to demonstrate using Leap’s hybrid solver service on a problem of arbitrary structure and size.

Social networks map relationships between people or organizations onto graphs, with the people/organizations as nodes and relationships as edges; for example, Facebook friends form a social network. Signed social networks map both friendly and hostile relationships by assigning to edges either positive or negative values. Such networks are said to be structurally balanced when they can be cleanly divided into two sets, with each set containing only friends, and all relations between these sets are hostile. The measure of structural imbalance or frustration for a signed social network, when it cannot be cleanly divided, is the minimum number of edges that violate the social rule, “the enemy of my friend is my enemy.”

Fig. 4: Juliet’s new love of Romeo introduces imbalance into the social network of Verona. Green edges represent friendly relationships (Juliet & Romeo and Juliet & Lord Capulet) while red edges represent hostile relationships (Romeo and Lord Capulet). The black vertical line dividing the set with Romeo from the set with Lord Capulet crosses the friendly edge between Juliet and Lord Capulet.

Finding a division that minimizes frustration is an NP-hard graph problem (it can be viewed as an expansion of the well-known maximum cut problem).
Example Requirements

To run the code in this example, the following is required.

- The requisite information for problem submission through SAPI, as described in *Configuring Access to D-Wave Solvers*.
- Ocean tools `dwave-system` and `dwave_networkx`.

If you installed `dwave-ocean-sdk` and ran `dwave setup`, your installation should meet these requirements. In D-Wave’s Leap IDE, the default workspace meets these requirements.

Solution Steps

Section *How a D-Wave System Solves Problems* describes the process of solving problems on the quantum computer in two steps: (1) Formulate the problem as a binary quadratic model (BQM) and (2) Solve the BQM with a D-wave system, classical sampler, or hybrid sampler. In this example, a function in Ocean software handles both steps. Our task is mainly to select the sampler used to solve the problem.

Formulate the Problem

For a social graph, $G$, this example simply builds a random sparse graph—using the `networkx` function, which places uniformly at random a specified number of nodes, `problem_node_count`, in a unit cube, joining edges of any two if the distance is below a given radius—and randomly assigns 0, 1 signs to represent friendly and hostile relationships.

```python
>>> import networkx as nx
>>> import random

>>> problem_node_count = 300

>>> G = nx.random_geometric_graph(problem_node_count, radius=0.0005*problem_node_count)

>>> G.add_edges_from([(u, v, {'sign': 2*random.randint(0, 1)-1}) for u, v in G.edges])
```

Solve the Problem by Sampling

As mentioned above, this example uses Ocean’s `dwave_networkx` function, `structural_imbalance()`, to create the appropriate BQM to represent the problem graph and return a solution. It requires just the selection of a sampler.

D-Wave’s quantum cloud service provides cloud-based hybrid solvers you can submit arbitrary BQMs to. These solvers, which implement state-of-the-art classical algorithms together with intelligent allocation of the quantum processing unit (QPU) to parts of the problem where it benefits most, are designed to accommodate even very large problems. Leap’s solvers can relieve you of the burden of any current and future development and optimization of hybrid algorithms that best solve your problem.

Ocean software’s `dwave-system LeapHybridSampler` class enables you to easily incorporate Leap’s hybrid solvers into your application:

```python
>>> from dwave.system import LeapHybridSampler
>>> sampler = LeapHybridSampler()  # doctest: +SKIP
```

Finally, the returned set of frustrated edges and a bicoloring are counted and printed.
The graphic below visualizes the result of one such run.

Large Map Coloring

This example solves a map coloring problem to demonstrate an out-of-the-box use of Ocean’s classical-quantum hybrid sampler, dwave-hybrid Kerberos, that enables you to solve problems of arbitrary structure and size.

Map coloring is an example of a constraint satisfaction problem (CSP). CSPs require that all a problem’s variables be assigned values, out of a finite domain, that result in the satisfying of all constraints. The map-coloring CSP is to assign a color to each region of a map such that any two regions sharing a border have different colors.

The Map Coloring advanced example demonstrates lower-level coding of a similar problem, which gives the user more control over the solution procedure but requires the knowledge of some system parameters (e.g., knowing the maximum number of supported variables for the problem). Example Problem With Many Variables demonstrates the hybrid approach to problem solving in more detail by explicitly configuring the classical and quantum workflows.

Example Requirements

To run the code in this example, the following is required.

- The requisite information for problem submission through SAPI, as described in Configuring Access to D-Wave Solvers
- Ocean tools dwave-hybrid and dwave_networkx.

If you installed dwave-ocean-sdk and ran dwave setup, your installation should meet these requirements. In D-Wave’s Leap IDE, the default workspace meets these requirements.

Solution Steps

Section How a D-Wave System Solves Problems describes the process of solving problems on the quantum computer in two steps: (1) Formulate the problem as a binary quadratic model (BQM) and (2) Solve the BQM with a D-wave system or classical sampler. In this example, a function in Ocean software handles both steps. Our task is mainly to select the sampler used to solve the problem.

Formulate the Problem

This example uses the NetworkX read_adjlist function to read a text file, usa.adj, containing the states of the USA and their adjacencies (states with a shared border) into a graph. The original map information was found here on write-only blog of Gregg Lind and looks like this:
Fig. 5: One solution found for a 300-node problem. Two circular sets, of blue or yellow nodes, are internally connected by solid green edges representing friendly relationships while red edges representing hostile relationships and dashed green edges representing frustrated relationships are stretched out between these.
Fig. 6: Coloring a map of the USA.

```python
# Author Gregg Lind
# License: Public Domain.  I would love to hear about any projects you use it for though!
# AK, HI
AL, MS, TN, GA, FL
AR, MO, TN, MS, LA, TX, OK
AZ, CA, NV, UT, CO, NM
CA, OR, NV, AZ
CO, WY, NE, KS, OK, NM, AZ, UT

# Snipped here for brevity

You can see in the first non-comment line that the state of Alaska ("AK") has Hawaii ("HI") as an adjacency and that Alabama ("AL") shares borders with four states.

```python
>>> import networkx as nx

```python
>>> G = nx.read_adjlist('usa.adj', delimiter = ',')  # doctest: +SKIP

Graph G now represents states as vertices and each state’s neighbors as shared edges.

### Solve the Problem by Sampling

Ocean’s dwave_networkx can return a minimum vertex coloring for a graph, which assigns a color to the vertices of a graph in a way that no adjacent vertices have the same color, using the minimum number of colors. Given a graph representing a map and a sampler, the min_vertex_coloring function tries to solve the map coloring problem.

dwave-hybrid Kerberos is classical-quantum hybrid asynchronous decomposition sampler, which can decompose large problems into smaller pieces that it can run both classically (on your local machine) and on the D-Wave system. Kerberos finds best samples by running in parallel tabu search, simulated annealing, and D-Wave subproblem sampling on problem variables that have high impact. The only optional parameters set here are a maximum number of iterations and number of iterations with no improvement that terminates sampling. (See the Problem With Many Variables example for more details on configuring the classical and quantum workflows.)

```python
>>> import dwave_networkx as dnx

```python
```python
>>> from hybrid.reference.kerberos import KerberosSampler

```python
```python
>>> coloring = dnx.min_vertex_coloring(G, sampler=KerberosSampler(), chromatic_ub=4, max_iter=10, convergence=3)  # doctest: +SKIP

(continues on next page)
Note: The next code requires Matplotlib.

Plot the solution, if valid.

```python
>>> import matplotlib.pyplot as plt    # doctest: +SKIP

>>> node_colors = [coloring.get(node) for node in G.nodes()]    # doctest: +SKIP

# Adjust the next line if using a different map
>>> if dnx.is_vertex_coloring(G, coloring):    # doctest: +SKIP
...     nx.draw(G, pos=nx.shell_layout(G, nlist = [list(G.nodes)[x:x+10] for x in range(0, 50, 10)] + [[list(G.nodes)[50]]]), with_labels=True, node_color=node_colors, node_size=400, cmap=plt.cm.rainbow)

>>> plt.show()    # doctest: +SKIP
```

The graphic below shows the result of one such run.

![USA map-coloring solution](image)

Fig. 7: One solution found for the USA map-coloring problem.

- *Structural Imbalance in a Social Network* solves an arbitrary-sized problem using a Leap hybrid solver.
- *Large Map Coloring* demonstrates using an out-of-the-box Ocean hybrid solver.

### 1.4.2 Beginner-Level Examples: Using the QPU

**Vertex Cover**

This example solves a few small examples of a known graph problem, *minimum vertex cover*. A vertex cover is a set of vertices such that each edge of the graph is incident with at least one vertex in the set. A minimum vertex cover is
the vertex cover of smallest size.

The purpose of this example is to help a new user to submit a problem to a D-Wave system using Ocean tools with little configuration or coding. Other examples demonstrate more advanced steps that might be needed for complex problems.

**Example Requirements**

To run the code in this example, the following is required.

- The requisite information for problem submission through SAPI, as described in *Configuring Access to D-Wave Solvers*.
- Ocean tools `dwave-system`, `dimod`, and `dwave_networkx`.

If you installed `dwave-ocean-sdk` and ran `dwave setup`, your installation should meet these requirements. In D-Wave’s Leap IDE, the default workspace meets these requirements.

**Solution Steps**

Section *How a D-Wave System Solves Problems* describes the process of solving problems on the quantum computer in two steps: (1) Formulate the problem as a binary quadratic model (BQM) and (2) Solve the BQM with a D-wave system or classical sampler. In this example, a function in Ocean software handles both steps. Our task is mainly to select the sampler used to solve the problem.

**Formulate the Problem**

The real-world application for this example might be a network provider’s routers interconnected by fiberoptic cables or traffic lights in a city’s intersections. It is posed as a graph problem; here, the five-node star graph shown below. Intuitively, the solution to this small example is obvious — the minimum set of vertices that touch all edges is node 0, but the general problem of finding such a set is NP hard.

![Fig. 8: A five-node star graph.](image)

First, we run the code snippet below to create a star graph where node 0 is hub to four other nodes. The code uses `NetworkX`, which is part of your `dwave_networkx` or `dwave-ocean-sdk` installation.
Solve the Problem by Sampling

For small numbers of variables, even your computer’s CPU can solve minimum vertex covers quickly. In this example, we demonstrate how to solve the problem both classically on your CPU and on the quantum computer.

Solving Classically on a CPU

Before using the D-Wave system, it can sometimes be helpful to test code locally. Here we select one of Ocean software’s test samplers to solve classically on a CPU. Ocean’s dimod provides a sampler that simply returns the BQM’s value for every possible assignment of variable values.

```python
>>> from dimod.reference.samplers import ExactSolver

sampler = ExactSolver()

print(dnx.min_vertex_cover(s5, sampler))
[0]
```

Solving on a D-Wave System

We now use a sampler from Ocean software’s dwave-system to solve on a D-Wave system. In addition to DWaveSampler(), we use EmbeddingComposite(), which maps unstructured problems to the graph structure of the selected sampler, a process known as minor-embedding: our problem star graph must be mapped to the QPU’s numerically indexed qubits.

Note: The code below sets a sampler without specifying SAPI parameters. Configure a default solver as described in Configuring Access to D-Wave Solvers to run the code as is, or see dwave-cloud-client to access a particular solver by setting explicit parameters in your code or environment variables.

```python
>>> from dwave.system import DWaveSampler, EmbeddingComposite

sampler = EmbeddingComposite(DWaveSampler())

print(dnx.min_vertex_cover(s5, sampler))
[0]
```

Additional Problem Graphs

The figure below shows another five-node (wheel) graph.

The code snippet below creates a new graph and solves on a D-Wave system.
Fig. 9: A five-node wheel graph.

```python
>>> w5 = nx.wheel_graph(5)
>>> print(dnx.min_vertex_cover(w5, sampler))  # doctest: +SKIP
[0, 1, 3]
```

Note that the solution found for this problem is not unique; for example, [0, 2, 4] is also a valid solution.

```python
>>> print(dnx.min_vertex_cover(w5, sampler))  # doctest: +SKIP
[0, 2, 4]
```

The figure below shows a ten-node (circular-ladder) graph.

Fig. 10: A ten-node circular-ladder graph.

The code snippet below replaces the problem graph and submits twice to the D-Wave system for solution, producing two of the possible valid solutions.
>>> c5 = nx.circular_ladder_graph(5)
>>> print(dnx.min_vertex_cover(c5, sampler)) # doctest: +SKIP
[0, 2, 3, 6, 8, 9]
>>> print(dnx.min_vertex_cover(c5, sampler)) # doctest: +SKIP
[1, 3, 4, 5, 7, 9]

Summary

In the terminology of *Ocean Software Stack*, Ocean tools moved the original problem through the following layers:

- **Application**: an example application might be placing limited numbers of traffic-monitoring equipment on routers in a telecommunication network. Such problems can be posed as graphs.

- **Method**: graph mapping. Many different real-world problems can be formulated as instances of classified graph problems. Some of these are hard and the best currently known algorithms for solution may not scale well. Quantum computing might provide better solutions. In this example, vertex cover is a hard problem that can be solved on D-Wave systems.

- **Sampler API**: the Ocean tool internally builds a BQM with lowest values (“ground states”) that correspond to a minimum vertex cover and uses our selected sampler to solve it.

- **Sampler**: classical `ExactSolver()` and then `DWaveSampler()`.

- **Compute resource**: first a local CPU then a D-Wave system.

Constrained Scheduling

This example solves a binary *constraint satisfaction problem* (CSP). CSPs require that all a problem’s variables be assigned values that result in the satisfying of all constraints. Here, the constraints are a company’s policy for scheduling meetings:

- **Constraint 1**: During business hours, all meetings must be attended in person at the office.
- **Constraint 2**: During business hours, participation in meetings is mandatory.
- **Constraint 3**: Outside business hours, meetings must be teleconferenced.
- **Constraint 4**: Outside business hours, meetings must not exceed 30 minutes.

Solving such a CSP means finding meetings that meet all the constraints.

The purpose of this example is to help a new user to formulate a constraint satisfaction problem using Ocean tools and solve it on a D-Wave system. Other examples demonstrate more advanced steps that might be needed for complex problems.

Example Requirements

To run the code in this example, the following is required.

- The requisite information for problem submission through SAPI, as described in *Configuring Access to D-Wave Solvers*.
- Ocean tools `dwave-binarycsp`, `dwave-system`, and `dimod`.

If you installed `dwave-ocean-sdk` and ran `dwave setup`, your installation should meet these requirements. In D-Wave’s Leap IDE, the default workspace meets these requirements.
Solution Steps

Section *How a D-Wave System Solves Problems* describes the process of solving problems on the quantum computer in two steps: (1) Formulate the problem as a *binary quadratic model* (BQM) and (2) Solve the BQM with a D-wave system or classical *sampler*. In this example, Ocean’s *dwavebinarycsp* tool builds the BQM based on the constraints we formulate.

Formulate the Problem

D-Wave systems solve binary quadratic models, so the first step is to express the problem with binary variables.

- Time of day is represented by binary variable `time` with value 1 for business hours and 0 for hours outside the business day.
- Venue is represented by binary variable `location` with value 1 for office and 0 for teleconference.
- Meeting duration is represented by variable `length` with value 1 for short meetings (under 30 minutes) and 0 for meetings of longer duration.
- Participation is represented by variable `mandatory` with value 1 for mandatory participation and 0 for optional participation.

For large numbers of variables and constraints, such problems can be hard. This example has four binary variables, so only $2^4 = 16$ possible meeting arrangements. As shown in the table below, it is a simple matter to work out all the combinations by hand to find solutions that meet all the constraints.

<table>
<thead>
<tr>
<th>Time of Day</th>
<th>Venue</th>
<th>Duration</th>
<th>Participation</th>
<th>Valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business hours</td>
<td>Office</td>
<td>Short</td>
<td>Mandatory</td>
<td>Yes</td>
</tr>
<tr>
<td>Business hours</td>
<td>Office</td>
<td>Short</td>
<td>Optional</td>
<td>No (violates 2)</td>
</tr>
<tr>
<td>Business hours</td>
<td>Office</td>
<td>Long</td>
<td>Mandatory</td>
<td>Yes</td>
</tr>
<tr>
<td>Business hours</td>
<td>Office</td>
<td>Long</td>
<td>Optional</td>
<td>No (violates 2)</td>
</tr>
<tr>
<td>Business hours</td>
<td>Teleconference</td>
<td>Short</td>
<td>Mandatory</td>
<td>No (violates 1)</td>
</tr>
<tr>
<td>Business hours</td>
<td>Teleconference</td>
<td>Short</td>
<td>Optional</td>
<td>No (violates 1, 2)</td>
</tr>
<tr>
<td>Business hours</td>
<td>Teleconference</td>
<td>Long</td>
<td>Mandatory</td>
<td>No (violates 1)</td>
</tr>
<tr>
<td>Business hours</td>
<td>Teleconference</td>
<td>Long</td>
<td>Optional</td>
<td>No (violates 1, 2)</td>
</tr>
<tr>
<td>Non-business hours</td>
<td>Office</td>
<td>Short</td>
<td>Mandatory</td>
<td>No (violates 3)</td>
</tr>
<tr>
<td>Non-business hours</td>
<td>Office</td>
<td>Short</td>
<td>Optional</td>
<td>No (violates 3)</td>
</tr>
<tr>
<td>Non-business hours</td>
<td>Office</td>
<td>Long</td>
<td>Mandatory</td>
<td>No (violates 3, 4)</td>
</tr>
<tr>
<td>Non-business hours</td>
<td>Office</td>
<td>Long</td>
<td>Optional</td>
<td>No (violates 3, 4)</td>
</tr>
<tr>
<td>Non-business hours</td>
<td>Teleconference</td>
<td>Short</td>
<td>Mandatory</td>
<td>Yes</td>
</tr>
<tr>
<td>Non-business hours</td>
<td>Teleconference</td>
<td>Short</td>
<td>Optional</td>
<td>Yes</td>
</tr>
<tr>
<td>Non-business hours</td>
<td>Teleconference</td>
<td>Long</td>
<td>Mandatory</td>
<td>No (violates 4)</td>
</tr>
<tr>
<td>Non-business hours</td>
<td>Teleconference</td>
<td>Long</td>
<td>Optional</td>
<td>No (violates 4)</td>
</tr>
</tbody>
</table>

Ocean’s *dwavebinarycsp* enables the definition of constraints in different ways, including by defining functions that evaluate True when the constraint is met. The code below defines a function that returns True when all this example’s constraints are met.

```python
def scheduling(time, location, length, mandatory):
    if time:
        return (location and mandatory)  # In office and mandatory participation
    else:
        return ((not location) and length)  # Teleconference for a short duration
```

1.4. Examples
The next code lines create a constraint from this function and adds it to CSP instance, `csp`, instantiated with binary variables.

```python
>>> import dwavebinarycsp
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(scheduling, ['time', 'location', 'length', 'mandatory'])
```

This tool, `dwavebinarycsp`, can also convert the binary CSP to a BQM. The following code does so and the graph below provides a view on the BQM’s linear and quadratic coefficients, $q_i$ and $q_{ij}$ respectively in $\sum_i q_i x_i + \sum_{i<j} q_{ij} x_i x_j$, which are the inputs for programming the quantum computer.

```python
>>> bqm = dwavebinarycsp.stitch(csp)
```

Fig. 11: A heatmap of the BQM, with darker colors for higher linear (node color) and quadratic (edge color) values. You can see the values simply by using the `print(bqm)` command.

### Solve the Problem by Sampling

For small numbers of variables, even your computer’s CPU can solve CSPs quickly. Here we solve both classically on your CPU and on the quantum computer.

#### Solving Classically on a CPU

Before using the D-Wave system, it can sometimes be helpful to test code locally. Here we select one of Ocean software’s test samplers to solve classically on a CPU. Ocean’s `dimod` provides a sampler that simply returns the BQM’s value (energy) for every possible assignment of variable values.

```python
>>> from dimod.reference.samplers import ExactSolver
>>> sampler = ExactSolver()
>>> solution = sampler.sample(bqm)
```

Valid solutions—assignments of variables that do not violate any constraint—should have the lowest value of the BQM:
Fig. 12: Energy of all 16 possible configurations. You can see the values simply by using the `print(solution)` command.
The code below prints all those solutions (assignments of variables) for which the BQM has its minimum value\(^1\).

```python
>>> from math import isclose
>>> min_energy = solution.record.energy.min()
>>> for sample, energy in solution.data(['sample', 'energy']):  # doctest: +SKIP
...     if isclose(energy, min_energy, abs_tol=1.0):
...         time = 'business hours' if sample['time'] else 'evenings'
...         location = 'office' if sample['location'] else 'home'
...         length = 'short' if sample['length'] else 'long'
...         mandatory = 'mandatory' if sample['mandatory'] else 'optional'
...         print("During {} at {}, you can schedule a {} meeting that is {}".format(time, location, length, mandatory))
```

During evenings at home, you can schedule a short meeting that is optional
During evenings at home, you can schedule a short meeting that is mandatory
During business hours at office, you can schedule a short meeting that is mandatory
During business hours at office, you can schedule a long meeting that is mandatory

**Solving on a D-Wave System**

We now solve on a D-Wave system using sampler `DWaveSampler()` from Ocean software’s `dwave-system`. We also use its `EmbeddingComposite()` composite to map our unstructured problem (variables such as `time` etc.) to the sampler’s graph structure (the QPU’s numerically indexed qubits) in a process known as minor-embedding. The next code sets up a D-Wave system as the sampler.

**Note:** The code below sets a sampler without specifying `SAPI` parameters. Configure a default `solver` as described in Configuring Access to D-Wave Solvers to run the code as is, or see `dwave-cloud-client` to access a particular solver by setting explicit parameters in your code or environment variables.

```python
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler())
```

Because the sampled solution is probabilistic, returned solutions may differ between runs. Typically, when submitting a problem to the system, we ask for many samples, not just one. This way, we see multiple “best” answers and reduce the probability of settling on a suboptimal answer. Below, we ask for 5000 samples.

```python
>>> sampleset = sampler.sample(bqm, num_reads=5000)
```

The code below prints all those solutions (assignments of variables) for which the BQM has its minimum value and the number of times it was found.

```python
>>> total = 0
... for sample, energy, occurrences in sampleset.data(['sample', 'energy', 'num_occurrences']):  # doctest: +SKIP
...     total = total + occurrences
...     if isclose(energy, min_energy, abs_tol=1.0):
...         time = 'business hours' if sample['time'] else 'evenings'
...         location = 'office' if sample['location'] else 'home'
...         length = 'short' if sample['length'] else 'long'
...         mandatory = 'mandatory' if sample['mandatory'] else 'optional'
...
(continues on next page)
```

---

\(^1\) Because it compares float values, this code uses the standard `isclose` function to find values that are approximately equal. A small tolerance is needed to overcome rounding errors but for simplicity a value of `abs_tol=1.0` is used because by default the `stich()` function increases the energy of solutions that violate one constraint by `min_classical_gap=2.0`. 

---
Summary

In the terminology of *Ocean Software Stack*, Ocean tools moved the original problem through the following layers:

- **Application**: scheduling under constraints. There exist many CSPs that are computationally hard problems; for example, the map-coloring problem is to color all regions of a map such that any two regions sharing a border have different colors. The job-shop scheduling problem is to schedule multiple jobs done on several machines with constraints on the machines’ execution of tasks.

- **Method**: constraint compilation.

- **Sampler API**: the Ocean tool builds a BQM with lowest values ("ground states") that correspond to assignments of variables that satisfy all constraints.

- **Sampler**: classical `ExactSolver()` and then `DWaveSampler()`.

- **Compute resource**: first a local CPU then a D-Wave system.

**Boolean NOT Gate**

This example solves a simple problem of a Boolean NOT gate to demonstrate the mathematical formulation of a problem as a *binary quadratic model* (BQM) and using Ocean tools to solve such problems on a D-Wave system. Other examples demonstrate the more advanced steps that are typically needed for solving actual problems.

**Example Requirements**

To run the code in this example, the following is required.

- The requisite information for problem submission through SAPI, as described in *Configuring Access to D-Wave Solvers*

- Ocean tools `dwave-system` and `dimod`.

If you installed `dwave-ocean-sdk` and ran `dwave setup`, your installation should meet these requirements. In D-Wave’s Leap IDE, the default workspace meets these requirements.

**Solution Steps**

Section *How a D-Wave System Solves Problems* describes the process of solving problems on the quantum computer in two steps: (1) Formulate the problem as a *binary quadratic model* (BQM) and (2) Solve the BQM with a D-wave system or classical sampler. In this example, we mathematically formulate the BQM and use Ocean tools to solve it on a D-Wave system.
Formulate the NOT Gate as a BQM

We use a sampler like the D-Wave systems to solve binary quadratic models (BQM)\(^1\): given \(M\) variables \(x_1, \ldots, x_N\), where each variable \(x_i\) can have binary values 0 or 1, the system tries to find assignments of values that minimize

\[
\sum_{i=1}^{N} q_i x_i + \sum_{i < j}^{N} q_{i,j} x_i x_j,
\]

where \(q_i\) and \(q_{i,j}\) are configurable (linear and quadratic) coefficients. To formulate a problem for the D-Wave system is to program \(q_i\) and \(q_{i,j}\) so that assignments of \(x_1, \ldots, x_N\) also represent solutions to the problem.

Ocean tools can automate the representation of logic gates as a BQM, as demonstrated in the Multiple-Gate Circuit example.

A NOT gate is shown in Figure 13.

```
Fig. 13: NOT gate
```

Representing the Problem With a Penalty Function

This example demonstrates a mathematical formulation of the BQM. We can represent a NOT gate, \(z \leftrightarrow \neg x\), where \(x\) is the gate’s input and \(z\) its output, using a penalty function:

\[
2xz - x - z + 1.
\]

This penalty function represents the NOT gate in that for assignments of variables that match valid states of the gate, the function evaluates at a lower value than assignments that would be invalid for the gate. Therefore, when the D-Wave minimizes a BQM based on this penalty function, it finds those assignments of variables that match valid gate states.

The table below shows that this function penalizes states that are not valid for the gate while no penalty is applied to assignments of variables that correctly represent a NOT gate. In this table, column \(x\) is all possible states of the gate’s input; column \(z\) is the corresponding output values; column \textbf{Valid?} shows whether the variables represent a valid state for a NOT gate; column \(P\) shows the value of the penalty for all possible assignments of variables.

```
<table>
<thead>
<tr>
<th>(x)</th>
<th>(z)</th>
<th>Valid?</th>
<th>(P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>No</td>
<td>1</td>
</tr>
</tbody>
</table>
```

\(^1\) The “native” forms of BQM programmed into a D-Wave system are the Ising model traditionally used in statistical mechanics and its computer-science equivalent, shown here, the QUBO.
For example, the state $x, z = 0, 1$ of the first row represents valid assignments, and the value of $P$ is

$$2xz - x - z + 1 = 2 \times 0 \times 1 - 0 - 1 + 1 = -1 + 1 = 0,$$

not penalizing the valid assignment of variables. In contrast, the state $x, z = 0, 0$ of the third row represents an invalid assignment, and the value of $P$ is

$$2xz - x - z + 1 = 2 \times 0 \times 0 - 0 - 0 + 1 = 1,$$

adding a value of 1 to the BQM being minimized. By penalizing both possible assignments of variables that represent invalid states of a NOT gate, the BQM based on this penalty function has minimal values (lowest energy states) for variable values that also represent a NOT gate.

See the system documentation for more information about penalty functions in general, and penalty functions for representing Boolean operations.

**Formulating the Problem as a QUBO**

Sometimes penalty functions are of cubic or higher degree and must be reformulated as quadratic to be mapped to a binary quadratic model. For this penalty function we just need to drop the freestanding constant: the function’s values are simply shifted by $-1$ but still those representing valid states of the NOT gate are lower than those representing invalid states. The remaining terms of the penalty function,

$$2xz - x - z,$$

are easily reordered in standard *QUBO* formulation:

$$-x_1 - x_2 + 2x_1x_2$$

where $z = x_2$ is the NOT gate’s output, $x = x_1$ the input, linear coefficients are $q_1 = q_2 = -1$, and quadratic coefficient is $q_{1,2} = 2$. These are the coefficients used to program a D-Wave system.

Often it is convenient to format the coefficients as an upper-triangular matrix:

$$Q = \begin{bmatrix} -1 & 2 \\ 0 & -1 \end{bmatrix}$$

See the system documentation for more information about formulating problems as QUBOs.

**Solve the Problem by Sampling**

We now solve on a D-Wave system using sampler *DWaveSampler()* from Ocean software’s *dwave-system*. We also use its *EmbeddingComposite()* composite to map our unstructured problem (variables such as time etc.) to the sampler’s graph structure (the QPU’s numerically indexed qubits) in a process known as *minor-embedding*.

The next code sets up a D-Wave system as the sampler.

**Note:** The code below sets a sampler without specifying *SAPI* parameters. Configure a default *solver* as described in *Configuring Access to D-Wave Solvers* to run the code as is, or see *dwave-cloud-client* to access a particular solver by setting explicit parameters in your code or environment variables.

```python
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler())
```
Because the sampled solution is probabilistic, returned solutions may differ between runs. Typically, when submitting a problem to the system, we ask for many samples, not just one. This way, we see multiple “best” answers and reduce the probability of settling on a suboptimal answer. Below, we ask for 5000 samples.

```python
>>> Q = {('x', 'x'): -1, ('x', 'z'): 2, ('z', 'x'): 0, ('z', 'z'): -1}
>>> sampleset = sampler.sample_qubo(Q, num_reads=5000)
>>> print(sampleset)
# doctest: +SKIP
x  z  energy  num_oc.  chain_.
0  0  1  -1.0  2266  0.0
1  1  0  -1.0  2732  0.0
2  0  0  0.0  1  0.0
3  1  1  0.0  1  0.0
['BINARY', 4 rows, 5000 samples, 2 variables]
```

Almost all the returned samples represent valid value assignments for a NOT gate, and minima (low-energy states) of the BQM, and with high likelihood the best (lowest-energy) samples satisfy the NOT gate formulation:

```python
>>> sampleset.first.sample['x'] != sampleset.first.sample['z']
True
```

**Summary**

In the terminology of *Ocean Software Stack*, Ocean tools moved the original problem through the following layers:

- The sampler API is a QUBO formulation of the problem.
- The sampler is `DWaveSampler()`.
- The compute resource is a D-Wave system.

**Boolean AND Gate**

This example solves a simple problem of a Boolean AND gate on a D-Wave system to demonstrate programming the underlying hardware more directly; in particular, *minor-embedding a chain*.

Other examples demonstrate more advanced steps that are typically needed for solving actual problems.

**Example Requirements**

To run the code in this example, the following is required.

- The requisite information for problem submission through SAPI, as described in *Configuring Access to D-Wave Solvers*.

If you installed `dwave-ocean-sdk` and ran `dwave setup`, your installation should meet these requirements. In D-Wave’s Leap IDE, the default workspace meets these requirements.

**Solution Steps**

Section *How a D-Wave System Solves Problems* describes the process of solving problems on the quantum computer in two steps: (1) Formulate the problem as a binary quadratic model (BQM) and (2) Solve the BQM with a D-wave system or classical `sampler`. In this example, we mathematically formulate the BQM and use Ocean tools to solve it on a D-Wave system.
Formulate the AND Gate as a BQM

Ocean tools can automate the representation of logic gates as a BQM, as demonstrated in the Multiple-Gate Circuit example. The Boolean NOT Gate example presents a mathematical formulation of a BQM for a Boolean gate in detail. Here we briefly repeat the steps of mathematically formulating a BQM while adding details on the underlying physical processes.

A D-Wave quantum processing unit (QPU) is a chip with interconnected qubits; for example, a D-Wave 2000Q has up to 2048 qubits connected in a Chimera topology. Programming it consists mostly of setting two inputs:

- Qubit bias weights: control the degree to which a qubit tends to a particular state.
- Qubit coupling strengths: control the degree to which two qubits tend to the same state.

The biases and couplings define an energy landscape, and the D-Wave quantum computer seeks the minimum energy of that landscape. Once you express your problem in a formulation such that desired outcomes have low energy values and undesired outcomes high energy values, the D-Wave system solves your problem by finding the low-energy states.

Here we use another binary quadratic model (BQM), the computer-science equivalent of the Ising model, the QUBO: given $M$ variables $x_1, \ldots, x_N$, where each variable $x_i$ can have binary values 0 or 1, the system tries to find assignments of values that minimize

$$E(q_i, q_{i,j}; x_i) = \sum_{i} q_i x_i + \sum_{i<j} q_{i,j} x_i x_j,$$

where $q_i$ and $q_{i,j}$ are configurable (linear and quadratic) coefficients. To formulate a problem for the D-Wave system is to program $q_i$ and $q_{i,j}$ so that assignments of $x_1, \ldots, x_N$ also represent solutions to the problem.

AND as a Penalty Function

This example represents the AND operation, $z \iff x_1 \land x_2$, where $x_1, x_2$ are the gate’s inputs and $z$ its output, using a penalty function:

$$x_1 x_2 - 2(x_1 + x_2)z + 3z.$$

This penalty function represents the AND gate in that for assignments of variables that match valid states of the gate, the function evaluates at a lower value than assignments that would be invalid for the gate. Therefore, when the D-Wave system minimizes a BQM based on this penalty function, it finds those assignments of variables that match valid gate states.

You can verify that this penalty function represents the AND gate in the same way as was done in the Boolean NOT Gate example. See the D-Wave Problem-Solving Handbook for more information about penalty functions in general, and penalty functions for representing Boolean operations in particular.

Formulating the Problem as a QUBO

For this example, the penalty function is quadratic, and easily reordered in the familiar QUBO formulation:

$$E(q_i, q_{i,j}; x_i) = 3x_3 + x_1 x_2 - 2x_1 x_3 - 2x_2 x_3$$

---

1 This formulation, called an objective function, corresponds to the Ising model traditionally used in statistical mechanics: given $N$ variables $s_1, \ldots, s_N$, corresponding to physical Ising spins, where each variable $s_i$ can have values $-1$ or $+1$, the system energy for an assignment of values is,

$$E(s|h, J) = \left\{ \sum_{i=1}^{N} h_i s_i + \sum_{i<j} J_{i,j} s_i s_j \right\}$$

where $h_i$ are biases and $J_{i,j}$ couplings between spins.
where \( z = x_3 \) is the AND gate’s output, \( x_1, x_2 \) the inputs, linear coefficients are \( q_1 = 3 \), and quadratic coefficients are \( q_{1,2} = 1, q_{1,3} = -2, q_{2,3} = -2 \). The coefficients matrix is,

\[
Q = \begin{bmatrix}
0 & 1 & -2 \\
0 & -2 & 3
\end{bmatrix}
\]

See the Getting Started with the D-Wave System and D-Wave Problem-Solving Handbook books for more information about formulating problems as QUBOs.

The line of code below sets the QUBO coefficients for this AND gate.

```python
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'): 3}
```

**Solve the Problem by Sampling: Automated Minor-Embedding**

For reference, we first solve with the same steps used in the Boolean NOT Gate example before solving again while manually controlling additional parameters.

Again we use sampler `DWaveSampler()` from Ocean software’s `dwave-system` and its `EmbeddingComposite()` composite to minor-embed our unstructured problem (variables \( x_1, x_2, \) and \( z \)) on the sampler’s graph structure (the QPU’s numerically indexed qubits).

The next code sets up a D-Wave system as the sampler.

---

**Note:** The code below sets a sampler without specifying SAPI parameters. Configure a default solver as described in Configuring Access to D-Wave Solvers to run the code as is, or see `dwave-cloud-client` to access a particular solver by setting explicit parameters in your code or environment variables.

```python
>>> from dwave.system import DWaveSampler, EmbeddingComposite

>>> sampler = DWaveSampler()

>>> sampler_embedded = EmbeddingComposite(sampler)
```

As before, we ask for 5000 samples.

```python
>>> sampleset = sampler_embedded.sample_qubo(Q, num_reads=5000)

>>> print(sampleset)  # doctest: +SKIP
x1 x2 z energy num_oc. chain_b.
0 0 1 0 0.0 1812 0.0
1 1 0 0 0.0 645 0.0
2 1 1 1 0.0 862 0.0
3 0 0 0 0.0 1676 0.0
5 1 0 0 0.0 1 0.333333
6 1 1 0 0.0 1 0.333333
4 0 1 1 1.0 3 0.0
['BINARY', 7 rows, 5000 samples, 3 variables]
```

All the returned samples from this execution represent valid value assignments for an AND gate, and minimize (are low-energy states of) the BQM.

Note that lines 5 and 6 of output from this execution show samples that seem identical to lines 1 and 2 (but with non-zero values in the rightmost column, chain_breaks). The next section addresses that.
Solve the Problem by Sampling: Non-automated Minor-Embedding

This section looks more closely into minor-embedding. Above and in the **Boolean NOT Gate** example, `dwave-system EmbeddingComposite()` composite abstracted the minor-embedding.

**Minor-Embedding a NOT Gate**

For simplicity, we first return to the NOT gate. The **Boolean NOT Gate** example found that a NOT gate can be represented by a BQM in QUBO form with the following coefficients:

```python
>>> Q_not = {('x', 'x'): -1, ('x', 'z'): 2, ('z', 'x'): 0, ('z', 'z'): -1}
```

Minor embedding maps the two problem variables x and z to the indexed qubits of the D-Wave QPU. Here we do this mapping ourselves.

The next line of code looks at properties of the sampler. We select the first node, which on a QPU is a qubit, and print its adjacent nodes, i.e., coupled qubits.

```python
>>> print(sampler.adjacency[sampler.nodelist[0]])
# doctest: +SKIP
{128, 4, 5, 6, 7}
```

For the D-Wave system the above code ran on, we see that the first available qubit is adjacent to qubit 4 and four others.

We can map the NOT problem’s two linear coefficients and single quadratic coefficient, \( q_1 = q_2 = -1 \) and \( q_{1,2} = 2 \), to biases on qubits 0 and 4 and coupling (0, 4). The figure below shows a minor embedding of the NOT gate into a D-Wave 2000Q QPU unit cell (four horizontal qubits connected to four vertical qubits via couplers).

![Fig. 15: A NOT gate minor embedded into the topmost left unit cell of a D-Wave 2000Q QPU. Variables \( x_1, x_2 \) are minor embedded as qubits 0 and 4 (blue circles). Biases \( q_1, q_2 = -1, -1 \) and coupling strength \( q_{1,2} = 2 \) are also shown.](image)

The following code uses the `FixedEmbeddingComposite` composite to manually minor-embed the problem. Its last line prints a confirmation that indeed the two selected qubits are adjacent (coupled).

```python
>>> from dwave.system import FixedEmbeddingComposite
>>> sampler_embedded = FixedEmbeddingComposite(sampler, {'x': [0], 'z': [4]})
>>> print(sampler_embedded.adjacency['x'])
{'z'}
```
As before, we ask for 5000 samples.

```python
>>> sampleset = sampler_embedded.sample_qubo(Q_not, num_reads=5000)
>>> print(sampleset)  # doctest: +SKIP
    x  z  energy  num_oc.  chain_.
  0  0  1 -1.0  2310  0.0
  1  1  0 -1.0  2688  0.0
  2  0  0  0.0   2  0.0
['BINARY', 3 rows, 5000 samples, 2 variables]
```

**From NOT to AND: an Important Difference**

- The BQM for a NOT gate, \(-x - z + 2xz\), can be represented by a fully connected \(K_2\) graph: its linear coefficients are weights of the two connected nodes with the single quadratic coefficient the weight of its connecting edge.
- The BQM for an AND gate, \(3z + x_1x_2 - 2x_1z - 2x_2z\), needs a \(K_3\) graph.

![Fig. 16: NOT gate \(K_2\) complete graph (top) versus AND gate \(K_3\) complete graph (bottom.)](image)

We saw above how to minor-embed a \(K_2\) graph on a D-Wave system. To minor-embed a fully connected \(K_3\) graph requires *chaining* qubits.

**Minor-Embedding an AND Gate**

To understand how a \(K_3\) graph fits on the *Chimera* topology of the QPU, look at the Chimera unit cell structure shown below. You cannot connect 3 qubits in a closed loop. However, you can make a closed loop of 4 qubits using, say, qubits 0, 1, 4, and 5.

To fit the 3-qubit loop into a 4-sided structure, create a chain of 2 qubits to represent a single variable. For example, chain qubit 0 and qubit 4 to represent variable \(z\).
The strength of the coupler between qubits 0 and 4, which represents variable $z$, must be set to correlate the qubits strongly, so that in most solutions they have a single value for $z$. (Remember the output in the Solve the Problem by Sampling: Automated Minor-Embedding section with its identical two last lines? This was likely due to the qubits in a chain taking different values.)

The code below uses Ocean’s `dwave-system` `FixedEmbeddingComposite()` composite for manual minor-embedding. Its last line prints a confirmation that indeed all three variables are connected. (coupled).

```python
>>> embedding = {'x1': {1}, 'x2': {5}, 'z': {0, 4}}
>>> sampler_embedded = FixedEmbeddingComposite(sampler, embedding)
>>> print(sampler_embedded.adjacency)  # doctest: +SKIP
{'x1': {'x2', 'z'}, 'x2': {'x1', 'z'}, 'z': {'x1', 'x2'}}
```

We ask for 5000 samples.

```python
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'): 3}
>>> sampleset = sampler_embedded.sample_qubo(Q, num_reads=5000)
>>> print(sampleset)  # doctest: +SKIP
['BINARY', 6 rows, 5000 samples, 3 variables]
```

Optionally, you can use the `problem-inspector` to view the solution on the QPU.
Note: The next code requires the use of Ocean’s problem inspector.

```python
>>> import dwave.inspector
>>> dwave.inspector.show(sampleset)  # doctest: +SKIP
```

![Fig. 19: View of the logical and embedded problem rendered by Ocean’s problem inspector. The AND gate’s original QUBO is represented on the left; its embedded representation, on the right, shows a two-qubit chain of qubits 0 and 4 for variable Z. The current solution displayed, \(X_1 = 1, X_2 = 0, Z = 0\), is represented by white and gold dots for binary 0, 1 and white and blue dots for spin values \(-1, 1\).]

For comparison, the following code purposely weakens the chain strength (strength of the coupler between qubits 0 and 4, which represents variable \(z\)). The first line prints the range of values available for the D-Wave system this code is executed on. By default, `FixedEmbeddingComposite()` used the maximum chain strength, which is 2. By setting it to a low value of 0.25, the two qubits are not strongly correlated and the result is that many returned samples represent invalid states for an AND gate.

```python
>>> print(sampler.properties['extended_j_range'])
[-2.0, 1.0]
>>> sampler_embedded = FixedEmbeddingComposite(sampler, embedding)
>>> sampleset = sampler_embedded.sample_qubo(Q, num_reads=5000, chain_strength=0.25)
>>> print(sampleset)  # doctest: +SKIP
  x1 x2 z energy num_oc. chain_b.
0 1 0 0 0.0 629 0.0
1 0 1 0 0.0 693 0.0
3 1 1 1 0.0 660 0.0
4 0 0 0 0.0 812 0.0
2 1 0 1 1.0 773 0.333333
5 0 1 1 1.0 1432 0.333333
6 0 1 1 1.0 1 0.0
['BINARY', 7 rows, 5000 samples, 3 variables]
```

In this case, you are likely to see broken chains (non-zero values in the `chain_breaks` column) and calling the problem inspector shows these:

- **Vertex Cover** solves a small graph problem.
• *Constrained Scheduling* solves a small constraint satisfaction problem.

• *Boolean NOT Gate* mathematically formulates a BQM for a two-variable problem.

• *Boolean AND Gate* demonstrates programming the QPU more directly (*minor-embedding*).

### 1.4.3 Intermediate-Level Examples

#### Map Coloring

This example solves a map-coloring problem to demonstrate using Ocean tools to solve a problem on a D-Wave system. It demonstrates using the D-Wave system to solve a more complex constraint satisfaction problem (CSP) than that solved in the example of *Constrained Scheduling*.

Constraint satisfaction problems require that all a problem’s variables be assigned values, out of a finite domain, that result in the satisfying of all constraints. The map-coloring CSP, for example, is to assign a color to each region of a map such that any two regions sharing a border have different colors.

The constraints for the map-coloring problem can be expressed as follows:

• Each region is assigned one color only, of \( C \) possible colors.

• The color assigned to one region cannot be assigned to adjacent regions.

#### Example Requirements

To run the code in this example, the following is required.

• The requisite information for problem submission through SAPI, as described in *Configuring Access to D-Wave Solvers*.

• Ocean tools *dwavebinarycsp* and *dwave-system*. For graphics, you will also need *NetworkX*. 
If you installed `dwave-ocean-sdk` and ran `dwave setup`, your installation should meet these requirements. In D-Wave’s Leap IDE, the default workspace meets these requirements.

**Solution Steps**

Following the standard solution process described in Section *How a D-Wave System Solves Problems*, we (1) formulate the problem as a *binary quadratic model* (BQM) by using unary encoding to represent the $C$ colors: each region is represented by $C$ variables, one for each possible color, which is set to value 1 if selected, while the remaining $C - 1$ variables are 0. (2) Solve the BQM with a D-Wave system as the sampler.

The full workflow is as follows:

1. Formulate the problem as a graph, with provinces represented as nodes and shared borders as edges, using 4 binary variables (one per color) for each province.
2. Create a binary constraint satisfaction problem and add all the needed constraints.
3. Convert to a binary quadratic model.
4. Sample.
5. Plot a valid solution.

**Four-Color Canadian Map**

This example finds a solution to the map-coloring problem for a map of Canada using four colors (the sample code can easily be modified to change the number of colors or use different maps). Canada’s 13 provinces are denoted by postal codes:
<table>
<thead>
<tr>
<th>Code</th>
<th>Province</th>
<th>Code</th>
<th>Province</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>Alberta</td>
<td>BC</td>
<td>British Columbia</td>
</tr>
<tr>
<td>MB</td>
<td>Manitoba</td>
<td>NB</td>
<td>New Brunswick</td>
</tr>
<tr>
<td>NL</td>
<td>Newfoundland and Labrador</td>
<td>NS</td>
<td>Nova Scotia</td>
</tr>
<tr>
<td>NT</td>
<td>Northwest Territories</td>
<td>NU</td>
<td>Nunavut</td>
</tr>
<tr>
<td>ON</td>
<td>Ontario</td>
<td>PE</td>
<td>Prince Edward Island</td>
</tr>
<tr>
<td>QC</td>
<td>Quebec</td>
<td>SK</td>
<td>Saskatchewan</td>
</tr>
<tr>
<td>YT</td>
<td>Yukon</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Map Coloring: Full Code**

See *Map Coloring* for a description of the following code.

```python
import dwavebinarycsp
from dwave.system import DWaveSampler, EmbeddingComposite
import networkx as nx
import matplotlib.pyplot as plt

# Represent the map as the nodes and edges of a graph
provinces = ['AB', 'BC', 'MB', 'NB', 'NL', 'NS', 'NT', 'NU', 'ON', 'PE', 'QC', 'SK', 'YT']
neighbors = [('AB', 'BC'), ('AB', 'NT'), ('AB', 'SK'), ('BC', 'NT'), ('BC', 'YT'), ('MB', 'NU'), ('MB', 'ON'), ('MB', 'SK'), ('NB', 'NS'), ('NB', 'QC'), ('NL', 'QC'), ('NT', 'NU'), ('NT', 'SK'), ('NT', 'YT'), ('ON', 'QC')]

def not_both_1(v, u):
    return not (v and u)

def plot_map(sample):
    G = nx.Graph()
    G.add_nodes_from(provinces)
    G.add_edges_from(neighbors)
    # Translate from binary to integer color representation
    color_map = {}
    for province in provinces:
        for i in range(colors):
            if sample[province+str(i)]:
                color_map[province] = i
    # Plot the sample with color-coded nodes
    node_colors = [color_map.get(node) for node in G.nodes()]
    nx.draw_circular(G, with_labels=True, node_color=node_colors, node_size=3000, cmap=plt.cm.rainbow)
    plt.show()

# Valid configurations for the constraint that each node select a single color
one_color_configurations = {(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)}
colors = len(one_color_configurations)
```

(continues on next page)
# Create a binary constraint satisfaction problem
csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)

# Add constraint that each node (province) select a single color
for province in provinces:
    variables = [province+str(i) for i in range(colors)]
    csp.add_constraint(one_color_configurations, variables)

# Add constraint that each pair of nodes with a shared edge not both select one color
for neighbor in neighbors:
    v, u = neighbor
    for i in range(colors):
        variables = [v+str(i), u+str(i)]
        csp.add_constraint(not_both_1, variables)

# Convert the binary constraint satisfaction problem to a binary quadratic model
bqm = dwavebinarycsp.stitch(csp)

# Set up a solver using the local system’s default D-Wave Cloud Client configuration
sampler = EmbeddingComposite(DWaveSampler())

# and sample 1000 times
sampleset = sampler.sample(bqm, num_reads=1000)

# Plot the lowest-energy sample if it meets the constraints
sample = sampleset.first.sample
if not csp.check(sample):
    print("Failed to color map")
else:
    plot_map(sample)

Note: You can skip directly to the complete code for the problem here: Map Coloring: Full Code.

The example uses the D-Wave binary CSP tool to set up constraints and convert the CSP to a binary quadratic model, dwave-system to set up a D-Wave system as the sampler, and NetworkX to plot results.

Start by formulating the problem as a graph of the map with provinces as nodes and shared borders between provinces as edges (e.g., ‘(‘AB’, ‘BC’)’ is an edge representing the shared border between British Columbia and Alberta).

Create a binary constraint satisfaction problem based on two types of constraints, where csp is the dwavebinarycsp CSP object:

* csp.add_constraint(one_color_configurations, variables) represents the con-
constraint that each node (province) select a single color, as represented by valid configurations:

\[
\text{one\_color\_configurations} = \{(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)\}
\]

- \text{csp.add\_constraint(not\_both\_1, variables)} represents the constraint that two nodes (provinces) with a shared edge (border) not both select the same color.

```python
>>> # Function for the constraint that two nodes with a shared edge not both select
>>> # one color
>>> def not_both_1(v, u):
...     return not (v and u)
...  # Valid configurations for the constraint that each node select a single color
>>> one_color_configurations = {(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)}
>>> colors = len(one_color_configurations)
...  # Create a binary constraint satisfaction problem
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
...  # Add constraint that each node (province) select a single color
>>> for province in provinces:
...     variables = [province+str(i) for i in range(colors)]
...     csp.add\_constraint(one\_color\_configurations, variables)
...  # Add constraint that each pair of nodes with a shared edge not both select one color
>>> for neighbor in neighbors:
...     v, u = neighbor
...     for i in range(colors):
...         variables = [v+str(i), u+str(i)]
...         csp.add\_constraint(not\_both\_1, variables)
```

Convert the CSP into a binary quadratic model so it can be solved on the D-Wave system.

```python
>>> bqm = dwavebinarycsp.stitch(csp)
```

The next code sets up a D-Wave system as the sampler and requests 1000 samples.

\[\text{Note: The code below sets a sampler without specifying SAPI parameters. Configure a default solver as described in Configuring Access to D-Wave Solvers to run the code as is, or see dwave-cloud-client to access a particular solver by setting explicit parameters in your code or environment variables.}\]

```python
>>> sampler = EmbeddingComposite(DWaveSampler())
# doctest: +SKIP
>>> sampleset = sampler.sample(bqm, num_reads=1000)
# doctest: +SKIP
...  # Check that a good solution was found
>>> sample = sampleset.first.sample
# doctest: +SKIP
>>> if not csp.check(sample):
...     print("Failed to color map. Try sampling again.")
... else:
...     print(sample)
```

\[\text{Note: The next code requires Matplotlib.}\]
Plot a valid solution.

```python
# Function that plots a returned sample
def plot_map(sample):
    G = nx.Graph()
    G.add_nodes_from(provinces)
    G.add_edges_from(neighbors)
    # Translate from binary to integer color representation
    color_map = {}
    for province in provinces:
        for i in range(colors):
            if sample[province+str(i)]:
                color_map[province] = i
    # Plot the sample with color-coded nodes
    node_colors = [color_map.get(node) for node in G.nodes()]
    nx.draw_circular(G, with_labels=True, node_color=node_colors, node_size=3000,
                     cmap=plt.cm.rainbow)
    plt.show()
```

```python
>>> plot_map(sample)   # doctest: +SKIP
```

The plot shows a solution returned by the D-Wave solver. No provinces sharing a border have the same color.

![Map Coloring: Full Code](Map_Coloring_Full_Code)

Fig. 21: Solution for a map of Canada with four colors. The graph comprises 13 nodes representing provinces connected by edges representing shared borders. No two nodes connected by an edge share a color.

**Note:** You can copy the complete code for the problem here: *Map Coloring: Full Code.*
Multiple-Gate Circuit

This example solves a logic circuit problem to demonstrate using Ocean tools to solve a problem on a D-Wave system. It builds on the discussion in the Boolean AND Gate example about the effect of minor-embedding on performance.

A simple circuit is shown in Figure 22.

Fig. 22: Logic circuit that implements $z = a \overline{b} c + a \overline{b} \overline{c} + b \overline{d} + \overline{c} d$.

Example Requirements

To run the code in this example, the following is required.

- The requisite information for problem submission through SAPI, as described in Configuring Access to D-Wave Solvers
- Ocean tools `dwavebinarycsp` and `dwave-system`. For the optional graphics, you will also need Matplotlib and `problem-inspector`.

If you installed `dwave-ocean-sdk` and ran `dwave setup`, your installation should meet these requirements. In D-Wave’s Leap IDE, the default workspace meets these requirements.

Formulating the Problem as a CSP

This example demonstrates two formulations of constraints from the problem’s logic gates:
1. Single comprehensive constraint.

```python
def logic_circuit(a, b, c, d, z):
    not1 = not b
    or2 = b or c
    and3 = a and not1
    or4 = or2 or d
    and5 = and3 and or4
    not6 = not or4
    or7 = and5 or not6
    return (z == or7)
```

csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
csp.add_constraint(logic_circuit, ['a', 'b', 'c', 'd', 'z'])
```

1. Multiple small constraints.

```python
import dwavebinarycsp
import dwavebinarycsp.factories.constraint.gates as gates
import operator

import dwavebinarycsp
import dwavebinarycsp.factories.constraint.gates as gates
import operator

csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
csp.add_constraint(operator.ne, ['b', 'not1'])  # add NOT 1 gate

csp.add_constraint(gates.or_gate(['b', 'c', 'or2']))  # add OR 2 gate

csp.add_constraint(gates.and_gate(['a', 'not1', 'and3']))  # add AND 3 gate

csp.add_constraint(gates.or_gate(['d', 'or2', 'or4']))  # add OR 4 gate

csp.add_constraint(gates.and_gate(['and3', 'or4', 'and5']))  # add AND 5 gate

csp.add_constraint(operator.ne, ['or4', 'not6'])  # add NOT 6 gate

csp.add_constraint(gates.or_gate(['and5', 'not6', 'z']))  # add OR 7 gate
```

**Note:** `dwavebinarycsp` works best for constraints of up to 4 variables; it may not function as expected for constraints of over 8 variables.

The next line of code converts the constraints into a BQM that we solve by sampling.

```python
# Convert the binary constraint satisfaction problem to a binary quadratic model
bqm = dwavebinarycsp.stitch(csp)
```

**Multiple-Gate Circuit: Further Details**

**Single Comprehensive Constraint**

Binary Quadratic Model:

```
BinaryQuadraticModel({'a': 6.0, 'c': -2.0, 'z': 6.0, 'd': 0.0, 'b': -2.0, 'aux0': 0.0, 'aux1': 0.0}, {('aux1', 'a'): -4.0, ('z', 'a'): 0.0, ('aux1', 'aux0'): 0.0, ('z', 'aux1'): -4.0, ('b', 'a'): -2.0, ('d', 'c'): 0.0, ('z', 'c'): -2.0, ('z', 'd'): 0.0, ('z', 'aux0'): -4.0, ('z', 'b'): 4.0, ('c', 'a'): -2.0, ('c', 'aux0'): 4.0, ('c', 'b'): 2.0, ('d', 'a'): -2.0, ('d', 'aux1'): 4.0, ('d', 'aux0'): -2.0, ('b', 'aux0'): 2.0,
``` (continues on next page)
Embedding 1:

{('aux0', 'a'): -2.0, ('c', 'aux1'): 2.0, ('b', 'aux1'): 0.0, ('d', 'b'): 0.0},
-2.5, Vartype.BINARY)

Embedding 2:

{('a': [946, 951, 959],
 'aux0': [954, 957],
 'aux1': [955, 958, 959],
 'b': [948, 956],
 'c': [944, 958, 950],
 'd': [953, 825, 831, 824],
 'z': [952])

Embedding 3:

{('a': [1838, 1846, 1840, 1847],
 'aux0': [1839],
 'aux1': [1836, 1835],
 'b': [1837, 1831, 1829, 1826, 1830],
 'c': [1706, 1834, 1711],
 'd': [1705, 1833, 1709],
 'z': [1704, 1832])

Embedding 4:

{('a': [665, 537, 540, 539],
 'aux0': [669, 661, 656],
 'aux1': [664, 668],
 'b': [670, 662, 659],
 'c': [667],
 'd': [666, 794, 798, 793, 795],
 'z': [671, 663])

Multiple Small Constraints

Binary Quadratic Model:

BinaryQuadraticModel({'a': 0.0, 'c': 2.0, 'b': 0.0, 'not1': -2.0, 'd': 2.0,
 'or4': 0.0, 'or2': 4.0, 'not6': 0.0, 'and5': 8.0, 'z': 2.0, 'and3': 6.0},
{('not6', 'and5'): 2.0, ('or2', 'c'): -4.0, ('or4', 'd'): -4.0, ('or4', 'and3'): 2.0,
 ('and3', 'a'): -4.0, ('and5', 'and3'): -4.0, ('z', 'and5'): -4.0, ('or4', 'not6'): 4.
 →0,
 ('or4', 'or2'): -4.0, ('z', 'not6'): -4.0, ('or2', 'd'): 2.0, ('or4', 'and5'): -4.0,
 ('c', 'b'): 2.0, ('b', 'not1'): 4.0, ('not1', 'and3'): -4.0, ('or2', 'b'): -4.0,
 ('not1', 'a'): 2.0}, -5.5, Vartype.BINARY)
The first approach, which consolidates the circuit as a single constraint, yields a binary quadratic model with 7 variables: 4 inputs, 1 output, and 2 ancillary variables. The second approach, which creates a constraint satisfaction
problem from multiple small constraints, yields a binary quadratic model with 11 variables: 4 inputs, 1 output, and 6 intermediate outputs of the logic gates.

You can see the binary quadratic models here: *Multiple-Gate Circuit: Further Details*.

**Minor-Embedding and Sampling**

Algorithmic minor-embedding is heuristic—solution results vary significantly based on the minor-embedding found. The next code sets up a D-Wave system as the sampler.

**Note:** The code below sets a sampler without specifying SAPI parameters. Configure a default solver as described in *Configuring Access to D-Wave Solvers* to run the code as is, or see *dwave-cloud-client* to access a particular solver by setting explicit parameters in your code or environment variables.

```python
from dwave.system import DWaveSampler, EmbeddingComposite

# Set up a D-Wave system as the sampler
sampler = EmbeddingComposite(DWaveSampler())
```

Next, we ask for 1000 samples and separate those that satisfy the CSP from those that fail to do so.

```python
sampleset = sampler.sample(bqm, num_reads=1000)

# Check how many solutions meet the constraints (are valid)
valid, invalid, data = 0, 0, []
for datum in sampleset.data(['sample', 'energy', 'num_occurrences']):
    if csp.check(datum.sample):
        valid = valid+datum.num_occurrences
        for i in range(datum.num_occurrences):
            data.append((datum.sample, datum.energy, '1'))
    else:
        invalid = invalid+datum.num_occurrences
        for i in range(datum.num_occurrences):
            data.append((datum.sample, datum.energy, '0'))

>>> print(valid, invalid) # doctest: +SKIP
```

For the single constraint approach, 4 runs with their different minor-embeddings yield significantly varied results, as shown in the following table:

<table>
<thead>
<tr>
<th>Embedding</th>
<th>(valid, invalid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(39, 961)</td>
</tr>
<tr>
<td>2</td>
<td>(1000, 0)</td>
</tr>
<tr>
<td>3</td>
<td>(998, 2)</td>
</tr>
<tr>
<td>4</td>
<td>(316, 684)</td>
</tr>
</tbody>
</table>

You can see the minor-embeddings found here: *Multiple-Gate Circuit: Further Details*; below those embeddings are visualized graphically.

Optionally, you can use the *problem-inspector* to view the solution on the QPU.
Fig. 24: Each of the figure’s 4 panels shows a minor-embedding found for one run of the example code above. The panels show part of the Chimera graph representation of a D-Wave QPU, where each unit cell is rendered as a cross of 4 horizontal and 4 vertical dots representing qubits and lines representing couplers between qubit pairs. Color indicates the strengths of linear (qubit) and quadratic (coupler) biases: darker blue for increasingly negative values and darker red for increasingly positive values.
Note: The next code requires the use of Ocean’s problem inspector.

```python
>>> import dwave.inspector
>>> dwave.inspector.show(sampleset)  # doctest: +SKIP
```

Fig. 25: View of the logical and embedded problem rendered by Ocean’s problem inspector for one arbitrary execution. The CSP’s original BQM, on the left, shows that the solution shown for variable \( z \) is based on a broken chain; its embedded representation, on the right, shows the broken three-qubit chain for variable \( z \) highlighted red. The current solution displayed for variable \( z \) is based on two qubits with spin value \(-1\) and one with value \(1\), and thus represented in the problem space with a broken dot (part white, part gold).

For the second approach, which creates a constraint satisfaction problem based on multiple small constraints, a larger number of variables (11 versus 7) need to be minor-embedded, resulting in worse performance. However, performance can be greatly improved in this case by increasing the chain strength (to 2 instead of the default of 1).

<table>
<thead>
<tr>
<th>Embedding</th>
<th>Chain Strength</th>
<th>(valid, invalid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>(7, 993)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>(417, 583)</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>(941, 59)</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>(923, 77)</td>
</tr>
</tbody>
</table>

You can see the minor-embeddings used here: *Multiple-Gate Circuit: Further Details*; below those embeddings are visualized graphically.

1.4. Examples
Fig. 26: Each of the figure’s 4 panels shows a minor-embedding found for one run of the example code above, as described for the previous figure. Here, the top two panels are for runs with the default chain-strength of 1 and the bottom two for chain-strengths of 2.
Looking at the Results

You can verify the solution to the circuit problem by checking an arbitrary valid or invalid sample:

```python
>>> print(sampleset.first.sample)  # doctest: +SKIP
{'a': 1, 'c': 0, 'b': 0, 'not1': 1, 'd': 1, 'or4': 1, 'or2': 0, 'not6': 0, 'and5': 1, 'z': 1, 'and3': 1}
```

For the lowest-energy sample of the last run, found above, the inputs are \( a, b, c, d = 1, 0, 0, 1 \) and the output is \( z = 1 \), which indeed matches the analytical solution for the circuit,

\[
z = \bar{b}(ac + ad + \bar{c}d) \\
= 1(0 + 1 + 0) \\
= 1
\]

The example code above converted the constraint satisfaction problem to a binary quadratic model using the default minimum energy gap of 2. Therefore, each constraint violated by the solution increases the energy level of the binary quadratic model by at least 2 relative to ground energy. You can also plot the energies for valid and invalid samples:

```python
import matplotlib.pyplot as plt
plt.ion()
plt.scatter(range(len(data)), [x[1] for x in data], c=['y' if (x[2] == '1') else 'r' for x in data], marker='.')
plt.xlabel('Sample')
plt.ylabel('Energy')
```

Fig. 27: Energies per sample for a 1000-sample problem submission of the logic circuit. Blue points represent valid solutions (solutions that solve the constraint satisfaction problem) and red points the invalid solutions. You can see in the graph that valid solutions have energy -9.5 and invalid solutions energies of -7.5, -5.5, and -3.5.

```python
>>> for datum in sampleset.data(['sample', 'energy', 'num_occurrences', 'chain_break_fraction']): # doctest: +SKIP
...     print(datum)
...
Sample(sample={'a': 1, 'c': 0, 'b': 1, 'not1': 0, 'd': 1, 'or4': 1, 'or2': 1, 'not6': 0, 'and5': 0, 'z': 0, 'and3': 0}, energy=-9.5, num_occurrences=13, chain_break_fraction=0.0)
Sample(sample={'a': 1, 'c': 1, 'b': 1, 'not1': 0, 'd': 0, 'or4': 1, 'or2': 1, 'not6': 0, 'and5': 0, 'z': 0, 'and3': 0}, energy=-9.5, num_occurrences=14, chain_break_fraction=0.0)
```

(continues on next page)
You can see, for example, that sample:

```
Sample(sample={'a': 1, 'c': 1, 'b': 0, 'not1': 1, 'd': 1, 'or4': 1, 'or2': 1, 'not6': 1, 'and5': 1, 'z': 1, 'and3': 1}, energy=-7.5, num_occurrences=1, chain_break_fraction=0.18181818181818182)
```

has a higher energy by 2 than the ground energy. It is expected that this solution violates a single constraint, and you can see that it violates constraint:

```
Constraint.from_configurations(frozenset([(1, 0, 0), (0, 1, 0), (0, 0, 0), (1, 1, 1)]), ('a', 'not1', 'and3'), Vartype.BINARY, name='AND')
```

on AND gate 3.

Note also that for samples with higher energy there tends to be an increasing fraction of broken chains: zero for the valid solutions but rising to almost 30% for solutions that have three broken constraints.

**Problem With Many Variables**

This example solves a graph problem with too many variables to fit onto the QPU.
The purpose of this example is to illustrate a hybrid solution—the combining of classical and quantum resources—to a problem that cannot be mapped in its entirety to the D-Wave system due to the number of its variables. Hard optimization problems might have many variables; for example, scheduling or allocation of resources. In such cases, quantum resources are used as an accelerator much as GPUs are for graphics.

**Note:** For fully connected graphs, the number of edges grows very quickly with increased nodes, degrading performance. The current example uses 100 nodes but with a degree of three (each node connects to three other nodes). You can increase the number of nodes substantially as long as you keep the graph sparse.

---

### Example Requirements

To run the code in this example, the following is required.

- The requisite information for problem submission through SAPI, as described in *Configuring Access to D-Wave Solvers*.
- Ocean tools *dwave-system*, *dimod*, and *dwave-hybrid*.

If you installed *dwave-ocean-sdk* and ran *dwave setup*, your installation should meet these requirements. In D-Wave’s Leap IDE, the default workspace meets these requirements.

### Solution Steps

Section *How a D-Wave System Solves Problems* describes the process of solving problems on the quantum computer in two steps: (1) Formulate the problem as a *binary quadratic model* (BQM) and (2) Solve the BQM with a D-wave system or classical sampler. This example uses *dwave-hybrid* to combine a tabu search on a CPU with the submission of parts of the (large) problem to the D-Wave system.

### Formulate the Problem

This example uses a synthetic problem for illustrative purposes: a NetworkX generated graph, NetworkX *barabasi_albert_graph()*, with random +1 or -1 couplings assigned to its edges.

```python
# Represent the graph problem as a binary quadratic model
import dimod
import networkx as nx
import random

graph = nx.barabasi_albert_graph(100, 3, seed=1)  # Build a quasi-random graph

# Set node and edge values for the problem
h = {v: 0.0 for v in graph.nodes}
J = {edge: random.choice([-1, 1]) for edge in graph.edges}
bqm = dimod.BQM(h, J, offset=0, vartype=dimod.SPIN)
```

### Create a Hybrid Workflow

The following simple workflow uses a *RacingBranches* class to iterate two *Branch* classes in parallel: a tabu search, *InterruptableTabuSampler*, which is interrupted to potentially incorporate samples from subproblems (subsets of the problem variables and structure) by *EnergyImpactDecomposer *|* QPUSubproblemAutoEmbeddingSampler *|* SplatComposer*, which decomposes the problem by selecting variables with the greatest energy impact, submits these to the D-Wave system, and merges the subproblem’s
samples into the latest problem samples. In this case, subproblems contain 30 variables in a rolling window that can cover up to 75 percent of the problem’s variables.

```python
# Set a workflow of tabu search in parallel to submissions to a D-Wave system
import hybrid
workflow = hybrid.Loop(
    hybrid.RacingBranches(
        hybrid.InterruptableTabuSampler(),
        hybrid.EnergyImpactDecomposer(size=30, rolling=True, rolling_history=0.75)
        | hybrid.QPUSubproblemAutoEmbeddingSampler()
        | hybrid.SplatComposer())
    | hybrid.ArgMin(), convergence=3)
```

**Solve the Problem Using Hybrid Resources**

Once you have a hybrid workflow, you can run and tune it within the dwave-hybrid framework or convert it to a `dimod` sampler.

```python
# Convert to dimod sampler and run workflow
result = hybrid.HybridSampler(workflow).sample(bqm)
```

While the tabu search runs locally, one or more subproblems are sent to the QPU.

```python
>>> print("Solution: sample=") .format(result.first)) # doctest: +SKIP
Solution: sample=Sample(sample={0: -1, 1: -1, 2: -1, 3: 1, 4: -1, ... energy=-169.0,
num_occurrences=1)
```

- *Map Coloring* example solves a more complex constraint satisfaction problem.
- *Multiple-Gate Circuit* looks more deeply at `minor-embedding`.
- *Problem With Many Variables* builds a hybrid workflow and solver for a large graph problem.

**1.4.4 Advanced-Level Examples**

**Using the Problem Inspector**

This example solves a graph partitioning problem to show how D-Wave’s Problem Inspector tool can help you evaluate the `minor-embedding` used in your problem submissions to the quantum computer.

**Example Requirements**

To run the code in this example, the following is required.

- The requisite information for problem submission through SAPI, as described in *Configuring Access to D-Wave Solvers*.
- Ocean tools `dwave-system` and `dwave-inspector`.

If you installed `dwave-ocean-sdk` and ran `dwave setup`, your installation should meet these requirements. In D-Wave’s Leap IDE, the default workspace meets these requirements.
Solution Steps

The How a D-Wave System Solves Problems section describes the process of solving problems on the quantum computer in two steps: (1) Formulate the problem as a binary quadratic model (BQM) and (2) Solve the BQM with a D-Wave system or classical sampler. In this example, a QUBO is formulated with simple math, the problem is submitted naively to the QPU, its minor embedding examined using the problem inspector, and the submission improved.

Formulate the Problem

This example uses a synthetic problem for illustrative purposes: a NetworkX generated graph, NetworkX random_geometric_graph(). The problem of interest here, which is NP-hard, is to try and find the best division of the graph’s nodes into two equal sets with a minimum number of edges between the two groups.

```python
import networkx as nx
graph_nodes = 16
G = nx.random_geometric_graph(n=graph_nodes, radius=.5, dim=2)
```

![Fig. 29: One arbitrary generation of the problem graph.](image)

Reformulate the Problem Graph as a BQM

This example formulates the BQM as a QUBO using the same steps described in detail in the Graph Partitioning code example of the D-Wave Code Examples GitHub repository.

```python
from collections import defaultdict from itertools import combinations

gamma = 60
```

(continues on next page)
Q = defaultdict(int)

# Fill in Q matrix
for u, v in G.edges:
    Q[(u, u)] += 1
    Q[(v, v)] += 1
    Q[(u, v)] += -2

for i in G.nodes:
    Q[(i, i)] += gamma*(1-len(G.nodes))

for i, j in combinations(G.nodes, 2):
    Q[(i, j)] += 2*gamma

Print the range of values for the generated QUBO’s elements:

```python
>>> print("Maximum element is {:.2f} and minimum is {:.2f}.").format(max(Q.values()),
     __min(Q.values())))  # doctest: +SKIP
Maximum element is 120.00 and minimum is -898.00.
```

Solve the Problem by Sampling

Note: Importing the problem inspector activates for the session the capture of data such as problems sent to the QPU and returned responses, relevant details of minor-embedding, and warnings. The recommended workflow is to import it at the start of your coding session or at least before submitting your problem, as is done below.

```python
import numpy as np
from dwave.system import DWaveSampler, EmbeddingComposite

# Import the problem inspector to begin data capture
import dwave.inspector

sampler = EmbeddingComposite(DWaveSampler(solver={'qpu': True}))

num_reads = 1000
sampleset = sampler.sample_qubo(Q, num_reads=num_reads)

Check the best returned answer:

```python
>>> print("Number of nodes in one set is {}, in the other, {}. \nEnergy is {}.").format(
     ... sum(sampleset.first.sample.values()),
     ... graph_nodes - sum(sampleset.first.sample.values()),
     ... sampleset.first.energy))  # doctest: +SKIP
Number of nodes in one set is 8, in the other, 8.
Energy is -3813.0.
```

One simple measure of the overall quality of the returned samples is the percentage of samples based on chains with high breakage rates. Here a rate above one third is chosen as the acceptable threshold:

```python
>>> print("Percentage of samples with high rates of breaks is {}.").format(
     ... np.count_nonzero(sampleset.record.chain_break_fraction > 0.33)/num_
     ... reads*100))  # doctest: +SKIP
```

(continues on next page)
Fig. 30: Best graph partition found for a submission with a lowest energy value of \(-3813\).

Percentage of samples with high rates of breaks is 78.7.

**Inspect the Submission**

Use the problem inspector on the returned samples:

```python
>>> dwave.inspector.show(sampleset)  # doctest: +SKIP
```

The problem inspector can also display the embedded problem, showing the qubits chains viewed on a background representation of the QPU topology:

Using the same logic described in the Graph Partitioning code example, the problem is resubmitted using a higher chain strength:

```python
>>> sampleset = sampler.sample_qubo(Q, num_reads=num_reads, chain_strength=1000)
```

Check the best returned answer and percentage of samples based on chains with breakage rates of over 33 percent. Results will vary due to the probabilistic nature of the quantum computer and its integrated control errors (ICE), but in this case the shown submission had a lower minimum energy and no samples based on high rates of broken chains.

```python
>>> print("Number of nodes in one set is {}, in the other, {}. \nEnergy is {}.").format(
...     sum(sampleset.first.sample.values()),
...     graph_nodes - sum(sampleset.first.sample.values()),
...     sampleset.first.energy))  # doctest: +SKIP
Number of nodes in one set is 8, in the other, 8.
Energy is -3815.0.
```

```python
>>> print("Percentage of samples with high rates of breaks is {}.").format(
... )
```

(continues on next page)
Fig. 31: Problem inspector displaying the logical problem: the problem BQM, on the left, and the returned energies histogram, on the right, for a submission with the chain strength set to its default value. For one arbitrary solution, selected by clicking an energy bar on the right (highlighted white), all values but two are shown as being based on broken chains. The selected variable on the left, variable 8, is shown to be represented on the QPU by a chain of 6 qubits.
Fig. 32: Problem inspector displaying the embedded problem, with broken chains highlighted, for a submission with the chain strength set to its default value.

(continued from previous page)

```python
... np.count_nonzero(sampleset.record.chain_break_fraction > 0.33) / num_reads * 100)
# doctest: +SKIP
Percentage of samples with high rates of breaks is 0.0.
```

If you again use the problem inspector on the returned samples, you see the improved chains.

```python
>>> dwave.inspector.show(sampleset)  # doctest: +SKIP
```

Also of interest is the “spread” of solution energies. For this submission there are a few distinct clusters. The problem inspector can zoom in on the lowest:

You see that most the returned solutions of lowest energy cluster closely around an energy of approximately -3807 but that the QPU has found some even lower-energy solutions. From this one can assume that it might be possible find a better solution by increasing the number of reads. Additionally however, the complete lack of broken chains for the current returned sample set suggests that the chain strength can likely be lowered while still maintaining a low rate of broken chains. Doing so enables the problem to be represented more accurately on the QPU.

```python
>>> sampleset = sampler.sample_qubo(Q, num_reads=num_reads, chain_strength=300)
```

Below is one run of a few iterations of adjusting chain strength. Notice that the acceptable rate of chain breaks was set lower, to breakage rates of over 5 percent.

```python
>>> print("Number of nodes in one set is {}, in the other, {}. Energy is {}.").format(
...    sum(sampleset.first.sample.values()),
...    graph_nodes - sum(sampleset.first.sample.values()),
```

(continues on next page)
Fig. 33: Best graph partition found for a submission with higher chain strength.

Fig. 34: Problem inspector displaying the logical problem: BQM and the returned energies histogram for a submission with the chain strength set to 1000. For one arbitrary solution, selected by clicking an energy bar on the right (highlighted white), all values are shown as being based on non-broken chains.
Fig. 35: Zoom on lowest-energy returned samples for chain strength set to a value of 1000.

The result of the shown submission, with a chain strength of 300, still had less than 2% of its samples based on broken chains.

**Working With Different Topologies**

The examples shows how to construct software samplers with the same structure as the QPU, and how to work with embeddings with different topologies.

The code examples below will use the following imports:

```python
>>> import neal
>>> import dimod
>>> import dwave_networkx as dnx
>>> import networkx as nx
>>> import dwave.embedding
...
>>> from dwave.system import DWaveSampler, EmbeddingComposite
```
Creating a Chimera Sampler

As detailed in Classical Solvers, you might want to use a classical solver while developing your code or writing tests. However, it is sometimes useful to work with a software solver that behaves more like a quantum computer.

One of the key features of the quantum computer is its working graph, which defines the connectivity allowed by the binary quadratic model.

To create a software solver with the same connectivity as a D-Wave 2000Q quantum computer we first need a representation of the Chimera graph which can be obtained from the dwave_networkx project using the `chimera_graph()` function.

```python
>>> C16 = dnx.chimera_graph(16)
```

Next, we need a software sampler. We will use the `neal.SimulatedAnnealingSampler` found in dwave_neal, though the `tabu.TabuSampler` from dwave-tabu would work equally well.

```python
>>> classical_sampler = neal.SimulatedAnnealingSampler()
```

Now, with a classical sampler and the desired graph, we can use dimod's `dimod.StructuredComposite` to create a Chimera-structured sampler.

```python
>>> sampler = dimod.StructureComposite(classical_sampler, C16.nodes, C16.edges)
```

This sampler accepts Chimera-structured problems. In this case we create an Ising problem.

```python
>>> h = {v: 0.0 for v in C16.nodes}
>>> J = {(u, v): 1 for u, v in C16.edges}
>>> sampleset = sampler.sample_ising(h, J)
```

We can even use the sampler with the `dwave.system.EmbeddingComposite`
Finally, we can confirm that our sampler matches the `dwave.system.DWaveSampler`'s structure. We make sure that our QPU has the same topology we have been simulating. Also note that the working graph of the QPU is usually a subgraph of the full hardware graph.

Creating a Pegasus Sampler

Another topology of interest is the Pegasus topology. As above, we can use the generator function `dwave_networkx.pegasus_graph()` found in `dwave_networkx` and the `neal.SimulatedAnnealingSampler` found in `dwave_neal` to construct a sampler.

Working With Embeddings

The example above using the `EmbeddingComposite` hints that we might be interested in trying embedding with different topologies. One thing we might be interested in is the chain length when embedding our problem. Say that we have a fully connected problem with 40 variables and we want to know the chain length needed to embed it on a 2048 node Chimera graph.

We can use `dwave-system`'s `find_clique_embedding()` function to find the embedding and determine the maximum chain length.

Similarly we can explore clique embeddings for a 40-variables fully connected problem with a 680 node Pegasus graph using `dwave-system`'s `find_clique_embedding()` function.

- Using the Problem Inspector improves minor-embedding on a graph partition problem.
- Working With Different Topologies runs your code on software samplers with different QPU-inspired topologies.
1.5 Demonstrations and Jupyter Notebooks

D-Wave’s dwave-examples GitHub repo contains demos, typically in the form of short code examples, you can open in the Leap IDE or copy (clone) locally and run.

D-Wave’s Leap Quantum Application Environment provides a number of Jupyter Notebooks with detailed code examples for various types of problems (for example, constraint satisfaction problems) and ways of using the quantum computer (for example, hybrid computing and reverse annealing). These can also serve as a framework in which to develop your own code.

1.6 Additional Tutorials

- **Getting Started with the D-Wave System**

  This guide in the System Documentation introduces the D-Wave quantum computer, provides some key background information on how the system works, and explains how to construct a simple problem that the system can solve.

- **D-Wave Problem-Solving Handbook**

  This guide for more advanced users has an opening chapter of illustrative examples that explain the main steps of solving problems on the D-Wave system through two “toy” problems.
CHAPTER 2

Concepts

See the glossary for short definitions of terminology or learn Ocean concepts here:

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Related terms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binary Quadratic Models</strong></td>
<td>BQM, Ising, QUBO</td>
</tr>
<tr>
<td><strong>Constraint Satisfaction</strong></td>
<td>CSP, binary CSP</td>
</tr>
<tr>
<td><strong>Hybrid</strong></td>
<td>quantum-classical hybrid, Leap’s hybrid solvers, hybrid workflows</td>
</tr>
<tr>
<td><strong>Minor-Embedding</strong></td>
<td>embedding, mapping logical variables to physical qubits, chains, chain strength</td>
</tr>
<tr>
<td><strong>QPU Topology</strong></td>
<td>Chimera, Pegasus</td>
</tr>
<tr>
<td><strong>Samplers and Composites</strong></td>
<td>solver</td>
</tr>
<tr>
<td><strong>Solutions</strong></td>
<td>samples, sampleset, probabilistic, energy</td>
</tr>
</tbody>
</table>

### 2.1 Binary Quadratic Models

The binary quadratic model (BQM) class encodes Ising and quadratic unconstrained binary optimization (QUBO) models used by samplers such as the D-Wave system.

The BQM equation,

$$E(v) = \sum_{i=1} a_i v_i + \sum_{i<j} b_{i,j} v_i v_j + c \quad v_i \in \{-1, +1\} \text{ or } \{0, 1\}$$

can represent both.

The *Ising* model is an objective function of $N$ variables $s = [s_1, ..., s_N]$ corresponding to physical Ising spins, where $h_i$ are the biases and $J_{i,j}$ the couplings (interactions) between spins.

$$\text{Ising:} \quad E(s) = \sum_{i=1} h_i s_i + \sum_{i<j} J_{i,j} s_i s_j \quad s_i \in \{-1, +1\}$$
The \textit{QUBO} model is an objective function of $N$ binary variables represented as an upper-diagonal matrix $Q$, where diagonal terms are the linear coefficients and the nonzero off-diagonal terms the quadratic coefficients.

\[
\text{QUBO: } E(x) = \sum_{i \leq j} x_i Q_{i,j} x_j \quad x_i \in \{0, 1\}
\]

The \textit{BinaryQuadraticModel} class can contain both these models and its methods provide convenient utilities for working with, and interworking between, the two representations of a problem.

These models and their use in solving problems on the D-Wave system is described in the following documentation:

- **Getting Started with the D-Wave System**
  Introduces key concepts such as objective functions, Ising models, QUBOs, and graphs, explains how these models are used to represent problems, and provides some simple examples.

- **D-Wave Problem-Solving Handbook**
  Provides a variety of techniques for, and examples of, reformulating problems as BQMs.

- **Solving Problems on a D-Wave System**
  Describes and demonstrates the use of BQM in the context of Ocean software.

### 2.2 Constraint Satisfaction

A constraint satisfaction problem (CSP) requires that all the problem’s variables be assigned values, out of a finite domain, that result in the satisfying of all constraints.

The map-coloring CSP, for example, is to assign a color to each region of a map such that any two regions sharing a border have different colors.

![Coloring a map of Canada with four colors.](image)

The constraints for the map-coloring problem can be expressed as follows:

- Each region is assigned one color only, of $C$ possible colors.
- The color assigned to one region cannot be assigned to adjacent regions.

A finite domain CSP consists of a set of variables, a specification of the domain of each variable, and a specification of the constraints over combinations of the allowed values of the variables. A constraint $C_{\alpha}(x_{\alpha})$ defined over a subset of variables $x_{\alpha}$ defines the set of feasible and infeasible combinations of $x_{\alpha}$. The constraint $C_{\alpha}$ may be viewed as
a predicate which evaluates to true on feasible configurations and to false on infeasible configurations. For example, if the domains of variables $X_1, X_2, X_3$ are all $\{0, 1, 2\}$, and the constraint is $X_1 + X_2 < X_3$ then the feasible set is $\{(0, 0, 1), (0, 0, 2), (0, 1, 2), (1, 0, 2)\}$, and all remaining combinations are infeasible.

### 2.2.1 Binary CSPs

Solving such problems as the map-coloring CSP on a sampler such as the D-Wave system necessitates that the mathematical formulation use binary variables because the solution is implemented physically with qubits, and so must translate to spins $s_i \in \{-1, +1\}$ or equivalent binary values $x_i \in \{0, 1\}$. This means that in formulating the problem by stating it mathematically, you might use unary encoding to represent the $C$ colors: each region is represented by $C$ variables, one for each possible color, which is set to value 1 if selected, while the remaining $C - 1$ variables are 0.

Another example is logical circuits. Logic gates such as AND, OR, NOT, XOR etc can be viewed as binary CSPs: the mathematically expressed relationships between binary inputs and outputs must meet certain validity conditions. For inputs $x_1, x_2$ and output $y$ of an AND gate, for example, the constraint to satisfy, $y = x_1 \cdot x_2$, can be expressed as a set of valid configurations: $(0, 0, 0)$, $(0, 1, 0)$, $(1, 0, 0)$, $(1, 1, 1)$, where the variable order is $(x_1, x_2, y)$.

<table>
<thead>
<tr>
<th>$x_1, x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 0</td>
<td>0</td>
</tr>
<tr>
<td>0, 1</td>
<td>0</td>
</tr>
<tr>
<td>1, 0</td>
<td>0</td>
</tr>
<tr>
<td>1, 1</td>
<td>1</td>
</tr>
</tbody>
</table>

You can use Ocean’s dwavebinarycsp to construct a BQM from a CSP. It maps each individual constraint in the CSP to a ‘small’ Ising model or QUBO, in a mapping called a penalty model.

For more information on using the D-Wave system to solve CSPs, see the following documentation:

- **Getting Started with the D-Wave System**
  
  Introduces the use of QUBOs to represent constraints in some simple examples.

- **D-Wave Problem-Solving Handbook**
  
  Provides a variety of techniques for, and examples of, reformulating CSPs as BQMs.

### 2.3 Hybrid

Quantum-classical hybrid is the use of both classical and quantum resources to solve problems, exploiting the complementary strengths that each provides. As quantum processors grow in size, offloading hard optimization problems to quantum computers promises performance benefits similar to CPUs’ outsourcing of compute-intensive graphics-display processing to GPUs.

For an overview of, and motivation for, hybrid computing, see this Medium Article.

D-Wave’s Leap quantum cloud service provides cloud-based hybrid solvers you can submit arbitrary BQMs to. These solvers, which implement state-of-the-art classical algorithms together with intelligent allocation of the quantum processing unit (QPU) to parts of the problem where it benefits most, are designed to accommodate even very large problems. Leap’s solvers can relieve you of the burden of any current and future development and optimization of hybrid algorithms that best solve your problem.

**dwave-hybrid** provides you with a Python framework for building a variety of flexible hybrid workflows. These use quantum and classical resources together to find good solutions to your problem. For example, a hybrid workflow
might use classical resources to find a problem’s hard core and send that to the QPU, or break a large problem into smaller pieces that can be solved on a QPU and then recombined.

The dwave-hybrid framework enables rapid development of experimental prototypes, which provide insight into expected performance of the productized versions. It provides reference samplers and workflows you can quickly plug into your application code. You can easily experiment with customizing workflows that best solve your problem. You can also develop your own hybrid components to optimize performance.

For more information on hybrid computing, see the following:

- dwave-hybrid
  Describes how to use reference hybrid solvers, build hybrid workflows, and your own hybrid components.
- Using Leap’s Hybrid Solvers
  Introduces Leap’s quantum-classical hybrid solvers and provides references to usage information.
- Getting Started Demonstrations and Jupyter Notebooks
  Provides pointers to a code-examples repository and Jupyter Notebooks, which have relevant content.

### 2.4 Minor-Embedding

To solve an arbitrarily posed binary quadratic problem directly on a D-Wave system requires mapping, called **minor embedding**, to a Chimera graph that represents the system’s quantum processing unit. This preprocessing can be done by a composed sampler consisting of the DWaveSampler() and a composite that performs minor-embedding. (This step is handled automatically by LeapHybridSampler() and dwave-hybrid reference samplers.)

For example, a simple two-variable bqm,

\[ E(s) = -s_0s_1 \]

\[ s_i \in \{-1, +1\} \]

might be embedded to two connected qubits, such as 1929 and 1801 on a D-Wave 2000Q system:

![Fig. 2: Two-variable problem, shown on the left as a graph, is embedded in two connected qubits on a D-Wave 2000Q shown on the right against the Chimera topology. Variable \( s_1 \), highlighted in dark magenta, is represented by qubit number 1929 and variable \( s_0 \) is represented by qubit 1801. (This and similar images in this section are generated by Ocean’s problem inspector tool.)](image-url)
$s_0 = 1801, s_0 = 1807$ or $s_0 = 0, s_1 = 4$.

### 2.4.1 Chains

Larger problems often require chains because the QPU topology is not fully connected. For example, a fully-connected $K_3$ three-variable bqm,

$$E(s) = -s_0s_1 + s_0s_2 + s_1s_2 \quad s \in \{-1, +1\}$$

cannot be represented by three qubits in the Chimera topology—a $K_3$ graph is not native to the Chimera graph. (Look at the Chimera “unit cell” shown in the QPU topology section and notice there is no way to connect three qubits in a closed loop to form a triangle graph.)

Instead, a variable is represented by a chain of physical qubits:

![Diagram](image)

**Fig. 3:** Three-variable $K_3$ fully-connected problem, shown on the left as a graph, is embedded in four qubits on a D-Wave 2000Q, shown on the right against the Chimera topology. Variable $s_0$, highlighted in dark magenta, is represented by two qubits, numbers 251 and 253.

The embedding above derived from the heuristic used by `EmbeddingComposite()` on the working graph of a D-Wave 2000Q selected by

`DWaveSampler()`.
sampler = EmbeddingComposite(DWaveSampler(solver={'qpu': True}))

Other qubits might have been chosen; for example,

\[
sampler = \text{FixedEmbeddingComposite}(\text{DWaveSampler}(\text{solver} = \{'qpu': \text{True} \}),
\text{embedding} = \{'s0': [0, 4, 7], 's1': [2], 's2': [3, 6]})
\]

intentionally sets the embedding shown below to represent this same \(K_3\) graph:

emphasized (and displaying a solution of +1).

### 2.4.2 Chain Strength

For a chain of qubits to represent a variable, all its constituent qubits must return the same value for a sample. This is accomplished by setting a strong coupling to the edges connecting these qubits. For the solutions shown above to the \(K_3\) problem, the default chain strength achieved identical values and the qubit chains properly represented the variables of the problem.

However, that is not always the case. For the qubits in a chain to be likely to return identical values, the coupling strength for their connecting edges must be strong compared to the coupling with other qubits that influence non-identical outcomes.

For example, another three-variable \(K_3\) fully-connected BQM,

\[
E(s) = -s_0 s_1 - s_0 s_2 + s_1 s_2 \quad s_i \in \{-1, +1\}
\]

can be embedded by representing one variable with two qubits, for example:

\[
sampler = \text{FixedEmbeddingComposite}(\text{DWaveSampler}(\text{solver} = \{'qpu': \text{True} \}),
\text{embedding} = \{'s0': [0, 4], 's1': [2], 's2': [7]})
\]

This BQM has six ground states (best solutions, shown below—solved by brute-force stepping through all possible configurations of values for the variables—with lowest energy of -1.0):

\[
\begin{array}{cccc}
| \text{variable} | \text{energy} | \text{num_occ} | \\
|---|---|---|
| 0 -1 -1 -1 | -1.0 | 1 \\
| 2 +1 +1 -1 | -1.0 | 1 \\
| 3 -1 +1 -1 | -1.0 | 1 \\
| 5 +1 +1 +1 | -1.0 | 1 \\
| 6 +1 -1 +1 | -1.0 | 1 \\
| 7 -1 -1 +1 | -1.0 | 1 \\
| 1 +1 -1 -1 | 3.0 | 1 \\
| 4 -1 +1 +1 | 3.0 | 1 \\
\end{array}
\]

In this case, the default chain strength is not always sufficient:

\[
\begin{array}{cccc}
| \text{variable} | \text{energy} | \text{num_occ} | \\
|---|---|---|
| 0 -1 +1 -1 | -1.0 | 85 \\
| 1 -1 -1 +1 | -1.0 | 147 \\
\end{array}
\]

(continues on next page)
The solutions of line 2 and 6 above shown a chains broken in a third of the variables, meaning that for variable $s_0$ the two qubits representing it did not return identical values.

Fig. 4: Three-variable $K_3$ fully-connected problem is embedded in six qubits on a D-Wave 2000Q. Variable $s_0$, highlighted in dark magenta, is represented by three qubits, numbers 0, 4, and 7; Variable $s_2$ is represented by two qubits, numbers 3 and 6, shown with their connecting edge.

For information on handling embedding and chains, see the following documentation:

- **Boolean AND Gate, Multiple-Gate Circuit, and Using the Problem Inspector** examples
  
  Shows through some simple examples how to embed and set chain strength.

- **minorminer** tool
  
  Is the hueristic used by common Ocean embedding **composites**.

- **problem inspector** tool
  
  Visualizes embeddings.

- **dwave-system**
  
  Provides embedding composites.
2.5 QPU Topology

To solve a bqm on the D-Wave system, you must map it to a graph that represents the topology of the system’s qubits. For D-Wave 2000Q systems, this is the chimera topology; for next-generation Advantage systems, this is the Pegasus topology.

Note: If you are sending your problem to a Leap quantum-classical hybrid solver, the solver handles all interactions with the QPU.

2.5.1 Chimera

The Chimera architecture comprises sets of connected unit cells, each with four horizontal qubits connected to four vertical qubits via couplers (bipartite connectivity). Unit cells are tiled vertically and horizontally with adjacent qubits connected, creating a lattice of sparsely connected qubits. A unit cell is typically rendered as either a cross or a column.

Chimera qubits are considered to have a nominal length of 4 (each qubit is connected to 4 orthogonal qubits through internal couplers) and degree of 6 (each qubit is coupled to 6 different qubits).

The notation CN refers to a Chimera graph consisting of an N x N grid of unit cells. The D-Wave 2000Q QPU supports a C16 Chimera graph: its 2048 qubits are logically mapped into a 16 x 16 matrix of unit cells of 8 qubits.

2.5.2 Pegasus

In Pegasus as in Chimera, qubits are “oriented” vertically or horizontally but similarly aligned qubits can also be also shifted by distances and in groupings that differ between Pegasus families. Pegasus qubits are also more densely connected and have three types of coupler:
Fig. 7: A 3x3 Chimera graph, denoted C3. Qubits are arranged in 9 unit cells.
• **Internal couplers.** Internal couplers connect pairs of orthogonal (with opposite orientation) qubits. In Pegasus, each qubit is connected via internal coupling to 12 other qubits (versus four in the Chimera topology).

• **External couplers.** External couplers connect vertical qubits to adjacent vertical qubits and horizontal qubits to adjacent horizontal qubits. Each qubit has one or two external couplers.

• **Odd couplers.** Odd couplers connect similarly aligned pairs of qubits. Each qubit has one odd coupler.

Fig. 8: Pegasus qubits. Qubits are drawn as horizontal and vertical loops. The horizontal qubit in the center, shown with its odd coupler in red and numbered 1, is internally coupled to vertical qubits, in pairs 3 through 8, each pair and its odd coupler shown in a different color, and externally coupled to horizontal qubits 2 and 9, each shown in a different color.

Fig. 9: Pegasus qubits. Qubits in this “roadway” graphic are represented as dots and couplers as lines. The top qubit in the center, shown in red and numbered 1, is oddly coupled to the (red) qubit shown directly below it, internally coupled to vertical qubits, in pairs 3 through 8, each pair and its odd coupler shown in a different color, and externally coupled to horizontal qubits 2 and 9, each shown in a different color.

Pegasus qubits are considered to have a nominal length of 12 (each qubit is connected to 12 orthogonal qubits through internal couplers) and degree of 15 (each qubit is coupled to 15 different qubits).

As we use the notation CN to refer to a Chimera graph with size parameter N, we refer to instances of Pegasus topologies by PN; for example, P3 is a graph with 144 nodes.

### 2.6 Samplers and Composites

#### 2.6.1 Samplers

*Samplers* are processes that sample from low energy states of a problem’s *objective function*. A BQM sampler samples from low energy states in models such as those defined by an Ising equation or a Quadratic Unconstrained Binary Optimization (QUBO) problem and returns an iterable of samples, in order of increasing energy.

Ocean software provides a variety of *dimod samplers*, which all support ‘sample_qubo’ and ‘sample_ising’ methods as well as the generic BQM sampler method. In addition to `DWaveSampler()`, classical solvers, which run on CPU or GPU, are available and useful for developing code or on a small versions of a problem to verify code.

**Hybrid Quantum-Classical Samplers**

Quantum-classical hybrid is the use of both classical and quantum resources to solve problems, exploiting the complementary strengths that each provides.

D-Wave’s Leap Quantum Application Environment provides state-of-the-art hybrid solvers you can submit arbitrary BQMs to. `dwave-hybrid` provides you with a Python framework for building a variety of flexible hybrid workflows that use quantum and classical resources together to find good solutions to your problem.
2.6.2 Solvers

Ocean software provides quantum, classical, and quantum-classical hybrid samplers that run either remotely (for example, in D-Wave’s Leap environment) or locally on your CPU. These compute resources are known as solvers.

Note: Some classical samplers actually brute-force solve small problems rather than sample, and these are also referred to as “solvers”.

2.6.3 Composites

Samplers can be composed. The composite pattern allows layers of pre- and post-processing to be applied to binary quadratic programs without needing to change the underlying sampler implementation. We refer to these layers as composites. A composed sampler includes at least one sampler and possibly many composites.

Examples of composites are `EmbeddingComposite()`, which handle the mapping known as minor-embedding, and `RoofDualityComposite()`, which uses roof duality to assign some variables as a pre-processing step before submitting the problem for sampling.

The use of samplers in solving problems is described in the following documentation:

- Solving Problems by Sampling
  
  Describes the available types of samplers in Ocean and their use in solving BQMs.

2.7 Solutions

Samplers sample from low-energy states of a problem’s objective function—BQM samplers sample from low-energy states in models such as those defined by an Ising equation or a QUBO problem—and return an iterable of samples, in order of increasing energy.

When the D-Wave quantum computer solves a problem, it uses quantum phenomena such as superposition and tunneling to explore all possible solutions simultaneously and find a set of the best ones. At the end of the computation (anneal), a single solution is sampled from a set of good solutions, with some probability, and returned. Because the sampled solution is probabilistic, different solutions may be returned in different runs. The standard way of submitting a problem to the system requests many samples, not just one. This not only returns multiple “best” answers but also reduces the probability of settling on a suboptimal answer.

Some classical samplers might return non-probabilistic solutions; for example, the `dimod ExactSolver` deterministically returns the best solution or solutions to small problems by calculating the result for every configuration of variable values. Such samplers are called solvers.

Some Ocean functions might return a single best solution; for example, some `dwave-networkx` graph algorithms return only the lowest-energy sample.

2.7.1 SampleSets

Ocean uses the `dimod SampleSet` class to hold samples and some additional information (e.g., timing information from some samplers).

For the simple example three-variable “triangular” BQM,

\[ E(s) = -s_0s_1 - s_0s_2 + s_1s_2 \quad s_i \in \{-1, +1\} \]
might be solved directly on a D-Wave 2000Q system by sampling 1000 times as follows, where the
EmbeddingComposite composite maps the symbolic BQM to qubits on the quantum processor, which is called
by the DWaveSampler sampler:

```python
>>> bqm = dimod.BQM({}, {('s0', 's1'): -1, ('s0', 's2'): -1, ('s1', 's2'): 1}, 0,
˓→dimod.Vartype.SPIN)
>>> sampler = EmbeddingComposite(DWaveSampler(solver={'qpu': True}))
>>> sampleset = sampler.sample(bqm, num_reads=1000)
>>> print(sampleset)  # doctest: +SKIP
+SKIP
 s0 s1 s2 energy num_oc. chain_b.
 0 -1 -1 +1 -1.0 141 0.0
 1 +1 +1 +1 -1.0 132 0.0
 2 -1 -1 -1 -1.0 159 0.0
 3 -1 +1 -1 -1.0 143 0.333333
 4 +1 +1 -1 -1.0 91 0.0
 5 -1 +1 -1 -1.0 86 0.0
 6 +1 +1 +1 -1.0 129 0.333333
 7 +1 -1 +1 -1.0 119 0.0
['SPIN', 8 rows, 1000 samples, 3 variables]
```

The returned SampleSet, in this case, shows eight solutions of equal energy \(-1.0\). Solution \(s_0 = -1, s_1 = -1, s_2 = +1\) occurred in 141 of the 1000 samples. Two solutions, shown in line 3 and 6, were based on a broken chain of qubits that represented one of the variables.

For this submission to a D-Wave 2000Q, the sampleset also contained the following additional information:

```python
>>> print(sampleset.info.keys())  # doctest: +SKIP
dict_keys(['timing', 'problem_id', 'embedding_context', 'warnings'])
```

For example, the timing information for the problem might look something like:

```python
>>> print(sampleset.info['timing'])  # doctest: +SKIP
{'gpu_sampling_time': 314960,
 'gpu_anneal_time_per_sample': 20,
 'gpu_readout_time_per_sample': 274,
 'gpu_access_time': 324321,
 'gpu_access_overhead_time': 5362,
 'gpu_programming_time': 9361,
 'gpu_delay_time_per_sample': 21,
 'total_post_processing_time': 409,
 'post_processing_overhead_time': 409,
 'total_real_time': 324321,
 'run_time_chip': 314960,
 'anneal_time_per_run': 20,
 'readout_time_per_run': 274)
```

### 2.8 Glossary

**binary quadratic model**

**BQM** A collection of binary-valued variables (variables that can be assigned two values, for example \(-1, 1\)) with associated linear and quadratic biases. Sometimes referred to in other tools as a problem. See a fuller description under Binary Quadratic Models.

**Chain** One or more nodes or qubits in a target graph that represent a single variable in the source graph. See embedding. See a fuller description under Minor-Embedding.
Chain length The number of qubits in a Chain. See a fuller description under Minor-Embedding.

Chain strength Magnitude of the negative quadratic bias applied between variables to form a chain. See a fuller description under Minor-Embedding.

Chimera The D-Wave QPU is a lattice of interconnected qubits. While some qubits connect to others via couplers, the D-Wave QPU is not fully connected. Instead, the qubits interconnect in an architecture known as Chimera. See a fuller description under QPU Topology.

Complete graph

Fully connected See complete graph on wikipedia. A fully connected or complete binary quadratic model is one that has interactions between all of its variables.

Composed sampler Samplers that apply pre- and/or post-processing to binary quadratic programs without changing the underlying sampler implementation by layering composite patterns on the sampler. For example, a composed sampler might add spin transformations when sampling from the D-Wave system.

Composite A sampler can be composed. The composite pattern allows layers of pre- and post-processing to be applied to binary quadratic programs without needing to change the underlying sampler implementation. We refer to these layers as “composites”. A composed sampler includes at least one sampler and possibly many composites.

Embed

Embedding

Minor-embedded

Minor-embedding The nodes and edges on the graph that represents an objective function translate to the qubits and couplers in Chimera. Each logical qubit, in the graph of the objective function, may be represented by one or more physical qubits. The process of mapping the logical qubits to physical qubits is known as minor embedding. See a fuller description under Minor-Embedding.

Excited state States of a quantum system that have higher energy than the ground state. Such states represent non-optimal solutions for problems represented by an Objective function and infeasible configurations for problems represented by a penalty model.

Graph A collection of nodes and edges. A graph can be derived from a model: a node for each variable and an edge for each pair of variables with a non-zero quadratic bias.

Ground state The lowest-energy state of a quantum-mechanical system and the global minimum of a problem represented by an Objective function.

Hamiltonian A classical Hamiltonian is a mathematical description of some physical system in terms of its energies. We can input any particular state of the system, and the Hamiltonian returns the energy for that state. For a quantum system, a Hamiltonian is a function that maps certain states, called eigenstates, to energies. Only when the system is in an eigenstate of the Hamiltonian is its energy well defined and called the eigenenergy. When the system is in any other state, its energy is uncertain. For the D-Wave system, the Hamiltonian may be represented as

\[
\mathcal{H}_{\text{ising}} = \frac{A(s)}{2} \left( \sum_i \hat{\sigma}_x^{(i)} \right) + \frac{B(s)}{2} \left( \sum_i h_i \hat{\sigma}_z^{(i)} + \sum_{i>j} J_{i,j} \hat{\sigma}_z^{(i)} \hat{\sigma}_z^{(j)} \right)
\]  

(2.1)

where \(\hat{\sigma}_x^{(i)}\) are Pauli matrices operating on a qubit \(q_i\), and \(h_i\) and \(J_{i,j}\) are the qubit biases and coupling strengths.

Hardware graph See hardware graph. The hardware graph is the physical lattice of interconnected qubits. See also working graph. See a fuller description under QPU Topology.

Ising Traditionally used in statistical mechanics. Variables are “spin up” (\(\uparrow\)) and “spin down” (\(\downarrow\)), states that correspond to +1 and −1 values. Relationships between the spins, represented by couplings, are correlations or
The objective function expressed as an Ising model is as follows:

\[ E_{\text{ising}}(s) = \sum_{i=1}^{N} h_i s_i + \sum_{i=1}^{N} \sum_{j=i+1}^{N} J_{i,j} s_i s_j \]  

(2.2)

where the linear coefficients corresponding to qubit biases are \( h_i \), and the quadratic coefficients corresponding to coupling strengths are \( J_{i,j} \). See also Ising Model on Wikipedia.

**Minimum gap** The minimum distance between the ground state and the first excited state throughout any point in the anneal.

**Model** A collection of variables with associated linear and quadratic biases. Sometimes referred to as a problem.

**Objective function** A mathematical expression of the energy of a system as a function of binary variables representing the qubits.

**Pegasus** The D-Wave QPU is a lattice of interconnected qubits. While some qubits connect to others via couplers, the D-Wave QPU is not fully connected. Instead, the qubits interconnect in an architecture known as Pegasus. See a fuller description under QPU Topology.

**Penalty function** An algorithm for solving constrained optimization problems. In the context of Ocean tools, penalty functions are typically employed to increase the energy level of a problem’s objective function by penalizing non-valid configurations. See Penalty method on Wikipedia.

**Penalty model** An approach to solving constraint satisfaction problems (CSP) using an Ising model or a QUBO by mapping each individual constraint in the CSP to a ‘small’ Ising model or QUBO.

**QPU** Quantum processing unit

**QUBO** Quadratic unconstrained binary optimization. QUBO problems are traditionally used in computer science. Variables are TRUE and FALSE, states that correspond to 1 and 0 values. A QUBO problem is defined using an upper-diagonal matrix \( Q \), which is an \( N \times N \) upper-triangular matrix of real weights, and \( x \), a vector of binary variables, as minimizing the function

\[ f(x) = \sum_{i} Q_{i,i} x_i + \sum_{i<j} Q_{i,j} x_i x_j \]  

(2.3)

where the diagonal terms \( Q_{i,i} \) are the linear coefficients and the nonzero off-diagonal terms are the quadratic coefficients \( Q_{i,j} \). This can be expressed more concisely as

\[ \min_{x \in \{0,1\}^n} x^T Q x. \]  

(2.4)

In scalar notation, the objective function expressed as a QUBO is as follows:

\[ E_{\text{qubo}}(a_i, b_{i,j}; q_i) = \sum_{i} a_i q_i + \sum_{i<j} b_{i,j} q_i q_j. \]  

(2.5)

See also QUBO on Wikipedia.

**Sampler** Samplers are processes that sample from low energy states of a problem’s objective function, which is a mathematical expression of the energy of a system. A binary quadratic model (BQM) sampler samples from low energy states in models such as those defined by an Ising equation or a QUBO problem and returns an iterable of samples, in order of increasing energy.

**SAPI** Solver API used by clients to communicate with a solver.

**Solver** A resource that runs a problem. Some solvers interface to the QPU; others leverage CPU and GPU resources.

**Source**

**Source graph** In the context of embedding, the model or induced graph that we wish to embed. Sometimes referred to as the logical graph/model. See a fuller description under Minor-Embedding.
Structured sampler  Samplers that are restricted to sampling only binary quadratic models defined on a specific graph.

Subgraph  See subgraph on wikipedia.

Target

Target graph  *Embedding* attempts to create a target model from a target graph. The process of embedding takes a source model, derives the source graph, maps the source graph to the target graph, then derives the target model. Sometimes referred to as the embedded graph/model. See a fuller description under Minor-Embedding.

Working graph  In a D-Wave QPU, the set of qubits and couplers that are available for computation is known as the working graph. The yield of a working graph is typically less than 100% of qubits and couplers that are fabricated and physically present in the QPU. See hardware graph.

![Diagram](image)

Fig. 5: Three-variable $K_3$ fully-connected problem is embedded in four qubits on a D-Wave 2000Q using the default chain strength. Variable $s_0$, highlighted in dark magenta, is represented by two qubits, numbers 0 and 4. The displayed solution has a broken chain: qubit 0 returned a value of $-1$ (represented by a white dot) while qubit 4 returned a value of $+1$ (a blue dot). The logical representation of the problem, on the left, shows a half-white, half-blue dot to represent a value based on a broken chain.
The goal of this document is to establish a common understanding among software contributors to D-Wave’s Ocean software projects based on the code conventions and best practices used at D-Wave.

This document is intended to be a living document, and is not complete. Feedback is welcome.

3.1 Testing

A feature or bugfix is complete only when the code has been unit-tested. Part of the pull-request process is to test your branch against master, and so the more tests you provide, the easier the pull-request process.

3.2 Submitting Changes

When contributing to a D-Wave project, fork the repository using the Github fork button and work in a feature branch in your own repository. In your feature branch, commit and push often, you can always rebase locally to edit your commit history and make it readable. To start a discussion, initiate a pull request. The sooner you start the pull request, the better. This allows early discussion on your feature and saves time and effort when you are ready to merge. When you are ready to merge, notify the repository owner and they will merge your code in.

Follow the commit conventions described here:

- https://chris.beams.io/posts/git-commit/

TL;DR:

- Separate subject from body with a blank line
- Limit the subject line to 50 characters
- Capitalize the subject line
- Do not end the subject line with a period
• Use the imperative mood in the subject line
• Wrap the body at 72 characters
• Use the body to explain what and why vs. how

If your branch is long-lived, rebase off of master periodically.

If you’re already familiar with Git, but rebasing is still scary, try this think like a Git guide.

The master branch in a dwavesystems repository reflects the bleeding-edge developments of the project. Significant old versions forked from master are kept in version branches. For example, we might keep version branches 2.x and 3.x while master tracks the latest 4.x version. We try to minimize the number of version branches kept alive/maintained.

Stable releases are tracked using tags. These tagged snapshots are deployed to PyPI after successfully passing the test cases during continuous integration.

### 3.3 Coding Conventions

• Variable naming should follow the well-known conventions of a language. Avoid uninformative or needlessly terse variable names.

• Code is read more often than written.

• Functions should do one thing.

• Early pull requests and code reviews.

• Early architecting/design. Code reviews can happen before any code has been written.

• Use a consistent character width of 120.

• Use 4 spaces instead of tabs.

• End all files with a newline.

### 3.3.1 Documentation and Comments

• Do a good job of commenting. Read this Coding Horror article.

• Comments should add, not repeat: avoid repeating in English or pseudo-code what the code does. Rather, discuss what the block is trying to achieve.

• Side effects should be visible on screen; if not in code, then in comments.

• Remember, the best documentation is clean, simple code with good variable names. When this is not possible, you must use a comment to explain the purpose of a functional block.

Example:

```python
# z must not be greater than 255.
if z > 255:
    raise RuntimeError('z must be <= 255!')
```

would be much more informative as
or even better:

```python
# See https://url.to.issue.tracker/IS-42
if z > 255:
    raise RuntimeError('z cannot be greater than 255, or this universe will collapse.')
```

### 3.3.2 Python

**pep8**

As a baseline, follow the pep8 style guide for python.

**Python 2/3**

All code should be both Python 2 and 3 compatible.

**Documentation**

- Google docstrings convention ([definition, example](https://url.to.issue.tracker/IS-42)) on all public-facing functions. The following are exceptions:
  - For D-Wave extensions of third-party projects, we match the existing convention (e.g. the D-Wave NetworkX project follows NumPy conventions).
  - Argument defaults are written “default=x” rather than “default x”.
- Private functions should include some sort of docstring.
- If your module has more than one public unit, it should have a module docstring with a table of contents.
- The docstring for the `__init__` method goes on the class.
- All docstrings should be parsable by the Sphinx documentation generation tool (i.e. reStructuredText) The sphinx theme should be [readthedocs](https://url.to.issue.tracker/IS-42).

### 3.3.3 C++

**.clang-format**

- When starting a new C++ project, copy the `.clang-format` file included here.
- Our style is based on Google (as opposed to LLVM, Chromium, Mozilla, or Webkit) with minor differences.
- ColumnLimit is set to 120, as specified in Coding Conventions.
- NamespaceIndentation is set to Inner as a middle ground between None (Google) and All, such that every line in a file defining a namespace isn’t indented, but nested namespaces are easily spotted.
- Various indent-width specifiers are scaled by a factor of 2 such that the base indent is 4, as specified in Coding Conventions, instead of 2 (Google). This is especially helpful for readability in cases like...
if (condition) {
    foo();
} else {
    bar();
}

as opposed to

if (condition) {
    foo();
} else {
    bar();
}

### Additional Style

Favor the use of the optional braces for single-line control statements, which enhance consistency and extensibility.

**Example:**

Use the following format

```java
if (a) {
    return;
}
```

as opposed to

```java
if (a) return;
```

This could potentially be enforced by `clang-tidy`.

### 3.3.4 Versioning Scheme

Our code follows Semantic Versioning conventions: major.minor.patch.

A change that breaks backwards compatibility must increment the major version. Anything below version 1.0.0 can break backwards compatibility.

### 3.3.5 Readme File

If you are creating a repository, don’t forget to include a `README.rst` containing a reasonable description of your project.
Below are the open-source licenses for the SDK and its tools:

4.1 dwave-ocean-sdk

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.
“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational
purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets “[ ]” replaced with your own identifying information. (Don’t include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same “printed page” as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

4.2 dwavebinarycsp

Apache License Version 2.0, January 2004 http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.
2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without
limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets “[]” replaced with your own identifying information. (Don’t include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same “printed page” as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

4.3 dwave-cloud-client

Apache License
Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.
“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in
any medium, with or without modifications, and in Source or Object form, provided that You meet
the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
(b) You must cause any modified files to carry prominent notices stating that You changed the files;
and
(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright,
patent, trademark, and attribution notices from the Source form of the Work, excluding those
notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works
that You distribute must include a readable copy of the attribution notices contained within
such NOTICE file, excluding those notices that do not pertain to any part of the Derivative
Works, in at least one of the following places: within a NOTICE text file distributed as part
of the Derivative Works; within the Source form or documentation, if provided along with the
Derivative Works; or, within a display generated by the Derivative Works, if and wherever such
third-party notices normally appear. The contents of the NOTICE file are for informational
purposes only and do not modify the License. You may add Your own attribution notices within
Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from
the Work, provided that such additional attribution notices cannot be construed as modifying
the License.

You may add Your own copyright statement to Your modifications and may provide additional or
different license terms and conditions for use, reproduction, or distribution of Your modifications,
or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of
the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally
submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions. Notwithstanding the above, nothing herein
shall supersede or modify the terms of any separate license agreement you may have executed with
Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service
marks, or product names of the Licensor, except as required for reasonable and customary use in
describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor pro-
vides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT
WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without
limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABIL-
ITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining
the appropriateness of using or redistributing the Work and assume any risks associated with Your
exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence),
contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent
acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any
direct, indirect, special, incidental, or consequential damages of any character arising as a result of
this License or out of the use or inability to use the Work (including but not limited to damages for
loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial
damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works
thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity,
or other liability obligations and/or rights consistent with this License. However, in accepting such
obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf
of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor
harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your
accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the
fields enclosed by brackets “[]” replaced with your own identifying information. (Don’t include
the brackets!) The text should be enclosed in the appropriate comment syntax for the file
format. We also recommend that a file or class name and description of purpose be included
on the same “printed page” as the copyright notice for easier identification within third-party
archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in
compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is dis-
tributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
express or implied. See the License for the specific language governing permissions and limitations under
the License.

4.4 dwave-hybrid

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by
Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is grant-
ing the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are con-
trolled by, or are under common control with that entity. For the purposes of this definition, “control”
means (i) the power, direct or indirect, to cause the direction or management of such entity, whether
by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares,
or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this
License.

“Source” form shall mean the preferred form for making modifications, including but not limited to
software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a
Source form, including but not limited to compiled object code, generated documentation, and con-
versions to other media types.
“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational
purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets “[ ]” replaced with your own identifying information. (Don’t include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same “printed page” as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at
Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

4.5 dwave-neal

Apache License
Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the
purpose of discussing and improving the Work, but excluding communication that is conspicuously
marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contri-
bution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor
hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform,
sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor
hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable
(except as stated in this section) patent license to make, have made, use, offer to sell, sell, import,
and otherwise transfer the Work, where such license applies only to those patent claims licensable
by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination
of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute
patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging
that the Work or a Contribution incorporated within the Work constitutes direct or contributory
patent infringement, then any patent licenses granted to You under this License for that Work shall
terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in
any medium, with or without modifications, and in Source or Object form, provided that You meet
the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files;

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright,
patent, trademark, and attribution notices from the Source form of the Work, excluding those
notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works
that You distribute must include a readable copy of the attribution notices contained within
such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in
at least one of the following places: within a NOTICE text file distributed as part of
the Derivative Works; within the Source form or documentation, if provided along with the
Derivative Works; or, within a display generated by the Derivative Works, if and wherever such
third-party notices normally appear. The contents of the NOTICE file are for informational
purposes only and do not modify the License. You may add Your own attribution notices within
Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from
the Work, provided that such additional attribution notices cannot be construed as modifying
the License.

You may add Your own copyright statement to Your modifications and may provide additional or
different license terms and conditions for use, reproduction, or distribution of Your modifications,
or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of
the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally
submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions. Notwithstanding the above, nothing herein
shall supersede or modify the terms of any separate license agreement you may have executed with
Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service
marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets “[]” replaced with your own identifying information. (Don’t include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same “printed page” as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

4.6 dwave-networkx

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION
1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory
patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

4.7 dwave-system

Apache License Version 2.0, January 2004 http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose
of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.
5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets “[]” replaced with your own identifying information. (Don’t include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same “printed page” as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF
ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

4.8 dwave-tabu

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
(b) You must cause any modified files to carry prominent notices stating that You changed the files; and
(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided You use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may
choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or
rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and
on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and
hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of
your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets
“[]” replaced with your own identifying information. (Don’t include the brackets!) The text should be enclosed in
the appropriate comment syntax for the file format. We also recommend that a file or class name and description of
purpose be included on the same “printed page” as the copyright notice for easier identification within third-party
archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an
“AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
License for the specific language governing permissions and limitations under the License.

4.9 minorminer

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by
Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is grant-
ing the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are con-
trolled by, or are under common control with that entity. For the purposes of this definition, “control”
means (i) the power, direct or indirect, to cause the direction or management of such entity, whether
by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares,
or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this
License.

“Source” form shall mean the preferred form for making modifications, including but not limited to
software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a
Source form, including but not limited to compiled object code, generated documentation, and con-
versions to other media types.
“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational
purposes only and do not modify the License. You may add Your own attribution notices within
Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from
the Work, provided that such additional attribution notices cannot be construed as modifying
the License.

You may add Your own copyright statement to Your modifications and may provide additional or
different license terms and conditions for use, reproduction, or distribution of Your modifications,
or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of
the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally
submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions. Notwithstanding the above, nothing herein
shall supersede or modify the terms of any separate license agreement you may have executed with
Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service
marks, or product names of the Licensor, except as required for reasonable and customary use in
describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor pro-
vides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT
WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without
limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY,
or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining
the appropriateness of using or redistributing the Work and assume any risks associated with Your
exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence),
contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent
acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any
direct, indirect, special, incidental, or consequential damages of any character arising as a result of
this License or out of the use or inability to use the Work (including but not limited to damages for
loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial
damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works
thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity,
or other liability obligations and/or rights consistent with this License. However, in accepting such
obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf
of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor
harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your
accepting any such warranty or additional liability.

4.10 penaltrymodel

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by
Sections 1 through 9 of this document.
“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in
any medium, with or without modifications, and in Source or Object form, provided that You meet
the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
(b) You must cause any modified files to carry prominent notices stating that You changed the files;
and
(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright,
patent, trademark, and attribution notices from the Source form of the Work, excluding those
notices that do not pertain to any part of the Derivative Works; and
(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works
that You distribute must include a readable copy of the attribution notices contained within
such NOTICE file, excluding those notices that do not pertain to any part of the Derivative
Works, in at least one of the following places: within a NOTICE text file distributed as part
of the Derivative Works; within the Source form or documentation, if provided along with the
Derivative Works; or, within a display generated by the Derivative Works, if and wherever such
third-party notices normally appear. The contents of the NOTICE file are for informational
purposes only and do not modify the License. You may add Your own attribution notices within
Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from
the Work, provided that such additional attribution notices cannot be construed as modifying
the License.

You may add Your own copyright statement to Your modifications and may provide additional or
different license terms and conditions for use, reproduction, or distribution of Your modifications,
or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of
the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally
submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions
of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein
shall supersede or modify the terms of any separate license agreement you may have executed with
Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service
marks, or product names of the Licensor, except as required for reasonable and customary use in
describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor pro-
vides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT
WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without
limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABIL-
ITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining
the appropriateness of using or redistributing the Work and assume any risks associated with Your
exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence),
contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent
acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any
direct, indirect, special, incidental, or consequential damages of any character arising as a result of
this License or out of the use or inability to use the Work (including but not limited to damages for
loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial
damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works
thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity,
or other liability obligations and/or rights consistent with this License. However, in accepting such
obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf
of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

Ocean software also includes non-open-source packages you can optionally install with the following D-Wave end-user license agreements (EULAs):

### 4.11 dwave-drivers

COPYRIGHT 2017 D-WAVE SYSTEMS INC. All Rights Reserved.

D-WAVE SYSTEMS INC. (“D-WAVE”) END-USER LICENSE AGREEMENT (“EULA”)

The software subject to this EULA is/are known as “dwave_system_tuning”.

IMPORTANT - READ CAREFULLY BEFORE COPYING, INSTALLING OR OTHERWISE USING; THIS EULA IS A LEGAL BINDING CONTRACT BETWEEN YOU AND D-WAVE. DO NOT COPY, INSTALL, OR USE THIS SOFTWARE UNTIL YOU HAVE CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS. BY USING, COPYING OR DISTRIBUTING ALL OR ANY PORTION OF THE SOFTWARE YOU ACCEPT ALL THE TERMS AND CONDITIONS OF THIS EULA AND YOU AGREE THAT THIS EULA IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU. THIS EULA IS ENFORCEABLE AGAINST YOU AND ANY LEGAL ENTITY THAT OBTAINED THE SOFTWARE AND ON WHOSE BEHALF IT IS USED, FOR EXAMPLE, YOUR EMPLOYER. IF YOU DO NOT AGREE, DO NOT INSTALL OR USE THIS SOFTWARE.

IF YOU OR THE LEGAL ENTITY ON WHOSE BEHALF THE SOFTWARE WAS OBTAINED HAVE A SEPARATE CONTRACT WITH D-WAVE (“ALTERNATE CONTRACT”), ADDITIONAL OR DIFFERENT TERMS MAY APPLY TO YOUR USE OF THIS SOFTWARE. IN THE EVENT OF A CONFLICT BETWEEN THE TERMS OF THIS EULA AND THE ALTERNATE CONTRACT, THE TERMS OF THE ALTERNATE CONTRACT APPLY.

US GOVERNMENT USERS: The Software and related documentation qualify as “commercial items,” as that term is defined at Federal Acquisition Regulation (“FAR”) (48 CFR 2.101, consisting of “commercial computer software” and “commercial computer software documentation” as such terms are used in FAR 12.212. Consistent with FAR 12.212 and DoD FAR Supp. 227.7202-1 through 227.7202-4, and notwithstanding any other FAR or other contractual clause to the contrary in any agreement into which this EULA may be incorporated, Government end users acquire the Software and related documentation with only those rights set forth in the EULA. Use of either the Software or the related documentation or both constitutes agreement by the Government that the Software and Documentation are “commercial computer software” and “commercial computer software documentation,” and constitutes acceptance of the rights and restrictions herein.

D-WAVE AND ITS SUPPLIERS OWN ALL INTELLECTUAL PROPERTY IN THIS SOFTWARE. D-WAVE PERMITS YOU TO COPY, INSTALL, USE AND DISTRIBUTE THE SOFTWARE ONLY IN ACCORDANCE WITH THE TERMS OF THIS EULA. USE OF SOME THIRD PARTY MATERIALS ASSOCIATED WITH THE SOFTWARE MAY BE SUBJECT TO OTHER TERMS AND CONDITIONS TYPICALLY FOUND IN A SEPARATE LICENSE AGREEMENT OR A “READ ME” OR “LICENCE.TXT” FILE PROVIDED IN ASSOCIATION WITH SUCH MATERIALS.

BY INSTALLING OR OTHERWISE USING THE SOFTWARE, YOU (THE “LICENSEE”) AGREE TO BE BOUND BY THE TERMS OF THIS EULA.

This EULA applies to the evaluation of the attached software product by Licensee during the evaluation period, which is without charge.

In this EULA, “Software” means (a) the object and/or source code version of the computer programs identified above as being subject to this EULA; (b) the object and/or source code version of any computer program related to the software of clause (a); (c) all of the contents of the files (provided either by electronic download, on physical media or any other method of distribution), disk(s), CD-ROM(s) or other media with which this EULA is provided; (d) related explanatory written materials, user manuals, electronic files and other documentation accompanying or published for
such program or files (“Documentation”); (e) upgrades, updates, additions to, and copies of the foregoing, if any, provided to you by D-WAVE or an authorized licensee, for example, downloaded software updates (collectively, “Updates”); and (f) any copies of the foregoing.

“D-WAVE” means D-Wave Systems Inc., a Canadian corporation, 3033 Beta Avenue, Burnaby, British Columbia, Canada.

1. SOFTWARE LICENSE.

a. In order to use the Software, Licensee must first agree to this EULA. Licensee may not use the Software if Licensee does not accept this EULA.

b. If Licensee is agreeing to be bound by this EULA on behalf of Licensee’s employer or other entity, Licensee represents and warrants that Licensee has full legal authority to bind Licensee’s employer or such entity to this EULA. If Licensee does not have the requisite authority, Licensee may not accept the EULA or use the Software on behalf of Licensee’s employer or other entity.

c. Licensee can accept this EULA by: i. clicking to accept or agree to the EULA before downloading the Software where this option is made available to Licensee; or ii. using the Software. d. Upon Licensee’s agreement to be bound by the terms of this EULA and subject to the terms of this EULA and Licensee’s compliance with same, D-WAVE grants to Licensee a limited, non-exclusive, non-transferable, non-sublicensable copyright license to use the Software solely to develop applications to run on or with D-WAVE’s product(s) for (i) non-commercial use, or (ii) Licensee’s internal product development purposes.

e. No Support. D-WAVE is not obligated to provide any technical or other support for the Software to Licensee.

f. Fees. D-WAVE may choose in the future to charge for use of the Software. If D-WAVE in its sole discretion chooses to establish fees and payment terms for such use, D-WAVE will provide notice of such terms as provided in Section 12 below, and Licensee may elect to stop using the Software rather than incurring fees.

2. USE RESTRICTIONS. At all times during the term of this EULA:

a. Permitted Users. Licensee may: (i) if an individual, himself or herself use the Software as permitted by this EULA; or (ii) if a corporation or other entity, allow only its then current employees to use the Software as permitted by this EULA; but may not allow any other persons to use the Software.

b. Permitted Computers. Licensee may install the Software and make the Software available for use only on computers owned or leased by Licensee which are located at the premises of Licensee. Licensee may make one backup copy of the Software, provided that the backup copy is not installed or used on any computer. Any copy of the Software made by Licensee must contain the same copyright and other proprietary notices that appear on or in the Software and may not be provided to any other person.

c. Other Restrictions. Except to the extent required by applicable third party licenses, Licensee may not:

i. use the Software except in accordance with this EULA; ii. copy, modify, adapt, translate or create any derivative works of or based upon the Software; iii. decompile, disassemble, reverse engineer, or otherwise attempt to discover or derive the source code for the Software; iv. load any part of the Software onto a mobile handset or any other hardware device except a personal computer, combine any part of the Software with other software, or distribute any software or device incorporating a part of the Software; v. distribute, sell, rent, lease, sublicense, assign or otherwise transfer this EULA or its rights to the Software; or vi. remove or alter any trademark, logo, copyright or other proprietary notices, legends, symbols or labels on the Software.

d. Use, reproduction and distribution of components of the Software licensed under a third party license are governed solely by the terms of that third party license and not this EULA.

e. Compliance with Laws. Licensee will comply with all export laws and regulations of Canada, the United States, or any other government having jurisdiction and agrees that it will not export or re-export the Software in violation of any such laws or regulations. Licensee will also comply with all applicable laws and regulations with respect to the import into any country or the use in any country of the Software. Licensee agrees to use the Software and write applications only for purposes that are permitted by this EULA and any applicable law, regulation or generally accepted practices or guidelines in the relevant jurisdictions (including any laws regarding the export of data or software to and from
Canada, the United States or other relevant countries). In addition, if the Software is identified as export controlled items under any applicable law, Licensee represents and warrants that it is not a citizen of, or otherwise located within, a nation sanctioned or embargoed by Canada, the United Nations, or the United States (including without limitation Belarus, Cote d’Ivoire, Cuba, Democratic Republic of the Congo, Egypt, Eritrea, Iran, Iraq, Lebanon, Liberia, Libya, Myanmar (Burma), North Korea, Sierra Leone, Somalia, Sudan, Syria, Tunisia, and Zimbabwe) and that it is not otherwise prohibited by law from receiving the Software. Licensee agrees not to use the Software for any military or defense-related purpose.

f. Licensee acknowledges and agrees that the form and nature of the Software that D-WAVE provides may change without prior notice to Licensee and that future versions of the Software may be incompatible with applications developed on previous versions of the Software. Licensee acknowledges and agrees that D-WAVE may stop (permanently or temporarily) providing the Software (or any features within the Software) to Licensee or to users generally at D-WAVE’s sole discretion, without prior notice to Licensee.

3. CONSENT TO USE OF DATA AND USER SUBMISSIONS. Licensee agrees that D-WAVE and its subsidiaries and agents may collect, maintain, process and use diagnostic, technical, usage and related information, including but not limited to information about Licensee’s computer, system and application software, and peripherals, that is gathered periodically to facilitate the provision of Updates, product support and other services to Licensee (if any) related to the Software, and to verify compliance with the terms of this EULA. D-WAVE may use this information in compliance with applicable privacy laws to provide and improve D-WAVE’s products and services or distribute the information to D-WAVE’s partners and third party developers to improve their software, hardware and services designed for use with D-WAVE products. Licensee agrees that any material, information or other communication, including all data, images, sounds, text, computer code in source or object form, and other things embodied therein, Licensee transmits or posts to any D-WAVE website or provides to D-WAVE under this EULA or through Licensee’s use of the Software will be deemed non-confidential ("Communications"). D-WAVE shall have no confidentiality obligations with respect to the Communications. Licensee agrees that D-WAVE will be free to copy, modify, create derivative works, publicly display, disclose, distribute, license and sublicense through multiple tiers of distribution and licensees, incorporate and otherwise use the Communications, including derivative works thereto, for any and all commercial or non-commercial purposes.

4. OWNERSHIP. Except for the limited license rights expressly granted to Licensee hereunder, D-WAVE and its suppliers retain all right, title and interest in the Software (including any copies that Licensee makes in accordance with this EULA), including any rights under the patent, trademark, copyright, trade secrets and other intellectual property laws, and this EULA does not grant to Licensee any intellectual property rights in the Software. D-WAVE reserves all rights not expressly granted to Licensee in this EULA. The structure, organization and code underlying the Software are the valuable rights of D-WAVE and its suppliers. Licensee agrees that Licensee will not take any action to jeopardize, limit or interfere in any manner with such ownership by D-WAVE and its suppliers.

5. THIRD PARTY SOFTWARE APPLICATIONS AND DATA SERVICES. Portions of the Software may utilize or include third party software and other copyrighted material ("Third Party Software"). Acknowledgements, licensing terms and disclaimers for such Third Party Software are contained in the Documentation, and Licensee’s use of such Third Party Software is governed by such acknowledgements, licensing terms and disclaimers. The Software may interoperate with and allow Licensee to use software applications, information and data not developed or offered by D-WAVE ("Third Party Information"). Licensee’s use of any Third Party Software or Third Party Information is governed by the terms and conditions made available to Licensee by the party from whom Licensee obtained such information. Neither Third Party Software nor Third Party Information is the responsibility of D-WAVE, and LICENSEE’S USE OF THIRD PARTY SOFTWARE OR THIRD PARTY INFORMATION IS AT LICENSEE’S OWN RISK. D-WAVE MAKES NO WARRANTIES, CONDITIONS, INDEMNITIES, REPRESENTATIONS OR TERMS, EXPRESS OR IMPLIED, WHETHER BY STATUTE, COMMON LAW, CUSTOM, USAGE OR OTHERWISE AS TO ANY MATTERS, INCLUDING BUT NOT LIMITED TO NON-INFRINGEMENT OF THIRD PARTY RIGHTS, TITLE, INTEGRATION, ACCURACY, SECURITY, AVAILABILITY, SATISFACTORY QUALITY, MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE WITH RESPECT TO THIRD PARTY SOFTWARE OR THIRD PARTY INFORMATION.

6. NO WARRANTY. THE SOFTWARE IS MADE AVAILABLE TO THE LICENSEE ON AN “AS-IS” BASIS AND WITH ALL FAULTS. D-WAVE MAKES NO WARRANTY OR REPRESENTATION AS TO THE USE OR
PERFORMANCE OF THE SOFTWARE OF ANY KIND. EXCEPT TO THE EXTENT TO WHICH THE SAME MAY NOT BE EXCLUDED AND/OR LIMITED UNDER APPLICABLE LAW, D-WAVE DOES NOT WARRANT THAT THE SOFTWARE IS FREE OF DEFECTS, MERCHANTABILITY, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY RIGHTS, AND DISCLAIMS AND EXCLUDES ALL WARRANTIES, EXPRESS OR IMPLIED (WHETHER BY STATUTE, COMMON LAW, CUSTOM, USAGE OR OTHERWISE), WITH RESPECT TO THE SOFTWARE, ITS USE, PERFORMANCE OR APPLICATION, OR ANY INFORMATION PROVIDED REGARDING THE SOFTWARE. D-WAVE PROVIDES NO TECHNICAL SUPPORT OR REMEDIES FOR THE SOFTWARE.

7. LIMITATION OF LIABILITY. D-WAVE WILL NOT BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE POSSESSION OF, USE OF OR INABILITY TO USE THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, LOST PROFITS, LOSS OF GOODWILL, WORK STOPPAGE, DATA LOSS, OR COMPUTER FAILURE OR MALFUNCTION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND REGARDLESS OF WHETHER THE CLAIM OR LIABILITY IS BASED UPON ANY CONTRACT, TORT, BREACH OF WARRANTY OR OTHER LEGAL OR EQUITABLE THEORY AND NOTWITHSTANDING THAT ANY REMEDY HEREIN FAILS OF ITS ESSENTIAL PURPOSE. THE MAXIMUM LIABILITY OF D-WAVE TO LICENSEE SHALL IN ANY EVENT NOT EXCEED THE LESSER OF (A) THE SUM OF THE LICENSE FEE PAID BY LICENSEE FOR THE SOFTWARE; OR (B) CAD$250. THE FOREGOING LIMITATIONS WILL APPLY EVEN IF THE ABOVE STATED REMEDY FAILS OF ITS ESSENTIAL PURPOSE. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, AND PARAGRAPHS 3 AND 4 WILL NOT APPLY ONLY TO THE EXTENT THAT APPLICABLE LAW REQUIRES LIABILITY NOTWITHSTANDING THE LIMITATIONS OR EXCLUSIONS THEREIN.

8. LICENSEE ACKNOWLEDGES AND AGREES THAT THE SOFTWARE IS NOT DESIGNED, INTENDED, AUTHORIZED OR SUITABLE FOR USE IN ANY TYPE OF SYSTEM, APPLICATION, SITUATION OR ENVIRONMENT WHERE THE FAILURE OR TIME DELAYS OF, OR ERRORS OR INACCURACIES IN, THE CONTENT, DATA OR INFORMATION PROVIDED BY THE SOFTWARE COULD LEAD TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE, INCLUDING WITHOUT LIMITATION THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, LIFE SUPPORT, MEDICAL SYSTEMS OR WEAPONS SYSTEMS AND THAT LICENSEE WILL NOT USE THE SOFTWARE IN SUCH SYSTEM, APPLICATION, SITUATION OR ENVIRONMENT.

9. INDEMNIFICATION. To the maximum extent permitted by law, Licensee agrees to defend, indemnify and hold harmless D-WAVE, its affiliates and their respective directors, officers, employees and agents from and against any and all claims, actions, suits or proceedings, as well as any and all losses, liabilities, damages, costs and expenses (including reasonable attorneys fees) arising out of or accruing from (a) Licensee’s use of the Software, (b) any application Licensee develops using the Software that infringes or violates any right of any person or defames any person, and (c) any non-compliance by Licensee with this EULA.

10. TERMINATION.

a. This EULA is effective from the date on which the attached Software is installed or used by Licensee until terminated. This EULA will terminate automatically without notice from D-WAVE if Licensee fails to comply with any provision of this EULA or if D-WAVE is required to do so by law. If Licensee commences or participates in any legal proceeding against D-WAVE, then D-WAVE may, in its sole discretion, suspend or terminate this EULA during the pendency of such legal proceedings. Licensee may voluntarily terminate this EULA at any time by ceasing to use the Software.

b. Upon termination of this EULA at any time and for any reason, Licensee will immediately discontinue any and all use of the Software, purge the Software from all computer systems, storage media and other files, including any back-up copy, and return to D-WAVE the Software, including the Documentation, and all copies thereof, or at the request of D-WAVE, destroy the Software, the Documentation, and all copies thereof, and deliver to D-WAVE certification that Licensee has complied with these termination obligations.

c. If this EULA is terminated at any time and for any reason, Licensee will not be entitled to compensation of any
kind, financial or otherwise.

d. Paragraphs 3 through 9, 13(a) and 13(f) of this EULA and such other provisions as may reasonably be expected to remain in force will survive the termination of this EULA and will remain in full force and effect following such termination.

11. NO WAIVERS. A waiver expressed or implied by a party of any default of the other party in the observance or performance of this EULA must be in writing, and such waiver does not constitute and is not to be construed as a waiver or condonation of any subsequent or other default, and a waiver of any provision of this EULA will not be effective unless in writing and signed by an authorized officer of D-WAVE.

12. MODIFICATIONS AND NOTICES. No reseller, distributor or dealer of the Software is authorized to make any modifications, extensions, or additions to this EULA. D-WAVE may, in its sole discretion, modify this EULA. If D-WAVE elects to do so, D-WAVE will notify Licensee of such change. If Licensee does not agree to the change, then this EULA automatically terminates and Licensee must stop using the Software and comply with Section 10(b) of this EULA. If Licensee does not stop using the Software, then Licensee’s use of the Software will continue under the changed EULA. D-WAVE may give notices to Licensee, at D-WAVE’s option, by posting on any portion of www.dwavesys.com or by electronic mail to any e-mail address provided by Licensee to D-WAVE.

13. GOVERNING LAW AND GENERAL PROVISIONS.

a. Governing Law and Other General Provisions. This EULA will be governed by and construed in accordance with the substantive laws in force in the Province of British Columbia, Canada, excluding its conflict of law rules. The courts of the Province of British Columbia shall have exclusive jurisdiction over all disputes relating to this EULA. Licensee and D-WAVE agree to submit to the exclusive jurisdiction of the courts of British Columbia. This EULA will not be governed by the conflict of law rules of any jurisdiction or the United Nations Convention on Contracts for the International Sale of Goods, the application of which is expressly excluded.

b. Time. Time is of the essence of this EULA.

c. Severance. If any provision in this EULA should be held invalid or unenforceable, such provision shall be modified to the extent necessary to render it valid or enforceable or severed from this EULA if no such modification is possible, and the other provisions of this EULA shall remain in full force and effect.

d. Assignment. Licensee may not assign its rights or delegate its obligations under this EULA without the prior written consent of D-WAVE, which consent may be withheld in the sole discretion of D-WAVE. D-WAVE may assign its rights or delegate its obligations under this EULA to any person in its sole discretion. This EULA shall be binding on and inure to the benefit of the parties and their successors and permitted assigns.

e. No Other EULAs. This EULA is the entire agreement of the parties with respect to the Software, and there are no other agreements, written, oral, electronic or otherwise, with respect to the Software, and it supersedes any prior representations, discussions, undertakings, communications or advertising relating to the Software.

f. Compliance with Licenses. If Licensee is a business or organization, Licensee agrees that upon request from D-WAVE or D-WAVE’s authorized representative, Licensee will within thirty (30) days fully document and certify that use of any and all Software at the time of the request is in conformity with Licensee’s valid licenses from D-WAVE.

g. English Language. Licensee and D-WAVE confirm that it is their wish that this EULA, as well as other documents relating hereto, including notices, have been and shall be written in English only. Licensee et D-WAVE confirment leur désir que cet accord ainsi que tous les documents, y compris tous avis qui s’y rattachent, soient rediges en anglais.

Updated January 4, 2017
The software distributed with this EULA is subject to the following terms.

**IMPORTANT - READ CAREFULLY BEFORE COPYING, INSTALLING OR OTHERWISE USING: THIS EULA IS A LEGAL BINDING CONTRACT BETWEEN YOU AND D-WAVE. DO NOT COPY, INSTALL, OR USE THIS SOFTWARE UNTIL YOU HAVE CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS. BY USING, COPYING OR DISTRIBUTING ALL OR ANY PORTION OF THE SOFTWARE YOU ACCEPT ALL THE TERMS AND CONDITIONS OF THIS EULA AND YOU AGREE THAT THIS EULA IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU. THIS EULA IS ENFORCEABLE AGAINST YOU AND ANY LEGAL ENTITY THAT OBTAINED THE SOFTWARE AND ON WHOSE BEHALF IT IS USED, FOR EXAMPLE, YOUR EMPLOYER. IF YOU DO NOT AGREE, DO NOT INSTALL OR USE THIS SOFTWARE.**

**IF YOU OR THE LEGAL ENTITY ON WHOSE BEHALF THE SOFTWARE WAS OBTAINED HAVE A SEPARATE CONTRACT WITH D-WAVE (“ALTERNATE CONTRACT”), ADDITIONAL OR DIFFERENT TERMS MAY APPLY TO YOUR USE OF THIS SOFTWARE. IN THE EVENT OF A CONFLICT BETWEEN THE TERMS OF THIS EULA AND THE ALTERNATE CONTRACT, THE TERMS OF THE ALTERNATE CONTRACT APPLY.**

US GOVERNMENT USERS: The Software and related documentation qualify as “commercial items,” as that term is defined at Federal Acquisition Regulation (“FAR”) (48 CFR 2.101, consisting of “commercial computer software” and “commercial computer software documentation” as such terms are used in FAR 12.212. Consistent with FAR 12.212 and DoD FAR Supp. 227.7202-1 through 227.7202-4, and notwithstanding any other FAR or other contractual clause to the contrary in any agreement into which this EULA may be incorporated, Government end users acquire the Software and related documentation with only those rights set forth in the EULA. Use of either the Software or the related documentation or both constitutes agreement by the Government that the Software and Documentation are “commercial computer software” and “commercial computer software documentation,” and constitutes acceptance of the rights and restrictions herein.

D-WAVE AND ITS SUPPLIERS OWN ALL INTELLECTUAL PROPERTY IN THIS SOFTWARE. D-WAVE PERMITS YOU TO COPY, INSTALL, USE AND DISTRIBUTE THE SOFTWARE ONLY IN ACCORDANCE WITH THE TERMS OF THIS EULA. USE OF SOME THIRD PARTY MATERIALS ASSOCIATED WITH THE SOFTWARE MAY BE SUBJECT TO OTHER TERMS AND CONDITIONS TYPICALLY FOUND IN A SEPARATE LICENSE AGREEMENT OR A “READ ME” OR “LICENCE.TXT” FILE PROVIDED IN ASSOCIATION WITH SUCH MATERIALS.

**BY INSTALLING OR OTHERWISE USING THE SOFTWARE, YOU (THE “LICENSEE”) AGREE TO BE BOUND BY THE TERMS OF THIS EULA.**

This EULA applies to the evaluation of the attached software product by Licensee during the evaluation period, which is without charge.

In this EULA, “Software” means (a) the object and/or source code version of the computer programs identified above as being subject to this EULA; (b) the object and/or source code version of any computer program related to the software of clause (a); (c) all of the contents of the files (provided either by electronic download, on physical media or any other method of distribution), disk(s), CD-ROM(s) or other media with which this EULA is provided; (d) related explanatory written materials, user manuals, electronic files and other documentation accompanying or published for such program or files (“Documentation”); (e) upgrades, updates, additions to, and copies of the foregoing, if any, provided to you by D-WAVE or an authorized licensee, for example, downloaded software updates (collectively, “Updates”); and (f) any copies of the foregoing.

“D-WAVE” means D-Wave Systems Inc., a Canadian corporation, 3033 Beta Avenue, Burnaby, British Columbia, Canada.

1. SOFTWARE LICENSE.

a. In order to use the Software, Licensee must first agree to this EULA. Licensee may not use the Software if Licensee does not accept this EULA.

b. If Licensee is agreeing to be bound by this EULA on behalf of Licensee’s employer or other entity, Licensee represents and warrants that Licensee has full legal authority to bind Licensee’s employer or such entity to this EULA.
If Licensee does not have the requisite authority, Licensee may not accept the EULA or use the Software on behalf of Licensee’s employer or other entity.

c. Licensee can accept this EULA by: i. clicking to accept or agree to the EULA before downloading the Software where this option is made available to Licensee; or ii. using the Software. d. Upon Licensee’s agreement to be bound by the terms of this EULA and subject to the terms of this EULA and Licensee’s compliance with same, D-WAVE grants to Licensee a limited, non-exclusive, non-transferable, non-sublicensable copyright license to use the Software solely to develop applications to run on or with D-WAVE’s product(s) for (i) non-commercial use, or (ii) Licensee’s internal product development purposes.

e. No Support. D-WAVE is not obligated to provide any technical or other support for the Software to Licensee.

f. Fees. D-WAVE may choose in the future to charge for use of the Software. If D-WAVE in its sole discretion chooses to establish fees and payment terms for such use, D-WAVE will provide notice of such terms as provided in Section 12 below, and Licensee may elect to stop using the Software rather than incurring fees.

2. USE RESTRICTIONS. At all times during the term of this EULA:

a. Permitted Users. Licensee may: (i) if an individual, himself or herself use the Software as permitted by this EULA; or (ii) if a corporation or other entity, allow only its then current employees to use the Software as permitted by this EULA; but may not allow any other persons to use the Software.

b. Permitted Computers. Licensee may install the Software and make the Software available for use only on computers owned or leased by Licensee which are located at the premises of Licensee. Licensee may make one backup copy of the Software, provided that the backup copy is not installed or used on any computer. Any copy of the Software made by Licensee must contain the same copyright and other proprietary notices that appear on or in the Software and may not be provided to any other person.

c. Other Restrictions. Except to the extent required by applicable third party licenses, Licensee may not:

i. use the Software except in accordance with this EULA; ii. copy, modify, adapt, translate or create any derivative works of or based upon the Software; iii. decompile, disassemble, reverse engineer, or otherwise attempt to discover or derive the source code for the Software; iv. load any part of the Software onto a mobile handset or any other hardware device except a personal computer, combine any part of the Software with other software, or distribute any software or device incorporating a part of the Software; v. distribute, sell, rent, lease, sublicense, assign or otherwise transfer this EULA or its rights to the Software; or vi. remove or alter any trademark, logo, copyright or other proprietary notices, legends, symbols or labels on the Software.

d. Use, reproduction and distribution of components of the Software licensed under a third party license are governed solely by the terms of that third party license and not this EULA.

e. Compliance with Laws. Licensee will comply with all export laws and regulations of Canada, the United States, or any other government having jurisdiction and agrees that it will not export or re-export the Software in violation of any such laws or regulations. Licensee will also comply with all applicable laws and regulations with respect to the import into any country or the use in any country of the Software. Licensee agrees to use the Software and write applications only for purposes that are permitted by this EULA and any applicable law, regulation or generally accepted practices or guidelines in the relevant jurisdictions (including any laws regarding the export of data or software to and from Canada, the United States or other relevant countries). In addition, if the Software is identified as export controlled items under any applicable law, Licensee represents and warrants that it is not a citizen of, or otherwise located within, a nation sanctioned or embargoed by Canada, the United Nations, or the United States (including without limitation Belarus, Cote d’Ivoire, Cuba, Democratic Republic of the Congo, Egypt, Eritrea, Iran, Iraq, Lebanon, Liberia, Libya, Myanmar (Burma), North Korea, Sierra Leone, Somalia, Sudan, Syria, Tunisia, and Zimbabwe) and that it is not otherwise prohibited by law from receiving the Software. Licensee agrees not to use the Software for any military or defense-related purpose.

f. Licensee acknowledges and agrees that the form and nature of the Software that D-WAVE provides may change without prior notice to Licensee and that future versions of the Software may be incompatible with applications developed on previous versions of the Software. Licensee acknowledges and agrees that D-WAVE may stop (permanently
or temporarily) providing the Software (or any features within the Software) to Licensee or to users generally at D-WAVE’s sole discretion, without prior notice to Licensee.

3. CONSENT TO USE OF DATA AND USER SUBMISSIONS. Licensee agrees that D-WAVE and its subsidiaries and agents may collect, maintain, process and use diagnostic, technical, usage and related information, including but not limited to information about Licensee’s computer, system and application software, and peripherals, that is gathered periodically to facilitate the provision of Updates, product support and other services to Licensee (if any) related to the Software, and to verify compliance with the terms of this EULA. D-WAVE may use this information in compliance with applicable privacy laws to provide and improve D-WAVE’s products and services or distribute the information to D-WAVE’s partners and third party developers to improve their software, hardware and services designed for use with D-WAVE products. Licensee agrees that any material, information or other communication, including all data, images, sounds, text, computer code in source or object form, and other things embodied therein, Licensee transmits or posts to any D-WAVE website or provides to D-WAVE under this EULA or through Licensee’s use of the Software will be deemed non-confidential (“Communications”). D-WAVE shall have no confidentiality obligations with respect to the Communications. Licensee agrees that D-WAVE will be free to copy, modify, create derivative works, publicly display, disclose, distribute, license and sublicense through multiple tiers of distribution and licensees, incorporate and otherwise use the Communications, including derivative works thereto, for any and all commercial or non-commercial purposes.

4. OWNERSHIP. Except for the limited license rights expressly granted to Licensee hereunder, D-WAVE and its suppliers retain all right, title and interest in the Software (including any copies that Licensee makes in accordance with this EULA), including any rights under the patent, trademark, copyright, trade secrets and other intellectual property laws, and this EULA does not grant to Licensee any intellectual property rights in the Software. D-WAVE reserves all rights not expressly granted to Licensee in this EULA. The structure, organization and code underlying the Software are the valuable rights of D-WAVE and its suppliers. Licensee agrees that Licensee will not take any action to jeopardize, limit or interfere in any manner with such ownership by D-WAVE and its suppliers.

5. THIRD PARTY SOFTWARE APPLICATIONS AND DATA SERVICES. Portions of the Software may utilize or include third party software and other copyrighted material (“Third Party Software”). Acknowledgements, licensing terms and disclaimers for such Third Party Software are contained in the Documentation, and Licensee’s use of such Third Party Software is governed by such acknowledgements, licensing terms and disclaimers. The Software may interoperate with and allow Licensee to use software applications, information and data not developed or offered by D-WAVE (“Third Party Information”). Licensee’s use of any Third Party Software or Third Party Information is governed by the terms and conditions made available to Licensee by the party from whom Licensee obtained such information. Neither Third Party Software nor Third Party Information is the responsibility of D-WAVE, and LICENSEE’S USE OF THIRD PARTY SOFTWARE OR THIRD PARTY INFORMATION IS AT LICENSEE’S OWN RISK. D-WAVE MAKES NO WARRANTIES, CONDITIONS, INDEMNITIES, REPRESENTATIONS OR TERMS, EXPRESS OR IMPLIED, WHETHER BY STATUTE, COMMON LAW, CUSTOM, USAGE OR OTHERWISE AS TO ANY MATTERS, INCLUDING BUT NOT LIMITED TO NON-INFRINGEMENT OF THIRD PARTY RIGHTS, TITLE, INTEGRATION, ACCURACY, SECURITY, AVAILABILITY, SATISFACTORY QUALITY, MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE WITH RESPECT TO THIRD PARTY SOFTWARE OR THIRD PARTY INFORMATION.

6. NO WARRANTY. THE SOFTWARE IS MADE AVAILABLE TO THE LICENSEE ON AN “AS-IS” BASIS AND WITH ALL FAULTS. D-WAVE MAKES NO WARRANTY OR REPRESENTATION AS TO THE USE OR PERFORMANCE OF THE SOFTWARE OF ANY KIND. EXCEPT TO THE EXTENT TO WHICH THE SAME MAY NOT BE EXCLUDED AND/OR LIMITED UNDER APPLICABLE LAW, D-WAVE DOES NOT WARRANT THAT THE SOFTWARE IS FREE OF DEFECTS, MERCHANTABILITY, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY RIGHTS, AND DISCLAIMS AND EXCLUDES ALL WARRANTIES, EXPRESS OR IMPLIED (WHETHER BY STATUTE, COMMON LAW, CUSTOM, USAGE OR OTHERWISE), WITH RESPECT TO THE SOFTWARE, ITS USE, PERFORMANCE OR APPLICATION, OR ANY INFORMATION PROVIDED REGARDING THE SOFTWARE. D-WAVE PROVIDES NO TECHNICAL SUPPORT OR REMEDIES FOR THE SOFTWARE.

7. LIMITATION OF LIABILITY. D-WAVE WILL NOT BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE POSSESSION OF, USE OF OR INABILITY TO
USE THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, LOST PROFITS, LOSS OF GOODWILL, WORK STOPPAGE, DATA LOSS, OR COMPUTER FAILURE OR MALFUNCTION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND REGARDLESS OF WHETHER THE CLAIM OR LIABILITY IS BASED UPON ANY CONTRACT, TORT, BREACH OF WARRANTY OR OTHER LEGAL OR EQUITABLE THEORY AND NOTWITHSTANDING THAT ANY REMEDY HEREIN FAILS OF ITS ESSENTIAL PURPOSE. THE MAXIMUM LIABILITY OF D-WAVE TO LICENSEE SHALL IN ANY EVENT NOT EXCEED THE LESSER OF (A) THE SUM OF THE LICENSE FEE PAID BY LICENSEE FOR THE SOFTWARE; OR (B) CAD$250. THE FOREGOING LIMITATIONS WILL APPLY EVEN IF THE ABOVE STATED REMEDY FAILS OF ITS ESSENTIAL PURPOSE. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, AND PARAGRAPHS 3 AND 4 WILL NOT APPLY ONLY TO THE EXTENT THAT APPLICABLE LAW REQUIRES LIABILITY NOTWITHSTANDING THE LIMITATIONS OR EXCLUSIONS THEREIN.

8. LICENSEE ACKNOWLEDGES AND AGREES THAT THE SOFTWARE IS NOT DESIGNED, INTENDED, AUTHORIZED OR SUITABLE FOR USE IN ANY TYPE OF SYSTEM, APPLICATION, SITUATION OR ENVIRONMENT WHERE THE FAILURE OR TIME DELAYS OF, OR ERRORS OR INACCURACIES IN, THE CONTENT, DATA OR INFORMATION PROVIDED BY THE SOFTWARE COULD LEAD TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE, INCLUDING WITHOUT LIMITATION THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, LIFE SUPPORT, MEDICAL SYSTEMS OR WEAPONS SYSTEMS AND THAT LICENSEE WILL NOT USE THE SOFTWARE IN SUCH SYSTEM, APPLICATION, SITUATION OR ENVIRONMENT.

9. INDEMNIFICATION. To the maximum extent permitted by law, Licensee agrees to defend, indemnify and hold harmless D-WAVE, its affiliates and their respective directors, officers, employees and agents from and against any and all claims, actions, suits or proceedings, as well as any and all losses, liabilities, damages, costs and expenses (including reasonable attorneys fees) arising out of or accruing from (a) Licensee’s use of the Software, (b) any application Licensee develops using the Software that infringes or violates any right of any person or defames any person, and (c) any non-compliance by Licensee with this EULA.

10. TERMINATION.

a. This EULA is effective from the date on which the attached Software is installed or used by Licensee until terminated. This EULA will terminate automatically without notice from D-WAVE if Licensee fails to comply with any provision of this EULA or if D-WAVE is required to do so by law. If Licensee commences or participates in any legal proceeding against D-WAVE, then D-WAVE may, in its sole discretion, suspend or terminate this EULA during the pendency of such legal proceedings. Licensee may voluntarily terminate this EULA at any time by ceasing to use the Software.

b. Upon termination of this EULA at any time and for any reason, Licensee will immediately discontinue any and all use of the Software, purge the Software from all computer systems, storage media and other files, including any backup copy, and return to D-WAVE the Software, including the Documentation, and all copies thereof, or at the request of D-WAVE, destroy the Software, the Documentation, and all copies thereof, and deliver to D-WAVE certification that Licensee has complied with these termination obligations.

c. If this EULA is terminated at any time and for any reason, Licensee will not be entitled to compensation of any kind, financial or otherwise.

d. Paragraphs 3 through 9, 13(a) and 13(f) of this EULA and such other provisions as may reasonably be expected to remain in force will survive the termination of this EULA and will remain in full force and effect following such termination.

11. NO WAIVERS. A waiver expressed or implied by a party of any default of the other party in the observance or performance of this EULA must be in writing, and such waiver does not constitute and is not to be construed as a waiver or condonation of any subsequent or other default, and a waiver of any provision of this EULA will not be effective unless in writing and signed by an authorized officer of D-WAVE.

12. MODIFICATIONS AND NOTICES. No reseller, distributor or dealer of the Software is authorized to make any
modifications, extensions, or additions to this EULA. D-WAVE may, in its sole discretion, modify this EULA. If D-WAVE elects to do so, D-WAVE will notify Licensee of such change. If Licensee does not agree to the change, then this EULA automatically terminates and Licensee must stop using the Software and comply with Section 10(b) of this EULA. If Licensee does not stop using the Software, then Licensee’s use of the Software will continue under the changed EULA. D-WAVE may give notices to Licensee, at D-WAVE’s option, by posting on any portion of www.dwavesys.com or by electronic mail to any e-mail address provided by Licensee to D-WAVE.

13. GOVERNING LAW AND GENERAL PROVISIONS.

a. Governing Law and Other General Provisions. This EULA will be governed by and construed in accordance with the substantive laws in force in the Province of British Columbia, Canada, excluding its conflict of law rules. The courts of the Province of British Columbia shall have exclusive jurisdiction over all disputes relating to this EULA. Licensee and D-WAVE agree to submit to the exclusive jurisdiction of the courts of British Columbia. This EULA will not be governed by the conflict of law rules of any jurisdiction or the United Nations Convention on Contracts for the International Sale of Goods, the application of which is expressly excluded.

b. Time. Time is of the essence of this EULA.

c. Severance. If any provision in this EULA should be held invalid or unenforceable, such provision shall be modified to the extent necessary to render it valid or enforceable or severed from this EULA if no such modification is possible, and the other provisions of this EULA shall remain in full force and effect.

d. Assignment. Licensee may not assign its rights or delegate its obligations under this EULA without the prior written consent of D-WAVE, which consent may be withheld in the sole discretion of D-WAVE. D-WAVE may assign its rights or delegate its obligations under this EULA to any person in its sole discretion. This EULA shall be binding on and inure to the benefit of the parties and their successors and permitted assigns.

e. No Other EULAs. This EULA is the entire agreement of the parties with respect to the Software, and there are no other agreements, written, oral, electronic or otherwise, with respect to the Software, and it supersedes any prior representations, discussions, undertakings, communications or advertising relating to the Software.

f. Compliance with Licenses. If Licensee is a business or organization, Licensee agrees that upon request from D-WAVE or D-WAVE’s authorized representative, Licensee will within thirty (30) days fully document and certify that use of any and all Software at the time of the request is in conformity with Licensee’s valid licenses from D-WAVE.

g. English Language. Licensee and D-WAVE confirm that it is their wish that this EULA, as well as other documents relating hereto, including notices, have been and shall be written in English only. Licensee et D-WAVE confirment leur désir que cet accord ainsi que tous les documents, y compris tous avis qui s’y rattachent, soient rédigés en anglais.

Updated January 4, 2017
As part of the installation of the `dwave-ocean-sdk` a `dwave` executable is installed; for example, in a virtual environment it might be installed as `<virtual_environment>/Scripts/dwave.exe`. Running this file from your system’s console opens an interactive command line interface (CLI) that can help you set up and configure your development environment, communicate with D-Wave compute resources, and other useful actions.

Run `dwave -- help` for information on all the CLI options. For SDK version 1.6.1 the CLI provided the following commands and options (see the output in your installation for the latest):

```
$ dwave
Usage: dwave [OPTIONS] COMMAND [ARGS]...

D-Wave Cloud Client interactive configuration tool.

Options:
--version    Show the version and exit.
--debug      Enable debug logging.
--trace      Enable trace-level debug logging.
--log LEVEL  Set custom numeric or symbolic log level.
--help       Show this message and exit.

Commands:
  config       Create, update or inspect cloud client configuration file(s).
  install      Install optional non-open-source Ocean packages.
  ping         Ping the QPU by submitting a single-qubit problem.
  sample       Submit Ising-formulated problem and return samples.
  setup        Setup optional Ocean packages and configuration file(s).
  solvers      Get solver details.
  upload       Multipart problem upload with cold restart support.
```

**Note:** If you work in a Bash shell and want command completion for `dwave`, add

```
eval "$_DWAVE_COMPLETE=source <path>/dwave"
```

129
to your shell’s `.bashrc` configuration file, where `<path>` is the absolute path to the installed `dwave` executable, for example `/home/Mary/my-quantum-app/env/bin`.

Below are shown example sessions for some of the commands:

- **Setup**
- **Configure**
- **Ping**
- **Solvers**

### 5.1 Setup

The `dwave setup` command optionally installs non-open-source packages and configures your environment. The output shown below includes the interactive prompts and placeholder replies for a full setup.

```
$ dwave setup
Optionally install non-open-source packages and configure your environment.
Do you want to **select** non-open-source packages to install (y/n)? [y]:
D-Wave Drivers
These drivers enable some automated performance-tuning features.
This package is available under the 'D-Wave EULA' license.
The terms of the license are available online: https://docs.ocean.dwavesys.com/eula
Install (y/n)? [y]:
Installing: D-Wave Drivers
Successfully installed D-Wave Drivers.

D-Wave Problem Inspector
This tool visualizes problems submitted to the quantum computer and the results returned.
This package is available under the 'D-Wave EULA' license.
The terms of the license are available online: https://docs.ocean.dwavesys.com/eula
Install (y/n)? [y]:
Installing: D-Wave Problem Inspector
Successfully installed D-Wave Problem Inspector.

Creating the D-Wave configuration file.
Configuration file not found; the default location is: /home/jane/.config/dwave/dwave.conf:
Confirm configuration file path [/home/jane/.config/dwave/dwave.conf]:
Profile (create new) [prod]:
API endpoint URL [skip]:
Authentication token [skip]: ABC-1234567890abcdef1234567890abcdef
Default client class (qpu or sw) [qpu]:
Default solver [skip]:
Configuration saved.
```
5.2 Configure

The `dwave config` command configures your environment.

The output shown below includes the interactive prompts and placeholder replies.

```
$ dwave config create
Configuration file not found; the default location is: /home/jane/.config/dwave/dwave.conf

Confirm configuration file path [/home/jane/.config/dwave/dwave.conf]:
Profile (create new) [prod]:
API endpoint URL [skip]:
Authentication token [skip]: ABC-1234567890abcdef1234567890abcdef
Default client class (qpu or sw) [qpu]:
Default solver [skip]:
Configuration saved.
```

5.3 Ping

The `dwave ping` command tests communications with the quantum computer configured using the `dwave setup` or `dwave config` commands.

```
$ dwave ping
Using endpoint: https://cloud.dwavesys.com/sapi
Using solver: My_DWAVE_2000Q

Wall clock time:
* Solver definition fetch: 2007.239 ms
* Problem submit and results fetch: 1033.931 ms
* Total: 3041.171 ms

QPU timing:
* total_real_time = 10493 us
* anneal_time_per_run = 20 us
* post_processing_overhead_time = 128 us
* qpu_anneal_time_per_sample = 20 us

# Snipped for brevity
```

5.4 Solvers

The `dwave solvers` command queries which D-Wave compute resources are currently available to your account based.

```
$ dwave solvers
Solver: DW_2000Q_33
Parameters:
   anneal_offsets: A list of anneal offsets for each working qubit (NaN if u...
   anneal_schedule: A piecewise linear annealing schedule specified by a list...
   annealing_time: A positive integer that sets the duration (in microsecond...

<Output snipped for brevity>
```

(continues on next page)
Properties:
  anneal_offset_ranges: [-0.18627387668142237, 0.09542224439071689], [-0.1836548, ...
  anneal_offset_step: 0.00426679499507194
  anneal_offset_step_phi0: 0.0002716837027763096
  annealing_time_range: [1, 150000]
  couplers: [[0, 4], [1, 4], [2, 4], [3, 4], [0, 5], [1, 5], [2, 5], ...

<Output snipped for brevity>
The SDK includes the *dwave CLI* and the following packages:
Table 1: Ocean Packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimod</td>
<td>Shared API for binary quadratic samplers. Provides a binary quadratic model (BQM) class that contains Ising and quadratic unconstrained binary optimization (QUBO) models used by samplers such as the D-Wave system. Also provides utilities for constructing new samplers and composed samplers.</td>
</tr>
<tr>
<td>dwave-binarycsp</td>
<td>Library to construct a binary quadratic model from a constraint satisfaction problem with small constraints over binary variables.</td>
</tr>
<tr>
<td>dwave-cloud-client</td>
<td>Minimal implementation of the REST interface used to communicate with D-Wave Sampler API (SAPI) servers.</td>
</tr>
<tr>
<td>dwave-hybrid</td>
<td>A general, minimal Python framework for building hybrid asynchronous decomposition samplers for quadratic unconstrained binary optimization (QUBO) problems.</td>
</tr>
<tr>
<td>dwave-inspector</td>
<td>A tool for visualizing problems submitted to, and answers received from, a D-Wave structured solver such as a D-Wave 2000Q quantum computer.</td>
</tr>
<tr>
<td>dwave-neal</td>
<td>An implementation of a simulated annealing sampler.</td>
</tr>
<tr>
<td>dwave-networkx</td>
<td>Extension of NetworkX—a Python language package for exploration and analysis of networks and network algorithms—for users of D-Wave Systems. dwave-networkx provides tools for working with Chimera graphs and implementations of graph-theory algorithms on the D-Wave system and other binary quadratic model samplers.</td>
</tr>
<tr>
<td>dwave-ocean-sdk</td>
<td>Installer for D-Wave’s Ocean Tools.</td>
</tr>
<tr>
<td>dwave-system</td>
<td>Basic API for easily incorporating the D-Wave system as a sampler in the D-Wave Ocean software stack. It includes DWaveSampler, a dimod sampler that accepts and passes system parameters such as system identification and authentication down the stack. It also includes several useful composites—layers of pre- and post-processing—that can be used with DWaveSampler to handle minor-embedding, optimize chain strength, etc.</td>
</tr>
<tr>
<td>dwave-tabu</td>
<td>An implementation of the MST2 multistart tabu search algorithm for quadratic unconstrained binary optimization (QUBO) problems with a dimod Python wrapper.</td>
</tr>
<tr>
<td>penalty-model</td>
<td>An approach to solve a constraint satisfaction problem (CSP) using an Ising model or a QUBO, is to map each individual constraint in the CSP to a ‘small’ Ising model or QUBO. Includes a local cache for penalty models and a factory that generates penalty models using SMT solvers.</td>
</tr>
<tr>
<td>minorminer</td>
<td>A tool for finding graph minor-embeddings, developed to embed Ising problems onto quantum annealers (QA). While it can be used to find minors in arbitrary graphs, it is particularly geared towards the state of the art in QA: problem graphs of a few to a few hundred variables, and hardware graphs of a few thousand qubits.</td>
</tr>
<tr>
<td>qbsolv</td>
<td>A decomposing solver that finds a minimum value of a large quadratic unconstrained binary optimization (QUBO) problem by splitting it into pieces. The pieces are solved using a classical solver running the tabu algorithm. qbsolv also enables configuring a D-Wave system as the solver.</td>
</tr>
</tbody>
</table>
6.1 dimod

dimod is a shared API for binary quadratic samplers. It provides a binary quadratic model (BQM) class that contains Ising and quadratic unconstrained binary optimization (QUBO) models used by samplers such as the D-Wave system. It also provides utilities for constructing new samplers and composed samplers and for minor-embedding. Its reference examples include several samplers and composed samplers.

6.1.1 Introduction

dimod provides a binary quadratic model (BQM) class that encodes Ising and quadratic unconstrained binary optimization (QUBO) models used by samplers such as the D-Wave system.

It provides useful functionality for working with these models and samplers; for example generators_dimod to build BQMs and Utilities for calculating the energy of a sample or serializing dimod objects.

It includes reference samplers and composites for processing binary quadratic programs and refining sampling and useful for testing your code during development.

It also provides an API for Samplers and Composites for constructing new samplers and composed samplers tailored for your problem.

Additionally, it provides some Higher-Order Composites and functionality such as reducing higher-order polynomials to BQMs.

• For an introduction to BQMs, see Binary Quadratic Models.
• For an introduction to samplers and composites, see Samplers and Composites.

Examples

Solving problems with large numbers of variables might necessitate the use of decomposition\(^1\) methods such as branch-and-bound to reduce the number of variables. The following illustrative example reduces an Ising model for a small problem (the K4 complete graph), and converts the reduced-variables model to QUBO formulation.

\[
\begin{align*}
\text{linear} & = \{1: 1, 2: 2, 3: 3, 4: 4\} \\
\text{quadratic} & = \{(1, 2): 12, (1, 3): 13, (1, 4): 14, \ldots (2, 3): 23, (2, 4): 24, \ldots (3, 4): 34\} \\
\text{bqm}_\text{k4} & = \text{dimod. BinaryQuadraticModel(linear, quadratic, 0.5, dimod.SPIN)} \\
\text{bqm}_\text{k4}.\text{vartype} & = \text{Vartype.SPIN: frozenset\{1, -1\}} \}
\end{align*}
\]

\[
\begin{align*}
\text{len(bqm}_\text{k4}.\text{linear}) & = 4 \\
\text{bqm}_\text{k4}.\text{contract_variables}(2, 3) \\
\text{len(bqm}_\text{k4}.\text{linear}) & = 3 \\
\text{bqm}_\text{no3}_\text{qubo} & = \text{bqm}_\text{k4}.\text{binary} \\
\text{bqm}_\text{no3}_\text{qubo}.\text{vartype} & = \text{Vartype.BINARY: frozenset\{0, 1\}} \\
\end{align*}
\]

The next example uses a composed sampler on the Boolean NOT Gate example detailed in the Getting Started documentation. The ExactSolver test sampler calculates the energy of all possible samples; the FixedVariableComposite composite sets the value and removes specified variables from the BQM before sending it to the sampler. Fixing variable \(x\), the input to the NOT gate, to 1 results in valid solution \(z = 0\) having lower energy (-1) than solution \(x = z = 1\), which is an invalid state for a NOT gate.

\(^1\) Ocean software’s D-Wave Hybrid provides tools for decomposing large problems.
```python
>>> from dimod import FixedVariableComposite, ExactSolver
>>> Q = {('x', 'x'): -1, ('x', 'z'): 2, ('z', 'x'): 0, ('z', 'z'): -1}
>>> composed_sampler = FixedVariableComposite(ExactSolver())
>>> sampleset = composed_sampler.sample_qubo(Q, fixed_variables={'x': 1})
>>> print(sampleset)
  x  z  energy  num_oc.
0  1  0    -1.0     1
1  1  1     0.0     1
['BINARY', 2 rows, 2 samples, 2 variables]
```

The next example creates a dimod sampler by implementing a single method (in this example the `sample_ising()` method).

```python
class LinearIsingSampler(dimod.Sampler):
    def sample_ising(self, h, J):
        sample = linear_ising(h, J)  # Defined elsewhere
        energy = dimod.ising_energy(sample, h, J)
        return dimod.Response.from_samples([sample], {'energy': [energy]})

    @property
def properties(self):
        return dict()

    @property
def parameters(self):
        return dict()
```

The `Sampler ABC provides the other sample methods “for free” as mixins.

### 6.1.2 Reference Documentation

**Release** 2.5.0  
**Date** Aug 12, 2020

**Binary Quadratic Models**

*Binary quadratic models* (BQMs) are described under **Binary Quadratic Models**.

**BQM Classes**

*dimod AdjArrayBQM*

```python
class AdjArrayBQM(vartype=None, *args)
    A binary quadratic model structured as two c++ vectors.
    Can be created in several ways:
    AdjArrayBQM(vartype) Creates an empty binary quadratic model.
    AdjArrayBQM(bqm) Creates a BQM from another BQM. See copy and cls kwargs below.
    AdjArrayBQM(bqm, vartype) Creates a BQM from another BQM, changing to the appropriate vartype if necessary.
```
**AdjArrayBQM(n, vartype)** Creates a BQM with \( n \) variables, indexed linearly from zero, setting all biases to zero.

**AdjArrayBQM(quadratic, vartype)** Creates a BQM from quadratic biases given as a square array_like or a dictionary of the form \( \{(u,v): b, \ldots\} \). Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.

**AdjArrayBQM(linear, quadratic, vartype)** Creates a BQM from linear and quadratic biases, where \( \text{linear} \) is a one-dimensional array_like or a dictionary of the form \( \{v: b, \ldots\} \), and \( \text{quadratic} \) is a square array_like or a dictionary of the form \( \{(u,v): b, \ldots\} \). Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.

**AdjArrayBQM(linear, quadratic, offset, vartype)** Creates a BQM from linear and quadratic biases, where \( \text{linear} \) is a one-dimensional array_like or a dictionary of the form \( \{v: b, \ldots\} \), and \( \text{quadratic} \) is a square array_like or a dictionary of the form \( \{(u,v): b, \ldots\} \), and \( \text{offset} \) is a numerical offset. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.

**Notes**

The AdjArrayBQM is implemented using two c++ vectors. The first vector contains the linear biases and the index of the start of each variable’s neighborhood in the second vector. The second vector contains the neighboring variables and their associated quadratic biases. The vectors, once initialized, are not resized.

Advantages:

- Very fast iteration over the biases

Disadvantages:

- Does not support incremental construction
- Only supports float64 biases

Intended Use:

- When performance is important and the BQM can be treated as read-only

**Examples**

```python
>>> import numpy as np
>>> from dimod import AdjArrayBQM

>>> from dimod import AdjArrayBQM
>>> AdjArrayBQM(np.triu(np.ones((2, 2))), 'BINARY')
AdjArrayBQM([0: 1.0, 1: 1.0], {(0, 1): 1.0}, 0.0, 'BINARY')

>>> AdjArrayBQM({'a': -1}, {('a', 'b'): 1}, 'SPIN')
AdjArrayBQM(a: -1.0, b: 0.0), {('a', 'b'): 1.0}, 0.0, 'SPIN')
```

**Attributes**

- **dtype**
  The data type of the linear biases, float64.

Continued on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>itype</code></td>
<td>The data type of the indices, uint32.</td>
</tr>
<tr>
<td><code>ntype</code></td>
<td>The data type of the neighborhood indices, varies by platform.</td>
</tr>
<tr>
<td><code>num_interactions</code></td>
<td>The number of interactions in the model.</td>
</tr>
<tr>
<td><code>num_variables</code></td>
<td>The number of variables in the model.</td>
</tr>
<tr>
<td><code>offset</code></td>
<td>The constant energy offset associated with the model.</td>
</tr>
<tr>
<td><code>shape</code></td>
<td>A 2-tuple, the <code>num_variables</code> and <code>num_interactions</code>.</td>
</tr>
<tr>
<td><code>variables</code></td>
<td>The variables of the binary quadratic model.</td>
</tr>
<tr>
<td><code>vartype</code></td>
<td>The variable type, Vartype.SPIN or Vartype.BINARY.</td>
</tr>
</tbody>
</table>

`dimod.bqm.adjarraybqm.AdjArrayBQM.dtype`

AdjArrayBQM.`dtype`

The data type of the linear biases, float64.

`dimod.bqm.adjarraybqm.AdjArrayBQM.itype`

AdjArrayBQM.`itype`

The data type of the indices, uint32.

`dimod.bqm.adjarraybqm.AdjArrayBQM.ntype`

AdjArrayBQM.`ntype`

The data type of the neighborhood indices, varies by platform.

`dimod.bqm.adjarraybqm.AdjArrayBQM.num_interactions`

AdjArrayBQM.`num_interactions`

The number of interactions in the model.

Type: `int`

`dimod.bqm.adjarraybqm.AdjArrayBQM.num_variables`

AdjArrayBQM.`num_variables`

The number of variables in the model.

Type: `int`

`dimod.bqm.adjarraybqm.AdjArrayBQM.offset`

AdjArrayBQM.`offset`

The constant energy offset associated with the model.
dimod.bqm.adjarraybqm.AdjArrayBQM.shape

AdjArrayBQM.shape
A 2-tuple, the num_variables and num_interactions.

dimod.bqm.adjarraybqm.AdjArrayBQM.variables

AdjArrayBQM.variables
The variables of the binary quadratic model.

dimod.bqm.adjarraybqm.AdjArrayBQM.vartype

AdjArrayBQM.vartype
The variable type, Vartype.SPIN or Vartype.BINARY.

Views

<table>
<thead>
<tr>
<th>adj</th>
<th>Quadratic biases as a nested dict of dicts.</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>Linear biases as a mapping.</td>
</tr>
<tr>
<td>quadratic</td>
<td>Quadratic biases as a flat mapping.</td>
</tr>
<tr>
<td>binary</td>
<td>The binary-valued version of the binary quadratic model.</td>
</tr>
<tr>
<td>spin</td>
<td>The spin-valued version of the binary quadratic model.</td>
</tr>
</tbody>
</table>

dimod.bqm.adjarraybqm.AdjArrayBQM.adj

AdjArrayBQM.adj
Quadratic biases as a nested dict of dicts.
Accessed like a dict of dicts, where the keys of the outer dict are all of the model’s variables (e.g. v) and the values are the neighborhood of v. Each neighborhood if a dict where the keys are the neighbors of v and the values are their associated quadratic biases.

dimod.bqm.adjarraybqm.AdjArrayBQM.linear

AdjArrayBQM.linear
Linear biases as a mapping.
Accessed like a dict, where keys are the variables of the binary quadratic model and values are the linear biases.

dimod.bqm.adjarraybqm.AdjArrayBQM.quadratic

AdjArrayBQM.quadratic
Quadratic biases as a flat mapping.
Accessed like a dict, where keys are 2-tuples of variables, which represent an interaction and values are the quadratic biases.
AdjArrayBQM.binary

The binary-valued version of the binary quadratic model.

If the binary quadratic model is binary-valued, this references itself, otherwise it is a BinaryView.

AdjArrayBQM.spin

The spin-valued version of the binary quadratic model.

If the binary quadratic model is spin-valued, this references itself, otherwise it is a SpinView.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_offset(offset)</td>
<td>Add specified value to the offset of a binary quadratic model.</td>
</tr>
<tr>
<td>change_vartype(self, vartype[, inplace])</td>
<td>Return a binary quadratic model with the specified vartype.</td>
</tr>
<tr>
<td>copy([deep])</td>
<td>Return a copy.</td>
</tr>
<tr>
<td>degree(self, v)</td>
<td></td>
</tr>
<tr>
<td>degrees([array, dtype])</td>
<td>Create a new empty binary quadratic model.</td>
</tr>
<tr>
<td>empty(vartype)</td>
<td></td>
</tr>
<tr>
<td>energies(self, samples[, dtype])</td>
<td>Determine the energies of the given samples.</td>
</tr>
<tr>
<td>energy(sample[, dtype])</td>
<td></td>
</tr>
<tr>
<td>flip_variable(v)</td>
<td>Flip variable v in a binary quadratic model.</td>
</tr>
<tr>
<td>from_coo(obj[, vartype])</td>
<td>Deserialize a binary quadratic model from a COOrdinate format encoding.</td>
</tr>
<tr>
<td>from_ising(h, J[, offset])</td>
<td>Create a binary quadratic model from an Ising problem.</td>
</tr>
<tr>
<td>from_networkx_graph(G[, vartype, ...])</td>
<td>Create a binary quadratic model from a NetworkX graph.</td>
</tr>
<tr>
<td>from_numpy_matrix(mat[, variable_order, ...])</td>
<td>Create a binary quadratic model from a NumPy array.</td>
</tr>
<tr>
<td>from_numpy_vectors(linear, quadratic, ...[, ...])</td>
<td>Create a binary quadratic model from vectors.</td>
</tr>
<tr>
<td>from_qubo(Q[, offset])</td>
<td>Create a binary quadratic model from a QUBO problem.</td>
</tr>
<tr>
<td>get_linear(self, v)</td>
<td></td>
</tr>
<tr>
<td>get_quadratic(self, u, v[, default])</td>
<td>Get the quadratic bias of (u, v).</td>
</tr>
<tr>
<td>has_variable(v)</td>
<td>Return True if v is a variable in the binary quadratic model.</td>
</tr>
<tr>
<td>iter_interactions()</td>
<td>Iterate over the interactions of the binary quadratic model.</td>
</tr>
<tr>
<td>iter_linear(self)</td>
<td></td>
</tr>
<tr>
<td>iter_neighbors(u)</td>
<td>Iterate over neighbors of a variable in the binary quadratic model.</td>
</tr>
<tr>
<td>iter_quadratic(self[, variables])</td>
<td>Iterate over the variables of the binary quadratic model.</td>
</tr>
<tr>
<td>iter_variables()</td>
<td></td>
</tr>
<tr>
<td>normalize([bias_range, quadratic_range, ...])</td>
<td>Normalizes the biases of the binary quadratic model such that they fall in the provided range(s), and adjusts the offset appropriately.</td>
</tr>
<tr>
<td>relabel_variables</td>
<td></td>
</tr>
</tbody>
</table>
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>relabel_variables_as_integers</code></td>
<td>Relabel the variables in the BQM to integers.</td>
</tr>
<tr>
<td><code>scale(scalar[, ignored_variables, ...])</code></td>
<td>Multiply all the biases by the specified scalar.</td>
</tr>
<tr>
<td><code>set_linear(self, v, Bias b)</code></td>
<td></td>
</tr>
<tr>
<td><code>set_quadratic(self, u, v, Bias b)</code></td>
<td></td>
</tr>
<tr>
<td><code>shapeable()</code></td>
<td></td>
</tr>
<tr>
<td><code>to_coo([fp, vartype_header])</code></td>
<td>Serialize the binary quadratic model to a COOrdinate format encoding.</td>
</tr>
<tr>
<td><code>to_ising()</code></td>
<td>Converts a binary quadratic model to Ising format.</td>
</tr>
<tr>
<td><code>to_networkx_graph([node_attribute_name, ...])</code></td>
<td>Convert a binary quadratic model to NetworkX graph format.</td>
</tr>
<tr>
<td><code>to_numpy_matrix([variable_order])</code></td>
<td>Convert a binary quadratic model to NumPy 2D array.</td>
</tr>
<tr>
<td><code>to_numpy_vectors(self[, variable_order, ...])</code></td>
<td>Convert to numpy vectors.</td>
</tr>
<tr>
<td><code>to_qubo()</code></td>
<td>Convert a binary quadratic model to QUBO format.</td>
</tr>
<tr>
<td><code>remove_offset()</code></td>
<td>Set the binary quadratic model’s offset to zero.</td>
</tr>
</tbody>
</table>

**dimod.bqm.adjarraybqm.AdjArrayBQM.add_offset**

AdjArrayBQM. **add_offset (offset)**

Add specified value to the offset of a binary quadratic model.

**dimod.bqm.adjarraybqm.AdjArrayBQM.change_vartype**

AdjArrayBQM. **change_vartype (self, vartype, inplace=True)**

Return a binary quadratic model with the specified vartype.

**Parameters**

- **vartype** *(Vartype/str/set, optional)* – Variable type for the changed model. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **inplace** *(bool, optional, default=True)* – If True, the binary quadratic model is updated in-place; otherwise, a new binary quadratic model is returned.

**Returns** A binary quadratic model with the specified vartype.

**Return type** AdjArrayBQM

**dimod.bqm.adjarraybqm.AdjArrayBQM.copy**

AdjArrayBQM. **copy (deep=False)**

Return a copy.

**dimod.bqm.adjarraybqm.AdjArrayBQM.degree**

AdjArrayBQM. **degree (self, v)**
dimod.bqm.adjarraybqm.AdjArrayBQM.degrees

AdjArrayBQM.degrees(array=False, dtype=<class 'int'>)

dimod.bqm.adjarraybqm.AdjArrayBQM.empty

classmethod AdjArrayBQM.empty (vartype)
Create a new empty binary quadratic model.

dimod.bqm.adjarraybqm.AdjArrayBQM.energies

AdjArrayBQM.energies (self, samples, dtype=None)
Determine the energies of the given samples.

Parameters

- samples_like (samples_like) – A collection of raw samples. samples_like is an extension of NumPy’s array_like structure. See as_samples().
- dtype (data-type, optional, default=none) – The desired NumPy data type for the energies. Matches dtype by default.

Returns The energies.

Return type numpy.ndarray

dimod.bqm.adjarraybqm.AdjArrayBQM.energy

AdjArrayBQM.energy (sample, dtype=None)

dimod.bqm.adjarraybqm.AdjArrayBQM.flip_variable

AdjArrayBQM.flip_variable (v)
Flip variable v in a binary quadratic model.

Parameters

- v (variable) – Variable in the binary quadratic model.

dimod.bqm.adjarraybqm.AdjArrayBQM.from_coo

classmethod AdjArrayBQM.from_coo (obj, vartype=None)
Deserialize a binary quadratic model from a COOrdinate format encoding.

COOrdinate is a sparse encoding for binary quadratic models.

Parameters

- obj – (str/file): Either a string or a .read()-supporting file object that represents linear and quadratic biases for a binary quadratic model. This data is stored as a list of 3-tuples, (i, j, bias), where i = j for linear biases.
- vartype (Vartype/str/set, optional) – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN,'SPIN',{-1, 1}
- Vartype.BINARY, 'BINARY', {0, 1}
If not provided, the vartype must be specified with a header in the file.

**Note:** Variables must use index labels (numeric labels). Binary quadratic models created from COOrdinate format encoding have offsets set to zero.

**Note:** This method will be deprecated in the future. The preferred pattern is to use `load()` or `loads()` directly.

**dimod.bqm.adjarraybqm.AdjArrayBQM.from_ising**

**classmethod AdjaBQM.from_ising**(h, J, offset=0)
Create a binary quadratic model from an Ising problem.

**Parameters**

- **h** (dict/list) – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where v is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.
- **offset** (optional, default=0.0) – Constant offset applied to the model.

**Returns** A spin-valued binary quadratic model.

**dimod.bqm.adjarraybqm.AdjArrayBQM.from_networkx_graph**

**classmethod AdjArrayBQM.from_networkx_graph**(G, vartype=None, node_attribute_name='bias', edge_attribute_name='bias')
Create a binary quadratic model from a NetworkX graph.

**Parameters**

- **G** (networkx.Graph) – A NetworkX graph with biases stored as node/edge attributes.
- **vartype** (Vartype/str/set, optional) – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
If not provided, the G should have a vartype attribute. If vartype is provided and G.vartype exists then the argument overrides the property.
- **node_attribute_name** (hashable, optional, default='bias') – Attribute name for linear biases. If the node does not have a matching attribute then the bias defaults to 0.
- **edge_attribute_name** (hashable, optional, default='bias') – Attribute name for quadratic biases. If the edge does not have a matching attribute then the bias defaults to 0.
Returns Binary quadratic model

Note: This method will be deprecated in the future. The preferred pattern is to use the from_networkx_graph() function.

dimod.bqm.adjarraybqm.AdjArrayBQM.from_numpy_matrix
classmethod AdjArrayBQM.from_numpy_matrix(mat, variable_order=None, offset=0.0, interactions=None)
Create a binary quadratic model from a NumPy array.

Parameters

- **mat (numpy.ndarray)** – Coefficients of a quadratic unconstrained binary optimization (QUBO) model formatted as a square NumPy 2D array.
- **variable_order (list, optional)** – If provided, labels the QUBO variables; otherwise, row/column indices are used. If variable_order is longer than the array, extra values are ignored.
- **offset (optional, default=0.0)** – Constant offset for the binary quadratic model.
- **interactions (iterable, optional, default=[])** – Any additional 0.0-bias interactions to be added to the binary quadratic model. Only works for shapeable binary quadratic models.

Returns Binary quadratic model with vartype set to Vartype.BINARY.

Note: This method will be deprecated in the future. The preferred pattern is to use the constructor directly.

dimod.bqm.adjarraybqm.AdjArrayBQM.from_numpy_vectors
classmethod AdjArrayBQM.from_numpy_vectors(linear, quadratic, offset, vartype, variable_order=None)
Create a binary quadratic model from vectors.

Parameters

- **linear (array_like)** – A 1D array-like iterable of linear biases.
- **quadratic (tuple[array_like, array_like, array_like])** – A 3-tuple of 1D array_like vectors of the form (row, col, bias).
- **offset (numeric, optional)** – Constant offset for the binary quadratic model.
- **vartype (Vartype/str/set)** – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **variable_order (iterable, optional)** – If provided, labels the variables; otherwise, indices are used.

Returns A binary quadratic model
dimod.bqm.adjarraybqm.AdjArrayBQM.from_qubo

classmethod AdjArrayBQM.from_qubo(Q, offset=0)
Create a binary quadratic model from a QUBO problem.

Parameters
  • Q(dict) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form {u, v}: bias, ... where u, v, are binary-valued variables and bias is their associated coefficient.
  • offset (optional, default=0.0) – Constant offset applied to the model.

Returns A binary-valued binary quadratic model.

dimod.bqm.adjarraybqm.AdjArrayBQM.get_linear

AdjArrayBQM.get_linear(self, v)

dimod.bqm.adjarraybqm.AdjArrayBQM.get_quadratic

AdjArrayBQM.get_quadratic(self, u, v, default=None)
Get the quadratic bias of (u, v).

Parameters
  • u (hashable) – A variable in the binary quadratic model.
  • v (hashable) – A variable in the binary quadratic model.
  • default (number, optional) – Value to return if there is no interactions between u and v.

Returns The quadratic bias of (u, v).

Raises
  • ValueError – If either u or v is not a variable in the binary quadratic model or if u == v
  • ValueError – If (u, v) is not an interaction and default is None.

dimod.bqm.adjarraybqm.AdjArrayBQM.has_variable

AdjArrayBQM.has_variable(v)
Return True if v is a variable in the binary quadratic model.

dimod.bqm.adjarraybqm.AdjArrayBQM.iter_interactions

AdjArrayBQM.iter_interactions()
Iterate over the interactions of the binary quadratic model.

Yields interaction – An interaction in the binary quadratic model.
AdjArrayBQM.iter_linear(self)

AdjArrayBQM.iter_neighbors(u)

Iterate over neighbors of a variable in the binary quadratic model.

Yields variable – The neighbors of v.

AdjArrayBQM.iter_quadratic(self, variables=None)

AdjArrayBQM.iter_variables()

Iterate over the variables of the binary quadratic model.

Yields hashable – A variable in the binary quadratic model.

AdjArrayBQM.normalize(bias_range=1, quadratic_range=None, ignored_variables=None, ignored_interactions=None, ignore_offset=False)

Normalize the biases of the binary quadratic model such that they fall in the provided range(s), and adjusts the offset appropriately.

If quadratic_range is provided, then bias_range will be treated as the range for the linear biases and quadratic_range will be used for the range of the quadratic biases.

Parameters

- bias_range (number/pair) – Value/range that the biases of the BQM will be scaled to fit within. If quadratic_range is provided, this range is used to fit the linear biases.
- quadratic_range (number/pair) – The BQM will be scaled so that the quadratic biases fit within this range.
- ignored_variables (iterable, optional) – Biases associated with these variables are not scaled.
- ignored_interactions (iterable[tuple], optional) – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.
- ignore_offset (bool, default=False) – If True, the offset is not scaled.
AdjArrayBQM.relabel_variables_as_integers

Relabel the variables in the BQM to integers.

Note that this function uses the natural labelling of the underlying c++ objects.

AdjArrayBQM.scale

Multiply all the biases by the specified scalar.

Parameters

- `scalar` (number) – Value by which to scale the energy range of the binary quadratic model.
- `ignored_variables` (iterable, optional) – Biases associated with these variables are not scaled.
- `ignored_interactions` (iterable[tuple], optional) – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.
- `ignore_offset` (bool, default=False) – If True, the offset is not scaled.

AdjArrayBQM.set_linear

AdjArrayBQM.set_quadratic

AdjArrayBQM.shapeable

serialized the binary quadratic model to a COOrdinate format encoding.

COOrdinate is a sparse encoding for binary quadratic models.

Parameters

- `fp` (file, optional) – .write()-supporting file object to save the linear and quadratic biases of a binary quadratic model to. The model is stored as a list of 3-tuples, (i, j, bias), where i = j for linear biases. If not provided, returns a string.
- `vartype_header` (bool, optional, default=False) – If true, the binary quadratic model’s variable type as prepended to the string or file as a header.
**Note:** Variables must use index labels (numeric labels). Binary quadratic models saved to COOrdinate format encoding do not preserve offsets.

**Note:** This method will be deprecated in the future. The preferred pattern is to use `dump()` or `dumps()` directly.

### dimod.bqm.adjarraybqm.AdjArrayBQM.to_ising

**AdjArrayBQM.to_ising()**

Converts a binary quadratic model to Ising format.

If the binary quadratic model’s vartype is not `Vartype.SPIN`, values are converted.

**Returns** 3-tuple of form `(linear, quadratic, offset)`, where `linear` is a dict of linear biases, `quadratic` is a dict of quadratic biases, and `offset` is a number that represents the constant offset of the binary quadratic model.

**Return type** `tuple`

### dimod.bqm.adjarraybqm.AdjArrayBQM.to_networkx_graph

**AdjArrayBQM.to_networkx_graph(node_attribute_name='bias', edge_attribute_name='bias')**

Convert a binary quadratic model to NetworkX graph format.

**Parameters**

- `node_attribute_name` (hashable, optional, default='bias') – Attribute name for linear biases.
- `edge_attribute_name` (hashable, optional, default='bias') – Attribute name for quadratic biases.

**Returns** A NetworkX graph with biases stored as node/edge attributes.

**Return type** `networkx.Graph`

**Note:** This method will be deprecated in the future. The preferred pattern is to use `to_networkx_graph()`.

### dimod.bqm.adjarraybqm.AdjArrayBQM.to_numpy_matrix

**AdjArrayBQM.to_numpy_matrix(variable_order=None)**

Convert a binary quadratic model to NumPy 2D array.

**Parameters**

- `variable_order` (list, optional) – If provided, indexes the rows/columns of the NumPy array. If `variable_order` includes any variables not in the binary quadratic model, these are added to the NumPy array.

**Returns** The binary quadratic model as a NumPy 2D array. Note that the binary quadratic model is converted to `BINARY` vartype.

**Return type** `numpy.ndarray`
**Note:** This method will be deprecated in the future. The preferred pattern is to use `to_dense()`.

---

**dimod.bqm.adjarraybqm.AdjArrayBQM.to_numpy_vectors**

`AdjArrayBQM.to_numpy_vectors(self, variable_order=None, dtype=None, index_dtype=None, sort_indices=False, sort_labels=True, return_labels=False)`

Convert to numpy vectors.

**dimod.bqm.adjarraybqm.AdjArrayBQM.to_qubo**

`AdjArrayBQM.to_qubo()`

Convert a binary quadratic model to QUBO format.

If the binary quadratic model’s vartype is not `Vartype.BINARY`, values are converted.

**Returns** 2-tuple of form `(biases, offset)`, where `biases` is a dict in which keys are pairs of variables and values are the associated linear or quadratic bias and `offset` is a number that represents the constant offset of the binary quadratic model.

**Return type** tuple

**dimod.bqm.adjarraybqm.AdjArrayBQM.remove_offset**

`AdjArrayBQM.remove_offset()`

Set the binary quadratic model’s offset to zero.

---

**dimod.AdjDictBQM**

**class AdjDictBQM(*args, vartype=None)**

A binary quadratic model structured as a dict-of-dicts.

Can be created in several ways:

- `AdjDictBQM(vartype)` Creates an empty binary quadratic model.
- `AdjDictBQM(bqm)` Creates a BQM from another BQM. See `copy` and `cls` kwargs below.
- `AdjDictBQM(bqm, vartype)` Creates a BQM from another BQM, changing to the appropriate `vartype` if necessary.
- `AdjDictBQM(n, vartype)` Creates a BQM with `n` variables, indexed linearly from zero, setting all biases to zero.
- `AdjDictBQM(quadratic, vartype)` Creates a BQM from quadratic biases given as a square `array_like` or a dictionary of the form `{(u, v): b, ...}`. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.
- `AdjDictBQM(linear, quadratic, vartype)` Creates a BQM from linear and quadratic biases, where `linear` is a one-dimensional `array_like` or a dictionary of the form `{v: b, ...}`, and `quadratic` is a square `array_like` or a dictionary of the form `{(u, v): b, ...}`. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.
**AdjDictBQM(linear, quadratic, offset, vartype)** Creates a BQM from linear and quadratic biases, where linear is a one-dimensional array_like or a dictionary of the form \{v: b, ...\}, and quadratic is a square array_like or a dictionary of the form \{(u, v): b, ...\}, and offset is a numerical offset. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.

**Notes**

The AdjDictBQM is implemented using a dict-of-dicts structure. The outer dict contains the BQM’s variables as keys and the neighborhoods as values. Each neighborhood dict contains the neighbors as keys and the quadratic biases as values. The linear biases are stored as self-interactions.

Advantages:

- Pure python implementation
- Supports arbitrary python types as biases
- Low complexity for lookup operations
- Supports incremental construction

Disadvantages:

- Slow iteration
- High memory usage

Intended Use:

- For small problems or when flexibility is important

**Examples**

The first example constructs a BQM from a dict.

```python
>>> dimod.AdjDictBQM({'a': -1.0}, {('a', 'b'): 1.0}, 'SPIN')
AdjDictBQM({'a': 0.0, 'b': 0.0}, {('a', 'b'): 1.0}, 0.0, 'SPIN')
```

The next example demonstrates incremental construction:

```python
>>> bqm = dimod.AdjDictBQM('SPIN')
>>> bqm.add_variable('a')
'a'
>>> bqm.add_variable()
1
>>> bqm.set_quadratic('a', 1, 3.0)
>>> bqm
AdjDictBQM({'a': 0.0, 1: 0.0}, {('a', 1): 3.0}, 0.0, 'SPIN')
```

This example shows support for arbitrary types.

```python
>>> import numpy as np
>>> from fractions import Fraction
>>> dimod.AdjDictBQM({('a', 'b'): Fraction(1, 3)}, 'BINARY')
AdjDictBQM({'a': 0.0, 'b': 0.0}, {('a', 'b'): 1/3}, 0.0, 'BINARY')
```
## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>num_interactions</code></td>
<td>The number of interactions in the model.</td>
</tr>
<tr>
<td><code>num_variables</code></td>
<td>The number of variables in the model.</td>
</tr>
<tr>
<td><code>offset</code></td>
<td>The constant energy offset associated with the model.</td>
</tr>
<tr>
<td><code>shape</code></td>
<td>A 2-tuple, the <code>num_variables</code> and <code>num_interactions</code>.</td>
</tr>
<tr>
<td><code>variables</code></td>
<td>The variables of the binary quadratic model.</td>
</tr>
<tr>
<td><code>vartype</code></td>
<td>The vartype of the binary quadratic model. One of Vartype.SPIN or Vartype.BINARY.</td>
</tr>
</tbody>
</table>

```python
dimod.AdjDictBQM.num_interactions

AdjDictBQM.num_interactions
    The number of interactions in the model.
    Type: int

dimod.AdjDictBQM.num_variables

AdjDictBQM.num_variables
    The number of variables in the model.
    Type: int

dimod.AdjDictBQM.offset

AdjDictBQM.offset
    The constant energy offset associated with the model.

dimod.AdjDictBQM.shape

AdjDictBQM.shape
    A 2-tuple, the `num_variables` and `num_interactions`.

dimod.AdjDictBQM.variables

AdjDictBQM.variables
    The variables of the binary quadratic model.

dimod.AdjDictBQM.vartype

AdjDictBQM.vartype
    The vartype of the binary quadratic model. One of Vartype.SPIN or Vartype.BINARY.
    Type: Vartype
```
Views

<table>
<thead>
<tr>
<th>adj</th>
<th>Quadratic biases as a nested dict of dicts.</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>Linear biases as a mapping.</td>
</tr>
<tr>
<td>quadratic</td>
<td>Quadratic biases as a flat mapping.</td>
</tr>
<tr>
<td>binary</td>
<td>The binary-valued version of the binary quadratic model.</td>
</tr>
<tr>
<td>spin</td>
<td>The spin-valued version of the binary quadratic model.</td>
</tr>
</tbody>
</table>

**dimod.AdjDictBQM.adj**

AdjDictBQM.adj

Quadratic biases as a nested dict of dicts.

Accessed like a dict of dicts, where the keys of the outer dict are all of the model’s variables (e.g. v) and the values are the neighborhood of v. Each neighborhood if a dict where the keys are the neighbors of v and the values are their associated quadratic biases.

**dimod.AdjDictBQM.linear**

AdjDictBQM.linear

Linear biases as a mapping.

Accessed like a dict, where keys are the variables of the binary quadratic model and values are the linear biases.

**dimod.AdjDictBQM.quadratic**

AdjDictBQM.quadratic

Quadratic biases as a flat mapping.

Accessed like a dict, where keys are 2-tuples of variables, which represent an interaction and values are the quadratic biases.

**dimod.AdjDictBQM.binary**

AdjDictBQM.binary

The binary-valued version of the binary quadratic model.

If the binary quadratic model is binary-valued, this references itself, otherwise it is a BinaryView.

**dimod.AdjDictBQM.spin**

AdjDictBQM.spin

The spin-valued version of the binary quadratic model.

If the binary quadratic model is spin-valued, this references itself, otherwise it is a SpinView.

Methods
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_offset(offset)</code></td>
<td>Add specified value to the offset of a binary quadratic model.</td>
</tr>
<tr>
<td><code>add_interaction(u, v, bias)</code></td>
<td>Add an interaction and/or quadratic bias to a binary quadratic model.</td>
</tr>
<tr>
<td><code>add_interactions_from(quadratic)</code></td>
<td>Add interactions and/or quadratic biases to a binary quadratic model.</td>
</tr>
<tr>
<td><code>add_variable(v, bias)</code></td>
<td>Add a variable to the binary quadratic model.</td>
</tr>
<tr>
<td><code>add_variables_from(linear)</code></td>
<td>Add variables and/or linear biases to a binary quadratic model.</td>
</tr>
<tr>
<td><code>change_vartype(vartype[, inplace])</code></td>
<td>Return a binary quadratic model with the specified vartype.</td>
</tr>
<tr>
<td><code>contract_variables(u, v)</code></td>
<td>Enforce u, v being the same variable in a binary quadratic model.</td>
</tr>
<tr>
<td><code>copy([deep])</code></td>
<td>Return a copy.</td>
</tr>
<tr>
<td><code>degree(v)</code></td>
<td>The number of variables sharing an interaction with v.</td>
</tr>
<tr>
<td><code>degrees([array, dtype])</code></td>
<td>Create a new empty binary quadratic model.</td>
</tr>
<tr>
<td><code>energies(samples_like[, dtype])</code></td>
<td>Determine the energies of the given samples.</td>
</tr>
<tr>
<td><code>energy(sample[, dtype])</code></td>
<td>Fix the value of the variables and remove them.</td>
</tr>
<tr>
<td><code>flip_variable(v)</code></td>
<td>Flip variable v in a binary quadratic model.</td>
</tr>
<tr>
<td><code>from_coo(obj[, vartype])</code></td>
<td>Deserialize a binary quadratic model from a COOrdinate format encoding.</td>
</tr>
<tr>
<td><code>from_ising(h, J[, offset])</code></td>
<td>Create a binary quadratic model from an Ising problem.</td>
</tr>
<tr>
<td><code>from_networkx_graph(G[, vartype, . . .])</code></td>
<td>Create a binary quadratic model from a NetworkX graph.</td>
</tr>
<tr>
<td><code>from_numpy_matrix(mat[, variable_order, . . .])</code></td>
<td>Create a binary quadratic model from a NumPy array.</td>
</tr>
<tr>
<td><code>from_numpy_vectors(linear, quadratic, . . .[, . . .])</code></td>
<td>Create a binary quadratic model from vectors.</td>
</tr>
<tr>
<td><code>from_qubo(Q[, offset])</code></td>
<td>Create a binary quadratic model from a QUBO problem.</td>
</tr>
<tr>
<td><code>get_linear(v)</code></td>
<td>Get the linear bias of v.</td>
</tr>
<tr>
<td><code>get_quadratic(u, v[, default])</code></td>
<td>Get the quadratic bias of (u, v).</td>
</tr>
<tr>
<td><code>has_variable(v)</code></td>
<td>Return True if v is a variable in the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_interactions()</code></td>
<td>Iterate over the interactions of the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_linear()</code></td>
<td>Iterate over neighbors of a variable in the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_quadratic([variables])</code></td>
<td>Iterate over the variables of the binary quadratic model.</td>
</tr>
<tr>
<td><code>iterate()</code></td>
<td>Iterate over the variables of the binary quadratic model.</td>
</tr>
<tr>
<td><code>normalize([bias_range, quadratic_range, . . .])</code></td>
<td>Normalizes the biases of the binary quadratic model such that they fall in the provided range(s), and adjusts the offset appropriately.</td>
</tr>
<tr>
<td><code>relabel_variables(mapping[, inplace])</code></td>
<td>Relabel variables of a binary quadratic model as specified by mapping.</td>
</tr>
<tr>
<td><code>relabel_variables_as_integers([inplace])</code></td>
<td>Relabel the variables of the BQM to integers.</td>
</tr>
<tr>
<td><code>scale(scalar[, ignored_variables, . . .])</code></td>
<td>Multiply all the biases by the specified scalar.</td>
</tr>
<tr>
<td><code>set_linear(v, bias)</code></td>
<td>Set the linear bias of a variable v.</td>
</tr>
<tr>
<td><code>set_quadratic(u, v, bias)</code></td>
<td>Set the quadratic bias of (u, v).</td>
</tr>
</tbody>
</table>

Continued on next page
Table 7 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_coo([fp, vartype_header])</code></td>
<td>Serialize the binary quadratic model to a COOrdinate format encoding.</td>
</tr>
<tr>
<td><code>to_ising()</code></td>
<td>Converts a binary quadratic model to Ising format.</td>
</tr>
<tr>
<td><code>to_networkx_graph([node_attribute_name,...])</code></td>
<td>Convert a binary quadratic model to NetworkX graph format.</td>
</tr>
<tr>
<td><code>to_numpy_matrix([variable_order])</code></td>
<td>Convert a binary quadratic model to NumPy 2D array.</td>
</tr>
<tr>
<td><code>to_numpy_vectors([variable_order, dtype,...])</code></td>
<td>The BQM as 4 numpy vectors, the offset and a list of variables.</td>
</tr>
<tr>
<td><code>to_qubo()</code></td>
<td>Convert a binary quadratic model to QUBO format.</td>
</tr>
<tr>
<td><code>remove_interaction(u, v)</code></td>
<td>Remove the interaction between variables u and v.</td>
</tr>
<tr>
<td><code>remove_interactions_from(interactions)</code></td>
<td>Remove the given interactions from the binary quadratic model.</td>
</tr>
<tr>
<td><code>remove_offset()</code></td>
<td>Set the binary quadratic model's offset to zero.</td>
</tr>
<tr>
<td><code>remove_variable([v])</code></td>
<td>Remove a variable and its associated interactions.</td>
</tr>
<tr>
<td><code>remove_variables_from(variables)</code></td>
<td>Remove the given variables from the binary quadratic model.</td>
</tr>
<tr>
<td><code>update(other)</code></td>
<td>Update the binary quadratic model, adding biases from another.</td>
</tr>
</tbody>
</table>

**dimod.AdjDictBQM.add_offset**

```
AdjDictBQM.add_offset(offset)
```

Add specified value to the offset of a binary quadratic model.

**dimod.AdjDictBQM.add_interaction**

```
AdjDictBQM.add_interaction(u, v, bias)
```

Add an interaction and/or quadratic bias to a binary quadratic model.

**Parameters**

- **u (variable)** – One of the pair of variables to add to the model. Can be any python object that is a valid dict key.
- **v (variable)** – One of the pair of variables to add to the model. Can be any python object that is a valid dict key.
- **bias (bias)** – Quadratic bias associated with u, v. If u, v is already in the model, this value is added to the current quadratic bias.

**dimod.AdjDictBQM.add_interactions_from**

```
AdjDictBQM.add_interactions_from(quadratic)
```

Add interactions and/or quadratic biases to a binary quadratic model.

**Parameters quadratic (dict/iterable)** – A collection of interactions and their associated quadratic bias. If a dict, should be of the form `{(u, v): bias, ...)` where u and v are variables in the model and `bias` is there associated quadratic bias. Otherwise, whould be an iterable of `(u, v, bias)` triplets.
**dimod.AdjDictBQM.add_variable**

`AdjDictBQM.add_variable(v=None, bias=0.0)`

Add a variable to the binary quadratic model.

**Parameters**

- **v** *(hashable, optional)* – A label for the variable. Defaults to the length of the binary quadratic model, if that label is available. Otherwise defaults to the lowest available positive integer label.

- **bias** *(numeric, optional, default=0)* – The initial bias value for the added variable. If `v` is already a variable, then `bias` (if any) is adding to its existing linear bias.

**Returns**
The label of the added variable.

**Return type**
hashable

**Raises**
`TypeError` – If the label is not hashable.

**Examples**

```python
>>> bqm = dimod.AdjDictBQM('SPIN')
>>> bqm.add_variable()
0
>>> bqm.add_variable('a')
'a'
>>> bqm.add_variable()
2
>>> bqm = dimod.AdjDictBQM('SPIN')
>>> bqm.add_variable(1)
1
>>> bqm.add_variable()  # 1 is taken
0
>>> bqm.add_variable()
2
```

**dimod.AdjDictBQM.add_variables_from**

`AdjDictBQM.add_variables_from(linear)`

Add variables and/or linear biases to a binary quadratic model.

**Parameters**

- **linear** *(dict/iterable)* – A collection of variables in their associated linear biases. If a dict, should be of the form `{v: bias, ...}` where `v` is a variable and `bias` is its associated linear bias. Otherwise should be an iterable of `(v, bias)` pairs.

**dimod.AdjDictBQM.change_vartype**

`AdjDictBQM.change_vartype(vartype, inplace=True)`

Return a binary quadratic model with the specified vartype.

**Parameters**

- **vartype** *(Vartype/str/set, optional)* – Variable type for the changed model. Accepted input values:
- Vartype.SPIN, 'SPIN', {-1, 1}
- Vartype.BINARY, 'BINARY', {0, 1}

• inplace (bool, optional, default=True) – If True, the binary quadratic model is updated in-place; otherwise, a new binary quadratic model is returned.

Returns  A binary quadratic model with the specified vartype.

Return type  AdjDictBQM

dimod.AdjDictBQM.contract_variables

AdjDictBQM.contract_variables(u, v)
Enforce u, v being the same variable in a binary quadratic model.

The resulting variable is labeled ‘u’. Values of interactions between v and variables that u interacts with are added to the corresponding interactions of u.

Parameters
• u (variable) – Variable in the binary quadratic model.
• v (variable) – Variable in the binary quadratic model.

dimod.AdjDictBQM.copy

AdjDictBQM.copy (deep=False)
Return a copy.

dimod.AdjDictBQM.degree

AdjDictBQM.degree(v)
The number of variables sharing an interaction with v.

dimod.AdjDictBQM.degrees

AdjDictBQM.degrees (array=False, dtype=<class 'int'>)

dimod.AdjDictBQM.empty

classmethod AdjDictBQM.empty(vartype)
Create a new empty binary quadratic model.

dimod.AdjDictBQM.energies

AdjDictBQM.energies(samples_like, dtype=None)
Determine the energies of the given samples.

Parameters
• samples_like (samples_like) – A collection of raw samples. samples_like is an extension of NumPy’s array_like structure. See as_samples().
- **dtype** ([`numpy.dtype`], optional) – The data type of the returned energies.

  Returns  The energies.

  Return type  `numpy.ndarray`

```python
dimod.AdjDictBQM.energy
```

```python
AdjDictBQM.energy(sample, dtype=None)
```

```python
dimod.AdjDictBQM.fix_variables
```

```python
AdjDictBQM.fix_variables(fixed)
```

Fix the value of the variables and remove them.

- **fixed** (dict/iterable) – A dictionary or an iterable of 2-tuples of variable assignments.

```python
dimod.AdjDictBQM.flip_variable
```

```python
AdjDictBQM.flip_variable(v)
```

Flip variable v in a binary quadratic model.

- **v** (variable) – Variable in the binary quadratic model.

```python
dimod.AdjDictBQM.from_coo
```

```python
classmethod AdjDictBQM.from_coo(obj, vartype=None)
```

Deserialize a binary quadratic model from a COOrdinate format encoding.

COOrdinate is a sparse encoding for binary quadratic models.

- **obj** (str/file): Either a string or a `.read()`-supporting file object that represents linear and quadratic biases for a binary quadratic model. This data is stored as a list of 3-tuples, (i, j, bias), where i = j for linear biases.

- **vartype** ([`Vartype`/str/set], optional) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, 'SPIN', {-1, 1}
  - `Vartype.BINARY`, 'BINARY', {0, 1}

If not provided, the vartype must be specified with a header in the file.

**Note:** Variables must use index labels (numeric labels). Binary quadratic models created from COOrdinate format encoding have offsets set to zero.

**Note:** This method will be deprecated in the future. The preferred pattern is to use `load()` or `loads()` directly.
docs Documentation, Release 2.5.0

dimod.AdjDictBQM.from_ising

classmethod AdjDictBQM.from_ising(h, J, offset=0)
    Create a binary quadratic model from an Ising problem.

    Parameters
    • h (dict/list) – Linear biases of the Ising problem. If a dict, should be of the form {v: bias, ...} where v is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
    • J (dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.
    • offset (optional, default=0.0) – Constant offset applied to the model.

    Returns A spin-valued binary quadratic model.

dimod.AdjDictBQM.from_networkx_graph

classmethod AdjDictBQM.from_networkx_graph(G, vartype=None, node_attribute_name='bias', edge_attribute_name='bias')
    Create a binary quadratic model from a NetworkX graph.

    Parameters
    • G (networkx.Graph) – A NetworkX graph with biases stored as node/edge attributes.
    • vartype (Vartype/str/set, optional) – Variable type for the binary quadratic model. Accepted input values:
      - Vartype.SPIN, 'SPIN', {-1, 1}
      - Vartype.BINARY, 'BINARY', {0, 1}
      If not provided, the G should have a vartype attribute. If vartype is provided and G.vartype exists then the argument overrides the property.
    • node_attribute_name (hashable, optional, default='bias') – Attribute name for linear biases. If the node does not have a matching attribute then the bias defaults to 0.
    • edge_attribute_name (hashable, optional, default='bias') – Attribute name for quadratic biases. If the edge does not have a matching attribute then the bias defaults to 0.

    Returns Binary quadratic model

    Note: This method will be deprecated in the future. The preferred pattern is to use the from_networkx_graph() function.

dimod.AdjDictBQM.from_numpy_matrix

classmethod AdjDictBQM.from_numpy_matrix(mat, variable_order=None, offset=0.0, interactions=None)
    Create a binary quadratic model from a NumPy array.
Parameters

- **mat** *(numpy.ndarray)* – Coefficients of a quadratic unconstrained binary optimization (QUBO) model formatted as a square NumPy 2D array.
- **variable_order** *(list, optional)* – If provided, labels the QUBO variables; otherwise, row/column indices are used. If `variable_order` is longer than the array, extra values are ignored.
- **offset** *(optional, default=0.0)* – Constant offset for the binary quadratic model.
- **interactions** *(iterable, optional, default=[])* – Any additional 0.0-bias interactions to be added to the binary quadratic model. Only works for shapeable binary quadratic models.

Returns Binary quadratic model with vartype set to `Vartype.BINARY`.

**Note:** This method will be deprecated in the future. The preferred pattern is to use the constructor directly.

dimod.AdjDictBQM.from_numpy_vectors
classmethod AdjDictBQM.from_numpy_vectors(linear, quadratic, offset, vartype, variable_order=None)
Create a binary quadratic model from vectors.

Parameters

- **linear** *(array_like)* – A 1D array-like iterable of linear biases.
- **quadratic** *(tuple[array_like, array_like, array_like])* – A 3-tuple of 1D array_like vectors of the form (row, col, bias).
- **offset** *(numeric, optional)* – Constant offset for the binary quadratic model.
- **vartype** *(Vartype/str/set)* – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN` or `'SPIN'`, `{−1, 1}`
  - `Vartype.BINARY` or `'BINARY'`, `{0, 1}`
- **variable_order** *(iterable, optional)* – If provided, labels the variables; otherwise, indices are used.

Returns A binary quadratic model

dimod.AdjDictBQM.from_qubo
classmethod AdjDictBQM.from_qubo(Q, offset=0)
Create a binary quadratic model from a QUBO problem.

Parameters

- **Q** *(dict)* – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v` are binary-valued variables and `bias` is their associated coefficient.
- **offset** *(optional, default=0.0)* – Constant offset applied to the model.

Returns A binary-valued binary quadratic model.
dimod.AdjDictBQM.get_linear

AdjDictBQM.get_linear(v)
Get the linear bias of v.

Parameters  
\(v (\text{hashable})\) – A variable in the binary quadratic model.

Returns  The linear bias of v.

Return type  float

Raises  ValueError – If v is not a variable in the binary quadratic model.

dimod.AdjDictBQM.get_quadratic

AdjDictBQM.get_quadratic(u, v, default=None)
Get the quadratic bias of (u, v).

Parameters
- \(u (\text{hashable})\) – A variable in the binary quadratic model.
- \(v (\text{hashable})\) – A variable in the binary quadratic model.
- \(\text{default} (\text{optional})\) – Value to return if there is no interactions between \(u\) and \(v\).

Returns  The quadratic bias of (u, v).

Raises
- ValueError – If either \(u\) or \(v\) is not a variable in the binary quadratic model or if \(u == v\)
- ValueError – If \((u, v)\) is not an interaction and \(\text{default}\) is \(\text{None}\).

dimod.AdjDictBQM.has_variable

AdjDictBQM.has_variable(v)
Return True if v is a variable in the binary quadratic model.

dimod.AdjDictBQM.iter_interactions

AdjDictBQM.iter_interactions()
Iterate over the interactions of the binary quadratic model.

Yields  interaction – An interaction in the binary quadratic model.

dimod.AdjDictBQM.iter_linear

AdjDictBQM.iter_linear()
dimod.AdjDictBQM.iter_neighbors

`AdjDictBQM.iter_neighbors(u)`
Iterate over neighbors of a variable in the binary quadratic model.

Yields `variable` – The neighbors of `v`.

dimod.AdjDictBQM.iter_quadratic

`AdjDictBQM.iter_quadratic(variables=None)`

dimod.AdjDictBQM.iter_variables

`AdjDictBQM.iter_variables()`
Iterate over the variables of the binary quadratic model.

Yields `hashable` – A variable in the binary quadratic model.

dimod.AdjDictBQM.normalize

`AdjDictBQM.normalize(bias_range=1, quadratic_range=None, ignored_variables=None, ignored_interactions=None, ignore_offset=False)`
Normalizes the biases of the binary quadratic model such that they fall in the provided range(s), and adjusts the offset appropriately.

If `quadratic_range` is provided, then `bias_range` will be treated as the range for the linear biases and `quadratic_range` will be used for the range of the quadratic biases.

Parameters

- `bias_range (number/pair)` – Value/range that the biases of the BQM will be scaled to fit within. If `quadratic_range` is provided, this range is used to fit the linear biases.
- `quadratic_range (number/pair)` – The BQM will be scaled so that the quadratic biases fit within this range.
- `ignored_variables (iterable, optional)` – Biases associated with these variables are not scaled.
- `ignored_interactions (iterable[tuple], optional)` – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.
- `ignore_offset (bool, default=False)` – If True, the offset is not scaled.

dimod.AdjDictBQM.relabel_variables

`AdjDictBQM.relabel_variables(mapping, inplace=True)`
Relabel variables of a binary quadratic model as specified by `mapping`.

Parameters

- `mapping (dict)` – Dict mapping current variable labels to new ones. If an incomplete mapping is provided, unmapped variables retain their current labels.
- `inplace (bool, optional, default=True)` – If True, the binary quadratic model is updated in-place; otherwise, a new binary quadratic model is returned.
**Returns** A binary quadratic model with the variables relabeled. If `inplace` is set to True, returns itself.

`dimod.AdjDictBQM.relabel_variables_as_integers`

`AdjDictBQM.relabel_variables_as_integers(inplace=True)`

Relabel the variables of the BQM to integers.

**Parameters**

- `inplace` *(bool, optional, default=True)* – If True, the binary quadratic model is updated in-place; otherwise, a new binary quadratic model is returned.

**Returns**

A 2-tuple containing:

- A binary quadratic model with the variables relabeled. If `inplace` is set to True, returns itself.
- `dict`: The mapping that will restore the original labels.

**Return type** *tuple*

`dimod.AdjDictBQM.scale`

`AdjDictBQM.scale(scalar, ignored_variables=None, ignored_interactions=None, ignore_offset=False)`

Multiply all the biases by the specified scalar.

**Parameters**

- `scalar` *(number)* – Value by which to scale the energy range of the binary quadratic model.
- `ignored_variables` *(iterable, optional)* – Biases associated with these variables are not scaled.
- `ignored_interactions` *(iterable[tuple], optional)* – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.
- `ignore_offset` *(bool, default=False)* – If True, the offset is not scaled.

`dimod.AdjDictBQM.set_linear`

`AdjDictBQM.set_linear(v, bias)`

Set the linear bias of a variable `v`.

**Parameters**

- `v` *(hashable)* – A variable in the binary quadratic model. It is added if not already in the model.
- `b` *(numeric)* – The linear bias of `v`.

**Raises** *TypeError* – If `v` is not hashable
**dimod.AdjDictBQM.set_quadratic**

```python
AdjDictBQM.set_quadratic(u, v, bias)
```
Set the quadratic bias of (u, v).

**Parameters**

- `u` *(hashable)* – A variable in the binary quadratic model.
- `v` *(hashable)* – A variable in the binary quadratic model.
- `b` *(numeric)* – The linear bias of v.

**Raises** `TypeError` – If u or v is not hashable.

**dimod.AdjDictBQM.shapeable**

```python
classmethod AdjDictBQM.shapeable()
```

**dimod.AdjDictBQM.to_coo**

```python
AdjDictBQM.to_coo(fp=None, vartype_header=False)
```
Serialize the binary quadratic model to a COOrdinate format encoding.

**COOrdinate** is a sparse encoding for binary quadratic models.

**Parameters**

- `fp` *(file, optional) – .write()*-supporting file object to save the linear and quadratic biases of a binary quadratic model to. The model is stored as a list of 3-tuples, (i, j, bias), where \(i=j\) for linear biases. If not provided, returns a string.
- `vartype_header` *(bool, optional, default=False)* – If true, the binary quadratic model’s variable type as prepended to the string or file as a header.

**Note:** Variables must use index lables (numeric lables). Binary quadratic models saved to COOrdinate format encoding do not preserve offsets.

**Note:** This method will be deprecated in the future. The preferred pattern is to use `dump()` or `dumps()` directly.

**dimod.AdjDictBQM.to_ising**

```python
AdjDictBQM.to_ising()
```
Converts a binary quadratic model to Ising format.

If the binary quadratic model’s vartype is not `Vartype.SPIN`, values are converted.

**Returns** 3-tuple of form *(linear, quadratic, offset)*, where `linear` is a dict of linear biases, `quadratic` is a dict of quadratic biases, and `offset` is a number that represents the constant offset of the binary quadratic model.

**Return type** tuple
dimod.AdjDictBQM.to_networkx_graph

AdjDictBQM.to_networkx_graph (node_attribute_name='bias', edge_attribute_name='bias')
Convert a binary quadratic model to NetworkX graph format.

Parameters

- **node_attribute_name** *(hashable, optional, default='bias')* – Attribute name for linear biases.
- **edge_attribute_name** *(hashable, optional, default='bias')* – Attribute name for quadratic biases.

Returns A NetworkX graph with biases stored as node/edge attributes.

Return type networkx.Graph

**Note:** This method will be deprecated in the future. The preferred pattern is to use to_networkx_graph().

dimod.AdjDictBQM.to_numpy_matrix

AdjDictBQM.to_numpy_matrix (variable_order=None)
Convert a binary quadratic model to NumPy 2D array.

Parameters **variable_order** *(list, optional)* – If provided, indexes the rows/columns of the NumPy array. If variable_order includes any variables not in the binary quadratic model, these are added to the NumPy array.

Returns The binary quadratic model as a NumPy 2D array. Note that the binary quadratic model is converted to BINARY vartype.

Return type numpy.ndarray

**Note:** This method will be deprecated in the future. The preferred pattern is to use to_dense().

dimod.AdjDictBQM.to_numpy_vectors

AdjDictBQM.to_numpy_vectors (variable_order=None, dtype=<class 'float'>, index_dtype=<class 'numpy.int32'>, sort_indices=False, sort_labels=True, return_labels=False)
The BQM as 4 numpy vectors, the offset and a list of variables.

dimod.AdjDictBQM.to_qubo

AdjDictBQM.to_qubo ()
Convert a binary quadratic model to QUBO format.

If the binary quadratic model’s vartype is not Vartype.BINARY, values are converted.

Returns 2-tuple of form (biases, offset), where biases is a dict in which keys are pairs of variables and values are the associated linear or quadratic bias and offset is a number that represents the constant offset of the binary quadratic model.
Return type  tuple

dimod.AdjDictBQM.remove_interaction

AdjDictBQM.remove_interaction(u, v)
    Remove the interaction between variables u and v.

Parameters
    • u (hashable) – A variable in the binary quadratic model.
    • v (hashable) – A variable in the binary quadratic model.

Returns  If there was an interaction to remove.

Return type  bool

 Raises
    • ValueError – If either u or v is not a variable in the binary quadratic model.

dimod.AdjDictBQM.remove_interactions_from

AdjDictBQM.remove_interactions_from(interactions)
    Remove the given interactions from the binary quadratic model.

dimod.AdjDictBQM.remove_offset

AdjDictBQM.remove_offset()
    Set the binary quadratic model’s offset to zero.

dimod.AdjDictBQM.remove_variable

AdjDictBQM.remove_variable(v=None)
    Remove a variable and its associated interactions.

Parameters  v (variable, optional) – The variable to be removed from the binary quadratic model (BQM). If not provided, the last variable added is removed.

Returns  The removed variable.

Return type  variable

 Raises
    • ValueError – If the BQM is empty or if v is not a variable.

dimod.AdjDictBQM.remove_variables_from

AdjDictBQM.remove_variables_from(variables)
    Remove the given variables from the binary quadratic model.
**dimod.AdjDictBQM.update**

*AdjDictBQM.update*(other)
Update the binary quadratic model, adding biases from another.

**dimod.AdjMapBQM**

class AdjMapBQM(vartype=None, *args)
A binary quadratic model where the neighborhoods are c++ maps.

Can be created in several ways:

- **AdjMapBQM(vartype)**: Creates an empty binary quadratic model.
- **AdjMapBQM(bqm)**: Creates a BQM from another BQM. See *copy* and *cls* kwargs below.
- **AdjMapBQM(bqm, vartype)**: Creates a BQM from another BQM, changing to the appropriate *vartype* if necessary.
- **AdjMapBQM(n, vartype)**: Creates a BQM with *n* variables, indexed linearly from zero, setting all biases to zero.
- **AdjMapBQM(quadratic, vartype)**: Creates a BQM from quadratic biases given as a square *array_like* or a dictionary of the form \{(u, v): b, \ldots\}. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.
- **AdjMapBQM(linear, quadratic, vartype)**: Creates a BQM from linear and quadratic biases, where *linear* is a one-dimensional *array_like* or a dictionary of the form \{v: b, \ldots\}, and *quadratic* is a square *array_like* or a dictionary of the form \{(u, v): b, \ldots\}. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.
- **AdjMapBQM(linear, quadratic, offset, vartype)**: Creates a BQM from linear and quadratic biases, where *linear* is a one-dimensional *array_like* or a dictionary of the form \{v: b, \ldots\}, and *quadratic* is a square *array_like* or a dictionary of the form \{(u, v): b, \ldots\}, and *offset* is a numerical offset. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.

**Notes**

The AdjMapBQM is implemented using an adjacency structure where the neighborhoods are implemented as c++ maps.

Advantages:

- Fast incremental construction and destruction

Disadvantages:

- Slower iteration than *AdjArrayBQM* and *AdjVectorBQM*
- Only supports float64 biases

Intended Use:

- When performance is important but the BQM’s shape is changing frequently
Examples

```python
>>> import numpy as np
>>> from dimod import AdjMapBQM

>>> # Construct from dicts
>>> AdjMapBQM({'a': -1.0}, {('a', 'b'): 1.0}, 'SPIN')
AdjMapBQM({a: -1.0, b: 0.0}, {('a', 'b'): 1.0}, 0.0, 'SPIN')

>>> # Incremental Construction
>>> bqm = AdjMapBQM('SPIN')
>>> bqm.add_variable('a')
'a'
>>> bqm.add_variable() 1
>>> bqm.set_quadratic('a', 1, 3.0)
>>> bqm
AdjMapBQM({a: 0.0, 1: 0.0}, {('a', 1): 3.0}, 0.0, 'SPIN')
```

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtype</td>
<td>The data type of the linear biases, float64.</td>
</tr>
<tr>
<td>itype</td>
<td>The data type of the indices, uint32.</td>
</tr>
<tr>
<td>ntype</td>
<td>The data type of the neighborhood indices, varies by platform.</td>
</tr>
<tr>
<td>num_interactions</td>
<td>The number of interactions in the model.</td>
</tr>
<tr>
<td>num_variables</td>
<td>The number of variables in the model.</td>
</tr>
<tr>
<td>offset</td>
<td>The constant energy offset associated with the model.</td>
</tr>
<tr>
<td>shape</td>
<td>A 2-tuple, the num_variables and num_interactions.</td>
</tr>
<tr>
<td>variables</td>
<td>The variables of the binary quadratic model.</td>
</tr>
<tr>
<td>vartype</td>
<td>The variable type, Vartype.SPIN or Vartype.BINARY.</td>
</tr>
</tbody>
</table>

`dimod.AdjMapBQM.dtype`

`AdjMapBQM.dtype`

The data type of the linear biases, float64.

`dimod.AdjMapBQM.itype`

`AdjMapBQM.itype`

The data type of the indices, uint32.

`dimod.AdjMapBQM.ntype`

`AdjMapBQM.ntype`

The data type of the neighborhood indices, varies by platform.
dimod.AdjMapBQM.num_interactions

AdjMapBQM.num_interactions
The number of interactions in the model.
Type int

dimod.AdjMapBQM.num_variables

AdjMapBQM.num_variables
The number of variables in the model.
Type int

dimod.AdjMapBQM.offset

AdjMapBQM.offset
The constant energy offset associated with the model.

dimod.AdjMapBQM.shape

AdjMapBQM.shape
A 2-tuple, the num_variables and num_interactions.

dimod.AdjMapBQM.variables

AdjMapBQM.variables
The variables of the binary quadratic model.

dimod.AdjMapBQM.vartype

AdjMapBQM.vartype
The variable type, Vartype.SPIN or Vartype.BINARY.

Views

<table>
<thead>
<tr>
<th>View</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adj</td>
<td>Quadratic biases as a nested dict of dicts.</td>
</tr>
<tr>
<td>linear</td>
<td>Linear biases as a mapping.</td>
</tr>
<tr>
<td>quadratic</td>
<td>Quadratic biases as a flat mapping.</td>
</tr>
<tr>
<td>binary</td>
<td>The binary-valued version of the binary quadratic model.</td>
</tr>
<tr>
<td>spin</td>
<td>The spin-valued version of the binary quadratic model.</td>
</tr>
</tbody>
</table>

dimod.AdjMapBQM.adj

AdjMapBQM.adj
Quadratic biases as a nested dict of dicts.
Accessed like a dict of dicts, where the keys of the outer dict are all of the model’s variables (e.g. \( v \)) and the values are the neighborhood of \( v \). Each neighborhood is a dict where the keys are the neighbors of \( v \) and the values are their associated quadratic biases.

**dimod.AdjMapBQM.linear**

AdjMapBQM.linear

Linear biases as a mapping.

Accessed like a dict, where keys are the variables of the binary quadratic model and values are the linear biases.

**dimod.AdjMapBQM.quadratic**

AdjMapBQM.quadratic

Quadratic biases as a flat mapping.

Accessed like a dict, where keys are 2-tuples of variables, which represent an interaction and values are the quadratic biases.

**dimod.AdjMapBQM.binary**

AdjMapBQM.binary

The binary-valued version of the binary quadratic model.

If the binary quadratic model is binary-valued, this references itself, otherwise it is a BinaryView.

**dimod.AdjMapBQM.spin**

AdjMapBQM.spin

The spin-valued version of the binary quadratic model.

If the binary quadratic model is spin-valued, this references itself, otherwise it is a SpinView.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_offset</code> (offset)</td>
<td>Add specified value to the offset of a binary quadratic model.</td>
</tr>
<tr>
<td><code>add_interaction</code> (u, v, bias)</td>
<td>Add an interaction and/or quadratic bias to a binary quadratic model.</td>
</tr>
<tr>
<td><code>add_interactions_from</code> (quadratic)</td>
<td>Add interactions and/or quadratic biases to a binary quadratic model.</td>
</tr>
<tr>
<td><code>add_variable</code> (self[, v])</td>
<td>Add a variable to the binary quadratic model.</td>
</tr>
<tr>
<td><code>add_variables_from</code> (linear)</td>
<td>Add variables and/or linear biases to a binary quadratic model.</td>
</tr>
<tr>
<td><code>change_vartype</code> (self, vartype[, inplace])</td>
<td>Return a binary quadratic model with the specified vartype.</td>
</tr>
<tr>
<td><code>contract_variables</code> (u, v)</td>
<td>Enforce ( u, v ) being the same variable in a binary quadratic model.</td>
</tr>
<tr>
<td><code>copy</code> ([deep])</td>
<td>Return a copy.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>degree(self, v)</code></td>
<td>Get the degree of variable v.</td>
</tr>
<tr>
<td><code>degrees([array, dtype])</code></td>
<td>Get the degrees of variables.</td>
</tr>
<tr>
<td><code>empty(vartype)</code></td>
<td>Create a new empty binary quadratic model.</td>
</tr>
<tr>
<td><code>energies(self, samples[, dtype])</code></td>
<td>Determine the energies of the given samples.</td>
</tr>
<tr>
<td><code>energy(sample[, dtype])</code></td>
<td>Determine the energy of the given sample.</td>
</tr>
<tr>
<td><code>fix_variables(fixed)</code></td>
<td>Fix the value of the variables and remove them.</td>
</tr>
<tr>
<td><code>flip_variable(v)</code></td>
<td>Flip variable v in a binary quadratic model.</td>
</tr>
<tr>
<td><code>from_coo(obj[, vartype])</code></td>
<td>Deserialize a binary quadratic model from a COOrdinate format encoding.</td>
</tr>
<tr>
<td><code>from_ising(h, J[, offset])</code></td>
<td>Create a binary quadratic model from an Ising problem.</td>
</tr>
<tr>
<td><code>from_networkx_graph(G[, vartype, ...])</code></td>
<td>Create a binary quadratic model from a NetworkX graph.</td>
</tr>
<tr>
<td><code>from_numpy_matrix(mat[, variable_order, ...])</code></td>
<td>Create a binary quadratic model from a NumPy array.</td>
</tr>
<tr>
<td><code>from_qubo(Q[, offset])</code></td>
<td>Create a binary quadratic model from a QUBO problem.</td>
</tr>
<tr>
<td><code>get_linear(self, v)</code></td>
<td>Get the linear bias of v.</td>
</tr>
<tr>
<td><code>get_quadratic(self, u, v[, default])</code></td>
<td>Get the quadratic bias of (u, v).</td>
</tr>
<tr>
<td><code>has_variable(v)</code></td>
<td>Return True if v is a variable in the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_interactions()</code></td>
<td>Iterate over the interactions of the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_linear(self)</code></td>
<td>Iterate over neighbors of a variable in the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_neighbors(u)</code></td>
<td>Iterate over neighbors of a variable in the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_quadratic(self[, variables])</code></td>
<td>Iterate over the variables of the binary quadratic model.</td>
</tr>
<tr>
<td><code>normalize([bias_range, quadratic_range, ...])</code></td>
<td>Normalize the biases of the binary quadratic model such that they fall in the provided range(s), and adjusts the offset appropriately.</td>
</tr>
<tr>
<td><code>relabel_variables</code></td>
<td>Relabel the variables in the BQM to integers.</td>
</tr>
<tr>
<td><code>relabel_variables_as_integers</code></td>
<td>Relabel the variables in the BQM to integers.</td>
</tr>
<tr>
<td><code>scale(scalar[, ignored_variables, ...])</code></td>
<td>Multiply all the biases by the specified scalar.</td>
</tr>
<tr>
<td><code>set_linear(self, v, Bias b)</code></td>
<td>Set the linear bias of a variable v.</td>
</tr>
<tr>
<td><code>set_quadratic(self, u, v, Bias b)</code></td>
<td>Set the quadratic bias of (u, v).</td>
</tr>
<tr>
<td><code>shapeable()</code></td>
<td>Convert to numpy vectors.</td>
</tr>
<tr>
<td><code>to_coo([fp, vartype_header])</code></td>
<td>Serialize the binary quadratic model to a COOrdinate format encoding.</td>
</tr>
<tr>
<td><code>to_ising()</code></td>
<td>Converts a binary quadratic model to Ising format.</td>
</tr>
<tr>
<td><code>to_networkx_graph([node_attribute_name, ...])</code></td>
<td>Convert a binary quadratic model to NetworkX graph format.</td>
</tr>
<tr>
<td><code>to_numpy_matrix([variable_order])</code></td>
<td>Convert a binary quadratic model to NumPy 2D array.</td>
</tr>
<tr>
<td><code>to_numpy_vectors(self[, variable_order, ...])</code></td>
<td>Convert to numpy vectors.</td>
</tr>
<tr>
<td><code>to_qubo()</code></td>
<td>Convert a binary quadratic model to QUBO format.</td>
</tr>
<tr>
<td><code>remove_interactions_from(interactions)</code></td>
<td>Remove the given interactions from the binary quadratic model.</td>
</tr>
<tr>
<td><code>remove_offset()</code></td>
<td>Set the binary quadratic model’s offset to zero.</td>
</tr>
<tr>
<td><code>remove_variable(self, v)</code></td>
<td>Remove a variable and its associated interactions.</td>
</tr>
<tr>
<td><code>remove_variables_from(variables)</code></td>
<td>Remove the given variables from the binary quadratic model.</td>
</tr>
</tbody>
</table>

Continued on next page
### dimod.AdjMapBQM.add_offset

**AdjMapBQM.add_offset**(offset)

Add specified value to the offset of a binary quadratic model.

### dimod.AdjMapBQM.add_interaction

**AdjMapBQM.add_interaction**(u, v, bias)

Add an interaction and/or quadratic bias to a binary quadratic model.

**Parameters**

- **u**(variable) – One of the pair of variables to add to the model. Can be any python object that is a valid dict key.
- **v**(variable) – One of the pair of variables to add to the model. Can be any python object that is a valid dict key.
- **bias**(bias) – Quadratic bias associated with u, v. If u, v is already in the model, this value is added to the current quadratic bias.

### dimod.AdjMapBQM.add_interactions_from

**AdjMapBQM.add_interactions_from**(quadratic)

Add interactions and/or quadratic biases to a binary quadratic model.

**Parameters**

- **quadratic**(dict/iterable) – A collection of interactions and their associated quadratic bias. If a dict, should be of the form {(u, v): bias, ...} where u and v are variables in the model and bias is the associated quadratic bias. Otherwise, should be an iterable of (u, v, bias) triplets.

### dimod.AdjMapBQM.add_variable

**AdjMapBQM.add_variable**(self, v=None, Bias bias=0.0)

Add a variable to the binary quadratic model.

**Parameters**

- **v**(hashable, optional) – A label for the variable. Defaults to the length of the binary quadratic model, if that label is available. Otherwise defaults to the lowest available positive integer label.
- **bias**(numeric, optional, default=0) – The initial bias value for the added variable. If v is already a variable, then bias (if any) is adding to its existing linear bias.

**Returns**

The label of the added variable.

**Return type**

hashable

**Raises**

`TypeError` – If the label is not hashable.

---

**Table 10 – continued from previous page**

<table>
<thead>
<tr>
<th><strong>update</strong>(other)</th>
<th>Update the binary quadratic model, adding biases from another.</th>
</tr>
</thead>
</table>
**dimod.AdjMapBQM.add_variables_from**

`AdjMapBQM.add_variables_from(linear)`

Add variables and/or linear biases to a binary quadratic model.

**Parameters**
- `linear` ([dict/iterable]) – A collection of variables in their associated linear biases. If a dict, should be of the form `{v: bias, ...}` where `v` is a variable and `bias` is its associated linear bias. Otherwise should be an iterable of `(v, bias)` pairs.

**dimod.AdjMapBQM.change_vartype**

`AdjMapBQM.change_vartype(self, vartype, inplace=True)`

Return a binary quadratic model with the specified vartype.

**Parameters**
- `vartype` ([Vartype/str/set, optional]) – Variable type for the changed model. Accepted input values:
  - `Vartype.SPIN`, 'SPIN', {-1, 1}
  - `Vartype.BINARY`, 'BINARY', {0, 1}
- `inplace` (bool, optional, default=True) – If True, the binary quadratic model is updated in-place; otherwise, a new binary quadratic model is returned.

**Returns** A binary quadratic model with the specified vartype.

**Return type** `AdjMapBQM`

**dimod.AdjMapBQM.contract_variables**

`AdjMapBQM.contract_variables(u, v)`

Enforce `u`, `v` being the same variable in a binary quadratic model.

The resulting variable is labeled ‘u’. Values of interactions between `v` and variables that `u` interacts with are added to the corresponding interactions of `u`.

**Parameters**
- `u` ([variable]) – Variable in the binary quadratic model.
- `v` ([variable]) – Variable in the binary quadratic model.

**dimod.AdjMapBQM.copy**

`AdjMapBQM.copy(deep=False)`

Return a copy.

**dimod.AdjMapBQM.degree**

`AdjMapBQM.degree(self, v)`
dimod.AdjMapBQM.degrees

AdjMapBQM.degrees(array=False, dtype=int)

dimod.AdjMapBQM.empty

classmethod AdjMapBQM.empty(vartype)
    Create a new empty binary quadratic model.

dimod.AdjMapBQM.energies

AdjMapBQM.energies(self, samples, dtype=None)
    Determine the energies of the given samples.

    Parameters
    • samples_like(samples_like) – A collection of raw samples. samples_like is an
      extension of NumPy’s array_like structure. See as_samples().
    • dtype(data-type, optional, default=None) – The desired NumPy data type
      for the energies. Matches dtype by default.

    Returns
    The energies.

    Return type
    numpy.ndarray

dimod.AdjMapBQM.energy

AdjMapBQM.energy(sample, dtype=None)

dimod.AdjMapBQM.fix_variables

AdjMapBQM.fix_variables(fixed)
    Fix the value of the variables and remove them.

    Parameters
    fixed(dict/iterable) – A dictionary or an iterable of 2-tuples of variable assigments.

dimod.AdjMapBQM.flip_variable

AdjMapBQM.flip_variable(v)
    Flip variable v in a binary quadratic model.

    Parameters
    v(variable) – Variable in the binary quadratic model.

dimod.AdjMapBQM.from_coo

classmethod AdjMapBQM.from_coo(obj, vartype=None)
    Deserialize a binary quadratic model from a COOrdinate format encoding.

    COOrdinate is a sparse encoding for binary quadratic models.
Parameters

- **obj** – *(str/file): Either a string or a*.read()-supporting* file object that represents linear and quadratic biases for a binary quadratic model. This data is stored as a list of 3-tuples, *(i, j, bias)*, where *i = j* for linear biases.

- **vartype**(Vartype/str/set, optional) – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}

If not provided, the vartype must be specified with a header in the file.

**Note:** Variables must use index labels (numeric labels). Binary quadratic models created from COOrdinate format encoding have offsets set to zero.

**Note:** This method will be deprecated in the future. The preferred pattern is to use load() or loads() directly.

dimod.AdjMapBQM.from_ising

**classmethod** AdjMapBQM.from_ising *(h, J, offset=0)*

Create a binary quadratic model from an Ising problem.

**Parameters**

- **h**(dict/list) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, ...} where v is a spin-valued variable and bias is its associated bias*. If a list, it is treated as a list of biases where the indices are the variable labels.

- **J**(dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.

- **offset**(optional, default=0.0) – Constant offset applied to the model.

**Returns** A spin-valued binary quadratic model.

dimod.AdjMapBQM.from_networkx_graph

**classmethod** AdjMapBQM.from_networkx_graph *(G, vartype=None, node_attribute_name='bias', edge_attribute_name='bias')*

Create a binary quadratic model from a NetworkX graph.

**Parameters**

- **G**(networkx.Graph) – A NetworkX graph with biases stored as node/edge attributes.

- **vartype**(Vartype/str/set, optional) – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
If not provided, the $G$ should have a vartype attribute. If vartype is provided and $G$.vartype exists then the argument overrides the property.

- **node_attribute_name** *(hashable, optional, default='bias')* – Attribute name for linear biases. If the node does not have a matching attribute then the bias defaults to 0.

- **edge_attribute_name** *(hashable, optional, default='bias')* – Attribute name for quadratic biases. If the edge does not have a matching attribute then the bias defaults to 0.

**Returns** Binary quadratic model

**Note:** This method will be deprecated in the future. The preferred pattern is to use the `from_networkx_graph()` function.

### dimod.AdjMapBQM.from_numpy_matrix

**classmethod** AdjMapBQM.from_numpy_matrix *(mat, variable_order=None, offset=0.0, interactions=None)*

Create a binary quadratic model from a NumPy array.

**Parameters**

- **mat** *(numpy.ndarray)* – Coefficients of a quadratic unconstrained binary optimization (QUBO) model formatted as a square NumPy 2D array.

- **variable_order** *(list, optional)* – If provided, labels the QUBO variables; otherwise, row/column indices are used. If `variable_order` is longer than the array, extra values are ignored.

- **offset** *(optional, default=0.0)* – Constant offset for the binary quadratic model.

- **interactions** *(iterable, optional, default=[])* – Any additional 0.0-bias interactions to be added to the binary quadratic model. Only works for shapeable binary quadratic models.

**Returns** Binary quadratic model with vartype set to `Vartype.BINARY`.

**Note:** This method will be deprecated in the future. The preferred pattern is to use the constructor directly.

### dimod.AdjMapBQM.from_numpy_vectors

**classmethod** AdjMapBQM.from_numpy_vectors *(linear, quadratic, offset, vartype, variable_order=None)*

Create a binary quadratic model from vectors.

**Parameters**

- **linear** *(array_like)* – A 1D array-like iterable of linear biases.

- **quadratic** *(tuple[array_like, array_like, array_like])* – A 3-tuple of 1D array_like vectors of the form (row, col, bias).

- **offset** *(numeric, optional)* – Constant offset for the binary quadratic model.
• **vartype** (*Vartype*/str/set) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{−1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

• **variable_order** (*iterable, optional*) – If provided, labels the variables; otherwise, indices are used.

**Returns** A binary quadratic model

`dimod.AdjMapBQM.from_qubo`

classmethod `AdjMapBQM.from_qubo(Q, offset=0)`
Create a binary quadratic model from a QUBO problem.

**Parameters**

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u`, `v` are binary-valued variables and `bias` is their associated coefficient.

- **offset** (*optional, default=0.0*) – Constant offset applied to the model.

**Returns** A binary-valued binary quadratic model.

`dimod.AdjMapBQM.get_linear`

`AdjMapBQM.get_linear(self, v)`
Get the linear bias of `v`.

**Parameters** **v** (*hashable*) – A variable in the binary quadratic model.

**Returns** The linear bias of `v`.

**Return type** float

**Raises** `ValueError` – If `v` is not a variable in the binary quadratic model.

`dimod.AdjMapBQM.get_quadratic`

`AdjMapBQM.get_quadratic(self, u, v, default=None)`
Get the quadratic bias of `(u, v)`.

**Parameters**

- **u** (*hashable*) – A variable in the binary quadratic model.
- **v** (*hashable*) – A variable in the binary quadratic model.
- **default** (*number, optional*) – Value to return if there is no interactions between `u` and `v`.

**Returns** The quadratic bias of `(u, v)`.

**Raises**

- `ValueError` – If either `u` or `v` is not a variable in the binary quadratic model or if `u == v`
• **ValueError** – If \((u, v)\) is not an interaction and default is

\[\text{None}.\]

**dimod.AdjMapBQM.has_variable**

`AdjMapBQM.has_variable(v)`

Return True if \(v\) is a variable in the binary quadratic model.

**dimod.AdjMapBQM.iter_interactions**

`AdjMapBQM.iter_interactions()`

Iterate over the interactions of the binary quadratic model.

Yields **interaction** – An interaction in the binary quadratic model.

**dimod.AdjMapBQM.iter_linear**

`AdjMapBQM.iter_linear(self)`

**dimod.AdjMapBQM.iter_neighbors**

`AdjMapBQM.iter_neighbors(u)`

Iterate over neighbors of a variable in the binary quadratic model.

Yields **variable** – The neighbors of \(v\).

**dimod.AdjMapBQM.iter_quadratic**

`AdjMapBQM.iter_quadratic(self, variables=None)`

**dimod.AdjMapBQM.iter_variables**

`AdjMapBQM.iter_variables()`

Iterate over the variables of the binary quadratic model.

Yields **hashable** – A variable in the binary quadratic model.

**dimod.AdjMapBQM.normalize**

`AdjMapBQM.normalize(bias_range=1, quadratic_range=None, ignored_variables=None, ignored_interactions=None, ignore_offset=False)`

Normalizes the biases of the binary quadratic model such that they fall in the provided range(s), and adjusts the offset appropriately.

If **quadratic_range** is provided, then **bias_range** will be treated as the range for the linear biases and **quadratic_range** will be used for the range of the quadratic biases.

Parameters
• **bias_range (number/pair)** – Value/range that the biases of the BQM will be scaled to fit within. If *quadratic_range* is provided, this range is used to fit the linear biases.

• **quadratic_range (number/pair)** – The BQM will be scaled so that the quadratic biases fit within this range.

• **ignored_variables (iterable, optional)** – Biases associated with these variables are not scaled.

• **ignored_interactions (iterable[tuple], optional)** – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.

• **ignore_offset (bool, default=False)** – If True, the offset is not scaled.

**dimod.AdjMapBQM.relabel_variables**

AdjMapBQM.relabel_variables

**dimod.AdjMapBQM.relabel_variables_as_integers**

AdjMapBQM.relabel_variables_as_integers

Relabel the variables in the BQM to integers.

Note that this function uses the natural labelling of the underlying c++ objects.

**dimod.AdjMapBQM.scale**

AdjMapBQM.scale (scalar, ignored_variables=None, ignored_interactions=None, ignore_offset=False)

Multiply all the biases by the specified scalar.

**Parameters**

• **scalar (number)** – Value by which to scale the energy range of the binary quadratic model.

• **ignored_variables (iterable, optional)** – Biases associated with these variables are not scaled.

• **ignored_interactions (iterable[tuple], optional)** – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.

• **ignore_offset (bool, default=False)** – If True, the offset is not scaled.

**dimod.AdjMapBQM.set_linear**

AdjMapBQM.set_linear (self, v, Bias b)

Set the linear bias of a variable v.

**Parameters**

• **v (hashable)** – A variable in the binary quadratic model. It is added if not already in the model.

• **b (numeric)** – The linear bias of v.

**Raises** **TypeError** – If v is not hashable
**dimod.AdjMapBQM.set_quadratic**

`AdjMapBQM.set_quadratic(self, u, v, Bias b)`

Set the quadratic bias of `(u, v)`.

**Parameters**

- **u (hashable)** – A variable in the binary quadratic model.
- **v (hashable)** – A variable in the binary quadratic model.
- **b (numeric)** – The linear bias of `v`.

**Raises** `TypeError` – If `u` or `v` is not hashable.

**dimod.AdjMapBQM.shapeable**

`classmethod AdjMapBQM.shapeable()`

**dimod.AdjMapBQM.to_coo**

`AdjMapBQM.to_coo(fp=None, vartype_header=False)`

Serialize the binary quadratic model to a COOrdinate format encoding. COOrdinate is a sparse encoding for binary quadratic models.

**Parameters**

- **fp (file, optional)** – `.write()`-supporting file object to save the linear and quadratic biases of a binary quadratic model to. The model is stored as a list of 3-tuples, `(i, j, bias)`, where `i = j` for linear biases. If not provided, returns a string.
- **vartype_header (bool, optional, default=False)** – If true, the binary quadratic model’s variable type as prepended to the string or file as a header.

**Note:** Variables must use index labels (numeric labels). Binary quadratic models saved to COOrdinate format encoding do not preserve offsets.

**Note:** This method will be deprecated in the future. The preferred pattern is to use `dump()` or `dumps()` directly.

**dimod.AdjMapBQM.to_ising**

`AdjMapBQM.to_ising()`

Converts a binary quadratic model to Ising format.

If the binary quadratic model's vartype is not `Vartype.SPIN`, values are converted.

**Returns** 3-tuple of form `(linear, quadratic, offset)`, where `linear` is a dict of linear biases, `quadratic` is a dict of quadratic biases, and `offset` is a number that represents the constant offset of the binary quadratic model.

**Return type** `tuple`
dimod.AdjMapBQM.to_networkx_graph

AdjMapBQM.to_networkx_graph(node_attribute_name='bias', edge_attribute_name='bias')
Convert a binary quadratic model to NetworkX graph format.

Parameters

- **node_attribute_name** (hashable, optional, default='bias') – Attribute name for linear biases.
- **edge_attribute_name** (hashable, optional, default='bias') – Attribute name for quadratic biases.

Returns A NetworkX graph with biases stored as node/edge attributes.

Return type networkx.Graph

Note: This method will be deprecated in the future. The preferred pattern is to use to_networkx_graph().

dimod.AdjMapBQM.to_numpy_matrix

AdjMapBQM.to_numpy_matrix(variable_order=None)
Convert a binary quadratic model to NumPy 2D array.

Parameters **variable_order** (list, optional) – If provided, indexes the rows/columns of the NumPy array. If variable_order includes any variables not in the binary quadratic model, these are added to the NumPy array.

Returns The binary quadratic model as a NumPy 2D array. Note that the binary quadratic model is converted to BINARY vartype.

Return type numpy.ndarray

Note: This method will be deprecated in the future. The preferred pattern is to use to_dense().

dimod.AdjMapBQM.to_numpy_vectors

AdjMapBQM.to_numpy_vectors(self, variable_order=None, dtype=None, index_dtype=None, sort_indices=False, sort_labels=True, return_labels=False)
Convert to numpy vectors.

dimod.AdjMapBQM.to_qubo

AdjMapBQM.to_qubo()
Convert a binary quadratic model to QUBO format.

If the binary quadratic model’s vartype is not Vartype.BINARY, values are converted.

Returns 2-tuple of form (biases, offset), where biases is a dict in which keys are pairs of variables and values are the associated linear or quadratic bias and offset is a number that represents the constant offset of the binary quadratic model.

Return type tuple
dimod.AdjMapBQM.remove_interaction

AdjMapBQM.remove_interaction(self, u, v)
Remove the interaction between variables u and v.

Parameters
- u (hashable) – A variable in the binary quadratic model.
- v (hashable) – A variable in the binary quadratic model.

Returns If there was an interaction to remove.
Return type bool

Raises
- ValueError – If either u or v is not a variable in the binary quadratic model.

dimod.AdjMapBQM.remove_interactions_from

AdjMapBQM.remove_interactions_from(interactions)
Remove the given interactions from the binary quadratic model.

dimod.AdjMapBQM.remove_offset

AdjMapBQM.remove_offset()
Set the binary quadratic model’s offset to zero.

dimod.AdjMapBQM.remove_variable

AdjMapBQM.remove_variable(self, v=None)
Remove a variable and its associated interactions.

Parameters v (variable, optional) – The variable to be removed from the bqm. If not provided, the last variable added is removed.

Returns The removed variable.
Return type variable

Raises
- ValueError – If the binary quadratic model is empty or if v is not a variable.

dimod.AdjMapBQM.remove_variables_from

AdjMapBQM.remove_variables_from(variables)
Remove the given variables from the binary quadratic model.
dimod.AdjMapBQM.update

**AdjMapBQM.update**(*other*)
Update the binary quadratic model, adding biases from another.

dimod.AdjVectorBQM

class AdjVectorBQM(*vartype=None*, **args)**
A binary quadratic model where the neighborhoods are c++ vectors.
Can be created in several ways:

- **AdjVectorBQM(vartype)**: Creates an empty binary quadratic model.
- **AdjVectorBQM(bqm)**: Creates a BQM from another BQM. See copy and cls kwargs below.
- **AdjVectorBQM(bqm, vartype)**: Creates a BQM from another BQM, changing to the appropriate vartype if necessary.
- **AdjVectorBQM(n, vartype)**: Creates a BQM with n variables, indexed linearly from zero, setting all biases to zero.
- **AdjVectorBQM(quadratic, vartype)**: Creates a BQM from quadratic biases given as a square array_like or a dictionary of the form {(*u*, *v*): *b*, . . .}. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.
- **AdjVectorBQM(linear, quadratic, vartype)**: Creates a BQM from linear and quadratic biases, where linear is a one-dimensional array_like or a dictionary of the form {*v*: *b*, . . .}, and quadratic is a square array_like or a dictionary of the form {(*u*, *v*): *b*, . . .}. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.
- **AdjVectorBQM(linear, quadratic, offset, vartype)**: Creates a BQM from linear and quadratic biases, where linear is a one-dimensional array_like or a dictionary of the form {*v*: *b*, . . .}, and quadratic is a square array_like or a dictionary of the form {(*u*, *v*): *b*, . . .}, and offset is a numerical offset. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.

Notes

The AdjVectorBQM is implemented using an adjacency structure where the neighborhoods are implemented as c++ vectors.

Advantages:

- Supports incremental construction
- Fast iteration over the biases

Disadvantages:

- Only supports float64 biases

Intended Use:

- When performance is important and the use case requires incremental construction
- This should be the default BQM type for large problems where arbitrary types are not needed
Examples

```python
>>> import numpy as np
>>> from dimod import AdjVectorBQM

>>> # Construct from dicts
>>> AdjVectorBQM({'a': -1.0}, {('a', 'b'): 1.0}, 'SPIN')
AdjVectorBQM({a: -1.0, b: 0.0}, {('a', 'b'): 1.0}, 0.0, 'SPIN')

>>> # Incremental Construction
>>> bqm = AdjVectorBQM('SPIN')
>>> bqm.add_variable('a')
'a'
>>> bqm.add_variable()
1
>>> bqm.set_quadratic('a', 1, 3.0)
>>> bqm
AdjVectorBQM({a: 0.0, 1: 0.0}, {('a', 1): 3.0}, 0.0, 'SPIN')
```

Attributes

```python
dtype
The data type of the linear biases, float64.

itype
The data type of the indices, uint32.

ntype
The data type of the neighborhood indices, varies by platform.

num_interactions
The number of interactions in the model.

num_variables
The number of variables in the model.

offset
The constant energy offset associated with the model.

shape
A 2-tuple, the num_variables and num_interactions.

variables
The variables of the binary quadratic model.

vartype
The variable type, Vartype.SPIN or Vartype.BINARY.
```

```
dimod.AdjVectorBQM.dtype
AdjVectorBQM.dtype
The data type of the linear biases, float64.

(dimod.AdjVectorBQM.itype
AdjVectorBQM.itype
The data type of the indices, uint32.

(dimod.AdjVectorBQM.ntype
AdjVectorBQM.ntype
The data type of the neighborhood indices, varies by platform.
```
dimod.AdjVectorBQM.num_interactions

AdjVectorBQM.num_interactions
The number of interactions in the model.
    Type  int

dimod.AdjVectorBQM.num_variables

AdjVectorBQM.num_variables
The number of variables in the model.
    Type  int

dimod.AdjVectorBQM.offset

AdjVectorBQM.offset
The constant energy offset associated with the model.

dimod.AdjVectorBQM.shape

AdjVectorBQM.shape
A 2-tuple, the num_variables and num_interactions.

dimod.AdjVectorBQM.variables

AdjVectorBQM.variables
The variables of the binary quadratic model.

dimod.AdjVectorBQM.vartype

AdjVectorBQM.vartype
The variable type, Vartype.SPIN or Vartype.BINARY.

Views

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adj</td>
<td>Quadratic biases as a nested dict of dicts.</td>
</tr>
<tr>
<td>linear</td>
<td>Linear biases as a mapping.</td>
</tr>
<tr>
<td>quadratic</td>
<td>Quadratic biases as a flat mapping.</td>
</tr>
<tr>
<td>binary</td>
<td>The binary-valued version of the binary quadratic model.</td>
</tr>
<tr>
<td>spin</td>
<td>The spin-valued version of the binary quadratic model.</td>
</tr>
</tbody>
</table>

dimod.AdjVectorBQM.adj

AdjVectorBQM.adj
Quadratic biases as a nested dict of dicts.
Accessed like a dict of dicts, where the keys of the outer dict are all of the model’s variables (e.g. \( v \)) and the values are the neighborhood of \( v \). Each neighborhood if a dict where the keys are the neighbors of \( v \) and the values are their associated quadratic biases.

**dimod.AdjVectorBQM.linear**

**AdjVectorBQM.linear**

Linear biases as a mapping.

Accessed like a dict, where keys are the variables of the binary quadratic model and values are the linear biases.

**dimod.AdjVectorBQM.quadratic**

**AdjVectorBQM.quadratic**

Quadratic biases as a flat mapping.

Accessed like a dict, where keys are 2-tuples of variables, which represent an interaction and values are the quadratic biases.

**dimod.AdjVectorBQM.binary**

**AdjVectorBQM.binary**

The binary-valued version of the binary quadratic model.

If the binary quadratic model is binary-valued, this references itself, otherwise it is a BinaryView.

**dimod.AdjVectorBQM.spin**

**AdjVectorBQM.spin**

The spin-valued version of the binary quadratic model.

If the binary quadratic model is spin-valued, this references itself, otherwise it is a SpinView.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_offset(offset)</code></td>
<td>Add specified value to the offset of a binary quadratic model.</td>
</tr>
<tr>
<td><code>add_interaction(u, v, bias)</code></td>
<td>Add an interaction and/or quadratic bias to a binary quadratic model.</td>
</tr>
<tr>
<td><code>add_interactions_from(quadratic)</code></td>
<td>Add interactions and/or quadratic biases to a binary quadratic model.</td>
</tr>
<tr>
<td><code>add_variable(self[, v])</code></td>
<td>Add a variable to the binary quadratic model.</td>
</tr>
<tr>
<td><code>add_variables_from(linear)</code></td>
<td>Add variables and/or linear biases to a binary quadratic model.</td>
</tr>
<tr>
<td><code>change_vartype(self, vartype[, inplace])</code></td>
<td>Return a binary quadratic model with the specified vartype.</td>
</tr>
<tr>
<td><code>contract_variables(u, v)</code></td>
<td>Enforce ( u, v ) being the same variable in a binary quadratic model.</td>
</tr>
<tr>
<td><code>copy([deep])</code></td>
<td>Return a copy.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>degree(self, v)</code></td>
<td>Get the degree of a variable (number of interactions with v)</td>
</tr>
<tr>
<td><code>degrees([array, dtype])</code></td>
<td>Get the degrees of all variables and associate them with a given array of types (dtype)</td>
</tr>
<tr>
<td><code>empty(vartype)</code></td>
<td>Create a new empty binary quadratic model.</td>
</tr>
<tr>
<td><code>energies(self, samples[, dtype])</code></td>
<td>Determine the energies of the given samples.</td>
</tr>
<tr>
<td><code>energy(sample[, dtype])</code></td>
<td>Determine the energy of a given sample.</td>
</tr>
<tr>
<td><code>fix_variables(fixed)</code></td>
<td>Fix the value of the variables and remove them.</td>
</tr>
<tr>
<td><code>flip_variable(v)</code></td>
<td>Flip variable v in a binary quadratic model.</td>
</tr>
<tr>
<td><code>from_coo(obj[, vartype])</code></td>
<td>Deserialize a binary quadratic model from a COOrdinate format encoding.</td>
</tr>
<tr>
<td><code>from_ising(h, J[, offset])</code></td>
<td>Create a binary quadratic model from an Ising problem.</td>
</tr>
<tr>
<td><code>from_networkx_graph(G[, vartype, ...])</code></td>
<td>Create a binary quadratic model from a NetworkX graph.</td>
</tr>
<tr>
<td><code>from_numpy_matrix(mat[, variable_order,...])</code></td>
<td>Create a binary quadratic model from a NumPy array.</td>
</tr>
<tr>
<td><code>from_numpy_vectors(linear, quadratic,...)[]</code></td>
<td>Create a binary quadratic model from vectors.</td>
</tr>
<tr>
<td><code>get_linear(self, v)</code></td>
<td>Get the linear bias of v.</td>
</tr>
<tr>
<td><code>get_quadratic(self, u, v[, default])</code></td>
<td>Get the quadratic bias of (u, v).</td>
</tr>
<tr>
<td><code>has_variable(v)</code></td>
<td>Return True if v is a variable in the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_interactions()</code></td>
<td>Iterate over the interactions of the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_linear(self)</code></td>
<td>Iterate over neighbors of a variable in the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_neighbors(u)</code></td>
<td>Iterate over neighbors of a variable in the binary quadratic model.</td>
</tr>
<tr>
<td><code>iter_quadratic(self[, variables])</code></td>
<td>Iterate over the variables of the binary quadratic model.</td>
</tr>
<tr>
<td><code>normalize([bias_range, quadratic_range,...])</code></td>
<td>Normalize the biases of the binary quadratic model such that they fall in the provided range(s), and adjusts the offset appropriately.</td>
</tr>
<tr>
<td><code>relabel_variables</code></td>
<td>Relabel the variables in the BQM to integers.</td>
</tr>
<tr>
<td><code>relabel_variables_as_integers</code></td>
<td>Relabel the variables in the BQM to integers.</td>
</tr>
<tr>
<td><code>scale(scalar[, ignored_variables,...])</code></td>
<td>Multiply all the biases by the specified scalar.</td>
</tr>
<tr>
<td><code>set_linear(self, v, Bias b)</code></td>
<td>Set the linear bias of a variable v.</td>
</tr>
<tr>
<td><code>set_quadratic(self, u, v, Bias b)</code></td>
<td>Set the quadratic bias of (u, v).</td>
</tr>
<tr>
<td><code>shapeable()</code></td>
<td></td>
</tr>
<tr>
<td><code>to_coo([fp, vartype_header])</code></td>
<td>Serialize the binary quadratic model to a COOrdinate format encoding.</td>
</tr>
<tr>
<td><code>to_ising()</code></td>
<td>Converts a binary quadratic model to Ising format.</td>
</tr>
<tr>
<td><code>to_networkx_graph([node_attribute_name,...])</code></td>
<td>Convert a binary quadratic model to NetworkX graph format.</td>
</tr>
<tr>
<td><code>to_numpy_matrix([variable_order])</code></td>
<td>Convert a binary quadratic model to NumPy 2D array.</td>
</tr>
<tr>
<td><code>to_numpy_vectors(self[, variable_order,...])</code></td>
<td>Convert to numpy vectors.</td>
</tr>
<tr>
<td><code>to_qubo()</code></td>
<td>Convert a binary quadratic model to QUBO format.</td>
</tr>
<tr>
<td><code>remove_interaction(self, u, v)</code></td>
<td>Remove the interaction between variables u and v.</td>
</tr>
<tr>
<td><code>remove_interactions_from(interactions)</code></td>
<td>Remove the given interactions from the binary quadratic model.</td>
</tr>
<tr>
<td><code>remove_offset()</code></td>
<td>Set the binary quadratic model's offset to zero.</td>
</tr>
<tr>
<td><code>remove_variable(self, v)</code></td>
<td>Remove a variable and its associated interactions.</td>
</tr>
<tr>
<td><code>remove_variables_from(variables)</code></td>
<td>Remove the given variables from the binary quadratic model.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 13 – continued from previous page

| **update**<br>(other) | Update the binary quadratic model, adding biases from another. |

**dimod.AdjVectorBQM.add_offset**

`AdjVectorBQM.add_offset(offset)`

Add specified value to the offset of a binary quadratic model.

**dimod.AdjVectorBQM.add_interaction**

`AdjVectorBQM.add_interaction(u, v, bias)`

Add an interaction and/or quadratic bias to a binary quadratic model.

**Parameters**

- **u (variable)** – One of the pair of variables to add to the model. Can be any python object that is a valid dict key.
- **v (variable)** – One of the pair of variables to add to the model. Can be any python object that is a valid dict key.
- **bias (bias)** – Quadratic bias associated with u, v. If u, v is already in the model, this value is added to the current quadratic bias.

**dimod.AdjVectorBQM.add_interactions_from**

`AdjVectorBQM.add_interactions_from(quadratic)`

Add interactions and/or quadratic biases to a binary quadratic model.

**Parameters**

- **quadratic (dict/iterable)** – A collection of interactions and their associated quadratic bias. If a dict, should be of the form `{(u, v): bias, ...}` where u and v are variables in the model and bias is there associated quadratic bias. Otherwise, should be an iterable of `(u, v, bias)` triplets.

**dimod.AdjVectorBQM.add_variable**

`AdjVectorBQM.add_variable(self, v=None, Bias bias=0.0)`

Add a variable to the binary quadratic model.

**Parameters**

- **v (hashable, optional)** – A label for the variable. Defaults to the length of the binary quadratic model, if that label is available. Otherwise defaults to the lowest available positive integer label.
- **bias (numeric, optional, default=0.0)** – The initial bias value for the added variable. If v is already a variable, then bias (if any) is adding to its existing linear bias.

**Returns**

- **The label of the added variable.**

**Return type**

- **hashable**

**Raises**

- **TypeError** – If the label is not hashable.
dimod.AdjVectorBQM.add_variables_from

AdjVectorBQM.add_variables_from(linear)
Add variables and/or linear biases to a binary quadratic model.

Parameters linear (dict/iterable) – A collection of variables in their associated linear biases. If a dict, should be of the form \{v: bias, \ldots\} where v is a variable and bias is its associated linear bias. Otherwise should be an iterable of (v, bias) pairs.

dimod.AdjVectorBQM.change_vartype

AdjVectorBQM.change_vartype (self, vartype, inplace=True)
Return a binary quadratic model with the specified vartype.

Parameters

• vartype (Vartype/str/set, optional) – Variable type for the changed model. Accepted input values:
  – Vartype.SPIN, 'SPIN', {−1, 1}
  – Vartype.BINARY, 'BINARY', {0, 1}
• inplace (bool, optional, default=True) – If True, the binary quadratic model is updated in-place; otherwise, a new binary quadratic model is returned.

Returns A binary quadratic model with the specified vartype.

Return type AdjVectorBQM

dimod.AdjVectorBQM.contract_variables

AdjVectorBQM.contract_variables (u, v)
Enforce u, v being the same variable in a binary quadratic model.

The resulting variable is labeled ‘u’. Values of interactions between v and variables that u interacts with are added to the corresponding interactions of u.

Parameters

• u (variable) – Variable in the binary quadratic model.
• v (variable) – Variable in the binary quadratic model.

dimod.AdjVectorBQM.copy

AdjVectorBQM.copy (deep=False)
Return a copy.

dimod.AdjVectorBQM.degree

AdjVectorBQM.degree (self, v)
dimod.AdjVectorBQM.degrees

AdjVectorBQM.degrees(array=False, dtype=int)

dimod.AdjVectorBQM.empty

classmethod AdjVectorBQM.empty(vartype)
Create a new empty binary quadratic model.

dimod.AdjVectorBQM.energies

AdjVectorBQM.energies(self, samples, dtype=None)
Determine the energies of the given samples.

Parameters

• samples_like(samples_like) – A collection of raw samples. samples_like is an extension of NumPy’s array_like structure. See as_samples().

• dtype(data-type, optional, default=None) – The desired NumPy data type for the energies. Matches dtype by default.

Returns The energies.

Return type numpy.ndarray

dimod.AdjVectorBQM.energy

AdjVectorBQM.energy(sample, dtype=None)

dimod.AdjVectorBQM.fix_variables

AdjVectorBQM.fix_variables(fixed)
Fix the value of the variables and remove them.

Parameters fixed(dict/iterable) – A dictionary or an iterable of 2-tuples of variable assignments.

dimod.AdjVectorBQM.flip_variable

AdjVectorBQM.flip_variable(v)
Flip variable v in a binary quadratic model.

Parameters v(variable) – Variable in the binary quadratic model.

dimod.AdjVectorBQM.from_coo

classmethod AdjVectorBQM.from_coo(obj, vartype=None)
Deserialize a binary quadratic model from a COOrdinate format encoding.

COOrdinate is a sparse encoding for binary quadratic models.
Parameters

- **obj** – (str/file): Either a string or a `.read()`-supporting file object that represents linear and quadratic biases for a binary quadratic model. This data is stored as a list of 3-tuples, \( (i, j, \text{bias}) \), where \( i = j \) for linear biases.
- **vartype** *(Vartype/str/set, optional)* – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN, 'SPIN', \{-1, 1\}
  - Vartype.BINARY, 'BINARY', \{0, 1\}

If not provided, the vartype must be specified with a header in the file.

**Note:** Variables must use index labels (numeric labels). Binary quadratic models created from COOrdinate format encoding have offsets set to zero.

**Note:** This method will be deprecated in the future. The preferred pattern is to use `load()` or `loads()` directly.

dimod.AdjVectorBQM.from_ising

**classmethod** AdjVectorBQM.from_ising(*h, J, offset=0*)

Create a binary quadratic model from an Ising problem.

**Parameters**

- **h** *(dict/list)* – Linear biases of the Ising problem. If a dict, should be of the form \( \{v: \text{bias}, \ldots\} \) where \( v \) is a spin-valued variable and \( \text{bias} \) is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

- **J** *(dict[(variable, variable), bias])* – Quadratic biases of the Ising problem.

- **offset** *(optional, default=0.0)* – Constant offset applied to the model.

**Returns** A spin-valued binary quadratic model.

dimod.AdjVectorBQM.from_networkx_graph

**classmethod** AdjVectorBQM.from_networkx_graph(*G, vartype=None, node_attribute_name='bias', edge_attribute_name='bias)*

Create a binary quadratic model from a NetworkX graph.

**Parameters**

- **G** *(networkx.Graph)* – A NetworkX graph with biases stored as node/edge attributes.

- **vartype** *(Vartype/str/set, optional)* – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN, 'SPIN', \{-1, 1\}
  - Vartype.BINARY, 'BINARY', \{0, 1\}
If not provided, the $G$ should have a vartype attribute. If vartype is provided and $G$.vartype exists then the argument overrides the property.

- **node_attribute_name** (hashable, optional, default='bias') – Attribute name for linear biases. If the node does not have a matching attribute then the bias defaults to 0.
- **edge_attribute_name** (hashable, optional, default='bias') – Attribute name for quadratic biases. If the edge does not have a matching attribute then the bias defaults to 0.

**Returns** Binary quadratic model

**Note:** This method will be deprecated in the future. The preferred pattern is to use the `from_networkx_graph()` function.

dimod.AdjVectorBQM.from_numpy_matrix
classmethod AdjVectorBQM.from_numpy_matrix(mat, variable_order=None, offset=0.0, interactions=None)
Create a binary quadratic model from a NumPy array.

**Parameters**

- **mat** (numpy.ndarray) – Coefficients of a quadratic unconstrained binary optimization (QUBO) model formatted as a square NumPy 2D array.
- **variable_order** (list, optional) – If provided, labels the QUBO variables; otherwise, row/column indices are used. If variable_order is longer than the array, extra values are ignored.
- **offset** (optional, default=0.0) – Constant offset for the binary quadratic model.
- **interactions** (iterable, optional, default=[]) – Any additional 0.0-bias interactions to be added to the binary quadratic model. Only works for shapeable binary quadratic models.

**Returns** Binary quadratic model with vartype set to Vartype.BINARY.

**Note:** This method will be deprecated in the future. The preferred pattern is to use the constructor directly.

dimod.AdjVectorBQM.from_numpy_vectors
classmethod AdjVectorBQM.from_numpy_vectors(linear, quadratic, offset, vartype, variable_order=None)
Create a binary quadratic model from vectors.

**Parameters**

- **linear** (array_like) – A 1D array-like iterable of linear biases.
- **quadratic** (tuple[array_like, array_like, array_like]) – A 3-tuple of 1D array_like vectors of the form (row, col, bias).
- **offset** (numeric, optional) – Constant offset for the binary quadratic model.
• **vartype** *(Vartype/str/set)* – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}

• **variable_order** *(iterable, optional)* – If provided, labels the variables; otherwise, indices are used.

**Returns** A binary quadratic model

**dimod.AdjVectorBQM.from_qubo**

*classmethod* AdjVectorBQM.from_qubo(Q, offset=0)  
Create a binary quadratic model from a QUBO problem.

**Parameters**

- **Q** *(dict)* – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v` are binary-valued variables and `bias` is their associated coefficient.

- **offset** *(optional, default=0.0)* – Constant offset applied to the model.

**Returns** A binary-valued binary quadratic model.

**dimod.AdjVectorBQM.get_linear**

AdjVectorBQM.get_linear(self, v)  
Get the linear bias of `v`.

**Parameters** **v** *(hashable)* – A variable in the binary quadratic model.

**Returns** The linear bias of `v`.

**Return type** float

**Raises** ValueError – If `v` is not a variable in the binary quadratic model.

**dimod.AdjVectorBQM.get_quadratic**

AdjVectorBQM.get_quadratic(self, u, v, default=None)  
Get the quadratic bias of `(u, v)`.

**Parameters**

- **u** *(hashable)* – A variable in the binary quadratic model.

- **v** *(hashable)* – A variable in the binary quadratic model.

- **default** *(number, optional)* – Value to return if there is no interactions between `u` and `v`.

**Returns** The quadratic bias of `(u, v)`.

**Raises**

- **ValueError** – If either `u` or `v` is not a variable in the binary quadratic model or if `u == v`
• `ValueError` – If (u, v) is not an interaction and `default` is `None`.

`dimod.AdjVectorBQM.has_variable`

`AdjVectorBQM.has_variable(v)`

Return `True` if `v` is a variable in the binary quadratic model.

`dimod.AdjVectorBQM.iter_interactions`

`AdjVectorBQM.iter_interactions()`

Iterate over the interactions of the binary quadratic model.

Yields `interaction` – An interaction in the binary quadratic model.

`dimod.AdjVectorBQM.iter_linear`

`AdjVectorBQM.iter_linear(self)`

`dimod.AdjVectorBQM.iter_neighbors`

`AdjVectorBQM.iter_neighbors(u)`

Iterate over neighbors of a variable in the binary quadratic model.

Yields `variable` – The neighbors of `v`.

`dimod.AdjVectorBQM.iter_quadratic`

`AdjVectorBQM.iter_quadratic(self, variables=[])`

`dimod.AdjVectorBQM.iter_variables`

`AdjVectorBQM.iter_variables()`

Iterate over the variables of the binary quadratic model.

Yields `hashable` – A variable in the binary quadratic model.

`dimod.AdjVectorBQM.normalize`

`AdjVectorBQM.normalize(bias_range=1, quadratic_range=None, ignored_variables=None, ignored_interactions=None, ignore_offset=False)`

Normalizes the biases of the binary quadratic model such that they fall in the provided range(s), and adjusts the offset appropriately.

If `quadratic_range` is provided, then `bias_range` will be treated as the range for the linear biases and `quadratic_range` will be used for the range of the quadratic biases.

Parameters
• **bias_range** (*number/pair*) – Value/range that the biases of the BQM will be scaled to fit within. If **quadratic_range** is provided, this range is used to fit the linear biases.

• **quadratic_range** (*number/pair*) – The BQM will be scaled so that the quadratic biases fit within this range.

• **ignored_variables** (*iterable, optional*) – Biases associated with these variables are not scaled.

• **ignored_interactions** (*iterable[tuple], optional*) – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.

• **ignore_offset** (*bool, default=False*) – If True, the offset is not scaled.

**dimod.AdjVectorBQM.relabel_variables**

**AdjVectorBQM.relabel_variables**

**dimod.AdjVectorBQM.relabel_variables_as_integers**

**AdjVectorBQM.relabel_variables_as_integers**
Relabel the variables in the BQM to integers.
Note that this function uses the natural labelling of the underlying c++ objects.

**dimod.AdjVectorBQM.scale**

**AdjVectorBQM.scale** (*scalar, ignored_variables=None, ignored_interactions=None, ignore_offset=False*)
Multiply all the biases by the specified scalar.

**Parameters**

• **scalar** (*number*) – Value by which to scale the energy range of the binary quadratic model.

• **ignored_variables** (*iterable, optional*) – Biases associated with these variables are not scaled.

• **ignored_interactions** (*iterable[tuple], optional*) – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.

• **ignore_offset** (*bool, default=False*) – If True, the offset is not scaled.

**dimod.AdjVectorBQM.set_linear**

**AdjVectorBQM.set_linear** (*self, v, Bias b*)
Set the linear bias of a variable v.

**Parameters**

• **v** (*hashable*) – A variable in the binary quadratic model. It is added if not already in the model.

• **b** (*numeric*) – The linear bias of v.

**Raises** **TypeError** – If v is not hashable
dimod.AdjVectorBQM.set_quadratic

adjvectorBQM.set_quadratic(self, u, v, Bias b)

Set the quadratic bias of (u, v).

Parameters

• u (hashable) – A variable in the binary quadratic model.
• v (hashable) – A variable in the binary quadratic model.
• b (numeric) – The linear bias of v.

Raises TypeError – If u or v is not hashable.

dimod.AdjVectorBQM.shapeable

classmethod AdjVectorBQM.shapeable()

dimod.AdjVectorBQM.to_coo

adjvectorBQM.to_coo(fp=None, vartype_header=False)

Serialize the binary quadratic model to a COOrdinate format encoding.

COOrdinate is a sparse encoding for binary quadratic models.

Parameters

• fp (file, optional) – A write()-supporting file object to save the linear and quadratic biases of a binary quadratic model to. The model is stored as a list of 3-tuples, (i, j, bias), where \(i = j\) for linear biases. If not provided, returns a string.
• vartype_header (bool, optional, default=False) – If true, the binary quadratic model’s variable type as prepended to the string or file as a header.

Note: Variables must use index labels (numeric labels). Binary quadratic models saved to COOrdinate format encoding do not preserve offsets.

Note: This method will be deprecated in the future. The preferred pattern is to use dump() or dumps() directly.

dimod.AdjVectorBQM.to_ising

adjvectorBQM.to_ising()

Converts a binary quadratic model to Ising format.

If the binary quadratic model’s vartype is not Vartype.SPIN, values are converted.

Returns 3-tuple of form (linear, quadratic, offset), where linear is a dict of linear biases, quadratic is a dict of quadratic biases, and offset is a number that represents the constant offset of the binary quadratic model.

Return type tuple
dimod.AdjVectorBQM.to_networkx_graph

AdjVectorBQM.to_networkx_graph(node_attribute_name='bias', edge_attribute_name='bias')
Convert a binary quadratic model to NetworkX graph format.

Parameters

- **node_attribute_name** (hashable, optional, default='bias') – Attribute name for linear biases.
- **edge_attribute_name** (hashable, optional, default='bias') – Attribute name for quadratic biases.

Returns
A NetworkX graph with biases stored as node/edge attributes.

Return type networkx.Graph

Note: This method will be deprecated in the future. The preferred pattern is to use to_networkx_graph().

dimod.AdjVectorBQM.to_numpy_matrix

AdjVectorBQM.to_numpy_matrix(variable_order=None)
Convert a binary quadratic model to NumPy 2D array.

Parameters

- **variable_order** (list, optional) – If provided, indexes the rows/columns of the NumPy array. If variable_order includes any variables not in the binary quadratic model, these are added to the NumPy array.

Returns
The binary quadratic model as a NumPy 2D array. Note that the binary quadratic model is converted to BINARY vartype.

Return type numpy.ndarray

Note: This method will be deprecated in the future. The preferred pattern is to use to_dense().

dimod.AdjVectorBQM.to_numpy_vectors

AdjVectorBQM.to_numpy_vectors(self, variable_order=None, dtype=None, index_dtype=None, sort_indices=False, sort_labels=True, return_labels=False)
Convert to numpy vectors.

dimod.AdjVectorBQM.to_qubo

AdjVectorBQM.to_qubo()
Convert a binary quadratic model to QUBO format.

If the binary quadratic model’s vartype is not Vartype.BINARY, values are converted.

Returns 2-tuple of form (biases, offset), where biases is a dict in which keys are pairs of variables and values are the associated linear or quadratic bias and offset is a number that represents the constant offset of the binary quadratic model.

Return type tuple
dimod.AdjVectorBQM.remove_interaction

AdjVectorBQM.remove_interaction(self, u, v)
    Remove the interaction between variables u and v.

    Parameters
    • u (hashable) – A variable in the binary quadratic model.
    • v (hashable) – A variable in the binary quadratic model.

    Returns
    If there was an interaction to remove.

    Return type  bool

    Raises
    • ValueError – If either u or v is not a variable in the binary
      quadratic model.

dimod.AdjVectorBQM.remove_interactions_from

AdjVectorBQM.remove_interactions_from(interactions)
    Remove the given interactions from the binary quadratic model.

dimod.AdjVectorBQM.remove_offset

AdjVectorBQM.remove_offset()
    Set the binary quadratic model’s offset to zero.

dimod.AdjVectorBQM.remove_variable

AdjVectorBQM.remove_variable(self, v=None)
    Remove a variable and its associated interactions.

    Parameters
    • v (variable, optional) – The variable to be removed from the bqm. If not
      provided, the last variable added is removed.

    Returns
    The removed variable.

    Return type  variable

    Raises
    • ValueError – If the binary quadratic model is empty or if v is not
      a variable.

dimod.AdjVectorBQM.remove_variables_from

AdjVectorBQM.remove_variables_from(variables)
    Remove the given variables from the binary quadratic model.
dimod.AdjVectorBQM.update

AdjVectorBQM.update(other)
Update the binary quadratic model, adding biases from another.

dimod.BinaryQuadraticModel

class BinaryQuadraticModel(*args, **kwargs)
Encodes a binary quadratic model.

Binary quadratic model is the superclass that contains the Ising model and the QUBO.

Parameters

- **linear** (dict [variable, bias]) – Linear biases as a dict, where keys are the variables of the binary quadratic model and values the linear biases associated with these variables. A variable can be any python object that is valid as a dictionary key. Biases are generally numbers but this is not explicitly checked.

- **quadratic** (dict [(variable, variable), bias]) – Quadratic biases as a dict, where keys are 2-tuples of variables and values the quadratic biases associated with the pair of variables (the interaction). A variable can be any python object that is valid as a dictionary key. Biases are generally numbers but this is not explicitly checked. Interactions that are not unique are added.

- **offset** (number) – Constant energy offset associated with the binary quadratic model. Any input type is allowed, but many applications assume that offset is a number. See BinaryQuadraticModel.energy().

- **vartype** (Vartype/str/set) – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}

- ****kwargs – Any additional keyword parameters and their values are stored in BinaryQuadraticModel.info.

Notes

The BinaryQuadraticModel class does not enforce types on biases and offsets, but most applications that use this class assume that they are numeric.

Examples

This example creates a binary quadratic model with three spin variables.

```python
>>> bqm = dimod.BinaryQuadraticModel({0: 1, 1: -1, 2: .5},
... {(0, 1): .5, (1, 2): 1.5},
... 1.4,
... dimod.Vartype.SPIN)
```

This example creates a binary quadratic model with non-numeric variables (variables can be any hashable object).
Linear biases as a dict, where keys are the variables of the binary quadratic model and values the linear biases associated with these variables.

    Type  dict[variable, bias]

Quadratic biases as a dict, where keys are 2-tuples of variables, which represent an interaction between the two variables, and values are the quadratic biases associated with the interactions.

    Type  dict[(variable, variable), bias]

The energy offset associated with the model. Same type as given on instantiation.

    Type  number

The model’s type. One of Vartype.SPIN or Vartype.BINARY.

    Type  Vartype

The variables in the binary quadratic model as a dictionary keys view object.

    Type  keysview

The model’s interactions as nested dicts. In graphic representation, where variables are nodes and interactions are edges or adjacencies, keys of the outer dict (adj) are all the model’s nodes (e.g. v) and values are the inner dicts. For the inner dict associated with outer-key/node ‘v’, keys are all the nodes adjacent to v (e.g. u) and values are quadratic biases associated with the pair of inner and outer keys (u, v).

    Type  dict

A place to store miscellaneous data about the binary quadratic model as a whole.

    Type  dict

An alias of Vartype.SPIN for easier access.

    Type  Vartype

An alias of Vartype.BINARY for easier access.

    Type  Vartype
**Examples**

This example creates an instance of the \texttt{BinaryQuadraticModel} class for the K4 complete graph, where the nodes have biases set equal to their sequential labels and interactions are the concatenations of the node pairs (e.g., 23 for \(u,v = 2,3\)).

```python
>>> linear = {1: 1, 2: 2, 3: 3, 4: 4}
>>> quadratic = {(1, 2): 12, (1, 3): 13, (1, 4): 14,
...              (2, 3): 23, (2, 4): 24,
...              (3, 4): 34}
>>> offset = 0.0
>>> vartype = dimod.BINARY
>>> bqm_k4 = dimod.BinaryQuadraticModel(linear, quadratic, offset, vartype)

```

- **BinaryQuadraticModel(*args, **kwargs)**: Encodes a binary quadratic model.
- **AdjArrayBQM([vartype])**: A binary quadratic model structured as two c++ vectors.
- **AdjDictBQM(*args[, vartype])**: A binary quadratic model structured as a dict-of-dicts.
- **AdjMapBQM([vartype])**: A binary quadratic model where the neighborhoods are c++ maps.
- **AdjVectorBQM([vartype])**: A binary quadratic model where the neighborhoods are c++ vectors.

**Functions**

Generic constructor:

```python
as_bqm(*args[, cls, copy])
```

Convert the input to a binary quadratic model.

**dimod.as_bqm**

```python
as_bqm(*args, cls=None, copy=False)
```

Convert the input to a binary quadratic model.

Converts the following input formats to a binary quadratic model (BQM):

- **as_bqm(vartype)**: Creates an empty binary quadratic model.
- **as_bqm(bqm)**: Creates a BQM from another BQM. See \texttt{copy} and \texttt{cls} kwargs below.
- **as_bqm(bqm, vartype)**: Creates a BQM from another BQM, changing to the appropriate \texttt{vartype} if necessary. See \texttt{copy} and \texttt{cls} kwargs below.
- **as_bqm(n, vartype)**: Creates a BQM with \(n\) variables, indexed linearly from zero, setting all biases to zero.
- **as_bqm(quadratic, vartype)**: Creates a BQM from quadratic biases given as a square \texttt{array_like} or a dictionary of the form \(\{ (u, v): b, \ldots \}\). Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.
- **as_bqm(linear, quadratic, vartype)**: Creates a BQM from linear and quadratic biases, where \texttt{linear} is a one-dimensional \texttt{array_like} or a dictionary of the form \(\{ v: b, \ldots \}\), and \texttt{quadratic} is a square matrix.
array_like or a dictionary of the form \{(u, v): b, \ldots\}. Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.

as_bqm(linear, quadratic, offset, vartype) Creates a BQM from linear and quadratic biases, where
linear is a one-dimensional array_like or a dictionary of the form \{v: b, \ldots\}, and quadratic is
a square array_like or a dictionary of the form \{(u, v): b, \ldots\}, and offset is a numerical offset.
Note that when formed with SPIN-variables, biases on the diagonal are added to the offset.

Parameters

• *args – See above.
• cls (type/list, optional, default=:class:AdjVectorBQM) – Class of the returned BQM. If
given as a list, the returned BQM is of one of the types in the list.
• copy (bool, optional, default=False) – If False, a new BQM is only con-
structed when necessary.

Returns A binary quadratic model.

Generating BQMs:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chimera_anticluster(m[, n, t, multiplier, ...])</td>
<td>Generate an anticluster problem on a Chimera lattice.</td>
</tr>
</tbody>
</table>
| combinations(n, k[, strength, vartype]) | Generate a BQM that is minimized when k of n vari-
| frustrated_loop(graph, num_cycles[, R, ...]) | Generate a frustrated loop problem. |
| randint(graph, vartype[, low, high, cls, seed]) | Generate a binary quadratic model with random biases
| ran_r(r, graph[, cls, seed]) | Generate an Ising model for a RANr problem. |
| uniform(graph, vartype[, low, high, cls, seed]) | Generate a binary quadratic model with random biases
| and offset. |

dimod.generators.chimera_anticluster

chimera_anticluster (m, n=None, t=4, multiplier=3.0, cls=<class 'di-
mod.binary_quadratic_model.BinaryQuadraticModel'>, subgraph=None,
seed=None) Generate an anticluster problem on a Chimera lattice.

An anticluster problem has weak interactions within a tile and strong interactions between tiles.

Parameters

• m (int) – Number of rows in the Chimera lattice.
• n (int, optional, default=m) – Number of columns in the Chimera lattice.
• t (int, optional, default=t) – Size of the shore within each Chimera tile.
• multiplier (number, optional, default=3.0) – Strength of the intertile
| edges. |
• cls (class, optional, default=:class:BinaryQuadraticModel) – Binary quadratic model
class to build from.
• subgraph (int/tuple[nodes, edges]/list[edge]/Graph) – A subgraph of a Chimera(m, n, t)
| graph to build the anticluster problem on. |
• seed (int, optional, default=None) – Random seed.
**Returns** spin-valued binary quadratic model.

**Return type** *BinaryQuadraticModel*

**dimod.generators.combinations**

**combinations** \((n, k, \text{strength}=1, \text{vartype}=<\text{Vartype.BINARY}: \text{frozenset}(\{0, 1\})>)\)

Generate a BQM that is minimized when *k* of *n* variables are selected.

More fully, we wish to generate a binary quadratic model (BQM) which is minimized for each of the *k*-combinations of its variables.

The energy for the BQM is given by \((\sum_i x_i - k)^2\).

**Parameters**

- \(n\ (\text{int/list/set})\) – If *n* is an integer, variables are labelled \([0, n-1]\). If *n* is list or set then the variables are labelled accordingly.
- \(k\ (\text{int})\) – The generated BQM will have 0 energy when any *k* of the variables are 1.
- \(\text{strength}\ (\text{number}, \text{optional}, \text{default}=1)\) – The energy of the first excited state of the BQM.
- \(\text{vartype}\ (\text{Vartype/str/set})\) – Variable type for the BQM. Accepted input values:
  - Vartype.SPIN, 'SPIN', \([-1, 1]\)
  - Vartype.BINARY, 'BINARY', \([0, 1]\)

**Returns** *BinaryQuadraticModel*

**Examples**

```python
>>> bqm = dimod.generators.combinations(['a', 'b', 'c'], 2)
>>> bqm.energy({'a': 1, 'b': 0, 'c': 1})
0.0
>>> bqm.energy({'a': 1, 'b': 1, 'c': 1})
1.0

>>> bqm = dimod.generators.combinations(5, 1)
>>> bqm.energy({0: 0, 1: 0, 2: 1, 3: 0, 4: 0})
0.0
>>> bqm.energy({0: 0, 1: 0, 2: 1, 3: 1, 4: 0})
1.0

>>> bqm = dimod.generators.combinations(['a', 'b', 'c'], 2, strength=3.0)
>>> bqm.energy({'a': 1, 'b': 0, 'c': 1})
0.0
>>> bqm.energy({'a': 1, 'b': 1, 'c': 1})
3.0
```

**dimod.generators.frustrated_loop**

**frustrated_loop** \((\text{graph}, \ \text{num_cycles}, \ R=\text{inf}, \ \text{cycle predicates}=(), \ \text{max failed cycles}=100, \ \text{planted solution}=\text{None}, \ \text{seed}=\text{None})\)

Generate a frustrated loop problem.
A (generic) frustrated loop (FL) problem is a sum of Hamiltonians, each generated from a single “good” loop.
1. Generate a loop by random walking on the support graph. 2. If the cycle is “good” (according to provided predicates), continue, else go to 1. 3. Choose one edge of the loop to be anti-ferromagnetic; all other edges are ferromagnetic. 4. Add the loop’s coupler values to the FL problem. If at any time the magnitude of a coupler in the FL problem exceeds a given precision $R$, remove that coupler from consideration in the loop generation procedure.

This is a generic generator of FL problems that encompasses both the original FL problem definition from\(^1\) and the limited FL problem definition from\(^2\)

**Parameters**
- **graph** (int/tuple[nodes, edges]/list[edge]/Graph) – The graph to build the frustrated loops on. Either an integer $n$, interpreted as a complete graph of size $n$, a nodes/edges pair, a list of edges or a NetworkX graph.
- **num_cyles** (int) – Desired number of frustrated cycles.
- **R** (int, optional, default=\(\infty\)) – Maximum interaction weight.
- **cycle_predicates** (tuple[function], optional) – An iterable of functions, which should accept a cycle and return a bool.
- **max_failed_cycles** (int, optional, default=100) – Maximum number of failures to find a cycle before terminating.
- **planted_solution** (dict, optional, default=None) – Other solutions with the same energy may exist, but the energy value of the (possibly degenerate) ground state will hold. If None, planted_solution is: \{v:-1 for v in graph\}
- **seed** (int, optional, default=None) – Random seed.

---


• **cls** (*BinaryQuadraticModel*) – BQM class to build from.
• **seed** (*int, optional, default=None*) – Random seed.

**Returns** *BinaryQuadraticModel*

dimod.generators.ran_r

**ran_r** (*r, graph, cls=*dimod.binary_quadratic_model.BinaryQuadraticModel*, *seed=None*)

Generate an Ising model for a RANr problem.

In RANr problems all linear biases are zero and quadratic values are uniformly selected integers between -r to r, excluding zero. This class of problems is relevant for binary quadratic models (BQM) with spin variables (Ising models).

This generator of RANr problems follows the definition in [Kin2015].

**Parameters**

- **r** (*int*) – Order of the RANr problem.
- **graph** (int/tuple[nodes, edges]/list[edge]/Graph) – The graph to build the bqm on. Either an integer n, interpreted as a complete graph of size n, a nodes/edges pair, a list of edges or a NetworkX graph.
- **cls** (*BinaryQuadraticModel*) – Binary quadratic model class to build from.
- **seed** (*int, optional, default=None*) – Random seed.

**Returns** *BinaryQuadraticModel*.

**Examples:**

```python
>>> import networkx as nx
>>> K_7 = nx.complete_graph(7)
>>> bqm = dimod.generators.random.ran_r(1, K_7)
>>> max(bqm.quadratic.values()) == -min(bqm.quadratic.values())
True
```

dimod.generators.uniform

**uniform** (*graph, vartype, low=0.0, high=1.0, cls=*dimod.binary_quadratic_model.BinaryQuadraticModel*, *seed=None*)

Generate a binary quadratic model with random biases and offset.

Biases and offset are drawn from a uniform distribution range (low, high).

**Parameters**

- **graph** (int/tuple[nodes, edges]/list[edge]/Graph) – The graph to build the binary quadratic model (BQM) on. Either an integer n, interpreted as a complete graph of size n, a nodes/edges pair, a list of edges or a NetworkX graph.
- **vartype** (*Vartype/str/set*) – Variable type for the BQM. Accepted input values:
  - *Vartype.SPIN, 'SPIN', {-1, 1}*
  - *Vartype.BINARY, 'BINARY', {0, 1}*
- **low** (*float, optional, default=0.0*) – The low end of the range for the random biases.
Fixing variables:

```python
fix_variables(bqm[, sampling_mode])
```

Determine assignments for some variables of a binary quadratic model.

**Parameters**

- **bqm** (*BinaryQuadraticModel*) – A binary quadratic model.
- **sampling_mode** (*bool*, optional, *default=True*) – In sampling mode, only roof-duality is used. When `sampling_mode` is false, strongly connected components are used to fix more variables, but in some optimal solutions these variables may take different values.

**Returns** Variable assignments for some variables of the specified binary quadratic model.

**Return type** *dict*

**Examples**

This example creates a binary quadratic model with a single ground state and fixes the model’s single variable to the minimizing assignment.

```python
>>> bqm = dimod.BinaryQuadraticModel.from_ising({ 'a': 1.0 }, {})
>>> dimod.fix_variables(bqm)
{'a': -1}
```

This example has two ground states, $a = b = -1$ and $a = b = 1$, with no variable having a single value for all ground states, so neither variable is fixed.

```python
>>> bqm = dimod.BinaryQuadraticModel.empty(dimod.SPIN)
>>> bqm.add_interaction('a', 'b', -1.0)
>>> dimod.fix_variables(bqm)  # doctest: +SKIP
[]
```

This example turns sampling model off, so variables are fixed to an assignment that attains the ground state.

```python
>>> bqm = dimod.BinaryQuadraticModel.from_ising({'a': 1.0}, {})
```

- **high** (*float*, optional, *default=1.0*) – The high end of the range for the random biases.
- **cls** (*BinaryQuadraticModel*) – BQM class to build from.
- **seed** (*int*, optional, *default=None*) – Random seed.

**Returns** *BinaryQuadraticModel*
```python
>>> bqm = dimod.BinaryQuadraticModel.empty(dimod.SPIN)
>>> bqm.add_interaction('a', 'b', -1.0)
>>> dimod.fix_variables(bqm, sampling_mode=False)  # doctest: +SKIP
{'a': 1, 'b': 1}
```

Traversing as a graph:

```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connected_components(bqm)</code></td>
<td>Yields sets of connected variables.</td>
</tr>
<tr>
<td><code>bfs_variables(bqm, source)</code></td>
<td>Yields variables in breadth-first search order.</td>
</tr>
</tbody>
</table>
```

**dimod.traversal.connected_components**

`connected_components(bqm)`

Yields sets of connected variables.

**Parameters**

- `bqm (dimod.BinaryQuadraticModel)` – A binary quadratic model (BQM).

**Yields**

- `set` – A set of variables in the BQM that form a connected component.

**dimod.traversal.bfs_variables**

`bfs_variables(bqm, source)`

Yields variables in breadth-first search order.

**Parameters**

- `bqm (dimod.BinaryQuadraticModel)` – A binary quadratic model.
- `source (variable)` – A variable in the binary quadratic model (BQM).

**Yields**

- `variable` – variables in the BQM, yielded in breadth-first search order starting at source.

Converting to and from other data structures:

```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_networkx_graph(bqm[, ...])</code></td>
<td>Convert a binary quadratic model to NetworkX graph format.</td>
</tr>
<tr>
<td><code>from_networkx_graph(G[, vartype, ...])</code></td>
<td>Create a binary quadratic model from a NetworkX graph.</td>
</tr>
</tbody>
</table>
```

**dimod.to_networkx_graph**

`to_networkx_graph(bqm, node_attribute_name='bias', edge_attribute_name='bias')`

Convert a binary quadratic model to NetworkX graph format.

**Parameters**

- `node_attribute_name (hashable, optional, default='bias')` – Attribute name for linear biases.
- `edge_attribute_name (hashable, optional, default='bias')` – Attribute name for quadratic biases.

**Returns**

- A NetworkX graph with biases stored as node/edge attributes.

**Return type**

`networkx.Graph`
**dimod.from_networkx_graph**

`from_networkx_graph(G, vartype=None, node_attribute_name='bias', edge_attribute_name='bias', cls=<class 'dimod.bqm.adjdictbqm.AdjDictBQM'>)`

Create a binary quadratic model from a NetworkX graph.

**Parameters**

- `G` (*networkx.Graph*) – A NetworkX graph with biases stored as node/edge attributes.
- `vartype` (*Vartype/str/set, optional*) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, 'SPIN', {-1, 1}
  - `Vartype.BINARY`, 'BINARY', {0, 1}
  If not provided, the `G` should have a vartype attribute. If `vartype` is provided and `G.vartype` exists then the argument overrides the property.
- `node_attribute_name` (*hashable, optional, default='bias'*) – Attribute name for linear biases. If the node does not have a matching attribute then the bias defaults to 0.
- `edge_attribute_name` (*hashable, optional, default='bias'*) – Attribute name for quadratic biases. If the edge does not have a matching attribute then the bias defaults to 0.

**Returns** A binary quadratic model of type `cls`.

See also: serialization functions

**Usage**

In Ocean, there are four objects that represent BQMs, differentiated by the data structure used to encode their structure and biases.

- `AdjArrayBQM`: Uses c++ vectors as arrays
- `AdjDictBQM`: Uses python dictionaries
- `AdjMapBQM`: Uses c++ maps
- `AdjVectorBQM`: Uses c++ vectors

The documentation for each class outlines some of the advantages and disadvantages of the different representations.

All of the BQM types use an adjacency structure, in which each variable tracks its own linear bias and its neighborhood. For instance, given a BQM,

\[ E(x) = 0.5x_0 - 3x_1 - x_0x_1 + x_0x_2 + 2x_0x_3 + x_2x_3 \]

its graph and adjacency representations are

The performance of various operations will depend on which binary quadratic model implementation you are using. Let \( n \) be the number of variables in the BQM.
Fig. 1: The adjacency structure of a 4-variable binary quadratic model.

Table 20: Complexity of various operations

<table>
<thead>
<tr>
<th></th>
<th>AdjArrayBQM</th>
<th>AdjDictBQM</th>
<th>AdjMapBQM</th>
<th>AdjVectorBQM</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_variable</td>
<td>n/a</td>
<td>O(1)(^1)</td>
<td>O(1)(^2)</td>
<td>O(1)(^2)</td>
</tr>
<tr>
<td>add_interaction</td>
<td>n/a</td>
<td>O(1)(^1)</td>
<td>O(\log v)</td>
<td>O(v)</td>
</tr>
<tr>
<td>get_linear</td>
<td>O(1)</td>
<td>O(1)(^1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>get_quadratic</td>
<td>O(\log v)</td>
<td>O(1)(^3)</td>
<td>O(\log v)</td>
<td>O(\log v)</td>
</tr>
<tr>
<td>num_variables</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>num_interactions</td>
<td>O(v)</td>
<td>O(v)</td>
<td>O(v)</td>
<td>O(v)</td>
</tr>
</tbody>
</table>

It is worth noting that although the AdjDictBQM is superior in terms of complexity, in practice it is much slower for large BQMs.

**Samplers and Composites**

**Samplers**

The *dimod* package includes several example samplers.

---

1 Average case, amortized worst case is O(v)
2 Amortized
3 Average case, amortized worst case is O(v^2)
Exact Solver

A simple exact solver for testing and debugging code using your local CPU.

**Note:** This sampler is designed for use in testing. Because it calculates the energy for every possible sample, it is very slow.

**Class**

```python
class ExactSolver
A simple exact solver for testing and debugging code using your local CPU.
```

**Notes**

This solver becomes slow for problems with 18 or more variables.

**Examples**

This example solves a two-variable Ising model.

```python
>>> h = {'a': -0.5, 'b': 1.0}
>>> J = {('a', 'b'): -1.5}
>>> sampleset = dimod.ExactSolver().sample_ising(h, J)
>>> print(sampleset)  # doctest: +SKIP
a b energy num_occ.
```

(continues on next page)
This example solves a two-variable QUBO.

```python
>>> Q = {('a', 'b'): 2.0, ('a', 'a'): 1.0, ('b', 'b'): -0.5}
>>> sampleset = dimod.ExactSolver().sample_qubo(Q)
>>> sampleset.first.sample
{'a': 0, 'b': 1}
```

This example solves a two-variable binary quadratic model.

```python
>>> bqm = dimod.BinaryQuadraticModel({'a': 1.5}, {('a', 'b'): -1}, 0.0, 'SPIN')
>>> sampleset = dimod.ExactSolver().sample(bqm)
>>> sampleset.first.energy
-2.5
```

## Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ExactSolver.sample(bqm)</code></td>
<td>Sample from a binary quadratic model.</td>
</tr>
<tr>
<td><code>ExactSolver.sample_ising(h, J, **parameters)</code></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><code>ExactSolver.sample_qubo(Q, **parameters)</code></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

### `dimod.reference.samplers.ExactSolver.sample`

`ExactSolver.sample(bqm)`

Sample from a binary quadratic model.

**Parameters**

- `bqm` (*BinaryQuadraticModel*) – Binary quadratic model to be sampled from.

**Returns**

`SampleSet`

### `dimod.reference.samplers.ExactSolver.sample_ising`

`ExactSolver.sample_ising(h, J, **parameters)`

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

- `h` (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

• **kwargs – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also:
sample(), sample_qubo()

dimod.reference.samplers.ExactSolver.sample_qubo

ExactSolver.sample_qubo(Q, **parameters)
Sample from a QUBO using the implemented sample method.
This method is inherited from the Sampler base class.
Converts the QUBO into a BinaryQuadraticModel and then calls sample().

Parameters

• Q(dict) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem.
  Should be a dict of the form {(u, v): bias, ...} where u, v, are binary-valued variables and
  bias is their associated coefficient.

• **kwargs – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also:
sample(), sample_ising()

Identity Sampler

A sampler that returns the provided initial states.

Class

class IdentitySampler
  A sampler that returns the provided initial states.

Examples:

```python
>>> samples = [{'a': -1, 'b': +1}, {'a': +1, 'b': +1}]
>>> Q = {('a', 'b'): -1}
>>> sampler = dimod.IdentitySampler()
>>> sampleset = sampler.sample_qubo(Q, initial_states=samples)
>>> print(sampleset)
  a  b  energy  num_oc.
1  1  1 -1.0  1
0  0  1  0.0  1
['BINARY', 2 rows, 2 samples, 2 variables]
```

Properties
IdentitySampler.parameters

Keyword arguments accepted by the sampling methods.

IdentitySampler.parameters = {'initial_states': [], 'initial_states_generator': [], 'num_reads': [], 'seed': []}

Contents are exactly {'initial_states': [], 'initial_states_generator': [], 'num_reads': [], 'seed': []}

Type dict

Methods

IdentitySampler.sample(bqm, *args, **kwargs)
Return exactly the provided initial states.

IdentitySampler.sample_ising(h, J, **parameters)
Sample from an Ising model using the implemented sample method.

IdentitySampler.sample_qubo(Q, **parameters)
Sample from a QUBO using the implemented sample method.

dimod.reference.samplers.identity_sampler.IdentitySampler.sample

IdentitySampler.sample(bqm, *args, **kwargs)
Return exactly the provided initial states.

Parameters

- **bqm** (BinaryQuadraticModel) – The binary quadratic model to be sampled.
- **num_reads** (int, optional, default=len(initial_states) or 1) – Number of reads. If num
reads is not explicitly given, it is selected to match the number of initial states given. If no initial states are given, it defaults to 1.
- **initial_states** (samples-like, optional, default=None) – One or more samples, each defining an initial state for all the problem variables. Initial states are given one per read, but if fewer than num
reads initial states are defined, additional values are generated as specified by initial
states_generator. See func..as_samples for a description of “samples-like”.
- **initial_states_generator** ("none", "tile", "random"), optional, default="random") – Defines the expansion of initial states if fewer than num
reads are specified:
  - "none": If the number of initial states specified is smaller than num
reads, raises Value
Error.
  - "tile": Reuses the specified initial states if fewer than num
reads or truncates if greater.
  - "random": Expands the specified initial states with randomly generated states if fewer than num
reads or truncates if greater.
- **seed** (int (32-bit unsigned integer), optional) – Seed to use for the PRNG. Specifying a particular seed with a constant set of parameters produces identical results. If not provided, a random seed is chosen.

Returns The initial states as provided, generated or augmented.
Return type \texttt{SampleSet}

\texttt{dimod.reference.samplers.identity_sampler.IdentitySampler.sample\_ising}

\texttt{IdentitySampler.sample\_ising}(h, J, **parameters)

Sample from an Ising model using the implemented sample method.

This method is inherited from the \texttt{Sampler} base class.

Converts the Ising model into a \texttt{BinaryQuadraticModel} and then calls \texttt{sample()}.

Parameters

- \texttt{h (dict/list)} – Linear biases of the Ising problem. If a dict, should be of the form \{\(v: bias,\ldots\)\} where \(v\) is a spin-valued variable and \(bias\) is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- \texttt{J (dict[(variable, variable), bias])} – Quadratic biases of the Ising problem.
- \texttt{**kwargs} – See the implemented sampling for additional keyword definitions.

Returns \texttt{SampleSet}

See also: \texttt{sample()}, \texttt{sample\_qubo()}

\texttt{dimod.reference.samplers.identity_sampler.IdentitySampler.sample\_qubo}

\texttt{IdentitySampler.sample\_qubo}(Q, **parameters)

Sample from a QUBO using the implemented sample method.

This method is inherited from the \texttt{Sampler} base class.

Converts the QUBO into a \texttt{BinaryQuadraticModel} and then calls \texttt{sample()}.

Parameters

- \texttt{Q (dict)} – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form \{(\(u, v\)): bias, \ldots\} where \(u, v\) are binary-valued variables and \(bias\) is their associated coefficient.
- \texttt{**kwargs} – See the implemented sampling for additional keyword definitions.

Returns \texttt{SampleSet}

See also: \texttt{sample()}, \texttt{sample\_ising()}

\textbf{Null Sampler}

A sampler that always returns an empty sample set.
Class

class NullSampler(parameters=None)

A sampler that always returns an empty sample set.

This sampler is useful for writing unit tests where the result is not important.

Parameters parameters (iterable/dict, optional) – If provided, sets the parameters accepted by the sample methods. The values given in these parameters are ignored.

Examples

```python
>>> bqm = dimod.BinaryQuadraticModel.from_qubo({('a', 'b'): 1})
>>> sampler = dimod.NullSampler()
>>> sampleset = sampler.sample(bqm)
>>> len(sampleset)
0
```

The next example shows how to enable additional parameters for the null sampler.

```python
>>> bqm = dimod.BinaryQuadraticModel.from_qubo({('a', 'b'): 1})
>>> sampler = dimod.NullSampler(parameters=['a'])
>>> sampleset = sampler.sample(bqm, a=5)
```

Properties

NullSampler.parameters

Keyword arguments accepted by the sampling methods

dimod.reference.samplers.null_sampler.NullSampler.parameters

NullSampler.parameters = None

Keyword arguments accepted by the sampling methods

Methods

NullSampler.sample(bqm, **kwargs) Return an empty sample set.

NullSampler.sample_ising(h, J, **parameters) Sample from an Ising model using the implemented sample method.

NullSampler.sample_qubo(Q, **parameters) Sample from a QUBO using the implemented sample method.

dimod.reference.samplers.null_sampler.NullSampler.sample

NullSampler.sample(bqm, **kwargs)

Return an empty sample set.

Parameters

• bqm (BinaryQuadraticModel) – The binary quadratic model determines the variables labels in the sample set.
• **kwargs – As specified when constructing the null sampler.

Returns The empty sample set.

Return type :class:`SampleSet`

dimod.reference.samplers.null_sampler.NullSampler.sample_ising

NullSampler.\texttt{sample\_ising}(h, J, **parameters)

Sample from an Ising model using the implemented sample method.

This method is inherited from the :class:`Sampler` base class.

Converts the Ising model into a :class:`BinaryQuadraticModel` and then calls :meth:`sample()`.

Parameters:

- **h** (dict/list) – Linear biases of the Ising problem. If a dict, should be of the form \{v: bias, . . . \} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.
- **kwargs – See the implemented sampling for additional keyword definitions.

Returns :class:`SampleSet`

See also: :meth:`sample()`, :meth:`sample_qubo()`

dimod.reference.samplers.null_sampler.NullSampler.sample_qubo

NullSampler.\texttt{sample\_qubo}(Q, **parameters)

Sample from a QUBO using the implemented sample method.

This method is inherited from the :class:`Sampler` base class.

Converts the QUBO into a :class:`BinaryQuadraticModel` and then calls :meth:`sample()`.

Parameters:

- **Q** (dict) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form \{(u, v): bias, . . . \} where u, v, are binary-valued variables and bias is their associated coefficient.
- **kwargs – See the implemented sampling for additional keyword definitions.

Returns :class:`SampleSet`

See also: :meth:`sample()`, :meth:`sample_ising()`

**Random Sampler**

A sampler that gives random samples.
Class

class RandomSampler
A sampler that gives random samples for testing.

Examples

This example produces 10 samples for a two-variable problem.

```python
>>> bqm = dimod.BinaryQuadraticModel.from_qubo({('a', 'b'): 1})
>>> sampler = dimod.RandomSampler()
>>> sampleset = sampler.sample(bqm, num_reads=10)
>>> len(sampleset)
10
```

Properties

```
RandomSampler.parameters
Keyword arguments accepted by the sampling methods.

dimod.reference.samplers.random_sampler.RandomSampler.parameters

RandomSampler.parameters = None
Keyword arguments accepted by the sampling methods.
Contents are exactly {'num_reads': []}
Type dict
```

Methods

```
RandomSampler.sample(bqm[, num_reads, seed])
Give random samples for a binary quadratic model.

RandomSampler.sample_ising(h, J[, **parameters])
Sample from an Ising model using the implemented sample method.

RandomSampler.sample_qubo(Q[, **parameters])
Sample from a QUBO using the implemented sample method.
```

```
dimod.reference.samplers.random_sampler.RandomSampler.sample

RandomSampler.sample(bqm, num_reads=10, seed=None)
Give random samples for a binary quadratic model.
Variable assignments are chosen by coin flip.

Parameters

- bqm (BinaryQuadraticModel) – Binary quadratic model to be sampled from.
- num_reads (int, optional, default=10) – Number of reads.
- seed (int (32-bit unsigned integer), optional) – Seed to use for the PRNG. Specifying a particular seed with a constant set of parameters produces identical results. If not provided, a random seed is chosen.
```
Returns \textit{SampleSet}

\texttt{dimod.reference.samplers.random\_sampler.RandomSampler.sample\_ising}

\texttt{RandomSampler.sample\_ising}(h, J, **parameters)

Sample from an Ising model using the implemented sample method.

This method is inherited from the \texttt{Sampler} base class.

Converts the Ising model into a \texttt{BinaryQuadraticModel} and then calls \texttt{sample()}.

**Parameters**

- \texttt{h (dict/list)} – Linear biases of the Ising problem. If a dict, should be of the form \{\texttt{v: bias,...}\} where \texttt{v} is a spin-valued variable and \texttt{bias} is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- \texttt{J (dict[(variable, variable), bias])} – Quadratic biases of the Ising problem.
- **\texttt{kwargs}** – See the implemented sampling for additional keyword definitions.

Returns \textit{SampleSet}

See also: \texttt{sample()}, \texttt{sample\_qubo()}

\texttt{dimod.reference.samplers.random\_sampler.RandomSampler.sample\_qubo}

\texttt{RandomSampler.sample\_qubo}(Q, **parameters)

Sample from a QUBO using the implemented sample method.

This method is inherited from the \texttt{Sampler} base class.

Converts the QUBO into a \texttt{BinaryQuadraticModel} and then calls \texttt{sample()}.

**Parameters**

- \texttt{Q (dict)} – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form \{(\texttt{u, v}): bias, ...\} where \texttt{u, v} are binary-valued variables and \texttt{bias} is their associated coefficient.
- **\texttt{kwargs}** – See the implemented sampling for additional keyword definitions.

Returns \textit{SampleSet}

See also: \texttt{sample()}, \texttt{sample\_ising()}

**Simulated Annealing Sampler**

A reference implementation of a simulated annealing sampler.

\texttt{neal.sampler.SimulatedAnnealingSampler} is a more performant implementation of simulated annealing you can use for solving problems.
Class

class SimulatedAnnealingSampler
A simple simulated annealing sampler for testing and debugging code.

Examples

This example solves a two-variable Ising model.

```python
>>> h = {'a': -0.5, 'b': 1.0}
>>> J = {('a', 'b'): -1.5}
>>> sampleset = dimod.SimulatedAnnealingSampler().sample_ising(h, J)
>>> sampleset.first.sample
{'a': -1, 'b': -1}
```

Properties

```python
SimulatedAnnealingSampler.parameters
Keyword arguments accepted by the sampling methods.

dimod.reference.samplers.simulated_annealing.SimulatedAnnealingSampler.parameters
SimulatedAnnealingSampler.parameters = None
Keyword arguments accepted by the sampling methods.

Contents are exactly {'beta_range': [], 'num_reads': [], 'num_sweeps': []}

Type dict
```

Methods

```python
SimulatedAnnealingSampler.sample(bqm, ...)
Sample from low-energy spin states using simulated annealing.

SimulatedAnnealingSampler.
sample_ising(h, J,...)
Sample from an Ising model using the implemented sample method.

SimulatedAnnealingSampler.
sample_qubo(Q,...)
Sample from a QUBO using the implemented sample method.
```

```python
SimulatedAnnealingSampler.sample(bqm, beta_range=None, num_reads=10, num_sweeps=1000)
Sample from low-energy spin states using simulated annealing.

Parameters

- **bqm** (*BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- **beta_range** (*tuple, optional*) – Beginning and end of the beta schedule (beta is the inverse temperature) as a 2-tuple. The schedule is applied linearly in beta. Default is chosen based on the total bias associated with each node.
```
• **num_reads** *(int, optional, default=10)* – Number of reads. Each sample is the result of a single run of the simulated annealing algorithm.

• **num_sweeps** *(int, optional, default=1000)* – Number of sweeps or steps.

Returns *SampleSet*

Note: This is a reference implementation, not optimized for speed and therefore not an appropriate sampler for benchmarking.

dimod.reference.samplers.simulated_annealing.SimulatedAnnealingSampler.sample_ising

SimulatedAnnealingSampler.*sample_ising**(h, J, **parameters)**

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

• **h** *(dict/list)* – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

• **J** *(dict[(variable, variable), bias])* – Quadratic biases of the Ising problem.

• ****kwargs – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also: *sample(), sample_qubo()*

dimod.reference.samplers.simulated_annealing.SimulatedAnnealingSampler.sample_qubo

SimulatedAnnealingSampler.*sample_qubo**(Q, **parameters)**

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.  

Parameters

• **Q** *(dict)* – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where u, v, are binary-valued variables and *bias* is their associated coefficient.

• ****kwargs – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also: *sample(), sample_ising()*
Composites

The `dimod` package includes several example composed samplers.

**Connected Components Composite**

A composite that breaks the problem into sub-problems corresponding to the connected components of the binary quadratic model graph before sending to its child sampler.

**Class**

```python
class ConnectedComponentsComposite(child_sampler):
    """Composite to decompose a problem to the connected components and solve each."
    """
    Connected components of a binary quadratic model (BQM) graph are computed (if not provided), and each subproblem is passed to the child sampler. Returned samples from each child sampler are merged. Only the best solution of each response is picked and merge with others (i.e. this composite returns a single solution).

    Parameters
    ----------
    sampler : dimod.Sampler
        A dimod sampler

Examples

This example uses `ConnectedComponentsComposite` to solve a simple Ising problem that can be separated into two components. This small example uses `dimod.ExactSolver` and is just illustrative.

```python
>>> h = {}
>>> J1 = {(1, 2): -1.0, (2, 3): 2.0, (3, 4): 3.0}
>>> J2 = {(12, 13): 6}
>>> sampler = dimod.ExactSolver()
>>> sampler_ccc = dimod.ConnectedComponentsComposite(sampler)
>>> e1 = sampler.sample_ising(h, J1).first.energy
>>> e2 = sampler.sample_ising(h, J2).first.energy
>>> e_ccc = sampler_ccc.sample_ising(h, {**J1, **J2}).first.energy
>>> e_ccc == e1 + e2
True
```

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ConnectedComponentsComposite.child</code></td>
<td>The child sampler.</td>
</tr>
<tr>
<td><code>ConnectedComponentsComposite.children</code></td>
<td>List of child samplers that are used by this composite.</td>
</tr>
<tr>
<td><code>ConnectedComponentsComposite.parameters</code></td>
<td>A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.</td>
</tr>
<tr>
<td><code>ConnectedComponentsComposite.properties</code></td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
</tbody>
</table>
ConnectedComponentsComposite.child

The child sampler. First sampler in Composite.children.

Type Sampler

ConnectedComponentsComposite.children

List of child samplers that are used by this composite.

Type list [Sampler]

ConnectedComponentsComposite.parameters

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Type dict

ConnectedComponentsComposite.properties

A dict containing any additional information about the sampler.

Type dict

Methods

| ConnectedComponentsComposite.sample(bqm[, ...]) | Sample from the provided binary quadratic model. |
| ConnectedComponentsComposite.sample_ising(h, ...) | Sample from an Ising model using the implemented sample method. |
| ConnectedComponentsComposite.sample_qubo(Q, ...) | Sample from a QUBO using the implemented sample method. |

ConnectedComponentsComposite.sample (bqm, components=None, **parameters)

Sample from the provided binary quadratic model.

Parameters

- bqm (dimod.BinaryQuadraticModel) – Binary quadratic model to be sampled from.
- components (list(set)) – A list of disjoint set of variables that fully partition the variables
- **parameters – Parameters for the sampling method, specified by the child sampler.

Returns dimod.SampleSet
dimod.reference.composites.connectedcomponent.ConnectedComponentsComposite.sample_ising

ConnectedComponentsComposite.sample_ising(h, J, **parameters)
Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls sample().

Parameters

- **h** *(dict/list)* – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where `v` is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

- **J** *(dict[(variable, variable), bias])* – Quadratic biases of the Ising problem.

- **kwargs** – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also: sample(), sample_qubo()

dimod.reference.composites.connectedcomponent.ConnectedComponentsComposite.sample_qubo

ConnectedComponentsComposite.sample_qubo(Q, **parameters)
Sample from a QUBO using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the QUBO into a BinaryQuadraticModel and then calls sample().

Parameters

- **Q** *(dict)* – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v` are binary-valued variables and `bias` is their associated coefficient.

- **kwargs** – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also: sample(), sample_ising()

Clip Composite

A composite that clips problem variables below and above threshold. If lower and upper bounds is not given it does nothing.

Class

```python
class ClipComposite(child_sampler)
    Composite to clip variables of a problem.

    Clips the variables of a binary quadratic model (BQM) and modifies linear and quadratic terms accordingly.
```
Parameters `sampler (dimod.Sampler)` – A dimod sampler.

Examples

This example uses `ClipComposite` to instantiate a composed sampler that submits a simple Ising problem to a sampler. The composed sampler clips linear and quadratic biases as indicated by options.

```python
>>> h = {'a': -4.0, 'b': -4.0}
>>> J = {('a', 'b'): 3.2}
>>> sampler = dimod.ClipComposite(dimod.ExactSolver())
>>> response = sampler.sample_ising(h, J, lower_bound=-2.0, upper_bound=2.0)
```

Properties

- `ClipComposite.child` The child sampler.
- `ClipComposite.children` List of child samplers that are used by this composite.
- `ClipComposite.parameters` A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.
- `ClipComposite.properties` A dict containing any additional information about the sampler.

**dimod.reference.composites.clipcomposite.ClipComposite.child**

`ClipComposite.child`  
The child sampler. First sampler in `Composite.children`.  
Type `Sampler`

**dimod.reference.composites.clipcomposite.ClipComposite.children**

`ClipComposite.children`  
List of child samplers that are used by this composite.  
Type `list[Sampler]`

**dimod.reference.composites.clipcomposite.ClipComposite.parameters**

`ClipComposite.parameters`  
A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.  
Type `dict`

**dimod.reference.composites.clipcomposite.ClipComposite.properties**

`ClipComposite.properties`  
A dict containing any additional information about the sampler.
**Type**  
*dict*

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dimod.reference.composites.clipcomposite.ClipComposite.sample</strong></td>
<td>Clip and sample from the provided binary quadratic model.</td>
</tr>
<tr>
<td><strong>ClipComposite.sample(bqm, lower_bound=None, upper_bound=None, <strong>parameters)</strong></strong></td>
<td>If lower_bound and upper_bound are given variables with value above or below are clipped.</td>
</tr>
<tr>
<td><strong>ClipComposite.sample_ising(h, J, <strong>parameters)</strong></strong></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><strong>ClipComposite.sample_qubo(Q, <strong>parameters)</strong></strong></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

**dimod.reference.composites.clipcomposite.ClipComposite.sample**

**ClipComposite.sample(bqm, lower_bound=None, upper_bound=None, **parameters)**

Clip and sample from the provided binary quadratic model. If lower_bound and upper_bound are given variables with value above or below are clipped.

**Parameters**

- **bqm** *(dimod.BinaryQuadraticModel)* – Binary quadratic model to be sampled from.
- **lower_bound** *(number)* – Value by which to clip the variables from below.
- **upper_bound** *(number)* – Value by which to clip the variables from above.
- **parameters** – Parameters for the sampling method, specified by the child sampler.

**Returns**  
*dimod.SampleSet*

**dimod.reference.composites.clipcomposite.ClipComposite.sample_ising**

**ClipComposite.sample_ising(h, J, **parameters)**

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

**Parameters**

- **h** *(dict/list)* – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where *v* is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** *(dict[(variable, variable), bias])* – Quadratic biases of the Ising problem.
- **kwargs** – See the implemented sampling for additional keyword definitions.

**Returns**  
*SampleSet*

See also:  
*sample(), sample_qubo()*
ClipComposite.sample_qubo(Q, **parameters)
Sample from a QUBO using the implemented sample method.
This method is inherited from the Sampler base class.
Converts the QUBO into a BinaryQuadraticModel and then calls sample().

Parameters
- **Q(dict)** - Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form \{(u, v): bias, \ldots\} where u, v, are binary-valued variables and bias is their associated coefficient.
- **kwargs** - See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also: sample(), sample_ising()

Fixed Variable Composite

A composite that fixes the variables provided and removes them from the binary quadratic model before sending to its child sampler.

Class
class FixedVariableComposite(child_sampler)
Composite to fix variables of a problem to provided.
Fixes variables of a binary quadratic model (BQM) and modifies linear and quadratic terms accordingly. Returned samples include the fixed variable

Parameters
- **sampler**(dimod.Sampler) - A dimod sampler

Examples

This example uses FixedVariableComposite to instantiate a composed sampler that submits a simple Ising problem to a sampler. The composed sampler fixes a variable and modifies linear and quadratic biases according.

```python
>>> h = {1: -1.3, 4: -0.5}
>>> J = {(1, 4): -0.6}
>>> sampler = dimod.FixedVariableComposite(dimod.ExactSolver())
>>> sampleset = sampler.sample_ising(h, J, fixed_variables={1: -1})
```

Properties

FixedVariableComposite.child The child sampler.

Continued on next page
FixedVariableComposite.children
List of child samplers that are used by this composite.

FixedVariableComposite.parameters
A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

FixedVariableComposite.properties
A dict containing any additional information about the sampler.

dimod.reference.composites.fixedvariable.FixedVariableComposite.child

FixedVariableComposite.child
The child sampler. First sampler in Composite.children.

   Type  Sampler

dimod.reference.composites.fixedvariable.FixedVariableComposite.children

FixedVariableComposite.children
List of child samplers that are used by this composite.

   Type  list[ Sampler]

dimod.reference.composites.fixedvariable.FixedVariableComposite.parameters

FixedVariableComposite.parameters
A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

   Type  dict

dimod.reference.composites.fixedvariable.FixedVariableComposite.properties

FixedVariableComposite.properties
A dict containing any additional information about the sampler.

   Type  dict

Methods

FixedVariableComposite.sample(bqm[, ....])  Sample from the provided binary quadratic model.

FixedVariableComposite.sample_ising(h, J,...)  Sample from an Ising model using the implemented sample method.

FixedVariableComposite.sample_qubo(Q, ...)  Sample from a QUBO using the implemented sample method.

dimod.reference.composites.fixedvariable.FixedVariableComposite.sample

FixedVariableComposite.sample(bqm, fixed_variables=None, **parameters)
Sample from the provided binary quadratic model.
Parameters

- **bqm** *(dimod.BinaryQuadraticModel)* – Binary quadratic model to be sampled from.
- **fixed_variables** *(dict)* – A dictionary of variable assignments.
- ****parameters – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

dimod.reference.composites.fixedvariable.FixedVariableComposite.sample_ising

FixedVariableComposite.sample_ising(h, J, **parameters)
Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** *(dict/list)* – Linear biases of the Ising problem. If a dict, should be of the form \{v: bias, ...\} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** *(dict[(variable, variable), bias])* – Quadratic biases of the Ising problem.
- ****kwargs – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

*sample()*,
*sample_qubo()*

dimod.reference.composites.fixedvariable.FixedVariableComposite.sample_qubo

FixedVariableComposite.sample_qubo(Q, **parameters)
Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** *(dict)* – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form \{(u, v): bias, ...\} where u, v, are binary-valued variables and bias is their associated coefficient.
- ****kwargs – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

*sample()*,
*sample_ising()*
Roof Duality Composite

A composite that uses the roof duality algorithm\(^1\) to fix some variables in the binary quadratic model before passing it on to its child sampler.

Class

class RoofDualityComposite(child_sampler)

Uses roof duality to assign some variables before invoking child sampler.

Uses the fix_variables() function to determine variable assignments, then fixes them before calling the child sampler. Returned samples include the fixed variables.

Parameters child(dimod.Sampler) – A dimod sampler. Used to sample the binary quadratic model after variables have been fixed.

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoofDualityComposite.child</td>
<td>The child sampler.</td>
</tr>
<tr>
<td>RoofDualityComposite.children</td>
<td>List of child samplers that are used by this composite.</td>
</tr>
<tr>
<td>RoofDualityComposite.parameters</td>
<td>A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.</td>
</tr>
<tr>
<td>RoofDualityComposite.properties</td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
</tbody>
</table>

---

dimod.reference.composites.roofduality.RoofDualityComposite.properties

RoofDualityComposite.properties  
A dict containing any additional information about the sampler.  
   Type  dict

Methods

---

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoofDualityComposite.sample(bqm[, sampling_mode])</td>
<td>Sample from the provided binary quadratic model.</td>
</tr>
<tr>
<td>RoofDualityComposite.sample_ising(h, J, **parameters)</td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td>RoofDualityComposite.sample_qubo(Q, **parameters)</td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

---

dimod.reference.composites.roofduality.RoofDualityComposite.sample

RoofDualityComposite.sample(bqm, sampling_mode=True, **parameters)  
Sample from the provided binary quadratic model.  
   Uses the fix_variables() function to determine which variables to fix.  
   Parameters  
     • bqm (dimod.BinaryQuadraticModel) – Binary quadratic model to be sampled from.  
     • sampling_mode (bool, optional, default=True) – In sampling mode, only roof-duality is used. When sampling_mode is false, strongly connected components are used to fix more variables, but in some optimal solutions these variables may take different values.  
     • **parameters – Parameters for the child sampler.  
   Returns dimod.SampleSet

---

dimod.reference.composites.roofduality.RoofDualityComposite.sample_ising

RoofDualityComposite.sample_ising(h, J, **parameters)  
Sample from an Ising model using the implemented sample method.  
   This method is inherited from the Sampler base class.  
   Converts the Ising model into a BinaryQuadraticModel and then calls sample().  
   Parameters  
     • h (dict/list) – Linear biases of the Ising problem. If a dict, should be of the form {v: bias, ...} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.  
     • J (dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.  
     • **kwargs – See the implemented sampling for additional keyword definitions.  
   Returns SampleSet
See also:

```
sample(), sample_qubo()
```

```

dimod.reference.composites.roofduality.RoofDualityComposite.sample_qubo
```

RoofDualityComposite.sample_qubo(Q, **parameters)
Sample from a QUBO using the implemented sample method.
This method is inherited from the Sampler base class.
Converts the QUBO into a BinaryQuadraticModel and then calls sample().

Parameters

- **Q (dict)** – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form \{(u, v): bias, ...\} where u, v, are binary-valued variables and bias is their associated coefficient.
- **kwargs** – See the implemented sampling for additional keyword definitions.

Returns :class:`SampleSet`

See also:

```
sample(), sample_ising()
```

Scale Composite

A composite that scales problem variables as directed. If a scaling value is not specified, calculates it based on quadratic and bias ranges.

Class

```
class ScaleComposite(child_sampler)
```

Composite that scales variables of a problem.
Scales the variables of a binary quadratic model (BQM) and modifies linear and quadratic terms accordingly.

Parameters

- **sampler (dimod.Sampler)** – A dimod sampler.

Examples

This example uses ScaleComposite to instantiate a composed sampler that submits a simple Ising problem to a sampler. The composed sampler scales linear biases, quadratic biases, and offset as indicated by options.

```
>>> h = {'a': -4.0, 'b': -4.0}
>>> J = {('a', 'b'): 3.2}
>>> sampler = dimod.ScaleComposite(dimod.ExactSolver())
>>> response = sampler.sample_ising(h, J, scalar=0.5,
...                                         ignored_interactions=[('a','b')])
```

Properties
**ScaleComposite.child**
The child sampler.

**ScaleComposite.children**
List of child samplers that are used by this composite.

**ScaleComposite.parameters**
A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

**ScaleComposite.properties**
A dict containing any additional information about the sampler.

### dimod.reference.composites.scalecomposite.ScaleComposite.child

**ScaleComposite.child**
The child sampler. First sampler in Composite.children.

**Type** Sampler

### dimod.reference.composites.scalecomposite.ScaleComposite.children

**ScaleComposite.children**
List of child samplers that are used by this composite.

**Type** list[Sampler]

### dimod.reference.composites.scalecomposite.ScaleComposite.parameters

**ScaleComposite.parameters**
A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

**Type** dict

### dimod.reference.composites.scalecomposite.ScaleComposite.properties

**ScaleComposite.properties**
A dict containing any additional information about the sampler.

**Type** dict

### Methods

**ScaleComposite.sample**(bqm[, scalar, ...])
Scale and sample from the provided binary quadratic model.

**ScaleComposite.sample_ising**(h, J, **parameters)**
Sample from an Ising model using the implemented sample method.

**ScaleComposite.sample_qubo**(Q, **parameters)**
Sample from a QUBO using the implemented sample method.
ScaleComposite.sample(bqm, scalar=None, bias_range=1, quadratic_range=None, ignored_variables=None, ignored_interactions=None, ignore_offset=False, **parameters)

Scale and sample from the provided binary quadratic model.

if scalar is not given, problem is scaled based on bias and quadratic ranges. See BinaryQuadraticModel.scale() and BinaryQuadraticModel.normalize()

Parameters

- **bqm**(dimod.BinaryQuadraticModel) – Binary quadratic model to be sampled from.
- **scalar**(number) – Value by which to scale the energy range of the binary quadratic model. Overrides bias_range and quadratic_range.
- **bias_range**(number/pair, default=1) – Value/range by which to normalize the all the biases, or if quadratic_range is provided, just the linear biases. Overridden by scalar.
- **quadratic_range**(number/pair) – Value/range by which to normalize the quadratic biases. Overridden by scalar.
- **ignored_variables**(iterable, optional) – Biases associated with these variables are not scaled.
- **ignored_interactions**(iterable[tuple], optional) – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.
- **ignore_offset**(bool, default=False) – If True, the offset is not scaled.
- ****parameters – Parameters for the sampling method, specified by the child sampler.

Returns dimod.SampleSet

dimod.reference.composites.scalecomposite.ScaleComposite.sample_ising

ScaleComposite.sample_ising(h, J, **parameters)

Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls sample().

Parameters

- **h**(dict/list) – Linear biases of the Ising problem. If a dict, should be of the form {v: bias, ...} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J**(dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.
- ****kwargs – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also:

sample(), sample_qubo()
dimod.reference.composites.scalecomposite.ScaleComposite.sample_qubo

ScaleComposite.sample_qubo(Q, **parameters)
Sample from a QUBO using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the QUBO into a BinaryQuadraticModel and then calls sample().

Parameters

- Q (dict) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem.
Should be a dict of the form \{(u, v): bias, ...\} where \(u, v\) are binary-valued variables and bias is their associated coefficient.

- **kwargs – See the implemented sampling for additional keyword definitions.

Returns

SampleSet

See also:
sample(), sample_ising()

Spin Reversal Transform Composite

On the D-Wave system, coupling \(J_{i,j}\) adds a small bias to qubits \(i\) and \(j\) due to leakage. This can become significant for chained qubits. Additionally, qubits are biased to some small degree in one direction or another. Applying a spin-reversal transform can improve results by reducing the impact of possible analog and systematic errors. A spin-reversal transform does not alter the Ising problem; the transform simply amounts to reinterpreting spin up as spin down, and visa-versa, for a particular spin.

Class

class SpinReversalTransformComposite(child)
Composite for applying spin reversal transform preprocessing.

Spin reversal transforms (or “gauge transformations”) are applied by flipping the spin of variables in the Ising problem. After sampling the transformed Ising problem, the same bits are flipped in the resulting sample.

Parameters

- sampler – A dimod sampler object.

Examples

This example composes a dimod ExactSolver sampler with spin transforms then uses it to sample an Ising problem.

```python
>>> base_sampler = dimod.ExactSolver()
>>> composed_sampler = dimod.SpinReversalTransformComposite(base_sampler)
... # Sample an Ising problem
>>> response = composed_sampler.sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b'): -1})
>>> response.first.sample
{'a': -1, 'b': -1}
```

References

Properties

SpinReversalTransformComposite.child
The child sampler.

SpinReversalTransformComposite.children

SpinReversalTransformComposite.parameters

SpinReversalTransformComposite.properties

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.child

SpinReversalTransformComposite.child
The child sampler. First sampler in Composite.children.

Type Sampler

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.children

SpinReversalTransformComposite.children = None

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.parameters

SpinReversalTransformComposite.parameters = None

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.properties

SpinReversalTransformComposite.properties = None

Methods

SpinReversalTransformComposite.sample(bqm[, ...])
Sample from the binary quadratic model.

SpinReversalTransformComposite.sample_ising(h, ...)
Sample from an Ising model using the implemented sample method.

SpinReversalTransformComposite.sample_qubo(Q, ...)
Sample from a QUBO using the implemented sample method.

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.sample

SpinReversalTransformComposite.sample(bqm, num_spin_reversal_transforms=2, **kwags)
Sample from the binary quadratic model.

Parameters

- bqm (BinaryQuadraticModel) – Binary quadratic model to be sampled from.
num_spin_reversal_transforms (integer, optional, default=2) –
Number of spin reversal transform runs.

Returns SampleSet

Examples

This example runs 100 spin reversals applied to one variable of a QUBO problem.

```python
>>> base_sampler = dimod.ExactSolver()
>>> composed_sampler = dimod.SpinReversalTransformComposite(base_sampler)
...  
>>> Q = {('a', 'a'): -1, ('b', 'b'): -1, ('a', 'b'): 2}
>>> response = composed_sampler.sample_qubo(Q,
...    num_spin_reversal_transforms=100)
>>> len(response)
400
```

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.sample_ising

SpinReversalTransformComposite.sample_ising (h, J, **parameters)
Sample from an Ising model using the implemented sample method.
This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls sample().

Parameters

- **h (dict/list)** – Linear biases of the Ising problem. If a dict, should be of the form {v: bias, ...} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J (dict[(variable, variable), bias])** – Quadratic biases of the Ising problem.
- ****kwargs – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also:

sample(), sample_qubo()

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.sample_qubo

SpinReversalTransformComposite.sample_qubo (Q, **parameters)
Sample from a QUBO using the implemented sample method.
This method is inherited from the Sampler base class.

Converts the QUBO into a BinaryQuadraticModel and then calls sample().

Parameters

- **Q (dict)** – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form {u, v]: bias, ...} where u, v, are binary-valued variables and bias is their associated coefficient.
Structured Composite

A composite that structures a sampler.

Class

class StructureComposite(sampler, nodelist, edgelist)

Creates a structured composed sampler from an unstructured sampler.

Parameters

• **Sampler**(Sampler) – Unstructured sampler.

• **nodelist**(list) – Nodes/variables allowed by the sampler formatted as a list.

• **edgelist**(list[(node, node)]) – Edges/interactions allowed by the sampler, formatted as a list where each edge/interaction is a 2-tuple.

Examples

This example creates a composed sampler from the unstructure dimod ExactSolver sampler. The target structure is a square graph.

```python
>>> base_sampler = dimod.ExactSolver()
>>> node_list = [0, 1, 2, 3]
>>> edge_list = [(0, 1), (1, 2), (2, 3), (0, 3)]
>>> structured_sampler = dimod.StructureComposite(base_sampler, node_list, edge_list)
... linear = {0: 0.0, 1: 0.0, 2: 0.0, 3: 0.0}
>>> quadratic = {(0, 1): 1.0, (1, 2): 1.0, (0, 3): 1.0, (2, 3): -1.0}
>>> bqm = dimod.BinaryQuadraticModel(linear, quadratic, 1.0, dimod.Vartype.SPIN)
... response = structured_sampler.sample(bqm)
... print(response.first.energy)
-1.0
```

The next part of the example tries giving the composed sampler a non-square model:

```python
>>> del quadratic[(0, 1)]
>>> quadratic[(0, 2)] = 1.0
>>> bqm = dimod.BinaryQuadraticModel(linear, quadratic, 1.0, dimod.Vartype.SPIN)
... try:
... response = structured_sampler.sample(bqm)
... except dimod.BinaryQuadraticModelStructureError as details:
... print(details)
... given bqm does not match the sampler's structure
```
Properties

<table>
<thead>
<tr>
<th>StructureComposite.child</th>
<th>The child sampler.</th>
</tr>
</thead>
<tbody>
<tr>
<td>StructureComposite.children</td>
<td></td>
</tr>
<tr>
<td>StructureComposite.parameters</td>
<td>A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.</td>
</tr>
<tr>
<td>StructureComposite.properties</td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
</tbody>
</table>

**dimod.reference.composites.structure.StructureComposite.child**

StructureComposite.child

The child sampler. First sampler in Composite.children.

Type Sampler

**dimod.reference.composites.structure.StructureComposite.children**

StructureComposite.children = None

**dimod.reference.composites.structure.StructureComposite.parameters**

StructureComposite.parameters

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

Type dict

**dimod.reference.composites.structure.StructureComposite.properties**

StructureComposite.properties

A dict containing any additional information about the sampler.

Type dict

Methods

<table>
<thead>
<tr>
<th>StructureComposite.sample(bqm, **sample_kwargs)</th>
<th>Sample from the binary quadratic model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>StructureComposite.sample_ising(h, J, ...)</td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td>StructureComposite.sample_qubo(Q, **parameters)</td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>
StructureComposite.sample(bqm, **sample_kwargs)

Sample from the binary quadratic model.

**Parameters**

- **bqm** (*BinaryQuadraticModel*) – Binary quadratic model to be sampled from.

**Returns**

*SampleSet*

**Examples**

This example submits an Ising problem to a composed sampler that uses the dimod *ExactSolver* only on problems structured for a K2 fully connected graph.

```python
>>> nodes = [0, 1]
>>> edges = [(0, 1)]
>>> composed_sampler = dimod.StructureComposite(dimod.ExactSolver(),
...                                               nodes, edges)
>>> response = composed_sampler.sample_ising({0: 1, 1: 1}, {})
>>> set(response.first.sample.values())
{-1}
```

dimod.reference.composites.structure.StructureComposite.sample_ising

StructureComposite.sample_ising(h, J, **parameters)

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

**Parameters**

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form \{v: bias, ...\} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.


- ****kwargs – See the implemented sampling for additional keyword definitions.

**Returns**

*SampleSet*

See also:

*sample(), sample_qubo()*

dimod.reference.composites.structure.StructureComposite.sample_qubo

StructureComposite.sample_qubo(Q, **parameters)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

**Parameters**
• **Q**(dict) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form \((u, v): bias, \ldots\) where \(u, v\) are binary-valued variables and \(bias\) is their associated coefficient.

• **kwargs** – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also:
sample(), sample_ising()

Tracking Composite

A composite that tracks inputs and outputs.

Class

class TrackingComposite(child, copy=False)

Composite that tracks inputs and outputs for debugging and testing.

Parameters

• child(dimod.Sampler) – A dimod sampler.

• copy(bool, optional, default=False) – If True, the inputs/outputs are copied (with copy.deepcopy()) before they are stored. This is useful if the child sampler mutates the values.

Examples

```python
>>> sampler = dimod.TrackingComposite(dimod.RandomSampler())
>>> sampleset = sampler.sample_ising({'a': -1}, {('a', 'b'): 1},
... num_reads=5)
>>> sampler.input
OrderedDict([('h', {'a': -1}), ('J', {('a', 'b'): 1}), ('num_reads', 5)])
>>> sampleset == sampler.output
True
```

If we make additional calls to the sampler, the most recent input/output are stored in input and output respectively. However, all are tracked in inputs and outputs.

```python
>>> sampleset2 = sampler.sample_qubo({('a', 'b'): 1})
>>> sampler.input
OrderedDict([('Q', {('a', 'b'): 1})])
>>> sampler.inputs # doctest: +SKIP
[OrderedDict([('h', {'a': -1}), ('J', {('a', 'b'): 1}), ('num_reads', 5))],
OrderedDict([('Q', {('a', 'b'): 1})])]
```

In the case that you want to nest the tracking composite, there are two patterns for retrieving the data.

```python
>>> from dimod import ScaleComposite, TrackingComposite, ExactSolver
... sampler = ScaleComposite(TrackingComposite(ExactSolver()))
>>> sampler.child.inputs # empty because we haven't called sample
[]
```
```python
>>> intermediate_sampler = TrackingComposite(ExactSolver())
>>> sampler = ScaleComposite(intermediate_sampler)
>>> intermediate_sampler.inputs
[]
```

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TrackingComposite.input</strong></td>
<td>The most recent input to any sampling method.</td>
</tr>
<tr>
<td><strong>TrackingComposite.inputs</strong></td>
<td>All of the inputs to any sampling methods.</td>
</tr>
<tr>
<td><strong>TrackingComposite.output</strong></td>
<td>The most recent output of any sampling method.</td>
</tr>
<tr>
<td><strong>TrackingComposite.outputs</strong></td>
<td>All of the outputs from any sampling methods.</td>
</tr>
<tr>
<td><strong>TrackingComposite.parameters</strong></td>
<td>A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.</td>
</tr>
<tr>
<td><strong>TrackingComposite.properties</strong></td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
</tbody>
</table>

**dimod.reference.composites.tracking.TrackingComposite.input**

- **TrackingComposite.input**
  - The most recent input to any sampling method.

**dimod.reference.composites.tracking.TrackingComposite.inputs**

- **TrackingComposite.inputs**
  - All of the inputs to any sampling methods.

**dimod.reference.composites.tracking.TrackingComposite.output**

- **TrackingComposite.output**
  - The most recent output of any sampling method.

**dimod.reference.composites.tracking.TrackingComposite.outputs**

- **TrackingComposite.outputs**
  - All of the outputs from any sampling methods.

**dimod.reference.composites.tracking.TrackingComposite.parameters**

- **TrackingComposite.parameters**
  - A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.
  
  **Type**: `dict`
dimod.reference.composites.tracking.TrackingComposite.properties

TrackingComposite.properties
A dict containing any additional information about the sampler.

    Type  dict

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrackingComposite.clear()</td>
<td>Clear all the inputs/outputs.</td>
</tr>
<tr>
<td>TrackingComposite.sample(bqm, **parameters)</td>
<td>Sample from the child sampler and store the given inputs/outputs.</td>
</tr>
<tr>
<td>TrackingComposite.sample_ising(h, J, **parameters)</td>
<td>Sample from the child sampler and store the given inputs/outputs.</td>
</tr>
<tr>
<td>TrackingComposite.sample_qubo(Q, **parameters)</td>
<td>Sample from the child sampler and store the given inputs/outputs.</td>
</tr>
</tbody>
</table>

dimod.reference.composites.tracking.TrackingComposite.clear

TrackingComposite.clear()
Clear all the inputs/outputs.

dimod.reference.composites.tracking.TrackingComposite.sample

TrackingComposite.sample(bqm, **parameters)
Sample from the child sampler and store the given inputs/outputs.

The binary quadratic model and any parameters are stored in inputs. The returned sample set is stored in outputs.

Parameters

- **bqm (dimod.BinaryQuadraticModel)** – Binary quadratic model to be sampled from.
- **kwargs** – Parameters for the sampling method, specified by the child sampler.

Returns  dimod.SampleSet

dimod.reference.composites.tracking.TrackingComposite.sample_ising

TrackingComposite.sample_ising(h, J, **parameters)
Sample from the child sampler and store the given inputs/outputs.

The binary quadratic model and any parameters are stored in inputs. The returned sample set is stored in outputs.

Parameters

- **h (dict/list)** – Linear biases of the Ising problem. If a dict, should be of the form {v: bias, ...} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J (dict [(variable, variable), bias])** – Quadratic biases of the Ising problem.
**kwargs – Parameters for the sampling method, specified by the child sampler.

Returns `dimod.SampleSet`

dimod.reference.composites.tracking.TrackingComposite.sample_qubo

TrackingComposite.sample_qubo(Q, **parameters)

Sample from the child sampler and store the given inputs/outputs.

The binary quadratic model and any parameters are stored in `inputs`. The returned sample set is stored in `outputs`.

Parameters

- **Q (dict)** – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v` are binary-valued variables and `bias` is their associated coefficient.
- **kwargs – Parameters for the sampling method, specified by the child sampler.

Returns `dimod.SampleSet`

Truncate Composite

A composite that truncates the response based on options provided by the user.

Class

class TruncateComposite (child_sampler, n, sorted_by='energy', aggregate=False)

Composite to truncate the returned samples

Inherits from `dimod.ComposedSampler`.

Post-processing is expensive and sometimes one might want to only treat the lowest energy samples. This composite layer allows one to pre-select the samples within a multi-composite pipeline

Parameters

- **child_sampler (dimod.Sampler)** – A dimod sampler
- **n (int)** – Maximum number of rows in the returned sample set.
- **sorted_by (str/None, optional, default='energy')** – Selects the record field used to sort the samples before truncating. Note that sample order is maintained in the underlying array.
- **aggregate (bool, optional, default=False)** – If True, aggregate the samples before truncating.

Note: If aggregate is True `SampleSet.record.num_occurrences` are accumulated but no other fields are.

Properties
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TruncateComposite.children</code></td>
<td>List of child samplers that that are used by this composite.</td>
</tr>
<tr>
<td><code>TruncateComposite.parameters</code></td>
<td>A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.</td>
</tr>
<tr>
<td><code>TruncateComposite.properties</code></td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
</tbody>
</table>

**dimod.reference.composites.truncatecomposite.TruncateComposite.child**

`TruncateComposite.child`

The child sampler. First sampler in `Composite.children`.

   Type  `Sampler`

**dimod.reference.composites.truncatecomposite.TruncateComposite.children**

`TruncateComposite.children`

List of child samplers that that are used by this composite.

   Type  `list[Sampler]`

**dimod.reference.composites.truncatecomposite.TruncateComposite.parameters**

`TruncateComposite.parameters`

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

   Type  `dict`

**dimod.reference.composites.truncatecomposite.TruncateComposite.properties**

`TruncateComposite.properties`

A dict containing any additional information about the sampler.

   Type  `dict`

**Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TruncateComposite.sample(bqm, **kwargs)</code></td>
<td>Sample from the problem provided by BQM and truncate output.</td>
</tr>
<tr>
<td><code>TruncateComposite.sample_ising(h, J, ...)</code></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><code>TruncateComposite.sample_qubo(Q, **parameters)</code></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>
**dimod.reference.composites.truncatecomposite.TruncateComposite.sample**

TruncateComposite.sample(bqm, **kwargs)

Sample from the problem provided by BQM and truncate output.

**Parameters**

- **bqm** (dimod.BinaryQuadraticModel) – Binary quadratic model to be sampled from.
- **kwargs** – Parameters for the sampling method, specified by the child sampler.

**Returns** dimod.SampleSet

**dimod.reference.composites.truncatecomposite.TruncateComposite.sample_ising**

TruncateComposite.sample_ising(h, J, **parameters)

Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls sample().

**Parameters**

- **h** (dict/list) – Linear biases of the Ising problem. If a dict, should be of the form \{v: bias, . . . \} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.
- **kwargs** – See the implemented sampling for additional keyword definitions.

**Returns** SampleSet

See also:

sample(), sample_qubo()

**dimod.reference.composites.truncatecomposite.TruncateComposite.sample_qubo**

TruncateComposite.sample_qubo(Q, **parameters)

Sample from a QUBO using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the QUBO into a BinaryQuadraticModel and then calls sample().

**Parameters**

- **Q** (dict) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form \{(u, v): bias, . . . \} where u, v, are binary-valued variables and bias is their associated coefficient.
- **kwargs** – See the implemented sampling for additional keyword definitions.

**Returns** SampleSet

See also:

sample(), sample_ising()
Higher-Order Samplers

The *dimod* package includes the following example higher-order samplers.

**Exact Polynomial Solver**

A simple exact solver for testing and debugging code using your local CPU.

**Note:** This sampler is designed for use in testing. Because it calculates the energy for every possible sample, it is very slow.

**Class**

```python
class ExactPolySolver
    A simple exact polynomial solver for testing/debugging code on your CPU.
```

**Notes**

This solver becomes slow for problems with 18 or more variables.

**Examples**

This example solves a three-variable Ising model.

```python
>>> h = {'a': -0.5, 'b': 1.0, 'c': 0.}
>>> J = {('a', 'b'): -1.5, ('a', 'b', 'c'): -1.0}
>>> sampleset = dimod.ExactPolySolver().sample_hising(h, J)
>>> print(sampleset)  # doctest: +SKIP
          a  b  c  energy  num_oc.
   1 -1  -1  1 -3.0   1
   5  1   1 -1 -2.0   1
   0 -1  -1  1 -1.0   1
   3  1  -1  1 -1.0   1
   4  1  -1  0  0.0   1
   2  1  -1  1  1.0   1
   7 -1  -1  2  1.0   1
   6 -1  -1  1  4.0   1
['SPIN', 8 rows, 8 samples, 3 variables]
```

This example solves a three-variable HUBO.

```python
>>> Q = {('a', 'b'): 2.0, ('c',): 1.0, ('a', 'b', 'c'): -0.5}
>>> sampleset = dimod.ExactPolySolver().sample_hubo(Q)
>>> sampleset.first.energy
0.0
```

This example solves a three-variable binary polynomial.

```python
>>> poly = dimod.BinaryPolynomial({('a',): 1.5, ('a', 'b'): -1, ('a', 'b', 'c'): -0.5}, 'SPIN')
>>> sampleset = dimod.ExactPolySolver().sample_poly(poly)
>>> sampleset.first.sample
{'a': -1, 'b': -1, 'c': -1}
```
Methods

```python
dimod.reference.samplers.ExactPolySolver.sample

ExactPolySolver.sample(bqm, *args, **kwargs)
```

```python
dimod.reference.samplers.ExactPolySolver.sample_ising

ExactPolySolver.sample_ising(*args, **kwargs)
```

```python
dimod.reference.samplers.ExactPolySolver.sample_qubo

ExactPolySolver.sample_qubo(*args, **kwargs)
```

Higher-Order Composites

The `dimod` package includes several example higher-order composed samplers.

**HigherOrderComposite**

```python
class HigherOrderComposite(child_sampler):
    Convert a binary quadratic model sampler to a binary polynomial sampler.
    Energies of the returned samples do not include the penalties.

    Parameters
    sampler (dimod.Sampler) -- A dimod sampler
```

**Example**

This example uses `HigherOrderComposite` to instantiate a composed sampler that submits a simple Ising problem to a sampler. The composed sampler creates a binary quadratic model (BQM) from a higher order problem.

```python
>>> sampler = dimod.HigherOrderComposite(dimod.ExactSolver())
>>> h = {0: -0.5, 1: -0.3, 2: -0.8}
>>> J = {(0, 1, 2): -1.7}
>>> sampleset = sampler.sample_hising(h, J, discard_unsatisfied=True)
>>> set(sampleset.first.sample.values()) == {1}
True
```
**Properties**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HigherOrderComposite.child</td>
<td>The child sampler.</td>
</tr>
<tr>
<td>HigherOrderComposite.children</td>
<td>A list containing the wrapped sampler.</td>
</tr>
<tr>
<td>HigherOrderComposite.parameters</td>
<td>A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.</td>
</tr>
<tr>
<td>HigherOrderComposite.properties</td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
</tbody>
</table>

**dimod.reference.composites.HigherOrderComposite.child**

HigherOrderComposite.child

The child sampler. First sampler in Composite.children.

*Type*  
Sampler

**dimod.reference.composites.HigherOrderComposite.children**

HigherOrderComposite.children

A list containing the wrapped sampler.

**dimod.reference.composites.HigherOrderComposite.parameters**

HigherOrderComposite.parameters

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

*Type*  
dict

**dimod.reference.composites.HigherOrderComposite.properties**

HigherOrderComposite.properties

A dict containing any additional information about the sampler.

*Type*  
dict

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HigherOrderComposite.sample_poly(poly[, ...)])</td>
<td>Sample from the given binary polynomial.</td>
</tr>
<tr>
<td>HigherOrderComposite.sample_hising(h, J[, ...])</td>
<td>Sample from a higher-order Ising model.</td>
</tr>
<tr>
<td>HigherOrderComposite.sample_hubo(H[, **kwargs])</td>
<td>Sample from a higher-order unconstrained binary optimization problem.</td>
</tr>
</tbody>
</table>
HigherOrderComposite.sample_poly(poly, penalty_strength=1.0, keep_penalty_variables=False, discard_unsatisfied=False, **parameters)

Sample from the given binary polynomial.

Introduces penalties to reduce the given higher-order binary polynomial to a quadratic problem and sends it to its child sampler.

Parameters

- **poly** (*BinaryPolynomial*) – A binary polynomial.
- **penalty_strength** (*float*, *optional*) – Strength of the reduction constraint. Insufficient strength can result in the binary quadratic model not having the same minimization as the polynomial.
- **keep_penalty_variables** (*bool*, *optional*, *default=True*) – Setting to False removes the variables used for penalty from the samples.
- **discard_unsatisfied** (*bool*, *optional*, *default=False*) – Setting to True discards samples that do not satisfy the penalty conditions.
- **initial_state** (*dict*, *optional*) – Only accepted when the child sampler accepts an initial state. The initial state is given in terms of the variables in the binary polynomial. The corresponding initial values are populated for use by the child sampler.
- **parameters** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

HigherOrderComposite.sample_hising(h, J, **kwargs)

Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a *BinaryPolynomial* and call *sample_poly()*.

Parameters

- **h** (*dict*) – Variable biases of the Ising problem as a dict of the form *{v: bias, ...}* where *v* is a variable in the polynomial and *bias* its associated coefficient.
- **J** (*dict*) – Interaction biases of the Ising problem as a dict of the form *{(u, v, ...): bias}* where *u, v*, are spin-valued variables in the polynomial and *bias* their associated coefficient.
- **kwargs** – See *sample_poly()* for additional keyword definitions.

Returns *SampleSet*

See also:

*sample_poly(), sample_hubo()*

dimod.reference.composites.HigherOrderComposite.sample_hubo

HigherOrderComposite.sample_hubo(H, **kwargs)

Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a *BinaryPolynomial* and then call *sample_poly()*.
Parameters

- **H (dict)** – Coefficients of the HUBO as a dict of the form \{(u, v, ...): bias, ...\}, where u, v, are binary-valued variables in the polynomial and bias their associated coefficient.

- **kwargs** – See `sample_poly()` for additional keyword definitions.

Returns `SampleSet`

See also: `sample_poly()`, `sample_hising()`

**PolyFixedVariableComposite**

class **PolyFixedVariableComposite** *(child_sampler)*

Composite that fixes variables of a problem.

Fixes variables of a binary polynomial and modifies linear and k-local terms accordingly. Returned samples include the fixed variable.

**Parameters**

- **sampler** *(dimod.PolySampler)* – A dimod polynomial sampler.

**Examples**

This example uses `PolyFixedVariableComposite` to instantiate a composed sampler that submits a simple high-order Ising problem to a sampler. The composed sampler fixes a variable and modifies linear and k-local terms biases.

```python
>>> h = {1: -1.3, 2: 1.2, 3: -3.4, 4: -0.5}
>>> J = {(1, 4): -0.6, (1, 2, 3): 0.2, (1, 2, 3, 4): -0.1}
>>> poly = dimod.BinaryPolynomial.from_hising(h, J, offset=0)
>>> sampler = dimod.PolyFixedVariableComposite(dimod.ExactPolySolver())
>>> sampleset = sampler.sample_poly(poly, fixed_variables={3: -1, 4: 1})
```

**Properties**

- **PolyFixedVariableComposite.child**
  The child sampler.

- **PolyFixedVariableComposite.children**
  List of child samplers that that are used by this composite.

- **PolyFixedVariableComposite.parameters**
  A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

- **PolyFixedVariableComposite.properties**
  A dict containing any additional information about the sampler.

**dimod.reference.composites.PolyFixedVariableComposite.child**

*PolyFixedVariableComposite.child*

The child sampler. First sampler in Composite.children.

Type `Sampler`
PolyFixedVariableComposite

**children**
List of child samplers that are used by this composite.

Type: `list[Sampler]`

**parameters**
A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

Type: `dict`

**properties**
A dict containing any additional information about the sampler.

Type: `dict`

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sample_poly(poly)</code></td>
<td>Sample from the provided binary quadratic model.</td>
</tr>
<tr>
<td><code>sample_hising(h, ...)</code></td>
<td>Sample from a higher-order Ising model.</td>
</tr>
<tr>
<td><code>sample_hubo(H, ...)</code></td>
<td>Sample from a higher-order unconstrained binary optimization problem.</td>
</tr>
</tbody>
</table>

### `sample_poly`

**PolyFixedVariableComposite.sample_poly**

Sample from the provided binary quadratic model.

Parameters:

- `poly` (`dimod.BinaryPolynomial`) – Binary polynomial model to be sampled from.
- `fixed_variables` (`dict`) – A dictionary of variable assignments.
- `**parameters` – Parameters for the sampling method, specified by the child sampler.

Returns: `dimod.SampleSet`

### `sample_hising`

**PolyFixedVariableComposite.sample_hising**

Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a `BinaryPolynomial` and call `sample_poly()`.
Parameters

- **h (dict)** – Variable biases of the Ising problem as a dict of the form \{v: bias, \ldots\}, where \(v\) is a variable in the polynomial and \(bias\) its associated coefficient.

- **J (dict)** – Interaction biases of the Ising problem as a dict of the form \{(u, v, \ldots): bias\}, where \(u, v\), are spin-valued variables in the polynomial and \(bias\) their associated coefficient.

- ****kwargs** – See `sample_poly()` for additional keyword definitions.

Returns `SampleSet`

See also:

`sample_poly()`, `sample_hubo()`

dimod.reference.composites.PolyFixedVariableComposite.sample_hubo

PolyFixedVariableComposite.sample_hubo(H, **kwargs)
Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a `BinaryPolynomial` and then call `sample_poly()`.

Parameters

- **H (dict)** – Coefficients of the HUBO as a dict of the form \{(u, v, \ldots): bias, \ldots\}, where \(u, v\), are binary-valued variables in the polynomial and \(bias\) their associated coefficient.

- ****kwargs** – See `sample_poly()` for additional keyword definitions.

Returns `SampleSet`

See also:

`sample_poly()`, `sample_hising()`

PolyScaleComposite

class PolyScaleComposite(child)
Composite to scale biases of a binary polynomial.

Parameters **child** (PolySampler) – A binary polynomial sampler.

Examples

```python
>>> linear = {'a': -4.0, 'b': -4.0}
>>> quadratic = {('a', 'b'): 3.2, ('a', 'b', 'c'): 1}
>>> sampler = dimod.PolyScaleComposite(dimod.HigherOrderComposite(dimod.ExactSolver()))
>>> response = sampler.sample_hising(linear, quadratic, scalar=0.5,
... ignored_terms=[('a','b')])
```

Properties
**PolyScaleComposite.child**

The child sampler.

**PolyScaleComposite.children**

The child sampler in a list

**PolyScaleComposite.parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

**PolyScaleComposite.properties**

A dict containing any additional information about the sampler.

dimod.reference.composites.PolyScaleComposite.child

**PolyScaleComposite.child**

The child sampler. First sampler in Composite.children.

  **Type** Sampler

dimod.reference.composites.PolyScaleComposite.children

**PolyScaleComposite.children**

The child sampler in a list

dimod.reference.composites.PolyScaleComposite.parameters

**PolyScaleComposite.parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

  **Type** dict

dimod.reference.composites.PolyScaleComposite.properties

**PolyScaleComposite.properties**

A dict containing any additional information about the sampler.

  **Type** dict

**Methods**

**PolyScaleComposite.sample_poly**(poly[, scalar=None, bias_range=1, poly_range=None, ignored_terms=None, **parameters])

Scale and sample from the given binary polynomial.

**PolyScaleComposite.sample_hising**(h, J, **kwargs)

Sample from a higher-order Ising model.

**PolyScaleComposite.sample_hubo**(H, **kwargs)

Sample from a higher-order unconstrained binary optimization problem.

dimod.reference.composites.PolyScaleComposite.sample_poly

**PolyScaleComposite.sample_poly**(poly, scalar=None, bias_range=1, poly_range=None, ignored_terms=None, **parameters)

Scale and sample from the given binary polynomial.
If `scalar` is not given, problem is scaled based on bias and polynomial ranges. See `BinaryPolynomial.scale()` and `BinaryPolynomial.normalize()`.

**Parameters**
- `scalar (number, optional) –` Value by which to scale the energy range of the binary polynomial.
- `bias_range (number/pair, optional, default=1) –` Value/range by which to normalize all the biases, or if `poly_range` is provided, just the linear biases.
- `poly_range (number/pair, optional) –` Value/range by which to normalize higher-order biases.
- `ignored_terms (iterable, optional) –` Biases associated with these terms are not scaled.
- `**parameters –` Other parameters for the sampling method, specified by the child sampler.

**dimod.reference.composites.PolyScaleComposite.sample_hising**

`PolyScaleComposite.sample_hising(h, J, **kwargs)`  
Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a `BinaryPolynomial` and call `sample_poly()`.

**Parameters**
- `h (dict) –` Variable biases of the Ising problem as a dict of the form `{v: bias, ...}`, where `v` is a variable in the polynomial and `bias` its associated coefficient.
- `J (dict) –` Interaction biases of the Ising problem as a dict of the form `{(u, v, ...): bias}`, where `u, v` are spin-valued variables in the polynomial and `bias` their associated coefficient.
- `**kwargs –` See `sample_poly()` for additional keyword definitions.

**Returns** `SampleSet`

See also:  
`sample_poly(), sample_hubo()`

**dimod.reference.composites.PolyScaleComposite.sample_hubo**

`PolyScaleComposite.sample_hubo(H, **kwargs)`  
Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a `BinaryPolynomial` and then call `sample_poly()`.

**Parameters**
- `H (dict) –` Coefficients of the HUBO as a dict of the form `{(u, v, ...): bias, ...}`, where `u, v` are binary-valued variables in the polynomial and `bias` their associated coefficient.
- `**kwargs –` See `sample_poly()` for additional keyword definitions.

**Returns** `SampleSet`
See also:

sample_poly(), sample_hising()

PolyTruncateComposite

class PolyTruncateComposite(child_sampler, n, sorted_by='energy', aggregate=False)
Composite that truncates returned samples.

Post-processing is expensive and sometimes one might want to only treat the lowest-energy samples. This composite layer allows one to pre-select the samples within a multi-composite pipeline.

Parameters

- **child_sampler** (*dimod.PolySampler*) – A dimod binary polynomial sampler.
- **n** (*int*) – Maximum number of rows in the returned sample set.
- **sorted_by** (*str/None, optional, default='energy'*) – Selects the record field used to sort the samples before truncating. Note that sample order is maintained in the underlying array.
- **aggregate** (*bool, optional, default=False*) – If True, aggregate the samples before truncating.

**Note:** If `aggregate` is True, `SampleSet.record.num_occurrences` are accumulated but no other fields are.

Properties

- **PolyTruncateComposite.child** The child sampler.
- **PolyTruncateComposite.children** List of child samplers that that are used by this composite.
- **PolyTruncateComposite.parameters** A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.
- **PolyTruncateComposite.properties** A dict containing any additional information about the sampler.

**dimod.reference.composites.PolyTruncateComposite.child**

PolyTruncateComposite.child

The child sampler. First sampler in Composite.children.

Type: `Sampler`

**dimod.reference.composites.PolyTruncateComposite.children**

PolyTruncateComposite.children

List of child samplers that that are used by this composite.

Type: `list[Sampler]`
dimod.reference.composites.PolyTruncateComposite.parameters

**PolyTruncateComposite.parameters**
A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

Type dict

dimod.reference.composites.PolyTruncateComposite.properties

**PolyTruncateComposite.properties**
A dict containing any additional information about the sampler.

Type dict

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>poly</em></td>
<td>Sample from the binary polynomial and truncate output.</td>
</tr>
<tr>
<td><em>hising</em></td>
<td>Sample from a higher-order Ising model.</td>
</tr>
<tr>
<td><em>hubo</em></td>
<td>Sample from a higher-order unconstrained binary optimization problem.</td>
</tr>
</tbody>
</table>

**dimod.reference.composites.PolyTruncateComposite.sample_poly**

**PolyTruncateComposite.sample_poly**(poly, **kwargs)
Sample from the binary polynomial and truncate output.

Parameters

- **(obj***(poly**) – *BinaryPolynomial*: A binary polynomial.
- **kwargs** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

**dimod.reference.composites.PolyTruncateComposite.sample_hising**

**PolyTruncateComposite.sample_hising**(h, J, **kwargs)
Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a *BinaryPolynomial* and call *sample_poly()*.

Parameters

- **h***(dict)* – Variable biases of the Ising problem as a dict of the form *(v: bias, ...)*, where *v* is a variable in the polynomial and *bias* its associated coefficient.
- **J***(dict)* – Interaction biases of the Ising problem as a dict of the form *(u, v, ...): bias*, where *u, v* are spin-valued variables in the polynomial and *bias* their associated coefficient.
- **kwargs** – See *sample_poly()* for additional keyword definitions.

Returns *SampleSet*
See also:

`sample_poly(), sample_hubo()`

dimod.reference.composites.PolyTruncateComposite.sample_hubo

PolyTruncateComposite.**sample_hubo**(H, **kwargs)

Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a `BinaryPolynomial` and then call `sample_poly()`.

**Parameters**

- H (dict) – Coefficients of the HUBO as a dict of the form `{(u, v, ...): bias, ...}`, where u, v, are binary-valued variables in the polynomial and bias their associated coefficient.
- **kwargs – See `sample_poly()` for additional keyword definitions.

**Returns** `SampleSet`

See also:

`sample_poly(), sample_hising()`

API for Samplers and Composites

You can create your own samplers with dimod’s `Sampler` abstract base class (ABC) providing complementary methods (e.g., `sample_qubo` if only `sample_ising` is implemented), consistent responses, etc.

Properties of dimod Sampler Abstract Base Classes

The following table describes the inheritance, properties, methods/mixins of sampler ABCs.

<table>
<thead>
<tr>
<th>ABC</th>
<th>Inherits from</th>
<th>Abstract Properties</th>
<th>Abstract Methods</th>
<th>Mixins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampler</td>
<td></td>
<td>parameters, properties</td>
<td>at least one of <code>sample()</code>, <code>sample_ising()</code>, <code>sample_qubo()</code></td>
<td><code>sample()</code>, <code>sample_ising()</code>, <code>sample_qubo()</code></td>
</tr>
<tr>
<td>Structured</td>
<td></td>
<td>nodelist, edgelist</td>
<td>structure, adjacency</td>
<td><code>structure</code>, <code>adjacency</code></td>
</tr>
<tr>
<td>Composite</td>
<td></td>
<td>children</td>
<td></td>
<td><code>child</code></td>
</tr>
<tr>
<td>ComposedSampler</td>
<td>Composite</td>
<td>parameters, properties, children</td>
<td>at least one of <code>sample()</code>, <code>sample_ising()</code>, <code>sample_qubo()</code></td>
<td><code>sample()</code>, <code>sample_ising()</code>, <code>sample_qubo()</code>, <code>child</code></td>
</tr>
<tr>
<td>Initialized</td>
<td></td>
<td></td>
<td></td>
<td><code>parse_initial_states</code></td>
</tr>
<tr>
<td>PolySampler</td>
<td></td>
<td>parameters, properties</td>
<td><code>sample_poly()</code></td>
<td><code>sample_hising()</code>, <code>sample_hubo()</code></td>
</tr>
<tr>
<td>ComposedPolySampler</td>
<td>PolySampler</td>
<td>parameters, properties, children</td>
<td><code>sample_poly()</code></td>
<td><code>sample_hising()</code>, <code>sample_hubo()</code></td>
</tr>
</tbody>
</table>

Chapter 6. Packages
The table shows, for example, that the \texttt{Sampler} class requires that you implement the \texttt{parameters} and \texttt{properties} properties and at least one sampler method; the class provides the unimplemented methods as mixins.

### Creating a Sampler

The \texttt{Sampler} abstract base class (see \texttt{abc}) helps you create new dimod samplers.

Any new dimod sampler must define a subclass of \texttt{Sampler} that implements abstract properties \texttt{parameters} and \texttt{properties} and one of the abstract methods \texttt{sample()}, \texttt{sample\_ising()}, or \texttt{sample\_qubo()}. The \texttt{Sampler} class provides the complementary methods as mixins and ensures consistent responses.

For example, the following steps show how to easily create a dimod sampler. It is sufficient to implement a single method (in this example the \texttt{sample\_ising()} method) to create a dimod sampler with the \texttt{Sampler} class.

```python
class LinearIsingSampler(dimod.Sampler):
    def sample\_ising(self, h, J):
        sample = linear\_ising(h, J)
        energy = dimod.\texttt{ising\_energy}(sample, h, J)
        return dimod.\texttt{SampleSet}.\texttt{from\_samples}([sample], vartype='\texttt{SPIN}', energy=[energy])

@property
    def properties(self):
        return dict()

@property
    def parameters(self):
        return dict()
```

For this example, the implemented sampler \texttt{sample\_ising()} can be based on a simple placeholder function, which returns a sample that minimizes the linear terms:

```python
def linear\_ising(h, J):
    sample = {}
    for v in h:
        if h[v] < 0:
            sample[v] = 1
        else:
            sample[v] = -1
    return sample
```

The \texttt{Sampler} ABC provides the other sample methods “for free” as mixins.

```python
>>> sampler = LinearIsingSampler()
... # Implemented by class LinearIsingSampler:
>>> response = sampler.sample\_ising({'a': -1}, {})
... # Mixins provided by Sampler class:
>>> response = sampler.sample\_qubo(((a', 'a'): 1))
>>> response = sampler.sample(dimod.\texttt{BinaryQuadraticModel}.\texttt{from\_ising}({'a': -1}, {}))
```

Below is a more complex version of the same sampler, where the \texttt{properties} and \texttt{parameters} properties return non-empty dicts.
class FancyLinearIsingSampler(dimod.Sampler):
    def __init__(self):
        self._properties = {'description': 'a simple sampler that only considers the linear terms'}
        self._parameters = {'verbose': []}

    def sample_ising(self, h, J, verbose=False):
        sample = linear_ising(h, J)
        energy = dimod.ising_energy(sample, h, J)
        if verbose:
            print(sample)
        return dimod.SampleSet.from_samples([sample], energy=[energy])

@property
    def properties(self):
        return self._properties

@property
    def parameters(self):
        return self._parameters

class Sampler
    Abstract base class for dimod samplers.

    Provides all methods sample(), sample_ising(), sample_qubo() assuming at least one is implemented.

Abstract Properties

<table>
<thead>
<tr>
<th>Sampler.parameters</th>
<th>A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampler.properties</td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
</tbody>
</table>

dimod.Sampler.parameters

Sampler.parameters
    A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

    Type  dict

dimod.Sampler.properties

Sampler.properties
    A dict containing any additional information about the sampler.

    Type  dict
### Mixin Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>**Sampler.sample(bqm, <strong>parameters)</strong></td>
<td>Sample from a binary quadratic model.</td>
</tr>
<tr>
<td>**Sampler.sample_ising(h, J, <strong>parameters)</strong></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td>**Sampler.sample_qubo(Q, <strong>parameters)</strong></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

#### dimod.Sampler.sample

**Syntax**

```python
Sampler.sample(bqm, **parameters)
```

**Description**

Sample from a binary quadratic model.

This method is inherited from the `Sampler` base class.

Converts the binary quadratic model to either Ising or QUBO format and then invokes an implemented sampling method (one of `sample_ising()` or `sample_qubo()`).

**Parameters**

- **bqm** (`BinaryQuadraticModel`): A binary quadratic model.
- **kwargs** (optional): See the implemented sampling for additional keyword definitions.

**Returns**

`SampleSet`

**See also:**

`sample_ising()`, `sample_qubo()`

#### dimod.Sampler.sample_ising

**Syntax**

```python
Sampler.sample_ising(h, J, **parameters)
```

**Description**

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

- **h** (`dict/list`): Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where `v` is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **kwargs** (optional): See the implemented sampling for additional keyword definitions.

**Returns**

`SampleSet`

**See also:**

`sample()`, `sample_qubo()`

#### dimod.Sampler.sample_qubo

**Syntax**

```python
Sampler.sample_qubo(Q, **parameters)
```

Sample from a QUBO using the implemented sample method.
This method is inherited from the `Sampler` base class.

Converts the QUBO into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

- `Q (dict)` – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v` are binary-valued variables and `bias` is their associated coefficient.

- `**kwargs` – See the implemented sampling for additional keyword definitions.

**Returns** `SampleSet`

See also:

- `sample()`, `sample_ising()`

**Creating a Composed Sampler**

Samplers can be composed. The composite pattern allows layers of pre- and post-processing to be applied to binary quadratic programs without needing to change the underlying sampler implementation.
We refer to these layers as **composites**. Each composed sampler must include at least one sampler, and possibly many composites.

Each composed sampler is itself a dimod sampler with all of the included methods and parameters. In this way complex samplers can be constructed.

The dimod `ComposedSampler` abstract base class inherits from `Sampler` class its abstract methods, properties, and mixins (for example, a `sample_Ising` method) and from `Composite` class the `children` property and `child` mixin (`children` being a list of supported samplers with `child` providing the first).

**Examples**

The dimod package’s `spin_transform.py` reference example creates a composed sampler, `SpinReversalTransformComposite(Sampler, Composite)`, that performs spin reversal transforms (“gauge transformations”) as a preprocessing step for a given sampler. The reference example implements the pseudocode below:

```python
class SpinReversalTransformComposite(Sampler, Composite):
    # Updates to inherited sampler properties and parameters
    # Definition of the composite's children (i.e., supported samplers):
    children = None

    def __init__(self, child):
        self.children = [child]

    # The composite's implementation of spin-transformation functionality:
    def sample(self, bqm, num_spin_reversal_transforms=2, spin_reversal_variables=None, **kwargs):
        response = None
        # Preprocessing code that includes instantiation of a sampler:
        flipped_response = self.child.sample(bqm, **kwargs)
        return response
```

Given dimod sampler `ExactSolver` for example, the composed sampler is used as any dimod sampler, inheriting an Ising sampling method:

```python
>>> composed_sampler = dimod.SpinReversalTransformComposite(dimod.ExactSolver())
>>> h = {0: -1, 1: 1}
>>> solutions = composed_sampler.sample_ising(h, {})
>>> len(solutions) == 8
True
```

**class ComposedSampler**

Abstract base class for dimod composed samplers.

Inherits from `Sampler` and `Composite`.

**class Composite**

Abstract base class for dimod composites.

Provides the `Composite.child` mixin property and defines the `Composite.children` abstract property to be implemented. These define the supported samplers for the composed sampler.

**Abstract Properties**
Composite.children

List of child samplers that are used by this composite.

Type list[Sampler]

Mixin Properties

Composite.child

The child sampler. First sampler in Composite.children.

Type Sampler

Creating a Structured Sampler

A structured sampler can only sample from binary quadratic models with a specific graph.

For structured samplers you must implement the nodelist and edgelist properties. The Structured abstract base class provides access to the structure and adjacency properties as well as any method or properties required by the Sampler abstract base class. The bqm_structured decorator verifies that any given binary quadratic model conforms to the supported structure.

Examples

This simple example shows a structured sampler that can only sample from a binary quadratic model with two variables and one interaction.

class TwoVariablesSampler(dimod.Sampler, dimod.Structured):
    @property
def nodelist(self):
        return [0, 1]

    @property
def edgelist(self):
        return [(0, 1)]

    @property
def properties(self):
        return dict()

    @property
def parameters(self):
        return dict()
@dimod.decorators.bqm_structured
def sample(self, bqm):
    # All bqm's passed in will be a subgraph of the sampler's structure
    variable_list = list(bqm.linear)
samples = []
energies = []
    for values in itertools.product(bqm.vartype.value, repeat=len(bqm)):
        sample = dict(zip(variable_list, values))
samples.append(sample)
        energies.append(bqm.energy(sample))
    return dimod.SampleSet.from_samples(samples, bqm.vartype, energies)
    return response

>>> import itertools
>>> sampler = TwoVariablesSampler()
>>> solutions = sampler.sample_ising({}, {(0, 1): -1})
>>> solutions.first.energy
-1.0

class Structured
The abstract base class for dimod structured samplers.

  Provides the Structured.adjustency and Structured.structure properties.

  Abstract properties nodelist and edgelist must be implemented.

Abstract Properties

<table>
<thead>
<tr>
<th>Structured.nodelist</th>
<th>Nodes/variables allowed by the sampler.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured.edgelist</td>
<td>Edges/interactions allowed by the sampler in the form ( {(u, v), \ldots }).</td>
</tr>
</tbody>
</table>

dimod.Structured.nodelist

Structured.nodelist
  Nodes/variables allowed by the sampler.

    Type list

dimod.Structured.edgelist

Structured.edgelist
  Edges/interactions allowed by the sampler in the form \( \{(u, v), \ldots \}\).

    Type list

Mixin Properties
Structured.adjacency

Adjacency structure formatted as a dict, where keys are the nodes of the structured sampler and values are sets of all adjacent nodes for each key node.

**Type**  dict[variable, set]

Structured.structure

Structure of the structured sampler formatted as a namedtuple, Structure(nodelist, edgelist, adjacency), where the 3-tuple values are the nodelist, edgelist and adjacency attributes.

dimod.Structured.adjacency

Structured.adjacency

Adjacency structure formatted as a dict, where keys are the nodes of the structured sampler and values are sets of all adjacent nodes for each key node.

**Type**  dict[variable, set]

dimod.Structured.structure

Structured.structure

Structure of the structured sampler formatted as a namedtuple, Structure(nodelist, edgelist, adjacency), where the 3-tuple values are the nodelist, edgelist and adjacency attributes.

Creating a Initialized Sampler

Some samplers require or accept an initial starting point in sample-space.

See the source code for IdentitySampler for an example of using this abstract base class in a sampler.

class Initialized

Mixin Methods

Initialized.parse_initial_states(bqm, ..., )

Parses/generates initial states for an initialized sampler.

dimod.Initialized.parse_initial_states

Initialized.parse_initial_states(bqm, initial_states=None, initial_states_generator='random', num_reads=None, seed=None)

Parses/generates initial states for an initialized sampler.

**Parameters**

- **bqm** (*BinaryQuadraticModel*) – The binary quadratic model.
- **num_reads** (*int, optional, default=len(initial_states) or 1*) – Number of reads. If num_reads is not explicitly given, it is selected to match the number of initial states given. If no initial states are given, it defaults to 1.
- **initial_states** (*samples-like, optional, default=None*) – One or more samples, each defining an initial state for all the problem variables. Initial states
are given one per read, but if fewer than `num_reads` initial states are defined, additional values are generated as specified by `initial_states_generator`. See func:`as_samples` for a description of “samples-like”.

- **`initial_states_generator`** (`{'none', 'tile', 'random'}, optional, default='random'`) – Defines the expansion of `initial_states` if fewer than `num_reads` are specified:
  - ’none’: If the number of initial states specified is smaller than `num_reads`, raises ValueError.
  - ’tile’: Reuses the specified initial states if fewer than `num_reads` or truncates if greater.
  - ’random’: Expands the specified initial states with randomly generated states if fewer than `num_reads` or truncates if greater.

- **`seed`** (`int (32-bit unsigned integer), optional`) – Seed to use for the PRNG. Specifying a particular seed with a constant set of parameters produces identical results. If not provided, a random seed is chosen.

Returns: A named tuple with ['initial_states', 'initial_states_generator', 'num_reads', 'seed'] as generated by this function.

Creating a Binary Polynomial Sampler

Samplers that handle binary polynomials: problems with binary variables that are not constrained to quadratic interactions.

**class PolySampler**

Sampler that supports binary polynomials.

Binary polynomials are an extension of binary quadratic models that allow higher-order interactions.

Abstract Properties

<table>
<thead>
<tr>
<th>PolySampler.parameters</th>
<th>A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolySampler.properties</td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
</tbody>
</table>

dimod.PolySampler.parameters

PolySampler.parameters

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

Type: `dict`

dimod.PolySampler.properties

PolySampler.properties

A dict containing any additional information about the sampler.

Type: `dict`
Abstract Methods

```python
PolySampler.sample_poly(polynomial, **kwargs)
```
Sample from a higher-order polynomial.

dimod.PolySampler.sample_poly

```python
PolySampler.sample_poly(polynomial, **kwargs)
```
Sample from a higher-order polynomial.

Mixin Methods

```python
PolySampler.sample_hising(h, J, **kwargs)
```
Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a `BinaryPolynomial` and call `sample_poly()`.

Parameters

- `h (dict)` – Variable biases of the Ising problem as a dict of the form `{v: bias, ...}`, where `v` is a variable in the polynomial and `bias` its associated coefficient.
- `J (dict)` – Interaction biases of the Ising problem as a dict of the form `{(u, v, ...): bias}`, where `u, v` are spin-valued variables in the polynomial and `bias` their associated coefficient.
- `**kwargs` – See `sample_poly()` for additional keyword definitions.

Returns `SampleSet`

See also:

`sample_poly(), sample_hubo()`

```python
PolySampler.sample_hubo(H, **kwargs)
```
Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a `BinaryPolynomial` and then call `sample_poly()`.

Parameters

- `H (dict)` – Coefficients of the HUBO as a dict of the form `{(u, v, ...): bias, ...}`, where `u, v` are binary-valued variables in the polynomial and `bias` their associated coefficient.
- `**kwargs` – See `sample_poly()` for additional keyword definitions.

Returns `SampleSet`
Creating a Composed Binary Polynomial Sampler

class ComposedPolySampler
    Abstract base class for dimod composed polynomial samplers.
    Inherits from PolySampler and Composite.

Samples

Returned solutions and samples are described under Binary Quadratic Models.

sample_like Objects

<table>
<thead>
<tr>
<th>as_samples(samples_like[, dtype, copy, order])</th>
<th>Convert a samples_like object to a NumPy array and list of labels.</th>
</tr>
</thead>
</table>

dimod.as_samples

as_samples (samples_like, dtype=None, copy=False, order='C')
    Convert a samples_like object to a NumPy array and list of labels.

Parameters

- samples_like (samples_like) – A collection of raw samples. samples_like is an extension of NumPy’s array_like structure. See examples below.
- dtype (data-type, optional) – dtype for the returned samples array. If not provided, it is either derived from samples_like, if that object has a dtype, or set to numpy.int8.
- copy (bool, optional, default=False) – If true, then samples_like is guaranteed to be copied, otherwise it is only copied if necessary.
- order (['K', 'A', 'C', 'F'], optional, default='C') – Specify the memory layout of the array. See numpy.array().

Returns

A 2-tuple containing:
    - numpy.ndarray: Samples.
    - list: Variable labels

Return type  tuple

Examples

The following examples convert a variety of samples_like objects:

NumPy arrays
>>> import numpy as np
...

```python
>>> dimod.as_samples(np.ones(5, dtype='int8'))
(array([[1, 1, 1, 1, 1]], dtype=int8), [0, 1, 2, 3, 4])

>>> dimod.as_samples(np.zeros((5, 2), dtype='int8'))
(array([[0, 0],
        [0, 0],
        [0, 0],
        [0, 0]], dtype=int8), [0, 1])
```

Lists

```python
>>> dimod.as_samples([-1, +1, -1])
(array([[-1, 1, -1]], dtype=int8), [0, 1, 2])

>>> dimod.as_samples([-1, 1, -1])
(array([[1],
        [-1]], dtype=int8), [0])
```

Dicts

```python
>>> dimod.as_samples({'a': 0, 'b': 1, 'c': 0})
(array([[0, 1, 0]], dtype=int8), ['a', 'b', 'c'])

>>> dimod.as_samples([{'a': -1, 'b': +1}, {'a': 1, 'b': 1}])
(array([[-1, 1],
        [ 1, 1]], dtype=int8), ['a', 'b'])
```

A 2-tuple containing an array_like object and a list of labels

```python
>>> dimod.as_samples(([-1, +1, -1], ['a', 'b', 'c']))
(array([[-1, 1, -1]], dtype=int8), ['a', 'b', 'c'])

>>> dimod.as_samples((np.zeros((5, 2), dtype='int8'), ['in', 'out']))
(array([[0, 0],
        [0, 0],
        [0, 0],
        [0, 0]], dtype=int8), ['in', 'out'])
```

**SampleSet**

```python
class SampleSet (record, variables, info, vartype)

Samples and any other data returned by dimod samplers.
```

**Parameters**

- `record (numpy.recarray)` – A NumPy record array. Must have ‘sample’, ‘energy’ and ‘num_occurrences’ as fields. The ‘sample’ field should be a 2D NumPy array where each row is a sample and each column represents the value of a variable.

- `variables (iterable)` – An iterable of variable labels, corresponding to columns in `record.samples`.

- `info (dict)` – Information about the `SampleSet` as a whole, formatted as a dict.

- `vartype (Vartype/str/set)` – Variable type for the `SampleSet`. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
- Vartype.BINARY, 'BINARY', {0, 1}

### Examples

This example creates a SampleSet out of a samples_like object (a NumPy array).

```python
>>> import numpy as np
... >>> sampleset = dimod.SampleSet.from_samples(np.ones(5, dtype='int8'),
...                                                'BINARY', 0)
... >>> sampleset.variables
Variables([0, 1, 2, 3, 4])
```

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleSet.first</td>
<td>Sample with the lowest-energy.</td>
</tr>
<tr>
<td>SampleSet.info</td>
<td>Dict of information about the SampleSet as a whole.</td>
</tr>
<tr>
<td>SampleSet.record</td>
<td><code>numpy.recarray</code> containing the samples, energies, number of occurrences, and other sample data.</td>
</tr>
<tr>
<td>SampleSet.variables</td>
<td><code>VariableIndexView</code> of variable labels.</td>
</tr>
<tr>
<td>SampleSet.vartype</td>
<td><code>Vartype</code> of the samples.</td>
</tr>
</tbody>
</table>

#### dimod.SampleSet.first

SampleSet.first

Sample with the lowest-energy.

Raises `ValueError` – If empty.

#### Example

```python
>>> sampleset = dimod.ExactSolver().sample_ising({'a': 1}, {('a', 'b'): 1})
... >>> sampleset.first
Sample(sample={'a': -1, 'b': 1}, energy=-2.0, num_occurrences=1)
```

#### dimod.SampleSet.info

SampleSet.info

Dict of information about the SampleSet as a whole.

### Examples

This example shows the type of information that might be returned by a dimod sampler by submitting a BQM that sets a value on a D-Wave system’s first listed coupler.

```python
>>> from dwave.system import DWaveSampler   # doctest: +SKIP
... >>> sampler = DWaveSampler()             # doctest: +SKIP
... >>> bqm = dimod.BQM({}, {sampler.edgelist[0]: -1}, 0, dimod.SPIN)   # doctest:
...                                                +SKIP
```

(continues on next page)
```python
>>> sampler.sample(bqm).info  # doctest: +SKIP
{'timing': {'qpu_sampling_time': 315,
            'qpu_anneal_time_per_sample': 20,
            'qpu_readout_time_per_sample': 274,
            # Snipped above response for brevity
}
```

**dimod.SampleSet.record**

SampleSet.record

A `numpy.recarray` containing the samples, energies, number of occurrences, and other sample data.

**Examples**

```python
>>> sampler = dimod.ExactSolver()
>>> sampleset = sampler.sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b'): -1.0})
>>> sampleset.record.sample  # doctest: +SKIP
array([[-1, -1],
       [ 1, -1],
       [ 1, 1],
       [-1, 1]], dtype=int8)
>>> len(sampleset.record.energy)
4
```

**dimod.SampleSet.variables**

SampleSet.variables

A `VariableIndexView` of variable labels.

Corresponds to columns of the sample field of `SampleSet.record`.

**dimod.SampleSet.vartype**

SampleSet.vartype

A `Vartype` of the samples.

**Methods**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SampleSet.aggregate()</code></td>
<td>Create a new SampleSet with repeated samples aggregated.</td>
</tr>
<tr>
<td><code>SampleSet.append_variables(samples_like[, ...])</code></td>
<td>Create a new sampleset with the given variables and values.</td>
</tr>
<tr>
<td><code>SampleSet.change_vartype(vartype[, ...])</code></td>
<td>Return the SampleSet with the given vartype.</td>
</tr>
<tr>
<td><code>SampleSet.copy()</code></td>
<td>Create a shallow copy.</td>
</tr>
<tr>
<td><code>SampleSet.data([fields, sorted_by, name, ...])</code></td>
<td>Iterate over the data in the SampleSet.</td>
</tr>
<tr>
<td><code>SampleSet.done()</code></td>
<td>Return True if a pending computation is done.</td>
</tr>
</tbody>
</table>
Table 69 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SampleSet.from_future(future[, result_hook])</code></td>
<td>Construct a <code>SampleSet</code> referencing the result of a future computation.</td>
</tr>
<tr>
<td><code>SampleSet.from_samples(samples_like, ...)</code></td>
<td>Build a <code>SampleSet</code> from raw samples.</td>
</tr>
<tr>
<td><code>SampleSet.from_samples_bqm(samples_like, ...)</code></td>
<td>Build a sample set from raw samples and a binary quadratic model.</td>
</tr>
<tr>
<td><code>SampleSet.from_serializable(obj)</code></td>
<td>Deserialize a <code>SampleSet</code>.</td>
</tr>
<tr>
<td><code>SampleSet.lowest([rtol, atol])</code></td>
<td>Return a sample set containing the lowest-energy samples.</td>
</tr>
<tr>
<td><code>SampleSet.resolve()</code></td>
<td>Ensure that the sample set is resolved if constructed from a future.</td>
</tr>
<tr>
<td><code>SampleSet.relabel_variables(mapping[, inplace])</code></td>
<td>Relabel the variables of a <code>SampleSet</code> according to the specified mapping.</td>
</tr>
<tr>
<td><code>SampleSet.samples([n, sorted_by])</code></td>
<td>Return an iterable over the samples.</td>
</tr>
<tr>
<td><code>SampleSet.slice(*slice_args, **kwargs)</code></td>
<td>Create a new sample set with rows sliced according to standard Python slicing syntax.</td>
</tr>
<tr>
<td><code>SampleSet.to_pandas_dataframe([sample_column])</code></td>
<td>Convert a sample set to a Pandas DataFrame</td>
</tr>
<tr>
<td><code>SampleSet.to_serializable([use_bytes,...])</code></td>
<td>Convert a <code>SampleSet</code> to a serializable object.</td>
</tr>
<tr>
<td><code>SampleSet.truncate(n[, sorted_by])</code></td>
<td>Create a new sample set with up to n rows.</td>
</tr>
</tbody>
</table>

**dimod.SampleSet.aggregate**

`SampleSet.aggregate()` creates a new `SampleSet` with repeated samples aggregated.

Returns `SampleSet`

**Note:** `SampleSet.record.num_occurrences` are accumulated but no other fields are.

**Examples**

This example aggregates a sample set with two identical samples out of three.

```python
>>> sampleset = dimod.SampleSet.from_samples(            
    [[0, 0, 1], [0, 0, 1],                        
     ...,                                        
     [1, 1, 1]],                                 
    dimod.BINARY,                                
    [0, 0, 1])
>>> print(sampleset)                                
0 1 2 energy num_oc.
0 0 0 1 0 1
1 0 0 1 0 1
2 1 1 1 1 1
['BINARY', 3 rows, 3 samples, 3 variables]
>>> print(sampleset.aggregate())                    
0 1 2 energy num_oc.
0 0 0 1 0 2
1 1 1 1 1 1
['BINARY', 2 rows, 3 samples, 3 variables]
```
dimod.SampleSet.append_variables

SampleSet.append_variables(samples_like, sort_labels=True)
Create a new sample set with the given variables and values.
Not defined for empty sample sets. If samples_like is a SampleSet, its data vectors and info are ignored.

Parameters

• samples_like – Samples to add to the sample set. Either a single sample or identical in length to the sample set. ‘samples_like’ is an extension of NumPy’s array_like. See as_samples().

• sort_labels (bool, optional, default=True) – Return SampleSet.variables in sorted order. For mixed (unsortable) types, the given order is maintained.

Returns New sample set with the variables/values added.

Return type SampleSet

Examples

```python
>>> sampleset = dimod.SampleSet.from_samples([{'a': -1, 'b': +1},
...                                           {'a': +1, 'b': +1}],
...                                           dimod.SPIN,
...                                           energy=[-1.0, 1.0])
>>> new = sampleset.append_variables({'c': -1})
>>> print(new)
a b c energy num_oc.
0 -1 +1 -1.0 1
1 +1 +1 1.0 1
['SPIN', 2 rows, 2 samples, 3 variables]
```

Add variables from another sample set to the previous example. Note that the energies remain unchanged.

```python
>>> another = dimod.SampleSet.from_samples([{'c': -1, 'd': +1},
...                                           {'c': +1, 'd': +1}],
...                                           dimod.SPIN,
...                                           energy=[-2.0, 1.0])
>>> new = sampleset.append_variables(another)
>>> print(new)
a b c d energy num_oc.
0 -1 +1 -1 1.0 1
1 +1 +1 1.0 1
['SPIN', 2 rows, 2 samples, 4 variables]
```

dimod.SampleSet.change_vartype

SampleSet.change_vartype(vartype, energy_offset=0.0, inplace=True)
Return the SampleSet with the given vartype.

Parameters

• vartype (Vartype/str/set) – Variable type to use for the new SampleSet. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
- `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

- **energy_offset** *(number, optional, default=0.0)* – Constant value applied to the ‘energy’ field of `SampleSet.record`.

- **inplace** *(bool, optional, default=True)* – If True, the instantiated `SampleSet` is updated; otherwise, a new `SampleSet` is returned.

**Returns** SampleSet with changed vartype. If `inplace` is True, returns itself.

**Return type** `SampleSet`

**Notes**

This function is non-blocking unless `inplace==True`, in which case the sample set is resolved.

**Examples**

This example creates a binary copy of a spin-valued `SampleSet`.

```python
>>> sampleset = dimod.ExactSolver().sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b'): -1})
>>> sampleset_binary = sampleset.change_vartype(dimod.BINARY, energy_offset=1.0, inplace=False)
>>> sampleset_binary.vartype is dimod.BINARY
True
>>> sampleset_binary.first.sample
{'a': 0, 'b': 0}
```

**dimod.SampleSet.copy**

`SampleSet.copy()`

Create a shallow copy.

**dimod.SampleSet.data**

`SampleSet.data` *(fields=*, sorted_by='energy', name='Sample', reverse=False, sample_dict_cast=True, index=False)*

Iterate over the data in the `SampleSet`.

**Parameters**

- **fields** *(list, optional, default=*) – If specified, only these fields are included in the yielded tuples. The special field name ‘sample’ can be used to view the samples.

- **sorted_by** *(str/None, optional, default='energy')* – Selects the record field used to sort the samples. If None, the samples are yielded in record order.

- **name** *(str/None, optional, default='Sample')* – Name of the yielded namedtuples or None to yield regular tuples.

- **reverse** *(bool, optional, default=False)* – If True, yield in reverse order.
• **sample_dict_cast** *(bool, optional, default=True)* – Samples are returned as dicts rather than `SampleView`, which requires heavy memory usage. Set to False to reduce load on memory.

• **index** *(bool, optional, default=False)* – If True, `datum.idx` gives the corresponding index of the `SampleSet.record`.

Yields *namedtuple/tuple* – The data in the `SampleSet`, in the order specified by the input `fields`.

### Examples

```python
>>> sampleset = dimod.ExactSolver().sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b ->'): -1})

>>> for datum in sampleset.data(fields=['sample', 'energy']):  # doctest: +SKIP
...     print(datum)
Sample(sample={'a': -1, 'b': -1}, energy=-1.5)
Sample(sample={'a': 1, 'b': -1}, energy=-0.5)
Sample(sample={'a': 1, 'b': 1}, energy=-0.5)
Sample(sample={'a': -1, 'b': 1}, energy=2.5)

>>> for energy, in sampleset.data(fields=['energy'], sorted_by='energy'):
...     print(energy)
...     -1.5
-0.5
-0.5
2.5

>>> print(next(sampleset.data(fields=['energy'], name='ExactSolverSample')))
ExactSolverSample(energy=-1.5)
```

**dimod.SampleSet.done**

`SampleSet.done()`

Return True if a pending computation is done.

Used when a `SampleSet` is constructed with `SampleSet.from_future()`.

### Examples

This example uses a `Future` object directly. Typically a `Executor` sets the result of the future (see documentation for `concurrent.futures`).

```python
>>> from concurrent.futures import Future
... >>> future = Future()
... >>> sampleset = dimod.SampleSet.from_future(future)
... >>> future.done()  # doctest: +SKIP
False
... >>> future.set_result(dimod.ExactSolver().sample_ising({0: -1}, {}))
... >>> future.done()  # doctest: +SKIP
True
... >>> sampleset.first.energy
-1.0
```
dimod.SampleSet.from_future

classmethod SampleSet.from_future(future, result_hook=None)

Construct a SampleSet referencing the result of a future computation.

Parameters

• future (object) – Object that contains or will contain the information needed to construct a SampleSet. If future has a done() method, this determines the value returned by SampleSet.done().

• result_hook (callable, optional) – A function that is called to resolve the future. Must accept the future and return a SampleSet. If not provided, set to

```python
def result_hook(future):
    return future.result()
```

Returns SampleSet

Notes

The future is resolved on the first read of any of the SampleSet properties.

Examples

Run a dimod sampler on a single thread and load the returned future into SampleSet.

```python
>>> from concurrent.futures import ThreadPoolExecutor
... bqm = dimod.BinaryQuadraticModel.from_ising({}, {('a', 'b'): -1})
>>> with ThreadPoolExecutor(max_workers=1) as executor:
...     future = executor.submit(dimod.ExactSolver().sample, bqm)
...     sampleset = dimod.SampleSet.from_future(future)
>>> sampleset.first.energy  # doctest: +SKIP
```

dimod.SampleSet.from_samples

classmethod SampleSet.from_samples(samples_like, vartype, energy, info=None, num_occurrences=None, aggregate_samples=False, sort_labels=True, **vectors)

Build a SampleSet from raw samples.

Parameters

• samples_like – A collection of raw samples. ‘samples_like’ is an extension of NumPy’s array_like. See as_samples().

• vartype (Vartype/str/set) – Variable type for the SampleSet. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}

• energy (array_like) – Vector of energies.

• info (dict, optional) – Information about the SampleSet as a whole formatted as a dict.
• **num_occurrences** (*array_like, optional*) – Number of occurrences for each sample. If not provided, defaults to a vector of 1s.

• **aggregate_samples** (*bool, optional, default=False*) – If True, all samples in returned *SampleSet* are unique, with *num_occurrences* accounting for any duplicate samples in *samples_like*.

• **sort_labels** (*bool, optional, default=True*) – Return *SampleSet* variables in sorted order. For mixed (unsortable) types, the given order is maintained.

• **vectors** (*array_like*) – Other per-sample data.

**Returns** *SampleSet*

### Examples

This example creates a SampleSet out of a samples_like object (a dict).

```python
>>> import numpy as np
...
>>> sampleset = dimod.SampleSet.from_samples(
...    dimod.as_samples({'a': 0, 'b': 1, 'c': 0}), 'BINARY', 0)
>>> sampleset.variables
Variables(['a', 'b', 'c'])
```

**dimod.SampleSet.from_samples_bqm**

**classmethod** *SampleSet.from_samples_bqm*(samples_like, bqm, **kwargs)*

Build a sample set from raw samples and a binary quadratic model.

The binary quadratic model is used to calculate energies and set the *vartype*.

**Parameters**

• **samples_like** – A collection of raw samples. ‘samples_like’ is an extension of NumPy’s array_like. See *as_samples()*.

• **bqm** (*BinaryQuadraticModel*) – A binary quadratic model.

• **info** (*dict, optional*) – Information about the *SampleSet* as a whole formatted as a dict.

• **num_occurrences** (*array_like, optional*) – Number of occurrences for each sample. If not provided, defaults to a vector of 1s.

• **aggregate_samples** (*bool, optional, default=False*) – If True, all samples in returned *SampleSet* are unique, with *num_occurrences* accounting for any duplicate samples in *samples_like*.

• **sort_labels** (*bool, optional, default=True*) – Return *SampleSet*.variables in sorted order. For mixed (unsortable) types, the given order is maintained.

• **vectors** (*array_like*) – Other per-sample data.

**Returns** *SampleSet*
Examples

```python
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {('a', 'b'): -1})
>>> sampleset = dimod.SampleSet.from_samples_bqm({'a': -1, 'b': 1}, bqm)
```

dimod.SampleSet.from_serializable

classmethod SampleSet.from_serializable(obj)

Deserialize a SampleSet.

Parameters

- obj (dict) – A SampleSet serialized by to_serializable().

Returns

- SampleSet

Examples

This example encodes and decodes using JSON.

```python
>>> import json
... >>> samples = dimod.SampleSet.from_samples([-1, 1, -1], dimod.SPIN, energy=-.5)
>>> s = json.dumps(samples.to_serializable())
>>> new_samples = dimod.SampleSet.from_serializable(json.loads(s))
```

See also:

to_serializable()

dimod.SampleSet.lowest

SampleSet.lowest (rtol=1e-05, atol=1e-08)

Return a sample set containing the lowest-energy samples.

A sample is included if its energy is within tolerance of the lowest energy in the sample set. The following equation is used to determine if two values are equivalent:

$$\text{absolute}(a - b) \leq (\text{atol} + \text{rtol} * \text{absolute}(b))$$

See numpy.isclose() for additional details and caveats.

Parameters

- rtol (float, optional, default=1.e-5) – The relative tolerance (see above).
- atol (float, optional, default=1.e-8) – The absolute tolerance (see above).

Returns

- A new sample set containing the lowest energy samples as delimited by configured tolerances from the lowest energy sample in the current sample set.

Return type

SampleSet

Examples
>>> sampleset = dimod.ExactSolver().sample_ising({'a': .001},
...       {'a', 'b': -1})
>>> print(sampleset.lowest())
a  b  energy  num_oc.
0 -1 -1 -1.001   1
['SPIN', 1 rows, 1 samples, 2 variables]
>>> print(sampleset.lowest(atol=.1))
a  b  energy  num_oc.
0 -1 -1 -1.001   1
1  +1  +1  -0.999   1
['SPIN', 2 rows, 2 samples, 2 variables]

**Note:** “Lowest energy” is the lowest energy in the sample set. This is not always the “ground energy” which is the lowest energy possible for a binary quadratic model.

**dimod.SampleSet.resolve**

SampleSet.resolve()
Ensure that the sampleset is resolved if constructed from a future.

**dimod.SampleSet.relabel_variables**

SampleSet.relabel_variables(mapping, inplace=True)
Relabel the variables of a SampleSet according to the specified mapping.

**Parameters**

- **mapping** (dict) – Mapping from current variable labels to new, as a dict. If incomplete mapping is specified, unmapped variables keep their current labels.
- **inplace** (bool, optional, default=True) – If True, the current SampleSet is updated; otherwise, a new SampleSet is returned.

**Returns** SampleSet with relabeled variables. If inplace is True, returns itself.

**Return type** SampleSet

**Notes**

This function is non-blocking unless inplace=True, in which case the sample set is resolved.

**Examples**

This example creates a relabeled copy of a SampleSet.

```python
>>> sampleset = dimod.ExactSolver().sample_ising({a: -0.5, 'b': 1.0}, {'a', 'b' -> -1})
>>> new_sampleset = sampleset.relabel_variables({a: 0, 'b': 1}, inplace=False)
>>> new_sampleset.variables
Variables([0, 1])
```
**dimod.SampleSet.samples**

`SampleSet.samples(n=None, sorted_by='energy')`  
Return an iterable over the samples.

**Parameters**

- `n (int, optional, default=None)` – Maximum number of samples to return in the view.
- `sorted_by (str/None, optional, default='energy')` – Selects the record field used to sort the samples. If None, samples are returned in record order.

**Returns** A view object mapping variable labels to values.

**Return type** SamplesArray

**Examples**

```python
>>> sampleset = dimod.ExactSolver().sample_ising({'a': 0.1, 'b': 0.0},
... {('a', 'b'): 1})
>>> for sample in sampleset.samples():  # doctest: +SKIP
...     print(sample)
{'a': -1, 'b': 1}
{'a': 1, 'b': -1}
{'a': -1, 'b': -1}
{'a': 1, 'b': 1}

>>> sampleset = dimod.ExactSolver().sample_ising({'a': 0.1, 'b': 0.0},
... {('a', 'b'): 1})
>>> samples = sampleset.samples()
>>> samples[0]
{'a': -1, 'b': 1}
>>> samples[0, 'a']
-1
>>> samples[0, ['b', 'a']]
array([ 1, -1], dtype=int8)
>>> samples[:1, ['a', 'b']]
array([[ 1, -1],
[-1, -1],
[ 1,  1]], dtype=int8)
```

**dimod.SampleSet.slice**

`SampleSet.slice(*slice_args, **kwargs)`  
Create a new sample set with rows sliced according to standard Python slicing syntax.

**Parameters**

- `start (int, optional, default=None)` – Start index for slice.
- `stop (int)` – Stop index for slice.
- `step (int, optional, default=None)` – Step value for slice.
- `sorted_by (str/None, optional, default='energy')` – Selects the record field used to sort the samples before slicing. Note that sorted_by determines the sample order in the returned sample set.
Returns `SampleSet`

Examples

```python
>>> import numpy as np
... >>> sampleset = dimod.SampleSet.from_samples(np.diag(range(1, 11)),
...                                            dimod.BINARY, energy=range(10))
... >>> print(sampleset)
0 1 2 3 4 5 6 7 8 9
0 1 0 0 0 0 0 0 0 0 0 0 1
1 0 1 0 0 0 0 0 0 0 0 1 1
2 0 0 1 0 0 0 0 0 0 0 2 1
3 0 0 0 1 0 0 0 0 0 0 3 1
4 0 0 0 0 1 0 0 0 0 0 4 1
5 0 0 0 0 0 1 0 0 0 0 5 1
6 0 0 0 0 0 0 1 0 0 0 6 1
7 0 0 0 0 0 0 0 1 0 0 7 1
8 0 0 0 0 0 0 0 0 1 0 8 1
9 0 0 0 0 0 0 0 0 0 1 9 1
['BINARY', 10 rows, 10 samples, 10 variables]
```

The above example's first 3 samples by energy == truncate(3):

```python
>>> print(sampleset.slice(3))
0 1 2 3 4 5 6 7 8 9
0 1 0 0 0 0 0 0 0 0 0 0 1
1 0 1 0 0 0 0 0 0 0 0 1 1
2 0 0 1 0 0 0 0 0 0 0 2 1
['BINARY', 3 rows, 3 samples, 10 variables]
```

The last 3 samples by energy:

```python
>>> print(sampleset.slice(-3, None))
0 1 2 3 4 5 6 7 8 9
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 7 1
1 0 0 0 0 0 0 0 0 1 0 0 8 1
2 0 0 0 0 0 0 0 0 0 0 0 1 9 1
['BINARY', 3 rows, 3 samples, 10 variables]
```

Every second sample in between, skipping top and bottom 3:

```python
>>> print(sampleset.slice(3, -3, 2))
0 1 2 3 4 5 6 7 8 9
0 0 0 0 0 0 0 0 0 0 0 1 3 1
1 0 0 0 0 0 0 0 0 0 0 0 5 1
['BINARY', 2 rows, 2 samples, 10 variables]
```

dimod.SampleSet.to_pandas_dataframe

`SampleSet.to_pandas_dataframe(sample_column=False)`

Convert a sample set to a Pandas DataFrame

Parameters

- `sample_column` *(bool, optional, default=False)* – If True, samples are
• as a column of type dict. \((\text{represented})\)

Returns \texttt{pandas.DataFrame}

**Examples**

```python
>>> samples = dimod.SampleSet.from_samples([[ 'a': -1, 'b': +1, 'c': -1],
...                                    ('a': -1, 'b': -1, 'c': +1)],
...                                    dimod.SPIN, energy=-.5)

```

```python
>>> samples.to_pandas_dataframe()  # doctest: +SKIP
a  b  c  energy  num_occurrences
0 -1 1 -1 -0.5  1
1 -1 -1  1 -0.5  1
```

```python
>>> samples.to_pandas_dataframe(sample_column=True)  # doctest: +SKIP
sample  energy  num_occurrences
0  {'a': -1, 'b': 1, 'c': -1} -0.5  1
1  {'a': -1, 'b': -1, 'c': 1} -0.5  1
```

dimod.SampleSet.to_serializable

\texttt{SampleSet.to_serializable}(\texttt{use\_bytes=False, bytes\_type=<class} \texttt{`bytes'>, \texttt{pack\_samples=True})

Convert a \texttt{SampleSet} to a serializable object.

Note that the contents of the \texttt{SampleSet.info} field are assumed to be serializable.

**Parameters**

- **use_bytes** (\texttt{bool, optional, default=False}) – If True, a compact representation of the biases as bytes is used.

- **bytes_type** (\texttt{class, optional, default=bytes}) – If use_bytes is True, this class is used to wrap the bytes objects in the serialization. Useful for Python 2 using BSON encoding, which does not accept the raw \texttt{bytes} type; \texttt{bson.Binary} can be used instead.

- **pack_samples** (\texttt{bool, optional, default=True}) – Pack the samples using 1 bit per sample.

**Returns** Object that can be serialized.

**Return type** \texttt{dict}

**Examples**

This example encodes using JSON.

```python
>>> import json

```

```python
>>> samples = dimod.SampleSet.from_samples([-1, 1, -1], dimod.SPIN, energy=-.5)

```

```python
>>> s = json.dumps(samples.to_serializable())
```

See also:

\texttt{from\_serializable}
dimod.SampleSet.truncate

`SampleSet.truncate(n, sorted_by='energy')`
Create a new sample set with up to n rows.

**Parameters**
- **n** *(int)* – Maximum number of rows in the returned sample set. Does not return any rows above this limit in the original sample set.
- **sorted_by**(str/None, optional, default='energy') – Selects the record field used to sort the samples before truncating. Note that this sort order is maintained in the returned sample set.

**Returns** `SampleSet`

**Examples**

```python
>>> import numpy as np
...
>>> sampleset = dimod.SampleSet.from_samples(np.ones((5, 5)), dimod.SPIN,
˓→energy=5)
>>> print(sampleset)
  0 1 2 3 4 energy num_oc.
0 +1 +1 +1 +1 +1 5 1
1 +1 +1 +1 +1 +1 5 1
2 +1 +1 +1 +1 +1 5 1
3 +1 +1 +1 +1 +1 5 1
4 +1 +1 +1 +1 +1 5 1
['SPIN', 5 rows, 5 samples, 5 variables]
>>> print(sampleset.truncate(2))
  0 1 2 3 4 energy num_oc.
0 +1 +1 +1 +1 +1 5 1
1 +1 +1 +1 +1 +1 5 1
['SPIN', 2 rows, 2 samples, 5 variables]
```

See: `SampleSet.slice()`

**Utility Functions**

`concatenate(samplesets[, defaults])` Combine sample sets.

**dimod.concatenate**

`concatenate(samplesets, defaults=None)` Combine sample sets.

**Parameters**
- **samplesets** *(iterable[SampleSet]*) – Iterable of sample sets.
- **defaults** *(dict, optional)* – Dictionary mapping data vector names to the corresponding default values.

**Returns** A sample set with the same vartype and variable order as the first given in `samplesets`. 
Return type  \texttt{SampleSet}

**Examples**

```python
>>> a = dimod.SampleSet.from_samples(([-1, +1], 'ab'), dimod.SPIN, energy=-1)
>>> b = dimod.SampleSet.from_samples(([-1, +1], 'ba'), dimod.SPIN, energy=-1)
>>> ab = dimod.concatenate((a, b))
>>> ab.record.sample
array([[-1, 1],
       [ 1, -1]], dtype=int8)
```

**Higher-Order Models**

Sometimes it is nice to work with problems that are not restricted to quadratic interactions.

**Binary Polynomials**

```python
class BinaryPolynomial (poly, vartype)
```

A polynomial with binary variables and real-valued coefficients.

**Parameters**

- **poly** (mapping/iterable) – Polynomial as a mapping of form \{term: bias, \ldots\}, where \textit{term} is a collection of variables and \textit{bias} the associated bias. It can also be an iterable of 2-tuples (term, bias).
- **vartype** (Vartype/str/set) – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN, 'SPIN', \{-1, 1\}
  - Vartype.BINARY, 'BINARY', \{0, 1\}

**degree**

The degree of the polynomial.

Type int

**variables**

The variables.

Type set

**vartype**

One of Vartype.SPIN or Vartype.BINARY.

Type Vartype

**Examples**

Binary polynomials can be constructed in many different ways. The following are all equivalent

```python
>>> poly = dimod.BinaryPolynomial({'a': -1, 'ab': 1}, dimod.SPIN)
>>> poly = dimod.BinaryPolynomial({('a',): -1, ('a', 'b'): 1}, dimod.SPIN)
>>> poly = dimod.BinaryPolynomial([('a', -1), (('a', 'b'), 1)], dimod.SPIN)
>>> poly = dimod.BinaryPolynomial({'a': -1, 'ab': .5, 'ba': .5}, dimod.SPIN)
```
Binary polynomials act a mutable mappings but the terms can be accessed with any sequence.

```python
>>> poly = dimod.BinaryPolynomial({'a': -1, 'ab': 1}, dimod.BINARY)
>>> poly['ab']
1
>>> poly['ba']
1
>>> poly[{'a', 'b'}]
1
>>> poly[('a', 'b')]
1
>>> poly['cd'] = 4
>>> poly['dc']
4
```

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>copy()</code></td>
<td>Create a shallow copy.</td>
</tr>
<tr>
<td><code>energies(samples_like[, dtype])</code></td>
<td>The energies of the given samples.</td>
</tr>
<tr>
<td><code>energy(sample_like[, dtype])</code></td>
<td>The energy of the given sample.</td>
</tr>
<tr>
<td><code>from_hising(h, J[, offset])</code></td>
<td>Construct a binary polynomial from a higher-order Ising problem.</td>
</tr>
<tr>
<td><code>from_hubo(H[, offset])</code></td>
<td>Construct a binary polynomial from a higher-order unconstrained binary optimization (HUBO) problem.</td>
</tr>
<tr>
<td><code>normalize([bias_range, ...])</code></td>
<td>Normalizes the biases of the binary polynomial such that they fall in the provided range(s).</td>
</tr>
<tr>
<td><code>relabel_variables(mapping)</code></td>
<td>Relabel variables of a binary polynomial as specified by mapping.</td>
</tr>
<tr>
<td><code>scale(scalar[, ignored_terms])</code></td>
<td>Multiply the polynomial by the given scalar.</td>
</tr>
<tr>
<td><code>to_binary([copy])</code></td>
<td>Return a binary polynomial over {0, 1} variables.</td>
</tr>
<tr>
<td><code>to_hising()</code></td>
<td>Construct a higher-order Ising problem from a binary polynomial.</td>
</tr>
<tr>
<td><code>to_hubo()</code></td>
<td>Construct a higher-order unconstrained binary optimization (HUBO) problem from a binary polynomial.</td>
</tr>
<tr>
<td><code>to_spin([copy])</code></td>
<td>Return a binary polynomial over {-1, +1} variables.</td>
</tr>
</tbody>
</table>

**dimod.higherorder.polynomial.BinaryPolynomial.copy**

`BinaryPolynomial.copy()`  
Create a shallow copy.

**dimod.higherorder.polynomial.BinaryPolynomial.energies**

`BinaryPolynomial.energies(samples_like[, dtype=<class 'float'>])`  
The energies of the given samples.

**Parameters**
• **samples_like** (*samples_like*) – A collection of raw samples. *samples_like* is an extension of NumPy’s array_like structure. See *as_samples()*.

• **dtype** (*numpy.dtype*, optional) – The data type of the returned energies. Defaults to float.

Returns The energies.

Return type *numpy.ndarray*

**dimod.higherorder.polynomial.BinaryPolynomial.energy**

*BinaryPolynomial.energy* (*sample_like, dtype=<class 'float'>*)

The energy of the given sample.

Parameters

• **sample_like** (*samples_like*) – A raw sample. *sample_like* is an extension of NumPy’s array_like structure. See *as_samples()*.

• **dtype** (*numpy.dtype*, optional) – The data type of the returned energies. Defaults to float.

Returns The energy.

**dimod.higherorder.polynomial.BinaryPolynomial.from_hising**

classmethod *BinaryPolynomial.from_hising* (*h, J, offset=None*)

Construct a binary polynomial from a higher-order Ising problem.

Parameters

• **h** (*dict*) – The linear biases.

• **J** (*dict*) – The higher-order biases.

• **offset** (optional, default=0.0) – Constant offset applied to the model.

Returns *BinaryPolynomial*

Examples

```python
>>> poly = dimod.BinaryPolynomial.from_hising({'a': 2}, {'ab': -1}, 0)
>>> poly.degree
2
```

**dimod.higherorder.polynomial.BinaryPolynomial.from_hubo**

classmethod *BinaryPolynomial.from_hubo* (*H, offset=None*)

Construct a binary polynomial from a higher-order unconstrained binary optimization (HUBO) problem.

Parameters

• **H** (*dict*) – Coefficients of a higher-order unconstrained binary optimization (HUBO) model.

Returns *BinaryPolynomial*
Examples

```python
>>> poly = dimod.BinaryPolynomial.from_hubo({('a', 'b', 'c'): -1})
>>> poly.degree
3
```

dimod.higherorder.polynomial.BinaryPolynomial.normalize

**BinaryPolynomial.normalize** *(bias_range=1, poly_range=None, ignored_terms=None)*

Normalizes the biases of the binary polynomial such that they fall in the provided range(s).

If *poly_range* is provided, then *bias_range* will be treated as the range for the linear biases and *poly_range* will be used for the range of the other biases.

**Parameters**

- **bias_range** *(number/pair)* – Value/range by which to normalize the all the biases, or if *poly_range* is provided, just the linear biases.
- **poly_range** *(number/pair, optional)* – Value/range by which to normalize the higher order biases.
- **ignored_terms** *(iterable, optional)* – Biases associated with these terms are not scaled.

dimod.higherorder.polynomial.BinaryPolynomial.relabel_variables

**BinaryPolynomial.relabel_variables** *(mapping, inplace=True)*

Relabel variables of a binary polynomial as specified by mapping.

**Parameters**

- **mapping** *(dict)* – Dict mapping current variable labels to new ones. If an incomplete mapping is provided, unmapped variables retain their current labels.
- **inplace** *(bool, optional, default=True)* – If True, the binary polynomial is updated in-place; otherwise, a new binary polynomial is returned.

**Returns** A binary polynomial with the variables relabeled. If *inplace* is set to True, returns itself.

**Return type** BinaryPolynomial

dimod.higherorder.polynomial.BinaryPolynomial.scale

**BinaryPolynomial.scale** *(scalar, ignored_terms=None)*

Multiply the polynomial by the given scalar.

**Parameters**

- **scalar** *(number)* – Value to multiply the polynomial by.
- **ignored_terms** *(iterable, optional)* – Biases associated with these terms are not scaled.
dimod.higherorder.polynomial.BinaryPolynomial.to_binary

```
BinaryPolynomial.to_binary(copy=False)
```

Return a binary polynomial over \([0, 1]\) variables.

**Parameters**
*copy* (optional, default=False) – If True, the returned polynomial is always a copy. Otherwise, if the polynomial is binary-valued already it returns itself.

**Returns**
*BinaryPolynomial*

dimod.higherorder.polynomial.BinaryPolynomial.to_hising

```
BinaryPolynomial.to_hising()
```

Construct a higher-order Ising problem from a binary polynomial.

**Returns**
A 3-tuple of the form \((h, J, offset)\) where \(h\) includes the linear biases, \(J\) has the higher-order biases and \(offset\) is the linear offset.

**Return type**
*tuple*

**Examples**

```python
>>> poly = dimod.BinaryPolynomial({'a': -1, 'ab': 1, 'abc': -1}, dimod.SPIN)
>>> h, J, off = poly.to_hising()
>>> h
{'a': -1}
```

dimod.higherorder.polynomial.BinaryPolynomial.to_hubo

```
BinaryPolynomial.to_hubo()
```

Construct a higher-order unconstrained binary optimization (HUBO) problem from a binary polynomial.

**Returns**
A 2-tuple of the form \((H, offset)\) where \(H\) is the HUBO and \(offset\) is the linear offset.

**Return type**
tuple

dimod.higherorder.polynomial.BinaryPolynomial.to_spin

```
BinaryPolynomial.to_spin(copy=False)
```

Return a binary polynomial over \([-1, 1]\) variables.

**Parameters**
*copy* (optional, default=False) – If True, the returned polynomial is always a copy. Otherwise, if the polynomial is spin-valued already it returns itself.

**Returns**
*BinaryPolynomial*

Reducing to a Binary Quadratic Model

```
make_quadratic(poly, strength[, vartype, bqm])
```

Create a binary quadratic model from a higher order polynomial.
make_quadratic (poly, strength, vartype=None, bqm=None)

Create a binary quadratic model from a higher order polynomial.

Parameters

- **poly** (*dict*) – Polynomial as a dict of form `{term: bias, ...}`, where `term` is a tuple of variables and `bias` the associated bias.

- **strength** (*float*) – The energy penalty for violating the product constraint. Insufficient strength can result in the binary quadratic model not having the same minimizations as the polynomial.

- **vartype** (*Vartype*/str/set, optional) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{−1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

  If `bqm` is provided, `vartype` is not required.

- **bqm** (*BinaryQuadraticModel*, optional) – The terms of the reduced polynomial are added to this binary quadratic model. If not provided, a new binary quadratic model is created.

Returns *BinaryQuadraticModel*

Examples

```python
>>> poly = {(0,): -1, (1,): 1, (2,): 1.5, (0, 1): -1, (0, 1, 2): -2}
>>> bqm = dimod.make_quadratic(poly, 5.0, dimod.SPIN)
```

Utilities

Contents

- **Utilities**
  - Decorators
  - Energy Calculations
  - Graph-like
  - Serialization
    - COOrdinate
    - FileView
    - JSON
  - Testing
    - API Asserts
    - Correctness Asserts
Decorators

Decorators can be imported from the `dimod.decorators` namespace. For example:

```python
>>> from dimod.decorators import vartype_argument
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bqm_index_labels(f)</code></td>
<td>Decorator to convert a BQM to index-labels and relabel the sample set output.</td>
</tr>
<tr>
<td><code>bqm_index_labelled_input(...)</code></td>
<td>Returns a decorator that ensures BQM variable labeling and specified sample_like inputs are index labeled and consistent.</td>
</tr>
<tr>
<td><code>bqm_structured(f)</code></td>
<td>Decorator to raise an error if the given BQM does not match the sampler's structure.</td>
</tr>
<tr>
<td><code>graph_argument(*arg_names, **options)</code></td>
<td>Decorator to coerce given graph arguments into a consistent form.</td>
</tr>
<tr>
<td><code>vartype_argument(*arg_names)</code></td>
<td>Ensures the wrapped function receives valid vartype argument(s).</td>
</tr>
</tbody>
</table>

### dimod.decorators.bqm_index_labels

**bqm_index_labels(f)**

Decorater to convert a BQM to index-labels and relabel the sample set output.

Designed to be applied to `Sampler.sample()`. Expects the wrapped function or method to accept a `BinaryQuadraticModel` as the second input and to return a `SampleSet`.

### dimod.decorators.bqm_index_labelled_input

**bqm_index_labelled_input (var_labels_arg_name, samples_arg_names)**

Returns a decorator that ensures BQM variable labeling and specified sample_like inputs are index labeled and consistent.

**Parameters**

- `var_labels_arg_name (str)` – Expected name of the argument used to pass in an index labeling for the binary quadratic model (BQM).
- `samples_arg_names (list[str])` – Expected names of sample_like inputs that should be indexed by the labels passed to the `var_labels_arg_name` argument. ‘samples_like’ is an extension of NumPy’s `array_like`. See `as_samples()`.

**Returns** Function decorator.

### dimod.decorators.bqm_structured

**bqm_structured(f)**

Decorator to raise an error if the given BQM does not match the sampler's structure.
Designed to be applied to `Sampler.sample()`. Expects the wrapped function or method to accept a `BinaryQuadraticModel` as the second input and for the `Sampler` to also be `Structured`.

dimod.decorators.graph_argument

`graph_argument(*arg_names, **options)`
Decorator to coerce given graph arguments into a consistent form.

The wrapped function accepts either an integer n, interpreted as a complete graph of size n, a nodes/edges pair, a sequence of edges, or a NetworkX graph. The argument is converted into a nodes/edges 2-tuple.

Parameters

- `*arg_names` *(optional, default='G')* – Names of the arguments for input graphs.
- `allow_None` *(bool, optional, default=False)* – If True, None can be passed through as an input graph.

dimod.decorators.vartype_argument

`vartype_argument(*arg_names)`
Ensures the wrapped function receives valid vartype argument(s).

One or more argument names can be specified as a list of string arguments.

Parameters

- `*arg_names` *(list[str], argument names, optional, default='vartype')* – Names of the constrained arguments in decorated function.

Returns Function decorator.

Examples

```python
>>> from dimod.decorators import vartype_argument

>>> @vartype_argument()
... def f(x, vartype):
...     print(vartype)
...     return x
... # f(1, 'SPIN')
>>> f(1, 'SPIN')
Vartype.SPIN
>>> f(1, vartype='SPIN')
Vartype.SPIN

>>> @vartype_argument('y')
... def f(x, y):
...     print(y)
... # f(1, 'SPIN')
>>> f(1, 'SPIN')
Vartype.SPIN
>>> f(1, y='SPIN')
Vartype.SPIN
```
```python
>>> @vartype_argument('z')
... def f(x, **kwargs):
...     print(kwargs['z'])
...
>>> f(1, z='SPIN')
Vartype.SPIN
```

**Note:** The decorated function can explicitly list (name) vartype arguments constrained by `vartype_argument()` or it can use a keyword arguments `dict`.

See also: `as_vartype()`

### Energy Calculations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ising_energy(sample, h, J, offset)</code>:</td>
<td>Calculate the energy for the specified sample of an Ising model.</td>
</tr>
<tr>
<td><code>qubo_energy(sample, Q, offset)</code>:</td>
<td>Calculate the energy for the specified sample of a QUBO model.</td>
</tr>
</tbody>
</table>

#### dimod.utilities.ising_energy

**ising_energy**(sample, h, J, offset=0.0)

Calculate the energy for the specified sample of an Ising model.

Energy of a sample for a binary quadratic model is defined as a sum, offset by the constant energy offset associated with the model, of the sample multiplied by the linear bias of the variable and all its interactions. For an Ising model,

\[
E(s) = \sum_v h_v s_v + \sum_{u,v} J_{u,v} s_u s_v + c
\]

where \(s_v\) is the sample, \(h_v\) is the linear bias, \(J_{u,v}\) the quadratic bias (interactions), and \(c\) the energy offset.

**Parameters**

- **sample** *(dict [variable, spin]*) – Sample for a binary quadratic model as a dict of form \{v: spin, \ldots\}, where keys are variables of the model and values are spins (either -1 or 1).

- **h** *(dict [variable, bias]*) – Linear biases as a dict of the form \{v: bias, \ldots\}, where keys are variables of the model and values are biases.

- **J** *(dict [(variable, variable), bias]*) – Quadratic biases as a dict of the form \{(u, v): bias, \ldots\}, where keys are 2-tuples of variables of the model and values are quadratic biases associated with the pair of variables (the interaction).

- **offset** *(numeric, optional, default=0)* – Constant offset to be applied to the energy. Default 0.

**Returns** The induced energy.

**Return type** float
Notes

No input checking is performed.

Examples

This example calculates the energy of a sample representing two down spins for an Ising model of two variables that have positive biases of value 1 and are positively coupled with an interaction of value 1.

```python
>>> sample = {1: -1, 2: -1}
>>> h = {1: 1, 2: 1}
>>> J = {(1, 2): 1}
>>> dimod.ising_energy(sample, h, J, 0.5)
-0.5
```

References

Ising model on Wikipedia

dimod.utilities.qubo_energy

`qubo_energy(sample, Q, offset=0.0)`

Calculate the energy for the specified sample of a QUBO model.

Energy of a sample for a binary quadratic model is defined as a sum, offset by the constant energy offset associated with the model, of the sample multiplied by the linear bias of the variable and all its interactions. For a quadratic unconstrained binary optimization (QUBO) model,

\[
E(x) = \sum_{u, v} Q_{u, v} x_u x_v + c
\]

where \(x_v\) is the sample, \(Q_{u, v}\) a matrix of biases, and \(c\) the energy offset.

Parameters

- `sample` *(dict [variable, spin])*
  Sample for a binary quadratic model as a dict of form `{v: bin, ...}`, where keys are variables of the model and values are binary (either 0 or 1).
- `Q` *(dict [(variable, variable), coefficient])*
  QUBO coefficients in a dict of form `{(u, v): coefficient, ...}`, where keys are 2-tuples of variables of the model and values are biases associated with the pair of variables. Tuples `(u, v)` represent interactions and `(v, v)` linear biases.
- `offset` *(numeric, optional, default=0)*
  Constant offset to be applied to the energy. Default 0.

Returns

The induced energy.

Return type

float

Notes

No input checking is performed.
Examples

This example calculates the energy of a sample representing two zeros for a QUBO model of two variables that have positive biases of value 1 and are positively coupled with an interaction of value 1.

```python
>>> sample = {1: 0, 2: 0}
>>> Q = {(1, 1): 1, (2, 2): 1, (1, 2): 1}
>>> dimod.qubo_energy(sample, Q, 0.5)
0.5
```

References

QUBO model on Wikipedia

Graph-like

```
class child_structure_dfs(sampler[, seen])
```

Return the structure of a composed sampler using a depth-first search on its children.

```
dimod.utilities.child_structure_dfs
```

`child_structure_dfs(sampler, seen=None)`

Return the structure of a composed sampler using a depth-first search on its children.

Parameters

- `sampler (Sampler)` – Structured or composed sampler with at least one structured child.
- `seen (set, optional, default=False)` – IDs of already checked child samplers.

Returns

A named tuple of the form `Structure(nodelist, edgelist, adjacency)`, where the 3-tuple values are the `Structured.nodelist, Structured.edgelist` and `Structured.adjacency` attributes of the first structured sampler found.

Return type

tuple

Raises

- `ValueError` – If no structured sampler is found.

Examples:

```python
>>> sampler = dimod.TrackingComposite(
...     ...
...     
...     ...
...     dimod.ExactSolver(), [0, 1], [{0, 1}])
>>> print(dimod.child_structure_dfs(sampler).nodelist)
[0, 1]
```

Serialization

**COOrdinate**

A simple text encoding of dimod BQMs.

6.1. dimod
The COOrdinate list is a sparse matrix representation which can be used to store binary quadratic models. This format is best used when readability is important.

Note that this format only works for BQMs that are labelled with positive integers.

**Examples**

```python
>>> from dimod.serialization import coo

Serialize a QUBO.

```coo.dumps(bqm)`

```0 0 -1.000000
0 1 1.000000
1 2 -4.500000
```

We can also include the vartype as a header.

```coo.dumps(bqm, vartype_header=True)`

# vartype=BINARY
```

Loading from a COO string. Note that you must specify a vartype.

```coo.loads(coo_string, vartype=dimod.BINARY)`

```

Or provide the vartype as a header.

```coo.loads(coo_string)`

```

```
dump(bqm, fp[, vartype_header]) Dump a binary quadratic model to a file in COOrdinate format.

dumps(bqm[, vartype_header]) Dump a binary quadratic model to a string in COOrdinate format.
```

Continued on next page
**Table 76 – continued from previous page**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>load(fp[, cls, vartype])</td>
<td>Load a COOrdinate formatted binary quadratic model from a file.</td>
</tr>
<tr>
<td>loads(s[, cls, vartype])</td>
<td>Load a COOrdinate formatted binary quadratic model from a string.</td>
</tr>
</tbody>
</table>

**dimod.serialization.coo.dump**

```python
dump (bqm, fp, vartype_header=False)
```

Dump a binary quadratic model to a file in COOrdinate format.

**dimod.serialization.coo.dumps**

```python
dumps (bqm, vartype_header=False)
```

Dump a binary quadratic model to a string in COOrdinate format.

**dimod.serialization.coo.load**

```python
load (fp, cls=<class 'dimod.binary_quadratic_model.BinaryQuadraticModel'>, vartype=None)
```

Load a COOrdinate formatted binary quadratic model from a file.

**dimod.serialization.coo.loads**

```python
loads (s, cls=<class 'dimod.binary_quadratic_model.BinaryQuadraticModel'>, vartype=None)
```

Load a COOrdinate formatted binary quadratic model from a string.

**FileView**

A format for saving large binary quadratic models.

This format is inspired by the NPY format.

**Extension Convention**

Binary quadratic models are typically saved using the .bqm extension.

**Format Version 1.0**

Format specification:

- The first 8 bytes are a magic string: exactly “DIMODBQM”.
- The next 1 byte is an unsigned byte: the major version of the file format.
- The next 1 byte is an unsigned byte: the minor version of the file format.
- The next 4 bytes form a little-endian unsigned int, the length of the header data HEADER_LEN.
- The next HEADER_LEN bytes form the header data. This is a json-serialized dictionary. The dictionary is exactly:
dict(shape=bqm.shape,
     dtype=bqm.dtype.name,
     itype=bqm.itype.name,
     ntype=bqm.ntype.name,
     vartype=bqm.vartype.name,
     type=type(bqm).__name__,
     variables=list(bqm.variables),
)

it is terminated by a newline character and padded with spaces to make the entire length of the entire header divisible by 16.

The binary quadratic model data comes after the header. The number of bytes can be determined by the data types and the number of variables and number of interactions (described in the shape).

The first \texttt{dtype.itemsize} bytes are the offset. The next \texttt{num\_variables} * (\texttt{ntype.itemsize} + \texttt{dtype.itemsize}) bytes are the linear data. The linear data includes the neighborhood starts and the biases. The final '2 * \texttt{num\_interactions} * (\texttt{itype.itemsize} + \texttt{dtype.itemsize}) bytes are the quadratic data. Stored as '\texttt{(outvar, bias)} pairs.

**Format Version 2.0**

In order to make the header a more reasonable length, the variable labels have been moved to the body. The \texttt{variables} field of the header dictionary now has a boolean value, making the dictionary:

\begin{verbatim}
dict(shape=bqm.shape,  
     dtype=bqm.dtype.name,  
     itype=bqm.itype.name,  
     ntype=bqm.ntype.name,  
     vartype=bqm.vartype.name,  
     type=type(bqm).__name__,  
     variables=any(v != i  
                    for  
                    i,  
                    v  
                    in  
                    enumerate(bqm.variables)),
)
\end{verbatim}

If the BQM is index-labeled, then no additional data is added. Otherwise, a new section is appended after the bias data.

The first 4 bytes are exactly “VARs”.

The next 4 bytes form a little-endian unsigned int, the length of the variables array \texttt{VARIABLES\_LENGTH}.

The next \texttt{VARIABLES\_LENGTH} bytes are a json-serialized array. As constructed by \texttt{‘json.dumps(list(bqm.variables))}. The variables section is padded with spaces to make the entire length of divisible by 16.

**Future**

If more sections are required in the future, they should be structured like the variables section from Version 2.0, i.e. a 4 byte section identifier and 4 bytes of length.

- \texttt{FileView(bqm[, version, ignore\_labels])} A seekable, readable view into a binary quadratic model.
- \texttt{load(fp[, cls])} Load a binary quadratic model from a file.
dimod.serialization.fileview.FileView

class FileView (bqm, version=(1, 0), ignore_labels=False)
   A seekable, readable view into a binary quadratic model.

   Parameters
   - `bqm` (BQM) – The binary quadratic model.
   - `version` (int/tuple, default=1) – The serialization version to use. Either as an integer defining the major version, or as a tuple, (major, minor).
   - `ignore_labels` (bool, default=False) – Treat the BQM as unlabelled. This is useful for large BQMs to save on space. `ignore_labels=True` is only supported in version 2.0+, trying to set it with version 1.x will raise a `ValueError`.

   Note: Currently the BQM is not locked while the file view is open, in the future this will change.

   __init__ (bqm, version=(1, 0), ignore_labels=False)
      Initialize self. See help(type(self)) for accurate signature.

   Methods

   __init__ (bqm[, version, ignore_labels]) Initialize self.
   close() Close the file view.
   fileno Returns underlying file descriptor if one exists.
   flush Flush write buffers, if applicable.
   isatty Return whether this is an ‘interactive’ stream.
   read
   readable() Return whether object was opened for reading.
   readall Read until EOF, using multiple read() call.
   readinto(buff) Read bytes into a pre-allocated, writable bytes-like object.
   readinto1(buff) Read bytes into a pre-allocated, writable bytes-like object.
   readline Read and return a line from the stream.
   readlines Return a list of lines from the stream.
   seek(offset[, whence]) Change the stream position to the given offset.
   seekable() Return whether object supports random access.
   tell Return current stream position.
   truncate Truncate file to size bytes.
   writable Return whether object was opened for writing.
   write
   writelines

   Attributes

   SEEK_LINEAR
   SEEK_OFFSET
   SEEK_QUADRATIC

   Continued on next page
Table 79 – continued from previous page

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>closed</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>header</td>
<td>The header associated with the BQM.</td>
</tr>
<tr>
<td>header_end</td>
<td>The location (in bytes) that the header ends.</td>
</tr>
<tr>
<td>linear_end</td>
<td>The location (in bytes) that the linear data end.</td>
</tr>
<tr>
<td>linear_start</td>
<td>The location (in bytes) that the linear data starts.</td>
</tr>
<tr>
<td>neighborhood_starts</td>
<td>The indices of the neighborhood starts.</td>
</tr>
<tr>
<td>offset_end</td>
<td>The location (in bytes) that the offset starts.</td>
</tr>
<tr>
<td>offset_start</td>
<td>The location (in bytes) that the offset starts.</td>
</tr>
<tr>
<td>quadratic_end</td>
<td>The location (in bytes) that the quadratic data end.</td>
</tr>
<tr>
<td>quadratic_start</td>
<td>The location (in bytes) that the quadratic data starts.</td>
</tr>
<tr>
<td>variables_end</td>
<td></td>
</tr>
<tr>
<td>variables_start</td>
<td>The variables section as bytes.</td>
</tr>
<tr>
<td>variables_section</td>
<td></td>
</tr>
</tbody>
</table>

**dimod.serialization.fileview.load**

**load** *(fp, cls=None)*

Load a binary quadratic model from a file.

**Parameters**

- **fp** *(bytes-like/file-like)* – If file-like, should be readable, seekable file-like object. If bytes-like it will be wrapped with `io.BytesIO`.
- **cls** *(class, optional)* – The class of binary quadratic model. If not provided, the bqm will be of the same class that was saved. Note: currently only works for AdjArrayBQM.

**Returns** The loaded bqm.

**JSON**

JSON-encoding of dimod objects.

**Examples**

```python
>>> import json
>>> from dimod.serialization.json import DimodEncoder, DimodDecoder
...
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {('a', 'b'): -1})
>>> s = json.dumps(bqm, cls=DimodEncoder)
>>> new = json.loads(s, cls=DimodDecoder)
>>> bqm == new
True

>>> import json
>>> from dimod.serialization.json import DimodEncoder, DimodDecoder
...
>>> sampleset = dimod.SampleSet.from_samples({'a': -1, 'b': 1}, dimod.SPIN, energy=5)
>>> s = json.dumps(sampleset, cls=DimodEncoder)
>>> new = json.loads(s, cls=DimodDecoder)
>>> sampleset == new
True
```
```python
>>> import json
>>> from dimod.serialization.json import DimodEncoder, DimodDecoder
...
>>> # now inside a list
>>> s = json.dumps([sampleset, bqm], cls=DimodEncoder)
>>> new = json.loads(s, cls=DimodDecoder)
>>> new == [sampleset, bqm]
True

DimodEncoder(*[, skipkeys, ensure_ascii,...]) Subclass the JSONEncoder for dimod objects.
DimodDecoder(*args, **kwargs) Subclass the JSONDecoder for dimod objects.
dimod_object_hook(obj) JSON-decoding for dimod objects.

dimod.serialization.json.DimodEncoder

class DimodEncoder(*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True,
                    sort_keys=False, indent=None, separators=None, default=None)
    Subclass the JSONEncoder for dimod objects.
    __init__(*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True,
             sort_keys=False, indent=None, separators=None, default=None)
    Constructor for JSONEncoder, with sensible defaults.
    If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If
    skipkeys is True, such items are simply skipped.
    If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters
    escaped. If ensure_ascii is false, the output can contain non-ASCII characters.
    If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular refer-
    ences during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise,
    no such check takes place.
    If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON
    specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it
    will be a ValueError to encode such floats.
    If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests
to ensure that JSON serializations can be compared on a day-to-day basis.
    If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed
    with that indent level. An indent level of 0 will only insert newlines. None is the most compact represen-
    tation.
    If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if
    indent is None and ('', ' ') otherwise. To get the most compact JSON representation, you should specify
    (',', ':') to eliminate whitespace.
    If specified, default is a function that gets called for objects that can’t otherwise be serialized. It should
    return a JSON encodable version of the object or raise a TypeError.

Methods

6.1. dimod

299
Constructor for JSONEncoder, with sensible defaults.

Implement this method in a subclass such that it returns a serializable object for \( o \), or calls the base implementation (to raise a `TypeError`).

Return a JSON string representation of a Python data structure.

Encode the given object and yield each string representation as available.

**Attributes**

- `item_separator`
- `key_separator`

**dimod.serialization.json.DimodDecoder**

**class** `DimodDecoder`(*args, **kwargs)

Subclass the JSONDecoder for dimod objects.

Uses `dimod_object_hook()`.

**Methods**

**__init__**(*args, **kwargs)

- `object_hook`, if specified, will be called with the result of every JSON object decoded and its return value will be used in place of the given `dict`. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).
- `object_pairs_hook`, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of `object_pairs_hook` will be used instead of the `dict`. This feature can be used to implement custom decoders. If `object_hook` is also defined, the `object_pairs_hook` takes priority.
- `parse_float`, if specified, will be called with the string of every JSON float to be decoded. By default this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. `decimal.Decimal`).
- `parse_int`, if specified, will be called with the string of every JSON int to be decoded. By default this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. `float`).
- `parse_constant`, if specified, will be called with one of the following strings: `-Infinity`, `Infinity`, `NaN`. This can be used to raise an exception if invalid JSON numbers are encountered.

If `strict` is false (true is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0-31 range, including '\t' (tab), '\n', '\r' and '\0'.
Table 83 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decode(s[, _w])</code></td>
<td>Return the Python representation of s (a str instance containing a JSON document).</td>
</tr>
<tr>
<td><code>raw_decode(s[, idx])</code></td>
<td>Decode a JSON document from s (a str beginning with a JSON document) and return a 2-tuple of the Python representation and the index in s where the document ended.</td>
</tr>
</tbody>
</table>

### dimod.serialization.json.dimod_object_hook

**dimod_object_hook** *(obj)*

JSON-decoding for dimod objects.

**See also:**

`json.JSONDecoder` for using custom decoders.

### Testing

The testing subpackage contains functions for verifying and testing dimod objects. Testing objects/functions can be imported from the `dimod.testing` namespace. For example:

```python
>>> from dimod.testing import assert_sampler_api
```

### API Asserts

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assert_composite_api</code></td>
<td>Assert that an instantiated composed sampler exposes correct composite properties and methods.</td>
</tr>
<tr>
<td><code>assert_sampler_api</code></td>
<td>Assert that an instantiated sampler exposes correct properties and methods.</td>
</tr>
<tr>
<td><code>assert_structured_api</code></td>
<td>Assert that an instantiated structured sampler exposes correct composite properties and methods.</td>
</tr>
</tbody>
</table>

### dimod.testing.asserts.assert_composite_api

**assert_composite_api** *(composed_sampler)*

Assert that an instantiated composed sampler exposes correct composite properties and methods.

**Parameters**

- `composed_sampler` *(Composite)* – User-made dimod composed sampler.

**Raises**

- `AssertionError` – If the given sampler does not match the composite API.

**See also:**

- `Composite` for the abstract base class that defines the composite API.
- `assert_sampler_api` to assert that the composed sampler matches the sampler API.

### dimod.testing.asserts.assert_sampler_api

**assert_sampler_api** *(sampler)*

Assert that an instantiated sampler exposes correct properties and methods.
Parameters **sampler** *(Sampler)* – User-made dimod sampler.

Raises *AssertionError* – If the given sampler does not match the sampler API.

See also:

*Sampler* for the abstract base class that defines the sampler API.

dimod.testing.asserts.assert_structured_api

**assert_structured_api** *(sampler)*

Assert that an instantiated structured sampler exposes correct composite properties and methods.

Parameters **sampler** *(Structured)* – User-made dimod structured sampler.

Raises *AssertionError* – If the given sampler does not match the structured API.

See also:

*Structured* for the abstract base class that defines the structured API.

assert_sampler_api to assert that the structured sampler matches the sampler API.

Correctness Asserts

| assert_bqm_almost_equal(actual, desired[, ...]) | Test if two binary quadratic models have almost equal biases. |
| assert_response_energies(response, bqm[, ...]) | Assert that each sample in the given response has the correct energy. |
| assert_sampleset_energies(sampleset, bqm[, ...]) | Assert that each sample in the given sample set has the correct energy. |

dimod.testing.asserts.assert_bqm_almost_equal

**assert_bqm_almost_equal** *(actual, desired, places=7, ignore_zero_interactions=False)*

Test if two binary quadratic models have almost equal biases.

Parameters

- **actual** *(BinaryQuadraticModel)* – First binary quadratic model.
- **desired** *(BinaryQuadraticModel)* – Second binary quadratic model.
- **places** *(int, optional, default=7)* – Bias equality is computed as \( \text{round}(b_0 - b_1, \text{places}) = 0 \).
- **ignore_zero_interactions** *(bool, optional, default=False)* – If true, interactions with 0 bias are ignored.

dimod.testing.asserts.assert_response_energies

**assert_response_energies** *(response, bqm, precision=7)*

Assert that each sample in the given response has the correct energy.

Parameters

- **response** *(SampleSet)* – Response as returned by a dimod sampler.
• **bqm** *(BinaryQuadraticModel)* – Binary quadratic model (BQM) used to generate the samples.

• **precision** *(int, optional, default=7)* – Equality of energy is tested by calculating the difference between the response’s sample energy and that returned by BQM’s energy(), rounding to the closest multiple of 10 to the power of minus precision.

**Raises**

• **AssertionError** – If any of the samples in the response do not match their associated energy.

**See also:**

assert_sampleset_energies()

(dimod.testing.asserts.assert_sampleset_energies)

**assert_sampleset_energies** *(sampleset, bqm, precision=7)*

Assert that each sample in the given sample set has the correct energy.

**Parameters**

• **sampleset** *(SampleSet)* – Sample set as returned by a dimod sampler.

• **bqm** *(BinaryQuadraticModel/BinaryPolynomial)* – The binary quadratic model (BQM) or binary polynomial used to generate the samples.

• **precision** *(int, optional, default=7)* – Equality of energy is tested by calculating the difference between the response’s sample energy and that returned by BQM’s energy(), rounding to the closest multiple of 10 to the power of minus precision.

**Raises**

• **AssertionError** – If any of the samples in the sample set do not match their associated energy.

**Examples**

```python
>>> import dimod.testing
... >>> sampler = dimod.ExactSolver()
... >>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {(0, 1): -1})
... >>> sampleset = sampler.sample(bqm)
... >>> dimod.testing.assert_response_energies(sampleset, bqm)
```

**Test Case Loader**

*load_sampler_bqm_tests*(sampler[,...]) Populate the decorated TestCase with sampler tests using small BQMs.*
**dimod.testing.sampler.load_sampler_bqm_tests**

`load_sampler_bqm_tests(sampler, max_num_variables=None, suppress_overload_warning=False)`

Populate the decorated TestCase with sampler tests using small BQMs.

**Parameters**

- `sampler (obj/callable)` – A `Sampler`. If given as an object, the same sampler will be used for all tests, if given as a callable a new sampler will be instantiated for each test.
- `max_num_variables (int, optional)` – The maximum BQM size to be tested. Defaults to unlimited.
- `suppress_overload_warning (bool, optional, default=False)` – If True and the decorated class has a method matching one of the ones generated by `load_sampler_bqm_tests`, a warning will be emitted.

**Example**

For example, if one wanted to test the `ExactSolver`,

```python
import unittest
import dimod.testing

@dimod.testing.load_sampler_bqm_tests(dimod.ExactSolver)
class TestExactSolver(unittest.TestCase):
    pass

unittest.main()
```

**Vartype Conversion**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ising_to_qubo(h, J, offset))</code></td>
<td>Convert an Ising problem to a QUBO problem.</td>
</tr>
<tr>
<td><code>qubo_to_ising(Q, offset)</code></td>
<td>Convert a QUBO problem to an Ising problem.</td>
</tr>
</tbody>
</table>

**dimod.utilities.ising_to_qubo**

`ising_to_qubo(h, J, offset=0.0)`

Convert an Ising problem to a QUBO problem.

Map an Ising model defined on spins (variables with \{-1, +1\} values) to quadratic unconstrained binary optimization (QUBO) formulation $x'Qx$ defined over binary variables (0 or 1 values), where the linear term is contained along the diagonal of Q. Return matrix Q that defines the model as well as the offset in energy between the two problem formulations:

$$s'Js + h's = offset + x'Qx$$

See `qubo_to_ising()` for the inverse function.

**Parameters**

- `h (dict[variable, bias])` – Linear biases as a dict of the form `{v: bias, ...}`, where keys are variables of the model and values are biases.
• \( \mathbf{J}(\text{dict}[(\text{variable}, \text{variable}), \text{bias}]) \) – Quadratic biases as a dict of the form \( \{(u, v): \text{bias}, \ldots\} \), where keys are 2-tuples of variables of the model and values are quadratic biases associated with the pair of variables (the interaction).

• \( \text{offset} (\text{numeric, optional, default=0}) \) – Constant offset to be applied to the energy. Default 0.

**Returns**

A 2-tuple containing:

- dict: QUBO coefficients.
- float: New energy offset.

**Return type** (dict, float)

**Examples**

This example converts an Ising problem of two variables that have positive biases of value 1 and are positively coupled with an interaction of value 1 to a QUBO problem and prints the resulting energy offset.

```python
>>> h = {1: 1, 2: 1}
>>> J = {(1, 2): 1}
>>> dimod.ising_to_qubo(h, J, 0.5)[1]
-0.5
```

**dimod.utilities.qubo_to_ising**

**qubo_to_ising** \((\mathbf{Q}, \text{offset}=0.0)\)

Convert a QUBO problem to an Ising problem.

Map a quadratic unconstrained binary optimization (QUBO) problem \( x'Qx \) defined over binary variables (0 or 1 values), where the linear term is contained along the diagonal of \( Q \), to an Ising model defined on spins (variables with \([-1, +1]\) values). Return \( h \) and \( J \) that define the Ising model as well as the offset in energy between the two problem formulations:

\[
x'Qx = \text{offset} + s'Js + h's
\]

See **ising_to_qubo()** for the inverse function.

**Parameters**

- \( \mathbf{Q} (\text{dict}[(\text{variable}, \text{variable}), \text{coefficient}]) \) – QUBO coefficients in a dict of form \( \{(u, v): \text{coefficient}, \ldots\} \), where keys are 2-tuples of variables of the model and values are biases associated with the pair of variables. Tuples \((u, v)\) represent interactions and \((v, v)\) linear biases.

- \( \text{offset} (\text{numeric, optional, default=0}) \) – Constant offset to be applied to the energy. Default 0.

**Returns**

A 3-tuple containing:

- float: New energy offset.
Return type (dict, dict, float)

Examples

This example converts a QUBO problem of two variables that have positive biases of value 1 and are positively coupled with an interaction of value 1 to an Ising problem, and shows the new energy offset.

```python
>>> Q = {(1, 1): 1, (2, 2): 1, (1, 2): 1}
>>> dimod.qubo_to_ising(Q, 0.5)[2]
1.75
```

Vartype

Enumeration of valid variable types for binary quadratic models.

Examples

Vartype is an Enum. Each vartype has a value and a name.

```python
>>> vartype = dimod.SPIN
>>> vartype.name
'SPIN'
>>> vartype.value == {-1, +1}
True
```

```python
>>> vartype = dimod.BINARY
>>> vartype.name
'BINARY'
>>> vartype.value == {0, 1}
True
```

The as_vartype() function allows the user to provide several convenient forms.

```python
>>> from dimod import as_vartype
```

```python
>>> as_vartype(dimod.SPIN) is dimod.SPIN
True
>>> as_vartype('SPIN') is dimod.SPIN
True
>>> as_vartype({-1, 1}) is dimod.SPIN
True
```

```python
>>> as_vartype(dimod.BINARY) is dimod.BINARY
True
>>> as_vartype('BINARY') is dimod.BINARY
True
>>> as_vartype({0, 1}) is dimod.BINARY
True
```

class Vartype

An Enum over the types of variables for the binary quadratic model.

**SPIN**

Vartype for spin-valued models; variables of the model are either -1 or 1.
Type `Vartype`

**BINARY**
Vartype for binary models; variables of the model are either 0 or 1.

Type `Vartype`  

`as_vartype(vartype)`
Cast various inputs to a valid vartype object.

**Parameters** `vartype` (*Vartype/str/set*) – Variable type. Accepted input values:
- `Vartype.SPIN`, `'SPIN'`, `-1, 1`
- `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

**Returns** Either `Vartype.SPIN` or `Vartype.BINARY`.

**Return type** `Vartype`

See also:
`vartype_argument()`

**Exceptions**

- **exception InvalidComposition**
  Raised for compositions of samplers that are invalid

- **exception MappingError**
  Raised when mapping causes conflicting values in samples

- **exception InvalidSampler**
  Raised when trying to use the specified sampler as a sampler

- **exception BinaryQuadraticModelValueError**
  Raised when a sampler cannot handle a specified binary quadratic model

- **exception BinaryQuadraticModelSizeError**
  Raised when the binary quadratic model has too many variables

- **exception BinaryQuadraticModelStructureError**
  Raised when the binary quadratic model does not fit the sampler

- **exception WriteableError**
  Raised when trying to modify an immutable object.

### 6.2 dwavebinarycsp

Library to construct a binary quadratic model from a constraint satisfaction problem with small constraints over binary variables.

Below is an example usage:

```python
import dwavebinarycsp
import dimod

csp = dwavebinarycsp.factories.random_2in4sat(8, 4)  # 8 variables, 4 clauses
bqm = dwavebinarycsp.stitch(csp)
```

(continues on next page)
6.2.1 Introduction

dwavebinarycsp is a library to construct a binary quadratic model from a constraint satisfaction problem (CSP) with small constraints over binary variables (represented as either binary values \{0, 1\} or spin values \{-1, 1\}).

For an introduction to CSPs, see Constraints.

Example: Solving a Map-Coloring CSP

The map-coloring CSP, for example, is to assign a color to each region of a map such that any two regions sharing a border have different colors.

Fig. 3: Coloring a map of Canada with four colors.

The constraints for the map-coloring problem can be expressed as follows:

- Each region is assigned one color only, of \(C\) possible colors.
- The color assigned to one region cannot be assigned to adjacent regions.

Solving such problems as the map-coloring CSP on a sampler such as the D-Wave system necessitates that the mathematical formulation use binary variables because the solution is implemented physically with qubits, and so must translate to spins \(s_i \in \{-1, +1\}\) or equivalent binary values \(x_i \in \{0, 1\}\). This means that in formulating the problem by stating it mathematically, you might use unary encoding to represent the \(C\) colors: each region is represented by \(C\) variables, one for each possible color, which is set to value 1 if selected, while the remaining \(C - 1\) variables are 0.

This example finds a solution to the map-coloring problem for a map of Canada using four colors. Canada’s 13 provinces are denoted by postal codes:

```python
resp = dimod.ExactSolver().sample(bqm)
for sample, energy in resp.data(['sample', 'energy']):
    print(sample, csp.check(sample), energy)
```
Table 88: Canadian Provinces’ Postal Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Province</th>
<th>Code</th>
<th>Province</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>Alberta</td>
<td>BC</td>
<td>British Columbia</td>
</tr>
<tr>
<td>MB</td>
<td>Manitoba</td>
<td>NB</td>
<td>New Brunswick</td>
</tr>
<tr>
<td>NL</td>
<td>Newfoundland and Labrador</td>
<td>NS</td>
<td>Nova Scotia</td>
</tr>
<tr>
<td>NT</td>
<td>Northwest Territories</td>
<td>NU</td>
<td>Nunavut</td>
</tr>
<tr>
<td>ON</td>
<td>Ontario</td>
<td>PE</td>
<td>Prince Edward Island</td>
</tr>
<tr>
<td>QC</td>
<td>Quebec</td>
<td>SK</td>
<td>Saskatchewan</td>
</tr>
<tr>
<td>YT</td>
<td>Yukon</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The workflow for solution is as follows:

1. Formulate the problem as a graph, with provinces represented as nodes and shared borders as edges, using 4 binary variables (one per color) for each province.
2. Create a binary constraint satisfaction problem and add all the needed constraints.
3. Convert to a binary quadratic model.
4. Sample.
5. Plot a valid solution, if such was found.

The following sample code creates a graph of the map with provinces as nodes and shared borders between provinces as edges (e.g., “(‘AB’, ‘BC’)” is an edge representing the shared border between British Columbia and Alberta). It creates a binary constraint satisfaction problem based on two types of constraints:

- `csp.add_constraint(one_color_configurations, variables)` represents the constraint that each node (province) select a single color, as represented by valid configurations `one_color_configurations = {(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)}`
- `csp.add_constraint(not_both_1, variables)` represents the constraint that two nodes (provinces) with a shared edge (border) not both select the same color.

```
import dwavebinarycsp
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import EmbeddingComposite
import networkx as nx
import matplotlib.pyplot as plt

# Represent the map as the nodes and edges of a graph
provinces = ['AB', 'BC', 'MB', 'NB', 'NL', 'NS', 'NT', 'NU', 'ON', 'PE', 'QC', 'SK', 'YT']
neighbors = [('AB', 'BC'), ('AB', 'NT'), ('AB', 'SK'), ('BC', 'NT'), ('BC', 'YT'), ('MB', 'NU'), ('MB', 'ON'), ('MB', 'SK'), ('NB', 'NS'), ('NB', 'QC'), ('NL', 'QC'), ('NT', 'NU'), ('NT', 'SK'), ('NT', 'YT'), ('ON', 'QC')]

# Function for the constraint that two nodes with a shared edge not both select one color
def not_both_1(v, u):
    return not (v and u)

# Function that plots a returned sample
def plot_map(sample):
    G = nx.Graph()
```

(continues on next page)
G.add_nodes_from(provinces)
G.add_edges_from(neighbors)

# Translate from binary to integer color representation
color_map = {}
for province in provinces:
    for i in range(colors):
        if sample[province+str(i)]:
            color_map[province] = i

# Plot the sample with color-coded nodes
node_colors = [color_map.get(node) for node in G.nodes()]
nx.draw_circular(G, with_labels=True, node_color=node_colors, node_size=3000,
                 cmap=plt.cm.rainbow)
plt.show()

# Valid configurations for the constraint that each node select a single color
one_color_configurations = {(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)}
colors = len(one_color_configurations)

csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)

# Add constraint that each node (province) select a single color
for province in provinces:
    variables = [province+str(i) for i in range(colors)]
    csp.add_constraint(one_color_configurations, variables)

# Add constraint that each pair of nodes with a shared edge not both select one color
for neighbor in neighbors:
    v, u = neighbor
    for i in range(colors):
        variables = [v+str(i), u+str(i)]
        csp.add_constraint(not_both_1, variables)

# Convert the binary constraint satisfaction problem to a binary quadratic model
bqm = dwavebinarycsp.stitch(csp)

# Set up a solver using the local system’s default D-Wave Cloud Client configuration
sampler = EmbeddingComposite(DWaveSampler())
response = sampler.sample(bqm, num_reads=50)

# Plot the lowest-energy sample if it meets the constraints
sample = next(response.samples())
if not csp.check(sample):
    print("Failed to color map")
else:
    plot_map(sample)

The plot shows a solution returned by the D-Wave solver. No provinces sharing a border have the same color.

### 6.2.2 Reference Documentation

**Release** 2.5.0

**Date** Aug 12, 2020
Fig. 4: Solution for a map of Canada with four colors. The graph comprises 13 nodes representing provinces connected by edges representing shared borders. No two nodes connected by an edge share a color.

**Defining Constraint Satisfaction Problems**

Constraint satisfaction problems require that all a problem’s variables be assigned values, out of a finite domain, that result in the satisfying of all constraints. The `ConstraintSatisfactionProblem` class aggregates all constraints and variables defined for a problem and provides functionality to assist in problem solution, such as verifying whether a candidate solution satisfies the constraints.

**Class**

```python
class ConstraintSatisfactionProblem(vartype)
    A constraint satisfaction problem.

    Parameters vartype (Vartype/str/set) – Variable type for the binary quadratic model. Supported values are:
        * SPIN, 'SPIN', {-1, 1}
        * BINARY, 'BINARY', {0, 1}

    constraints
        Constraints that together constitute the constraint satisfaction problem. Valid solutions satisfy all of the constraints.

    Type list[Constraint]

    variables
        Variables of the constraint satisfaction problem as a dict, where keys are the variables and values a list of
```

6.2. dwavebinarycsp 311
all of constraints associated with the variable.

**vartype**

Enumeration of valid variable types. Supported values are *SPIN* or *BINARY*. If *vartype* is *SPIN*, variables can be assigned -1 or 1; if *BINARY*, variables can be assigned 0 or 1.

**Example**

This example creates a binary-valued constraint satisfaction problem, adds two constraints, \(a = b\) and \(b \neq c\), and tests \(a, b, c = 1, 1, 0\).

```python
>>> import operator
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem('BINARY')
>>> csp.add_constraint(operator.eq, ['a', 'b'])
>>> csp.add_constraint(operator.ne, ['b', 'c'])
>>> csp.check({'a': 1, 'b': 1, 'c': 0})
True
```

**Methods**

**Adding variables and constraints**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ConstraintSatisfactionProblem.add_constraint(...)</code></td>
<td>Add a constraint.</td>
</tr>
<tr>
<td><code>ConstraintSatisfactionProblem.add_variable(v)</code></td>
<td>Add a variable.</td>
</tr>
</tbody>
</table>

*dwavebinarycsp.ConstraintSatisfactionProblem.add_constraint*

*ConstraintSatisfactionProblem.add_constraint*(constraint, variables=())

Add a constraint.

**Parameters**

- **constraint** (function/iterable/Constraint) – Constraint definition in one of the supported formats:
  1. Function, with input arguments matching the order and *vartype* type of the *variables* argument, that evaluates True when the constraint is satisfied.
  2. List explicitly specifying each allowed configuration as a tuple.
  3. *Constraint* object built either explicitly or by *dwavebinarycsp.factories*.
- **variables** (iterable) – Variables associated with the constraint. Not required when *constraint* is a *Constraint* object.
Examples

This example defines a function that evaluates True when the constraint is satisfied. The function’s input arguments match the order and type of the variables argument.

```python
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> def all_equal(a, b, c):
...     return (a == b) and (b == c)
... csp.add_constraint(all_equal, ['a', 'b', 'c'])
>>> csp.check({'a': 0, 'b': 0, 'c': 0})
True
>>> csp.check({'a': 0, 'b': 0, 'c': 1})
False
```

This example explicitly lists allowed configurations.

```python
>>> eq_configurations = {(-1, -1), (1, 1)}
>>> csp.add_constraint(eq_configurations, ['v0', 'v1'])
>>> csp.check({'v0': -1, 'v1': +1})
False
>>> csp.check({'v0': -1, 'v1': -1})
True
```

This example uses a Constraint object built by dwavebinarycsp.factories.

```python
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.and_gate(['a', 'b', 'c']))  # add an AND gate
>>> csp.add_constraint(gates.xor_gate(['a', 'c', 'd']))  # add an XOR gate
>>> csp.check({'a': 1, 'b': 0, 'c': 0, 'd': 1})
True
```

dwavebinarycsp.ConstraintSatisfactionProblem.add_variable

ConstraintSatisfactionProblem.add_variable(v)

Add a variable.

Parameters

v (variable) – Variable in the constraint satisfaction problem. May be of any type that can be a dict key.

Examples

This example adds two variables, one of which is already used in a constraint of the constraint satisfaction problem.

```python
>>> import operator
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.SPIN)
>>> csp.add_constraint(operator.eq, ['a', 'b'])
>>> csp.add_variable('a')  # does nothing, already added as part of the constraint
>>> csp.add_variable('c')
>>> csp.check({'a': -1, 'b': -1, 'c': 1})
True
```
Satisfiability

ConstraintSatisfactionProblem. check(solution)  
Check that a solution satisfies all of the constraints.

Parameters  
solution (container) – An assignment of values for the variables in the constraint satisfaction problem.

Returns  
True if the solution satisfies all of the constraints; False otherwise.

Return type  
bool

Examples

This example creates a binary-valued constraint satisfaction problem, adds two logic gates implementing Boolean constraints, \( c = a \land b \) and \( d = a \oplus c \), and verifies that the combined problem is satisfied for a given assignment.

```python
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.and_gate(['a', 'b', 'c']))  # add an AND gate
>>> csp.add_constraint(gates.xor_gate(['a', 'c', 'd']))  # add an XOR gate
>>> csp.check({'a': 1, 'b': 0, 'c': 0, 'd': 1})
True
```

Transformations

ConstraintSatisfactionProblem. fix_variable(v, ...)
Fix the value of a variable and remove it from the constraint satisfaction problem.

Parameters

- \( v \) (variable) – Variable to be fixed in the constraint satisfaction problem.
- value (int) – Value assigned to the variable. Values must match the vartype of the constraint satisfaction problem.
Examples

This example creates a spin-valued constraint satisfaction problem, adds two constraints, \( a = b \) and \( b \neq c \), and fixes variable \( b \) to +1.

```python
>>> import operator
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.SPIN)
>>> csp.add_constraint(operator.eq, ['a', 'b'])
>>> csp.add_constraint(operator.ne, ['b', 'c'])
>>> csp.check({'a': +1, 'b': +1, 'c': -1})
True
>>> csp.check({'a': -1, 'b': -1, 'c': +1})
True
>>> csp.fix_variable('b', +1)
>>> csp.check({'a': +1, 'b': +1, 'c': -1})  # 'b' is ignored
True
>>> csp.check({'a': -1, 'b': -1, 'c': +1})
False
>>> csp.check({'a': +1, 'c': -1})
True
>>> csp.check({'a': -1, 'c': +1})
False
```

Converting to a Binary Quadratic Model

Constraint satisfaction problems can be converted to binary quadratic models to be solved on samplers such as the D-Wave system.

Compilers

Compilers accept a constraint satisfaction problem and return a `dimod.BinaryQuadraticModel`.

<table>
<thead>
<tr>
<th><code>stitch(csp[, min_classical_gap, max_graph_size])</code></th>
<th>Build a binary quadratic model with minimal energy levels at solutions to the specified constraint satisfaction problem.</th>
</tr>
</thead>
</table>

`dwavebinarycsp.stitch`

```python
stitch(csp, min_classical_gap=2.0, max_graph_size=8)
Build a binary quadratic model with minimal energy levels at solutions to the specified constraint satisfaction problem.
```

Parameters

- `csp` (ConstraintSatisfactionProblem) – Constraint satisfaction problem.
- `min_classical_gap` (float, optional, default=2.0) – Minimum energy gap from ground. Each constraint violated by the solution increases the energy level of the binary quadratic model by at least this much relative to ground energy.
- `max_graph_size` (int, optional, default=8) – Maximum number of variables in the binary quadratic model that can be used to represent a single constraint.

Returns BinaryQuadraticModel
Notes

For a \texttt{min\_classical\_gap} > 2 or constraints with more than two variables, requires access to factories from the \texttt{penaltymodel} ecosystem to construct the binary quadratic model.

Examples

This example creates a binary-valued constraint satisfaction problem with two constraints, \(a = b\) and \(b \neq c\), and builds a binary quadratic model such that each constraint violation by a solution adds the default minimum energy gap.

```python
>>> import operator

>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)

>>> csp.add_constraint(operator.eq, ['a', 'b'])  # a == b

>>> csp.add_constraint(operator.ne, ['b', 'c'])  # b != c

>>> bqm = dwavebinarycsp.stitch(csp)
```

Variable assignments that satisfy the CSP above, violate one, then two constraints, produce energy increases of the default minimum classical gap:

```python
>>> bqm.energy(('a': 0, 'b': 0, 'c': 1))  # doctest: +SKIP
-2.0

>>> bqm.energy(('a': 0, 'b': 0, 'c': 0))  # doctest: +SKIP
0.0

>>> bqm.energy(('a': 1, 'b': 0, 'c': 0))  # doctest: +SKIP
2.0
```

This example creates a binary-valued constraint satisfaction problem with two constraints, \(a = b\) and \(b \neq c\), and builds a binary quadratic model with a minimum energy gap of 4. Note that in this case the conversion to binary quadratic model adds two ancillary variables that must be minimized over when solving.

```python
>>> import operator
>>> import itertools

>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)

>>> csp.add_constraint(operator.eq, ['a', 'b'])  # a == b

>>> csp.add_constraint(operator.ne, ['b', 'c'])  # b != c

>>> bqm = dwavebinarycsp.stitch(csp, min_classical_gap=4.0)

>>> list(bqm)  # doctest: +SKIP
['a', 'aux1', 'aux0', 'b', 'c']
```

Variable assignments that satisfy the CSP above, violate one, then two constraints, produce energy increases of the specified minimum classical gap:

```python
>>> min([bqm.energy(('a': 0, 'b': 0, 'c': 1, 'aux0': aux0, 'aux1': aux1)) for ... aux0, aux1 in list(itertools.product([0, 1], repeat=2))])  # doctest: +SKIP
-6.0

>>> min([bqm.energy(('a': 0, 'b': 0, 'c': 0, 'aux0': aux0, 'aux1': aux1)) for ... aux0, aux1 in list(itertools.product([0, 1], repeat=2))])  # doctest: +SKIP
-2.0

>>> min([bqm.energy(('a': 1, 'b': 0, 'c': 0, 'aux0': aux0, 'aux1': aux1)) for ... aux0, aux1 in list(itertools.product([0, 1], repeat=2))])  # doctest: +SKIP
2.0
```

This example finds for the previous example the minimum graph size.
```python
>>> import operator
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(operator.eq, ['a', 'b'])  # a == b
>>> csp.add_constraint(operator.ne, ['b', 'c'])  # b != c
>>> for n in range(8, 1, -1):
...     try:
...         bqm = dwavebinarycsp.stitch(csp, min_classical_gap=4.0, max_graph_size=n)
...     except dwavebinarycsp.exceptions.ImpossibleBQM:
...         print(n+1)
...     ...
3
```

Other CSP Formats

DIMACS

The DIMACS format is used to encode boolean satisfiability problems in conjunctive normal form.

CNF

`load_cnf(fp)` Load a constraint satisfaction problem from a .cnf file.

dwavebinarycsp.io.cnf.load_cnf

`load_cnf(fp)` Load a constraint satisfaction problem from a .cnf file.

**Parameters**

- **fp** (*file, optional*) – `.write()`-supporting file object DIMACS CNF formatted file.

**Returns**

- `ConstraintSatisfactionProblem` a binary-valued SAT problem.

**Examples**

```python
>>> import dwavebinarycsp as dbcsp
... >>> with open('test.cnf', 'r') as fp:  # doctest: +SKIP
...     csp = dbcsp.cnf.load_cnf(fp)
```

Reducing Constraints

Constraints can sometimes be reduced into several smaller constraints.

Functions
**irreducible_components** *(constraint) – Determine the sets of variables that are irreducible.*

**dwavebinarycsp.irreducible_components**

**irreducible_components** *(constraint) – Determine the sets of variables that are irreducible.*

Let \( V(C) \) denote the variables of constraint \( C \). For a configuration \( x \), let \( x|A \) denote the restriction of the configuration to the variables of \( A \). Constraint \( C \) is reducible if there is a partition of \( V(C) \) into nonempty subsets \( A \) and \( B \), and two constraints \( C_A \) and \( C_B \), with \( V(C_A) = A \) and \( V(C_B) = B \), such that a configuration \( x \) is feasible in \( C \) if and only if \( x|A \) is feasible in \( C_A \) and \( x|B \) is feasible in \( C_B \). A constraint is irreducible if it is not reducible.

**Parameters**

- **constraint** *(Constraint) – Constraint to attempt to reduce.*

**Returns**

- **List of tuples in which each tuple is a set of variables that is irreducible.**

**Return type**

- **list[tuple]**

**Examples**

This example reduces a constraint, created by specifying its valid configurations, to two constraints. The original constraint, that valid configurations for \( a,b,c \) are \( 0,0,1 \) and \( 1,1,1 \), can be represented by two reduced constraints, for example, \( (c=1) \land (a=b) \). For comparison, an attempt to reduce a constraint representing an AND gate fails to find a valid reduction.

```python
>>> const = dwavebinarycsp.Constraint.from_configurations([(0, 0, 1), (1, 1, 1)],
...                                   ['a', 'b', 'c'],
...                                   dwavebinarycsp.BINARY)
>>> dwavebinarycsp.irreducible_components(const)
[('c',), ('a', 'b')]
```

**Defining Constraints**

Solutions to a constraint satisfaction problem must satisfy certain conditions, the constraints of the problem, such as equality and inequality constraints. The **Constraint** class defines constraints and provides functionality to assist in constraint definition, such as verifying whether a candidate solution satisfies a constraint.

**Class**

```python
class Constraint (func, configurations, variables, vartype, name=None)
A constraint.

variables
Variables associated with the constraint.

Type tuple
```
func
   Function that returns True for configurations of variables that satisfy the constraint. Inputs to the function are ordered by variables.

   Type function

configurations
   Valid configurations of the variables. Each configuration is a tuple of variable assignments ordered by variables.

   Type frozenset[tuple]

type
   Variable type for the constraint. Accepted input values:

   • SPIN, 'SPIN', {-1, 1}
   • BINARY, 'BINARY', {0, 1}

   Type dimod.Vartype

name
   Name for the constraint. If not provided on construction, defaults to ‘Constraint’.

   Type str

Examples

This example defines a constraint, named “plus1”, based on a function that is True for \((y_1, y_0) = (x_1, x_0) + 1\) on binary variables, and demonstrates some of the constraint’s functionality.

```python
>>> def plus_one(y1, y0, x1, x0):
...     # y=x++ for two bit binary numbers
...     return (y1, y0, x1, x0) in [(0, 1, 0, 0), (1, 0, 0, 1), (1, 1, 1, 0)]
... 
>>> const = dwavebinarycsp.Constraint.from_func(
...     plus_one,
...     ['out1', 'out0', 'in1', 'in0'],
...     dwavebinarycsp.BINARY,
...     name='plus1')
>>> print(const.name)  # Check constraint defined as intended
plus1
>>> len(const)
4
>>> in0, in1, out0, out1 = 0, 0, 1, 0
>>> const.func(out1, out0, in1, in0)  # Order matches variables
True
```

This example defines a constraint based on specified valid configurations that represents an AND gate for spin variables, and demonstrates some of the constraint’s functionality.

```python
>>> const = dwavebinarycsp.Constraint.from_configurations(
...     [(-1, -1, -1), (-1, -1, 1), (-1, 1, -1), (1, 1, 1)],
...     ['y', 'x1', 'x2'],
...     dwavebinarycsp.SPIN)
>>> print(const.name)  # Check constraint defined as intended
Constraint
>>> isinstance(const, dwavebinarycsp.core.constraint.Constraint)
True
```
>>> (-1, 1, -1) in const.configurations  # Order matches variables: y,x1,x2
True

### Methods

#### Construction

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Constraint.from_configurations(...[, name])</code></td>
<td>Construct a constraint from valid configurations.</td>
</tr>
<tr>
<td><code>Constraint.from_func(func, variables, var_type)</code></td>
<td>Construct a constraint from a validation function.</td>
</tr>
</tbody>
</table>

**`dwavebinarycsp.Constraint.from_configurations`**

**classmethod** `Constraint.from_configurations(configurations, variables, var_type, name=None)`

Construct a constraint from valid configurations.

**Parameters**

- **configurations** *(iterable[tuple])* – Valid configurations of the variables. Each configuration is a tuple of variable assignments ordered by `variables`.
- **variables** *(iterable)* – Iterable of variable labels.
- **var_type** *(Vartype/str/set)* – Variable type for the constraint. Accepted input values:
  - `SPIN`, `'SPIN'`, `{−1, 1}
  - `BINARY`, `'BINARY'`, `{0, 1}
- **name** *(string, optional, default='Constraint')* – Name for the constraint.

**Examples**

This example creates a constraint that variables `a` and `b` are not equal.

```python
>>> const = dwavebinarycsp.Constraint.from_configurations([[0, 1], (1, 0)],
... ['a', 'b'], dwavebinarycsp.BINARY)
>>> print(const.name)
Constraint
>>> (0, 0) in const.configurations  # Order matches variables: a,b
False
```

This example creates a constraint based on specified valid configurations that represents an OR gate for spin variables.

```python
>>> const = dwavebinarycsp.Constraint.from_configurations(
... [(-1, -1, -1), (1, -1, 1), (1, 1, -1), (1, 1, 1)],
... ['y', 'x1', 'x2'],
... dwavebinarycsp.SPIN, name='or_spin')
>>> print(const.name)
or_spin
```
>>> (1, 1, -1) in const.configurations  # Order matches variables: y,x1,x2
True

dwavebinarycsp.Constraint.from_func

classmethod Constraint.from_func(func, variables, vartype, name=None)
Construct a constraint from a validation function.

Parameters
• func(function) – Function that evaluates True when the variables satisfy the constraint.
• variables(iterable) – Iterable of variable labels.
• vartype(Vartype/str/set) – Variable type for the constraint. Accepted input values:
  – SPIN, 'SPIN', {-1, 1}
  – BINARY, 'BINARY', {0, 1}
• name(string, optional, default='Constraint') – Name for the constraint.

Examples

This example creates a constraint that binary variables a and b are not equal.

```python
>>> import operator
>>> const = dwavebinarycsp.Constraint.from_func(operator.ne, ['a', 'b'], 'BINARY')
>>> print(const.name)
Constraint
>>> (0, 1) in const.configurations
True
```

This example creates a constraint that out = NOT(x) for spin variables.

```python
>>> def not_(y, x):  # y=NOT(x) for spin variables
...     return (y == -x)
... >>> const = dwavebinarycsp.Constraint.from_func(  
...     not_,  
...     ['out', 'in'],  
...     {1, -1},  
...     name='not_spin')
>>> print(const.name)
not_spin
>>> (1, -1) in const.configurations
True
```

Satisfiability

Constraint.check(solution) Check that a solution satisfies the constraint.
**docs Documentation, Release 2.5.0**

**dwavebinarycsp.Constraint.check**

Constraint.check(solution)

Check that a solution satisfies the constraint.

**Parameters**

- **solution** (container) – An assignment for the variables in the constraint.

**Returns**

- True if the solution satisfies the constraint; otherwise False.

**Return type**

- bool

**Examples**

This example creates a constraint that \( a \neq b \) on binary variables and tests it for two candidate solutions, with additional unconstrained variable \( c \).

```python
>>> const = dwavebinarycsp.Constraint.from_configurations([(0, 1), (1, 0)], ['a', 'b'], dwavebinarycsp.BINARY)
>>> solution = {'a': 1, 'b': 1, 'c': 0}
>>> const.check(solution)
False
>>> solution = {'a': 1, 'b': 0, 'c': 0}
>>> const.check(solution)
True
```

**Transformations**

- **Constraint.fix_variable**

  Fix the value of a variable and remove it from the constraint.

- **Constraint.flip_variable**

  Flip a variable in the constraint.

**dwavebinarycsp.Constraint.fix_variable**

Constraint.fix_variable(v, value)

Fix the value of a variable and remove it from the constraint.

**Parameters**

- **v** (variable) – Variable in the constraint to be set to a constant value.

- **val** (int) – Value assigned to the variable. Values must match the Vartype of the constraint.

**Examples**

This example creates a constraint that \( a \neq b \) on binary variables, fixes variable \( a \) to 0, and tests two candidate solutions.

```python
>>> const = dwavebinarycsp.Constraint.from_func(operator.ne, ['a', 'b'], dwavebinarycsp.BINARY)
>>> const.fix_variable('a', 0)
>>> const.check({'b': 1})
True
```
```python
>>> const.check({'b': 0})
False
```

dwavebinarycsp.Constraint.flip_variable

```python
Constraint.flip_variable(v)
Flip a variable in the constraint.

Parameters
v (variable) – Variable in the constraint to take the complementary value of its construction value.
```

Examples

This example creates a constraint that \(a = b\) on binary variables and flips variable \(a\).
```python
>>> const = dwavebinarycsp.Constraint.from_func(operator.eq, ...
... ['a', 'b'], dwavebinarycsp.BINARY)
>>> const.check({'a': 0, 'b': 0})
True
>>> const.flip_variable('a')
>>> const.check({'a': 1, 'b': 0})
True
>>> const.check({'a': 0, 'b': 0})
False
```

Copies and projections

<table>
<thead>
<tr>
<th>Constraint.copy()</th>
<th>Create a copy.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint.projection(variables)</td>
<td>Create a new constraint that is the projection onto a sub-set of the variables.</td>
</tr>
</tbody>
</table>

dwavebinarycsp.Constraint.copy

```python
Constraint.copy() Create a copy.
```

Examples

This example copies constraint \(a \neq b\) and tests a solution on the copied constraint.
```python
>>> import operator
>>> const = dwavebinarycsp.Constraint.from_func(operator.ne, ...
... ['a', 'b'], 'BINARY')
>>> const2 = const.copy()
>>> const2 is const
False
>>> const2.check({'a': 1, 'b': 1})
False
```
Constraint.projection \( (\text{variables}) \)

Create a new constraint that is the projection onto a subset of the variables.

**Parameters**

- **variables** *(iterable)* – Subset of the constraint’s variables.

**Returns**

A new constraint over a subset of the variables.

**Return type**

`Constraint`

**Examples**

```python
>>> const = dwavebinarycsp.Constraint.from_configurations([[0, 0], [0, 1]],
                                             ['a', 'b'],
                                             dwavebinarycsp.BINARY)
>>> proj = const.projection(['a'])
>>> proj.variables
('a',)
>>> proj.configurations
frozenset({(0,)})
```

**Factories**

`dwavebinarycsp` currently provides factories for constraints representing Boolean gates and satisfiability problems and CSPs for circuits and satisfiability problems.

**Constraints**

**Gates**

- `gates.and_gate` *(variables[, vartype, name])* – AND gate.
- `gates.or_gate` *(variables[, vartype, name])* – OR gate.
- `gates.xor_gate` *(variables[, vartype, name])* – XOR gate.
- `gates.halfadder_gate` *(variables[, vartype, name])* – Half adder.
- `gates.fulladder_gate` *(variables[, vartype, name])* – Full adder.

**dwavebinarycsp.factories.constraint.gates.and_gate**

`and_gate` *(variables, vartype=<Vartype.BINARY: frozenset({0, 1})>, name='AND')*

AND gate.

**Parameters**

- **variables** *(list)* – Variable labels for the and gate as \([\text{in}1, \text{in}2, \text{out}]\), where \(\text{in}1, \text{in}2\) are inputs and \(\text{out}\) the gate’s output.

- **vartype** *(Vartype, optional, default='BINARY')* – Variable type. Accepted input values:
  - Vartype.SPIN, ‘SPIN’, \([-1, 1]\)
- Vartype.BINARY, 'BINARY', {0, 1}
  - name (str, optional, default='AND') – Name for the constraint.

Returns Constraint that is satisfied when its variables are assigned values that match the valid states of an AND gate.

Return type Constraint

Examples

```python
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.and_gate(['a', 'b', 'c'], name='AND1'))
>>> csp.check({'a': 1, 'b': 0, 'c': 0})
True
```

dwavebinarycsp.factories.constraint.gates.or_gate

or_gate (variables, vartype=<Vartype.BINARY: frozenset({0, 1})>, name='OR')

OR gate.

Parameters
- variables (list) – Variable labels for the and gate as [in1, in2, out], where in1, in2 are inputs and out the gate’s output.
- vartype (Vartype, optional, default='BINARY') – Variable type. Accepted input values:
  - Vartype.SPIN, ‘SPIN’, {-1, 1}
  - Vartype.BINARY, ‘BINARY’, {0, 1}
  - name (str, optional, default='OR') – Name for the constraint.

Returns Constraint that is satisfied when its variables are assigned values that match the valid states of an OR gate.

Return type Constraint

Examples

```python
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.SPIN)
>>> csp.add_constraint(gates.or_gate(['x', 'y', 'z'], {-1, 1}, name='OR1'))
>>> csp.check({'x': 1, 'y': -1, 'z': 1})
True
```

dwavebinarycsp.factories.constraint.gates.xor_gate

xor_gate (variables, vartype=<Vartype.BINARY: frozenset({0, 1})>, name='XOR')

XOR gate.

Parameters
• **variables** *(list)* – Variable labels for the and gate as `[in1, in2, out]`, where `in1, in2` are inputs and `out` the gate’s output.

• **vartype** *(Vartype, optional, default='BINARY')* – Variable type. Accepted input values:
  – Vartype.SPIN, ‘SPIN’, {-1, 1}
  – Vartype.BINARY, ‘BINARY’, {0, 1}

• **name** *(str, optional, default='XOR')* – Name for the constraint.

**Returns** Constraint that is satisfied when its variables are assigned values that match the valid states of an XOR gate.

**Return type** `Constraint(Constraint)`

**Examples**

```python
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.xor_gate(['x', 'y', 'z'], name='XOR1'))
>>> csp.check({'x': 1, 'y': 1, 'z': 1})
False
```

dwavebinarycsp.factories.constraint.gates.halfadder_gate

**halfadder_gate** *(variables, vartype=<Vartype.BINARY: frozenset({0, 1})>, name='HALF_ADDER')* Half adder.

**Parameters**

• **variables** *(list)* – Variable labels for the and gate as `[in1, in2, sum, carry]`, where `in1, in2` are inputs to be added and `sum` and `carry` the resultant outputs.

• **vartype** *(Vartype, optional, default='BINARY')* – Variable type. Accepted input values:
  – Vartype.SPIN, ‘SPIN’, {-1, 1}
  – Vartype.BINARY, ‘BINARY’, {0, 1}

• **name** *(str, optional, default='HALF_ADDER')* – Name for the constraint.

**Returns** Constraint that is satisfied when its variables are assigned values that match the valid states of a Boolean half adder.

**Return type** `Constraint(Constraint)`

**Examples**

```python
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.halfadder_gate(['a', 'b', 'total', 'carry'], name='HA1'))
>>> csp.check({'a': 1, 'b': 1, 'total': 0, 'carry': 1})
True
```
**fulladder_gate** *(variables, vartype=<Vartype.BINARY: frozenset({0, 1})>, name='FULL_ADDER')*

Full adder.

**Parameters**

- **variables** *(list)* – Variable labels for the and gate as *\[in1, in2, in3, sum, carry\]*, where *in1, in2, in3* are inputs to be added and *sum* and ‘carry’ the resultant outputs.
- **vartype** *(Vartype, optional, default='BINARY')* – Variable type. Accepted input values:
  - Vartype.SPIN, ‘SPIN’, {-1, 1}
  - Vartype.BINARY, ‘BINARY’, {0, 1}
- **name** *(str, optional, default='FULL_ADDER')* – Name for the constraint.

**Returns** Constraint that is satisfied when its variables are assigned values that match the valid states of a Boolean full adder.

**Examples**

```python
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.fulladder_gate(['a', 'b', 'c_in', 'total', 'c_out'],
                                          name='FA1'))
>>> csp.check({'a': 1, 'b': 0, 'c_in': 1, 'total': 0, 'c_out': 1})
True
```

**Satisfiability Problems**

---

**sat2in4** *(pos[, neg, vartype, name])* Two-in-four (2-in-4) satisfiability.

---

**sat2in4** *(pos, neg=(), vartype=<Vartype.BINARY: frozenset({0, 1})>, name='2-in-4')* Two-in-four (2-in-4) satisfiability.

**Parameters**

- **pos** *(iterable)* – Variable labels, as an iterable, for non-negated variables of the constraint. Exactly four variables are specified by *pos* and *neg* together.
- **neg** *(tuple)* – Variable labels, as an iterable, for negated variables of the constraint. Exactly four variables are specified by *pos* and *neg* together.
- **vartype** *(Vartype, optional, default='BINARY')* – Variable type. Accepted input values:
  - Vartype.SPIN, ‘SPIN’, {-1, 1}
  - Vartype.BINARY, ‘BINARY’, {0, 1}
- **name** *(str, optional, default='2-in-4')* – Name for the constraint.
**Returns** Constraint that is satisfied when its variables are assigned values that satisfy a two-in-four satisfiability problem.

**Return type** Constraint($\text{Constraint}$)

**Examples**

```python
>>> import dwavebinarycsp.factories.constraint.sat as sat
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(sat.sat2in4(['w', 'x', 'y', 'z'], vartype='BINARY', name='sat1'))
>>> csp.check({'w': 1, 'x': 1, 'y': 0, 'z': 0})
True
```

---

**CSPs**

<table>
<thead>
<tr>
<th>circuits.multiplication_circuit(nbit[, vartype])</th>
<th>Multiplication circuit constraint satisfaction problem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>sat.random_2in4sat(num_variables, num_clauses)</td>
<td>Random two-in-four (2-in-4) constraint satisfaction problem.</td>
</tr>
<tr>
<td>sat.random_xorsat(num_variables, num_clauses)</td>
<td>Random XOR constraint satisfaction problem.</td>
</tr>
</tbody>
</table>

---

**dwavebinarycsp.factories.csp.circuits.multiplication_circuit**

**multiplication_circuit** (nbit, vartype=$\langle\text{Vartype.BINARY: frozenset([0, 1])}\rangle$)

Multiplication circuit constraint satisfaction problem.

A constraint satisfaction problem that represents the binary multiplication $a \cdot b = p$, where the multiplicands are binary variables of length $nbit$; for example, $2^n a_{nbit} + \ldots + 4 a_2 + 2 a_1 + a_0$.

The square below shows a graphic representation of the circuit:

```
| | and20 | and10 | and00 |
| | | | |
| | and21 | add11 | add01 | add01 |
| | | | |
| | | | |
| | and22 | add12 | add02 | add02 |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
```

| p5 | p4 | p3 | p2 | p1 | p0 |
|----------------------|------------------|

**Parameters**

- **nbit** ($\text{int}$) – Number of bits in the multiplicands.
- **vartype** ($\text{Vartype}$, optional, default='BINARY') – Variable type. Accepted input values:
  - Vartype.SPIN, `SPIN`, [-1, 1]
Returns CSP that is satisfied when variables $a, b, p$ are assigned values that correctly solve binary multiplication $ab = p$.

Return type CSP (ConstraintSatisfactionProblem)

Examples

This example creates a multiplication circuit CSP that multiplies two 3-bit numbers, which is then formulated as a binary quadratic model (BQM). It fixes the multiplicands as $a = 5, b = 3$ (101 and 011) and uses a simulated annealing sampler to find the product, $p = 15$ (001111).

```python
>>> from dwavebinarycsp.factories.csp.circuits import multiplication_circuit
>>> import neal

>>> csp = multiplication_circuit(3)
>>> bqm = dwavebinarycsp.stitch(csp)
>>> bqm.fix_variable('a0', 1); bqm.fix_variable('a1', 0); bqm.fix_variable('a2', 1)
>>> bqm.fix_variable('b0', 1); bqm.fix_variable('b1', 1); bqm.fix_variable('b2', 0)
>>> sampler = neal.SimulatedAnnealingSampler()
>>> response = sampler.sample(bqm)
>>> p = next(response.samples(n=1, sorted_by='energy'))
>>> print(p['p5'], p['p4'], p['p3'], p['p2'], p['p1'], p['p0'])
# doctest: +SKIP
0 0 1 1 1 1
```

dwavebinarycsp.factories.csp.sat.random_2in4sat

random_2in4sat (num_variables, num_clauses, vartype=Vartype.BINARY: frozenset({0, 1}), satisfiable=True)
Random two-in-four (2-in-4) constraint satisfaction problem.

Parameters

- **num_variables** (integer) – Number of variables (at least four).
- **num_clauses** (integer) – Number of constraints that together constitute the constraint satisfaction problem.
- **vartype** (Vartype, optional, default='BINARY') – Variable type. Accepted input values:
  - Vartype.SPIN, ‘SPIN’, {-1, 1}
  - Vartype.BINARY, ‘BINARY’, {0, 1}
- **satisfiable** (bool, optional, default=True) – True if the CSP can be satisfied.

Returns CSP that is satisfied when its variables are assigned values that satisfy a two-in-four satisfiability problem.

Return type CSP (ConstraintSatisfactionProblem)
Examples

This example creates a CSP with 6 variables and two random constraints and checks whether a particular assignment of variables satisfies it.

```python
>>> import dwavebinarycsp.factories as sat
>>> csp = sat.random_2in4sat(6, 2)
>>> csp.constraints  # doctest: +SKIP
[Constraint.from_configurations(frozenset({(1, 0, 1, 0), (1, 0, 0, 1), (1, 1, 1, 0), (0, 1, 0, 1), (0, 0, 0, 0), (0, 1, 0, 0)}), (2, 4, 0, 1), Vartype.BINARY, name='2-in-4'),
Constraint.from_configurations(frozenset({(1, 0, 1, 1), (1, 1, 0, 1), (1, 1, 1, 0), (0, 0, 0, 1), (0, 1, 0, 0), (0, 0, 1, 0)}), (1, 2, 4, 5), Vartype.BINARY, name='2-in-4'))
>>> csp.check({0: 1, 1: 0, 2: 1, 3: 1, 4: 0, 5: 0})  # doctest: +SKIP
True
```

dwavebinarycsp.factories.csp.sat.random_xorsat

`random_xorsat(num_variables, num_clauses, vartype=<Vartype.BINARY: frozenset({0, 1}), satisfiable=True)`

Random XOR constraint satisfaction problem.

Parameters

- **num_variables** (`integer`) – Number of variables (at least three).
- **num_clauses** (`integer`) – Number of constraints that together constitute the constraint satisfaction problem.
- **vartype** (`Vartype`, optional, default='BINARY') – Variable type. Accepted input values:
  - Vartype.SPIN, ‘SPIN’, {-1, 1}
  - Vartype.BINARY, ‘BINARY’, {0, 1}
- **satisfiable** (`bool`, optional, default=True) – True if the CSP can be satisfied.

Returns CSP that is satisfied when its variables are assigned values that satisfy a XOR satisfiability problem.

Return type CSP (`ConstraintSatisfactionProblem`)

Examples

This example creates a CSP with 5 variables and two random constraints and checks whether a particular assignment of variables satisfies it.

```python
>>> import dwavebinarycsp.factories as sat
>>> csp = sat.random_xorsat(5, 2)
>>> csp.constraints  # doctest: +SKIP
[Constraint.from_configurations(frozenset({(1, 0, 0), (1, 1, 1), (0, 1, 0), (0, 0, 1)}), (4, 3, 0), Vartype.BINARY, name='XOR (0 flipped)'),
Constraint.from_configurations(frozenset({(1, 1, 0), (0, 1, 1), (0, 0, 0), (1, 0, 1)}), (2, 0, 4), Vartype.BINARY, name='2-in-4'))
>>> csp.check({0: 1, 1: 1, 2: 1, 3: 1, 4: 0, 5: 0})  # doctest: +SKIP
True
```
6.2.3 Examples

Release 2.5.0
Date Aug 12, 2020

Circuit Fault Diagnosis

Fault diagnosis is the combinational problem of quickly localizing failures as soon as they are detected in systems. Circuit fault diagnosis (CFD) is the problem of identifying a minimum-sized set of gates that, if faulty, explains an observation of incorrect outputs given a set of inputs.

Fig. 5: A Half Adder made up of an XOR gate and an AND gate.

Fig. 6: A Full Adder made up of two Half Adders.

The following example demonstrates some of the techniques available to formulate a given problem so it can be solved on the D-Wave system.

Circuit Fault Diagnosis with Explicit Fault Variables

We can construct the constraints for the circuit fault diagnosis in the following way:

- Each input/output/wire in the circuit is represented by a binary variable in the problem.

- Each gate can either be:
  - Healthy, in which case it behaves according to its normal truth table.
  - Faulty, in which case it does not.
To build these constraints, we start with the truth table for the gate we wish to encode, say an AND gate:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We then add a new *explicit fault variable*, which encodes whether the gate is faulty or not.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Output</th>
<th>Faulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

This new truth table with the explicit fault variable allows the CSP to be satisfied even when the gate is not healthy.

**Example Code**

The following code demonstrates how to find both fault diagnoses and minimum fault diagnoses for the above Full Adder.

```python
import dwavebinarycsp
from dimod import ExactSolver

def xor_fault(a, b, out, fault):
    """Returns True if XOR(a, b) == out and fault == 0 or XOR(a, b) != out and fault == 1."""
    if (a != b) == out:
        return fault == 0
    else:
        return fault == 1

def and_fault(a, b, out, fault):
    """Returns True if AND(a, b) == out and fault == 0 or AND(a, b) != out and fault == 1."""
    if (a and b) == out:
        return fault == 0
    else:
        return fault == 1

def or_fault(a, b, out, fault):
    """Returns True if OR(a, b) == out and fault == 0 or OR(a, b) != out and fault == 1."""
```

(continues on next page)
if (a or b) == out:
    return fault == 0
else:
    return fault == 1

csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)

# the first half adder
csp.add_constraint(xor fault, ['A1', 'B1', 'S1/A2', 'xor_fault_1'])
csp.add_constraint(and fault, ['A1', 'B1', 'C1', 'and_fault_1'])

# the second half adder
# csp.add_constraint(xor fault, ['S1/A2', 'B2', 'S2', 'xor_fault_2'])
csp.add_constraint(and fault, ['S1/A2', 'B2', 'C2', 'and_fault_2'])

# finally the AND gate
# csp.add_constraint(or fault, ['C1', 'C2', 'ORout', 'or_fault'])

csp.add_constraint(or fault, ['C1', 'C2', 'ORout', 'or_fault'])

# now, say that the behaviour we witnessed was HA(0, 1, 0) -> 1, 1.
# The 'A' input to the circuit is 'A1'
csp.fix_variable('A1', 0)
# The 'B' input to the circuit is 'B1'
csp.fix_variable('B1', 1)
# the 'Cin' input to the circuit is 'B2'
csp.fix_variable('B2', 0)
# the sum output of the circuit is 'S2'
csp.fix_variable('S2', 1)
# the carry output of the circuit is 'ORout'
csp.fix_variable('ORout', 1)

# convert the csp to a bqm. We specify that the energy gap between the valid
# configurations and
# the invalid ones must be at least 2.0
bqm = dwavebinarycsp.stitch(csp, min_classical_gap=2.0)

# set up any dimod solver. In this case we use the ExactSolver but any unstructured
# solver would
# work.
sampler = ExactSolver()

# we can determine the minimum and maximum number of faults that will induce this
# behavior
response = sampler.sample(bqm)
min_energy = min(response.data_vectors['energy'])

fault_counts = []
for sample, energy in response.data([sample', 'energy']):
    if csp.check(sample):
        n_faults = sum(sample[v] for v in sample if 'fault' in v)
        fault_counts.append(n_faults)
    else:
        # if the CSP is not satisfied, the energy should be above ground
assert energy > min_energy

print('Minimum number of faults: ', min(fault_counts))
print('Maximum number of faults: ', max(fault_counts))

# If, instead of the ground states corresponding to all possible fault configurations, we instead only wanted to sample from minimum fault configurations, we need to bias against higher fault cardinalities. To do this, we add a small linear bias to the fault variables. We also make sure that the bias we add is less than 2.0, or else we would affect the energy levels.
bqm.add_variable('xor_fault_1', .5)  # if the variable is present, add_variable adds to the linear bias
bqm.add_variable('and_fault_1', .5)
bqm.add_variable('xor_fault_2', .5)
bqm.add_variable('and_fault_2', .5)
bqm.add_variable('or_fault', .5)

# now the samples that satisfy the csp and are minimum energy should be exactly the fault diagnosis with only a single fault
response = sampler.sample(bqm)
min_energy = min(response.data_vectors['energy'])

min_fault_diagnoses = []
for sample, energy in response.data(data_vectors=['sample', 'energy']):
    if csp.check(sample) and energy == min_energy:
        min_fault_diagnoses.append([v for v in sample if ('fault' in v and sample[v])])
    else:
        # if the CSP is not satisfied, the energy should be above ground
        assert energy > min_energy

print('min fault diagnoses: ', min_fault_diagnoses)

6.2.4 Bibliography

6.3 dwave-cloud-client

D-Wave Cloud Client is a minimal implementation of the REST interface used to communicate with D-Wave Sampler API (SAPI) servers.

SAPI is an application layer built to provide resource discovery, permissions, and scheduling for quantum annealing resources at D-Wave Systems. This package provides a minimal Python interface to that layer without compromising the quality of interactions and workflow.

The example below instantiates a D-Wave Cloud Client and solver based on the local system’s auto-detected default configuration file and samples a random Ising problem tailored to fit the solver’s graph.

```python
import random
from dwave.cloud import Client
```
# Connect using the default or environment connection information

```python
with Client.from_config() as client:

    # Load the default solver
    solver = client.get_solver()

    # Build a random Ising model to exactly fit the graph the solver supports
    linear = {index: random.choice([-1, 1]) for index in solver.nodes}
    quad = {key: random.choice([-1, 1]) for key in solver.undirected_edges}

    # Send the problem for sampling, include solver-specific parameter 'num_reads'
    computation = solver.sample_ising(linear, quad, num_reads=100)

    # Print the first sample out of a hundred
    print(computation.samples[0])
```

## 6.3.1 Introduction

D-Wave Cloud Client is a minimal implementation of the REST interface used to communicate with D-Wave Sampler API (SAPI) servers.

SAPI is an application layer built to provide resource discovery, permissions, and scheduling for quantum annealing resources at D-Wave Systems. This package provides a minimal Python interface to that layer without compromising the quality of interactions and workflow.

The D-Wave Cloud Client `Solver` class enables low-level control of problem submission. It is used, for example, by the `dwave-system DWaveSampler`, which enables quick incorporation of the D-Wave system as a sampler in your code.

### Configuration

It’s recommended you set up your D-Wave Cloud Client configuration through the interactive CLI utility.

As described in the Using a D-Wave System section of Ocean Documentation, for your code to access remote D-Wave compute resources, you must configure communication through SAPI; for example, your code needs your API token for authentication. D-Wave Cloud Client provides multiple options for configuring the required information:

- One or more locally saved configuration files
- Environment variables
- Direct setting of key values in functions

These options can be flexibly used together.

### Configuration Files

If a D-Wave Cloud Client configuration file is not explicitly specified when instantiating a client or solver, auto-detection searches for candidate files in a number of standard directories, depending on your local system’s operating system. You can see the standard locations with the `get_configfile_paths()` method.

For example, on a Unix system, depending on its flavor, these might include (in order):
On Windows 7+, configuration files are expected to be located under:

```
C:\Users\<username>\AppData\Local\dwavesystem\dwave\dwave.conf
```

On Mac OS X, configuration files are expected to be located under:

```
~/.Library/Application Support/dwave/dwave.conf
```

(For details on the D-Wave API for determining platform-independent paths to user data and configuration folders see the `homebase` tool.)

You can check the directories searched by `get_configfile_paths()` from a console using the interactive CLI utility; for example:

```
$ dwave config ls -m
/etc/xdg/xdg-ubuntu/dwave/dwave.conf
/usr/share/upstart/xdg/dwave/dwave.conf
/etc/xdg/dwave/dwave.conf
/home/jane/.config/dwave/dwave.conf
./dwave.conf
```

A single D-Wave Cloud Client configuration file can contain multiple profiles, each defining a separate combination of communication parameters such as the URL to the remote resource, authentication token, solver, etc. Configuration files conform to a standard Windows INI-style format: profiles are defined by sections such as, `[profile-a]` and `[profile-b]`. Default values for undefined profile keys are taken from the `[defaults]` section.

For example, if the configuration file, `~/.config/dwave/dwave.conf`, selected through auto-detection as the default configuration, contains the following profiles:

```
[defaults]
token = ABC-123456789123456789123456789

[first-available-qpu]
solver = {"qpu": true}

[software]
client = sw
solver = c4-sw_sample
token = DEF-987654321987654321987654321
proxy = http://user:pass@myproxy.com:8080/

[backup-dwave2000q]
endpoint = https://url.of.my.backup.dwavesystem.com/sapi
solver = {"num_qubits__gt": 2000}
```

You can instantiate clients for a D-Wave system and a CPU with:

```
>>> from dwave.cloud import Client
>>> client_qpu = Client.from_config()
>>> client_cpu = Client.from_config(profile='software')
```
Environment Variables

In addition to files, you can set configuration information through environment variables; for example:

- `DWAVE_CONFIG_FILE` may select the configuration file path.
- `DWAVE_PROFILE` may select the name of a profile (section).
- `DWAVE_API_TOKEN` may select the API token.

For details on supported environment variables and prioritizing between these and values set explicitly or through a configuration file, see `dwave.cloud.config`.

Work Flow

A typical workflow may include the following steps:

1. Instantiate a `Client` to manage communication with remote `solver` resources, selecting and authenticating access to available solvers; for example, you can list all solvers available to a client with its `get_solvers()` method and select and return one with its `get_solver()` method.

   Preferred use is with a context manager—a `with Client.from_config(...) as` construct—to ensure proper closure of all resources. The following example snippet creates a client based on an auto-detected configuration file and instantiates a solver.

   ```python
   >>> with Client.from_config() as client:
   ...     solver = client.get_solver(qpu=True)
   ```

   Alternatively, the following example snippet creates a client for software resources that it later explicitly closes.

   ```python
   >>> client = Client.from_config(client='sw')
   >>> # code that uses client
   >>> client.close()
   ```

2. Instantiate a selected `Solver`, a resource for solving problems. Solvers are responsible for:

   - Encoding submitted problems
   - Checking submitted parameters
   - Adding problems to a client’s submission queue

   Solvers that provide sampling for solving `Ising` and `QUBO` problems, such as a D-Wave 2000Q sampler `DWaveSampler` or software sampler `SimulatedAnnealingSampler`, might be remote resources.

3. Submit your problem, using your solver, and then process the returned `Future`, instantiated by your solver to handle remotely executed problem solving.

Terminology

**Ising** Traditionally used in statistical mechanics. Variables are “spin up” (↑) and “spin down” (↓), states that correspond to +1 and −1 values. Relationships between the spins, represented by couplings, are correlations or anti-correlations. The objective function expressed as an Ising model is as follows:

$$E_{\text{ising}}(s) = \sum_{i=1}^{N} h_i s_i + \sum_{i=1}^{N} \sum_{j=i+1}^{N} J_{i,j}s_is_j$$ (6.1)
where the linear coefficients corresponding to qubit biases are $h_i$, and the quadratic coefficients corresponding to coupling strengths are $J_{i,j}$.

A collection of variables with associated linear and quadratic biases.

Quadratic unconstrained binary optimization. QUBO problems are traditionally used in computer science. Variables are TRUE and FALSE, states that correspond to 1 and 0 values. A QUBO problem is defined using an upper-diagonal matrix $Q$, which is an $N \times N$ upper-triangular matrix of real weights, and $x$, a vector of binary variables, as minimizing the function

$$f(x) = \sum_i Q_{i,i} x_i + \sum_{i<j} Q_{i,j} x_i x_j \quad (6.2)$$

where the diagonal terms $Q_{i,i}$ are the linear coefficients and the nonzero off-diagonal terms are the quadratic coefficients $Q_{i,j}$. This can be expressed more concisely as

$$\min_{x \in \{0,1\}^n} x^T Q x \quad (6.3)$$

In scalar notation, the objective function expressed as a QUBO is as follows:

$$E_{qubo}(a_i, b_{i,j}; q_i) = \sum_i a_i q_i + \sum_{i<j} b_{i,j} q_i q_j \quad (6.4)$$

A process that samples from low energy states of a problem’s objective function. A binary quadratic model (BQM) sampler samples from low energy states in models such as those defined by an Ising equation or a Quadratic Unconstrained Binary Optimization (QUBO) problem and returns an iterable of samples, in order of increasing energy. A dimod sampler provides ‘sample_qubo’ and ‘sample_ising’ methods as well as the generic BQM sampler method.

A resource that runs a problem. Some solvers interface to the QPU; others leverage CPU and GPU resources.

### 6.3.2 Reference Documentation

- **Release**: 2.5.0
- **Date**: Aug 12, 2020
- noindex
Configuration

Configuration for communicating with a solver.

Communicating with a solver—submitting a problem, monitoring its progress, receiving samples—requires configuration of several parameters such as the selected solver, its URL, an API token, etc. D-Wave Cloud Client provides multiple options for configuring those parameters:

- One or more locally saved configuration files.
- Environment variables
- Direct setting of key values in functions

These options can be flexibly used together. The standard use is through the `from_config()` classmethod.

Configuration values can be specified in multiple ways, ranked in the following order (with 1 the highest ranked):

1. Values specified as keyword arguments
2. Values specified as environment variables
3. Values specified in the configuration file

Configuration files comply with standard Windows INI-like format, parsable with Python’s `configparser`. An optional `defaults` section provides default key-value pairs for all other sections. User-defined key-value pairs (unrecognized keys) are passed through to the client.

Typically configuration files are created, inspected, and changed using interactive CLI commands from your system’s console, such as `dwave config create` and `dwave config inspect` (run `dwave --help` for information on CLI options).

Environment variables:

- `DWAVE_CONFIG_FILE`: Configuration file path.
- `DWAVE_PROFILE`: Name of profile (section).
- `DWAVE_API_CLIENT`: API client class. Supported values are `qpu` or `sw`.
- `DWAVE_API_ENDPOINT`: API endpoint URL.
- `DWAVE_API_TOKEN`: API authorization token.
- `DWAVE_API_SOLVER`: Default solver.
- `DWAVE_API_PROXY`: URL for proxy connections to D-Wave API.
- `DWAVE_API_HEADERS`: Optional additional HTTP headers.

Examples

The following are typical examples of using `from_config()` to create a configured client.

This first example initializes `Client` from an explicitly specified configuration file, `~/.jane/my_path_to_config/my_cloud_conf.conf`:

```python
[defaults]
token = ABC-123456789123456789123456789

[first-qpu]
solver = {"qpu": true}

[feature]
```

(continues on next page)
endpoint = https://url.of.some.dwavesystem.com/sapi
token = DEF-987654321987654321987654321
solver = {'num_qubits__gte': 2000, 'max_anneal_schedule_points__gte': 4}

The example code below creates a client object that connects to a D-Wave QPU, using `dwave.cloud.qpu`. `Client` and the first available online D-Wave system at the default API endpoint URL (https://cloud.dwavesys.com/sapi). The feature profile specifies a solver selected based on available features, namely we’re requesting the first solver that has at least 2000 qubits and the anneal schedule can be described with at least 4 points.

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config(config_file='~/.jane/my_path_to_config/my_cloud.conf')
# doctest: +SKIP
# code that uses client
>>> client.close()  # doctest: +SKIP
```

This second example auto-detects a configuration file on the local system following the user/system configuration paths of `get_configfile_paths()`. It passes through to the instantiated client an unrecognized key-value pair `my_param='my_value'`.

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config(my_param='my_value')
# doctest: +SKIP
# code that uses client
>>> client.close()  # doctest: +SKIP
```

This third example instantiates two clients, for managing both QPU and software solvers. Common key-value pairs are taken from the defaults section of a shared configuration file:

```
[defaults]
token = ABC-123456789123456789123456789

[primary-qpu]
solver = {'qpu': true}

[sw-solver]
client = sw
solver = c4-sw_sample
default_solver = c4-sw_sample
endpoint = https://url.of.some.software.resource.com/my_if
token = DEF-987654321987654321987654321

[backup-qpu]
solver = {'qpu': true, 'num_qubits__gte': 2000}
default_solver = 'EXAMPLE_2000Q_SYSTEM_A'
endpoint = https://url.of.some.dwavesystem.com/sapi
proxy = http://user:pass@myproxy.com:8080/
token = XYZ-0101010100112341234123412341234
```

The example code below creates client objects for two QPU solvers (at the same URL but each with its own solver ID and token) and one software solver.

```python
>>> from dwave.cloud import Client
>>> client_qpu1 = Client.from_config(profile='primary-qpu')  # doctest: +SKIP
>>> client_qpu2 = Client.from_config(profile='backup-qpu')  # doctest: +SKIP
>>> client_sw1 = Client.from_config(profile='sw-solver')  # doctest: +SKIP
>>> client_qpu1.default_solver  # doctest: +SKIP
'EXAMPLE_2000Q_SYSTEM_A'
>>> client_qpu2.endpoint  # doctest: +SKIP
'https://url.of.some.dwavesystem.com/sapi'
```
This fourth example loads configurations auto-detected in more than one configuration file, with the higher priority file (in the current working directory) supplementing and overriding values from the lower priority user-local file. After instantiation, an endpoint from the default section and client from the profile section is provided from the user-local /usr/local/share/dwave/dwave.conf file:

```
[defaults]
solver = {"qpu": true}

[dw2000]
episode = https://int.se.dwavesystems.com/sapi
token = ABC-123456789123456789123456789
```

A solver is supplemented from the file in the current working directory, which also overrides the token value. ./dwave.conf is the file in the current directory:

```
[dw2000]
token = DEF-987654321987654321987654321
```

The next example uses load_config() to load profile values. **Most users do not need to use this method.** It loads from the following configuration file, dwave_c.conf, located in the current working directory, and specified explicitly:

```
[defaults]
endpoint = https://url.of.some.dwavesystem.com/sapi
solver = {"qpu": true}

[dw2000a]
solver = {"software": true, "name": "EXAMPLE_2000Q"}
token = ABC-123456789123456789123456789

[dw2000b]
solver = {"qpu": true}
token = DEF-987654321987654321987654321
```

This configuration file contains two profiles in addition to the defaults section. In the following example code, first no profile is specified, and the first profile after the defaults section is loaded with the solver overridden by the environment variable. Next, the second profile is selected with the explicitly named solver overriding the environment variable setting.
>>> import dwave.cloud as dc
>>> import os
>>> os.environ['DWAVE_API_SOLVER'] = 'EXAMPLE_2000Q_SYSTEM' # doctest: +SKIP
>>> dc.config.load_config("./dwave_c.conf") # doctest: +SKIP
{'client': 'sw',
'endpoint': 'https://url.of.some.dwavesystem.com/sapi',
'proxy': None,
'solver': 'EXAMPLE_2000Q_SYSTEM',
'token': 'ABC-123456789123456789123456789'}

Methods

Most users do not need to use these methods.

Loading Configuration

These functions deploy D-Wave Cloud Client settings from a configuration file.

\[\text{load_config}([\text{config_file, profile, client, ...}])\] Load D-Wave Cloud Client configuration based on a configuration file.

dwave.cloud.config.load_config

\text{load_config}(\text{config_file}=\text{None}, \text{profile}=\text{None}, \text{client}=\text{None}, \text{endpoint}=\text{None}, \text{token}=\text{None}, \text{solver}=\text{None}, \text{proxy}=\text{None}, \text{headers}=\text{None})  
Load D-Wave Cloud Client configuration based on a configuration file.

Configuration values can be specified in multiple ways, ranked in the following order (with 1 the highest ranked):

1. Values specified as keyword arguments in \text{load_config}(). These values replace values read from a configuration file, and therefore must be \text{strings}, including float values for timeouts, boolean flags (tested for “truthiness”), and solver feature constraints (a dictionary encoded as JSON).
2. Values specified as environment variables.
3. Values specified in the configuration file.

Configuration-file format is described in \text{dwave.cloud.config}.

If the location of the configuration file is not specified, auto-detection searches for existing configuration files in the standard directories of \text{get_configfile_paths}().

If a configuration file explicitly specified, via an argument or environment variable, does not exist or is unreadable, loading fails with \text{ConfigFileReadError}. Loading fails with \text{ConfigFileParseError} if the file is readable but invalid as a configuration file.
Similarly, if a profile explicitly specified, via an argument or environment variable, is not present in the loaded configuration, loading fails with `ValueError`. Explicit profile selection also fails if the configuration file is not explicitly specified, detected on the system, or defined via an environment variable.

Environment variables: `DWave_CONFIG_FILE`, `DWave_PROFILE`, `DWave_API_CLIENT`, `DWave_API_ENDPOINT`, `DWave_API_TOKEN`, `DWave_API_SOLVER`, `DWave_API_PROXY`, `DWave_API_HEADERS`.

Environment variables are described in `dwave.cloud.config`.

Parameters

- **config_file**(str/[str]/None/False/True, default=None) – Path to configuration file(s).
  
  If `None`, the value is taken from `DWave_CONFIG_FILE` environment variable if defined. If the environment variable is undefined or empty, auto-detection searches for existing configuration files in the standard directories of `get_configfile_paths()`.
  
  If `False`, loading from file(s) is skipped; if `True`, forces auto-detection (regardless of the `DWave_CONFIG_FILE` environment variable).

- **profile**(str, default=None) – Profile name (name of the profile section in the configuration file).
  
  If undefined, inferred from `DWave_PROFILE` environment variable if defined. If the environment variable is undefined or empty, a profile is selected in the following order:
  
  1. From the default section if it includes a profile key.
  2. The first section (after the default section).
  3. If no other section is defined besides `[defaults]`, the defaults section is promoted and selected.

- **client**(str, default=None) – Client type used for accessing the API. Supported values are `qpu` for `dwave.cloud.qpu.Client` and `sw` for `dwave.cloud.sw.Client`.

- **endpoint**(str, default=None) – API endpoint URL.

- **token**(str, default=None) – API authorization token.

- **solver**(dict/str, default=None) – `solver` features, as a JSON-encoded dictionary of feature constraints, the client should use. See `get_solvers()` for semantics of supported feature constraints.
  
  If undefined, the client uses a solver definition from environment variables, a configuration file, or falls back to the first available online solver.
  
  For backward compatibility, solver name in string format is accepted and converted to `{"name": <solver name>}`.

- **proxy**(str, default=None) – URL for proxy to use in connections to D-Wave API. Can include username/password, port, scheme, etc. If undefined, client uses the system-level proxy, if defined, or connects directly to the API.

- **headers**(dict/str, default=None) –

  Header lines to include in API calls, each line formatted as `Key: value`, or a parsed dictionary.
**Returns**  Mapping of configuration keys to values for the profile (section), as read from the configuration file and optionally overridden by environment values and specified keyword arguments. Always contains the *client*, *endpoint*, *token*, *solver*, and *proxy* keys.

**Return type**  `dict`

**Raises**

- `ValueError` – Invalid (non-existing) profile name.
- `ConfigFileReadError` – Config file specified or detected could not be opened or read.
- `ConfigFileParseError` – Config file parse failed.

**Examples**

This example loads the configuration from an auto-detected configuration file in the home directory of a Windows system user.

```python
>>> from dwave.cloud import config
>>> config.load_config()  # doctest: +SKIP
{'client': 'qpu',
 'endpoint': 'https://url.of.some.dwavesystem.com/sapi',
 'proxy': None,
 'solver': 'EXAMPLE_2000Q_SYSTEM_A',
 'token': 'DEF-987654321987654321987654321',
 'headers': None}
```

... # See which configuration file was loaded
```python
>>> config.get_configfile_paths()  # doctest: +SKIP
['C:\Users\jane\AppData\Local\dwavesystem\dwave\dwave.conf']
```

Additional examples are given in `dwave.cloud.config`.

**Managing Files**

These functions manage your D-Wave Cloud Client configuration files. It’s recommended you set up your configuration through the interactive CLI utility instead.

```python
get_configfile_paths(...)  Return a list of local configuration file paths.
get_configfile_path()     Return the highest-priority local configuration file.
get_default_configfile_path()  Return the default configuration-file path.
```

**get_configfile_paths**

*(system=True, user=True, local=True, only_existing=True)*

Return a list of local configuration file paths.

Search paths for configuration files on the local system are based on homedir and depend on operating system; for example, for Linux systems these might include `dwave.conf` in the current working directory (CWD), user-local . `config/dwave/`, and system-wide /etc/dwave/.

**Parameters**

- `system` *(boolean, default=True)* – Search for system-wide configuration files.
- `user` *(boolean, default=True)* – Search for user-local configuration files.
- `local` *(boolean, default=True)* – Search for local configuration files (in CWD).
• only_existing (boolean, default=True) – Return only paths for files that exist on the local system.

Returns List of configuration file paths.

Return type list[str]

Examples
This example displays all paths to configuration files on a Windows system running Python 2.7 and then finds the single existing configuration file.

```python
>>> import dwave.cloud as dc
>>> # Display paths
>>> dc.config.get_configfile_paths(only_existing=False) # doctest: +SKIP
['C:\ProgramData\dwavesystem\dwave\dwave.conf',
 'C:\Users\jane\AppData\Local\dwavesystem\dwave\dwave.conf',
 '\dwave.conf']
>>> # Find existing files
>>> dc.config.get_configfile_paths() # doctest: +SKIP
['C:\Users\jane\AppData\Local\dwavesystem\dwave\dwave.conf']
```

dwave.cloud.config.get_configfile_path

get_configfile_path ()
Return the highest-priority local configuration file.

Selects the top-ranked configuration file path from a list of candidates returned by get_configfile_paths (), or None if no candidate path exists.

Returns Configuration file path.

Return type str

Examples
This example displays the highest-priority configuration file on a Windows system running Python 2.7.

```python
>>> import dwave.cloud as dc
>>> # Display paths
>>> dc.config.get_configfile_paths(only_existing=False) # doctest: +SKIP
['C:\ProgramData\dwavesystem\dwave\dwave.conf',
 'C:\Users\jane\AppData\Local\dwavesystem\dwave\dwave.conf',
 '\dwave.conf']
>>> # Find highest-priority local configuration file
>>> dc.config.get_configfile_path() # doctest: +SKIP
'C:\Users\jane\AppData\Local\dwavesystem\dwave\dwave.conf'
```

dwave.cloud.config.get_default_configfile_path

get_default_configfile_path ()
Return the default configuration-file path.

Typically returns a user-local configuration file: e.g: ~/.config/dwave/dwave.conf.
Returns  Configuration file path.

Return type  str

Examples

This example displays the default configuration file on an Ubuntu Unix system running IPython 2.7.

```python
g>>> import dwave.cloud as dc
g>>> # Display paths
g>>> dc.config.get_configfile_paths(only_existing=False)  # doctest: +SKIP
['/etc/xdg/xdg-ubuntu/dwave/dwave.conf',
'#/usr/share/upstart/xdg/dwave/dwave.conf',
'/etc/xdg/dwave/dwave.conf',
'#/home/mary/.config/dwave/dwave.conf',
'./dwave.conf']
```  

Clients

The solvers that provide sampling for solving Ising and QUBO problems, such as a D-Wave 2000Q QPU or a software sampler such as the dimod simulated annealing sampler, are typically remote resources. The D-Wave Cloud Client Client class manages such remote solver resources.

Preferred use is with a context manager—a `with Client.from_config(...)` as construct—to ensure proper closure of all resources. The following example snippet creates a client based on an auto-detected configuration file and instantiates a solver.

```python
g>>> with Client.from_config() as client:  # doctest: +SKIP
...   solver = client.get_solver(num_qubits__gt=2000)
```  

Alternatively, the following example snippet creates a client for software resources that it later explicitly closes.

```python
g>>> client = Client.from_config(software=True)  # doctest: +SKIP
```  

Typically you use the Client class. By default, it instantiates a QPU client. You can also use the specialized QPU and CPU/GPU clients directly.

Client (Base Client)

D-Wave API clients handle communications with solver resources: problem submittal, monitoring, samples retrieval, etc.

Examples

This example creates a client using the local system’s default D-Wave Cloud Client configuration file, which is configured to access a D-Wave 2000Q QPU, submits a QUBO problem (a Boolean NOT gate represented by a penalty model), and samples 5 times.

```python
g```
```python
>>> from dwave.cloud import Client
>>> Q = {(0, 0): -1, (0, 4): 0, (4, 0): 2, (4, 4): -1}
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()
...     computation = solver.sample_qubo(Q, num_reads=5)
... >>> for i in range(5):  # doctest: +SKIP
...     print(computation.samples[i][0], computation.samples[i][4])
... (1, 0)
(1, 0)
(0, 1)
(0, 1)
```
```python
>>> from dwave.cloud import Client
>>> client = Client(token='secret')  # doctest: +SKIP
>>> # code that uses client
>>> client.close()  # doctest: +SKIP
```

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>client.Client.from_config(config_file, ...)</code></td>
<td>Client factory method to instantiate a client instance from configuration.</td>
</tr>
<tr>
<td><code>client.Client.solvers(refresh)</code></td>
<td>Deprecated in favor of <code>get_solvers()</code>.</td>
</tr>
<tr>
<td><code>client.Client.retrieve_answer(id)</code></td>
<td>Retrieve a problem by id.</td>
</tr>
<tr>
<td><code>client.Client.get_solver(name, refresh, ...)</code></td>
<td>Load the configuration for a single solver.</td>
</tr>
<tr>
<td><code>client.Client.get_solvers(refresh, order_by)</code></td>
<td>Return a filtered list of solvers handled by this client.</td>
</tr>
<tr>
<td><code>client.Client.is_solver_handled(solver)</code></td>
<td>Determine if the specified solver should be handled by this client.</td>
</tr>
<tr>
<td><code>client.Client.close()</code></td>
<td>Perform a clean shutdown.</td>
</tr>
</tbody>
</table>

### dwave.cloud.client.Client.from_config

**classmethod** `Client.from_config(config_file=None, profile=None, client=None, endpoint=None, token=None, solver=None, proxy=None, headers=None, legacy_config_fallback=False, **kwargs)`

Client factory method to instantiate a client instance from configuration.

Configuration values can be specified in multiple ways, ranked in the following order (with 1 the highest ranked):

1. Values specified as keyword arguments in `from_config()`
2. Values specified as environment variables
3. Values specified in the configuration file

Configuration-file format is described in `dwave.cloud.config`.

If the location of the configuration file is not specified, auto-detection searches for existing configuration files in the standard directories of `get_configfile_paths()`.

If a configuration file explicitly specified, via an argument or environment variable, does not exist or is unreadable, loading fails with `ConfigFileReadError`. Loading fails with `ConfigFileParseError` if the file is readable but invalid as a configuration file.

Similarly, if a profile explicitly specified, via an argument or environment variable, is not present in the loaded configuration, loading fails with `ValueError`. Explicit profile selection also fails if the configuration file is not explicitly specified, detected on the system, or defined via an environment variable.

Environment variables: `DWAVE_CONFIG_FILE`, `DWAVE_PROFILE`, `DWAVE_API_CLIENT`, `DWAVE_API_ENDPOINT`, `DWAVE_API_TOKEN`, `DWAVE_API_SOLVER`, `DWAVE_API_PROXY`, `DWAVE_API_HEADERS`.

Environment variables are described in `dwave.cloud.config`.

**Parameters**

- `config_file (str/[str]/None/False/True, default=None)` – Path to configuration file.
If None, the value is taken from DWAVE_CONFIG_FILE environment variable if defined. If the environment variable is undefined or empty, auto-detection searches for existing configuration files in the standard directories of get_configfile_paths().

If False, loading from file is skipped; if True, forces auto-detection (regardless of the DWAVE_CONFIG_FILE environment variable).

• profile (str, default=None) – Profile name (name of the profile section in the configuration file).

If undefined, inferred from DWAVE_PROFILE environment variable if defined. If the environment variable is undefined or empty, a profile is selected in the following order:

1. From the default section if it includes a profile key.
2. The first section (after the default section).
3. If no other section is defined besides [defaults], the defaults section is promoted and selected.

• client (str, default=None) – Client type used for accessing the API. Supported values are qpu for dwave.cloud.qpu.Client, sw for dwave.cloud.sw.Client and hybrid for dwave.cloud.hybrid.Client.

• endpoint (str, default=None) – API endpoint URL.

• token (str, default=None) – API authorization token.

• solver (dict/str, default=None) – Default solver features to use in get_solver().

Defined via dictionary of solver feature constraints (see get_solvers()). For backward compatibility, a solver name, as a string, is also accepted and converted to {"name": <solver name>}. If undefined, get_solver() uses a solver definition from environment variables, a configuration file, or falls back to the first available online solver.

• proxy (str, default=None) – URL for proxy to use in connections to D-Wave API. Can include username/password, port, scheme, etc. If undefined, client uses the system-level proxy, if defined, or connects directly to the API.

• headers (dict/str, default=None) – Newline-separated additional HTTP headers to include with each API request, or a dictionary of (key, value) pairs.

• legacy_config_fallback (bool, default=False) – If True and loading from a standard D-Wave Cloud Client configuration file (dwave.conf) fails, tries loading a legacy configuration file (~/.dwrc).

Other Parameters Unrecognized keys (str) – All unrecognized keys are passed through to the appropriate client class constructor as string keyword arguments.

An explicit key value overrides an identical user-defined key value loaded from a configuration file.

Returns Appropriate instance of a QPU/software/hybrid client.

Return type Client subclass

Raises

• ConfigFileReadError – Config file specified or detected could not be opened or read.

• ConfigFileParseError – Config file parse failed.
Examples

A variety of examples are given in `dwave.cloud.config`.

This example initializes `Client` from an explicitly specified configuration file, “~/.jane/my_path_to_config/my_cloud_conf.conf”:

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config(config_file='~/.jane/my_path_to_config/my_cloud_conf.conf')
>>> # code that uses client
>>> client.close()
```

dwave.cloud.client.Client.solvers

`Client.solvers(refresh=False, **filters)`

Deprecated in favor of `get_solvers()`.

dwave.cloud.client.Client.retrieve_answer

`Client.retrieve_answer(id_)`

Retrieve a problem by id.

Parameters

- `id_` *(str)* – As returned by `Future.id`.

Returns

`Future`

dwave.cloud.client.Client.get_solver

`Client.get_solver(name=None, refresh=False, order_by='avg_load', **filters)`

Load the configuration for a single solver.

Makes a blocking web call to `{endpoint}/solvers/remote/{solver_name}/`, where `{endpoint}` is a URL configured for the client, and returns a `Solver` instance that can be used to submit sampling problems to the D-Wave API and retrieve results.

Parameters

- `name` *(str)* – ID of the requested solver. `None` returns the default solver. If default solver is not configured, `None` returns the first available solver in `Client`’s class (QPU/software/base).
- `**filters` *(keyword arguments, optional)* – Dictionary of filters over features this solver has to have. For a list of feature names and values, see: `get_solvers()`.
- `order_by` *(callable/str/None, default='avg_load')* – Solver sorting key function (or `Solver` attribute/item dot-separated path). By default, solvers are sorted by average load. For details, see `get_solvers()`.
- `refresh` *(bool)* – Return solver from cache (if cached with `get_solvers()`), unless set to `True`.

Returns

`Solver`
Examples

This example creates two solvers for a client instantiated from a local system’s auto-detected default configuration file, which configures a connection to a D-Wave resource that provides two solvers. The first uses the default solver, the second explicitly selects another solver.

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config()  # doctest: +SKIP
>>> client.get_solvers()  # doctest: +SKIP
[Solver(id='2000Q_ONLINE_SOLVER1'), Solver(id='2000Q_ONLINE_SOLVER2')]
>>> solver1 = client.get_solver()  # doctest: +SKIP
>>> solver1.id  # doctest: +SKIP
'2000Q_ONLINE_SOLVER1'
>>> solver2 = client.get_solver(name='2000Q_ONLINE_SOLVER2')  # doctest: +SKIP
>>> solver2.id  # doctest: +SKIP
'2000Q_ONLINE_SOLVER2'
```

**dwave.cloud.client.Client.get_solvers**

`Client.get_solvers(refresh=False, order_by='avg_load', **filters)`

Return a filtered list of solvers handled by this client.

**Parameters**

- `refresh` (bool, default=False) – Force refresh of cached list of solvers/properties.
- `order_by` (callable/str/None, default='avg_load') – Solver sorting key function (or `Solver` attribute/item dot-separated path). By default, solvers are sorted by average load. To explicitly not sort the solvers (and use the API-returned order), set `order_by=None`.

Signature of the **key callable** is:

```python
key :: (Solver s, Ord k) => s -> k
```

Basic structure of the **key string path** is:

```python
"-"? (attr|item) ( "." (attr|item) )*  #
```

For example, to use solver property named `max_anneal_schedule_points`, available in `Solver.properties` dict, you can either specify a callable `key`:

```python
key=lambda solver: solver.properties['max_anneal_schedule_points']
```

or, you can use a short string path based key:

```python
key='properties.max_anneal_schedule_points'
```

Solver derived properties, available as `Solver` properties can also be used (e.g. `num_active_qubits`, `online`, `avg_load`, etc).

Ascending sort order is implied, unless the key string path does not start with `-`, in which case descending sort is used.
Note: the sort used for ordering solvers by key is **stable**, meaning that if multiple solvers have the same value for the key, their relative order is preserved, and effectively they are in the same order as returned by the API.

Note: solvers with None for key appear last in the list of solvers. When providing a key callable, ensure all values returned are of the same type (particularly in Python 3). For solvers with undefined key value, return None.

**filters** – See Filtering forms and Operators below.

Solver filters are defined, similarly to Django QuerySet filters, with keyword arguments of form `<key1>__. . . __<keyN>__[<operator>]=<value>`. Each `<operator>` is a predicate (boolean) function that acts on two arguments: value of feature `<name>` (described with keys path `<key1.key2. . . keyN>`) and the required `<value>`.

Feature `<name>` can be:

1) a derived solver property, available as an identically named Solver’s property (`name`, `qpu`, `hybrid`, `software`, `online`, `num_active_qubits`, `avg_load`)
2) a solver parameter, available in `Solver.parameters`
3) a solver property, available in `Solver.properties`
4) a path describing a property in nested dictionaries

Filtering forms are:

- `<derived_property>__<operator> (object <value>)`
- `<derived_property> (bool)`

This form ensures the value of solver’s property bound to `derived_property`, after applying `operator` equals the `value`. The default operator is `eq`.

For example:

```python
>>> client.get_solvers(avg_load__gt=0.5)
```

but also:

```python
>>> client.get_solvers(online=True)
>>> # identical to:
>>> client.get_solvers(online__eq=True)
```

- `<parameter>__<operator> (object <value>)`
- `<parameter> (bool)`

This form ensures that the solver supports `parameter`. General operator form can be used but usually does not make sense for parameters, since values are human-readable descriptions. The default operator is `available`.

Example:

```python
>>> client.get_solvers(flux_biases=True)
>>> # identical to:
>>> client.get_solvers(flux_biases__available=True)
```

- `<property>__<operator> (object <value>)`
- `<property> (bool)`
This form ensures the value of the solver’s property, after applying operator equals the righthand side value. The default operator is eq.

Note: if a non-existing parameter/property name/key given, the default operator is eq.

Operators are:

- **available (<name>: str, <value>: bool):** Test availability of <name> feature.
- **eq, lt, lte, gt, gte (<name>: str, <value>: any):** Standard relational operators that compare feature <name> value with <value>.
- **regex (<name>: str, <value>: str):** Test regular expression matching feature value.
- **covers (<name>: str, <value>: single value or range expressed as 2-tuple/list):** Test feature <name> value (which should be a range) covers a given value or a subrange.
- **within (<name>: str, <value>: range expressed as 2-tuple/list):** Test feature <name> value (which can be a single value or a range) is within a given range.
- **in (<name>: str, <value>: container type):** Test feature <name> value is in <value> container.
- **contains (<name>: str, <value>: any):** Test feature <name> value (container type) contains <value>.
- **issubset (<name>: str, <value>: container type):** Test feature <name> value (container type) is a subset of <value>.
- **issuperset (<name>: str, <value>: container type):** Test feature <name> value (container type) is a superset of <value>.

Derived properties are:

- **name (str):** Solver name/id.
- **qpu (bool):** Solver is a QPU?
- **software (bool):** Solver is a software solver?
- **online (bool, default=True):** Is solver online?
- **num_active_qubits (int):** Number of active qubits. Less then or equal to num_qubits.
- **avg_load (float):** Solver’s average load (similar to Unix load average).

Common solver parameters are:

- **flux_biases:** Should solver accept flux biases?
- **anneal_schedule:** Should solver accept anneal schedule?

Common solver properties are:

- **num_qubits (int):** Number of qubits available.
- **vfyc (bool):** Should solver work on “virtual full-yield chip”?
- **max_anneal_schedule_points (int):** Piecewise linear annealing schedule points.
- **h_range ([int,int]), j_range ([int,int]):** Biases/couplings values range.
- **num_reads_range ([int,int]):** Range of allowed values for num_reads parameter.

**Returns** List of all solvers that satisfy the conditions.

**Return type** list[Solver]
Note: Client subclasses (e.g. `dwave.cloud.qpu.Client` or `dwave.cloud.hybrid.Client`) already filter solvers by resource type, so for `qpu` and `hybrid` filters to have effect, call `get_solvers()` on base `Client` class.

Examples:

```python
client.get_solvers(
    num_qubits__gt=2000,  # we need more than 2000 qubits
    num_qubits__lt=4000,  # ... but fewer than 4000 qubits
    num_qubits__within=(2000, 4000),  # an alternative to the previous two lines
    num_active_qubits=1089,  # we want a particular number of active qubits
    vfyc=True,  # we require a fully yielded Chimera
    vfyc__in=[False, None],  # inverse of the previous filter
    vfyc__available=False,  # we want solvers that do not advertise
    anneal_schedule=True,  # we need support for custom anneal schedule
    max_anneal_schedule_points__gte=4,  # we need at least 4 points for our anneal schedule
    num_reads_range__covers=1000,  # our solver must support returning 1000 reads
    extended_j_range__covers=[-2, 2],  # we need extended J range to contain
    couplers__contains=[0, 128],  # coupler (edge between) qubits (0,128)
    qubits__issuperset={0, 4, 215},  # qubits 0, 4 and 215 must exist
    supported_problem_types__issubset={'ising', 'qubo'},  # require Ising, QUBO or both to be supported
    name='DW_2000Q_5',  # full solver name/ID match
    name__regex='.*2000.*',  # partial/regex-based solver name match
    chip_id__regex='DW_.*',  # chip ID prefix must be DW
    topology__type_eq="chimera",  # topology.type must be chimera
    topology__type="chimera",  # same as above, 'eq' implied even for nested properties
)
```

dwave.cloud.client.Client.is_solver_handled

```python
static Client.is_solver_handled(solver)
```

Determine if the specified solver should be handled by this client.

Default implementation accepts all solvers (always returns True). Override this predicate function with a subclass if you want to specialize your client for a particular type of solvers.

Examples

This function accepts only solvers named “My_Solver_*”.
@staticmethod
def is_solver_handled(solver):
    return solver and solver.id.startswith('My_Solver_')

dwave.cloud.client.Client.close

Client.close()
    Perform a clean shutdown.
    Waits for all the currently scheduled work to finish, kills the workers, and closes the connection pool.

    **Note:** Ensure your code does not submit new work while the connection is closing.

Where possible, it is recommended you use a context manager (``with Client.from_config(...) as construct``) to ensure your code properly closes all resources.

**Examples**

This example creates a client (based on an auto-detected configuration file), executes some code (represented by a placeholder comment), and then closes the client.

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config()  # doctest: +SKIP
>>> # code that uses client
>>> client.close()  # doctest: +SKIP
```

**Specialized Clients**

Typically you use the **Client** class. By default, it instantiates a QPU client. You can also instantiate a QPU or CPU/GPU client directly.

**QPU Client**

An implementation of the REST client for D-Wave Solver API (SAPI) service.

SAPI servers provide authentication, queuing, and scheduling services, and provide a network interface to **solvers**. This API enables you submit a binary quadratic (**Ising** or **QUBO**) model and receive samples from a distribution over the model as defined by a selected solver.

SAPI server workflow is roughly as follows:

1. Submitted problems enter an input queue. Each user has an input queue per solver.
2. Drawing from all input queues for a solver, problems are scheduled.
3. Results are cached for retrieval by the client.
Class

class Client(endpoint=None, token=None, solver=None, proxy=None, permissive_ssl=False, request_timeout=60, polling_timeout=None, connection_close=False, headers=None, **kwargs)
D-Wave Solver API client specialized to work only with QPU solvers.

This class can be instantiated explicitly, or via (base) Client’s factory method, from_config() by supplying "qpu" for client.

Examples

This example explicitly instantiates a dwave.cloud.qpu.Client.get_solver() is guaranteed to return a QPU solver.

```python
from dwave.cloud.qpu import Client
with Client(token='...') as client:
    solver = client.get_solver()
    response = solver.sample_ising(...)
```

The following example instantiates a QPU client indirectly. Again, get_solver() / get_solvers() are guaranteed to return only QPU solver(s).

```python
from dwave.cloud import Client
with Client.from_config(client='qpu') as client:
    solver = client.get_solver()
    response = solver.sample_ising(...)
```

Methods

qpu.Client.is_solver_handled(solver) Determine if the specified solver should be handled by this client.

dwave.cloud.qpu.Client.is_solver_handled

static Client.is_solver_handled(solver) Determine if the specified solver should be handled by this client.

This predicate function (used from the base class) allows only remote QPU solvers.

Software-Samplers Client

Class

class Client(endpoint=None, token=None, solver=None, proxy=None, permissive_ssl=False, request_timeout=60, polling_timeout=None, connection_close=False, headers=None, **kwargs)
D-Wave Solver API client specialized to work only with remote software solvers.
This class can be instantiated explicitly, or via (base) Client’s factory method, `from_config()` by supplying "sw" for `client`.

**Examples**

This example explicitly instantiates a `dwave.cloud.sw.Client.get_solver()` is guaranteed to return a software solver.

```python
from dwave.cloud.sw import Client

with Client(token='...') as client:
    solver = client.get_solver()
    response = solver.sample_ising(...)
```

The following example instantiates a software-solver-only client indirectly. Again, `get_solver()`/`get_solvers()` are guaranteed to return only software solver(s).

```python
from dwave.cloud import Client

with Client.from_config(client='sw') as client:
    solver = client.get_solver()
    response = solver.sample_ising(...)
```

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sw.Client.is_solver_handled(solver)</code></td>
<td>Determine if the specified solver should be handled by this client.</td>
</tr>
</tbody>
</table>

**Solve**

A `solver` is a resource for solving problems.

Solvers are responsible for:

- Encoding submitted problems
- Checking submitted parameters
- Decoding answers
- Adding problems to a client’s submission queue

You can list all solvers available to a `Client` with its `get_solvers()` method and select and return one with its `get_solver()` method.
Class

Solver

alias of `dwave.cloud.solver.StructuredSolver`

class BaseSolver(client, data)

Base class for a general D-Wave solver.

This class provides Ising, QUBO and BQM sampling methods and encapsulates the solver description returned from the D-Wave cloud API.

Parameters

- `client (Client)` – Client that manages access to this solver.
- `data (dict)` – Data from the server describing this solver.

Examples

This example creates a client using the local system’s default D-Wave Cloud Client configuration file and checks the identity of its default solver.

```python
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:
...     solver = client.get_solver()
...     solver.id  # doctest: +SKIP
'EXAMPLE_2000Q_SYSTEM'
```

class StructuredSolver(*args, **kwargs)

class for D-Wave structured solvers.

This class provides Ising, QUBO and BQM sampling methods and encapsulates the solver description returned from the D-Wave cloud API.

Parameters

- `client (Client)` – Client that manages access to this solver.
- `data (dict)` – Data from the server describing this solver.

class UnstructuredSolver(client, data)

Class for D-Wave unstructured solvers.

This class provides Ising, QUBO and BQM sampling methods and encapsulates the solver description returned from the D-Wave cloud API.

Parameters

- `client (Client)` – Client that manages access to this solver.
- `data (dict)` – Data from the server describing this solver.

Note: Events are not yet dispatched from unstructured solvers.

Methods
StructuredSolver.check_problem(linear, quadratic)
Test if an Ising model matches the graph provided by the solver.

Parameters
- **linear** (list/dict) – Linear terms of the model (h).
- **quadratic** (dict[(int, int), float]) – Quadratic terms of the model (J).

Returns boolean

Examples

This example creates a client using the local system’s default D-Wave Cloud Client configuration file, which is configured to access a D-Wave 2000Q QPU, and tests a simple Ising model for two target embeddings (that is, representations of the model’s graph by coupled qubits on the QPU’s sparsely connected graph), where only the second is valid.

```python
generate a skip function
from dwave.cloud import Client
>>> print((0, 1) in solver.edges) # doctest: +SKIP
False
>>> print((0, 4) in solver.edges) # doctest: +SKIP
True
>>> with Client.from_config() as client: # doctest: +SKIP
...    solver = client.get_solver()
...    print(solver.check_problem({0: -1, 1: 1}, {(0, 1):0.5}))
False
>>> print(solver.check_problem({0: -1, 4: 1}, {(0, 4):0.5}))
True
```

dwave.cloud.solver.StructuredSolver.sample_ising

StructuredSolver.sample_ising(linear, quadratic, **params)
Sample from the specified Ising model.

Parameters

4.3. dwave-cloud-client 360
• **linear** (*list/dict*) – Linear terms of the model (h).

• **quadratic** (*dict[(int, int), float]*) – Quadratic terms of the model (J), stored in a dict. With keys that are 2-tuples of variables and values are quadratic biases associated with the pair of variables (the interaction).

• **params** – Parameters for the sampling method, solver-specific.

**Returns** Future

**Examples**

This example creates a client using the local system’s default D-Wave Cloud Client configuration file, which is configured to access a D-Wave 2000Q QPU, submits a simple *Ising* problem (opposite linear biases on two coupled qubits), and samples 5 times.

```python
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:
...    solver = client.get_solver()
...    u, v = next(iter(solver.edges))
...    computation = solver.sample_ising({u: -1, v: 1}, {}, num_reads=5)  # doctest: +SKIP
...    for i in range(5):
...        print(computation.samples[i][u], computation.samples[i][v])
...    ...
(1, -1)
(1, -1)
(1, -1)
(1, -1)
(1, -1)
```

**dwave.cloud.solver.StructuredSolver.sample_qubo**

**StructuredSolver**. **sample_qubo**(qubo, **params**)  
Sample from the specified *QUBO*.

**Parameters**

• **qubo** (*dict[(int, int), float]*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) model.

• **params** – Parameters for the sampling method, solver-specific.

**Returns** Future

**Examples**

This example creates a client using the local system’s default D-Wave Cloud Client configuration file, which is configured to access a D-Wave 2000Q QPU, submits a *QUBO* problem (a Boolean NOT gate represented by a penalty model), and samples 5 times.

```python
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:  # doctest: +SKIP
...    solver = client.get_solver()
...    u, v = next(iter(solver.edges))
...    ...
```
... Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
... computation = solver.sample_qubo(Q, num_reads=5)
... for i in range(5):
...     print(computation.samples[i][u], computation.samples[i][v])
...
...
(0, 1)
(1, 0)
(1, 0)
(0, 1)
(1, 0)

**dwave.cloud.solver.StructuredSolver.max_num_reads**

StructuredSolver.max_num_reads(**params**)

Returns the maximum number of reads for the given solver parameters.

**Parameters**

**params** – Parameters for the sampling method. Relevant to num_reads:

- annealing_time
- readout_thermalization
- num_reads
- programming_thermalization

**Returns**

The maximum number of reads.

**Return type**

**int**

**dwave.cloud.solver.UnstructuredSolver.sample_ising**

UnstructuredSolver.sample_ising(linear, quadratic, **params**)

Sample from the specified BQM.

**Parameters**

- **bqm** (*BinaryQuadraticModel*) – A binary quadratic model.
- **params** – Parameters for the sampling method, solver-specific.

**Returns**

**Future**

**Note:** To use this method, dimod package has to be installed.

**dwave.cloud.solver.UnstructuredSolver.sample_qubo**

UnstructuredSolver.sample_qubo(qubo, **params**)

Sample from the specified QUBO.

**Parameters**

- **qubo** (*dict[(int, int), float]*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) model.
• **params – Parameters for the sampling method, solver-specific.

Returns Future

Note: To use this method, dimod package has to be installed.

dwave.cloud.solver.UnstructuredSolver.sample_bqm

UnstructuredSolversample_bqm(bqm, **params)
Sample from the specified BQM.

Parameters
• bqm(BinaryQuadraticModel/str) – A binary quadratic model, or a reference to one
  (Problem ID returned by upload_bqm method).
• **params – Parameters for the sampling method, solver-specific.

Returns Future

Note: To use this method, dimod package has to be installed.

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseSolver.name</td>
<td>Is this solver online (or offline)?</td>
</tr>
<tr>
<td>BaseSolver.online</td>
<td>Is this a QPU-based solver?</td>
</tr>
<tr>
<td>BaseSolver.avg_load</td>
<td>Solver’s average load, at the time of description fetch.</td>
</tr>
<tr>
<td>BaseSolver.qpu</td>
<td>Is this a software-based solver?</td>
</tr>
<tr>
<td>StructuredSolver.num_active_qubits</td>
<td>The number of active (encoding) qubits.</td>
</tr>
<tr>
<td>StructuredSolver.num_qubits</td>
<td>Nominal number of qubits on chip (includes active AND inactive).</td>
</tr>
<tr>
<td>StructuredSolver.is_vfyc</td>
<td>Is this a virtual full-yield chip?</td>
</tr>
<tr>
<td>StructuredSolver.has_flux_biases</td>
<td>Solver supports/accepts flux_biases.</td>
</tr>
<tr>
<td>StructuredSolver.has_anneal_schedule</td>
<td>Solver supports/accepts anneal_schedule.</td>
</tr>
</tbody>
</table>

362 Chapter 6. Packages
**dwave.cloud.solver.BaseSolver.avg_load**

BaseSolver.**avg_load**  
Solver’s average load, at the time of description fetch.

**dwave.cloud.solver.BaseSolver.qpu**

BaseSolver.**qpu**  
Is this a QPU-based solver?

**dwave.cloud.solver.BaseSolver.software**

BaseSolver.**software**  
Is this a software-based solver?

**dwave.cloud.solver.StructuredSolver.num_active_qubits**

StructuredSolver.**num_active_qubits**  
The number of active (encoding) qubits.

**dwave.cloud.solver.StructuredSolver.num_qubits**

StructuredSolver.**num_qubits**  
Nominal number of qubits on chip (includes active AND inactive).

**dwave.cloud.solver.StructuredSolver.is_vfyc**

StructuredSolver.**is_vfyc**  
Is this a virtual full-yield chip?

**dwave.cloud.solver.StructuredSolver.has_flux_biases**

StructuredSolver.**has_flux_biases**  
Solver supports/accepts flux_biases.

**dwave.cloud.solver.StructuredSolver.has_anneal_schedule**

StructuredSolver.**has_anneal_schedule**  
Solver supports/accepts anneal_schedule.

**dwave.cloud.solver.StructuredSolver.lower_noise**

StructuredSolver.**lower_noise**
Computation

Computation manages the interactions between your code and a solver, which manages interactions between the remote resource and your submitted problems.

Your solver instantiates a `Future` object for its calls, via D-Wave Sampler API (SAPI) servers, to the remote resource.

You can interact through the `Future` object with pending (running) or completed computation—sampling on a QPU or software solver—executed remotely, monitoring problem status, waiting for and retrieving results, cancelling enqueued jobs, etc.

Some `Future` methods are blocking.

Class

```python
class Future(solver, id_, return_matrix=False)
```

Class for interacting with jobs submitted to SAPI.

*Solver* uses `Future` to construct objects for pending SAPI calls that can wait for requests to complete and parse returned messages.

Objects are blocked for the duration of any data accessed on the remote resource.

**Warning:** `Future` objects are not intended to be directly created. Problem submittal is initiated by one of the solvers in `solver` module and executed by one of the clients.

**Parameters**

- `solver (Solver)` – Solver responsible for this `Future` object.
- `id (str, optional, default=None)` – Identification for a query submitted by a solver to SAPI. May be None following submission until an identification number is set.
- `return_matrix (bool, optional, default=False)` – Return values for this `Future` object are NumPy matrices.

**Examples**

This example creates a solver using the local system’s default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource for 100 samples, and checks a couple of times whether the sampling is completed.

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config()  # doctest: +SKIP
>>> solver = client.get_solver()  # doctest: +SKIP
>>> u, v = next(iter(solver.edges))  # doctest: +SKIP
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}  # doctest: +SKIP
>>> computation = solver.sample_qubo(Q, num_reads=100)  # doctest: +SKIP
>>> computation.done()  # doctest: +SKIP
False
>>> computation.id  # doctest: +SKIP
'1cefeb6d-ebd5-4592-87c0-4cc43ec03e27'
>>> computation.done()  # doctest: +SKIP
True
>>> client.close()  # doctest: +SKIP
```
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future.result()</td>
<td>Results for a submitted job.</td>
</tr>
<tr>
<td>Future.as_completed(fs[, timeout])</td>
<td>Yield Futures objects as they complete.</td>
</tr>
<tr>
<td>Future.wait([timeout])</td>
<td>Wait for the solver to receive a response for a submitted problem.</td>
</tr>
<tr>
<td>Future.wait_multiple(futures[, min_done, ...])</td>
<td>Wait for multiple Future objects to complete.</td>
</tr>
<tr>
<td>Future.done()</td>
<td>Check whether the solver received a response for a submitted problem.</td>
</tr>
<tr>
<td>Future.cancel()</td>
<td>Try to cancel the problem corresponding to this result.</td>
</tr>
</tbody>
</table>

**dwave.cloud.computation.Future.result**

*Future.result()*

Results for a submitted job.

Retrieves raw result data in a Future object that the solver submitted to a remote resource. First calls to access this data are blocking.

**Returns**

Results of the submitted job. Should be considered read-only.

**Return type**

dict

**Note:** Helper properties on Future object are preferred to reading raw results, as they abstract away the differences in response between some solvers like. Available methods are: samples(), energies(), occurrences(), variables(), timing(), problem_type(), sampleset() (only if dimod package is installed).

**Warning:** The dictionary returned by result() depends on the solver used. Starting with version 0.7.0 we will not try to standardize them anymore, on client side. For QPU solvers, please replace ‘samples’ with ‘solutions’ and ‘occurrences’ with ‘num_occurrences’. Better yet, use Future.samples() and Future.occurrences() instead.

**Examples**

This example creates a solver using the local system’s default D-Wave Cloud Client configuration file, submits a simple QUBO problem (representing a Boolean NOT gate by a penalty function) to a remote D-Wave resource for 5 samples, and prints part of the returned result (the relevant samples).

```python
>>> from dwave.cloud import Client
>>> with Client.from_config() as client: # doctest: +SKIP
...     solver = client.get_solver()
...     u, v = next(iter(solver.edges))
...     Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
...     computation = solver.sample_qubo(Q, num_reads=5)
...     for i in range(5):
...         result = computation.result()
...         print(result['solutions'][i][u], result['solutions'][i][v])
```

(continues on next page)
dwave.cloud.computation.Future.as_completed

`static Future.as_completed(fs, timeout=None)`

Yield Futures objects as they complete.

Returns an iterator over the specified list of `Future` objects that yields those objects as they complete. Completion occurs when the submitted job is finished or cancelled.

Emulates the behavior of the `concurrent.futures.as_completed()` function.

**Parameters**

- `fs` *(list)* — List of `Future` objects to iterate over.
- `timeout` *(float, optional, default=None)* — Maximum number of seconds to await completion. If None, awaits indefinitely.

**Returns**

Listed `Future` objects as they complete.

**Return type**

Generator (`Future` objects)

**Raises**

- `concurrent.futures.TimeoutError` is raised if per-future timeout is exceeded.

**Examples**

This example creates a solver using the local system’s default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource 3 times for differing numbers of samples, and yields timing information for each job as it completes.

```python
>>> import dwave.cloud as dc
>>> client = dc.Client.from_config()  # doctest: +SKIP
>>> solver = client.get_solver()  # doctest: +SKIP
>>> u, v = next(iter(solver.edges))  # doctest: +SKIP
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}  # doctest: +SKIP
>>> computation = [solver.sample_qubo(Q, num_reads=1000),
...                 solver.sample_qubo(Q, num_reads=50),
...                 solver.sample_qubo(Q, num_reads=10)]  # doctest: +SKIP
>>> for tasks in dc.computation.Future.as_completed(computation, timeout=10)
...     print(tasks.timing)  # doctest: +SKIP
...     {'total_real_time': 17318, ... 'qpu_readout_time_per_sample': 123}
{'total_real_time': 10816, ... 'qpu_readout_time_per_sample': 123}
{'total_real_time': 26285, ... 'qpu_readout_time_per_sample': 123}
... >>> client.close()  # doctest: +SKIP
```
Future.wait (timeout=None)

Wait for the solver to receive a response for a submitted problem.

Blocking call that waits for a Future object to complete.

Parameters timeout (float, optional, default=None) – Maximum number of seconds to await completion. If None, waits indefinitely.

Returns True if solver received a response.

Return type Boolean

Examples

This example creates a solver using the local system’s default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource for 100 samples, and tries waiting for 10 seconds for sampling to complete.

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config()  # doctest: +SKIP
>>> solver = client.get_solver()  # doctest: +SKIP
>>> u, v = next(iter(solver.edges))  # doctest: +SKIP
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}  # doctest: +SKIP
>>> computation = solver.sample_qubo(Q, num_reads=100)  # doctest: +SKIP
>>> computation.wait(timeout=10)  # doctest: +SKIP
False
>>> computation.remote_status  # doctest: +SKIP
'IN_PROGRESS'
>>> computation.wait(timeout=10)  # doctest: +SKIP
True
>>> computation.remote_status  # doctest: +SKIP
'COMPLETED'
>>> client.close()  # doctest: +SKIP
```

dwave.cloud.computation.Future.wait_multiple

static Future.wait_multiple (futures, min_done=None, timeout=None)

Wait for multiple Future objects to complete.

Blocking call that uses an event object to emulate multi-wait for Python.

Parameters

- futures (list of Futures) – List of Future objects to await.
- min_done (int, optional, default=None) – Minimum required completions to end the waiting. The wait is terminated when this number of results are ready. If None, waits for all the Future objects to complete.
- timeout (float, optional, default=None) – Maximum number of seconds to await completion. If None, waits indefinitely.

Returns completed and not completed submitted tasks. Similar to concurrent.futures.wait() method’s returned two-tuple of done and not_done sets.

Return type Two-tuple of Future objects
See also:

```
as_completed() for a blocking iterable of resolved futures similar to concurrent.futures.as_completed() method.
```

Examples

This example creates a solver using the local system’s default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource 3 times for differing numbers of samples, and waits for sampling to complete on any two of the submissions. The wait ends with the completion of two submissions while the third is still in progress. (A more typical approach would use something like first = next(Future.
```

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config()  # doctest: +SKIP
>>> solver = client.get_solver()  # doctest: +SKIP
>>> u, v = next(iter(solver.edges))  # doctest: +SKIP
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}  # doctest: +SKIP
>>> computation = [solver.sample_qubo(Q, num_reads=1000), ...
... solver.sample_qubo(Q, num_reads=50), ...
... solver.sample_qubo(Q, num_reads=10)]  # doctest: +SKIP
>>> dc.computation.Future.wait_multiple(computation, min_done=1)  # doctest: +SKIP
[(<dwave.cloud.computation.Future at 0x17dde518>,
  <dwave.cloud.computation.Future at 0x17ddee80>],
 [<dwave.cloud.computation.Future at 0x15078080>])
```

```python
>>> print(computation[0].done())  # doctest: +SKIP
False
>>> print(computation[1].done())  # doctest: +SKIP
True
>>> print(computation[2].done())  # doctest: +SKIP
True
```
dwave.cloud.computation.Future.done

```
Future.done()
```

Check whether the solver received a response for a submitted problem.

Non-blocking call that checks whether the solver has received a response from the remote resource.

Returns True if solver received a response.

Return type Boolean

Examples

This example creates a solver using the local system’s default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource for 100 samples, and checks a couple of times whether sampling is completed.

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config()  # doctest: +SKIP
>>> solver = client.get_solver()  # doctest: +SKIP
>>> u, v = next(iter(solver.edges))  # doctest: +SKIP
```
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}  # doctest: +SKIP
>>> computation = solver.sample_qubo(Q, num_reads=100)  # doctest: +SKIP
>>> computation.done()  # doctest: +SKIP
False
>>> computation.done()  # doctest: +SKIP
True
>>> client.close()  # doctest: +SKIP

**dwave.cloud.computation.Future.cancel**

**Future.cancel()**

Try to cancel the problem corresponding to this result.

Non-blocking call to the remote resource in a best-effort attempt to prevent execution of a problem.

**Examples**

This example creates a solver using the local system’s default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource for 100 samples, and tries (and in this case succeeds) to cancel it.

```python
>>> from dwave.cloud import Client
>>> client = Client.from_config()  # doctest: +SKIP
>>> solver = client.get_solver()  # doctest: +SKIP
>>> u, v = next(iter(solver.edges))  # doctest: +SKIP
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}  # doctest: +SKIP
>>> computation = solver.sample_qubo(Q, num_reads=100)  # doctest: +SKIP
>>> computation.cancel()  # doctest: +SKIP
>>> computation.done()  # doctest: +SKIP
True
>>> computation.remote_status  # doctest: +SKIP
'CANCELLED'
>>> client.close()  # doctest: +SKIP
```

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future.samples</td>
<td>State buffer for the submitted job.</td>
</tr>
<tr>
<td>Future.variables</td>
<td>List of active variables in response/answer.</td>
</tr>
<tr>
<td>Future.energies</td>
<td>Energy buffer for the submitted job.</td>
</tr>
<tr>
<td>Future.occurrences</td>
<td>Occurrences buffer for the submitted job.</td>
</tr>
<tr>
<td>Future.sampleset</td>
<td>Return SampleSet representation of the results.</td>
</tr>
<tr>
<td>Future.problem_type</td>
<td>Submitted problem type for this computation, as returned by the solver API.</td>
</tr>
<tr>
<td>Future.timing</td>
<td>Timing information about a solver operation.</td>
</tr>
</tbody>
</table>

**dwave.cloud.computation.Future.samples**

**Future.samples**

State buffer for the submitted job.
First calls to access data of a `Future` object are blocking; subsequent access to this property is non-blocking.

**Returns**  
Samples on the nodes of solver’s graph.

**Return type**  
list of lists or NumPy matrix

### Examples

This example creates a solver using the local system’s default D-Wave Cloud Client configuration file, submits a simple QUBO problem (representing a Boolean NOT gate by a penalty function) to a remote D-Wave resource for 5 samples, and prints part of the returned result (the relevant samples).

```python
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()
...     u, v = next(iter(solver.edges))
...     Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
...     computation = solver.sample_qubo(Q, num_reads=5)
...     for i in range(5):
...         print(computation.samples[i][u], computation.samples[i][v])
...
...
(1, 0)
(0, 1)
(0, 1)
(1, 0)
(0, 1)
```

**`dwave.cloud.computation.Future.variables`**

**Future.**variables  
List of active variables in response/answer.

**`dwave.cloud.computation.Future.energies`**

**Future.**energies  
Energy buffer for the submitted job.

First calls to access data of a `Future` object are blocking; subsequent access to this property is non-blocking.

**Returns**  
Energies for each set of samples.

**Return type**  
list or NumPy matrix of doubles

### Examples

This example creates a solver using the local system’s default D-Wave Cloud Client configuration file, submits a random Ising problem (+1 or -1 values of linear and quadratic biases on all nodes and edges, respectively, of the solver’s garph) to a remote D-Wave resource for 10 samples, and prints the returned energies.

```python
>>> import random
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()  # doctest: +SKIP
```
linear = {index: random.choice([-1, 1]) for index in solver.nodes}
quad = {key: random.choice([-1, 1]) for key in solver.undirected_edges}
computation = solver.sample_ising(linear, quad, num_reads=10)
print(computation.energies)

[-3976.0, -3974.0, -3972.0, -3970.0, -3968.0, -3968.0, -3966.0, -3964.0, -3964.0, -3960.0]

**Examples**

This example creates a solver using the local system’s default D-Wave Cloud Client configuration file, submits a simple Ising problem with several ground states to a remote D-Wave resource for 20 samples, and prints the returned results, which are ordered as a histogram. The problem’s ground states tend to recur frequently, and so those solutions have *occurrences* greater than 1.

```python
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()
...     computation = solver.sample_ising({}, quad, num_reads=500, answer_mode='histogram')
...     for i in range(len(computation.occurrences)):
...         print(computation.samples[i][16], computation.samples[i][17], ' --> ', computation.samples[i][20], computation.samples[i][21], ' --> ', computation.energies[i], computation.occurrences[i])
...
(-1, 1, -1, -1, ' --> ', -2.0, 41)
(-1, -1, -1, ' --> ', -2.0, 53)
(1, -1, 1, ' --> ', -2.0, 55)
(1, 1, -1, ' --> ', -2.0, 52)
(1, 1, -1, ' --> ', -2.0, 60)
(1, -1, 1, ' --> ', -2.0, 196)
(-1, -1, 1, ' --> ', -2.0, 15)
(-1, -1, 1, ' --> ', -2.0, 28)
```
dwave.cloud.computation.Future.problem_type

`Future.problem_type`
Submitted problem type for this computation, as returned by the solver API. Typical values are ‘ising’ and ‘qubo’.

dwave.cloud.computation.Future.timing

`Future.timing`
Timing information about a solver operation.

Mapping from string keys to numeric values representing timing details for a submitted job as returned from the remote resource. Keys are dependant on the particular solver.

First calls to access data of a `Future` object are blocking; subsequent access to this property is non-blocking.

**Returns** Mapping from string keys to numeric values representing timing information.

**Return type** dict

**Examples**

This example creates a client using the local system’s default D-Wave Cloud Client configuration file, which is configured to access a D-Wave 2000Q QPU, submits a simple Ising problem (opposite linear biases on two coupled qubits) for 5 samples, and prints timing information for the job.

```python
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:  # doctest: +SKIP
... solver = client.get_solver()
... u, v = next(iter(solver.edges))
... computation = solver.sample_ising({u: -1, v: 1}, {}, num_reads=5)
... print(computation.timing)
... {'total_real_time': 10961, 'anneal_time_per_run': 20, ...}
```

**Exceptions**

**exception ConfigFileError**
Base exception for all config file processing errors.

**exception ConfigFileReadError**
Non-existing or unreadable config file specified or implied.

**exception ConfigFileParseError**
Invalid format of config file.

**exception SolverError**
Generic base class for all solver-related errors.

**exception SolverFailureError**
An exception raised when there is a remote failure calling a solver.

**exception SolverNotFoundError**
Solver with matching feature set not found / not available.

**exception SolverOfflineError**
Action attempted on an offline solver.
exception SolverAuthenticationError
   An exception raised when there is an authentication error.

exception UnsupportedSolverError
   The solver we received from the API is not supported by the client.

exception SolverPropertyMissingError
   The solver we received from the API does not have required properties.

exception Timeout
   General timeout error.

exception RequestTimeout
   REST API request timed out.

exception PollingTimeout
   Problem polling timed out.

exception CanceledFutureError
   An exception raised when code tries to read from a canceled future.

exception InvalidAPIResponseError
   Raised when an invalid/unexpected response from D-Wave Solver API is received.

exception InvalidProblemError
   Solver cannot handle the given binary quadratic model.

exception ProblemUploadError
   Problem multipart upload failed.

6.3.3 Bibliography

6.4 dwave-hybrid

A general, minimal Python framework for building hybrid asynchronous decomposition samplers for quadratic unconstrained binary optimization (QUBO) problems.

dwave-hybrid facilitates three aspects of solution development:

- Hybrid approaches to combining quantum and classical compute resources
- Evaluating a portfolio of algorithmic components and problem-decomposition strategies
- Experimenting with workflow structures and parameters to obtain the best application results

The framework enables rapid development and insight into expected performance of productized versions of its experimental prototypes.

Your optimized algorithmic components and other contributions to this project are welcome!

6.4.1 Introduction

dwave-hybrid provides a framework for iterating arbitrary-sized sets of samples through parallel solvers to find an optimal solution.

For the documentation of a particular code element, see the Reference Documentation section. This introduction gives an overview of the package; steps you through using it, starting with running a provided hybrid solver that handles arbitrary-sized QUBOs; and points out the way to developing your own components in the framework.

- Overview presents the framework and explains key concepts.
Using the Framework shows how to use the framework. You can quickly get started by using a provided reference sampler built with this framework, Kerberos, to solve a problem too large to minor-embed on a D-Wave system. Next, use the framework to build (hybrid) workflows; for example, a solver similar to qbsolv, which can employ tabu search on a whole problem while submitting parts of the problem to a D-Wave system.

Developing New Components guides you to developing your own hybrid components.

Reference Examples describes some workflow examples included in the code.

Overview

The dwave-hybrid framework enables you to quickly design and test workflows that iterate sets of samples through samplers to solve arbitrary QUBOs. Large problems can be decomposed and two or more solution techniques can run in parallel.

The Schematic Representation figure below shows an example configuration. Samples are iterated over four parallel solvers. The top branch represents a classical tabu search that runs on the entire problem until interrupted by another branch completing. These use different decomposers to parcel out parts of the current sample set (iteration i) to samplers such as a D-Wave system (second-highest branch) or another structure of parallel simulated annealing and tabu search. A generic representation of a branch’s components—decomposer, sampler, and composer—is shown in the lowest branch. A user-defined criterion selects from current samples and solver outputs a sample set for iteration i + 1.

Fig. 7: Schematic Representation

You can use the framework to run a provided hybrid solver or to configure workflows using provided components such as tabu samplers and energy-based decomposers.

You can also use the framework to build your own components to incorporate into your workflow.

Using the Framework

This section helps you quickly use a provided reference sampler to solve arbitrary-sized problems and then shows you how to build (hybrid) workflows using provided components.

Reference Hybrid Sampler: Kerberos

dwave-hybrid includes a reference example sampler built using the framework: Kerberos is a dimod-compatible hybrid asynchronous decomposition sampler that enables you to solve problems of arbitrary structure and size. It finds
best samples by running in parallel tabu search, simulated annealing, and D-Wave subproblem sampling on problem variables that have high-energy impact.

The example below uses Kerberos to solve a large QUBO.

```python
>>> import dimod
>>> from hybrid.reference.kerberos import KerberosSampler
>>> with open('../problems/random-chimera/8192.01.qubo') as problem:  # doctest: +SKIP
...   bqm = dimod.BinaryQuadraticModel.from_coo(problem)
>>> len(bqm)  # doctest: +SKIP
8192
>>> solution = KerberosSampler().sample(bqm, max_iter=10, convergence=3)  # doctest: +SKIP
>>> solution.first.energy  # doctest: +SKIP
-4647.0
```

Building Workflows

As shown in the *Overview* section, you build hybrid solvers by arranging components such as samplers in a workflow.

### Building Blocks

The basic components—building blocks—you use are based on the *Runnable* class: decomposers, samplers, and composers. Such components input a set of samples, a *SampleSet*, and output updated samples. A *State* associated with such an iteration of a component holds the problem, samples, and optionally additional information.

The following example demonstrates a simple workflow that uses just one *Runnable*, a sampler representing the classical tabu search algorithm, to solve a problem (fully classically, without decomposition). The example solves a small problem of a triangle graph of nodes identically coupled. An initial *State* of all-zero samples is set as a starting point. The solution, *new_state*, is derived from a single iteration of the *TabuProblemSampler Runnable*.

```python
>>> import dimod
>>> # Define a problem
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {'ab': 0.5, 'bc': 0.5, 'ca': 0.5})
>>> # Set up the sampler with an initial state
>>> sampler = TabuProblemSampler(tenure=2, timeout=5)
>>> state = State.from_sample({'a': 0, 'b': 0, 'c': 0}, bqm)
>>> # Sample the problem
>>> new_state = sampler.run(state).result()
>>> print(new_state.samples)  # doctest: +SKIP
a b c energy num_occ.
0 +1 -1 -1 -0.5 1
['SPIN', 1 rows, 1 samples, 3 variables]
```

Flow Structuring

The framework provides classes for structuring workflows that use the “building-block” components. As shown in the *Overview* section, you can create a *branch* of *Runnable* classes: for example decomposer | sampler | composer, which delegates part of a problem to a sampler such as the D-Wave system.

The following example shows a branch comprising a decomposer, local Tabu solver, and a composer. A 10-variable binary quadratic model is decomposed by the energy impact of its variables into a 6-variable subproblem to be sampled twice. An initial state of all -1 values is set using the utility function *min_sample()*.
```python
>>> import dimod
>>> bqm = dimod.BinaryQuadraticModel({t: 0 for t in range(10)},
... (t, (t+1) % 10): 1 for t in range(10)},
... 0, 'SPIN')
>>> branch = (EnergyImpactDecomposer(size=6, min_gain=-10) |
... TabuSubproblemSampler(num_reads=2) |
... SplatComposer())
>>> new_state = branch.next(State.from_sample(min_sample(bqm), bqm))
>>> print(new_state.subsamples)  # doctest: +SKIP
4 5 6 7 8 9 energy num_occ.
0 +1 -1 -1 +1 -1 +1 -5.0 1
1 +1 -1 -1 +1 -1 +1 -5.0 1
['SPIN', 2 rows, 2 samples, 6 variables]
```

Such Branch classes can be run in parallel using the RacingBranches class. From the outputs of these parallel branches, ArgMin selects a new current sample. And instead of a single iteration on the sample set, you can use the Loop to iterate a set number of times or until a convergence criteria is met.

This example of Racing Branches solves a binary quadratic model by iteratively producing best samples. Similar to qbsolv, it employs both tabu search on the entire problem and a D-Wave system on subproblems. In addition to building-block components such as employed above, this example also uses infrastructure classes to manage the decomposition and parallel running of branches.

Fig. 8: Racing Branches

```python
import dimod
import hybrid

# Construct a problem
bqm = dimod.BinaryQuadraticModel({}, {'ab': 1, 'bc': -1, 'ca': 1}, 0, dimod.SPIN)

# Define the workflow
iteration = hybrid.RacingBranches(
    hybrid.InterruptableTabuSampler(),
    hybrid.EnergyImpactDecomposer(size=2)
    | hybrid.OPUSubproblemAutoEmbeddingSampler()
    | hybrid.SplatComposer()
) |
hybrid.ArgMin()
workflow = hybrid.LoopUntilNoImprovement(iteration, convergence=3)

# Solve the problem
```
init_state = hybrid.State.from_problem(bqm)
final_state = workflow.run(init_state).result()

# Print results
print("Solution: sample={.samples.first}".format(final_state))

Flow Refining

The framework enables quick modification of work flows to improve solutions and performance. For example, after verifying the Racing Branches workflow above on its small problem, you might make a series of modifications such as the examples below to better fit it to problems with large numbers of variables.

1. Configure a decomposition window that moves down a fraction of problem variables, ordered from highest to lower energy impact, and submit those subproblems to the D-Wave system while tabu searches globally. This example submits 50-variable subproblems on up to 15% of the total variables.

```python
# Redefine the workflow: a rolling decomposition window
subproblem = hybrid.EnergyImpactDecomposer(size=50, rolling_history=0.15)
subsampler = hybrid.QPUProblemAutoEmbeddingSampler() | hybrid.SplatComposer()
iteration = hybrid.RacingBranches(hybrid.InterruptableTabuSampler(), subproblem | subsampler) | hybrid.ArgMin()
workflow = hybrid.LoopUntilNoImprovement(iteration, convergence=3)
```

2. Instead of sequentially producing a sample per subproblem, a further modification might be to process all the subproblems in parallel and merge the returned samples. Here the EnergyImpactDecomposer is iterated until it raises a EndOfStream() exception when it reaches 15% of the variables, and then all the 50-variable subproblems are submitted to the D-Wave system.

```python
# Redefine the workflow: parallel subproblem solving for a single sample
subproblem = hybrid.Unwind(hybrid.EnergyImpactDecomposer(size=50, rolling_history=0.15))
subsampler = hybrid.Map(hybrid.QPUProblemAutoEmbeddingSampler()) | hybrid.Reduce(hybrid.Lambda(merge_substates)) | hybrid.SplatComposer()
```

3. Change the criterion for selecting subproblems. By default, the variables are selected by maximal energy impact but selection can be better tailored to a problem’s structure.

   For example, for binary quadratic model representing the problem graph shown in the Traversal by Energy Impact graphic, if you select a subproblem size of four, these nodes selected by descending energy impact are not directly connected (no shared edges, and might not represent a local structure of the problem).

Additional Examples
Configuring a mode of traversal such as breadth-first (BFS) or priority-first selection (PFS) can capture features that represent local structures within a problem.

```python
# Redefine the workflow: subproblem selection
subproblem = hybrid.Unwind(
    hybrid.EnergyImpactDecomposer(size=50, rolling_history=0.15, traversal='bfs'))
```

These two selection modes are shown in the **Traversal by BFS or PFS** graphic. BFS starts with the node with maximal energy impact, from which its graph traversal proceeds to directly connected nodes, then nodes directly connected to those, and so on, with graph traversal ordered by node index. In PFS, graph traversal selects the node with highest energy impact among unselected nodes directly connected to any already selected node.
Tailoring State Selection

The next example tailors a state selector for a sampler that does some post-processing and can alert upon suspect samples. Sampler output modified by ellipses ("...") for readability is shown below for an Ising model of a triangle problem with zero biases and interactions all equal to 0.5. The first of three `State` classes is flagged as problematic using the `info` field:

```python
[...,'samples': SampleSet(rec.array([[0, 1, 0], 0., 1]), ...), ['a', 'b', 'c'], {...
  'Postprocessor': 'Excessive chain breaks'}, 'SPIN')},
[...,'samples': SampleSet(rec.array([[1, 1, 1], 1.5, 1]), ...), ['a', 'b', 'c'], {},
  'SPIN')},
[...,'samples': SampleSet(rec.array([[0, 0, 0], 0., 1]), ...), ['a', 'b', 'c'], {},
  'SPIN')}
```

This code snippet defines a metric for the key argument in `ArgMin`:

```python
def preempt(si):
    if 'Postprocessor' in si.samples.info:
        return (math.inf)
    else:
        return (si.samples.first.energy)
```

Using the defined key on the above input, `ArgMin` finds the state with the lowest energy (zero) excluding the flagged state (which also has energy of zero):

```python
>>> ArgMin(key=preempt).next(states)  # doctest: +SKIP
{'problem': BinaryQuadraticModel({'a': 0.0, 'b': 0.0, 'c': 0.0}, {('a', 'b'): 0.5, ('b
  ->', 'c'): 0.5, ('c', 'a'): 0.5}, 0.0, Vartype.SPIN), 'samples': SampleSet(rec.array([[0, 0, 0], 0., 1]),
  dtype=[('sample', 'i1', (3,)), ('energy', '<f8'), ('num_occurrences', '<i4')], ['a',
  '->' 'b', 'c'], {}, 'SPIN')}
```

Parallel Sampling

The code snippet below uses `Map` to run a tabu search on two states in parallel.

```python
>>> Map(TabuProblemSampler()).run(States(
    State.from_sample({'a': 0, 'b': 0, 'c': 1}, bqm1),
    State.from_sample({'a': 1, 'b': 1, 'c': 0}, bqm2)))
```

```python
[{'samples': SampleSet(rec.array([[-1, -1, 1], -0.5, 1]), dtype=[('sample', 'i1',
  (3,)), ('energy', '<f8'), ('num_occurrences', '<i4')], ['a', 'b', 'c'], {}, 'SPIN'),
  'problem': BinaryQuadraticModel({'a': 0.0, 'b': 0.0, 'c': 0.0}, {('a', 'b'): 0.5, ('b
  ->', 'c'): 0.5, ('c', 'a'): 0.5}, 0.0, Vartype.SPIN)},
{'samples': SampleSet(rec.array([[1, 1, -1], -1., 1]), dtype=[('sample', 'i1', (3,
  ->)), ('energy', '<f8'), ('num_occurrences', '<i4')], ['a', 'b', 'c'], {}, 'SPIN'),
  'problem': BinaryQuadraticModel({'a': 0.0, 'b': 0.0, 'c': 0.0}, {('a', 'b'): 1, ('b
  ->', 'c'): 1, ('c', 'a'): 1}, 0.0, Vartype.SPIN)])
```
Logging and Execution Information

You can see detailed execution information by setting the level of logging.

The package supports logging levels TRACE, DEBUG, INFO, WARNING, ERROR, and CRITICAL in ascending order of severity. By default, logging level is set to ERROR. You can select the logging level with environment variable D Wave_HYBRID_LOG_LEVEL.

For example, on a Windows operating system, set this environment variable to INFO level as:

```
set D WAVE_HYBRID_LOG_LEVEL=INFO
```

or on a Unix-based system as:

```
D WAVE_HYBRID_LOG_LEVEL=INFO
```

The previous example above might output something like the following:

```
>>> print("Solution: sample={(s.samples.first)}".format(s=solution))  # doctest: +SKIP

2018-12-10 15:18:30,634 hybrid.flow INFO Loop Iteration(iterno=0, best_state_quality=-3.0)
2018-12-10 15:18:31,511 hybrid.flow INFO Loop Iteration(iterno=1, best_state_quality=-3.0)
2018-12-10 15:18:35,889 hybrid.flow INFO Loop Iteration(iterno=2, best_state_quality=-3.0)
2018-12-10 15:18:37,377 hybrid.flow INFO Loop Iteration(iterno=3, best_state_quality=-3.0)
Solution: sample=Sample(sample={'a': 1, 'b': -1, 'c': -1}, energy=-3.0, num_occurrences=1)
```

Developing New Components

The d wave-hybrid framework enables you to build your own components to incorporate into your workflow.

The key superclass is the Runnable class: all basic components—samplers, decomposers, composers—and flow-structuring components such as branches inherit from this class. A Runnable is run for an iteration in which it updates the State it receives. Typical methods are run or next to execute an iteration and stop to terminate the Runnable.

The Primitives and Flow Structuring sections describe, respectively, the basic Runnable classes (building blocks) and flow-structuring ones and their methods. If you are implementing these methods for your own Runnable class, see comments in the code.

The Racing Branches graphic below shows the top-down composition (tree structure) of a hybrid loop.

State traits are verified for all Runnable objects that inherit from StateTraits or its subclasses. Verification includes:

1. Minimal checks of workflow construction (composition of Runnable classes)
2. Runtime checks

All built-in Runnable classes declare state traits requirements that are either independent (for simple ones) or derived from a child workflow. Traits of a new Runnable must be expressed and modified at construction time by its parent. When developing new Runnable classes, constructing composite traits can be nontrivial for some advanced flow-control runnables.
Fig. 11: Top-Down Composition

The *Dimod Conversion* section describes the *HybridRunnable* class you can use to produce a *Runnable* sampler based on a *dimod* sampler.

The *Utilities* section provides a list of useful utility methods.

**Reference Examples**

The *examples* directory of the code includes implementations of some *Reference Workflows* you can incorporate into your application and also use to jumpstart your development of custom workflows.

A typical first use of dwave-hybrid might be to simply use the Kerberos reference sampler to solve a QUBO, as shown in *Using the Framework*. Next, you might tune its configurable parameters, described under *Reference Workflows*.

To further improve performance, you can step up from using a generic workflow to one tailored for your application and its problem. As a first step you can modify a reference workflow with existing components. After that, you can implement your own components as described in *Developing New Components*.

### 6.4.2 Reference Documentation

**Primitives**

Basic building-block classes and superclasses for hybrid workflows.

**Classes**

```python
class Present (result=None, exception=None)
    Already resolved Future object.
```

6.4. dwave-hybrid
Users should treat this class as just another Future, the difference being an implementation detail: Present is “resolved” at construction time.

See the example of the run() method.

class Runnable (**runopts)**

Components such as samplers and branches that can be run for an iteration.

Parameters **runopts** (dict) – Keyword arguments passed down to each Runnable.run call.

Note: The base class Runnable does not enforce traits validation. To enable validation, derive your subclass from one of the state structure, I/O dimensionality, or I/O validation mixins in traits.

Examples

This example runs a tabu search on a binary quadratic model. An initial state is manually set to \(x = y = 0, z = 1, a = b = 1, c = 0\) and an updated state is created by running the sampler for one iteration.

```python
>>> import dimod
# Create a binary quadratic model
>>> bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b' →: 0.0, 'c': 6.0},
...   {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y' →): -4.0,
...    ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b' →): -4.0, ('a', 'z'): -4.0},
...   -1.0, 'BINARY')
>>> # Set up the sampler runnable
>>> sampler = TabuProblemSampler(tenure=2, timeout=5)
>>> # Run one iteration of the sampler
>>> new_state = sampler.next(State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b' →: 1, 'c': 0}, bqm))
>>> print(new_state.samples)  # doctest: +SKIP
 a b c x y z energy num_occ.
0 1 1 1 1 1 1 -1.0 1
[ 1 rows, 6 variables ]
```

class State (**args, **kwargs)**

Computation state passed along a branch between connected components.

State is a dict subclass and usually contains at least two keys: samples and problem.

Examples

```python
>>> import dimod
# Create a binary quadratic model
>>> bqm = dimod.BinaryQuadraticModel({0: -1, 1: -1}, {(0, 1): 2}, 0.0, dimod.
˓→BINARY)
>>> hybrid.core.State.from_sample(hybrid.utils.min_sample(bqm), bqm)  # doctest:
˓→+SKIP
{'problem': BinaryQuadraticModel({0: -1, 1: -1}, {0, 1: 2}, 0.0, Vartype.˓→BINARY),
'samples': SampleSet(rec.array([[(0, 0), 0., 1]],
   dtype=[('sample', 'i1', (2,)), ('energy', '<f8'), ('num_occurrences', '<i4' ˓→')]], [0, 1], {}, 'BINARY')}
```
class States(*args)
    List of states.

Properties

Runnable.name
    Return the Runnable class name.

SampleSet.first
    Sample with the lowest-energy.

hybrid.core.Runnable.name

Runnable.name
    Return the Runnable class name.

hybrid.core.SampleSet.first

SampleSet.first
    Sample with the lowest-energy.
    Raises ValueError – If empty.

Example

```python
>>> sampleset = dimod.ExactSolver().sample_ising({'a': 1}, {('a', 'b'): 1})
>>> sampleset.first
Sample(sample={'a': -1, 'b': 1}, energy=-2.0, num_occurrences=1)
```

Methods

Runnable.dispatch(future, **kwargs)
    Dispatch state from resolving future to either next or error methods.

Runnable.error(exc)
    Execute one blocking iteration of an instantiated Runnable with an exception as input.

Runnable.init(state, **runopts)
    Run prior to the first next/run, with the first state received.

Runnable.halt()
    Called by stop().

Runnable.next(state, **runopts)
    Execute one blocking iteration of an instantiated Runnable with a valid state as input.

Runnable.run(state, **kwargs)
    Execute the next step/iteration of an instantiated Runnable.

Runnable.stop()
    Terminate an iteration of an instantiated Runnable.

State.updated(**kwargs)
    Return a (deep) copy of the state, updated from kwargs.

State.from_sample(sample, bqm, **kwargs)
    Convenience method for constructing a state from a raw (dict) sample.

State.from_samples(samples, bqm, **kwargs)
    Convenience method for constructing a state from raw (dict) samples.
hybrid.coreRunnable.dispatch

Runnable.dispatch(future, **kwargs)
Dispatch state from resolving future to either next or error methods.

Parameters:

Returns:
- state from next() or error(), or passes through an exception raised there.

Blocks on state resolution and execution of next() or error().

hybrid.coreRunnable.error

Runnable.error(exc)
Execute one blocking iteration of an instantiated Runnable with an exception as input.
Called when the previous component raised an exception instead of generating a new state.
The default implementation raises again the input exception. Runnable errors must be explicitly silenced.

hybrid.coreRunnable.init

Runnable.init(state, **runopts)
Run prior to the first next/run, with the first state received.
Default to NOP.

hybrid.coreRunnable.halt

Runnable.halt()
Called by stop(). Override this method (instead of stop) to handle stopping of one blocking call of next. Defaults to NOP.

hybrid.coreRunnable.next

Runnable.next(state, **runopts)
Execute one blocking iteration of an instantiated Runnable with a valid state as input.

Parameters:
- state (State) – Computation state passed between connected components.

Returns:
- The new state.

Return type:
- State

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```python
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```
hybrid.coreRunnable.run

Runnable.run(state, **kwargs)

Execute the next step/iteration of an instantiated Runnable.

Accepts a state in a Future-like object and returns a new state in a Future-like object.

Parameters

• state (State) – Computation state future-like object passed between connected components.
• executor (Executor, optional, default=None) – The Executor to which the execution of this block is scheduled. By default hybrid.concurrency.thread_executor is used.

Examples

These two code snippets run one iteration of a sampler to produce a new state. The first is an asynchronous call and the second a blocking call.

```python
>>> sampler.run(State.from_sample(min_sample(bqm), bqm))  # doctest: +SKIP
<Future at 0x20cbe22ea20 state=running>
```

```python
>>> sampler.run(State.from_sample(min_sample(bqm), bqm),
... executor=hybrid.immediate_executor)  # doctest: +SKIP
<Present at 0x20ca68cd2b0 state=finished returned State>
```

hybrid.coreRunnable.stop

Runnable.stop()

Terminate an iteration of an instantiated Runnable.

hybrid.core.State.updated

State.updated(**kwargs)

Return a (deep) copy of the state, updated from kwargs.

This method has dict.update semantics with immutability of sorted. Currently an exception is the debug key, if it exists, for which a depth-unlimited recursive merge is executed.

Example

```python
>>> state = State()
>>> state
{}
>>> newstate = state.updated(problem="test")
>>> newstate
{'problem': 'test'}
```
hybrid.core.State.from_sample

**classmethod** State.from_sample(sample, bqm, **kwargs)

Convenience method for constructing a state from a raw (dict) sample.

Energy is calculated from the binary quadratic model (BQM), and State.problem is also set to that BQM.

**Example**

```python
>>> import dimod
>>> bqm = dimod.BQM.from_ising({}, {'ab': 0.5, 'bc': 0.5, 'ca': 0.5})
>>> state = State.from_sample({'a': -1, 'b': -1, 'c': -1}, bqm)
```

hybrid.core.State.from_samples

**classmethod** State.from_samples(samples, bqm, **kwargs)

Convenience method for constructing a state from raw (dict) samples.

Per-sample energy is calculated from the binary quadratic model (BQM), and State.problem is set to the BQM.

**Example**

```python
>>> import dimod
>>> bqm = dimod.BQM.from_ising({}, {'ab': 0.5, 'bc': 0.5, 'ca': 0.5})
>>> state = State.from_samples([{'a': -1, 'b': -1, 'c': -1},
... {'a': -1, 'b': -1, 'c': 1}], bqm)
```

Samplers

Classical and quantum Runnable dimod samplers for problems and subproblems.

Classes

**class** InterruptableTabuSampler (max_time=None, **tabu)

An interruptable tabu sampler for a binary quadratic problem.

**Parameters**

- **num_reads** (int, optional, default=1) – Number of states (output solutions) to read from the sampler.
- **tenure** (int, optional) – Tabu tenure, which is the length of the tabu list, or number of recently explored solutions kept in memory. Default is a quarter of the number of problem variables up to a maximum value of 20.
- **timeout** (int, optional, default=20) – Timeout for non-interruptable operation of tabu search. At the completion of each loop of tabu search through its problem variables, if this time interval has been exceeded, the search can be stopped by an interrupt signal or expiration of the timeout parameter.
• **initial_states_generator** *(str, 'none'/'tile'/'random', optional, default='random')* – Defines the expansion of input state samples into *initial_states* for the Tabu search, if fewer than *num_reads* samples are present. See *sample()*.

• **max_time** *(float, optional, default=None)* – Total running time in milliseconds.

See *Examples*.

**class QPUSubproblemExternalEmbeddingSampler**( *num_reads=100, qpu_sampler=None, sampling_params=None, **runopts**)

A quantum sampler for a subproblem with a defined minor-embedding.

**Note:** Externally supplied embedding must be present in the input state.

**Parameters**

• **num_reads** *(int, optional, default=100)* – Number of states (output solutions) to read from the sampler.

• **qpu_sampler** *(dimod.Sampler, optional, default=DWaveSampler(client="qpu"))* – Quantum sampler such as a D-Wave system.

• **sampling_params** *(dict)* – Dictionary of keyword arguments with values that will be used on every call of the (external-embedding-wrapped QPU) sampler.

See *Examples*.

**class QPUSubproblemAutoEmbeddingSampler**( *num_reads=100, qpu_sampler=None, sampling_params=None, auto_embedding_params=None, **runopts**)

A quantum sampler for a subproblem with automated heuristic minor-embedding.

**Parameters**

• **num_reads** *(int, optional, default=100)* – Number of states (output solutions) to read from the sampler.

• **qpu_sampler** *(dimod.Sampler, optional, default=DWaveSampler(client="qpu"))* – Quantum sampler such as a D-Wave system. Subproblems that do not fit the sampler’s structure are minor-embedded on the fly with *AutoEmbeddingComposite*.

• **sampling_params** *(dict)* – Dictionary of keyword arguments with values that will be used on every call of the (embedding-wrapped QPU) sampler.

• **auto_embedding_params** *(dict, optional)* – If provided, parameters are passed to the *AutoEmbeddingComposite* constructor as keyword arguments.

See *Examples*.

**class RandomSubproblemSampler**

A random sample generator for a subproblem.

**class ReverseAnnealingAutoEmbeddingSampler**( *num_reads=100, anneal_schedule=None, qpu_sampler=None, sampling_params=None, auto_embedding_params=None, **runopts**)

A quantum reverse annealing sampler for a subproblem with automated heuristic minor-embedding.

**Parameters**
• **num_reads** *(int, optional, default=100)* – Number of states (output solutions) to read from the sampler.

• **anneal_schedule** *(list(list), optional, default=[[0, 1], [0.5, 0.5], [1, 1]])* – An anneal schedule defined by a series of pairs of floating-point numbers identifying points in the schedule at which to change slope. The first element in the pair is time \( t \) in microseconds; the second, normalized persistent current \( s \) in the range \([0,1]\). The resulting schedule is the piecewise-linear curve that connects the provided points. For more details, see `validate_anneal_schedule()`.

• **qpu_sampler** *(dimod.Sampler, optional)* – Quantum sampler such as a D-Wave system. Subproblems that do not fit the sampler’s structure are minor-embedded on the fly with `AutoEmbeddingComposite`

If sampler is not provided, it defaults to:

```python
default_sampler = DWaveSampler(
    client="qpu",
    solver=dict(max_anneal_schedule_points__gte=len(anneal_schedule)))
```

• **sampling_params** *(dict)* – Dictionary of keyword arguments with values that will be used on every call of the (embedding-wrapped QPU) sampler.

• **auto_embedding_params** *(dict, optional)* – If provided, parameters are passed to the `AutoEmbeddingComposite` constructor as keyword arguments.

**class SimulatedAnnealingProblemSampler** *(num_reads=None, num_sweeps=1000, beta_range=None, beta_schedule_type='geometric', initial_states_generator='random', **runopts)*

A simulated annealing sampler for a complete problem.

**Parameters**

• **num_reads** *(int, optional, default=len(state.samples) or 1)* – Number of states (output solutions) to read from the sampler.

• **num_sweeps** *(int, optional, default=1000)* – Number of sweeps or steps.

• **beta_range** *(tuple, optional)* – A 2-tuple defining the beginning and end of the beta schedule, where beta is the inverse temperature. The schedule is applied linearly in beta. Default range is set based on the total bias associated with each node.

• **beta_schedule_type** *(string, optional, default='geometric')* – Beta schedule type, or how the beta values are interpolated between the given ‘beta_range’. Supported values are: linear and geometric.

• **initial_states_generator** *(str, 'none'/'tile'/'random', optional, default='random')* – Defines the expansion of input state samples into `initial_states` for the simulated annealing, if fewer than `num_reads` samples are present. See `sample()`.

**class SimulatedAnnealingSubproblemSampler** *(num_reads=None, num_sweeps=1000, beta_range=None, beta_schedule_type='geometric', initial_states_generator='random', **runopts)*

A simulated annealing sampler for a subproblem.

**Parameters**

• **num_reads** *(int, optional, default=len(state.subsamples) or 1)* – Number of states (output solutions) to read from the sampler.
• **num_sweeps** (*int*, *optional*, *default=1000*) – Number of sweeps or steps.

• **beta_range** (*tuple*, *optional*) – A 2-tuple defining the beginning and end of the beta schedule, where beta is the inverse temperature. The schedule is applied linearly in beta. Default range is set based on the total bias associated with each node.

• **beta_schedule_type** (*string*, *optional*, *default='geometric'*) – Beta schedule type, or how the beta values are interpolated between the given `beta_range`. Supported values are: linear and geometric.

• **initial_states_generator** (*str*, 'none'/'tile'/'random', *optional*, *default='random'*) – Defines the expansion of input state subsamples into `initial_states` for the simulated annealing, if fewer than `num_reads` subsamples are present. See `sample()`.

See Examples.

class TabuProblemSampler(*num_reads=None*, *tenure=None*, *timeout=100*, *initial_states_generator='random'*, **runopts*)

A tabu sampler for a binary quadratic problem.

Parameters

• **num_reads** (*int*, *optional*, *default=len(state.samples) or 1*) – Number of states (output solutions) to read from the sampler.

• **tenure** (*int*, *optional*) – Tabu tenure, which is the length of the tabu list, or number of recently explored solutions kept in memory. Default is a quarter of the number of problem variables up to a maximum value of 20.

• **timeout** (*int*, *optional*, *default=100*) – Total running time in milliseconds.

• **initial_states_generator** (*str*, 'none'/'tile'/'random', *optional*, *default='random'*) – Defines the expansion of input state subsamples into `initial_states` for the Tabu search, if fewer than `num_reads` samples are present. See `sample()`.

See Examples.

class TabuSubproblemSampler(*num_reads=None*, *tenure=None*, *timeout=100*, *initial_states_generator='random'*, **runopts*)

A tabu sampler for a subproblem.

Parameters

• **num_reads** (*int*, *optional*, *default=len(state.subsamples) or 1*) – Number of states (output solutions) to read from the sampler.

• **tenure** (*int*, *optional*) – Tabu tenure, which is the length of the tabu list, or number of recently explored solutions kept in memory. Default is a quarter of the number of problem variables up to a maximum value of 20.

• **timeout** (*int*, *optional*, *default=100*) – Total running time in milliseconds.

• **initial_states_generator** (*str*, 'none'/'tile'/'random', *optional*, *default='random'*) – Defines the expansion of input state subsamples into `initial_states` for the Tabu search, if fewer than `num_reads` subsamples are present. See `sample()`.

See Examples.
Examples

QPUSubproblemExternalEmbeddingSampler

This example works on a binary quadratic model of two AND gates in series by sampling a BQM representing just one of the gates. Output $z$ of gate $z = x \land y$ connects to input $a$ of gate $c = a \land b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated by sampling the subproblem 100 times on a D-Wave system. The execution results shown here were three valid solutions to the subproblem; for example, $x = 0, y = 1, z = 0$ occurred 22 times.

```python
import dimod
import minorminer
from dwave.system.samplers import DWaveSampler
from hybrid.samplers import QPUSubproblemExternalEmbeddingSampler
from hybrid.core import State

# Define a problem and a subproblem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c': 6.0},
                                  {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0,
                                   ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0,
                                   ('a', 'z'): -4.0}, -1.0, 'BINARY')
sub_bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0},
                                     {('x', 'y'): 2.0, ('x', 'z'): -4.0, ('y', 'z'): -4.0}, -1.0, 'BINARY')

# Find a minor-embedding for the subproblem
qpu_sampler = DWaveSampler()
sub_embedding = minorminer.find_embedding(list(sub_bqm.quadratic.keys()), qpu_sampler.edgelist)

# Set up the sampler with an initial state
sampler = QPUSubproblemExternalEmbeddingSampler(num_reads=100)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
state.update(subproblem=sub_bqm, embedding=sub_embedding)

# Sample the subproblem on the QPU (REPL)
>>> new_state = sampler.run(state).result()
>>> print(new_state.subsamples.record)
[[[0, 1, 0], -1., 22] [[0, 0, 0], -1., 47] [[1, 0, 0], -1., 31]]
```

QPUSubproblemAutoEmbeddingSampler

This example works on a binary quadratic model of two AND gates in series by sampling a BQM representing just one of the gates. Output $z$ of gate $z = x \land y$ connects to input $a$ of gate $c = a \land b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated by sampling the subproblem 100 times on a D-Wave system. The execution results shown here were four valid solutions to the subproblem; for example, $x = 0, y = 0, z = 0$ occurred 53 times.

```python
import dimod
from hybrid.samplers import QPUSubproblemAutoEmbeddingSampler
from hybrid.core import State
```

(continues on next page)
# Define a problem and a subproblem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c': -6.0},
   {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0, ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0},
   -1.0, 'BINARY')
sub_bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0},
   {('x', 'y'): 2.0, ('x', 'z'): -4.0, ('y', 'z'): -4.0},
   -1.0, dimod.Vartype.BINARY)

# Set up the sampler with an initial state
sampler = QPUSubproblemAutoEmbeddingSampler(num_reads=10)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
state.update(subproblem=sub_bqm)

# Sample the subproblem on the QPU (REPL)
>>> new_state = sampler.run(state).result()
>>> print(new_state.subsamples.record)
[(0, 0, 0), -1., 53] [(0, 1, 0), -1., 15] [(1, 0, 0), -1., 31] [(1, 1, 1), 1., 1]

SimulatedAnnealingSubproblemSampler

This example works on a binary quadratic model of two AND gates in series by sampling a BQM representing just one of the gates. Output \( z \) of gate \( z = x \land y \) connects to input \( a \) of gate \( c = a \land b \). An initial state is manually set with invalid solution \( x = y = 0, z = 1; a = b = 1, c = 0 \). The state is updated by sampling the subproblem 10 times. The execution results shown here were valid solutions to the subproblem; for example, \( x = 0, y = 1, z = 0 \).
This example works on a binary quadratic model of two AND gates in series by sampling a BQM representing just one of the gates. Output $z$ of gate $z = x \land y$ connects to input $a$ of gate $c = a \land b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated by a tabu search on the subproblem. The execution results shown here was a valid solution to the subproblem: example, $x = 0, y = 1, z = 0$.

```python
import dimod
from hybrid.samplers import TabuSubproblemSampler
from hybrid.core import State

# Define a problem and a subproblem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c': 6.0},
    {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0, ('a', 'z'): -4.0, ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0},
    -1.0, 'BINARY')
sub_bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0},
    {('x', 'y'): 2.0, ('x', 'z'): -4.0, ('y', 'z'): -4.0},
    -1.0, dimod.Vartype.BINARY)

# Set up the sampler with an initial state
sampler = TabuSubproblemSampler(tenure=2, timeout=5)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
state.update(subproblem=sub_bqm)

>>> # Sample the subproblem (REPL)
>>> print(new_state.subsamples.record)
[[[0, 1, 0], -1., 1]]
```

## TabuProblemSampler

This example works on a binary quadratic model of two AND gates in series, where output $z$ of gate $z = x \land y$ connects to input $a$ of gate $c = a \land b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated by a tabu search. The execution results shown here was a valid solution to the problem: example, $x = y = z = a = b = c = 1$.

```python
import dimod
from hybrid.samplers import TabuProblemSampler
from hybrid.core import State

# Define a problem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c': 6.0},
    {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0, ('a', 'z'): -4.0, ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0},
    -1.0, 'BINARY')

# Set up the sampler with an initial state
sampler = TabuProblemSampler(tenure=2, timeout=5)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
state.update(subproblem=sub_bqm)
```
import dimod
from hybrid.samplers import InterruptableTabuSampler
from hybrid.core import State

# Define a problem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c': 6.0},
        {'(y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0, ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0, ('a', 'z'): -4.0}, -1.0, 'BINARY')

# Set up the sampler with an initial state
sampler = InterruptableTabuSampler(tenure=2, quantum_timeout=30, timeout=5000)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)

# Sample the problem (REPL)
>>> new_state = sampler.run(state)
>>> print(new_state.samples)
Response(rec.array([([1, 1, 1, 1, 1, 1], -1.0, 1)],
    dtype=[('sample', 'i1', (6,)), ('energy', '<f8'), ('num_occurrences', '<i4')],
    ['a', 'b', 'c', 'x', 'y', 'z'], {}, 'BINARY'))
RandomSubproblemSampler

This example works on a binary quadratic model of two AND gates in series by sampling a BQM representing just one of the gates. Output $z$ of gate $z = x \land y$ connects to input $a$ of gate $c = a \land b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated with a random sample.

```python
import dimod
from hybrid.samplers import RandomSubproblemSampler
from hybrid.core import State

# Define a problem and a subproblem
bqm = dimod.BinaryQuadraticModel({"x": 0.0, "y": 0.0, "z": 8.0, "a": 2.0, "b": 0.0, "c": -6.0},
\n  {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0,
   ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0,
\n  },
\n-1.0, 'BINARY')

sub_bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0},
\n  {('x', 'y'): 2.0, ('x', 'z'): -4.0, ('y', 'z'): -4.0},
\n-1.0, dimod.Vartype.BINARY)

# Set up the sampler with an initial state
sampler = RandomSubproblemSampler()
state = State.from_sample({"x": 0, "y": 0, "z": 1, "a": 1, "b": 1, "c": 0}, bqm)
state.update(subproblem=sub_bqm)

# Sample the subproblem a couple of times (REPL)
>>> new_state = sampler.run(state).result()
>>> print(new_state.subsamples.record)
[[[0, 0, 0], -1., 1]]

>>> new_state = sampler.run(state).result()
>>> print(new_state.subsamples.record)
[[[1, 1, 1], 1., 1]]
```

Composers

Class

class IdentityComposer

Copy `subsamples` to `samples` verbatim.

class SplatComposer

A composer that overwrites current samples with subproblem samples.

See Examples.

class GreedyPathMerge

Dialectic-search merge operation [KS]. Generates a path from one input state, representing the thesis, to another input state, representing the antithesis, using a greedy method of single bit flips selected by decreasing energy.

Returns the best sample on the path, which represents the synthesis.

Note: only the lowest-energy sample, is considered from either input state.

See Examples.
Examples

SplatComposer

This example runs one iteration of a SplatComposer composer, overwriting an initial solution to a 6-variable binary quadratic model of all zeros with a solution to a 3-variable subproblem that was manually set to all ones.

```python
import dimod
from hybrid.composers import SplatComposer
from hybrid.core import State, SampleSet
from hybrid.utils import min_sample

bqm = dimod.BinaryQuadraticModel({t: 0 for t in range(6)},
    {(t, (t+1) % 6): 1 for t in range(6)},
    0, 'BINARY')

composer = SplatComposer()
state0 = State.from_sample(min_sample(bqm), bqm)
state1 = state0.updated(subsamples=SampleSet.from_samples({3: 1, 4: 1, 5: 1}, 'BINARY'), 0.0)
composed_state = composer.run(state1).result()

>>> print(composed_state.samples)
Response(rec.array(
    [[0, 0, 0, 1, 1, 1], 1, 2]),
    dtype=[('sample', 'i1', (6,)), ('num_occurrences', '<i8'), ('energy', '<i8')])
```

GreedyPathMerge

This example runs one iteration of a GreedyPathMerge composer on a thesis and antithesis State to find a ground state of a square graph. By inverting the state of variable \(d\) and \(c\) in samples_d and then variable \(a\) of the lowest energy sample of samples_a (second sample), the composer finds a path between these two samples that contains the ground state shown on the right of the top figure.

Fig. 12: Square problem with two ground states.
import dimod
bqm = dimod.BinaryQuadraticModel({}, {'ab': 1.0, 'bc': 1.0, 'cd': 1.0, 'da': 1}, 0, 'SPIN')
samples_d = {'a': 1, 'b': 1, 'c': -1, 'd': -1}
samples_a = [{'a': -1, 'b': -1, 'c': 1, 'd': 1}, {'a': -1, 'b': 1, 'c': 1, 'd': 1}]
states = [hybrid.State.from_samples(samples_d, bqm),
          hybrid.State.from_samples(samples_a, bqm)]
synthesis = GreedyPathMerge().next(states)

>>> print(synthesis.samples)
a   b   c   d   energy   num_occ.
0  +1  +1  +1   -4.0    1
[ 1 rows, 4 variables ]

Decomposers

Classes

class EnergyImpactDecomposer(size, min_gain=None, rolling=True, rolling_history=1.0, silent_rollback=True, traversal='energy', **runopts)

Selects a subproblem of variables maximally contributing to the problem energy.

The selection currently implemented does not ensure that the variables are connected in the problem graph.

Parameters

- **size (int)** – Nominal number of variables in the subproblem. Actual subproblem can be smaller, depending on other parameters (e.g. min_gain).
- **min_gain (int, optional, default=-inf)** – Minimum reduction required to BQM energy, given the current sample. A variable is included in the subproblem only if inverting its sample value reduces energy by at least this amount.
• **rolling** *(bool, optional, default=True)* – If True, successive calls for the same problem (with possibly different samples) produce subproblems on different variables, selected by rolling down the list of all variables sorted by decreasing impact.

• **rolling_history** *(float, optional, default=1.0)* – Fraction of the problem size, as a float in range 0.0 to 1.0, that should participate in the rolling selection. Once reached, subproblem unrolling is reset.

• **silent_rewind** *(bool, optional, default=True)* – If False, raises EndOfStream when resetting/rewinding the subproblem generator upon the reset condition for unrolling.

• **traversal** *(str, optional, default='energy')* – Traversal algorithm used to pick a subproblem of size variables. Options are:

  - energy: Use the next size variables in the list of variables ordered by descending energy impact.
  - bfs: Breadth-first traversal seeded by the next variable in the energy impact list.
  - pfs: Priority-first traversal seeded by variables from the energy impact list, proceeding with the variable on the search boundary that has the highest energy impact.

See *Examples*.

**class IdentityDecomposer**
Selects a subproblem that is a full copy of the problem.

**class RandomConstraintDecomposer(size, constraints, **runopts)**
Selects variables randomly as constrained by groupings.

By grouping related variables, the problem’s structure can guide the random selection of variables so subproblems are related to the problem’s constraints.

**Parameters**

- **size** *(int)* – Number of variables in the subproblem.

- **constraints** *(list[set])* – Groups of variables in the BQM, as a list of sets, where each set is associated with a constraint.

See *Examples*.

**class RandomSubproblemDecomposer(size, **runopts)**
Selects a subproblem of size random variables.

The selection currently implemented does not ensure that the variables are connected in the problem graph.

**Parameters size** *(int)* – Number of variables in the subproblem.

See *Examples*.

**class RoofDualityDecomposer(sampling_mode=True, **runopts)**
Selects a subproblem with variables that cannot be fixed by roof duality.

Roof duality finds a lower bound for the minimum of a quadratic polynomial. It can also find minimizing assignments for some of the polynomial's variables; these fixed variables take the same values in all optimal solutions [BHT] [BH]. A quadratic pseudo-Boolean function can be represented as a network to find the lower bound through network-flow computations. This decomposer can also use maximum flow in the implication network to fix variables. Consequently, you can find an assignment for the remaining variables that attains the optimal value.

**Parameters sampling_mode** *(bool, optional, default=True)* – In sampling mode, only roof-duality is used. When sampling_mode is false, strongly connected components are
used to fix more variables, but in some optimal solutions these variables may take different values.

**class TilingChimeraDecomposer** *(size=(4, 4, 4), loop=True, **runopts)*

Returns sequential Chimera lattices that tile the initial problem.

A Chimera lattice is an m-by-n grid of Chimera tiles, where each tile is a bipartite graph with shores of size t. The problem is decomposed into a sequence of subproblems with variables belonging to the Chimera lattices that tile the problem Chimera lattice. For example, a 2x2 Chimera lattice could be tiled 64 times (8x8) on a fully-yielded D-Wave 2000Q system (16x16).

**Parameters**

- **size** *(int, optional, default=(4, 4, 4)) – Size of the Chimera lattice as (m, n, t), where m is the number of rows, n the columns, and t the size of shore in the Chimera lattice.*
- **loop** *(Bool, optional, default=True) – Cycle continually through the tiles.*

See **Examples**.

**Examples**

**EnergyImpactDecomposer**

This example iterates twice on a 10-variable binary quadratic model with a random initial sample set. size configuration limits the subproblem in the first iteration to the first 4 variables shown in the output of flip_energy_gains.

```python
import dimod
from hybrid.decomposers import EnergyImpactDecomposer
from hybrid.core import State
from hybrid.utils import min_sample, flip_energy_gains

bqm = dimod.BinaryQuadraticModel({t: 0 for t in range(10)},
                               {(t, (t+1) % 10): 1 for t in range(10)},
                               0, 'BINARY')

decomposer = EnergyImpactDecomposer(size=4, rolling=True, rolling_history=1.0)
state0 = State.from_sample(min_sample(bqm), bqm)

>>> flip_energy_gains(bqm, state0.samples.first.sample)
[(0, 9), (0, 8), (0, 7), (0, 6), (0, 5), (0, 4), (0, 3), (0, 2), (0, 1), (0, 0)]

>>> state1 = decomposer.run(state0).result()

>>> list(state1.subproblem.variables)
[8, 7, 9, 6]

>>> state2 = decomposer.run(state1).result()

>>> list(state2.subproblem.variables)
[2, 3, 4, 5]
```

**RandomSubproblemDecomposer**

This example decomposes a 6-variable binary quadratic model with a random initial sample set to create a 3-variable subproblem.
import dimod
from hybrid.decomposers import RandomSubproblemDecomposer
from hybrid.core import State
from hybrid.utils import random_sample

bqm = dimod.BinaryQuadraticModel({t: 0 for t in range(6)}, {(t, (t+1) % 6): 1 for t in range(6)}, 0, 'BINARY')
decomposer = RandomSubproblemDecomposer(bqm, size=3)
state0 = State.from_sample(random_sample(bqm), bqm)
state1 = decomposer.run(state0).result()

>>> print(state1.subproblem)
BinaryQuadraticModel({2: 1.0, 3: 0.0, 4: 0.0}, {(2, 3): 1.0, (3, 4): 1.0}, 0.0, Vartype.BINARY)

TilingChimeraDecomposer

This example decomposes a 2048-variable Chimera structured binary quadratic model read from a file into 2x2x4-lattice subproblems.

import dimod
from hybrid.decomposers import TilingChimeraDecomposer
from hybrid.core import State
from hybrid.utils import random_sample

with open('problems/random-chimera/2048.09.qubo', 'r') as fp:
    bqm = dimod.BinaryQuadraticModel.from_coo(fp)
decomposer = TilingChimeraDecomposer(size=(2,2,4))
state0 = State.from_sample(random_sample(bqm), bqm)
state1 = decomposer.run(state0).result()

>>> print(state1.subproblem)
BinaryQuadraticModel({0: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: -3.0, 1: 0.0, 2: 0.0, 3: -4.0, 1024: -7.0, 1028: 0.0, ...}

RandomConstraintDecomposer

This example decomposes a 4-variable binary quadratic model that represents three serial NOT gates into 2-variable subproblems. The expected decomposition should use variables that represent one of the NOT gates rather than two arbitrary variables.

import dimod
from hybrid.decomposers import RandomConstraintDecomposer
from hybrid.core import State
from hybrid.utils import random_sample

... (continues on next page)
bqm = dimod.BinaryQuadraticModel({'w': -2.0, 'x': -4.0, 'y': -4.0, 'z': -2.0},
({('w', 'x'): 4.0, ('x', 'y'): 4.0, ('y', 'z'): 4.0},
3.0, 'BINARY')
decomposer = RandomConstraintDecomposer(2, [{'w', 'x'}, {'x', 'y'}, {'y', 'z'}])
state0 = State.from_sample(random_sample(bqm), bqm)
state1 = decomposer.run(state0).result()

>>> print(state1.subproblem)
BinaryQuadraticModel({'z': -2.0, 'y': 0.0}, {('z', 'y'): 4.0}, 0.0, Vartype.BINARY)

Flow Structuring

Classes that structure (hybrid) workflows.

Classes

class ArgMin (key=None, **runopts)
Selects the best state from a sequence of States.

Parameters key (callable/str) – Best state is judged according to a metric defined with a key.
The key can be a callable with a signature:

key :: (State s, Ord k) => s -> k

or a string holding a key name/path to be extracted from the input state with operator.attrgetter method.

By default, key == operator.attrgetter('samples.first.energy'), thus favoring states containing a sample with the minimal energy.

Examples

This example runs two branches—a classical tabu search interrupted by samples of subproblems returned from a D-Wave system—and selects the state with the minimum-energy sample:

RacingBranches(
    InterruptableTabuSampler(),
    EnergyImpactDecomposer(size=2)
    | QPUSubproblemAutoEmbeddingSampler()
    | SplatComposer()
) | ArgMin()

class Branch (components=(), **runopts)
Sequentially executed Runnable components.

Parameters components (iterable of Runnable) – Complete processing sequence to update a current set of samples, such as: decomposer | sampler | composer.

Input: Defined by the first branch component.

Output: Defined by the last branch component.
Examples

This example runs one iteration of a branch comprising a decomposer, local Tabu solver, and a composer. A 10-variable binary quadratic model is decomposed by the energy impact of its variables into a 6-variable subproblem to be sampled twice with a manually set initial state of all -1 values.

```python
>>> import dimod
# Create a binary quadratic model
>>> bqm = dimod.BQM({t: 0 for t in range(10)},
... (t, (t+1) % 10): 1 for t in range(10)},
... 0, 'SPIN')
>>> # Run one iteration on a branch
>>> branch = (EnergyImpactDecomposer(size=6, min_gain=-10) |
... TabuSubproblemSampler(num_reads=2) |
... SplatComposer())
>>> new_state = branch.next(State.from_sample(min_sample(bqm), bqm))
>>> print(new_state.subsamples) # doctest: +SKIP
   4 5 6 7 8 9 energy num_occ.
 0 +1 -1 -1 +1 -1 +1 -5.0 1
 1 +1 -1 -1 +1 -1 +1 -5.0 1
[ 2 rows, 6 variables ]
```

class **Branches** (*branches, **runopts*)

Runs multiple workflows of type Runnable in parallel, blocking until all finish.

Branches operates similarly to ParallelBranches, but each branch runs on a separate input State (while parallel branches all use the same input state).

Parameters

- **branches** ([Runnable]) – Runnable branches listed as positional arguments.

Input: States

Output: States

Note: Branches is also available via implicit parallelization binary operator &.

Examples

This example runs two branches, a classical tabu search and a random sampler, until both terminate:

```python
Branches(TabuSubproblemSampler(), RandomSubproblemSampler())
```

Alternatively:

TabuSubproblemSampler() & RandomSubproblemSampler()

class **Const** (**consts**)  
Set state variables to constant values.

Parameters

- **consts** (dict, optional) – Mapping of state variables to constant values, as keyword arguments.

Example

This example defines a workflow that resets the set of samples before a Tabu sampler call in order to avoid using existing samples as initial states. Instead, Tabu will use randomly generated initial states:
random_tabu = Const(samples=None) | TabuProblemSampler(initial_states_generator='random')

class Dup(n, *args, **kwargs):
    Duplicates input State, n times, into output States.

class Identity
    Trivial identity runnable. The output is a direct copy of the input.

class BlockingIdentity(*args, **kwargs)
    Trivial identity runnable that blocks indefinitely before producing output, but is interruptable. The output is a direct copy of the input, but to receive the output, the block has to be explicitly stopped (useful for example in RacingBranches to prevent short-circuiting of racing branches with the identity branch).

BlockingIdentity := Identity | Wait

Due to nature of Identity, BlockingIdentity is functionally equivalent to Wait.

class Lambda(next, error=none, init=none, **runopts)
    Creates a runnable on fly, given just its next function (optionally init and error functions can be specified too).

Parameters
    • next (callable) – Implementation of runnable’s next method, provided as a callable (usually a lambda expression for simple operations). Signature of the callable has to match the signature of next(); i.e., it accepts two arguments: runnable instance and state instance.
    • error (callable) – Implementation of runnable’s error method. See error().
    • init (callable) – Implementation of runnable’s init method. See init().

Note: Traits are not enforced, apart from the SISO requirement. Also, note Lambda runnables can only implement SISO systems.

Examples

This example creates and runs a simple runnable that multiplies state variables \(a\) and \(b\), storing them in \(c\).

```python
>>> Lambda(lambda _, s: s.updated(c=s.a * s.b)).run(State(a=2, b=3)).result()
{'a': 2, 'b': 3, 'c': 6}
```

This example applies \(x += 1\) to a sequence of input states.

```python
>>> Map(Lambda(lambda _, s: s.updated(x=s.x + 1))).run(States(State(x=0), State(x=1))).result()
[{'x': 1}, {'x': 2}]
```

class Loop(*args, **kwargs)
    Alias for LoopUntilNoImprovement.

class LoopUntilNoImprovement(*args, **kwargs)
    Iterates Runnable for up to max_iter times, or until a state quality metric, defined by the key function, shows no improvement for at least convergence number of iterations. Alternatively, maximum allowed runtime can be defined with max_time, or a custom termination Boolean function can be given with terminate (a predicate on key). Loop is always terminated on EndOfStream raised by body runnable.
Parameters

- **runnable** (*Runnable*) – A runnable that’s looped over.
- **max_iter** (int/None, optional, default=None) – Maximum number of times the runnable is run, regardless of other termination criteria. This is the upper bound. By default, an upper bound on the number of iterations is not set.
- **convergence** (int/None, optional, default=None) – Terminates upon reaching this number of iterations with unchanged output. By default, convergence is not checked, so the only termination criteria is defined with **max_iter**. Setting neither creates an infinite loop.
- **max_time** (float/None, optional, default=None) – Wall clock runtime termination criterion. Unlimited by default.
- **key** (callable/str) – Best state is judged according to a metric defined with a key. key can be a callable with a signature:

  ```python
  key :: (State s, Ord k) => s -> k
  ```

  or a string holding a key name/path to be extracted from the input state with `operator.attrgetter` method.

  By default, `key == operator.attrgetter('samples.first.energy')`, thus favoring states containing a sample with the minimal energy.
- **terminate** (callable, optional, default=None) – Loop termination Boolean function (a predicate on key value):

  ```python
  terminate :: (Ord k) => k -> Bool
  ```

**class LoopWhileNoImprovement** (*runnable*, **max_iter=**None, **max_tries=**None, **max_time=**None, **key=**None, **terminate=**None, **runopts**)

Iterates *Runnable* until a state quality metric, defined by the key function, shows no improvement for at least **max_tries** number of iterations or until **max_iter** number of iterations is exceeded. Alternatively, maximum allowed runtime can be defined with **max_time**, or a custom termination Boolean function can be given with **terminate** (a predicate on key).

**Note:** Unlike *LoopUntilNoImprovementLoop*, *LoopWhileNoImprovement* will run the loop body runnable with the same input if output shows no improvement (up to **max_tries** times), and it will use the new output if it’s better than the input.

Parameters

- **runnable** (*Runnable*) – A runnable that’s looped over.
- **max_iter** (int/None, optional, default=None) – Maximum number of times the runnable is run, regardless of other termination criteria. This is the upper bound. By default, an upper bound on the number of iterations is not set.
- **max_tries** (int, optional, default=None) – Maximum number of times the runnable is run for the same input state. On each improvement, the better state is used for the next input state, and the try/trial counter is reset. Defaults to an infinite loop (unbounded number of tries).
- **max_time** (float/None, optional, default=None) – Wall clock runtime termination criterion. Unlimited by default.
• **key** (*callable/str*) – Best state is judged according to a metric defined with a *key*. *key* can be a *callable* with a signature:

```python
def key(s, k): return s[k]
```

or a string holding a key name/path to be extracted from the input state with `operator.attrgetter` method.

By default, `key == operator.attrgetter('samples.first.energy')`, thus favoring states containing a sample with the minimal energy.

• **terminate** (*callable, optional, default=None*) – Loop termination

  Boolean function (a predicate on *key* value):

```python
def terminate(k): return k
```

### class Map(*runnable, **runopts*)

Runs a specified *Runnable* in parallel on all input states.

**Parameters**

- **runnable** (*Runnable*) – A runnable executed for every input state.

**Examples**

This example runs `TabuProblemSampler` on two input states in parallel, returning when both are done.

```python
>>> states = States(State(problem=bqm1), State(problem=bqm2))
>>> Map(TabuProblemSampler()).run(states).result()  # doctest: +SKIP
[<state_1_with_solution>, <state_2_with_solution>]
```

---

### Parallel

alias of `hybrid.flow.ParallelBranches`

---

### class ParallelBranches(*branches, **runopts*)

Runs multiple workflows of type *Runnable* in parallel, blocking until all finish.

Parallel/ParallelBranches operates similarly to Branches, but every branch re-uses the same input *State*.

**Parameters**

- **branches** ([*Runnable]*) – Comma-separated branches.

**Input**: *State*

**Output**: *States*

**Note**: *Parallel* is implemented as:

```python
Parallel(*branches) := Dup(len(branches)) | Branches(*branches)
```

**Note**: *ParallelBranches* is also available as *Parallel*.

---

### Examples

This example runs two branches, a classical tabu search and a random sampler, until both terminate:
Parallel(
    TabuSubproblemSampler(),
    RandomSubproblemSampler()
) | ArgMin()

### Race
alias of *hybrid.flow.RacingBranches*

class *RacingBranches*(*branches*, **runopts*)

Runs (races) multiple workflows of type *Runnable* in parallel, stopping all once the first finishes. Returns the results of all, in the specified order.

**Parameters**

*branches* ([Runnable]) – Comma-separated branches.

**Note:** Each branch runnable is called with run option *racing_context=True*, so it can adapt its behaviour to the context.

**Note:** *RacingBranches* is also available as *Race*.

### Examples

This example runs two branches: a classical tabu search interrupted by samples of subproblems returned from a D-Wave system.

```python
RacingBranches(
    InterruptableTabuSampler(),
    EnergyImpactDecomposer(size=2)
    | QPUSubproblemAutoEmbeddingSampler()
    | SplatComposer()
) | ArgMin()
```

class *Reduce*(*runnable*, *initial_state=None*, **runopts*)

Fold-left using the specified *Runnable* on a sequence of input states, producing a single output state.

**Parameters**

* runnable* ([Runnable]) – A runnable used as the fold-left operator. It should accept a 2-State input and produce a single State on output.

* initial_state* ([State], optional, default=None) – Optional starting state into which input states will be folded in. If undefined, the first input state is used as the *initial_state*.

class *TrackMin*(*key=None*, **runopts*)

Tracks and records the best *State* according to a metric defined with a *key* function; typically this is the minimal state.

**Parameters**

* key* ([callable/str, optional, default=None]) – Best state is judged according to a metric defined with a *key*. *key* can be a *callable* with a signature:

```python
key :: (State s, Ord k) => s -> k
```

or a string holding a key name/path to be extracted from the input state with *operator.attrgetter* method.
By default, `key == operator.attrgetter('samples.first.energy')`, thus favoring states containing a sample with the minimal energy.

- **output** *(bool, optional, default=False)* – Update the output state’s `output_key` with the `input_key` of the best state seen so far.

- **input_key** *(str, optional, default='samples')* – If `output=True`, then this defines the variable/key name in the input state that shall be included in the output state.

- **output_key** *(str, optional, default='samples')* – If `output=True`, then the key under which the `input_key` from the best state seen so far is stored in the output state.

**Note:** If `output` option is turned on, and `output_key` is not changed, the output will by default change the state’s `samples` on output.

```python
class Unwind(runnable, **runopts)
```

Iterates `Runnable` until `EndOfStream` is raised, collecting all output states along the way.

**Note:** the child runnable is called with run option `silent_rewind=False`, and it is expected to raise `EndOfStream` on unwind completion.

```python
class Wait(*args, **kwargs)
```

Run indefinitely (effectively blocking branch execution). Has to be explicitly stopped.

**Example**

To effectively exclude one branch from the race, i.e. prevent premature stopping of the race between the remaining branches, use `Wait` as the last element in a (fast-executing) racing branch:

```python
Race(
    Identity() | Wait(),
    InterruptableTabuSampler(),
    SimulatedAnnealingProblemSampler()
)
```

This is functionally identical to:

```python
Parallel(
    Identity(),
    Race(
        InterruptableTabuSampler(),
        SimulatedAnnealingProblemSampler()
    )
)
```

**Methods**

See *Primitives* for methods inherited from the `Runnable` superclass.

**Utilities**
**Methods**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bqm_edges_between_variables(bqm, variables)</code></td>
<td>Return edges connecting specified variables of a binary quadratic model.</td>
</tr>
<tr>
<td><code>bqm_induced_by(bqm, variables, sample)</code></td>
<td>Induce a binary quadratic model by fixing values of boundary variables.</td>
</tr>
<tr>
<td><code>bqm_reduced_to(bqm, variables, sample[, ...])</code></td>
<td>Reduce a binary quadratic model by fixing values of some variables.</td>
</tr>
<tr>
<td><code>chimera_tiles(bqm, m, n, t)</code></td>
<td>Map a binary quadratic model to a set of Chimera tiles.</td>
</tr>
<tr>
<td><code>flip_energy_gains(bqm, sample[, variables, ...])</code></td>
<td>Order variable flips by descending contribution to energy changes in a BQM.</td>
</tr>
<tr>
<td><code>max_sample(bqm)</code></td>
<td>Return a sample with all variables set to the maximal value for a binary quadratic model.</td>
</tr>
<tr>
<td><code>min_sample(bqm)</code></td>
<td>Return a sample with all variables set to the minimal value for a binary quadratic model.</td>
</tr>
<tr>
<td><code>random_sample(bqm)</code></td>
<td>Return a random sample for a binary quadratic model.</td>
</tr>
<tr>
<td><code>random_sample_seq(size, vartype)</code></td>
<td>Return a random sample.</td>
</tr>
<tr>
<td><code>sample_as_dict(sample)</code></td>
<td>Return sample object in dict format.</td>
</tr>
<tr>
<td><code>sample_as_list(sample)</code></td>
<td>Return sample object in list format.</td>
</tr>
<tr>
<td><code>select_localsearch_adversaries(bqm, sample)</code></td>
<td>Find variable flips that contribute high energy changes to a BQM.</td>
</tr>
<tr>
<td><code>select_random_subgraph(bqm, n)</code></td>
<td>Select randomly n variables of the specified binary quadratic model.</td>
</tr>
<tr>
<td><code>updated_sample(sample, replacements)</code></td>
<td>Update a copy of a sample with replacement values.</td>
</tr>
</tbody>
</table>

**hybrid.utils.bqm_edges_between_variables**

`bqm_edges_between_variables(bqm, variables)`

Return edges connecting specified variables of a binary quadratic model.

**Parameters**

- `bqm (dimod(BinaryQuadraticModel))` – Binary quadratic model (BQM).
- `variables (list/set)` – Subset of variables in the BQM.

**Returns** All edges connecting variables as tuples plus the variables themselves as tuples (v, v).

**Return type** list

**Examples**

This example returns connecting edges between 3 nodes of a BQM based on a 4-variable path graph.

```python
>>> import dimod
>>> bqm = dimod.BQM({}, {(0, 1): 1, (1, 2): 1, (2, 3): 1}, 0, 'BINARY')
>>> bqm_edges_between_variables(bqm, {0, 1, 3})
[(0, 1), (0, 0), (1, 1), (3, 3)]
```

**hybrid.utils.bqm_induced_by**

`bqm_induced_by(bqm, variables, sample)`

Induce a binary quadratic model by fixing values of boundary variables.
The function is optimized for \( \text{len}(\text{variables}) \ll \text{len}(\text{bqm}) \), that is, for fixing the majority of variables.

**Parameters**

- **\textbf{variables}** (list/set) – Subset of variables to keep in the reduced BQM, typically a subgraph.
- **\textbf{sample}** (dict/list) – Mapping of variable labels to values or a list when labels are sequential integers. Values are required only for boundary variables, that is, for variables with interactions with \textit{variables} (having edges with non-zero quadratic biases connected to the subgraph).

**Returns** A BQM induced by fixing values of those variables adjacent to its subset of variables and setting the energy offset to zero.

**Return type** dimod.BinaryQuadraticModel

**Examples**

This example induces a 2-variable BQM from a 6-variable path graph—the subset of nodes 2 and 3 of nodes 0 to 5—by fixing values of boundary variables 1 and 4.

```python
>>> import dimod
>>> import networkx as nx

>>> bqm = dimod.BinaryQuadraticModel({},
... {edge: edge[0] + 0.5 for edge in set(nx.path_graph(6).edges)}, 0, 'BINARY
˓→
')
>>> sample = {1: 3, 4: 3}
>>> len(bqm_induced_by(bqm, [2, 3], sample))
2
```

**hybrid.utils.bqm_reduced_to**

**bqm_reduced_to** \((\text{bqm}, \text{variables}, \text{sample}, \text{keep_offset}=\text{True})\)

Reduce a binary quadratic model by fixing values of some variables.

The function is optimized for \( \text{len}(\text{variables}) \sim \text{len}(\text{bqm}) \), that is, for small numbers of fixed variables.

**Parameters**

- **\textbf{variables}** (list/set) – Subset of variables to keep in the reduced BQM.
- **\textbf{sample}** (dict/list) – Mapping of variable labels to values or a list when labels are sequential integers. Must include all variables not specified in \textit{variables}.
- **\textbf{keep_offset}** (bool, optional, default=True) – If false, set the reduced binary quadratic model’s offset to zero; otherwise, uses the calculated energy offset.

**Returns** A reduced BQM.

**Return type** dimod.BinaryQuadraticModel
Examples

This example reduces a 3-variable BQM to two variables.

```python
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': -1, 'bc': -1, 'ca': -1}, 0, 'BINARY')
>>> sample = {'a': 1, 'b': 1, 'c': 0}
>>> subbqm = bqm_reduced_to(bqm, ['a', 'b'], sample)
>>> len(subbqm)
2
```

hybrid.utils.chimera_tiles

chimera_tiles(bqm, m, n, t)

Map a binary quadratic model to a set of Chimera tiles.

A Chimera lattice is an m-by-n grid of Chimera tiles, where each tile is a bipartite graph with shores of size t.

Parameters

- `bqm` (BinaryQuadraticModel) – Binary quadratic model (BQM).
- `m` (int) – Rows.
- `n` (int) – Columns.
- `t` (int) – Size of shore.

Returns Map as a dict where keys are tile coordinates (row, column, aisle) and values are partial embeddings of part of the BQM to a Chimera tile. Embeddings are those that would be generated by dwave_networkx's chimera_graph() function.

Return type dict

Examples

This example maps a 1-by-2 Chimera-derived BQM to 2 side-by-side tiles.

```python
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 2)  # Create a Chimera-based BQM
>>> bqm = dimod.BinaryQuadraticModel({}, {edge: edge[0] for edge in G.edges}, 0, 'BINARY')
>>> chimera_tiles(bqm, 1, 1, 4)  # doctest: +SKIP
{(0, 0, 0): {0: [0], 1: [1], 2: [2], 3: [3], 4: [4], 5: [5], 6: [6], 7: [7]},
 (0, 1, 0): {8: [0], 9: [1], 10: [2], 11: [3], 12: [4], 13: [5], 14: [6], 15: [7]}}
```

hybrid.utils.flip_energy_gains

flip_energy_gains(bqm, sample, variables=None, min_gain=None)

Order variable flips by descending contribution to energy changes in a BQM.

Parameters

- `bqm` (dimod.BinaryQuadraticModel) – Binary quadratic model (BQM).
• **sample** *(list/dict)* – Sample values as returned by dimod samplers (0 or 1 values for *dimod.BINARY* and -1 or +1 for *dimod.SPIN)*

• **variables** *(sequence, optional, default=None)* – Consider only flips of these variables. If undefined, consider all variables in *sample*.

• **min_gain** *(float, optional, default=None)* – Minimum required energy increase from flipping a sample value to return its corresponding variable.

Returns

**Energy changes in descending order, in the format of tuples** *(energy_gain, variable)*, for flipping the given sample value for each variable.

**Return type** *list*

**Examples**

This example returns the variable with maximum contribution to energy for the given sample.

```python
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': 0, 'bc': 1, 'cd': 2}, 0, 'SPIN')
>>> flip_energy_gains(bqm, {'a': -1, 'b': 1, 'c': 1, 'd': -1})[0][1]
'd'
```

---

**hybrid.utils.max_sample**

**max_sample** *(bqm)*

Return a sample with all variables set to the maximal value for a binary quadratic model.

**Parameters**

- **bqm** *(BinaryQuadraticModel)* – Binary quadratic model (BQM).

**Returns**

- A sample with maximal values for all variables of the BQM.

**Return type** *dict*

**Examples**

```python
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': -1, 'bc': -1, 'ca': -1}, 0, 'BINARY')
>>> max_sample(bqm)  # doctest: +SKIP
{'a': 1, 'b': 1, 'c': 1}
```

---

**hybrid.utils.min_sample**

**min_sample** *(bqm)*

Return a sample with all variables set to the minimal value for a binary quadratic model.

**Parameters**

- **bqm** *(BinaryQuadraticModel)* – Binary quadratic model (BQM).

**Returns**

- A sample with minimal values for all variables of the BQM.

**Return type** *dict*
Examples

```python
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': -1, 'bc': -1, 'ca': -1}, 0, 'BINARY')
>>> min_sample(bqm)
# doctest: +SKIP
{'a': 0, 'b': 0, 'c': 0}
```

**hybrid.utils.random_sample**

```python
random_sample(bqm)
```

Return a random sample for a binary quadratic model.

Parameters

- **bqm** (*BinaryQuadraticModel*) – Binary quadratic model (BQM).

Returns

A sample with random values for the BQM.

Return type

dict

Examples

```python
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': -1, 'bc': -1, 'ca': -1}, 0, 'BINARY')
>>> random_sample(bqm)
# doctest: +SKIP
{'a': 0, 'b': 1, 'c': 1}
```

**hybrid.utils.random_sample_seq**

```python
random_sample_seq(size, vartype)
```

Return a random sample.

Parameters

- **size** (*int*) – Sample size (number of variables).
- **vartype** (*dimod.Vartype*) – Variable type: for example, `Vartype.SPIN`, `BINARY`, or `{-1, 1}`.

Returns

Random sample of `size` in length, with values from `vartype`.

Return type

dict

Examples

```python
>>> random_sample_seq(4, dimod.BINARY)
# doctest: +SKIP
{0: 0, 1: 1, 2: 0, 3: 0}
```

**hybrid.utils.sample_as_dict**

```python
sample_as_dict(sample)
```

Return sample object in dict format.

Parameters

6.4. dwave-hybrid
• **sample** *(list/dict/dimod.SampleView)* – Sample object formatted as a list.

• **array, dict, or as returned by dimod samplers. (Numpy)* –

  Returns  Copy of *sample* formatted as a dict, with variable indices as keys.

  Return type  list

**Examples**

```python
g: sample = [1, 2, 3]  
g: sample_as_dict(sample)  
{0: 1, 1: 2, 2: 3}
```

**hybrid.utils.sample_as_list**

```python
sample_as_list(sample)
```

Return sample object in list format.

**Parameters**

• **sample** *(list/dict/dimod.SampleView)* – Sample object formatted as a list,

• **array, dict, or as returned by dimod samplers. Variable labeling (Numpy)* –

  • **be numerical. (must)* –

  Returns  Copy of *sample* formatted as a list.

  Return type  list

**Examples**

```python
g: sample = {0: 1, 1: 1}  
g: sample_as_list(sample)  
[1, 1]
```

**hybrid.utils.select_localsearch_adversaries**

```python
select_localsearch_adversaries(bqm, sample, max_n=None, min_gain=None)
```

Find variable flips that contribute high energy changes to a BQM.

**Parameters**

• **bqm** *(dimod.BinaryQuadraticModel)* – Binary quadratic model (BQM).

• **sample** *(list/dict)* – Sample values as returned by dimod samplers (0 or 1 values for dimod.BINARY and -1 or +1 for dimod.SPIN)

• **max_n** *(int, optional, default=None)* – Maximum contributing variables to return. By default, returns any variable for which flipping its sample value results in an energy gain of *min_gain*.

• **min_gain** *(float, optional, default=None)* – Minimum required energy increase from flipping a sample value to return its corresponding variable.
Returns Up to $max_n$ variables for which flipping the corresponding sample value increases the BQM energy by at least $min_gain$.

Return type list

Examples

This example returns 2 variables (out of up to 3 allowed) for which flipping sample values changes BQM energy by 1 or more. The BQM has energy gains of 0, -2, 2, 4 for variables a, b, c, d respectively for the given sample.

```python
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': 0, 'bc': 1, 'cd': 2}, 0, 'SPIN')
>>> select_localsearch_adversaries(bqm, {'a': -1, 'b': 1, 'c': 1, 'd': -1}, max_n=3, min_gain=1)
['d', 'c']
```

**hybrid.utils.select_random_subgraph**

`select_random_subgraph (bqm, n)`

Select randomly $n$ variables of the specified binary quadratic model.

Parameters

- `n` (*int*) – Number of requested variables. Must be between 0 and `len(bqm)`.

Returns $n$ variables selected randomly from the BQM.

Return type list

Examples

This example returns 2 variables of a 4-variable BQM.

```python
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': 0, 'bc': 1, 'cd': 2}, 0, 'BINARY')
>>> select_random_subgraph(bqm, 2)  # doctest: +SKIP
['d', 'b']
```

**hybrid.utils.updated_sample**

`updated_sample (sample, replacements)`

Update a copy of a sample with replacement values.

Parameters

- `sample` (*list/dict*) – Sample values as returned by dimod samplers to be copied.
- `replacements` (*list/dict*) – Sample values to replace in the copied `sample`.

Returns Copy of `sample` overwritten by specified values.

Return type list/dict
Examples

```python
>>> sample = {'a': 1, 'b': 1}
>>> updated_sample(sample, {'b': 2})  # doctest: +SKIP
{'a': 1, 'b': 2}
```

Dimod Conversion

These classes handle conversion between *dwave-hybrid* Runnable classes and dimod samplers.

Classes

**class** HybridSampler(workflow)

Produces a dimod.Sampler from a hybrid.Runnable-based sampler.

**Parameters**

- **workflow** (*Runnable*) – Hybrid workflow, likely composed, that accepts a binary quadratic model in the input state and produces sample(s) in the output state.

**Example**

This example produces a dimod sampler from TabuProblemSampler and uses its sample_ising mixin to solve a simple Ising problem.

```python
>>> hybrid_sampler = TabuProblemSampler()
>>> dimod_sampler = HybridSampler(hybrid_sampler)
>>> solution = dimod_sampler.sample_ising({}, {'ab': 0.5, 'bc': 0.5, 'ca': 0.5})
>>> solution.first.energy
-0.5
```

**class** HybridRunnable(sampler, fields, **sample_kwargs)

Produces a hybrid.Runnable from a dimod.Sampler (dual of HybridSampler).

The runnable samples from a problem defined in a state field named fields[0] and populates the state field referred to by fields[1].

**Parameters**

- **sampler** (*dimod.Sampler*) – dimod-compatible sampler which is run on every iteration of the runnable.
- **fields** (*tuple(str, str]*) – Input and output state field names.
- **sample_kwargs** (*dict*) – Sampler-specific parameters passed to sampler on every call.

**Example**

This example creates a Runnable from dimod sampler TabuSampler, runs it on an Ising model, and finds the lowest energy.

```python
>>> from tabu import TabuSampler
>>> import dimod
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {'ab': 0.5, 'bc': 0.5, 'ca': ˓
  0.5})
```
```python
>>> runnable = HybridRunnable(TabuSampler(), fields=('subproblem', 'subsamples'), timeout=100)
>>> state0 = State(subproblem=bqm, subsamples=SampleSet.from_samples_bqm(min_sample(bqm), bqm))
>>> state = runnable.run(state0)
>>> state.result()['subsamples'].first.energy  # doctest: +SKIP
-0.5
```

class HybridProblemRunnable(sampler, **sample_kwargs)

Produces a `hybridRunnable` from a `dimod.Sampler` (dual of `HybridSampler`).

The runnable that samples from `state.problem` and populates `state.samples`.

See an example in `hybrid.core.HybridRunnable`. An example of the duality with `HybridSampler` is:

```python
HybridProblemRunnable(HybridSampler(TabuProblemSampler())) == TabuProblemSampler()
```

class HybridSubproblemRunnable(sampler, **sample_kwargs)

Produces a `hybridRunnable` from a `dimod.Sampler` (dual of `HybridSampler`).

The runnable that samples from `state.subproblem` and populates `state.subsamples`.

See an example in `hybrid.core.HybridRunnable`.

### Traits

State traits are verified for all `Runnable` objects that inherit from `StateTraits` or its subclasses. Verification includes:

1. Minimal checks of workflow construction (composition of `Runnable` classes)
2. Runtime checks

All built-in `Runnable` classes declare state traits requirements that are either independent (for simple ones) or derived from a child workflow. Traits of a new `Runnable` must be expressed and modified at construction time by its parent. When developing new `Runnable` classes, constructing composite traits can be nontrivial for some advanced flow-control runnables. State traits validation base class and related state validation mixins (i/o validation toggle, i/o dimensionality, state structure).

When subclassing (combining with `Runnable`), list them in the following order, left to right:

- structure mixins (e.g. `SubsamplesIntaking` and `SubproblemSampler`)
- dimensionality mixins (e.g. `MultiInputStates` and `MISO`)
- validation toggles (e.g. `InputValidated` and `NotValidated`)
- StateTraits base class (not required if any of the above is used)
- Runnable base class

For example:

```python
class MyRunnable(hybrid.traits.SubsamplesIntaking, hybrid.traits.MISO, hybrid.Runnable):  pass
```

class StateTraits

Set of traits imposed on State. By default, not validated.

```python
validate_state_trait(state, trait, io)
```

Validate single input/output (io) state trait.

class InputValidated
class OutputValidated
class InputNotValidated
class OutputNotValidated

class Validated
    Validated input state(s) and output state(s).

class NotValidated
    Input state(s) and output state(s) are not validated.

class SingleInputState
class MultiInputStates
class SingleOutputState
class MultiOutputStates

class SISO
    Single Input, Single Output.

class SIMO
    Single Input, Multiple Outputs.

class MIMO
    Multiple Inputs, Multiple Outputs.

class MISO
    Multiple Inputs, Single Output.

class ProblemIntaking
class ProblemProducing
class SamplesIntaking
class SamplesProducing
class SubproblemIntaking
class SubproblemProducing
class SubsamplesIntaking
class SubsamplesProducing
class EmbeddingIntaking
class EmbeddingProducing
class ProblemDecomposer
class SubsamplesComposer
class ProblemSampler
class SubproblemSampler
class SamplesProcessor
class SubsamplesProcessor
Exceptions

exception RunnableError (message, state)
    Generic Runnable exception error that includes the error context, in particular, the State that caused the runnable component to fail.

exception InvalidStateError
    General state error.

exception StateTraitMissingError
    State missing a trait.

exception StateDimensionalityError
    Single state expected instead of a state sequence, or vice versa.

exception EndOfStream
    Signals end of stream for streaming runnables.

Reference Workflows

The code includes implementations of some reference workflows you can incorporate as provided into your application and also use to jumpstart development of custom workflows.

Kerberos

Kerberos hybrid sampler runs 3 sampling branches in parallel. In each iteration, best results from tabu search and simulated annealing are combined with best results from QPU sampling a subproblem.

Kerberos (max_iter=100, max_time=None, convergence=3, energy_threshold=None, sa_reads=1, sa_sweeps=10000, tabu_timeout=500, qpu_reads=100, qpu_sampler=None, qpu_params=None, max_subproblem_size=50)
An opinionated hybrid asynchronous decomposition sampler for problems of arbitrary structure and size. Runs Tabu search, Simulated annealing and QPU subproblem sampling (for high energy impact problem variables) in parallel and returns the best samples.

Kerberos workflow is used by KerberosSampler.

Termination Criteria Args:

    max_iter (int): Number of iterations in the hybrid algorithm.

    max_time (float/None, optional, default=None): Wall clock runtime termination criterion. Unlimitted by default.

    convergence (int): Number of iterations with no improvement that terminates sampling.

    energy_threshold (float, optional): Terminate when this energy threshold is surpassed. Check is performed at the end of each iteration.

Simulated Annealing Parameters:

    sa_reads (int): Number of reads in the simulated annealing branch.

    sa_sweeps (int): Number of sweeps in the simulated annealing branch.

Tabu Search Parameters:

    tabu_timeout (int): Timeout for non-interruptable operation of tabu search (time in milliseconds).

QPU Sampling Parameters:
qpu_reads (int): Number of reads in the QPU branch.

qpu_sampler (dimod.Sampler, optional, default=DWaveSampler()): Quantum sampler such as a D-Wave system.

qpu_params (dict): Dictionary of keyword arguments with values that will be used on every call of the QPU sampler.

max_subproblem_size (int): Maximum size of the subproblem selected in the QPU branch.

Returns Workflow (Runnable instance).

class KerberosSampler
An opinionated dimod-compatible hybrid asynchronous decomposition sampler for problems of arbitrary structure and size.

Examples

This example solves a two-variable Ising model.

```python
>>> import dimod
>>> import hybrid

>>> response = hybrid.KerberosSampler().sample_ising(...
...     {'a': -0.5, 'b': 1.0}, {('a', 'b'): -1})
# doctest: +SKIP

>>> response.data_vectors['energy']  # doctest: +SKIP
array([-1.5])
```

sample (bqm, init_sample=None, num_reads=1, **kwargs)
Run Tabu search, Simulated annealing and QPU subproblem sampling (for high energy impact problem variables) in parallel and return the best samples.

Sampling Args:

bqm (BinaryQuadraticModel): Binary quadratic model to be sampled from.

init_sample (SampleSet, callable, None): Initial sample set (or sample generator) used for each “read”. Use a random sample for each read by default.

num_reads (int): Number of reads. Each sample is the result of a single run of the hybrid algorithm.

Termination Criteria Args:

max_iter (int): Number of iterations in the hybrid algorithm.

max_time (float/None, optional, default=None): Wall clock runtime termination criterion. Unlimited by default.

convergence (int): Number of iterations with no improvement that terminates sampling.

energy_threshold (float, optional): Terminate when this energy threshold is surpassed. Check is performed at the end of each iteration.

Simulated Annealing Parameters:

sa_reads (int): Number of reads in the simulated annealing branch.

sa_sweeps (int): Number of sweeps in the simulated annealing branch.

Tabu Search Parameters:
**tabu_timeout** *(int)*: Timeout for non-interruptable operation of tabu search (time in milliseconds).

QPU Sampling Parameters:

**qpu_reads** *(int)*: Number of reads in the QPU branch.

**qpu_sampler** *(dimod.Sampler, optional, default=DWaveSampler())*: Quantum sampler such as a D-Wave system.

**qpu_params** *(dict)*: Dictionary of keyword arguments with values that will be used on every call of the QPU sampler.

**max_subproblem_size** *(int)*: Maximum size of the subproblem selected in the QPU branch.

Returns A *dimod* SampleSet object.

Return type *SampleSet*

### Parallel Tempering

Parallel tempering support and a reference workflow implementation.

**class FixedTemperatureSampler** *(beta=None, num_sweeps=10000, num_reads=None, aggregate=False, seed=None, **runopts)*

Parallel tempering propagate/update step.

The temperature (*beta*) can be specified upon object construction, and/or given externally (dynamically) in the input state.

On each call, run fixed temperature (~1/β) simulated annealing for *num_sweeps* (seeded by input sample(s)), effectively producing a new state by sampling from a Boltzmann distribution at the given temperature.

**Parameters**

- **beta** *(float, optional)* – Inverse of constant sampling temperature. If not supplied on construction, it must be present in the input state.

- **num_sweeps** *(int, optional, default=10k)* – Number of fixed temperature sampling sweeps.

- **num_reads** *(int, optional, default=len(state.samples))* – Number of samples produced. If undefined, inferred from the size of the input sample set.

- **aggregate** *(bool, optional, default=False)* – Aggregate samples (duplicity stored in *num_occurrences*).

- **seed** *(int, optional, default=None)* – Pseudo-random number generator seed.

**next**(state, **runopts)**

Execute one blocking iteration of an instantiated Runnable with a valid state as input.

**Parameters**

- **state** *(State)* – Computation state passed between connected components.

**Returns** The new state.

**Return type** State

### Examples

This code snippet runs one iteration of a sampler to produce a new state:
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))

class SwapReplicaPairRandom(betas=None, seed=None, **runopts)
    Parallel tempering swap replicas step.
    On each call, choose a random input state (replica), and probabilistically accept a swap with the adjacent state (replica). If swap is accepted, only samples contained in the selected states are exchanged.

    Betas can be supplied in constructor, or otherwise they have to present in the input states.

    Parameters

    • **betas** ([float], optional) – List of betas (inverse temperature), one for each input state. If not supplied, betas have to be present in the input states.

    • **seed** (int, default=None) – Pseudo-random number generator seed.

    swap_pair(betas, states, i, j)
        One pair of states’ (i, j) samples probabilistic swap.

    next(states, **runopts)
        Execute one blocking iteration of an instantiated Runnable with a valid state as input.

        Parameters

        • **state** (State) – Computation state passed between connected components.

        Returns

        The new state.

        Return type State

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

class SwapReplicasDownsweep(betas=None, **runopts)
    Parallel tempering swap replicas step.
    On each call, sweep down and probabilistically swap all adjacent pairs of replicas (input states).

    Betas can be supplied in constructor, or otherwise they have to present in the input states.

    Parameters

    • **betas** ([float], optional) – List of betas (inverse temperature), one for each input state. If not supplied, betas have to be present in the input states.

    next(states, **runopts)
        Execute one blocking iteration of an instantiated Runnable with a valid state as input.

        Parameters

        • **state** (State) – Computation state passed between connected components.

        Returns

        The new state.

        Return type State

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```
**ParallelTempering** \((\text{num\_sweeps} = 10000, \text{num\_replicas} = 10, \text{max\_iter} = \text{None}, \text{max\_time} = \text{None}, \text{convergence} = 3)\)

Parallel tempering workflow generator.

**Parameters**

- **num\_sweeps** (*int, optional*) – Number of sweeps in the fixed temperature sampling.
- **num\_replicas** (*int, optional*) – Number of replicas (parallel states / workflow branches).
- **max\_iter** (*int/None, optional*) – Maximum number of iterations of the update/swaps loop.
- **max\_time** (*int/None, optional*) – Maximum wall clock runtime (in seconds) allowed in the update/swaps loop.
- **convergence** (*int/None, optional*) – Number of times best energy of the coldest replica has to repeat before we terminate.

**Returns** Workflow (*Runnable instance*).

**HybridizedParallelTempering** \((\text{num\_sweeps} = 10000, \text{num\_replicas} = 10, \text{max\_iter} = \text{None}, \text{max\_time} = \text{None}, \text{convergence} = 3)\)

Parallel tempering workflow generator.

**Parameters**

- **num\_sweeps** (*int, optional*) – Number of sweeps in the fixed temperature sampling.
- **num\_replicas** (*int, optional*) – Number of replicas (parallel states / workflow branches).
- **max\_iter** (*int/None, optional*) – Maximum number of iterations of the update/swaps loop.
- **max\_time** (*int/None, optional*) – Maximum wall clock runtime (in seconds) allowed in the update/swaps loop.
- **convergence** (*int/None, optional*) – Number of times best energy of the coldest replica has to repeat before we terminate.

**Returns** Workflow (*Runnable instance*).

**Population Annealing**

Population annealing support and a reference workflow implementation.

**class** EnergyWeightedResampler \((\text{beta} = \text{None}, \text{seed} = \text{None}, **runopts)\)

Sample from the input sample set according to a distribution defined with sample energies (with replacement):

\[ p \sim \exp(-\text{sample.energy} / \text{temperature}) \sim \exp(-\beta \times \text{sample.energy}) \]

**Parameters**

- **beta** (*float*) – Inverse of sampling temperature. Can be defined on sampler construction, on run method invocation, or in the input state's \text{beta} variable.
- **seed** (*int, default=\text{None}* ) – Pseudo-random number generator seed.
Returns Input state with new samples. The lower the energy of an input sample, the higher will be its relative frequency in the output sample set.

next (state, **runopts)
Execute one blocking iteration of an instantiated Runnable with a valid state as input.

Parameters state (State) – Computation state passed between connected components.

Returns The new state.

Return type State

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```python
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

class ProgressBetaAlongSchedule (beta_schedule=None, **runopts)
Sets beta state variable to a schedule given on construction or in state at first run.

Parameters beta_schedule (iterable(float)) – The beta schedule. State’s beta is iterated according to the beta schedule.

Raises EndOfStream when beta schedule is depleted.

init (state, **runopts)
Run prior to the first next/run, with the first state received.
Default to NOP.

next (state, **runopts)
Execute one blocking iteration of an instantiated Runnable with a valid state as input.

Parameters state (State) – Computation state passed between connected components.

Returns The new state.

Return type State

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```python
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

class CalculateAnnealingBetaSchedule (length=2, interpolation='geometric', **runopts)
Calculate a best-guess beta schedule estimate for annealing methods, based on magnitudes of biases of the input problem, and the requested method of interpolation.

Parameters

- length (int) – Length of the produced beta schedule.
- interpolation (str, optional, default='geometric') – Interpolation used between the hot and the cold beta. Supported values are:
  - linear
  - geometric
next (state, **runopts)
Execute one blocking iteration of an instantiated Runnable with a valid state as input.

Parameters state (State) – Computation state passed between connected components.

Returns The new state.

Return type State

Examples
This code snippet runs one iteration of a sampler to produce a new state:

```python
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

PopulationAnnealing (num_reads=20, num_iter=20, num_sweeps=1000)
Population annealing workflow generator.

Parameters

- num_reads (int) – Size of the population of samples.
- num_iter (int) – Number of temperatures over which we iterate fixed-temperature sampling / resampling.
- num_sweeps (int) – Number of sweeps in the fixed temperature sampling step.

Returns Workflow (Runnable instance).

HybridizedPopulationAnnealing (num_reads=20, num_iter=20, num_sweeps=1000)
Workflow generator for population annealing initialized with QPU samples.

Parameters

- num_reads (int) – Size of the population of samples.
- num_iter (int) – Number of temperatures over which we iterate fixed-temperature sampling / resampling.
- num_sweeps (int) – Number of sweeps in the fixed temperature sampling step.

Returns Workflow (Runnable instance).

qbsolv

QBSolv inspired simple workflows.

SimplifiedQbsolv (max_iter=10, max_time=None, convergence=3, energy_threshold=None, max_subproblem_size=30)
Races a Tabu solver and a QPU-based sampler of flip-energy-impact induced subproblems.

For arguments description see: Kerberos.

6.5 dwave-inspector

A tool for visualizing problems submitted to, and answers received from, a D-Wave structured solver such as a D-Wave 2000Q quantum computer.
6.5.1 Introduction

dwave-inspector provides a graphic interface for examining D-Wave quantum computers’ problems and answers. As described in the Ocean documentation’s Getting Started, the D-Wave system solves problems formulated as binary quadratic models (BQM) that are mapped to its qubits in a process called minor-embedding. Because the way you choose to minor-embed a problem (the mapping and related parameters) affects solution quality, it can be helpful to see it.

For example, embedding a K3 fully-connected graph, such as the Boolean AND gate example into a D-Wave 2000Q, with its Chimera topology, requires representing one of the three variables with a “chain” of two physical qubits:

Fig. 14: The AND gate’s original BQM is represented on the left; its embedded representation, on the right, shows a two-qubit chain of qubits 1195 and 1199 for one variable.

The problem inspector shows you your chains at a glance: you see lengths, any breakages, and physical layout.

Usage and Examples

Import the problem inspector to enable it\(^1\) to hook into your problem submissions.

Use the `show()` method to visualize the embedded problem, and optionally the logical problem, in your default browser.

- **Inspecting an Embedded Problem**
- **Inspecting a Logical Problem**

Inspecting an Embedded Problem

This example shows the canonical usage: samples representing physical qubits on a quantum processing unit (QPU).\(^2\)

\(^1\) Importing the problem inspector activates for the session the capture of data such as problems sent to the QPU and returned responses, relevant details of minor-embedding, and warnings. The recommended workflow is to import it at the start of your coding session as is typical for Python packages (it is also possible, but less convenient, to specify in the submission that data such as embedding be returned with the response).
>>> from dwave.system import DWaveSampler
>>> import dwave.inspector
...  
>>> # Get solver
>>> sampler = DWaveSampler(solver=dict(qpu=True))
...  
>>> # Define a problem (actual qubits depend on the selected QPU’s working graph)
>>> h = {}
>>> J = {(0, 4): 1, (0, 5): 1, (1, 4): 1, (1, 5): -1}
>>> all(edge in sampler.edgelist for edge in J)
True
>>> # Sample
>>> response = sampler.sample_ising(h, J, num_reads=100)
...  
>>> # Inspect
>>> dwave.inspector.show(response) # doctest: +SKIP

Fig. 15: Edge values between qubits 0, 1, 4, 5, and the selected solution, are shown by color on the left; a histogram, on the right, shows the energies of returned samples.

**Inspecting a Logical Problem**

This example visualizes a problem specified logically and then automatically minor-embedded by Ocean’s EmbeddingComposite. For illustrative purposes it sets a weak chain_strength to show broken chains.

Define a problem and sample it for solutions:

```python
import dimod
import dwave.inspector
from dwave.system import DWaveSampler, EmbeddingComposite
```
# Define problem
bqm = dimod.BQM.from_ising({}, {'ab': 1, 'bc': 1, 'ca': 1})

# Get sampler
sampler = EmbeddingComposite(DWaveSampler(solver=dict(qpu=True)))

# Sample with low chain strength
sampleset = sampler.sample(bqm, num_reads=1000, chain_strength=0.1)

Inspect the problem:
dwave.inspector.show(sampleset)

Fig. 16: The logical problem, on the left, shows that the value for variable $b$ is based on a broken chain; the embedded problem, on the right, highlights the broken chain (its two qubits have different values) in bold red.

6.5.2 Reference Documentation

Release 2.5.0

Date Aug 12, 2020

Visualizing Problems

Typically you use the `show()` function on a `SampleSet` returned from the quantum computer or on the SAPI problem ID\(^1\). Other problem inputs, such as the binary quadratic model—in BQM, Ising, or QUBO formats—and an

\(^1\) For problems submitted in the active session (i.e., once the problem inspector has been imported).
embedding, are optional. However, to visualize a logical problem if dimod’s EmbeddingComposite or derived classes are not used, you must supply the embedding.

Below are some options for providing problem data to the `show()` function, where response was returned for a problem defined directly on physical qubits and sampleset returned from a problem submitted using EmbeddingComposite:

```python
show(response)
show('69ace80c-d3b1-448a-a028-b51b94f4a49d')  # Using a SAPI problem ID
show((h, J), response)
show(Q, response)
show((h, J), response, dict(embedding=embedding, chain_strength=5))
show(sampleset)
show(bqm, sampleset)
```

To see detailed parameter information, see the relevant function below.

The `show()` function supports flow control for scripts with the `block` parameter. For example, the default setting of `once` (dwave.inspector.Block.ONCE) blocks until your problem is loaded from the inspector web server and `forever` blocks until you terminate with a Ctrl+C/SIGTERM.

###Classes

**class Block**

Flow-control settings for scripts.

An enum with values: NEVER, ONCE, FOREVER. The default setting of once (dwave.inspector.Block.ONCE) blocks until your problem is loaded from the inspector web server

###Examples

This example does not block while the problem is loaded.

```python
>>> dwave.inspector.show(response, block='never')  # doctest: +SKIP
```

###Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>show(*args, **kwargs)</code></td>
<td>Auto-detect and forward to the <code>show_*</code> optimal for the specified arguments.</td>
</tr>
<tr>
<td><code>show_bqm_response(bqm, embedding_context, ...)</code></td>
<td>Visualize a quantum machine instruction (QMI) response and binary quadratic model.</td>
</tr>
<tr>
<td><code>show_bqm_sampleset(bqm, sampleset, sampler)</code></td>
<td>Visualize a returned sampleset and binary quadratic model.</td>
</tr>
<tr>
<td><code>show_qmi(problem, response[, ....])</code></td>
<td>Visualize a quantum machine instruction (QMI).</td>
</tr>
</tbody>
</table>

`dwave.inspector.show`

`show (*args, **kwargs)`

Auto-detect and forward to the `show_*` optimal for the specified arguments.

For description of accepted arguments, see of `show_qmi()`, `show_bqm_response()`, or
**show_bqm_sampleset()**.

**Note:** Low-level data capture is enabled on `dwave.inspector` import. Data captured includes the full quantum machine instruction (QMI), QPU response, embedding context, warnings, and sampling parameters.

If data capture is enabled prior to embedding/sampling, you need provide to `show()` only a response or problem ID for QMI inspection or a `SampleSet` for logical problem and QMI inspection.

If data capture is not enabled prior to embedding/sampling, provide all relevant data explicitly to `show()`.

**Examples**

This example shows ways to visualize just a QMI (not a logical problem):

```python
show(response)
show((h, J), response)
show(Q, response)
show('69ace80c-d3b1-448a-a028-b51b94f4a49d')
```

This example visualizes a QMI and explicit embedding:

```python
show((h, J), response, dict(embedding=embedding, chain_strength=5))
```

This example shows embedding and warnings read from the sampleset:

```python
show(bqm, sampleset)
```

This example shows embedding and warnings read from a `SampleSet`, from which the logical problem is reconstructed:

```python
show(sampleset)
```

**dwave.inspector.show_bqm_response**

```python
show_bqm_response(bqm, embedding_context, response, warnings=None, params=None)
```

Visualize a quantum machine instruction (QMI) response and binary quadratic model.

**dwave.inspector.show_bqm_sampleset**

```python
show_bqm_sampleset(bqm, sampleset, sampler, embedding_context=None, warnings=None, params=None)
```

Visualize a returned sampleset and binary quadratic model.

**dwave.inspector.show_qmi**

```python
show_qmi(problem, response, embedding_context=None, warnings=None, params=None)
```

Visualize a quantum machine instruction (QMI).
6.6 dwave-neal

An implementation of a simulated annealing sampler.

6.6.1 Example Usage

```python
import neal

sampler = neal.SimulatedAnnealingSampler()

h = {0: -1, 1: -1}
J = {(0, 1): -1}
response = sampler.sample_ising(h, J)
```

Introduction

Samplers are processes that sample from low energy states of a problem’s objective function. A binary quadratic model (BQM) sampler samples from low energy states in models such as those defined by an Ising equation or a Quadratic Unconstrained Binary Optimization (QUBO) problem and returns an iterable of samples, in order of increasing energy. A dimod sampler provides ‘sample_qubo’ and ‘sample_ising’ methods as well as the generic BQM sampler method.

The SimulatedAnnealingSampler sampler implements the simulated annealing algorithm, based on the technique of cooling metal from a high temperature to improve its structure (annealing). This algorithm often finds good solutions to hard optimization problems.

Reference Documentation

- **Release**: 2.5.0
- **Date**: Aug 12, 2020

Simulated Annealing Sampler

A dimod sampler that uses the simulated annealing algorithm.

Class

```python
class SimulatedAnnealingSampler
    Simulated annealing sampler.
    Also aliased as Neal.
```

Examples

This example solves a simple Ising problem.
>>> import neal
>>> sampler = neal.SimulatedAnnealingSampler()
>>> h = {'a': 0.0, 'b': 0.0, 'c': 0.0}
>>> J = {('a', 'b'): 1.0, ('b', 'c'): 1.0, ('a', 'c'): 1.0}
>>> response = sampler.sample_ising(h, J)
>>> for sample in response: # doctest: +SKIP
...    print(sample)
...    # doctest: +SKIP
...    {'a': -1, 'b': 1, 'c': -1}
...    {'a': -1, 'b': 1, 'c': 1}
...    {'a': 1, 'b': 1, 'c': -1}
...    {'a': 1, 'b': -1, 'c': -1}
...    {'a': 1, 'b': -1, 'c': -1}
...    {'a': 1, 'b': -1, 'c': -1}
...    {'a': -1, 'b': 1, 'c': 1}
...    {'a': -1, 'b': 1, 'c': 1}
...    {'a': -1, 'b': -1, 'c': 1}
...    {'a': -1, 'b': -1, 'c': -1}

Sampler Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimulatedAnnealingSampler.properties</td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
<tr>
<td>SimulatedAnnealingSampler.parameters</td>
<td>A dict where keys are the keyword parameters accepted by the sampler methods (allowed kwargs) and values are lists of SimulatedAnnealingSampler.properties relevant to each parameter.</td>
</tr>
</tbody>
</table>

neal.sampler.SimulatedAnnealingSampler.properties

SimulatedAnnealingSampler.properties = None
A dict containing any additional information about the sampler.

Examples

This example looks at the values set for a sampler property.

>>> import neal
>>> sampler = neal.SimulatedAnnealingSampler()
>>> sampler.properties['beta_schedule_options']
('linear', 'geometric')

Type  dict

neal.sampler.SimulatedAnnealingSampler.parameters

SimulatedAnnealingSampler.parameters = None
A dict where keys are the keyword parameters accepted by the sampler methods (allowed kwargs) and values are lists of SimulatedAnnealingSampler.properties relevant to each parameter.

See SimulatedAnnealingSampler.sample() for a description of the parameters.
Examples

This example looks at a sampler’s parameters and some of their values.

```python
>>> import neal
>>> sampler = neal.SimulatedAnnealingSampler()
>>> for kwarg in sorted(sampler.parameters):
...   print(kwarg)
beta_range
beta_schedule_type
initial_states
initial_states_generator
interrupt_function
num_reads
num_sweeps
seed
>>> sampler.parameters['beta_range']
[]
>>> sampler.parameters['beta_schedule_type']
['beta_schedule_options']
```

Type  dict

Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SimulatedAnnealingSampler.sample(bqm, ...)</code></td>
<td>Sample from a binary quadratic model using an implemented sample method.</td>
</tr>
<tr>
<td><code>SimulatedAnnealingSampler.sample_ising(h, J, ...)</code></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><code>SimulatedAnnealingSampler.sample_qubo(Q, ...)</code></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

neal.sampler.SimulatedAnnealingSampler.sample

`SimulatedAnnealingSampler.sample(bqm, beta_range=None, num_reads=None, num_sweeps=1000, beta_schedule_type='geometric', seed=None, interrupt_function=None, initial_states=None, initial_states_generator='random', **kwargs)`

Sample from a binary quadratic model using an implemented sample method.

Parameters

- **bqm** *(dimod.BinaryQuadraticModel)* – The binary quadratic model to be sampled.
- **beta_range** *(tuple, optional)* – A 2-tuple defining the beginning and end of the beta schedule, where beta is the inverse temperature. The schedule is applied linearly in beta. Default range is set based on the total bias associated with each node.
- **num_reads** *(int, optional, default=len(initial_states) or 1)* – Number of reads. Each read is generated by one run of the simulated annealing algorithm. If *num_reads* is not explicitly given, it is selected to match the number of initial states given. If initial states are not provided, only one read is performed.
- **num_sweeps** *(int, optional, default=1000)* – Number of sweeps or steps.
• **beta_schedule_type** (string, optional, default='geometric') – Beta schedule type, or how the beta values are interpolated between the given `beta_range`. Supported values are:
  - linear
  - geometric

• **seed**(int, optional) – Seed to use for the PRNG. Specifying a particular seed with a constant set of parameters produces identical results. If not provided, a random seed is chosen.

• **initial_states**(dimod.SampleSet or tuple(numpy.ndarray, dict), optional) – One or more samples, each defining an initial state for all the problem variables. Initial states are given one per read, but if fewer than `num_reads` initial states are defined, additional values are generated as specified by `initial_states_generator`. Initial states are provided either as:
  - `dimod.SampleSet`, or
  - [deprecated] tuple, where the first value is a numpy array of initial states to seed the simulated annealing runs, and the second is a dict defining a linear variable labelling. In tuple format, initial states provided are assumed to use the same vartype the BQM is using.

• **initial_states_generator**(str, 'none'/'tile'/random', optional, default='random') – Defines the expansion of `initial_states` if fewer than `num_reads` are specified:
  - "none": If the number of initial states specified is smaller than `num_reads`, raises ValueError.
  - "tile": Reuses the specified initial states if fewer than `num_reads` or truncates if greater.
  - "random": Expands the specified initial states with randomly generated states if fewer than `num_reads` or truncates if greater.

• **interrupt_function**(function, optional) – If provided, interrupt_function is called with no parameters between each sample of simulated annealing. If the function returns True, then simulated annealing will terminate and return with all of the samples and energies found so far.

**Returns** A `dimod` Response object.

**Return type** dimod.Response

**Examples**

This example runs simulated annealing on a binary quadratic model with some different input parameters.

```python
>>> import dimod
>>> import neal
... >>> sampler = neal.SimulatedAnnealingSampler()
>>> bqm = dimod.BinaryQuadraticModel({'a': .5, 'b': -.5}, {('a', 'b'): -1}, 0.0, dimod.SPIN)
>>> # Run with default parameters
>>> response = sampler.sample(bqm)
>>> # Run with specified parameters
```
>>> response = sampler.sample(bqm, seed=1234, beta_range=[0.1, 4.2],
...                            num_reads=1, num_sweeps=20,
...                            beta_schedule_type='geometric')
>>> # Reuse a seed
>>> a1 = next((sampler.sample(bqm, seed=88)).samples())['a']
>>> a2 = next((sampler.sample(bqm, seed=88)).samples())['a']
>>> a1 == a2
True

neal.sampler.SimulatedAnnealingSampler.sample_ising

SimulatedAnnealingSampler.sample_ising(h, J, **parameters)
Sample from an Ising model using the implemented sample method.
This method is inherited from the Sampler base class.
Converts the Ising model into a BinaryQuadraticModel and then calls sample().

Parameters

- **h** (dict/list) – Linear biases of the Ising problem. If a dict, should be of the form {v: bias, ...} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.
- ****kwargs – See the implemented sampling for additional keyword definitions.

Returns: SampleSet

See also:
sample(), sample_qubo()

neal.sampler.SimulatedAnnealingSampler.sample_qubo

SimulatedAnnealingSampler.sample_qubo(Q, **parameters)
Sample from a QUBO using the implemented sample method.
This method is inherited from the Sampler base class.
Converts the QUBO into a BinaryQuadraticModel and then calls sample().

Parameters

- **Q** (dict) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form {(u, v): bias, ...} where u, v, are binary-valued variables and bias is their associated coefficient.
- ****kwargs – See the implemented sampling for additional keyword definitions.

Returns: SampleSet

See also:
sample(), sample_ising()
6.7 dwave-networkx

D-Wave NetworkX is an extension of NetworkX—a Python language package for exploration and analysis of networks and network algorithms—for users of D-Wave Systems. It provides tools for working with Chimera graphs and implementations of graph-theory algorithms on the D-Wave system and other binary quadratic model samplers.

The example below generates a graph for a Chimera unit cell (eight nodes in a 4-by-2 bipartite architecture).

```python
>>> import dwave_networkx as dnx

>>> graph = dnx.chimera_graph(1, 1, 4)
```

See the documentation for more examples.

### 6.7.1 Introduction

D-Wave NetworkX provides tools for working with Chimera and Pegasus graphs and implementations of graph-theory algorithms on the D-Wave system and other binary quadratic model samplers; for example, functions such as `draw_chimera()` provide easy visualization for Chimera graphs; functions such as `maximum_cut()` or `min_vertex_cover()` provide graph algorithms useful to optimization problems that fit well with the D-Wave system.

Like the D-Wave system, all other supported samplers must have `sample_qubo` and `sample_ising` methods for solving Ising and QUBO models and return an iterable of samples in order of increasing energy. You can set a default sampler using the `set_default_sampler()` function.

- For an introduction to quantum processing unit (QPU) topologies such as the Chimera and Pegasus graphs, see Topology.
- For an introduction to binary quadratic models (BQMs), see Binary Quadratic Models.
- For an introduction to samplers, see Samplers and Composites.

### Example

Below you can see how to create Chimera graphs implemented in the D-Wave 2X and D-Wave 2000Q systems:

```python
import dwave_networkx as dnx

# D-Wave 2X
C = dnx.chimera_graph(12, 12, 4)

# D-Wave 2000Q
C = dnx.chimera_graph(16, 16, 4)
```

### 6.7.2 Reference Documentation

**Release** 2.5.0

**Date** Aug 12, 2020
Algorithms

Implementations of graph-theory algorithms on the D-Wave system and other binary quadratic model samplers.

Canonicalization

\[ \text{canonical_chimera_labeling}(G[, t]) \]

Returns a mapping from the labels of \( G \) to chimera-indexed labeling.

**canonical_chimera_labeling** \((G, t=None)\)

Returns a mapping from the labels of \( G \) to chimera-indexed labeling.

**Parameters**

- \( G \) (*NetworkX graph*) A Chimera-structured graph.
- \( t \) (*int (optional, default 4)*) Size of the shore within each Chimera tile.

**Returns**

- \( \text{chimera_indices} \) A mapping from the current labels to a 4-tuple of Chimera indices.

**Return type** dict

Clique

A clique in an undirected graph \( G = (V, E) \) is a subset of the vertex set such that for every two vertices in \( C \) there exists an edge connecting the two.

\[ \text{maximum_clique}(G[, \text{sampler}, \text{lagrange}]) \]

Returns an approximate maximum clique.

\[ \text{clique_number}(G[, \text{sampler}, \text{lagrange}]) \]

Returns the number of vertices in the maximum clique of a graph.

\[ \text{is_clique}(G, \text{clique_nodes}) \]

Determines whether the given nodes form a clique.

**maximum_clique** \((G, \text{sampler=None}, \text{lagrange}=2.0, **\text{sampler_args})\)

Returns an approximate maximum clique. A clique in an undirected graph, \( G = (V, E) \), is a subset of the vertex set \( C \subseteq V \) such that for every two vertices in \( C \) there exists an edge connecting the two. This is equivalent to saying that the subgraph induced by \( C \) is complete (in some cases, the term clique may also refer to the subgraph). A maximum clique is a clique of the largest possible size in a given graph.
This function works by finding the maximum independent set of the complement graph of the given graph \( G \) which is equivalent to finding maximum clique. It defines a QUBO with ground states corresponding to a maximum weighted independent set and uses the sampler to sample from it.

**Parameters**

- \( G \) (NetworkX graph) – The graph on which to find a maximum clique.
- \texttt{sampler} – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the \texttt{set_default_sampler} function.
- \texttt{lagrange} (optional (default 2)) – Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).
- \texttt{sampler_args} – Additional keyword parameters are passed to the sampler.

**Returns**

- \texttt{clique_nodes} – List of nodes that form a maximum clique, as determined by the given sampler.

**Return type**

- list

**Notes**

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

**References**

- Maximum Clique on Wikipedia
- Independent Set on Wikipedia
- QUBO on Wikipedia

dwave_networkx.clique_number

clique_number \((G, \texttt{sampler=None}, \texttt{lagrange=2.0}, **\texttt{sampler_args})\)  
 Returns the number of vertices in the maximum clique of a graph. A maximum clique is a clique of the largest possible size in a given graph. The clique number \( \omega(G) \) of a graph \( G \) is the number of vertices in a maximum clique in \( G \). The intersection number of \( G \) is the smallest number of cliques that together cover all edges of \( G \).

This function works by finding the maximum independent set of the complement graph of the given graph \( G \) which is equivalent to finding maximum clique. It defines a QUBO with ground states corresponding to a maximum weighted independent set and uses the sampler to sample from it.

**Parameters**

- \( G \) (NetworkX graph) – The graph on which to find a maximum clique.
- \texttt{sampler} – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in
order of increasing energy. If no sampler is provided, one must be provided using the 
\texttt{set_default_sampler} function.

\begin{itemize}
  \item \texttt{lagrange (optional (default 2))} -- Lagrange parameter to weight constraints (no
edges within set) versus objective (largest set possible).
  \item \texttt{sampler_args} -- Additional keyword parameters are passed to the sampler.
\end{itemize}

\textbf{Returns} \texttt{clique_nodes} -- List of nodes that form a maximum clique, as determined by the given
sampler.

\textbf{Return type} \texttt{list}

\section*{Notes}

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the
quality of the returned sample.

\section*{References}

Maximum Clique on Wikipedia

dwave_networkx.is_clique

\texttt{is_clique}(G, clique_nodes)

Determines whether the given nodes form a clique.

A clique is a subset of nodes of an undirected graph such that every two distinct nodes in the clique are adjacent.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{G (NetworkX graph)} -- The graph on which to check the clique nodes.
  \item \texttt{clique_nodes (list)} -- List of nodes that form a clique, as determined by the given
sampler.
\end{itemize}

\textbf{Returns} \texttt{is_clique} -- True if clique_nodes forms a clique.

\textbf{Return type} \texttt{bool}

\section*{Example}

This example checks two sets of nodes, both derived from a single Chimera unit cell, for an independent set.
The first set is the horizontal tile’s nodes; the second has nodes from the horizontal and vertical tiles.

\begin{verbatim}
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> dnx.is_clique(G, [0, 1, 2, 3])
False
>>> dnx.is_clique(G, [0, 4])
True
\end{verbatim}
## Coloring

Graph coloring is the problem of assigning a color to the vertices of a graph in a way that no adjacent vertices have the same color.

### Example

The map-coloring problem is to assign a color to each region of a map (represented by a vertex on a graph) such that any two regions sharing a border (represented by an edge of the graph) have different colors.

![Coloring a map of Canada with four colors.](image)

**Fig. 17:** Coloring a map of Canada with four colors.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_vertex_coloring(G, coloring)</code></td>
<td>Determines whether the given coloring is a vertex coloring of graph G.</td>
</tr>
<tr>
<td><code>min_vertex_color(G[, sampler, chromatic_lb, ...])</code></td>
<td>Returns an approximate minimum vertex coloring.</td>
</tr>
<tr>
<td><code>min_vertex_color_qubo(G[, chromatic_lb, ...])</code></td>
<td>Return a QUBO with ground states corresponding to a minimum vertex coloring.</td>
</tr>
<tr>
<td><code>vertex_color(G, colors[, sampler])</code></td>
<td>Returns an approximate vertex coloring.</td>
</tr>
<tr>
<td><code>vertex_color_qubo(G, colors)</code></td>
<td>Return the QUBO with ground states corresponding to a vertex coloring.</td>
</tr>
</tbody>
</table>

### `is_vertex_coloring` in `dwave_networkx.algorithms.coloring`

`is_vertex_coloring(G, coloring)`

Determines whether the given coloring is a vertex coloring of graph G.

**Parameters**

- `G (NetworkX graph)` – The graph on which the vertex coloring is applied.
- `coloring (dict)` – A coloring of the nodes of G. Should be a dict of the form `{node: color, ...}`.

**Returns** `is_vertex_coloring` – True if the given coloring defines a vertex coloring; that is, no two adjacent vertices share a color.

**Return type** `bool`
Example

This example colors checks two colorings for a graph, G, of a single Chimera unit cell. The first uses one color (0) for the four horizontal qubits and another (1) for the four vertical qubits, in which case there are no adjacencies; the second coloring swaps the color of one node.

```python
>>> G = dnx.chimera_graph(1,1,4)
>>> colors = {0: 0, 1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 1, 7: 1}
>>> dnx.is_vertex_coloring(G, colors)
True
>>> colors[4]=0
>>> dnx.is_vertex_coloring(G, colors)
False
```

dwave_networkx.algorithms.coloring.min_vertex_color

`min_vertex_color(G, sampler=None, chromatic_lb=None, chromatic_ub=None, **sampler_args)`

Returns an approximate minimum vertex coloring.

Vertex coloring is the problem of assigning a color to the vertices of a graph in a way that no adjacent vertices have the same color. A minimum vertex coloring is the problem of solving the vertex coloring problem using the smallest number of colors.

Defines a QUBO [DWMP] with ground states corresponding to minimum vertex colorings and uses the sampler to sample from it.

Parameters

- `G` (*NetworkX graph*) – The graph on which to find a minimum vertex coloring.
- `sampler` – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a `sample_qubo` and `sample_ising` method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- `chromatic_lb` (*int*, *optional*) – A lower bound on the chromatic number. If one is not provided, a bound is calculated.
- `chromatic_ub` (*int*, *optional*) – An upper bound on the chromatic number. If one is not provided, a bound is calculated.
- `sampler_args` – Additional keyword parameters are passed to the sampler.

Returns

`coloring` – A coloring for each vertex in G such that no adjacent nodes share the same color. A dict of the form `{node: color, ...}`

Return type

`dict`

References

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.
**min_vertex_color_qubo**

Return a QUBO with ground states corresponding to a minimum vertex coloring.

Vertex coloring is the problem of assigning a color to the vertices of a graph in a way that no adjacent vertices have the same color. A minimum vertex coloring is the problem of solving the vertex coloring problem using the smallest number of colors.

Defines a QUBO [DWMP] with ground states corresponding to minimum vertex colorings and uses the sampler to sample from it.

**Parameters**

- **G** (*NetworkX graph*) – The graph on which to find a minimum vertex coloring.
- **chromatic_lb** (*int, optional*) – A lower bound on the chromatic number. If one is not provided, a bound is calculated.
- **chromatic_ub** (*int, optional*) – An upper bound on the chromatic number. If one is not provided, a bound is calculated.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

**Returns**

QUBO – The QUBO with ground states corresponding to minimum colorings of the graph. The QUBO variables are labelled \((v, c)\) where \(v\) is a node in \(G\) and \(c\) is a color. In the ground state of the QUBO, a variable \((v, c)\) has value 1 if \(v\) should be colored \(c\) in a valid coloring.

**Return type**

dict

**vertex_color**

Returns an approximate vertex coloring.

Vertex coloring is the problem of assigning a color to the vertices of a graph in a way that no adjacent vertices have the same color.

Defines a QUBO [DWMP] with ground states corresponding to valid vertex colorings and uses the sampler to sample from it.

**Parameters**

- **G** (*NetworkX graph*) – The graph on which to find a minimum vertex coloring.
- **colors** (*int/sequence*) – The colors. If an int, the colors are labelled \([0, n)\). The number of colors must be greater or equal to the chromatic number of the graph.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the set_default_sampler function.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

**Returns**

coloring – A coloring for each vertex in \(G\) such that no adjacent nodes share the same color. A dict of the form \{node: color, \ldots\}

**Return type**

dict
References

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

`dwave_networkx.algorithms.coloring.vertex_color_qubo`

`vertex_color_qubo(G, colors)`

Return the QUBO with ground states corresponding to a vertex coloring.

If \( V \) is the set of nodes, \( E \) is the set of edges and \( C \) is the set of colors the resulting qubo will have:

- \(|V| \cdot |C| \) variables/nodes
- \(|V| \cdot |C| \cdot (|C| - 1)/2 + |E| \cdot |C| \) interactions/edges

The QUBO has ground energy \(-|V|\) and an infeasible gap of 1.

Parameters

- \( G \) (NetworkX graph) – The graph on which to find a minimum vertex coloring.
- \( colors \) (int/sequence) – The colors. If an int, the colors are labelled \([0, n)\). The number of colors must be greater or equal to the chromatic number of the graph.

Returns QUBO – The QUBO with ground states corresponding to valid colorings of the graph. The QUBO variables are labelled \((v, c)\) where \( v \) is a node in \( G \) and \( c \) is a color. In the ground state of the QUBO, a variable \((v, c)\) has value 1 if \( v \) should be colored \( c \) in a valid coloring.

Return type dict

Cover

Vertex covering is the problem of finding a set of vertices such that all the edges of the graph are incident to at least one of the vertices in the set.

![Cover for a Chimera unit cell](image)

Fig. 18: Cover for a Chimera unit cell: the nodes of both the blue set of vertices (the horizontal tile of the Chimera unit cell) and the red set (vertical tile) connect to all 16 edges of the graph.

`min_weighted_vertex_cover(G[, weight, ...])` Returns an approximate minimum weighted vertex cover.

`min_vertex_cover(G[, sampler, lagrange])` Returns an approximate minimum vertex cover.

Continued on next page

6.7. dwave-networkx
Table 120 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_vertex_cover(G, vertex_cover)</code></td>
<td>Determines whether the given set of vertices is a vertex cover of graph G.</td>
</tr>
</tbody>
</table>

`dwave_networkx.algorithms.cover.min_weighted_vertex_cover`

`min_weighted_vertex_cover(G, weight=None, sampler=None, lagrange=2.0, **sampler_args)`

Returns an approximate minimum weighted vertex cover.

Defines a QUBO with ground states corresponding to a minimum weighted vertex cover and uses the sampler to sample from it.

A vertex cover is a set of vertices such that each edge of the graph is incident with at least one vertex in the set. A minimum weighted vertex cover is the vertex cover of minimum total node weight.

**Parameters**

- `G` *(NetworkX graph)*
- `weight` *(string, optional (default None))* – If None, every node has equal weight. If a string, use this node attribute as the node weight. A node without this attribute is assumed to have max weight.
- `sampler` – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- `lagrange` *(optional (default 2))* – Lagrange parameter to weight constraints versus objective.
- `sampler_args` – Additional keyword parameters are passed to the sampler.

**Returns** `vertex_cover` – List of nodes that the form a the minimum weighted vertex cover, as determined by the given sampler.

**Return type** list

**Notes**

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

- https://en.wikipedia.org/wiki/Vertex_cover

**References**

Based on the formulation presented in [AL]
min_vertex_cover \( (G, \text{sampler}=\text{None}, \text{lagrange}=2.0, **\text{sampler\_args}) \)

Returns an approximate minimum vertex cover.

Defines a QUBO with ground states corresponding to a minimum vertex cover and uses the sampler to sample from it.

A vertex cover is a set of vertices such that each edge of the graph is incident with at least one vertex in the set. A minimum vertex cover is the vertex cover of smallest size.

**Parameters**

- \( G \) (NetworkX graph) – The graph on which to find a minimum vertex cover.
- \text{sampler} – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the \text{set\_default\_sampler} function.
- \text{lagrange} (optional (default 2)) – Lagrange parameter to weight constraints versus objective.
- \text{sampler\_args} – Additional keyword parameters are passed to the sampler.

**Returns**

- \text{vertex\_cover} – List of nodes that form a minimum vertex cover, as determined by the given sampler.

**Return type** list

**Examples**

This example uses a sampler from \text{dimod} to find a minimum vertex cover for a Chimera unit cell. Both the horizontal (vertices 0,1,2,3) and vertical (vertices 4,5,6,7) tiles connect to all 16 edges, so repeated executions can return either set.

```python
>>> import dwave_networkx as dnx
>>> import dimod
>>> sampler = dimod.ExactSolver()  # small testing sampler
>>> G = dnx.chimera_graph(1, 1, 4)
>>> G.remove_node(7)  # to give a unique solution
>>> dnx.min_vertex_cover(G, sampler, lagrange=2.0)
[4, 5, 6]
```

**Notes**

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

**References**

https://en.wikipedia.org/wiki/Vertex_cover
**is_vertex_cover** *(G, vertex_cover)*

Determines whether the given set of vertices is a vertex cover of graph G.

A vertex cover is a set of vertices such that each edge of the graph is incident with at least one vertex in the set.

**Parameters**

- **G** *(NetworkX graph)* – The graph on which to check the vertex cover.
- **vertex_cover** – Iterable of nodes.

**Returns**

* is_cover – True if the given iterable forms a vertex cover.

**Return type**

* bool

**Examples**

This example checks two covers for a graph, G, of a single Chimera unit cell. The first uses the set of the four horizontal qubits, which do constitute a cover; the second set removes one node.

```python
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> cover = [0, 1, 2, 3]
>>> dnx.is_vertex_cover(G, cover)
True
>>> cover = [0, 1, 2]
>>> dnx.is_vertex_cover(G, cover)
False
```

**Elimination Ordering**

Many algorithms for NP-hard problems are exponential in treewidth. However, finding a lower bound on treewidth is in itself NP-complete. [GD] describes a branch-and-bound algorithm for computing the treewidth of an undirected graph by searching in the space of *perfect elimination ordering* of vertices of the graph.

A *clique* of a graph is a fully-connected subset of vertices; that is, every pair of vertices in the clique share an edge. A *simplicial* vertex is one whose neighborhood induces a clique. A perfect elimination ordering is an ordering of vertices 1..n such that any vertex i is simplicial for the subset of vertices i..n.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chimera_elimination_order(m, n, t)</td>
<td>Provides a variable elimination order for a Chimera graph.</td>
</tr>
<tr>
<td>elimination_order_width(G, order)</td>
<td>Calculates the width of the tree decomposition induced by a variable elimination order.</td>
</tr>
<tr>
<td>is_almost_simplicial(G, n)</td>
<td>Determines whether a node n in G is almost simplicial.</td>
</tr>
<tr>
<td>is_simplicial(G, n)</td>
<td>Determines whether a node n in G is simplicial.</td>
</tr>
<tr>
<td>max_cardinality_heuristic(G)</td>
<td>Computes an upper bound on the treewidth of graph G based on the max-cardinality heuristic for the elimination ordering.</td>
</tr>
<tr>
<td>minor_min_width(G)</td>
<td>Computes a lower bound for the treewidth of graph G.</td>
</tr>
<tr>
<td>min_fill_heuristic(G)</td>
<td>Computes an upper bound on the treewidth of graph G based on the min-fill heuristic for the elimination ordering.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 121 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>min_width_heuristic(G)</code></td>
<td>Computes an upper bound on the treewidth of graph G based on the min-width heuristic for the elimination ordering.</td>
</tr>
<tr>
<td><code>pegasus_elimination_order(n[, coordinates])</code></td>
<td>Provides a variable elimination order for the Pegasus graph.</td>
</tr>
<tr>
<td><code>treewidth_branch_and_bound(G[, ...])</code></td>
<td>Computes the treewidth of graph G and a corresponding perfect elimination ordering.</td>
</tr>
</tbody>
</table>

`dwave_networkx.algorithms.elimination_ordering.chimera_elimination_order`

`chimera_elimination_order(m, n=None, t=None)`

Provides a variable elimination order for a Chimera graph.

A graph defined by `chimera_graph(m,n,t)` has treewidth $\max(m,n) \times t$. This function outputs a variable elimination order inducing a tree decomposition of that width.

**Parameters**

- `m` (int) – Number of rows in the Chimera lattice.
- `n` (int (optional, default m)) – Number of columns in the Chimera lattice.
- `t` (int (optional, default 4)) – Size of the shore within each Chimera tile.

**Returns**

- `order` – An elimination order that induces the treewidth of `chimera_graph(m,n,t)`.

**Return type** list

**Examples**

```python
>>> G = dnx.chimera_elimination_order(1, 1, 4)  # a single Chimera tile
```

`dwave_networkx.algorithms.elimination_ordering.elimination_order_width`

`elimination_order_width(G, order)`

Calculates the width of the tree decomposition induced by a variable elimination order.

**Parameters**

- `G` (NetworkX graph) – The graph on which to compute the width of the tree decomposition.
- `order` (list) – The elimination order. Must be a list of all of the variables in G.

**Returns**

- `treewidth` – The width of the tree decomposition induced by order.

**Return type** int

**Examples**

This example computes the width of the tree decomposition for the $K_4$ complete graph induced by an elimination order found through the min-width heuristic.
$$\textbf{dwave}\_\text{networkx}\_\text{algorithms}\_\text{elimination}\_\text{ordering}.\text{is}_\text{almost}_\text{simplicial}\ (G, n)$$

Determines whether a node n in G is almost simplicial.

Parameters

- G (NetworkX graph) – The graph on which to check whether node n is almost simplicial.
- n (node) – A node in graph G.

Returns is_almost_simplicial – True if all but one of its neighbors induce a clique

Return type bool

Examples

This example checks whether node 0 is simplicial or almost simplicial for a $K_5$ complete graph with one edge removed.

```python
>>> K_5 = nx.complete_graph(5)
>>> K_5.remove_edge(1, 3)
>>> dnx.is_simplicial(K_5, 0)
False
>>> dnx.is_almost_simplicial(K_5, 0)
True
```

$$\textbf{dwave}\_\text{networkx}\_\text{algorithms}\_\text{elimination}\_\text{ordering}.\text{is}_\text{simplicial}\ (G, n)$$

Determines whether a node n in G is simplicial.

Parameters

- G (NetworkX graph) – The graph on which to check whether node n is simplicial.
- n (node) – A node in graph G.

Returns is_simplicial – True if its neighbors form a clique.

Return type bool

Examples

This example checks whether node 0 is simplicial for two graphs: G, a single Chimera unit cell, which is bipartite, and K_5, the $K_5$ complete graph.
```python
>>> G = dnx.chimera_graph(1, 1, 4)
>>> K_5 = nx.complete_graph(5)
>>> dnx.is_simplicial(G, 0)
False
>>> dnx.is_simplicial(K_5, 0)
True
```

dwave_networkx.algorithms.elimination_ordering.max_cardinality_heuristic

**max_cardinality_heuristic**(*G*)

Computes an upper bound on the treewidth of graph *G* based on the max-cardinality heuristic for the elimination ordering.

**Parameters**

*G* (*NetworkX graph*) — The graph on which to compute an upper bound for the treewidth.

**Returns**

- **treewidth_upper_bound** (*int*) — An upper bound on the treewidth of the graph *G*.
- **order** (*list*) — An elimination order that induces the treewidth.

**Examples**

This example computes an upper bound for the treewidth of the $K_4$ complete graph.

```python
>>> K_4 = nx.complete_graph(4)
>>> tw, order = dnx.max_cardinality_heuristic(K_4)
```

**References**

Based on the algorithm presented in [GD]

dwave_networkx.algorithms.elimination_ordering.minor_min_width

**minor_min_width**(*G*)

Computes a lower bound for the treewidth of graph *G*.

**Parameters**

*G* (*NetworkX graph*) — The graph on which to compute a lower bound on the treewidth.

**Returns**

- **lb** (*int*) — A lower bound on the treewidth.

**Return type**

*int*

**Examples**

This example computes a lower bound for the treewidth of the $K_7$ complete graph.

```python
>>> K_7 = nx.complete_graph(7)
>>> dnx.minor_min_width(K_7)
6
```
References

Based on the algorithm presented in [GD]

dwave_networkx.algorithms.elimination_ordering.min_fill_heuristic

min_fill_heuristic(G)
Computes an upper bound on the treewidth of graph G based on the min-fill heuristic for the elimination ordering.

Parameters G (NetworkX graph) – The graph on which to compute an upper bound for the treewidth.

Returns

- treewidth_upper_bound (int) – An upper bound on the treewidth of the graph G.
- order (list) – An elimination order that induces the treewidth.

Examples

This example computes an upper bound for the treewidth of the $K_4$ complete graph.

```python
>>> K_4 = nx.complete_graph(4)
>>> tw, order = dnx.min_fill_heuristic(K_4)
```

References

Based on the algorithm presented in [GD]

dwave_networkx.algorithms.elimination_ordering.min_width_heuristic

min_width_heuristic(G)
Computes an upper bound on the treewidth of graph G based on the min-width heuristic for the elimination ordering.

Parameters G (NetworkX graph) – The graph on which to compute an upper bound for the treewidth.

Returns

- treewidth_upper_bound (int) – An upper bound on the treewidth of the graph G.
- order (list) – An elimination order that induces the treewidth.

Examples

This example computes an upper bound for the treewidth of the $K_4$ complete graph.

```python
>>> K_4 = nx.complete_graph(4)
>>> tw, order = dnx.min_width_heuristic(K_4)
```
References

Based on the algorithm presented in [GD]

dwave_networkx.algorithms.elimination_ordering.pegasus_elimination_order

pegasus_elimination_order \( (n, \text{coordinates}=\text{False}) \)
Provides a variable elimination order for the Pegasus graph.

The treewidth of a Pegasus graph \( P(n) \) is lower-bounded by \( 12n-11 \) and upper bounded by \( 12-4 \) \([\text{bbrr}]\).

Simple pegasus variable elimination order rules:
• eliminate vertical qubits, one column at a time
• eliminate horizontal qubits in each column once their adjacent vertical qubits have been eliminated

Parameters
• \( n \) (\text{int}) – The size parameter for the Pegasus lattice.
• \( \text{coordinates} \) (\text{bool}, \text{optional (default False)}) – If True, the elimination order is given in terms of 4-term Pegasus coordinates, otherwise given in linear indices.

Returns \( \text{order} \) – An elimination order that provides an upper bound on the treewidth.

Return type \( \text{list} \)

dwave_networkx.algorithms.elimination_ordering.treewidth_branch_and_bound

treewidth_branch_and_bound \((G, \text{elimination_order}=None, \text{treewidth_upperbound}=None)\)
Computes the treewidth of graph \( G \) and a corresponding perfect elimination ordering.

Alogorithm based on [GD].

Parameters
• \( G \) (\text{NetworkX graph}) – The graph on which to compute the treewidth and perfect elimination ordering.
• \( \text{elimination_order} \) (\text{list (optional, Default None)}) – An elimination order used as an initial best-known order. If a good order is provided, it may speed up computation. If not provided, the initial order is generated using the min-fill heuristic.
• \( \text{treewidth_upperbound} \) (\text{int (optional, Default None)}) – An upper bound on the treewidth. Note that using this parameter can result in no returned order.

Returns
• \( \text{treewidth} \) (\text{int}) – The treewidth of graph \( G \).
• \( \text{order} \) (\text{list}) – An elimination order that induces the treewidth.

Examples

This example computes the treewidth for the \( K_7 \) complete graph using an optionally provided elimination order (a sequential ordering of the nodes, arbitrarily chosen).
```python
>>> K_7 = nx.complete_graph(7)
>>> dnx.treewidth_branch_and_bound(K_7, [0, 1, 2, 3, 4, 5, 6])
(6, [0, 1, 2, 3, 4, 5, 6])
```

## References

### Markov Networks

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sample_markov_network</code></td>
<td>Samples from a markov network using the provided sampler.</td>
</tr>
<tr>
<td><code>markov_network_bqm</code></td>
<td>Construct a binary quadratic model for a markov network.</td>
</tr>
</tbody>
</table>

### `dnx.markov_network(potentials)`

Sample a Markov Network. Each element in the dictionary `potentials` is a dict containing two keys: `0` and `1`. The value corresponding to each key is another dictionary. This function is used to create a Markov Network for a given set of potentials.

```python
>>> import dimod
... potentials = {('a', 'b'): {(0, 0): -1,
... (0, 1): .5,
... (1, 0): .5,
... (1, 1): 2}}
>>> MN = dnx.markov_network(potentials)
```
```python
>>> sampler = dimod.ExactSolver()
>>> samples = dnx.sample_markov_network(MN, sampler)
>>> samples[0]  # doctest: +SKIP
{'a': 0, 'b': 0}
```

```python
>>> import dimod
...
>>> potentials = {('a', 'b'): {(0, 0): -1,
... (0, 1): .5,
... (1, 0): .5,
... (1, 1): 2},
... ('b', 'c'): {(0, 0): -9,
... (0, 1): 1.2,
... (1, 0): 7.2,
... (1, 1): 5}}
>>> MN = dnx.markov_network(potentials)
>>> sampler = dimod.ExactSolver()
>>> samples = dnx.sample_markov_network(MN, sampler, fixed_variables={'b': 0})
>>> samples[0]  # doctest: +SKIP
{'a': 0, 'c': 0, 'b': 0}
```

**Notes**

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

### `dwave_networkx.algorithms.markov.markov_network_bqm`

#### `markov_network_bqm(MN)`

Construct a binary quadratic model for a markov network.

- **Parameters**
  - `G (NetworkX graph)` – A Markov Network as returned by `markov_network()`
  - `bqm` – A binary quadratic model.

  **Returns**
  - `dimod.BinaryQuadraticModel`

### Matching

A matching is a subset of graph edges in which no vertex occurs more than once.
Fig. 19: A matching for a Chimera unit cell: no vertex is incident to more than one edge in the set of blue edges

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>min_maximal_matching(G[, sampler])</code></td>
<td>Returns an approximate minimum maximal matching.</td>
</tr>
<tr>
<td><code>is_matching(edges)</code></td>
<td>Determines whether the given set of edges is a matching.</td>
</tr>
<tr>
<td><code>is_maximal_matching(G, matching)</code></td>
<td>Determines whether the given set of edges is a maximal matching.</td>
</tr>
</tbody>
</table>

**Parameters**

- **G** *(NetworkX graph)* – The graph on which to find a minimum maximal matching.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

**Returns**

- **matching** – A minimum maximal matching of the graph.

**Return type** set

**Example**

This example uses a sampler from dimod to find a minimum maximal matching for a Chimera unit cell.

```python
>>> import dimod
>>> sampler = dimod.ExactSolver()
>>> G = dnx.chimera_graph(1, 1, 4)
>>> matching = dnx.min_maximal_matching(G, sampler)
```

**Notes**
**is_matching** *(edges)*

Determines whether the given set of edges is a matching.

A matching is a subset of edges in which no node occurs more than once.

**Parameters**

- **edges** *(iterable)* – A iterable of edges.

**Returns**

- **is_matching** – True if the given edges are a matching.

**Return type**

- bool

**Example**

This example checks two sets of edges, both derived from a single Chimera unit cell, for a matching. Because every node in a Chimera unit cell connects to four other nodes in the cell, the first set, which contains all the edges, repeats each node 4 times; the second is a subset of those edges found using the `min_maximal_matching()` function.

```python
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> dnx.is_matching(G.edges())
False
>>> dnx.is_matching({(0, 4), (1, 5), (2, 7), (3, 6)})
True
```

---

**is_maximal_matching** *(G, matching)*

Determines whether the given set of edges is a maximal matching.

A matching is a subset of edges in which no node occurs more than once. The cardinality of a matching is the number of matched edges. A maximal matching is one where one cannot add any more edges without violating the matching rule.

**Parameters**

- **G** *(NetworkX graph)* – The graph on which to check the maximal matching.
- **edges** *(iterable)* – A iterable of edges.

**Returns**

- **is_matching** – True if the given edges are a maximal matching.

**Return type**

- bool

**Example**

This example checks two sets of edges, both derived from a single Chimera unit cell, for a matching. The first set (a matching) is a subset of the second, which was found using the `min_maximal_matching()` function.

```python
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> dnx.is_matching({(0, 4), (1, 5), (2, 7), (3, 6)})
True
>>> dnx.is_maximal_matching(G,{(0, 4), (2, 7)})
```

(continues on next page)
Maximum Cut

A maximum cut is a subset of a graph’s vertices such that the number of edges between this subset and the remaining vertices is as large as possible.

![Diagram of a maximum cut](image)

Fig. 20: Maximum cut for a Chimera unit cell: the blue line around the subset of nodes \{4, 5, 6, 7\} cuts 16 edges; adding or removing a node decreases the number of edges between the two complementary subsets of the graph.

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum_cut(G[, sampler])</td>
<td>Returns an approximate maximum cut.</td>
</tr>
<tr>
<td>weighted_maximum_cut(G[, sampler])</td>
<td>Returns an approximate weighted maximum cut.</td>
</tr>
</tbody>
</table>

**dwave_networkx.algorithms.max_cut.maximum_cut**

**maximum_cut** *(G, sampler=None, **sampler_args]*)  
Returns an approximate maximum cut.

Defines an Ising problem with ground states corresponding to a maximum cut and uses the sampler to sample from it.

A maximum cut is a subset \( S \) of the vertices of \( G \) such that the number of edges between \( S \) and the complementary subset is as large as possible.

**Parameters**

- **G** *(NetworkX graph)* – The graph on which to find a maximum cut.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’
and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in
order of increasing energy. If no sampler is provided, one must be provided using the
set_default_sampler function.

- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns S – A maximum cut of G.

Return type set

Example

This example uses a sampler from dimod to find a maximum cut for a graph of a Chimera unit cell created using the
chimera_graph() function.

```python
>>> import dimod
... >>> sampler = dimod.SimulatedAnnealingSampler()
>>> G = dnx.chimera_graph(1, 1, 4)
>>> cut = dnx.maximum_cut(G, sampler)
```

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the
quality of the returned sample.

**dwave_networkx.algorithms.max_cut.weighted_maximum_cut**

**weighted_maximum_cut(G, sampler=None, **sampler_args)**

Returns an approximate weighted maximum cut.

Defines an Ising problem with ground states corresponding to a weighted maximum cut and uses the sampler to
sample from it.

A weighted maximum cut is a subset S of the vertices of G that maximizes the sum of the edge weights between
S and its complementary subset.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a weighted maximum cut. Each edge
  in G should have a numeric **weight** attribute.

- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from
  low energy states in models defined by an Ising equation or a Quadratic Unconstrained
  Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’
  and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in
  order of increasing energy. If no sampler is provided, one must be provided using the
  set_default_sampler function.

- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns S – A maximum cut of G.

Return type set
Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

Independent Set

An independent set is a set of a graph’s vertices with no edge connecting any of its member pairs.

Fig. 21: Independent sets for a Chimera unit cell: the nodes of both the blue set of vertices (the horizontal tile of the Chimera unit cell) and the red set (vertical tile) are independent sets of the graph, with no blue node adjacent to another blue node and likewise for red nodes.

Maximum Weighted Independent Set

Returns an approximate maximum weighted independent set.

```python
maximum_weighted_independent_set(G, weight=None, sampler=None, lagrange=2.0, **sampler_args)
```

Defines a QUBO with ground states corresponding to a maximum weighted independent set and uses the sampler to sample from it.

An independent set is a set of nodes such that the subgraph of G induced by these nodes contains no edges. A maximum independent set is an independent set of maximum total node weight.

Parameters

- `G (NetworkX graph)` – The graph on which to find a maximum cut weighted independent set.
- `weight (string, optional (default None))` – If None, every node has equal weight. If a string, use this node attribute as the node weight. A node without this attribute is assumed to have max weight.
- `sampler` – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained
Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the set_default_sampler function.

- **lagrange** *(optional (default 2))* – Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).
- **sampler_args** – Additional keyword parameters are passed to the sampler.

**Returns**  
**indep_nodes** – List of nodes that form a maximum weighted independent set, as determined by the given sampler.

**Return type**  
**list**

**Notes**
Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

**References**
Independent Set on Wikipedia
QUBO on Wikipedia

dwave_networkx.maximum_independent_set

**maximum_independent_set** *(G, sampler=None, lagrange=2.0, **sampler_args)*

Returns an approximate maximum independent set.

Defines a QUBO with ground states corresponding to a maximum independent set and uses the sampler to sample from it.

An independent set is a set of nodes such that the subgraph of G induced by these nodes contains no edges. A maximum independent set is an independent set of largest possible size.

**Parameters**
- **G** *(NetworkX graph)* – The graph on which to find a maximum cut independent set.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the set_default_sampler function.
- **lagrange** *(optional (default 2))* – Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).
- **sampler_args** – Additional keyword parameters are passed to the sampler.

**Returns**  
**indep_nodes** – List of nodes that form a maximum independent set, as determined by the given sampler.

**Return type**  
**list**
Example

This example uses a sampler from `dimod` to find a maximum independent set for a graph of a Chimera unit cell created using the `chimera_graph()` function.

```python
>>> import dimod
>>> sampler = dimod.SimulatedAnnealingSampler()
>>> G = dnx.chimera_graph(1, 1, 4)
>>> indep_nodes = dnx.maximum_independent_set(G, sampler)
```

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

References

Independent Set on Wikipedia
QUBO on Wikipedia

`dwave_networkx.is_independent_set`

`is_independent_set(G, indep_nodes)`

Determines whether the given nodes form an independent set.

An independent set is a set of nodes such that the subgraph of $G$ induced by these nodes contains no edges.

**Parameters**

- $G$ ([NetworkX graph](https://networkx.github.io/)) – The graph on which to check the independent set.
- `indep_nodes` ([list](https://docs.python.org/3/library/stdtypes.html#list)) – List of nodes that form a maximum independent set, as determined by the given sampler.

**Returns**

- `is_independent` – True if indep_nodes form an independent set.
- **Return type** `bool`

Example

This example checks two sets of nodes, both derived from a single Chimera unit cell, for an independent set. The first set is the horizontal tile’s nodes; the second has nodes from the horizontal and vertical tiles.

```python
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> dnx.is_independent_set(G, [0, 1, 2, 3])
True
>>> dnx.is_independent_set(G, [0, 4])
False
```

Helper Functions
Return the QUBO with ground states corresponding to a maximum weighted independent set.

Parameters

- \( G \) (NetworkX graph) -
- \( \text{weight} \) (string, optional (default None)) - If None, every node has equal weight. If a string, use this node attribute as the node weight. A node without this attribute is assumed to have max weight.
- \( \text{lagrange} \) (optional (default 2)) - Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).

Returns QUBO - The QUBO with ground states corresponding to a maximum weighted independent set.

Return type dict

Examples

```python
>>> from dwave_networkx.algorithms.independent_set import maximum_weighted_independent_set_qubo
... >>> G = nx.path_graph(3)
... >>> Q = maximum_weighted_independent_set_qubo(G, weight='weight', lagrange=2.0)
... >>> Q[(0, 0)]
-1.0
... >>> Q[(1, 1)]
-1.0
... >>> Q[(0, 1)]
2.0
```

Social

A signed social network graph is a graph whose signed edges represent friendly/hostile interactions between vertices.

Returns an approximate set of frustrated edges and a bicoloring.

A signed social network graph is a graph whose signed edges represent friendly/hostile interactions between vertices.

```python
>>> from dwave_networkx.algorithms.independent_set import maximum_weighted_independent_set_qubo
... >>> G = nx.path_graph(3)
... >>> Q = maximum_weighted_independent_set_qubo(G, weight='weight', lagrange=2.0)
... >>> Q[(0, 0)]
-1.0
... >>> Q[(1, 1)]
-1.0
... >>> Q[(0, 1)]
2.0
```

Social

A signed social network graph is a graph whose signed edges represent friendly/hostile interactions between vertices.

Returns an approximate set of frustrated edges and a bicoloring.

A signed social network graph is a graph whose signed edges represent friendly/hostile interactions between vertices.
Fig. 22: A signed social graph for three nodes, where Eve and Bob are friendly with each other and hostile to Alice. This network is balanced because it can be cleanly divided into two subsets, \{Bob, Eve\} and \{Alice\}, with friendly relations within each subset and only hostile relations between the subsets.
nodes. A signed social network is considered balanced if it can be cleanly divided into two factions, where all relations within a faction are friendly, and all relations between factions are hostile. The measure of imbalance or frustration is the minimum number of edges that violate this rule.

Parameters

- **S** (*NetworkX graph*) – A social graph on which each edge has a ‘sign’ attribute with a numeric value.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrainted Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns

- **frustrated_edges** (*dict*) – A dictionary of the edges that violate the edge sign. The imbalance of the network is the length of frustrated_edges.
- **colors** (*dict*) – A bicoloring of the nodes into two factions.

Raises **ValueError** – If any edge does not have a ‘sign’ attribute.

Examples

```python
>>> import dimod
>>> sampler = dimod.ExactSolver()
>>> S = nx.Graph()
>>> S.add_edge('Alice', 'Bob', sign=1)  # Alice and Bob are friendly
>>> S.add_edge('Alice', 'Eve', sign=-1)  # Alice and Eve are hostile
>>> S.add_edge('Bob', 'Eve', sign=-1)  # Bob and Eve are hostile
>>> frustrated_edges, colors = dnx.structural_imbalance(S, sampler)
>>> print(frustrated_edges)
{}  # doctest: +SKIP
>>> print(colors)
{'Alice': 0, 'Bob': 0, 'Eve': 1}  # doctest: +SKIP

>>> S.add_edge('Ted', 'Bob', sign=1)  # Ted is friendly with all
>>> S.add_edge('Ted', 'Alice', sign=1)
>>> S.add_edge('Ted', 'Eve', sign=1)
>>> frustrated_edges, colors = dnx.structural_imbalance(S, sampler)
>>> print(frustrated_edges)  # doctest: +SKIP
{(('Ted', 'Eve'): {'sign': 1})}
>>> print(colors)  # doctest: +SKIP
{'Bob': 1, 'Ted': 1, 'Alice': 1, 'Eve': 0}
```

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

References

Ising model on Wikipedia
**docs Documentation, Release 2.5.0**

**dwave_networkx.algorithms.social.structural_imbalance_ising**

**structural_imbalance_ising**($S$)

Construct the Ising problem to calculate the structural imbalance of a signed social network.

A signed social network graph is a graph whose signed edges represent friendly/hostile interactions between nodes. A signed social network is considered balanced if it can be cleanly divided into two factions, where all relations within a faction are friendly, and all relations between factions are hostile. The measure of imbalance or frustration is the minimum number of edges that violate this rule.

**Parameters** $S$ *(NetworkX graph)* – A social graph on which each edge has a 'sign' attribute with a numeric value.

**Returns**

- $h$ *(dict)* – The linear biases of the Ising problem. Each variable in the Ising problem represent a node in the signed social network. The solution that minimized the Ising problem will assign each variable a value, either -1 or 1. This bi-coloring defines the factions.


**Raises** `ValueError` – If any edge does not have a 'sign' attribute.

**Examples**

```python
>>> import dimod
>>> from dwave_networkx.algorithms.social import structural_imbalance_ising
... >>> S = nx.Graph()
>>> S.add_edge('Alice', 'Bob', sign=1)  # Alice and Bob are friendly
>>> S.add_edge('Alice', 'Eve', sign=-1)  # Alice and Eve are hostile
>>> S.add_edge('Bob', 'Eve', sign=-1)  # Bob and Eve are hostile
... >>> h, J = structural_imbalance_ising(S)
>>> h  # doctest: +SKIP
{'Alice': 0.0, 'Bob': 0.0, 'Eve': 0.0}
>>> J  # doctest: +SKIP
{('Alice', 'Bob'): -1.0, ('Alice', 'Eve'): 1.0, ('Bob', 'Eve'): 1.0}
```

**Traveling Salesperson**

A traveling salesperson route is an ordering of the vertices in a complete weighted graph.

```
**traveling_salesperson**($G$, sampler, ...) Returns an approximate minimum traveling salesperson route.

**traveling_salesperson_qubo**($G$, lagrange, weight) Return the QUBO with ground states corresponding to a minimum TSP route.
```

**dwave_networkx.algorithms.tsp.traveling_salesperson**

**traveling_salesperson**($G$, sampler=None, lagrange=None, weight='weight', start=None, **sampler_args)

Returns an approximate minimum traveling salesperson route.
Defines a QUBO with ground states corresponding to the minimum routes and uses the sampler to sample from it.

A route is a cycle in the graph that reaches each node exactly once. A minimum route is a route with the smallest total edge weight.

**Parameters**

- **G** (*NetworkX graph*) – The graph on which to find a minimum traveling salesperson route. This should be a complete graph with non-zero weights on every edge.

- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a `sample_qubo` and `sample_ising` method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.

- **lagrange** (*number, optional (default None]*) – Lagrange parameter to weight constraints (visit every city once) versus objective (shortest distance route).

- **weight** (*optional (default 'weight')*) – The name of the edge attribute containing the weight.

- **start** (*node, optional*) – If provided, the route will begin at `start`.

- **sampler_args** – Additional keyword parameters are passed to the sampler.

**Returns**

- **route** – List of nodes in order to be visited on a route

**Return type**

- **list**
Examples

```python
>>> import dimod
...
>>> G = nx.Graph()
>>> G.add_weighted_edges_from({(0, 1, .1), (0, 2, .5), (0, 3, .1), (1, 2, .1),
... (1, 3, .5), (2, 3, .1)})
>>> dnx.traveling_salesperson(G, dimod.ExactSolver(), start=0)  # doctest: +SKIP
[0, 1, 2, 3]
```

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

dwave_networkx.algorithms.tsp.traveling_salesperson_qubo

```
dwave_networkx.algorithms.tsp.traveling_salesperson_qubo

traveling_salesperson_qubo (G, lagrange=None, weight='weight')

Return the QUBO with ground states corresponding to a minimum TSP route.

If |G| is the number of nodes in the graph, the resulting qubo will have:

- \(|G|^2\) variables/nodes
- \(2|G|^2(|G| - 1)\) interactions/edges

Parameters

- G (NetworkX graph) – A complete graph in which each edge has a attribute giving its weight.
- lagrange (number, optional (default None)) – Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).
- weight (optional (default 'weight')) – The name of the edge attribute containing the weight.

Returns QUBO – The QUBO with ground states corresponding to a minimum travelling salesperson route. The QUBO variables are labelled \((c, t)\) where \(c\) is a node in \(G\) and \(t\) is the time index. For instance, if \((\text{'a'}, 0)\) is 1 in the ground state, that means the node ‘a’ is visited first.

Return type dict
```

Drawing

Tools to visualize Chimera lattices and weighted graph problems on them.

Note: Some functionality requires NumPy and/or Matplotlib.

Chimera Graph Functions
chimera_layout(G[, scale, center, dim]) Positions the nodes of graph G in a Chimera cross topology.

draw_chimera(G, **kwargs) Draws graph G in a Chimera cross topology.

dwave_networkx.drawing.chimera_layout.chimera_layout

chimera_layout (G, scale=1.0, center=None, dim=2)
Positions the nodes of graph G in a Chimera cross topology.

NumPy (http://scipy.org) is required for this function.

Parameters

• G (NetworkX graph) – Should be a Chimera graph or a subgraph of a Chimera graph. If every node in G has a chimera_index attribute, those are used to place the nodes. Otherwise makes a best-effort attempt to find positions.

• scale (float (default 1.)) – Scale factor. When scale = 1, all positions fit within [0, 1] on the x-axis and [-1, 0] on the y-axis.

• center (None or array (default None)) – Coordinates of the top left corner.

• dim (int (default 2)) – Number of dimensions. When dim > 2, all extra dimensions are set to 0.

Returns pos – A dictionary of positions keyed by node.

Return type dict

Examples

```python
>>> G = dnx.chimera_graph(1)
>>> pos = dnx.chimera_layout(G)
```

dwave_networkx.drawing.chimera_layout.draw_chimera

draw_chimera (G, **kwargs)
Draws graph G in a Chimera cross topology.

If linear_biases and/or quadratic_biases are provided, these are visualized on the plot.

Parameters

• G (NetworkX graph) – Should be a Chimera graph or a subgraph of a Chimera graph.

• linear_biases (dict (optional, default {})) – A dict of biases associated with each node in G. Should be of form {node: bias, ...}. Each bias should be numeric.

• quadratic_biases (dict (optional, default {})) – A dict of biases associated with each edge in G. Should be of form {edge: bias, ...}. Each bias should be numeric. Self-loop edges (i.e., i = j) are treated as linear biases.

• kwargs (optional keywords) – See networkx.draw_networkx() for a description of optional keywords, with the exception of the pos parameter which is not used by this function. If linear_biases or quadratic_biases are provided, any provided node_color or edge_color arguments are ignored.
Examples

```python
>>> # Plot 2x2 Chimera unit cells
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> G = dnx.chimera_graph(2, 2, 4)
>>> dnx.draw_chimera(G)  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

Example

This example uses the `chimera_layout()` function to show the positions of nodes of a simple 5-node NetworkX graph in a Chimera lattice. It then uses the `chimera_graph()` and `draw_chimera()` functions to display those positions on a Chimera unit cell.

```python
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import matplotlib.pyplot as plt

>>> H = nx.Graph()
>>> H.add_nodes_from([0, 4, 5, 6, 7])
>>> H.add_edges_from([(0, 4), (0, 5), (0, 6), (0, 7)])
>>> pos = dnx.chimera_layout(H)
>>> pos
{0: array([-0.5, 0. ]),
4: array([-0.5, 0. ]),
5: array([-0.25, 0. ]),
6: array([-0.75, 0. ]),
7: array([-1. , 0. ])}
>>> # Show graph H on a Chimera unit cell
>>> plt.ion()
>>> G = dnx.chimera_graph(1, 1, 4)  # Draw a Chimera unit cell
>>> dnx.draw_chimera(G)
>>> dnx.draw_chimera(H, node_color='b', node_shape='*', style='dashed', edge_color='b', width=3)
```

Graph Generators

Generators for graphs, such as the graphs (topologies) of D-Wave System QPUs.

D-Wave Systems

```python
chimera_graph(m[, n, t, create_using, ...]) Creates a Chimera lattice of size (m, n, t).
pegasus_graph(m[, create_using, node_list, ...]) Creates a Pegasus graph with size parameter m.
dwave_networkx.chimera_graph
```

`chimera_graph(m, n=None, t=None, create_using=None, node_list=None, edge_list=None, data=True, coordinates=False)`

Creates a Chimera lattice of size (m, n, t).
Parameters

- \textbf{m (int)} – Number of rows in the Chimera lattice.
- \textbf{n (int (optional, default m))} – Number of columns in the Chimera lattice.
- \textbf{t (int (optional, default 4))} – Size of the shore within each Chimera tile.
- \textbf{create_using (Graph (optional, default None))} – If provided, this graph is cleared of nodes and edges and filled with the new graph. Usually used to set the type of the graph.
- \textbf{node_list (iterable (optional, default None))} – Iterable of nodes in the graph. If None, calculated from \((m, n, t)\). Note that this list is used to remove nodes, so any nodes specified not in \(range(m \times n \times 2 \times t)\) are not added.
- \textbf{edge_list (iterable (optional, default None))} – Iterable of edges in the graph. If None, edges are generated as described below. The nodes in each edge must be integer-labeled in \(range(m \times n \times t \times 2)\).
- \textbf{data (bool (optional, default True))} – If True, each node has a \textit{chimera_index attribute}. The attribute is a 4-tuple Chimera index as defined below.
- \textbf{coordinates (bool (optional, default False))} – If True, node labels are 4-tuples, equivalent to the \textit{chimera_index attribute} as below. In this case, the \textit{data} parameter controls the existence of a \textit{linear_index attribute}, which is an int.

\textbf{Returns G} – An \((m, n, t)\) Chimera lattice. Nodes are labeled by integers.

\textbf{Return type} NetworkX Graph

A Chimera lattice is an \(m\)-by-\(n\) grid of Chimera tiles. Each Chimera tile is itself a bipartite graph with shores of size \(t\). The connection in a Chimera lattice can be expressed using a node-indexing notation \((i,j,u,k)\) for each node.

- \((i,j)\) indexes the (row, column) of the Chimera tile. \(i\) must be between 0 and \(m-1\), inclusive, and \(j\) must be between 0 and \(n-1\), inclusive.
- \(u=0\) indicates the left-hand nodes in the tile, and \(u=1\) indicates the right-hand nodes.

Fig. 24: Graph H (blue) overlaid on a Chimera unit cell (red nodes and black edges).
• k=0,1,…,t-1 indexes nodes within either the left- or right-hand shores of a tile.

In this notation, two nodes (i, j, u, k) and (i’, j’, u’, k’) are neighbors if and only if:

(i = i’ AND j = j’ AND u != u’) OR (i = i’ +/- 1 AND j = j’ AND u = 0 AND u’ = 0 AND k = k’) OR
(i = i’ AND j = j’ +/- 1 AND u = 1 AND u’ = 1 AND k = k’)

The first of the three terms of the disjunction gives the bipartite connections within the tile. The second and third terms give the vertical and horizontal connections between blocks respectively.

Node (i, j, u, k) is labeled by:

label = i * n * 2 * t + j * 2 * t + u * t + k

Examples

```python
>>> G = dnx.chimera_graph(1, 1, 2)  # a single Chimera tile
>>> len(G)
4
>>> list(G.nodes())  # doctest: +SKIP
[0, 1, 2, 3]
```  
```python
>>> list(G.nodes(data=True))  # doctest: +SKIP
[(0, {'chimera_index': (0, 0, 0, 0)}),
 (1, {'chimera_index': (0, 0, 0, 1)}),
 (2, {'chimera_index': (0, 0, 1, 0)}),
 (3, {'chimera_index': (0, 0, 1, 1)})]
```  
```python
>>> list(G.edges())  # doctest: +SKIP
[(0, 2), (0, 3), (1, 2), (1, 3)]
```

dwave_networkx.pegasus_graph

`pegasus_graph(m, create_using=None, node_list=None, edge_list=None, data=True, offset_lists=None, offsets_index=None, coordinates=False, fabric_only=True, nice_coordinates=False)`  
Creates a Pegasus graph with size parameter m.

Parameters

• `m (int)` – Size parameter for the Pegasus lattice.

• `create_using (Graph, optional (default None))` – If provided, this graph is cleared of nodes and edges and filled with the new graph. Usually used to set the type of the graph.

• `node_list (iterable, optional (default None))` – Iterable of nodes in the graph. If None, calculated from m. Note that this list is used to remove nodes, so any nodes specified not in range(24 * m * (m−1)) are not added.

• `edge_list (iterable, optional (default None))` – Iterable of edges in the graph. If None, edges are generated as described below. The nodes in each edge must be integer-labeled in range(24 * m * (m−1)).

• `data (bool, optional (default True))` – If True, each node has a Pegasus_index attribute. The attribute is a 4-tuple Pegasus index as defined below. If the coordinates parameter is True, a linear_index, which is an integer, is used.

• `coordinates (bool, optional (default False))` – If True, node labels are 4-tuple Pegasus indices. Ignored if the nice_coordinates parameter is True.
• **offset_lists** *(pair of lists, optional (default None)) –* Directly controls the offsets. Each list in the pair must have length 12 and contain even ints. If `offset_lists` is not None, the `offsets_index` parameter must be None.

• **offsets_index** *(int, optional (default None)) –* A number between 0 and 7, inclusive, that selects a preconfigured set of topological parameters. If both the `offsets_index` and `offset_lists` parameters are None, the `offsets_index` parameters is set to zero. At least one of these two parameters must be None.

• **fabric_only** *(bool, optional (default True)) –* The Pegasus graph, by definition, has some disconnected components. If True, the generator only constructs nodes from the largest component. If False, the full disconnected graph is constructed. Ignored if the `edge_lists` parameter is not None or `nice_coordinates` is True.

• **nice_coordinates** *(bool, optional (default False)) –* If the `offsets_index` parameter is 0, the graph uses a “nicer” coordinate system, more compatible with Chimera addressing. These coordinates are 5-tuples taking the form $(t, y, x, u, k)$ where $0 < x < M - 1$, $0 < y < M - 1$, $0 < u < 2$, $0 < k < 4$, and $0 < t < 3$. For any given $0 <= t0 < 3$, the subgraph of nodes with $t = t0$ has the structure of chimera(M-1, M-1, 4) with the addition of odd couplers. Supercedes both the `fabric_only` and `coordinates` parameters.

Returns **G** – A Pegasus lattice for size parameter $m$.

Return type NetworkX Graph

The maximum degree of this graph is 15. The number of nodes depends on multiple parameters; for example,

• **pegasus_graph(1)**: zero nodes

• **pegasus_graph(m, fabric_only=False)**: $24m(m - 1)$ nodes

• **pegasus_graph(m, fabric_only=True)**: $24m(m - 1) - 8(m - 1)$ nodes

• **pegasus_graph(m, nice_coordinates=True)**: $24(m - 1)^2$ nodes

Counting formulas for edges have a complicated dependency on parameter settings. Some example upper bounds are:

• **pegasus_graph(1, fabric_only=False)**: zero edges

• **pegasus_graph(m, fabric_only=False)**: $12 * (15 * (m - 1)^2 + m - 3)$ edges if $m > 1$

Note that the formulas above are valid for default offset parameters.

A Pegasus lattice is a graph minor of a lattice similar to Chimera, where unit tiles are completely connected. In its most general definition, prelattice $Q(N0, N1)$ contains nodes of the form

• vertical nodes: $(i, j, 0, k)$ with $0 <= k < 2$

• horizontal nodes: $(i, j, 1, k)$ with $0 <= k < 2$

for $0 <= i <= N0$ and $0 <= j < N1$, and edges of the form

• external: $(i, j, u, k) ~ (i + u, j + 1 - u, u, k)$

• internal: $(i, j, 0, k) ~ (i, j, 1, k)$

• odd: $(i, j, u, 0) ~ (i, j, u, 1)$

Given two lists of offsets, $S0$ and $S1$, of length $L0$ and $L1$, where both lengths and values must be divisible by 2, the minor—a Pegasus lattice—is constructed by contracting the complete intervals of external edges:

\[
\begin{align*}
I(0, w, k, z) & = [(L1\ast w + k, L0\ast z + S0[k] + r, 0, k \mod 2) \for 0 <= r < L0] \\
I(1, w, k, z) & = [(L1\ast z + S1[k] + r, L0\ast w + k, 1, k \mod 2) \for 0 <= r < L1]
\end{align*}
\]
and deleting the prelattice nodes of any interval not fully contained in \(Q(N0, N1)\).

This generator, `pegasus_graph()`, is specialized for the minor constructed by prelattice and offset parameters \(L0 = 12 \) and \(N0 = N1 = 12m\).

The *Pegasus index* of a node in a Pegasus lattice, \((u, w, k, z)\), can be interpreted as:

- \(u\): qubit orientation \((0 = \text{vertical}, \ 1 = \text{horizontal})\)
- \(w\): orthogonal major offset
- \(k\): orthogonal minor offset
- \(z\): parallel offset

Edges in the minor have the form

- external: \((u, w, k, z) \sim (u, w, k, z + 1)\)
- internal: \((0, w0, k0, z0) \sim (1, w1, k1, z1)\)
- odd: \((u, w, 2k, z) \sim (u, w, 2k + 1, z)\)

where internal edges only exist when

1. \(w1 = z0 + (1 \text{ if } k1 < S0[k0] \text{ else } 0)\)
2. \(z1 = w0 - (1 \text{ if } k0 < S1[k1] \text{ else } 0)\)

Linear indices are computed from Pegasus indices by the formula:

\[
q = ((u \times m + w) \times 12 + k) \times (m - 1) + z
\]

Examples

```python
>>> G = dnx.pegasus_graph(2, nice_coordinates=True)
>>> G.nodes(data=True)[(0, 0, 0, 0, 0)]
# doctest: +SKIP
{'linear_index': 4, 'pegasus_index': (0, 0, 4, 0)}
```

Example

This example uses the the `chimera_graph()` function to create a Chimera lattice of size \((1, 1, 4)\), which is a single unit cell in Chimera topology, and the `find_chimera()` function to determine the Chimera indices.

```python
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> chimera_indices = dnx.find_chimera_indices(G)
>>> print chimera_indices
{(0, 0, 0, 0),
 1: (0, 0, 0, 1),
 2: (0, 0, 0, 2),
 3: (0, 0, 0, 3),
 4: (0, 0, 1, 0),
 5: (0, 0, 1, 1),
 6: (0, 0, 1, 2),
 7: (0, 0, 1, 3)}
```
Fig. 25: Indices of a Chimera unit cell found by creating a lattice of size (1, 1, 4).

Other Graphs

**markov_network**(potentials)  Creates a Markov Network from potentials.

**dwave_networkx.markov_network**

**markov_network**(potentials)

Creates a Markov Network from potentials.

A Markov Network is also known as a Markov Random Field

**Parameters** potentials (dict[tuple, dict]) – A dict where the keys are either nodes or edges and the values are a dictionary of potentials. The potential dict should map each possible assignment of the nodes/edges to their energy.

**Returns** MN – A markov network as a graph where each node/edge stores its potential dict as above.

**Return type** networkx.Graph

**Examples**

```python
>>> potentials = {('a', 'b'): {(0, 0): -1,
...                (0, 1): .5,
...                (1, 0): .5,
...                (1, 1): 2}}
>>> MN = dnx.markov_network(potentials)
>>> MN['a']['b']['potential'][0, 0]
-1
```

Utilities

Decorators

Decorators allow for input checking and default parameter setting for algorithms.
Examples

Decorate functions like this:

```python
@binary_quadratic_model_sampler(1)
def maximal_matching(G, sampler, **sampler_args):
    pass
```

This example validates two placeholder samplers, which return a correct response only in the case of finding an independent set on a complete graph (where one node is always an independent set), the first valid, the second missing a method.

```python
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> from dwave_networkx.utils import decorators

# Create two placeholder samplers
>>> class WellDefinedSampler:
...     # an example sampler, only works for independent set on complete graphs
...     def __init__(self, name):
...         self.name = name
...     def sample_ising(self, h, J):
...         sample = {v: -1 for v in h}
...         sample[0] = 1  # set one node to true
...         return [sample]
...     def sample_qubo(self, Q):
...         sample = {v: 0 for v in set().union(*Q)}
...         sample[0] = 1  # set one node to true
...         return [sample]
...     def __str__(self):
...         return self.name
...

>>> class IllDefinedSampler:
...     # an example sampler missing a `sample_qubo` method
...     def __init__(self, name):
...         self.name = name
...     def sample_ising(self, h, J):
...         sample = {v: -1 for v in h}
...         sample[0] = 1  # set one node to true
...         return [sample]
...     def __str__(self):
...         return self.name
...

>>> sampler1 = WellDefinedSampler('sampler1')
>>> sampler2 = IllDefinedSampler('sampler2')

# Define a placeholder independent-set function with the decorator
>>> @dnx.utils.binary_quadratic_model_sampler(1)
... def independent_set(G, sampler, **sampler_args):
...     Q = {(node, node): -1 for node in G}
...     Q.update({edge: 2 for edge in G.edges})
...     response = sampler.sample_qubo(Q, **sampler_args)
...     sample = next(iter(response))
...     return [node for node in sample if sample[node] > 0]
...
>>> # Validate the samplers
>>> G = nx.complete_graph(5)
>>> independent_set(G, sampler1)[0]
```
>>> independent_set(G, sampler2)  # doctest: +SKIP
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-35-670b71b268c7> in <module>()
----> 1 independent_set(G, IllDefinedSampler)
<decorator-gen-628> in independent_set(G, sampler, **sampler_args)/usr/local/lib/python2.7/dist-packages/dwave_networkx/utils/decorators.pyc in _
    binary_quadratic_model_sampler(f, *args, **kw)
       61       62 if not hasattr(sampler, "sample_qubo") or not_
    callable(sampler.sample_qubo):
       63 raise TypeError("expected sampler to have a 'sample_qubo'_
    method")
       64 if not hasattr(sampler, "sample_ising") or not_
    callable(sampler.sample_ising):
       65 raise TypeError("expected sampler to have a 'sample_ising_
    ' method")
TypeException: expected sampler to have a 'sample_qubo' method

Graph Indexing

Chimera

Chimera Coordinates Conversion

class chimera_coordinates (m, n=None, t=None)
Provides coordinate converters for the chimera indexing scheme.

Parameters

- m (int) – The number of rows in the Chimera lattice.
- n (int, optional (default m)) – The number of columns in the Chimera lattice.
- t (int, optional (default 4)) – The size of the shore within each Chimera tile.

Examples

Convert between Chimera coordinates and linear indices directly

```python
>>> coords = dnx.chimera_coordinates(16, 16, 4)
>>> coords.chimera_to_linear((0, 2, 0, 1))
17
>>> coords.linear_to_chimera(17)
(0, 2, 0, 1)
```

Construct a new graph with the coordinate labels

```python
>>> C16 = dnx.chimera_graph(16)
>>> coords = dnx.chimera_coordinates(16)
>>> G = nx.Graph()
>>> G.add_nodes_from(coords.iter_linear_to_chimera(C16.nodes))
>>> G.add_edges_from(coords.iter_linear_to_chimera_pairs(C16.edges))
```
Chimera Coordinates Conversion

find_chimera_indices(G) Attempts to determine the Chimera indices of the nodes in graph G.

See also:
chimera_graph()

Pegasus

For the iterator versions of these functions, see the code.
**dwave_networkx.pegasus_coordinates.linear_to_nice**

`pegasus_coordinates.linear_to_nice(r)`  
Convert a linear index into a 5-term nice coordinate.  

**Parameters**  
`r (int)` – Linear index.

**Examples**

```python  
>>> dnx.pegasus_coordinates(2).linear_to_nice(4)  
(0, 0, 0, 0, 0)  
```

**dwave_networkx.pegasus_coordinates.linear_to_pegasus**

`pegasus_coordinates.linear_to_pegasus(r)`  
Convert a linear index into a 4-term Pegasus coordinate.  

**Parameters**  
`r (int)` – Linear index.

**Examples**

```python  
>>> dnx.pegasus_coordinates(2).linear_to_pegasus(4)  
(0, 0, 4, 0)  
```

**dwave_networkx.pegasus_coordinates.nice_to_linear**

`pegasus_coordinates.nice_to_linear(n)`  
Convert a 5-term nice coordinate into a linear index.  

**Parameters**  
`n (5-tuple)` – Nice coordinate.

**Examples**

```python  
>>> dnx.pegasus_coordinates(2).nice_to_linear((0, 0, 0, 0, 0))  
4  
```

**dwave_networkx.pegasus_coordinates.nice_to_pegasus**

`static pegasus_coordinates.nice_to_pegasus(n)`  
Convert a 5-term nice coordinate into a 4-term Pegasus coordinate.  

**Parameters**  
`n (5-tuple)` – Nice coordinate.

**Examples**

```python  
>>> dnx.pegasus_coordinates.nice_to_pegasus((0, 0, 0, 0, 0))  
(0, 0, 4, 0)  
```
Note that this method does not depend on the size of the Pegasus lattice.

**`dwave_networkx.pegasus_coordinates.pegasus_to_linear`**

`pegasus_to_linear(q)`  
Convert a 4-term Pegasus coordinate into a linear index.  

**Parameters**  
`q (4-tuple)` – Pegasus indices.

**Examples**

```python
dnx.pegasus_coordinates(2).pegasus_to_linear((0, 0, 4, 0))
```

4

**`dwave_networkx.pegasus_coordinates.pegasus_to_nice`**

`static pegasus_to_nice(p)`  
Convert a 4-term Pegasus coordinate to a 5-term nice coordinate.  

**Parameters**  
`p (4-tuple)` – Pegasus coordinate.

**Examples**

```python
dnx.pegasus_coordinates.pegasus_to_nice((0, 0, 4, 0))
```

(0, 0, 0, 0, 0)

Note that this method does not depend on the size of the Pegasus lattice.

**Exceptions**

Base exceptions and errors for D-Wave NetworkX.  
All exceptions are derived from NetworkXException.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DWaveNetworkXException</code></td>
<td>Base class for exceptions in <code>DWaveNetworkX</code>.</td>
</tr>
<tr>
<td><code>DWaveNetworkXMissingSampler</code></td>
<td>Exception raised by an algorithm requiring a discrete model sampler when none is provided.</td>
</tr>
</tbody>
</table>

---

476 Chapter 6. Packages
Default sampler

Sets a binary quadratic model sampler used by default for functions that require a sample when none is specified.

A sampler is a process that samples from low-energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO).

Sampler API

- Required Methods: ‘sample_qubo’ and ‘sample_ising’
- Return value: iterable of samples, in order of increasing energy

See dimod for details.

Example

This example creates and uses a placeholder for binary quadratic model samplers that returns a correct response only in the case of finding an independent set on a complete graph (where one node is always an independent set). The placeholder sampler can be used to test the simple examples of the functions for configuring a default sampler.

```python
>>> # Create a placeholder sampler
>>> class ExampleSampler:
...     # an example sampler, only works for independent set on complete graphs
...     def __init__(self, name):
...         self.name = name
...     def sample_ising(self, h, J):
...         sample = {v: -1 for v in h}
...         sample[0] = 1  # set one node to true
...         return [sample]
...     def sample_qubo(self, Q):
...         sample = {v: 0 for v in set().union(*Q)}
...         sample[0] = 1  # set one node to true
...         return [sample]
...     def __str__(self):
...         return self.name
...
>>> # Identify the new sampler as the default sampler
>>> sampler0 = ExampleSampler('sampler0')
>>> dnx.set_default_sampler(sampler0)
>>> # Find an independent set using the default sampler
>>> G = nx.complete_graph(5)
>>> dnx.maximum_independent_set(G)
[0]
```

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_default_sampler</code></td>
<td>Sets a default binary quadratic model sampler.</td>
</tr>
<tr>
<td><code>unset_default_sampler</code></td>
<td>Resets the default sampler back to None.</td>
</tr>
<tr>
<td><code>get_default_sampler</code></td>
<td>Queries the current default sampler.</td>
</tr>
</tbody>
</table>

6.7. dwave-networkx 477
**set_default_sampler** *(sampler)*

Sets a default binary quadratic model sampler.

**Parameters**

- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low-energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ’sample_qubo’ and ’sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy.

**Examples**

This example sets sampler0 as the default sampler and finds an independent set for graph G, first using the default sampler and then overriding it by specifying a different sampler.

```python
>>> dnx.set_default_sampler(sampler0) # doctest: +SKIP
>>> indep_set = dnx.maximum_independent_set_dm(G) # doctest: +SKIP
>>> indep_set = dnx.maximum_independent_set_dm(G, sampler1) # doctest: +SKIP
```

**unset_default_sampler** *(*)

Resets the default sampler back to None.

**Examples**

This example sets sampler0 as the default sampler, verifies the setting, then resets the default, and verifies the resetting.

```python
>>> dnx.set_default_sampler(sampler0) # doctest: +SKIP
>>> print(dnx.get_default_sampler()) # doctest: +SKIP
'sampler0'
>>> dnx.unset_default_sampler() # doctest: +SKIP
>>> print(dnx.get_default_sampler()) # doctest: +SKIP
None
```

**get_default_sampler** *(*)

Queries the current default sampler.

**Examples**

This example queries the default sampler before and after specifying a default sampler.

```python
>>> print(dnx.get_default_sampler()) # doctest: +SKIP
None
>>> dnx.set_default_sampler(sampler) # doctest: +SKIP
>>> print(dnx.get_default_sampler()) # doctest: +SKIP
'sampler'
```
6.7.3 Bibliography

6.8 dwave-system

dwave-system is a basic API for easily incorporating the D-Wave system as a sampler in the D-Wave Ocean software stack, directly or through Leap’s cloud-based hybrid solvers. It includes DWaveSampler, a dimod sampler that accepts and passes system parameters such as system identification and authentication down the stack, and LeapHybridSampler, for Leap’s hybrid solvers. It also includes several useful composites—layers of pre- and post-processing—that can be used with DWaveSampler to handle minor-embedding, optimize chain strength, etc.

6.8.1 Introduction

dwave-system enables easy incorporation of the D-Wave system as a sampler in either a hybrid quantum-classical solution, using LeapHybridSampler() or dwave-hybrid samplers such as KerberosSampler(), or directly using DWaveSampler().

Note: For applications that require detailed control on communication with the remote compute resource (a D-Wave QPU or Leap’s hybrid solvers), see dwave-cloud-client.

D-Wave System Documentation describes D-Wave quantum computers and Leap hybrid solvers, including features, parameters, and properties. It also provides guidance on programming the D-Wave system, including how to formulate problems and configure parameters.

Example

This example solves a small example of a known graph problem, minimum vertex cover. It uses the NetworkX graphic package to create the problem, Ocean’s dwave_networkx to formulate the graph problem as a BQM, and dwave-system’s DWaveSampler() to use a D-Wave system as the sampler. dwave-system’s EmbeddingComposite() handles mapping between the problem graph to the D-Wave system’s numerically indexed qubits, a mapping known as minor-embedding.

```python
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> from dwave.system import DWaveSampler, EmbeddingComposite
...
>>> s5 = nx.star_graph(4)  # a star graph where node 0 is hub to four other nodes
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> print(dnx.min_vertex_cover(s5, sampler))
[0]
```

6.8.2 Reference Documentation

Samplers

A sampler accepts a binary quadratic model (BQM) and returns variable assignments. Samplers generally try to find minimizing values but can also sample from distributions defined by the BQM.
class `DWaveSampler`(failover=False, retry_interval=-1, order_by=None, **config)

A class for using the D-Wave system as a sampler.

Uses parameters set in a configuration file, as environment variables, or explicitly as input arguments for selecting and communicating with a D-Wave system. For more information, see D-Wave Cloud Client.

Inherits from `dimod.Sampler` and `dimod.Structured`.

Parameters

- **failover** (bool, optional, default=False) – Switch to a new QPU in the rare event that the currently connected system goes offline. Note that different QPUs may have different hardware graphs and a failover will result in a regenerated `nodelist`, `edgelist`, `properties` and `parameters`.

- **retry_interval** (number, optional, default=-1) – The amount of time (in seconds) to wait to poll for a solver in the case that no solver is found. If `retry_interval` is negative then it will instead propagate the `SolverNotFoundError` to the user.

- **order_by** (callable/str/None) – Solver sorting key function or (or `Solver` attribute/item dot-separated path). See `get_solvers` for a more detailed description of the parameter.

- **config_file** (str, optional) – Path to a configuration file that identifies a D-Wave system and provides connection information.

- **profile** (str, optional) – Profile to select from the configuration file.

- **endpoint** (str, optional) – D-Wave API endpoint URL.

- **token** (str, optional) – Authentication token for the D-Wave API to authenticate the client session.

- **solver** (dict/str, optional) – Solver (a D-Wave system on which to run submitted problems) to select given as a set of required features. Supported features and values are described in `get_solvers()`. For backward compatibility, a solver name, formatted as a string, is accepted.

- **proxy** (str, optional) – Proxy URL to be used for accessing the D-Wave API.

- ****config – Keyword arguments passed directly to `from_config()`.

Examples

This example submits a two-variable Ising problem mapped directly to qubits 0 and 1 on a D-Wave system selected by explicitly requiring that it have these two active qubits. Other required parameters for communication with the system, such as its URL and an authentication token, are implicitly set in a configuration file or as environment variables, as described in Configuring a D-Wave System. Given sufficient reads (here 100), the quantum computer should return the best solution, 1, −1 on qubits 0 and 1, respectively, as its first sample (samples are ordered from lowest energy).

```python
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler(solver={'qubits__issuperset': {0, 1}})
>>> sampleset = sampler.sample_ising({0: -1, 1: 1}, {}, num_reads=100)
>>> sampleset.first.sample[0] == 1 and sampleset.first.sample[1] == -1
True
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.
Properties

For parameters and properties of D-Wave systems, see D-Wave System Documentation.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWaveSampler.properties</td>
<td>D-Wave solver properties as returned by a SAPI query.</td>
</tr>
<tr>
<td>DWaveSampler.parameters</td>
<td>D-Wave solver parameters in the form of a dict, where keys are keyword parameters accepted by a SAPI query and values are lists of properties in DWaveSampler.properties for each key.</td>
</tr>
<tr>
<td>DWaveSampler.nodelist</td>
<td>List of active qubits for the D-Wave solver.</td>
</tr>
<tr>
<td>DWaveSampler.edgelist</td>
<td>List of active couplers for the D-Wave solver.</td>
</tr>
<tr>
<td>DWaveSampler.adjacency</td>
<td>Adjacency structure formatted as a dict, where keys are the nodes of the structured sampler and values are sets of all adjacent nodes for each key node.</td>
</tr>
<tr>
<td>DWaveSampler.structure</td>
<td>Structure of the structured sampler formatted as a namedtuple, Structure(nodelist, edgelist, adjacency), where the 3-tuple values are the nodelist, edgelist and adjacency attributes.</td>
</tr>
</tbody>
</table>

dwave.system.samplers.DWaveSampler.properties

**DWaveSampler.properties**
D-Wave solver properties as returned by a SAPI query.

Solver properties are dependent on the selected D-Wave solver and subject to change; for example, new released features may add properties. D-Wave System Documentation describes the parameters and properties supported on the D-Wave system.

**Examples**

```python
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.properties  # doctest: +SKIP
{u'anneal_offset_ranges': [[-0.219746375553704, 0.03821687759418928],
 [-0.2242514597680286, 0.01718456460967399],
 [-0.20860153999435985, 0.05511969218508182],
 # Snipped above response for brevity
See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.
```

**Type** dict

dwave.system.samplers.DWaveSampler.parameters

**DWaveSampler.parameters**
D-Wave solver parameters in the form of a dict, where keys are keyword parameters accepted by a SAPI query and values are lists of properties in DWaveSampler.properties for each key.

Solver parameters are dependent on the selected D-Wave solver and subject to change; for example, new released features may add parameters. D-Wave System Documentation describes the parameters and properties supported on the D-Wave system.
Examples

```python
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.parameters  # doctest: +SKIP
{u'anneal_offsets': ['parameters'],
 u'anneal_schedule': ['parameters'],
 u'annealing_time': ['parameters'],
 u'answer_mode': ['parameters'],
 u'auto_scale': ['parameters'],
 # Snipped above response for brevity
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

Type  `dict[str, list]`

dwave.system.samplers.DWaveSampler.nodelist

`DWaveSampler.nodelist`

List of active qubits for the D-Wave solver.

Examples

Node list for one D-Wave 2000Q system (output snipped for brevity).

```python
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.nodelist
[0, 1, 2, ...
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

Type  `list`

dwave.system.samplers.DWaveSampler.edgelist

`DWaveSampler.edgelist`

List of active couplers for the D-Wave solver.

Examples

Coupler list for one D-Wave 2000Q system (output snipped for brevity).

```python
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.edgelist
[(0, 4), (0, 5), (0, 6), (0, 7), ...
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

Type  `list`
**DWaveSampler.adjacency**

Adjacency structure formatted as a dict, where keys are the nodes of the structured sampler and values are sets of all adjacent nodes for each key node.

*Type*  
`dict[variable, set]`

---

**DWaveSampler.structure**

Structure of the structured sampler formatted as a namedtuple, `Structure(nodelist, edgelist, adjacency)`, where the 3-tuple values are the `nodelist`, `edgelist` and `adjacency` attributes.

---

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DWaveSampler.sample(bqm[, warnings])</code></td>
<td>Sample from the specified Ising model.</td>
</tr>
<tr>
<td><code>DWaveSampler.sample_ising(h, *args, **kwargs)</code></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><code>DWaveSampler.sample_qubo(Q, **parameters)</code></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
<tr>
<td><code>DWaveSampler.validate_anneal_schedule(...)</code></td>
<td>Raise an exception if the specified schedule is invalid for the sampler.</td>
</tr>
</tbody>
</table>

---

**DWaveSampler.sample(bqm, warnings=None, **kwargs)**

Sample from the specified Ising model.

**Parameters**

- **bqm** *(BinaryQuadraticModel)* – The binary quadratic model. Must match `nodelist` and `edgelist`.
- **warnings** *(WarningAction, optional)* – Defines what warning action to take, if any. See `warnings`. The default behaviour is defined by `warnings_default`, which itself defaults to `IGNORE`
- **kwargs** – Optional keyword arguments for the sampling method, specified per solver in `DWaveSampler.parameters`. D-Wave System Documentation’s `solver guide` describes the parameters and properties supported on the D-Wave system.

**Returns** A `dimod.SampleSet` object. In it this sampler also provides timing information in the `info` field as described in the D-Wave System Documentation’s `timing guide`.

**Return type** `dimod.SampleSet`

---

### Examples

This example submits a two-variable Ising problem mapped directly to qubits 0 and 1 on a D-Wave system. Given sufficient reads (here 100), the quantum computer should return the best solution, 1, \(-1\) on qubits 0 and 1, respectively, as its first sample (samples are ordered from lowest energy).
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler(solver={'qubits__issuperset': {0, 1}})
>>> sampleset = sampler.sample_ising({0: -1, 1: 1}, {}, num_reads=100)
>>> sampleset.first.sample[0] == 1 and sampleset.first.sample[1] == -1
True

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

**dwave.system.samplers.DWaveSampler.sample_ising**

DWaveSampler.sample_ising(h, *args, **kwargs)
Sample from an Ising model using the implemented sample method.
This method is inherited from the Sampler base class.
Converts the Ising model into a BinaryQuadraticModel and then calls sample().

**Parameters**

- **h (dict/list)** – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where `v` is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

- **J (dict[(variable, variable), bias])** – Quadratic biases of the Ising problem.

- ****kwargs – See the implemented sampling for additional keyword definitions.

**Returns** SampleSet

See also: sample(), sample_qubo()
**DWaveSampler**.validate_anneal_schedule

Raise an exception if the specified schedule is invalid for the sampler.

**Parameters**

*anneal_schedule* *(list)* – An anneal schedule variation is defined by a series of pairs of floating-point numbers identifying points in the schedule at which to change slope. The first element in the pair is time t in microseconds; the second, normalized persistent current s in the range [0,1]. The resulting schedule is the piecewise-linear curve that connects the provided points.

**Raises**

- *ValueError* – If the schedule violates any of the conditions listed below.
- *RuntimeError* – If the sampler does not accept the *anneal_schedule* parameter or if it does not have `annealing_time_range` or `max_anneal_schedule_points` properties.

As described in [D-Wave System Documentation](https://docs.dwavesys.com), an anneal schedule must satisfy the following conditions:

- Time t must increase for all points in the schedule.
- For forward annealing, the first point must be (0,0) and the anneal fraction s must increase monotonically.
- For reverse annealing, the anneal fraction s must start and end at s=1.
- In the final point, anneal fraction s must equal 1 and time t must not exceed the maximum value in the `annealing_time_range` property.
- The number of points must be >=2.
- The upper bound is system-dependent; check the `max_anneal_schedule_points` property. For reverse annealing, the maximum number of points allowed is one more than the number given by this property.

**Examples**

This example sets a quench schedule on a D-Wave system.

```python
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler()
>>> quench_schedule=[[0.0, 0.0], [12.0, 0.6], [12.8, 1.0]]
>>> DWaveSampler().validate_anneal_schedule(quench_schedule)        # doctest: +SKIP
```

**DWaveCliqueSampler**

**class** DWaveCliqueSampler (**config**)

A sampler for solving clique problems on the D-Wave system.

The **DWaveCliqueSampler** wraps minorminer.busclique.find_clique_embedding() to generate embeddings with even chain length. These embeddings will work well for dense binary quadratic models. For sparse models, using EmbeddingComposite with **DWaveSampler** is preferred.

**Parameters**

**config** – Keyword arguments, as accepted by **DWaveSampler**

**Properties**

6.8. dwave-system 485
### DWaveCliqueSampler

- **largest_clique_size**
  - The maximum number of variables.

- **properties**
  - A dict containing any additional information about the sampler.

- **parameters**
  - A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

---

**Methods**

- **largest_clique()**
  - Return a largest-size clique embedding.

- **sample(bqm[, chain_strength])**
  - Sample from the specified binary quadratic model.

- **sample_ising(h, J, **kwargs)**
  - Sample from an Ising model using the implemented sample method.

- **sample_qubo(Q, **parameters)**
  - Sample from a QUBO using the implemented sample method.

---

**Examples**

```python
DWaveCliqueSampler().largest_clique()  # Return a largest-size clique embedding.
DWaveCliqueSampler().sample(bqm)  # Sample from the specified binary quadratic model.
```
• **bqm** (*BinaryQuadraticModel*) – Any binary quadratic model with up to `largest_clique_size` variables. This BQM is embedded using a dense clique embedding.

• **chain_strength** (*float, optional*) – The (relative) chain strength to use in the embedding. By default a chain strength of $1.5 \sqrt{N}$ where $N$ is the size of the largest clique, as returned by `largest_clique_size`.

• ****kwargs – Optional keyword arguments for the sampling method, specified per solver in `DWaveCliqueSampler.parameters`. D-Wave System Documentation’s solver guide describes the parameters and properties supported on the D-Wave system. Note that `auto_scale` is not supported by this sampler, because it scales the problem as part of the embedding process.

dwave.system.samplers.DWaveCliqueSampler.sample_ising

`DWaveCliqueSampler.sample_ising(h, J, **parameters)`  
Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

• **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

• **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.

• ****kwargs – See the implemented sampling for additional keyword definitions.

**Returns** `SampleSet`

See also: `sample()`, `sample_qubo()`

dwave.system.samplers.DWaveCliqueSampler.sample_qubo

`DWaveCliqueSampler.sample_qubo(Q, **parameters)`  
Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the QUBO into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

• **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v` are binary-valued variables and `bias` is their associated coefficient.

• ****kwargs – See the implemented sampling for additional keyword definitions.

**Returns** `SampleSet`
See also:

`sample()`, `sample_ising()`

**LeapHybridSampler**

class LeapHybridSampler (solver=None, connection_close=True, **config)

A class for using Leap’s cloud-based hybrid solvers.

Uses parameters set in a configuration file, as environment variables, or explicitly as input arguments for selecting and communicating with a hybrid solver. For more information, see D-Wave Cloud Client.

Inherits from `dimod.Sampler`.

**Parameters**

- **config_file** *(str, optional)* – Path to a configuration file that identifies a hybrid solver and provides connection information.
- **profile** *(str, optional)* – Profile to select from the configuration file.
- **endpoint** *(str, optional)* – D-Wave API endpoint URL.
- **token** *(str, optional)* – Authentication token for the D-Wave API to authenticate the client session.
- **solver** *(dict/str, optional)* – Solver (a hybrid solver on which to run submitted problems) to select, formatted as a string.
- **proxy** *(str, optional)* – Proxy URL to be used for accessing the D-Wave API.
- ****config – Keyword arguments passed directly to `from_config()`.

**Examples**

This example builds a random sparse graph and uses a hybrid solver to find a maximum independent set.

```py
>>> import dimod
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import numpy as np
>>> from dwave.system import LeapHybridSampler
...
>>> # Create a maximum-independent set problem from a random graph
>>> problem_node_count = 300
>>> G = nx.random_geometric_graph(problem_node_count, radius=0.0005*problem_node_count)
>>> qubo = dnx.algorithms.independent_set.maximum_weighted_independent_set_qubo(G)
>>> bqm = dimod.BQM.from_qubo(qubo)
...
>>> # Find a good solution
>>> sampler = LeapHybridSampler()  # doctest: +SKIP
>>> sampleset = sampler.sample(bqm)  # doctest: +SKIP
```

**Properties**
LeapHybridSampler.properties

Solver properties as returned by a SAPI query.

Type dict

LeapHybridSampler.parameters

Solver parameters in the form of a dict, where keys are keyword parameters accepted by a SAPI query and values are lists of properties in LeapHybridSampler.properties for each key.

Type dict[str, list]

Methods

LeapHybridSampler.sample(bqm[, time_limit])

Sample from the specified binary quadratic model.

Parameters

- **bqm** (dimod.BinaryQuadraticModel) – The binary quadratic model.
- **time_limit** (int) – Maximum run time, in seconds, to allow the solver to work on the problem. Must be at least the minimum required for the number of problem variables, which is calculated and set by default. The minimum time for a hybrid solver is specified as a piecewise-linear curve defined by a set of floating-point pairs, the minimum_time_limit field under LeapHybridSampler.properties. The first element in each pair is the number of problem variables; the second is the minimum required time. The minimum time for any particular number of variables is a linear interpolation calculated on two pairs that represent the relevant range for the given number of variables. For example, if LeapHybridSampler().properties[“minimum_time_limit”] returns [[1, 0.1], [100, 10.0], [1000, 20.0]], then the minimum time for a 50-variable problem is 5 seconds, the linear interpolation of
the first two pairs that represent problems with between 1 to 100 variables.

• **kwargs – Optional keyword arguments for the solver, specified in
  LeapHybridSampler.parameters.

Returns A dimod SampleSet object.

Return type dimod.SampleSet

Examples

This example builds a random sparse graph and uses a hybrid solver to find a maximum independent set.

```python
>>> import dimod
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import numpy as np
...
>>> # Create a maximum-independent set problem from a random graph
>>> problem_node_count = 300
>>> G = nx.random_geometric_graph(problem_node_count, radius=0.0005*problem_node_count)
>>> qubo = dnx.algorithms.independent_set.maximum_weighted_independent_set_qubo(G)
>>> bqm = dimod.BQM.from_qubo(qubo)
...
>>> # Find a good solution
>>> sampler = LeapHybridSampler()  # doctest: +SKIP
>>> sampleset = sampler.sample(bqm)  # doctest: +SKIP
```

dwave.system.samplersLeapHybridSampler.sample_ising

LeapHybridSampler.sample_ising(h, J, **parameters)

Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls sample().

Parameters

• **kwargs – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also: sample(), sample_qubo()
**Sample from a QUBO using the implemented sample method.**

This method is inherited from the `Sampler` base class.

Converts the QUBO into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

- `Q (dict)` – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v` are binary-valued variables and `bias` is their associated coefficient.

- `**kwargs` – See the implemented sampling for additional keyword definitions.

**Returns** `SampleSet`  

See also: `sample()`, `sample_ising()`

---

**Composites**

dimod composites that provide layers of pre- and post-processing (e.g., `minor-embedding`) when using the D-Wave system.

---

**Contents**

- Composites
  - CutOffs
    - `CutOffComposite`
    - `PolyCutOffComposite`
  - Embedding
    - `AutoEmbeddingComposite`
    - `EmbeddingComposite`
    - `FixedEmbeddingComposite`
    - `LazyFixedEmbeddingComposite`
    - `TilingComposite`
    - `VirtualGraphComposite`
  - Reverse Anneal
    - `ReverseBatchStatesComposite`
    - `ReverseAdvanceComposite`
CutOffs

Prunes the binary quadratic model (BQM) submitted to the child sampler by retaining only interactions with values commensurate with the sampler’s precision.

CutOffComposite

class CutOffComposite(child_sampler, cutoff, cutoff_vartype=<Vartype.SPIN: frozenset({1, -1})>, comparison=<built-in function lt>)

Composite to remove interactions below a specified cutoff value.

Prunes the binary quadratic model (BQM) submitted to the child sampler by retaining only interactions with values commensurate with the sampler’s precision as specified by the cutoff argument. Also removes variables isolated post- or pre-removal of these interactions from the BQM passed on to the child sampler, setting these variables to values that minimize the original BQM’s energy for the returned samples.

Parameters

- **sampler**(dimod.Sampler) – A dimod sampler.
- **cutoff**(number) – Lower bound for absolute value of interactions. Interactions with absolute values lower than cutoff are removed. Isolated variables are also not passed on to the child sampler.
- **cutoff_vartype**(Vartype/str/set, default='SPIN') – Variable space to execute the removal in. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **comparison**(function, optional) – A comparison operator for comparing interaction values to the cutoff value. Defaults to operator.lt().

Examples

This example removes one interaction, ‘ac’: -0.7, before embedding on a D-Wave system. Note that the lowest-energy sample for the embedded problem is {‘a’: 1, ‘b’: -1, ‘c’: -1} but with a large enough number of samples (here num_reads=1000), the lowest-energy solution to the complete BQM is likely found and its energy recalculated by the composite.

```python
>>> import dimod
>>> sampler = DWaveSampler(solver={'qpu': True})
>>> bqm = dimod.BinaryQuadraticModel({'a': -1, 'b': 1, 'c': 1},
... {'ab': -0.8, 'ac': -0.7, 'bc': -1},
... 0,
... dimod.SPIN)
>>> CutOffComposite(AutoEmbeddingComposite(sampler), 0.75).sample(bqm,
... num_reads=1000).first.energy
-3.5
```

Properties
### dwave.system.composites.CutOffComposite.child

**CutOffComposite.child**

The child sampler. First sampler in `Composite.children`.

**Type** Sampler

### dwave.system.composites.CutOffComposite.children

**CutOffComposite.children**

List of child samplers that that are used by this composite.

### dwave.system.composites.CutOffComposite.properties

**CutOffComposite.properties**

A dict containing any additional information about the sampler.

### dwave.system.composites.CutOffComposite.parameters

**CutOffComposite.parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CutOffComposite.sample</strong>(bqm, <strong>parameters)</strong></td>
<td>Cut off interactions and sample from the provided binary quadratic model.</td>
</tr>
<tr>
<td><strong>CutOffComposite.sample_ising</strong>(h, J, <strong>parameters)</strong></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><strong>CutOffComposite.sample_qubo</strong>(Q, <strong>parameters)</strong></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

### dwave.system.composites.CutOffComposite.sample

**CutOffComposite.sample**(bqm, **parameters)**

Cut off interactions and sample from the provided binary quadratic model.

Prunes the binary quadratic model (BQM) submitted to the child sampler by retaining only interactions with values commensurate with the sampler’s precision as specified by the `cutoff` argument. Also removes variables
isolated post- or pre-removal of these interactions from the BQM passed on to the child sampler, setting these variables to values that minimize the original BQM’s energy for the returned samples.

Parameters
- `bqm(dimod.BinaryQuadraticModel)` – Binary quadratic model to be sampled from.
- **parameters** – Parameters for the sampling method, specified by the child sampler.

Returns `dimod.SampleSet` 

Examples
See the example in `CutOffComposite`.

**dwave.system.composites.CutOffComposite.sample_ising**

`CutOffComposite.sample_ising(h, J, **parameters)` Sample from an Ising model using the implemented sample method. This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

Parameters
- `h` (`dict/list`) – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **kwargs – See the implemented sampling for additional keyword definitions.

Returns `SampleSet`

See also:
`sample()`, `sample_qubo()`

**dwave.system.composites.CutOffComposite.sample_qubo**

`CutOffComposite.sample_qubo(Q, **parameters)` Sample from a QUBO using the implemented sample method. This method is inherited from the `Sampler` base class.

Converts the QUBO into a `BinaryQuadraticModel` and then calls `sample()`.

Parameters
- `Q` (`dict`) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u`, `v`, are binary-valued variables and `bias` is their associated coefficient.
- **kwargs – See the implemented sampling for additional keyword definitions.

Returns `SampleSet`
See also:

```
sample(), sample_ising()
```

**PolyCutOffComposite**

Prunes the polynomial submitted to the child sampler by retaining only interactions with values commensurate with the sampler’s precision.

```python
class PolyCutOffComposite(child_sampler, cutoff, cutoff_vartype=<Vartype.SPIN: frozenset({1, -1}), comparison=<built-in function lt>)
```

Composite to remove polynomial interactions below a specified cutoff value.

Prunes the binary polynomial submitted to the child sampler by retaining only interactions with values commensurate with the sampler’s precision as specified by the `cutoff` argument. Also removes variables isolated post- or pre-removal of these interactions from the polynomial passed on to the child sampler, setting these variables to values that minimize the original polynomial’s energy for the returned samples.

**Parameters**

- `sampler (dimod.PolySampler)` – A dimod binary polynomial sampler.
- `cutoff (number)` – Lower bound for absolute value of interactions. Interactions with absolute values lower than `cutoff` are removed. Isolated variables are also not passed on to the child sampler.
- `cutoff_vartype (Vartype/str/set, default='SPIN')` – Variable space to do the cutoff in. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- `comparison (function, optional)` – A comparison operator for comparing the interaction value to the cutoff value. Defaults to `operator.lt()`.

**Examples**

This example removes one interaction, ‘ac’: 0.2, before submitting the polynomial to child sampler ExactSolver().

```python
>>> import dimod
>>> sampler = dimod.HigherOrderComposite(dimod.ExactSolver())
>>> poly = dimod.BinaryPolynomial({'a': 3, 'abc':-4, 'ac': 0.2}, dimod.SPIN)
>>> PolyCutOffComposite(sampler, 1).sample_poly(poly).first.sample['a']
-1
```

**Properties**

- `PolyCutOffComposite.child` The child sampler.
- `PolyCutOffComposite.children` List of child samplers that that are used by this composite.
- `PolyCutOffComposite.properties` A dict containing any additional information about the sampler.
Table 145 – continued from previous page

| PolyCutOffComposite.parameters | A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter. |

**dwave.system.composites.PolyCutOffComposite.child**

PolyCutOffComposite.child

The child sampler. First sampler in Composite.children.

Type: Sampler

**dwave.system.composites.PolyCutOffComposite.children**

PolyCutOffComposite.children

List of child samplers that are used by this composite.

**dwave.system.composites.PolyCutOffComposite.properties**

PolyCutOffComposite.properties

A dict containing any additional information about the sampler.

**dwave.system.composites.PolyCutOffComposite.parameters**

PolyCutOffComposite.parameters

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

**Methods**

| PolyCutOffComposite.sample_poly(poly, **kwargs) | Cutoff and sample from the provided binary polynomial. |
| PolyCutOffComposite.sample_hising(h, J, **kwargs) | Sample from a higher-order Ising model. |
| PolyCutOffComposite.sample_huboc(H, **kwargs) | Sample from a higher-order unconstrained binary optimization problem. |

**dwave.system.composites.PolyCutOffComposite.sample_poly**

PolyCutOffComposite.sample_poly(poly, **kwargs)

Cutoff and sample from the provided binary polynomial.

Prunes the binary polynomial submitted to the child sampler by retaining only interactions with values commensurate with the sampler’s precision as specified by the cutoff argument. Also removes variables isolated post- or pre-removal of these interactions from the polynomial passed on to the child sampler, setting these variables to values that minimize the original polynomial’s energy for the returned samples.

**Parameters**

- poly (dimod.BinaryPolynomial) – Binary polynomial to be sampled from.
**parameters** – Parameters for the sampling method, specified by the child sampler.

Returns `dimod.SampleSet`

Examples

See the example in `PolyCutOffComposite`.

dwave.system.composites.PolyCutOffComposite.sample_hising

`PolyCutOffComposite.sample_hising(h, J, **kwargs)`

Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a `BinaryPolynomial` and call `sample_poly()`.

Parameters

- **h (dict)** – Variable biases of the Ising problem as a dict of the form `{v: bias, ...}`, where `v` is a variable in the polynomial and `bias` its associated coefficient.
- **J (dict)** – Interaction biases of the Ising problem as a dict of the form `{(u, v, ...): bias}`, where `u, v` are spin-valued variables in the polynomial and `bias` their associated coefficient.
- **kwargs** – See `sample_poly()` for additional keyword definitions.

Returns `SampleSet`

See also:

`sample_poly(), sample_hubo()`

dwave.system.composites.PolyCutOffComposite.sample_hubo

`PolyCutOffComposite.sample_hubo(H, **kwargs)`

Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a `BinaryPolynomial` and then call `sample_poly()`.

Parameters

- **H (dict)** – Coefficients of the HUBO as a dict of the form `{(u, v, ...): bias, ...}`, where `u, v` are binary-valued variables in the polynomial and `bias` their associated coefficient.
- **kwargs** – See `sample_poly()` for additional keyword definitions.

Returns `SampleSet`

See also:

`sample_poly(), sample_hising()`

Embedding

*Minor-embed* a problem `BQM` into a D-Wave system.

Embedding composites for various types of problems and application. For example:

- `EmbeddingComposite` for a problem with arbitrary structure that likely requires heuristic embedding.
• AutoEmbeddingComposite can save unnecessary embedding for problems that might have a structure similar to the child sampler.
• LazyFixedEmbeddingComposite can benefit applications that resubmit a BQM with changes in some values.

**AutoEmbeddingComposite**

class AutoEmbeddingComposite(child_sampler, **kwargs)
Maps problems to a structured sampler, embedding if needed.

This composite first tries to submit the binary quadratic model directly to the child sampler and only embeds if a `dimod.exceptions.BinaryQuadraticModelStructureError` is raised.

**Parameters**

- **sampler** (`dimod.Sampler`) – Structured dimod sampler, such as a `DWaveSampler()`.
- **find_embedding** (function, optional) – A function `find_embedding(S, T, **kwargs)` where `S` and `T` are edgelists. The function can accept additional keyword arguments. Defaults to `minorminer.find_embedding()`.
- **kwargs** – See the EmbeddingComposite class for additional keyword arguments.

**Properties**

<table>
<thead>
<tr>
<th>AutoEmbeddingComposite.child</th>
<th>The child sampler.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoEmbeddingComposite.parameters</td>
<td>None</td>
</tr>
<tr>
<td>AutoEmbeddingComposite.properties</td>
<td>None</td>
</tr>
</tbody>
</table>

dwave.system.composites.AutoEmbeddingComposite.child

AutoEmbeddingComposite.child

The child sampler. First sampler in Composite.children.

Type `Sampler`

dwave.system.composites.AutoEmbeddingComposite.parameters

AutoEmbeddingComposite.parameters = None

dwave.system.composites.AutoEmbeddingComposite.properties

AutoEmbeddingComposite.properties = None

**Methods**
**AutoEmbeddingComposite.sample**

Sample from the provided binary quadratic model.

**Parameters**

- **bqm** (*dimod.BinaryQuadraticModel*) - Binary quadratic model to be sampled from.
- **chain_strength** (*float, optional, default=1.0*) - Magnitude of the quadratic bias (in SPIN-space) applied between variables to create chains. The energy penalty of chain breaks is 2 * chain_strength.
- **chain_break_method** (*function/list, optional*) - Method or methods used to resolve chain breaks. If multiple methods are given, the results are concatenated and a new field called “chain_break_method” specifying the index of the method is appended to the sample set. See unembed_sampleset() and dwave.embedding.chain_breaks.
- **chain_break_fraction** (*bool, optional, default=True*) - Add a chain_break_fraction field to the unembedded response with the fraction of chains broken before unembedding.
- **embedding_parameters** (*dict, optional*) - If provided, parameters are passed to the embedding method as keyword arguments. Overrides any embedding_parameters passed to the constructor.
- **return_embedding** (*bool, optional*) - If True, the embedding, chain strength, chain break method and embedding parameters are added to *dimod.SampleSet.info* of the returned sample set. The default behaviour is defined by return_embedding_default, which itself defaults to False.
- **warnings** (*WarningAction, optional*) - Defines what warning action to take, if any. See warnings. The default behaviour is defined by warnings_default, which itself defaults to IGNORE.
- ****parameters** - Parameters for the sampling method, specified by the child sampler.

**Returns** *dimod.SampleSet*

**Examples**

See the example in *EmbeddingComposite*.

**AutoEmbeddingComposite.sample_ising**

Sample from an Ising model using the implemented sample method.

**AutoEmbeddingComposite.sample_qubo**

Sample from a QUBO using the implemented sample method.
This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

- `h` (dict/list) – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where `v` is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

- `J` (dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.

- `**kwargs` – See the implemented sampling for additional keyword definitions.

**Returns** `SampleSet`

See also:

`sample()`, `sample_qubo()`

dwave.system.composites.AutoEmbeddingComposite.sample_qubo

`AutoEmbeddingComposite.sample_qubo(Q, **parameters)`

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the QUBO into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

- `Q` (dict) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v` are binary-valued variables and `bias` is their associated coefficient.

- `**kwargs` – See the implemented sampling for additional keyword definitions.

**Returns** `SampleSet`

See also:

`sample()`, `sample_ising()`

**EmbeddingComposite**

class `EmbeddingComposite` (child_sampler, find_embedding=<function find_embedding>, embedding_parameters=None, scale_aware=False, child_structure_search=<function child_structure_dfs>)

Maps problems to a structured sampler.

Automatically minor-embeds a problem into a structured sampler such as a D-Wave system. A new minor-embedding is calculated each time one of its sampling methods is called.

**Parameters**

- `child_sampler` (dimod.Sampler) – A dimod sampler, such as a `DWaveSampler`, that accepts only binary quadratic models of a particular structure.

- `find_embedding` (function, optional) – A function `find_embedding(S, T, **kwargs)` where `S` and `T` are edgelists. The function can accept additional keyword arguments. Defaults to `minorminer.find_embedding()`.
- **embedding_parameters** *(dict, optional)* – If provided, parameters are passed to the embedding method as keyword arguments.

- **scale_aware** *(bool, optional, default=False)* – Pass chain interactions to child samplers that accept an `ignored_interactions` parameter.

- **child_structure_search** *(function, optional)* – A function `child_structure_search(sampler)` that accepts a sampler and returns the `dimod.StructuredStructure`. Defaults to `dimod.child_structure_dfs()`.

### Examples

```python
>>> from dwave.system import DWaveSampler, EmbeddingComposite
... >> s = EmbeddingComposite(DWaveSampler())
>>> h = {'a': -1., 'b': 2}
>>> J = {('a', 'b'): 1.5}
>>> sampleset = s.sample_ising(h, J, num_reads=100)
>>> sampleset.first.energy
-4.5
```

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EmbeddingComposite.child</code></td>
<td>The child sampler.</td>
</tr>
<tr>
<td><code>EmbeddingComposite.parameters</code></td>
<td>Parameters in the form of a dict.</td>
</tr>
<tr>
<td><code>EmbeddingComposite.properties</code></td>
<td>Properties in the form of a dict.</td>
</tr>
<tr>
<td><code>EmbeddingComposite.return_embedding_default</code></td>
<td>Defines the default behaviour for <code>sample()</code>'s <code>return_embedding</code> kwarg.</td>
</tr>
<tr>
<td><code>EmbeddingComposite.warnings_default</code></td>
<td>Defines the default behavior for <code>sample()</code>'s <code>warnings</code> kwarg.</td>
</tr>
</tbody>
</table>

**`dwave.system.composites.EmbeddingComposite.child`**

EmbeddingComposite.child

The child sampler. First sampler in Composite.children.

Type `Sampler`

**`dwave.system.composites.EmbeddingComposite.parameters`**

EmbeddingComposite.parameters = None

Parameters in the form of a dict.

For an instantiated composed sampler, keys are the keyword parameters accepted by the child sampler and parameters added by the composite.

Type `dict[str, list]`
**EmbeddingComposite.properties**

*EmbeddingComposite.properties = None*

Properties in the form of a dict.

Contains the properties of the child sampler.

**Type**: dict

**EmbeddingComposite.return_embedding_default**

*EmbeddingComposite.return_embedding_default = True*

Defines the default behaviour for `sample()`’s `return_embedding` kwarg.

**EmbeddingComposite.warnings_default**

*EmbeddingComposite.warnings_default = 'save'*

Defines the default behaviour for `sample()`’s `warnings` kwarg.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EmbeddingComposite.sample(bqm[, ...])</code></td>
<td>Sample from the provided binary quadratic model.</td>
</tr>
<tr>
<td><code>EmbeddingComposite.sample_ising(h, J, ...)</code></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><code>EmbeddingComposite.sample_qubo(Q, **parameters)</code></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

**EmbeddingComposite.sample**

*EmbeddingComposite.sample (bqm, chain_strength=1.0, chain_break_method=None, chain_break_fraction=True, embedding_parameters=None, return_embedding=None, warnings=None, **parameters)*

Sample from the provided binary quadratic model.

**Parameters**

- **bqm** (*dimod.BinaryQuadraticModel*) – Binary quadratic model to be sampled from.

- **chain_strength** (*float, optional, default=1.0*) – Magnitude of the quadratic bias (in SPIN-space) applied between variables to create chains. The energy penalty of chain breaks is \(2 \times \text{chain_strength}\).

- **chain_break_method** (*function/list, optional*) – Method or methods used to resolve chain breaks. If multiple methods are given, the results are concatenated and a new field called “chain_break_method” specifying the index of the method is appended to the sample set. See `unembed_sampleset()` and `dwave.embedding.chain_breaks`.

- **chain_break_fraction** (*bool, optional, default=True*) – Add a `chain_break_fraction` field to the unembedded response with the fraction of chains broken before unembedding.

- **embedding_parameters** (*dict, optional*) – If provided, parameters are passed
to the embedding method as keyword arguments. Overrides any embedding_parameters passed to the constructor.

- **return_embedding** (bool, optional) – If True, the embedding, chain strength, chain break method and embedding parameters are added to dimod.
  SampleSet.info of the returned sample set. The default behaviour is defined by return_embedding_default, which itself defaults to False.

- **warnings** (WarningAction, optional) – Defines what warning action to take, if any.
  See warnings. The default behaviour is defined by warnings_default, which itself
defaults to IGNORE

- ****parameters – Parameters for the sampling method, specified by the child sampler.

Returns dimod.SampleSet

Examples
See the example in EmbeddingComposite.

dwave.system.composites.EmbeddingComposite.sample_ising

EmbeddingComposite.sample_ising(h, J, **parameters)
Sample from an Ising model using the implemented sample method.
This method is inherited from the Sampler base class.
Converts the Ising model into a BinaryQuadraticModel and then calls sample().

Parameters

- **h** (dict/list) – Linear biases of the Ising problem. If a dict, should be of the form {v: bias, ...} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

- **J** (dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.

- ****kwargs – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also:
sample(), sample_qubo()
FixedEmbeddingComposite

class FixedEmbeddingComposite(child_sampler, embedding=None, source_adjacency=None, **kwargs)

Maps problems to a structured sampler with the specified minor-embedding.

Parameters

- `sampler` (dimod.Sampler) – Structured dimod sampler such as a D-Wave system.
- `embedding` (dict[hashable, iterable], optional) – Mapping from a source graph to the specified sampler’s graph (the target graph).
- `source_adjacency` (dict[hashable, iterable]) – Deprecated. Dictionary to describe source graph. Ex. `{node: [node neighbours]}`.
- `**kwargs` – See the EmbeddingComposite class for additional keyword arguments. Note that find_embedding and embedding_parameters keyword arguments are ignored.

Examples

```python
>>> from dwave.system import DWaveSampler, FixedEmbeddingComposite
...  
>>> embedding = {'a': [0, 4], 'b': [1, 5], 'c': [2, 6]}
>>> sampler = FixedEmbeddingComposite(DWaveSampler(), embedding)
>>> sampler.nodelist
['a', 'b', 'c']
>>> sampler.edgelist
[('a', 'b'), ('a', 'c'), ('b', 'c')]
>>> sampleset = sampler.sample_ising({'a': .5, 'c': 0}, {('a', 'c'): -1}, num_reads=500)
>>> sampleset.first.energy
-1.5
```

Properties

- `FixedEmbeddingComposite.properties`
- `FixedEmbeddingComposite.parameters`
- `FixedEmbeddingComposite.children`
- `FixedEmbeddingComposite.child` – The child sampler.
- `FixedEmbeddingComposite.nodelist` – Nodes available to the composed sampler.
- `FixedEmbeddingComposite.edgelist` – Edges available to the composed sampler.
- `FixedEmbeddingComposite.adjacency` – Adjacency structure for the composed sampler.
Table 151 – continued from previous page

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FixedEmbeddingComposite.structure</code></td>
<td>Structure of the structured sampler formatted as a namedtuple, <code>Structure(nodelist, edgelist, adjacency)</code>, where the 3-tuple values are the <code>nodelist</code>, <code>edgelist</code> and <code>adjacency</code> attributes.</td>
</tr>
</tbody>
</table>

```python
dwave.system.composites.FixedEmbeddingComposite.properties

FixedEmbeddingComposite.properties = None

dwave.system.composites.FixedEmbeddingComposite.parameters

FixedEmbeddingComposite.parameters = None

dwave.system.composites.FixedEmbeddingComposite.children

FixedEmbeddingComposite.children = None

dwave.system.composites.FixedEmbeddingComposite.child

FixedEmbeddingComposite.child
    The child sampler. First sampler in Composite.children.
        Type `Sampler`

dwave.system.composites.FixedEmbeddingComposite.nodelist

FixedEmbeddingComposite.nodelist
    Nodes available to the composed sampler.
        Type `list`

dwave.system.composites.FixedEmbeddingComposite.edgelist

FixedEmbeddingComposite.edgelist
    Edges available to the composed sampler.
        Type `list`

dwave.system.composites.FixedEmbeddingComposite.adjacency

FixedEmbeddingComposite.adjacency
    Adjacency structure for the composed sampler.
        Type `dict[variable, set]`
```
**FixedEmbeddingComposite.structure**

Structure of the structured sampler formatted as a namedtuple, `Structure(nodelist, edgelist, adjacency)`, where the 3-tuple values are the `nodelist`, `edgelist` and `adjacency` attributes.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FixedEmbeddingComposite.sample(bqm, **parameters)</code></td>
<td>Sample the binary quadratic model.</td>
</tr>
<tr>
<td><code>FixedEmbeddingComposite.sample_ising(h, J, ...)</code></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><code>FixedEmbeddingComposite.sample_qubo(Q, ...)</code></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

**FixedEmbeddingComposite.sample**

Sample the binary quadratic model.

On the first call of a sampling method, finds a *minor-embedding* for the given binary quadratic model (BQM). All subsequent calls to its sampling methods reuse this embedding.

**Parameters**

- `bqm` (*dimod.BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- `chain_strength` (*float, optional, default=1.0*) – Magnitude of the quadratic bias (in SPIN-space) applied between variables to create chains. The energy penalty of chain breaks is 2 * `chain_strength`.
- `chain_break_method` (*function, optional*) – Method used to resolve chain breaks during sample unembedding. See `unembed_sampleset()`.
- `chain_break_fraction` (*bool, optional, default=True*) – Add a `chain_break_fraction` field to the unembedded response with the fraction of chains broken before unembedding.
- `embedding_parameters` (*dict, optional*) – If provided, parameters are passed to the embedding method as keyword arguments. Overrides any `embedding_parameters` passed to the constructor. Only used on the first call.
- **parameters** – Parameters for the sampling method, specified by the child sampler.

**Returns** `dimod.SampleSet`

**FixedEmbeddingComposite.sample_ising**

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**
• **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where `v` is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

• **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.

• **kwargs** – See the implemented sampling for additional keyword definitions.

Returns **SampleSet**

See also: `sample()`, `sample_qubo()`

dwave.system.composites.FixedEmbeddingComposite.sample_qubo

FixedEmbeddingComposite.sample_qubo(Q, **parameters)

Sample from a QUBO using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls `sample()`.

Parameters

• **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v` are binary-valued variables and `bias` is their associated coefficient.

• **kwargs** – See the implemented sampling for additional keyword definitions.

Returns **SampleSet**

See also: `sample()`, `sample_ising()`

LazyFixedEmbeddingComposite

class LazyFixedEmbeddingComposite(child_sampler, find_embedding=<function find_embedding>, embedding_parameters=None, scale_aware=False, child_structure_search=<function child_structure_dfs>)

Maps problems to the structure of its first given problem.

This composite reuses the minor-embedding found for its first given problem without recalculating a new minor-embedding for subsequent calls of its sampling methods.

Parameters

• **sampler** (*dimod.Sampler*) – Structured dimod sampler.

• **find_embedding** (function, default=:`func: minorminer.find_embedding`) – A function `find_embedding(S, T, **kwargs)` where `S` and `T` are edgelists. The function can accept additional keyword arguments. The function is used to find the embedding for the first problem solved.

• **embedding_parameters** (*dict, optional*) – If provided, parameters are passed to the embedding method as keyword arguments.
Examples

```python
>>> from dwave.system import LazyFixedEmbeddingComposite, DWaveSampler
...
>>> sampler = LazyFixedEmbeddingComposite(DWaveSampler())
>>> sampler.nodelist is None  # no structure prior to first sampling
True
>>> __ = sampler.sample_ising({}, {('a', 'b'): -1})
>>> sampler.nodelist  # has structure based on given problem
['a', 'b']
```

Properties

| LazyFixedEmbeddingComposite. parameters | Nodes available to the composed sampler. |
| LazyFixedEmbeddingComposite. properties | Edges available to the composed sampler. |
| LazyFixedEmbeddingComposite.nodelist | Adjacency structure for the composed sampler. |
| LazyFixedEmbeddingComposite.edgelist | Structure of the structured sampler formatted as a namedtuple, Structure(nodelist, edgelist, adjacency), where the 3-tuple values are the nodelist, edgelist and adjacency attributes. |

```python
LazyFixedEmbeddingComposite.parameters = None
```

```python
LazyFixedEmbeddingComposite.properties = None
```

```python
LazyFixedEmbeddingComposite.nodelist
Nodes available to the composed sampler.
    Type  list
```

```python
LazyFixedEmbeddingComposite.edgelist
Edges available to the composed sampler.
    Type  list
```
**dwave.system.composites.LazyFixedEmbeddingComposite.adjacency**

LazyFixedEmbeddingComposite.**adjacency**

Adjacency structure for the composed sampler.

**Type**  
dict[variable, set]

**dwave.system.composites.LazyFixedEmbeddingComposite.structure**

LazyFixedEmbeddingComposite.**structure**

Structure of the structured sampler formatted as a namedtuple, *Structure(nodelist, edgelist, adjacency)*, where the 3-tuple values are the *nodelist, edgelist* and *adjacency* attributes.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LazyFixedEmbeddingComposite.<strong>sample</strong>(bqm, ...)</td>
<td>Sample the binary quadratic model.</td>
</tr>
<tr>
<td>LazyFixedEmbeddingComposite.<strong>sample_ising</strong>(h, ...)</td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td>LazyFixedEmbeddingComposite.<strong>sample_qubo</strong>(Q, ...)</td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

**dwave.system.composites.LazyFixedEmbeddingComposite.sample**

LazyFixedEmbeddingComposite.**sample**(bqm, **parameters**)

Sample the binary quadratic model.

On the first call of a sampling method, finds a *minor-embedding* for the given binary quadratic model (BQM). All subsequent calls to its sampling methods reuse this embedding.

**Parameters**

- **bqm**(dimod.BinaryQuadraticModel) – Binary quadratic model to be sampled from.
- **chain_strength**(float, optional, default=1.0) – Magnitude of the quadratic bias (in SPIN-space) applied between variables to create chains. The energy penalty of chain breaks is 2 * chain_strength.
- **chain_break_method**(function, optional) – Method used to resolve chain breaks during sample unembedding. See unembed_sampleset().
- **chain_break_fraction**(bool, optional, default=True) – Add a ‘chain_break_fraction’ field to the unembedded response with the fraction of chains broken before unembedding.
- **embedding_parameters**(dict, optional) – If provided, parameters are passed to the embedding method as keyword arguments. Overrides any embedding_parameters passed to the constructor. Only used on the first call.
- ****parameters** – Parameters for the sampling method, specified by the child sampler.

**Returns** dimod.SampleSet
**Sample from an Ising model using the implemented sample method.**

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

- `h (dict/list)` – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where `v` is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

- `J (dict[(variable, variable), bias])` – Quadratic biases of the Ising problem.

- `**kwargs` – See the implemented sampling for additional keyword definitions.

**Returns** `SampleSet`

See also:

`sample()`, `sample_qubo()`

**Sample from a QUBO using the implemented sample method.**

This method is inherited from the `Sampler` base class.

Converts the QUBO into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

- `Q (dict)` – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v` are binary-valued variables and `bias` is their associated coefficient.

- `**kwargs` – See the implemented sampling for additional keyword definitions.

**Returns** `SampleSet`

See also:

`sample()`, `sample_ising()`

**Composite to tile a small problem across a Chimera-structured sampler.**

Enables parallel sampling for small problems (problems that are minor-embeddable in a small part of a D-Wave solver’s Chimera graph).

Notation `CN` refers to a Chimera graph consisting of an NxN grid of unit cells, where each unit cell is a bipartite graph with shores of size `t`. The D-Wave 2000Q QPU supports a C16 Chimera graph: its 2048 qubits are logically mapped into a 16x16 matrix of unit cell of 8 qubits (`t=4`).
A problem that can be minor-embedded in a single unit cell, for example, can therefore be tiled across the unit cells of a D-Wave 2000Q as 16x16 duplicates. This enables sampling 256 solutions in a single call.

Parameters

- **sampler** *(dimod.Sampler)* — Structured dimod sampler such as a DWaveSampler().
- **sub_m** *(int)* — Number of rows of Chimera unit cells for minor-embedding the problem once.
- **sub_n** *(int)* — Number of columns of Chimera unit cells for minor-embedding the problem once.
- **t** *(int, optional, default=4)* — Size of the shore within each Chimera unit cell.

Examples

This example submits a two-variable QUBO problem representing a logical NOT gate to a D-Wave system. The QUBO—two nodes with biases of -1 that are coupled with strength 2—needs only any two coupled qubits and so is easily minor-embedded in a single unit cell. Composite TilingComposite tiles it multiple times for parallel solution: the two nodes should typically have opposite values.

```python
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> from dwave.system import TilingComposite
...
>>> sampler = EmbeddingComposite(TilingComposite(DWaveSampler(), 1, 1, 4))
>>> Q = {(1, 1): -1, (1, 2): 2, (2, 1): 0, (2, 2): -1}
>>> sampleset = sampler.sample_qubo(Q)
>>> len(sampleset) > 1
True
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TilingComposite.properties</td>
<td>Properties in the form of a dict.</td>
</tr>
<tr>
<td>TilingComposite.parameters</td>
<td>Parameters in the form of a dict.</td>
</tr>
<tr>
<td>TilingComposite.children</td>
<td>The single wrapped structured sampler.</td>
</tr>
<tr>
<td>TilingComposite.child</td>
<td>The child sampler.</td>
</tr>
<tr>
<td>TilingComposite.nodelist</td>
<td>List of active qubits for the structured solver.</td>
</tr>
<tr>
<td>TilingComposite.edgelist</td>
<td>List of active couplers for the D-Wave solver.</td>
</tr>
<tr>
<td>TilingComposite.adjacency</td>
<td>Adjacency structure formatted as a dict, where keys are the nodes of the structured sampler and values are sets of all adjacent nodes for each key node.</td>
</tr>
<tr>
<td>TilingComposite.structure</td>
<td>Structure of the structured sampler formatted as a namedtuple, Structure(nodelist, edgelist, adjacency), where the 3-tuple values are the nodelist, edgelist and adjacency attributes.</td>
</tr>
</tbody>
</table>

```
TilingComposite.properties = None
Properties in the form of a dict.
```

6.8. dwave-system
Type `dict`

dwave.system.composites.TilingComposite.parameters

TilingComposite.parameters = None
Parameters in the form of a dict.
  Type `dict[str, list]`

dwave.system.composites.TilingComposite.children

TilingComposite.children = None
The single wrapped structured sampler.
  Type `list`

dwave.system.composites.TilingComposite.child

TilingComposite.child
The child sampler. First sampler in Composite.children.
  Type `Sampler`

dwave.system.composites.TilingComposite.nodelist

TilingComposite.nodelist = None
List of active qubits for the structured solver.
  Type `list`

dwave.system.composites.TilingComposite.edgelist

TilingComposite.edgelist = None
List of active couplers for the D-Wave solver.
  Type `list`

dwave.system.composites.TilingComposite.adjacency

TilingComposite.adjacency
Adjacency structure formatted as a dict, where keys are the nodes of the structured sampler and values are sets of all adjacent nodes for each key node.
  Type `dict[variable, set]`

dwave.system.composites.TilingComposite.structure

TilingComposite.structure
Structure of the structured sampler formatted as a namedtuple, Structure(nodelist, edgelist, adjacency), where the 3-tuple values are the nodelist, edgelist and adjacency attributes.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TilingComposite.sample(bqm, **kwargs)</code></td>
<td>Sample from the specified binary quadratic model.</td>
</tr>
<tr>
<td><code>TilingComposite.sample_ising(h, J, **parameters)</code></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><code>TilingComposite.sample_qubo(Q, **parameters)</code></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

### dwave.system.composites.TilingComposite.sample

**Signature**

```
TilingComposite.sample(bqm, **kwargs)
```

**Description**

Sample from the specified binary quadratic model.

**Parameters**

- `bqm` ([`dimod.BinaryQuadraticModel`]) – Binary quadratic model to be sampled from.
- `**kwargs` – Optional keyword arguments for the sampling method, specified per solver.

**Returns**

`dimod.SampleSet`

**Examples**

This example submits a simple Ising problem of just two variables on a D-Wave system. Because the problem fits in a single Chimera unit cell, it is tiled across the solver’s entire Chimera graph, resulting in multiple samples (the exact number depends on the working Chimera graph of the D-Wave system).

```python
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> from dwave.system import TilingComposite
... >>> sampler = EmbeddingComposite(TilingComposite(DWaveSampler(), 1, 1, 4))
>>> response = sampler.sample_ising({},{('a', 'b'): 1})
>>> len(response)  # doctest: +SKIP
246
```

See [Ocean Glossary](#) for explanations of technical terms in descriptions of Ocean tools.

### dwave.system.composites.TilingComposite.sample_ising

**Signature**

```
TilingComposite.sample_ising(h, J, **parameters)
```

**Description**

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

- `h` (dict/list) – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, …}` where `v` is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- `J` (dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.
- `**kwargs` – See the implemented sampling for additional keyword definitions.
Returns `SampleSet`

See also:

`sample()`, `sample_qubo()`

dwave.system.composites.TilingComposite.sample_qubo

TilingComposite.sample_qubo(Q, **parameters)

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the QUBO into a `BinaryQuadraticModel` and then calls `sample()`.

Parameters

- `Q (dict)` — Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, ...}` where `u, v,` are binary-valued variables and `bias` is their associated coefficient.

- `**kwargs` — See the implemented sampling for additional keyword definitions.

Returns `SampleSet`

See also:

`sample()`, `sample_ising()`

VirtualGraphComposite

class VirtualGraphComposite(sampler, embedding, chain_strength=None, flux_biases=None, flux_bias_num_reads=1000, flux_bias_max_age=3600)

Composite to use the D-Wave virtual graph feature for minor-embedding.

Calibrates qubits in chains to compensate for the effects of biases and enables easy creation, optimization, use, and reuse of an embedding for a given working graph.

Inherits from `dimod.ComposedSampler` and `dimod.Structured`.

Parameters

- `sampler (DWaveSampler)` — A dimod `dimod.Sampler`. Typically a `DWaveSampler` or derived composite sampler; other samplers may not work or make sense with this composite layer.

- `embedding (dict[hashable, iterable])` — Mapping from a source graph to the specified sampler’s graph (the target graph).

- `chain_strength (float, optional, default=None)` — Desired chain coupling strength. This is the magnitude of couplings between qubits in a chain. If None, uses the maximum available as returned by a SAPI query to the D-Wave solver.

- `flux_biases (list/False/None, optional, default=None)` — Per-qubit flux bias offsets in the form of a list of lists, where each sublist is of length 2 and specifies a variable and the flux bias offset associated with that variable. Qubits in a chain with strong negative J values experience a J-induced bias; this parameter compensates by recalibrating to remove that bias. If False, no flux bias is applied or calculated. If None, flux biases are pulled from the database or calculated empirically.
• **flux_bias_num_reads** *(int, optional, default=1000)* – Number of samples to collect per flux bias value to calculate calibration information.

• **flux_bias_max_age** *(int, optional, default=3600)* – Maximum age (in seconds) allowed for a previously calculated flux bias offset to be considered valid.

**Attention:** D-Wave’s virtual graphs feature can require many seconds of D-Wave system time to calibrate qubits to compensate for the effects of biases. If your account has limited D-Wave system access, consider using `FixedEmbeddingComposite()` instead.

### Examples

This example uses `VirtualGraphComposite` to instantiate a composed sampler that submits a QUBO problem to a D-Wave solver. The problem represents a logical AND gate using penalty function \( P = xy - 2(x + y)z + 3z \), where variables \( x \) and \( y \) are the gate’s inputs and \( z \) the output. This simple three-variable problem is manually minor-embedded to a single Chimera unit cell: variables \( x \) and \( y \) are represented by qubits 1 and 5, respectively, and \( z \) by a two-qubit chain consisting of qubits 0 and 4. The chain strength is set to the maximum allowed found from querying the solver’s extended J range. In this example, the ten returned samples all represent valid states of the AND gate.

```python
>>> from dwave.system import DWaveSampler, VirtualGraphComposite
>>> embedding = {'x': {1}, 'y': {5}, 'z': {0, 4}}
>>> DWaveSampler().properties['extended_j_range']
[-2.0, 1.0]
>>> sampler = VirtualGraphComposite(DWaveSampler(), embedding, chain_strength=2)
→ # doctest: +SKIP
>>> Q = {('x', 'y'): 1, ('x', 'z'): -2, ('y', 'z'): -2, ('z', 'z'): 3}
>>> sampleset = sampler.sample_qubo(Q, num_reads=10) # doctest: +SKIP
>>> print(sampleset)
# doctest: +SKIP
{'BINARY', 4 rows, 10 samples, 3 variables}
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

### Properties

- **VirtualGraphComposite.properties**
- **VirtualGraphComposite.parameters**
- **VirtualGraphComposite.children**
- **VirtualGraphComposite.child** The child sampler.
- **VirtualGraphComposite.nodelist** Nodes available to the composed sampler.
- **VirtualGraphComposite.edgelist** Edges available to the composed sampler.
- **VirtualGraphComposite.adjacency** Adjacency structure for the composed sampler.
- **VirtualGraphComposite.structure** Structure of the structured sampler formatted as a namedtuple, `Structure(nodelist, edgelist, adjacency)`, where the 3-tuple values are the `nodelist`, `edgelist` and `adjacency` attributes.
`dwave.system.composites.VirtualGraphComposite.properties`

`VirtualGraphComposite.properties = None`

`dwave.system.composites.VirtualGraphComposite.parameters`

`VirtualGraphComposite.parameters = None`

`dwave.system.composites.VirtualGraphComposite.children`

`VirtualGraphComposite.children = None`

`dwave.system.composites.VirtualGraphComposite.child`

`VirtualGraphComposite.child`

The child sampler. First sampler in `Composite.children`.

`Type` `Sampler`

`dwave.system.composites.VirtualGraphComposite.nodelist`

`VirtualGraphComposite.nodelist`

Nodes available to the composed sampler.

`Type` `list`

`dwave.system.composites.VirtualGraphComposite.edgelist`

`VirtualGraphComposite.edgelist`

Edges available to the composed sampler.

`Type` `list`

`dwave.system.composites.VirtualGraphComposite.adjacency`

`VirtualGraphComposite.adjacency`

Adjacency structure for the composed sampler.

`Type` `dict[variable, set]`

`dwave.system.composites.VirtualGraphComposite.structure`

`VirtualGraphComposite.structure`

Structure of the structured sampler formatted as a namedtuple, `Structure(nodelist, edgelist, adjacency)`, where the 3-tuple values are the `nodelist`, `edgelist` and `adjacency` attributes.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VirtualGraphComposite.sample(bqm[, ...])</code></td>
<td>Sample from the given Ising model.</td>
</tr>
<tr>
<td><code>VirtualGraphComposite.sample_ising(h, J, ...)</code></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><code>VirtualGraphComposite.sample_qubo(Q, ...)</code></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

dwave.system.composites.VirtualGraphComposite.sample

**VirtualGraphComposite.sample** *(bqm, apply_flux_bias_offsets=True, **kwargs)*

Sample from the given Ising model.

**Parameters**

- **h** *(list/dict)* – Linear biases of the Ising model. If a list, the list’s indices are used as variable labels.
- **J** *(dict of (int, int) – float)* – Quadratic biases of the Ising model.
- **apply_flux_bias_offsets** *(bool, optional)* – If True, use the calculated flux_bias_offsets (if available).
- **kwargs** – Optional keyword arguments for the sampling method, specified per solver.

**Examples**

This example uses `VirtualGraphComposite` to instantiate a composed sampler that submits an Ising problem to a D-Wave solver. The problem represents a logical NOT gate using penalty function $P = xy$, where variable $x$ is the gate’s input and $y$ the output. This simple two-variable problem is manually minor-embedded to a single Chimera unit cell: each variable is represented by a chain of half the cell’s qubits, $x$ as qubits 0, 1, 4, 5, and $y$ as qubits 2, 3, 6, 7. The chain strength is set to half the maximum allowed found from querying the solver’s extended $J$ range. In this example, the ten returned samples all represent valid states of the NOT gate.

```python
>>> from dwave.system import DWaveSampler, VirtualGraphComposite
>>> embedding = {'x': {0, 4, 1, 5}, 'y': {2, 6, 3, 7}}
>>> DWaveSampler().properties['extended_j_range']
[-2.0, 1.0]
>>> sampler = VirtualGraphComposite(DWaveSampler(), embedding, chain_strength=1)
ROWSER SKIP
>>> h = {}
>>> J = {('x', 'y'): 1}
>>> sampleset = sampler.sample_ising(h, J, num_reads=10) # doctest: +SKIP
>>> print(sampleset)
 y energy num_oc. chain_.
 0 -1 -1.0 6 0.0
 1 +1 -1.0 4 0.0
['SPIN', 2 rows, 10 samples, 2 variables]
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.
**VirtualGraphComposite**

**VirtualGraphComposite**.

**sample_ising**(*h*, *J*, **parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the **Sampler** base class.

Converts the Ising model into a **BinaryQuadraticModel** and then calls **sample()**.

**Parameters**

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, . . .}` where `v` is a spin-valued variable and `bias` is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.


- ****kwargs – See the implemented sampling for additional keyword definitions.

**Returns** **SampleSet**

See also:

**sample()**, **sample_qubo()**

**sample_qubo**(*Q*, **parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the **Sampler** base class.

Converts the QUBO into a **BinaryQuadraticModel** and then calls **sample()**.

**Parameters**

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form `{(u, v): bias, . . .}` where `u`, `v`, are binary-valued variables and `bias` is their associated coefficient.

- ****kwargs – See the implemented sampling for additional keyword definitions.

**Returns** **SampleSet**

See also:

**sample()**, **sample_ising()**

**Reverse Anneal**

Composites that do batch operations for reverse annealing based on sets of initial states or anneal schedules.

**ReverseBatchStatesComposite**

**class ReverseBatchStatesComposite**(child_sampler)

Composite that reverse anneals from multiple initial samples. Each submission is independent from one another.

**Parameters** **sampler**(*dimod.Sampler*) – A dimod sampler.
Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReverseBatchStatesComposite.child</td>
<td>The child sampler.</td>
</tr>
<tr>
<td>ReverseBatchStatesComposite.children</td>
<td>List of child samplers that are used by this composite.</td>
</tr>
<tr>
<td>ReverseBatchStatesComposite.properties</td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
<tr>
<td>ReverseBatchStatesComposite.parameters</td>
<td>A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.</td>
</tr>
</tbody>
</table>

**dwave.system.composites.ReverseBatchStatesComposite.child**

ReverseBatchStatesComposite.child

The child sampler. First sampler in Composite.children.

    Type  Sampler

**dwave.system.composites.ReverseBatchStatesComposite.children**

ReverseBatchStatesComposite.children

List of child samplers that are used by this composite.

    Type  list[ Sampler ]

**dwave.system.composites.ReverseBatchStatesComposite.properties**

ReverseBatchStatesComposite.properties

A dict containing any additional information about the sampler.

    Type  dict

**dwave.system.composites.ReverseBatchStatesComposite.parameters**

ReverseBatchStatesComposite.parameters

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

    Type  dict

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReverseBatchStatesComposite.sample(bqm,...)</td>
<td>Sample the binary quadratic model using reverse annealing from multiple initial states.</td>
</tr>
<tr>
<td>ReverseBatchStatesComposite.sample_ising(h,...)</td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td>ReverseBatchStatesComposite.sample_qubo(Q,...)</td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>
ReverseBatchStatesComposite.sample

Sample the binary quadratic model using reverse annealing from multiple initial states.

Parameters

- **bqm** (dimod.BinaryQuadraticModel) – Binary quadratic model to be sampled from.
- **parameters** – Parameters for the sampling method, specified by the child sampler.

Returns dimod.SampleSet that has initial_state field.

ReverseBatchStatesComposite.sample_ising

Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls sample().

Parameters

- **h** (dict/list) – Linear biases of the Ising problem. If a dict, should be of the form \{v: bias, . . .\} where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (dict[(variable, variable), bias]) – Quadratic biases of the Ising problem.
- **kwargs** – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also:

sample(), sample_qubo()

ReverseBatchStatesComposite.sample_qubo

Sample from a QUBO using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the QUBO into a BinaryQuadraticModel and then calls sample().

Parameters

- **Q** (dict) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form \{(u, v): bias, . . .\} where u, v, are binary-valued variables and bias is their associated coefficient.
- **kwargs** – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also:

sample(), sample_ising()
ReverseAdvanceComposite

```python
class ReverseAdvanceComposite(child_sampler)

Composite that reverse anneals an initial sample through a sequence of anneal schedules.

If you do not specify an initial sample, a random sample is used for the first submission. By default, each subsequent submission selects the most-found lowest-energy sample as its initial state. If you set reinitialize_state to False, which makes each submission behave like a random walk, the subsequent submission selects the last returned sample as its initial state.

Parameters

- **sampler** *(dimod.Sampler)*
  - A dimod sampler.
```

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>child</code></td>
<td>The child sampler.</td>
</tr>
<tr>
<td><code>children</code></td>
<td>List of child samplers that that are used by this composite.</td>
</tr>
<tr>
<td><code>properties</code></td>
<td>A dict containing any additional information about the sampler.</td>
</tr>
<tr>
<td><code>parameters</code></td>
<td>A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.</td>
</tr>
</tbody>
</table>

```python
dwave.system.composites.ReverseAdvanceComposite.child

ReverseAdvanceComposite.child

The child sampler. First sampler in Composite.children.

Type: `Sampler`

``` python
dwave.system.composites.ReverseAdvanceComposite.children

ReverseAdvanceComposite.children

List of child samplers that that are used by this composite.

Type: `list[Sampler]`

``` python
dwave.system.composites.ReverseAdvanceComposite.properties

ReverseAdvanceComposite.properties

A dict containing any additional information about the sampler.

Type: `dict`

``` python
dwave.system.composites.ReverseAdvanceComposite.parameters

ReverseAdvanceComposite.parameters

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
**Type**

dict

**Methods**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ReverseAdvanceComposite.sample(bqm, ...)</code></td>
<td>Sample the binary quadratic model using reverse annealing along a given set of anneal schedules.</td>
</tr>
<tr>
<td><code>ReverseAdvanceComposite.sample_ising(h, J, ...)</code></td>
<td>Sample from an Ising model using the implemented sample method.</td>
</tr>
<tr>
<td><code>ReverseAdvanceComposite.sample_qubo(Q, ...)</code></td>
<td>Sample from a QUBO using the implemented sample method.</td>
</tr>
</tbody>
</table>

**dwave.system.composites.ReverseAdvanceComposite.sample**

`ReverseAdvanceComposite.sample(bqm, anneal_schedules=None, **parameters)`

Sample the binary quadratic model using reverse annealing along a given set of anneal schedules.

**Parameters**

- `bqm (dimod.BinaryQuadraticModel)` – Binary quadratic model to be sampled from.
- `anneal_schedules (list of lists)` – Anneal schedules in order of submission. Each schedule is formatted as a list of [time, s] pairs.
- `initial_state (dict, optional)` – the state to reverse anneal from. If not provided, it will be randomly generated.
- `**parameters` – Parameters for the sampling method, specified by the child sampler.

**Returns**

`dimod.SampleSet` that has initial_state and schedule_index fields.

**dwave.system.composites.ReverseAdvanceComposite.sample_ising**

`ReverseAdvanceComposite.sample_ising(h, J, **parameters)`

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

**Parameters**

- `h (dict/list)` – Linear biases of the Ising problem. If a dict, should be of the form `{v: bias, ...}` where is a spin-valued variable and bias is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- `J (dict[(variable, variable), bias])` – Quadratic biases of the Ising problem.
- `**kwargs` – See the implemented sampling for additional keyword definitions.

**Returns**

`SampleSet`

See also:

`sample(), sample_qubo()`
ReverseAdvanceComposite.sample_qubo(*parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the QUBO into a BinaryQuadraticModel and then calls sample().

Parameters

- **Q** (dict) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form \{(u, v): bias, \ldots\} where u, v are binary-valued variables and bias is their associated coefficient.

- **kwargs** – See the implemented sampling for additional keyword definitions.

Returns SampleSet

See also: sample(), sample_ising()

**Embedding**

Provides functions that map binary quadratic models and samples between a source graph and a target graph. For an introduction to minor-embedding, see Minor-Embedding.

**Generators**

Tools for finding embeddings.

**Generic**

minorminer is a heuristic tool for minor embedding: given a minor and target graph, it tries to find a mapping that embeds the minor into the target.

\[ \text{minorminer.find_embedding}(S, T, **params) \]

Heuristically attempt to find a minor-embedding of a graph representing an Ising/QUBO into a target graph.

**find_embedding**

Heuristically attempt to find a minor-embedding of a graph representing an Ising/QUBO into a target graph.

Args:

- S: an iterable of label pairs representing the edges in the source graph, or a NetworkX Graph
- T: an iterable of label pairs representing the edges in the target graph, or a NetworkX Graph
- **params** (optional): see below
Returns:

When `return_overlap = False` (the default), returns a `dict` that maps labels `in S` to lists of labels `in T`. If the heuristic fails to find an embedding, an empty dictionary is returned.

When `return_overlap = True`, returns a `tuple` consisting of a `dict` that maps labels `in S` to lists of labels `in T` and a `bool` indicating whether or not a valid embedding was found.

When interrupted by Ctrl-C, returns the best embedding found so far.

Note that failure to return an embedding does not prove that no embedding exists.

Optional parameters:

- `max_no_improvement`: Maximum number of failed iterations to improve the current solution, where each iteration attempts to find an embedding for each variable of `S` such that it is adjacent to all its neighbours. Integer >= 0 (default = 10)

- `random_seed`: Seed for the random number generator that `find_embedding` uses. Integer >=0 (default is randomly set)

- `timeout`: Algorithm gives up after timeout seconds. Number >= 0 (default is approximately 1000 seconds, stored as a double)

- `max_beta`: Qubits are assigned weight according to a formula `beta^n` where `n` is the number of chains containing that qubit. This value should never be less than or equal to 1. (default is effectively infinite, stored as a double)

- `tries`: Number of restart attempts before the algorithm stops. On D-WAVE 2000Q, a typical restart takes between 1 and 60 seconds. Integer >= 0 (default = 10)

- `inner_rounds`: The algorithm takes at most this many iterations between restart attempts; restart attempts are typically terminated due to `max_no_improvement`. Integer >= 0 (default = effectively infinite)

- `chainlength_patience`: Maximum number of failed iterations to improve chainlengths in the current solution, where each iteration attempts to find an embedding for each variable of `S` such that it is adjacent to all its neighbours. Integer >= 0 (default = 10)

- `max_fill`: Restricts the number of chains that can simultaneously incorporate the same qubit during the search. Integer >= 0, values above 63 are treated as 63 (default = effectively infinite)

- `threads`: Maximum number of threads to use. Note that the parallelization is only advantageous where the expected degree of variables is significantly greater than the number of threads. Integer >= 1 (default = 1)

- `return_overlap`: This function returns an embedding whether or not qubits are used by multiple variables. Set this value to 1 to capture both return values to determine whether or not the returned embedding is valid. Logical 0/1 integer (default = 0)
skip_initialization: Skip the initialization pass. Note that this only works if the chains passed in through initial_chains and fixed_chains are semi-valid. A semi-valid embedding is a collection of chains such that every adjacent pair of variables (u,v) has a coupler (p,q) in the hardware graph where p is in chain(u) and q is in chain(v). This can be used on a valid embedding to immediately skip to the chainlength improvement phase. Another good source of semi-valid embeddings is the output of this function with the return_overlap parameter enabled. Logical 0/1 integer (default = 0)

verbose: Level of output verbosity. Integer < 4 (default = 0).
   When set to 0, the output is quiet until the final result.
   When set to 1, output looks like this:

   initialized
   max qubit fill 3; num maxfull qubits=3
   embedding trial 1
   max qubit fill 2; num maxfull qubits=21
   embedding trial 2
   embedding trial 3
   embedding trial 4
   embedding trial 5
   embedding found.
   max chain length 4; num max chains=1
   reducing chain lengths
   max chain length 3; num max chains=5

   When set to 2, outputs the information for lower levels and also reports progress on minor statistics (when searching for an embedding, this is when the number of maxfull qubits decreases; when improving, this is when the number of max chains decreases)
   When set to 3, report before each before each pass. Look here when tweaking `tries`, `inner_rounds`, and `chainlength_patience`
   When set to 4, report additional debugging information. By default, this package is built without this functionality. In the c++ headers, this is controlled by the CPPDEBUG flag

Detailed explanation of the output information:
   max qubit fill: largest number of variables represented in a qubit
   num maxfull: the number of qubits that has max overfill
   max chain length: largest number of qubits representing a single variable
   num max chains: the number of variables that has max chain size

interactive: If `logging` is None or False, the verbose output will be printed to stdout/stderr as appropriate, and keyboard interrupts will stop the
   embedding process and the current state will be returned to the user. Otherwise, output will be directed to the logger `logging.getLogger(minorminer.__name__)` and keyboard interrupts will be propagated back to the user. Errors will use `logger.error()`, verbosity levels 1 through 3 will use `logger.info()` and `level`
   4 will use `logger.debug()`. bool, default False

initial_chains: Initial chains inserted into an embedding before
   fixed_chains are placed, which occurs before the initialization pass. These can be used to restart the algorithm in a similar state to a previous embedding; for example, to improve chainlength of a
valid embedding or to reduce overlap in a semi-valid embedding (see skip_initialization) previously returned by the algorithm. Missing or empty entries are ignored. A dictionary, where initial_chains[i] is a list of qubit labels.

fixed_chains: Fixed chains inserted into an embedding before the initialization pass. As the algorithm proceeds, these chains are not allowed to change, and the qubits used by these chains are not used by other chains. Missing or empty entries are ignored. A dictionary, where fixed_chains[i] is a list of qubit labels.

restrict_chains: Throughout the algorithm, we maintain the condition that chain[i] is a subset of restrict_chains[i] for each i, except those with missing or empty entries. A dictionary, where restrict_chains[i] is a list of qubit labels.

suspend_chains: This is a metafeature that is only implemented in the Python interface. suspend_chains[i] is an iterable of iterables; for example suspend_chains[i] = [blob_1, blob_2], with each blob_j an iterable of target node labels. this enforces the following:
for each suspended variable i,
for each blob_j in the suspension of i,
at least one qubit from blob_j will be contained in the chain for i
we accomplish this through the following problem transformation
for each iterable blob_j in suspend_chains[i],
* add an auxiliary node Zij to both source and target graphs
* set fixed_chains[Zij] = [Zij]
* add the edge (i,Zij) to the source graph
* add the edges (q,Zij) to the target graph for each q in blob_j

Chimera

Minor-embedding in Chimera-structured target graphs.

\[
\begin{align*}
\text{chimera.find_clique_embedding} & \quad \text{Find an embedding for a clique in a Chimera graph.} \\
\text{chimera.find_biclique_embedding} & \quad \text{Find an embedding for a biclique in a Chimera graph.} \\
\text{chimera.find_grid_embedding} & \quad \text{Find an embedding for a grid in a Chimera graph.}
\end{align*}
\]

\text{dwave.embedding.chimera.find_clique_embedding}

\text{find_clique_embedding} \quad (k, m[, n, t]) \quad \text{Find an embedding for a clique in a Chimera graph.}

Given the node labels or size of a clique (fully connected graph) and size or edges of the target Chimera graph, attempts to find an embedding.

Parameters

- **k (int/iterable)** – Clique to embed. If k is an integer, generates an embedding for a
clique of size k labelled [0,k-1]. If k is an iterable of nodes, generates an embedding for a clique of size len(k) labelled for the given nodes.

- **m (int)** – Number of rows in the Chimera lattice.
- **n (int, optional, default=m)** – Number of columns in the Chimera lattice.
- **t (int, optional, default 4)** – Size of the shore within each Chimera tile.
- **target_edges (iterable [edge])** – A list of edges in the target Chimera graph. Nodes are labelled as returned by chimera_graph().

**Returns** An embedding mapping a clique to the Chimera lattice.

**Return type** dict

**Examples**

The first example finds an embedding for a $K_4$ complete graph in a single Chimera unit cell. The second for an alphanumerically labeled $K_3$ graph in 4 unit cells.

```python
>>> from dwave.embedding.chimera import find_clique_embedding
... >>> embedding = find_clique_embedding(4, 1, 1)  
... # doctest: +SKIP  
{0: [4, 0], 1: [5, 1], 2: [6, 2], 3: [7, 3]}

>>> from dwave.embedding.chimera import find_clique_embedding
... >>> embedding = find_clique_embedding(['a', 'b', 'c'], m=2, n=2, t=4)  
... # doctest: +SKIP  
{'a': [20, 16], 'b': [21, 17], 'c': [22, 18]}
```

dwave.embedding.chimera.find_biclique_embedding

**find_biclique_embedding** ($a, b, m, n=None, t=None, target_edges=None$)

Find an embedding for a biclique in a Chimera graph.

Given a biclique (a bipartite graph where every vertex in a set in connected to all vertices in the other set) and a target Chimera graph size or edges, attempts to find an embedding.

**Parameters**

- **a (int/iterable)** – Left shore of the biclique to embed. If a is an integer, generates an embedding for a biclique with the left shore of size $a$ labelled [0,a-1]. If a is an iterable of nodes, generates an embedding for a biclique with the left shore of size len(a) labelled for the given nodes.

- **b (int/iterable)** – Right shore of the biclique to embed. If b is an integer, generates an embedding for a biclique with the right shore of size $b$ labelled [0,b-1]. If b is an iterable of nodes, generates an embedding for a biclique with the right shore of size len(b) labelled for the given nodes.

- **m (int)** – Number of rows in the Chimera lattice.

- **n (int, optional, default=m)** – Number of columns in the Chimera lattice.

- **t (int, optional, default 4)** – Size of the shore within each Chimera tile.
• **target_edges** *(iterable[edge]*) – A list of edges in the target Chimera graph. Nodes are labelled as returned by `chimera_graph()`.

**Returns**

A 2-tuple containing:

- dict: An embedding mapping the left shore of the biclique to the Chimera lattice.
- dict: An embedding mapping the right shore of the biclique to the Chimera lattice.

**Return type** tuple

**Examples**

This example finds an embedding for an alphanumerically labeled biclique in a single Chimera unit cell.

```python
>>> from dwave.embedding.chimera import find_biclique_embedding
... >>> left, right = find_biclique_embedding(['a', 'b', 'c'], ['d', 'e'], 1, 1)
>>> print(left, right)
# doctest: +SKIP
{'a': [4], 'b': [5], 'c': [6]} {'d': [0], 'e': [1]}
```

dwave.embedding.chimera.find_grid_embedding

**find_grid_embedding** *(dim, m, n=None, t=4)*

Find an embedding for a grid in a Chimera graph.

Given grid dimensions and a target Chimera graph size, attempts to find an embedding.

**Parameters**

- **dim** *(iterable[int]*) – Sizes of each grid dimension. Length can be between 1 and 3.
- **m** *(int)* – Number of rows in the Chimera lattice.
- **n** *(int, optional, default=m)* – Number of columns in the Chimera lattice.
- **t** *(int, optional, default 4)* – Size of the shore within each Chimera tile.

**Returns** An embedding mapping a grid to the Chimera lattice.

**Return type** dict

**Examples**

This example finds an embedding for a 2x3 grid in a 12x12 lattice of Chimera unit cells.

```python
>>> from dwave.embedding.chimera import find_grid_embedding
... >>> embedding = find_grid_embedding([2, 3], m=12, n=12, t=4)
>>> embedding
# doctest: +SKIP
{(0, 0): [0, 4],
 (0, 1): [8, 12],
 (0, 2): [16, 20],
 (1, 0): [96, 100],
 (1, 1): [104, 108],
 (1, 2): [112, 116]}
```
Pegasus

Minor-embedding in Pegasus-structured target graphs.

```python
_pegasus.find_clique_embedding(k[, m, ...])  # Find an embedding for a clique in a Pegasus graph.
```

**Pegasus**

Minor-embedding in *Pegasus*-structured target graphs.

```python
dwave.embedding.pegasus.find_clique_embedding
```

**find_clique_embedding** *(k, m=None, target_graph=None)*

Find an embedding for a clique in a Pegasus graph.

Given a clique (fully connected graph) and target Pegasus graph, attempts to find an embedding by transforming the Pegasus graph into a $K_{2,2}$ Chimera graph and then applying a Chimera clique-finding algorithm. Results are converted back to Pegasus coordinates.

**Parameters**

- **k** *(int/iterable/networkx.Graph)* – A complete graph to embed, formatted as a number of nodes, node labels, or a NetworkX graph.
- **m** *(int)* – Number of tiles in a row of a square Pegasus graph. Required to generate an $m$-by-$m$ Pegasus graph when `target_graph` is None.
- **target_graph** *(networkx.Graph)* – A Pegasus graph. Required when `m` is None.

**Returns**

An embedding as a dict, where keys represent the clique’s nodes and values, formatted as lists, represent chains of pegasus coordinates.

**Return type** **dict**

**Examples**

This example finds an embedding for a $K_3$ complete graph in a 2-by-2 Pegasus graph.

```python
>>> from dwave.embedding.pegasus import find_clique_embedding
... >>> print(find_clique_embedding(3, 2))  # doctest: +SKIP
{0: [10, 34], 1: [35, 11], 2: [32, 12]}
```

**Utilities**

```python
dwave.embedding.embed_bqm
```

**embed_bqm** *(source_bqm, embedding, target_adjacency, chain_strength=1.0, smear_vartype=None)*

Embed a binary quadratic model onto a target graph.

**Parameters**

```python
dwave.embedding.embed_ising
```

**embed_ising** *(source_h, source_J, embedding, target_adjacency)*

Embed an Ising problem onto a target graph.

**embed_qubo** *(source_Q, embedding, target_adjacency)*

Embed a QUBO onto a target graph.

**unembed_sampleset** *(target_sampleset, target_target, ...) Unembed a samples set.

**Utilities**

```python
dwave.embedding.embed_bqm
```

**embed_bqm** *(source_bqm, embedding, target_adjacency, chain_strength=1.0, smear_vartype=None)*

Embed a binary quadratic model onto a target graph.

**Parameters**
• `source_bqm` (*BinaryQuadraticModel*) – Binary quadratic model to embed.

• `embedding` (*dict*) – Mapping from source graph to target graph as a dict of form `{s: {t, ...}, ...}`, where `s` is a source-model variable and `t` is a target-model variable.

• `target_adjacency` (*dict/*networkx.Graph*) – Adjacency of the target graph as a dict of form `{t: Nt, ...}`, where `t` is a variable in the target graph and `Nt` is its set of neighbours.

• `chain_strength` (*float*, optional) – Magnitude of the quadratic bias (in SPIN-space) applied between variables to create chains, with the energy penalty of chain breaks set to `2 * chain_strength`.

• `smear_vartype` (*Vartype*, optional, default=None) – Determines whether the linear bias of embedded variables is smeared (the specified value is evenly divided as biases of a chain in the target graph) in SPIN or BINARY space. Defaults to the `Vartype` of `source_bqm`.

**Returns**  Target binary quadratic model.

**Return type**  *BinaryQuadraticModel*

**Examples**

This example embeds a triangular binary quadratic model representing a $K_3$ clique into a square target graph by mapping variable `c` in the source to nodes 2 and 3 in the target.

```python
>>> import networkx as nx
... >>> target = nx.cycle_graph(4)
>>> # Binary quadratic model for a triangular source graph
>>> h = {'a': 0, 'b': 0, 'c': 0}
>>> J = {('a', 'b'): 1, ('b', 'c'): 1, ('a', 'c'): 1}
>>> bqm = dimod.BinaryQuadraticModel.from_ising(h, J)
>>> # Variable c is a chain
>>> embedding = {'a': {0}, 'b': {1}, 'c': {2, 3}}
>>> # Embed and show the chain strength
>>> target_bqm = dwave.embedding.embed_bqm(bqm, embedding, target)
>>> target_bqm.quadratic[(2, 3)]
-1.0
>>> print(target_bqm.quadratic)  # doctest: +SKIP
{(0, 1): 1.0, (0, 3): 1.0, (1, 2): 1.0, (2, 3): -1.0}
```

**See also:**

`embed_ising()`, `embed_qubo()`

dwave.embedding.embed_ising

**embed_ising** (*source_h*, *source_J*, *embedding*, *target_adjacency*, *chain_strength=1.0*)

Embed an Ising problem onto a target graph.

**Parameters**

• `source_h` (*dict*[variable, bias]/list[bias]) – Linear biases of the Ising problem. If a list, the list’s indices are used as variable labels.

• `source_J` (*dict*[variable, variable, bias]) – Quadratic biases of the Ising problem.
• **embedding (dict)** – Mapping from source graph to target graph as a dict of form `{s: {t, ...}, ...}`, where s is a source-model variable and t is a target-model variable.

• **target_adjacency (dict/networkx.Graph)** – Adjacency of the target graph as a dict of form `{t: Nt, ...}`, where t is a target-graph variable and Nt is its set of neighbours.

• **chain_strength (float, optional)** – Magnitude of the quadratic bias (in SPIN-space) applied between variables to form a chain, with the energy penalty of chain breaks set to `2 * chain_strength`.

Returns

A 2-tuple:

- dict[variable, bias]: Linear biases of the target Ising problem.
- dict[(variable, variable), bias]: Quadratic biases of the target Ising problem.

Return type: tuple

**Examples**

This example embeds a triangular Ising problem representing a $K_3$ clique into a square target graph by mapping variable $c$ in the source to nodes 2 and 3 in the target.

```python
>>> import networkx as nx
... # Ising problem biases
>>> h = {'a': 0, 'b': 0, 'c': 0}
>>> J = {('a', 'b') : 1, ('b', 'c') : 1, ('a', 'c') : 1}
>>> # Variable c is a chain
>>> embedding = { 'a': {0}, 'b': {1}, 'c': {2, 3} }
>>> # Embed and show the resulting biases
>>> th, tJ = dwave.embedding.embed_ising(h, J, embedding, target)
>>> th # doctest: +SKIP
{0: 0.0, 1: 0.0, 2: 0.0, 3: 0.0}
>>> tJ # doctest: +SKIP
{(0, 1): 1.0, (0, 3): 1.0, (1, 2): 1.0, (2, 3): -1.0}
```

See also:

embed_bqm(), embed_qubo()

dwave.embedding.embed_qubo

**embed_qubo (source_Q, embedding, target_adjacency, chain_strength=1.0)**

Embed a QUBO onto a target graph.

Parameters

- **source_Q (dict[(variable, variable), bias])** – Coefficients of a quadratic unconstrained binary optimization (QUBO) model.

- **embedding (dict)** – Mapping from source graph to target graph as a dict of form `{s: {t, ...}, ...}`, where s is a source-model variable and t is a target-model variable.

- **target_adjacency (dict/networkx.Graph)** – Adjacency of the target graph as a dict of form `{t: Nt, ...}`, where t is a target-graph variable and Nt is its set of neighbours.
• **chain_strength** (*float*, *optional*) – Magnitude of the quadratic bias (in SPIN-space) applied between variables to form a chain, with the energy penalty of chain breaks set to $2 \times \text{chain_strength}$.

**Returns** Quadratic biases of the target QUBO.

**Return type** `dict[(variable, variable), bias]`

### Examples

This example embeds a triangular QUBO representing a $K_3$ clique into a square target graph by mapping variable $c$ in the source to nodes 2 and 3 in the target.

```python
>>> import networkx as nx
...
>>> target = nx.cycle_graph(4)
>>> # QUBO
>>> Q = {('a', 'b'): 1, ('b', 'c'): 1, ('a', 'c'): 1}
>>> # Variable c is a chain
>>> embedding = {'a': {0}, 'b': {1}, 'c': {2, 3}}
>>> # Embed and show the resulting biases
>>> tQ = dwave.embedding.embed_qubo(Q, embedding, target)
>>> tQ  # doctest: +SKIP
{(0, 1): 1.0,
 (0, 3): 1.0,
 (1, 2): 1.0,
 (2, 3): -4.0,
 (0, 0): 0.0,
 (1, 1): 0.0,
 (2, 2): 2.0,
 (3, 3): 2.0}
```

**See also:**

`embed_bqm()`, `embed_ising()`
• **chain_break_fraction** *(bool, optional, default=False)* – Add a `chain_break_fraction` field to the unembedded `dimod.SampleSet` with the fraction of chains broken before unembedding.

• **return_embedding** *(bool, optional, default=False)* – If True, the embedding is added to `dimod.SampleSet.info` of the returned sample set. Note that if an `embedding` key already exists in the sample set then it is overwritten.

**Returns** Sample set in the source BQM.

**Return type** SampleSet

**Examples**

This example unembeds from a square target graph samples of a triangular source BQM.

```python
>>> # Triangular binary quadratic model and an embedding
>>> J = {('a', 'b'): -1, ('b', 'c'): -1, ('a', 'c'): -1}
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, J)
>>> embedding = {'a': [0, 1], 'b': [2], 'c': [3]}
>>> # Samples from the embedded binary quadratic model
>>> samples = [{0: -1, 1: -1, 2: -1, 3: -1},  # [0, 1] is unbroken
...             {0: -1, 1: +1, 2: +1, 3: +1}]  # [0, 1] is broken
>>> energies = [-3, 1]
>>> embedded = dimod.SampleSet.from_samples(samples, dimod.SPIN, energies)
>>> # Unembed
>>> samples = dwave.embedding.unembed_sampleset(embedded, embedding, bqm)
```

Diagnostics

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chain_break_frequency</code></td>
<td>Determine the frequency of chain breaks in the given samples.</td>
</tr>
<tr>
<td><code>diagnose_embedding</code></td>
<td>Diagnose a minor embedding.</td>
</tr>
<tr>
<td><code>is_valid_embedding</code></td>
<td>A simple (bool) diagnostic for minor embeddings.</td>
</tr>
<tr>
<td><code>verify_embedding</code></td>
<td>A simple (exception-raising) diagnostic for minor embeddings.</td>
</tr>
</tbody>
</table>

**6.8. dwave-system**

**chain_break_frequency** *(samples_like, embedding)*

Determine the frequency of chain breaks in the given samples.

**Parameters**

- **samples_like** *(samples_like/dimod.SampleSet)* – A collection of raw samples. `samples_like` is an extension of NumPy’s array_like. See `dimod.as_samples()`.

- **embedding** *(dict)* – Mapping from source graph to target graph as a dict of form `{s: {t, ...}, ...}`, where `s` is a source-model variable and `t` is a target-model variable.

**Returns** Frequency of chain breaks as a dict in the form `{s: f, ...}`, where `s` is a variable in the source graph and float `f` the fraction of broken chains.
Return type  dict

Examples

This example embeds a single source node, ‘a’, as a chain of two target nodes (0, 1) and uses `chain_break_frequency()` to show that out of two synthetic samples, one ([−1, +1]) represents a broken chain.

```python
>>> import numpy as np
... >>> samples = np.array([[-1, +1], [+1, +1]])
>>> embedding = {'a': {0, 1}}
>>> print(dwave.embedding.chain_break_frequency(samples, embedding)['a'])
0.5
```

dwave.embedding.diagnose_embedding

diagnose_embedding(emb, source, target)

Diagnose a minor embedding.

Produces a generator that lists all issues with the embedding. User-friendly variants of this function are `is_valid_embedding()`, which returns a bool, and `verify_embedding()`, which raises the first observed error.

Parameters

- `emb` (dict) – A mapping of source nodes to arrays of target nodes as a dict of form {s: [t, ...], ...}, where s is a source-graph variable and t is a target-graph variable.
- `source` (list/networkx.Graph) – Graph to be embedded as a NetworkX graph or a list of edges.
- `target` (list/networkx.Graph) – Graph being embedded into as a NetworkX graph or a list of edges.

Yields

Errors yielded in the form `ExceptionClass, arg1, arg2,...`, where the arguments following the class are used to construct the exception object, which are subclasses of `EmbeddingError`.

- `MissingChainError`, snode: a source node label that does not occur as a key of `emb`, or for which emb[snode] is empty.
- `ChainOverlapError`, tnode, snode0, snode1: a target node which occurs in both emb[snode0] and emb[snode1].
- `DisconnectedChainError`, snode: a source node label whose chain is not a connected subgraph of `target`.
- `InvalidNodeError`, tnode, snode: a source node label and putative target node label that is not a node of `target`.
- `MissingEdgeError`, snode0, snode1: a pair of source node labels defining an edge that is not present between their chains.

Examples

This example diagnoses an invalid embedding from a triangular source graph to a square target graph. A valid embedding, such as `emb = {0: [1], 1: [0], 2: [2, 3]}`, yields no errors.
```python
>>> from dwave.embedding import diagnose_embedding
>>> import networkx as nx
>>> source = nx.complete_graph(3)
>>> target = nx.cycle_graph(4)
>>> embedding = {0: [2], 1: [1, 'a'], 2: [2, 3]}
>>> diagnosis = diagnose_embedding(embedding, source, target)
>>> for problem in diagnosis: # doctest: +SKIP
...   print(problem)
(<class 'dwave.embedding.exceptions.InvalidNodeError'>, 1, 'a')
(<class 'dwave.embedding.exceptions.ChainOverlapError'>, 2, 2, 0)
```

dwave.embedding.is_valid_embedding

`is_valid_embedding(emb, source, target)`

A simple (bool) diagnostic for minor embeddings.

See `diagnose_embedding()` for a more detailed diagnostic and more information.

Parameters

- `emb (dict)` – A mapping of source nodes to arrays of target nodes as a dict of form `{s: [t, ...], ...}`, where `s` is a source-graph variable and `t` is a target-graph variable.
- `source (graph or edgelist)` – Graph to be embedded.
- `target (graph or edgelist)` – Graph being embedded into.

Returns  True if `emb` is valid.

Return type  bool

dwave.embedding.verify_embedding

`verify_embedding(emb, source, target, ignore_errors=())`

A simple (exception-raising) diagnostic for minor embeddings.

See `diagnose_embedding()` for a more detailed diagnostic and more information.

Parameters

- `emb (dict)` – A mapping of source nodes to arrays of target nodes as a dict of form `{s: [t, ...], ...}`, where `s` is a source-graph variable and `t` is a target-graph variable.
- `source (graph or edgelist)` – Graph to be embedded
- `target (graph or edgelist)` – Graph being embedded into

Raises `EmbeddingError` – A catch-all class for the following errors:

- MissingChainError: A key is missing from `emb` or the associated chain is empty.
- ChainOverlapError: Two chains contain the same target node.
- DisconnectedChainError: A chain is disconnected.
- InvalidNodeError: A chain contains a node label not found in `target`.
- MissingEdgeError: A source edge is not represented by any target edges.

Returns  True if no exception is raised.

Return type  bool
Chain-Break Resolution

Handling samples with broken chains when unembedding.

Generators

| chain_breaks.discard(samples, chains) | Discard broken chains. |
| chain_breaks.majority_vote(samples, chains) | Unembed samples using the most common value for broken chains. |
| chain_breaks.weighted_random(samples, chains) | Unembed samples using weighed random choice for broken chains. |

dwave.embedding.chain_breaks.discard

discard(samples, chains)
Discard broken chains.

Parameters

- **samples (samples_like)** – A collection of samples. samples_like is an extension of NumPy’s array_like. See dimod.as_samples().
- **chains (list[array_like])** – List of chains, where each chain is an array_like collection of the variables in the same order as their representation in the given samples.

Returns

A 2-tuple containing:

- numpy.ndarray: Unembedded samples as an array of dtype ‘int8’. Broken chains are discarded.
- numpy.ndarray: Indicies of rows with unbroken chains.

Return type  tuple

Examples

This example unembeds two samples that chains nodes 0 and 1 to represent a single source node. The first sample has an unbroken chain, the second a broken chain.

```python
>>> import dimod
>>> import numpy as np
... >>> chains = [(0, 1), (2,)]
>>> samples = np.array([[1, 1, 0], [1, 0, 0]], dtype=np.int8)
>>> unembedded, idx = dwave.embedding.discard(samples, chains)
>>> unembedded
array([[1, 0]], dtype=int8)
>>> print(idx)
[0]
```
**Chain Breaks**

### majority_vote(samples, chains)

Unembed samples using the most common value for broken chains.

**Parameters**

- **samples (samples_like)**: A collection of samples. *samples_like* is an extension of NumPy’s `array_like`. See `dimod.as_samples()`.

- **chains (list[array_like])**: List of chains, where each chain is an `array_like` collection of the variables in the same order as their representation in the given samples.

**Returns**

A 2-tuple containing:

- `numpy.ndarray`: Unembedded samples as an nS-by-nC array of dtype `int8`, where nC is the number of chains and nS the number of samples. Broken chains are resolved by setting the sample value to that of most the chain’s elements or, for chains without a majority, an arbitrary value.

- `numpy.ndarray`: Indices of the samples. Equivalent to `np.arange(nS)` because all samples are kept and none added.

**Return type**: tuple

### Examples

This example unembeds samples from a target graph that chains nodes 0 and 1 to represent one source node and nodes 2, 3, and 4 to represent another. Both samples have one broken chain, with different majority values.

```python
>>> import dimod
>>> import numpy as np
...
>>> chains = [[0, 1], [2, 3, 4]]
>>> samples = np.array([[1, 1, 0, 0, 1], [1, 1, 1, 0, 1]], dtype=np.int8)
>>> unembedded, idx = dimod.majority_vote(samples, chains)
>>> print(unembedded)

[[1 0]
 [1 1]]

>>> print(idx)
[0 1]
```

### weighted_random(samples, chains)

Unembed samples using weighed random choice for broken chains.

**Parameters**

- **samples (samples_like)**: A collection of samples. *samples_like* is an extension of NumPy’s `array_like`. See `dimod.as_samples()`.

- **chains (list[array_like])**: List of chains, where each chain is an `array_like` collection of the variables in the same order as their representation in the given samples.
Returns

A 2-tuple containing:

- **numpy.ndarray**: Unembedded samples as an nS-by-nC array of dtype `int8`, where nC is the number of chains and nS the number of samples. Broken chains are resolved by setting the sample value to a random value weighted by frequency of the value in the chain.

- **numpy.ndarray**: Indices of the samples. Equivalent to `np.arange(nS)` because all samples are kept and no samples are added.

**Return type** tuple

**Examples**

This example unembeds samples from a target graph that chains nodes 0 and 1 to represent one source node and nodes 2, 3, and 4 to represent another. The sample has broken chains for both source nodes.

```python
>>> import dimod
>>> import numpy as np
... 
>>> chains = [(0, 1), (2, 3, 4)]
>>> samples = np.array([[1, 0, 1, 0, 1]], dtype=np.int8)
>>> unembedded, idx = dwave.embedding.weighted_random(samples, chains)  # doctest: +SKIP

unembedded
# doctest: +SKIP
array([[1, 1]], dtype=int8)

idx
# doctest: +SKIP
array([0, 1])
```

**Callable Objects**

- `chain_breaks.MinimizeEnergy(bqm, embedding)`
  Unembed samples by minimizing local energy for broken chains.

- `dwave.embedding.chain_breaks.MinimizeEnergy`

**class MinimizeEnergy (bqm, embedding)**

Unembed samples by minimizing local energy for broken chains.

**Parameters**

- **bqm** (`BinaryQuadraticModel`) – Binary quadratic model associated with the source graph.
- **embedding** (`dict`) – Mapping from source graph to target graph as a dict of form `{s: [t, ...], ...}`, where s is a source-model variable and t is a target-model variable.

**Examples**

This example embeds from a triangular graph to a square graph, chaining target-nodes 2 and 3 to represent source-node c, and unembeds minimizing the energy for the samples. The first two sample have unbroken chains, the second two have broken chains.
```python
>>> import dimod
>>> import numpy as np
...
>>> h = {'a': 0, 'b': 0, 'c': 0}
>>> J = {('a', 'b'): 1, ('b', 'c'): 1, ('a', 'c'): 1}
>>> bqm = dimod.BinaryQuadraticModel.from_ising(h, J)
>>> embedding = {'a': [0], 'b': [1], 'c': [2, 3]}
>>> cbm = dwave.embedding.MinimizeEnergy(bqm, embedding)
>>> samples = np.array([[+1, -1, +1, +1],
... [-1, -1, -1, -1],
... [-1, -1, +1, -1],
... [+1, +1, -1, +1]], dtype=np.int8)
>>> chains = [embedding['a'], embedding['b'], embedding['c']]  
>>> unembedded, idx = cbm(samples, chains)
>>> unembedded
array([[ 1, -1, 1],
       [-1, -1, -1],
       [-1, -1, 1],
       [ 1, 1, -1]], dtype=int8)
>>> idx
array([0, 1, 2, 3])
```

__call__ (samples, chains)

**Parameters**

- **samples** *(samples_like)* – A collection of samples. *samples_like* is an extension of NumPy’s `array_like`. See `dimod.as_samples()`.
- **chains** *(list[array_like])* – List of chains, where each chain is an `array_like` collection of the variables in the same order as their representation in the given samples.

**Returns**

A 2-tuple containing:

- `numpy.ndarray`: Unembedded samples as an `nS`-by-`nC` array of dtype ‘int8’, where `nC` is the number of chains and `nS` the number of samples. Broken chains are resolved by greedy energy descent.
- `numpy.ndarray`: Indices of the samples. Equivalent to `np.arange(nS)` because all samples are kept and none added.

**Return type**  `tuple`

**Exceptions**

- `exceptions.EmbeddingError`  
  Base class for all embedding exceptions.
- `exceptions.MissingChainError`(snode)  
  Raised if a node in the source graph has no associated chain.
- `exceptions.ChainOverlapError`(node, snode0,...)  
  Raised if two source nodes have an overlapping chain.
- `exceptions.DisconnectedChainError`(snode)  
  Raised if a chain is not connected in the target graph.
- `exceptions.InvalidNodeError`(snode, tnode)  
  Raised if a chain contains a node not in the target graph.
- `exceptions.MissingEdgeError`(snode0, snodel)  
  Raised when two source nodes sharing an edge to not have a corresponding edge between their chains.
**dwave.embedding.exceptions.EmbeddingError**

*exception EmbeddingError*

Base class for all embedding exceptions.

**dwave.embedding.exceptions.MissingChainError**

*exception MissingChainError* *(snode)*

Raised if a node in the source graph has no associated chain.

Parameters

- **snode** – The source node with no associated chain.

**dwave.embedding.exceptions.ChainOverlapError**

*exception ChainOverlapError* *(tnode, snode0, snode1)*

Raised if two source nodes have an overlapping chain.

Parameters

- **tnode** – Location where the chains overlap.
- **snode0** – First source node with overlapping chain.
- **snode1** – Second source node with overlapping chain.

**dwave.embedding.exceptions.DisconnectedChainError**

*exception DisconnectedChainError* *(snode)*

Raised if a chain is not connected in the target graph.

Parameters

- **snode** – The source node associated with the broken chain.

**dwave.embedding.exceptions.InvalidNodeError**

*exception InvalidNodeError* *(snode, tnode)*

Raised if a chain contains a node not in the target graph.

Parameters

- **snode** – The source node associated with the chain.
- **tnode** – The node in the chain not in the target graph.

**dwave.embedding.exceptions.MissingEdgeError**

*exception MissingEdgeError* *(snode0, snode1)*

Raised when two source nodes sharing an edge to not have a corresponding edge between their chains.

Parameters

- **snode0** – First source node.
- **snode1** – Second source node.
Utilities

Utility functions.

`common_working_graph(graph0, graph1)` Creates a graph using the common nodes and edges of two given graphs.

**Parameters**

- `graph0` – A NetworkX graph or a dictionary of dictionaries adjacency representation.
- `graph1` – A NetworkX graph or a dictionary of dictionaries adjacency representation.

**Returns** A graph with the nodes and edges common to both input graphs.

**Return type** Graph

**Examples**

This example creates a graph that represents a quarter (4 by 4 Chimera tiles) of a particular D-Wave system’s working graph.

```python
>>> import dwave_networkx as dnx
>>> from dwave.system import DWaveSampler, common_working_graph
... >>> sampler = DWaveSampler(solver={‘qpu’: True})
>>> C4 = dnx.chimera_graph(4)  # a 4x4 lattice of Chimera tiles
>>> C4_working_graph = common_working_graph(C4, sampler.adjacency)
```

## 6.9 dwave-tabu

An implementation of the MST2 multistart tabu search algorithm for quadratic unconstrained binary optimization (QUBO) problems with a dimod Python wrapper.

### 6.9.1 Introduction

_Samplers_ are processes that sample from low energy states of a problem’s objective function. A binary quadratic model (BQM) sampler samples from low energy states in models such as those defined by an Ising equation or a Quadratic Unconstrained Binary Optimization (QUBO) problem and returns an iterable of samples, in order of increasing energy. A dimod _sampler_ provides ‘sample_qubo’ and ‘sample_ising’ methods as well as the generic BQM sampler method.

The TabuSampler sampler implements the MST2 multistart tabu search algorithm for quadratic unconstrained binary optimization (QUBO) problems with a dimod Python wrapper.
For a description of the tabu search algorithm, see tabu search.

Example

This example solves a two-variable Ising model.

```python
>>> from tabu import TabuSampler
>>> response = TabuSampler().sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b'): -1})
```

6.9.2 Reference Documentation

Release 2.5.0

Date Aug 12, 2020

D-Wave Tabu Sampler

A dimod sampler that uses the MST2 multistart tabu search algorithm.

```python
class TabuSampler
    A tabu-search sampler.
```

Examples

This example solves a two-variable Ising model.

```python
>>> from tabu import TabuSampler
>>> samples = TabuSampler().sample_ising({'a': -0.5, 'b': 1.0}, {'ab': -1})
>>> list(samples.data()) # doctest: +SKIP
[Sample(sample={'a': -1, 'b': -1}, energy=-1.5, num_occurrences=1)]
>>> samples.first.energy
-1.5
```

Parameters

- `bqm (BinaryQuadraticModel)` – The binary quadratic model (BQM) to be sampled.
- `initial_states` (SampleSet, optional, default=None) – One or more samples, each defining an initial state for all the problem variables. Initial states are given one per read, but if fewer than `num_reads` initial states are defined, additional values are generated as specified by `initial_states_generator`.
- `initial_states_generator` (str, 'none'/'tile'/random', optional, default='random') – Defines the expansion of `initial_states` if fewer than `num_reads` are specified:
  - "none": If the number of initial states specified is smaller than `num_reads`, raises ValueError.
  - "tile": Reuses the specified initial states if fewer than `num_reads` or truncates if greater.
  - "random": Generates random values.
- "random": Expands the specified initial states with randomly generated states if fewer than num_reads or truncates if greater.

- `num_reads (int, optional, default=len(initial_states) or 1)`
  Number of reads. Each read is generated by one run of the tabu algorithm. If num_reads is not explicitly given, it is selected to match the number of initial states given. If initial states are not provided, only one read is performed.

- `tenure (int, optional)`
  Tabu tenure, which is the length of the tabu list, or number of recently explored solutions kept in memory. Default is a quarter of the number of problem variables up to a maximum value of 20.

- `timeout (int, optional)`
  Total running time in milliseconds.

- `scale_factor (number, optional)`
  Scaling factor for linear and quadratic biases in the BQM. Internally, the BQM is converted to a QUBO matrix, and elements are stored as long ints using internal_q = long int (q * scale_factor).

- `init_solution (SampleSet, optional)`
  Deprecated. Alias for initial_states.

Returns A dimod SampleSet object.

Return type SampleSet

Examples

This example samples a simple two-variable Ising model.

```python
>>> import dimod
>>> bqm = dimod.BQM.from_ising({}, {'ab': 1})

>>> import tabu
>>> sampler = tabu.TabuSampler()

>>> samples = sampler.sample(bqm)
>>> samples.record[0].energy
-1.0
```

6.10 minorminer

minorminer is a heuristic tool for minor embedding: given a minor and target graph, it tries to find a mapping that embeds the minor into the target.

The primary utility function, `find_embedding()`, is an implementation of the heuristic algorithm described in [1]. It accepts various optional parameters used to tune the algorithm’s execution or constrain the given problem.

This implementation performs on par with tuned, non-configurable implementations while providing users with hooks to easily use the code as a basic building block in research.


6.10.1 Introduction

Examples

This example minor embeds a triangular source K4 graph onto a square target graph.
from minorminer import find_embedding

# A triangle is a minor of a square.
triangle = [(0, 1), (1, 2), (2, 0)]
square = [(0, 1), (1, 2), (2, 3), (3, 0)]

# Find an assignment of sets of square variables to the triangle variables
embedding = find_embedding(triangle, square, random_seed=10)
print(len(embedding))  # 3, one set for each variable in the triangle
print(embedding)
# We don’t know which variables will be assigned where, here are a
# couple possible outputs:
# [[0, 1], [2], [3]]
# [[3], [1, 0], [2]]

Fig. 26: Embedding a $K_3$ source graph into a square target graph by chaining two target nodes to represent one source
node.

This minorminer execution of the example requires that source variable 0 always be assigned to target node 2.

embedding = find_embedding(triangle, square, fixed_chains={0: [2]})
print(embedding)
# [[2], [3, 0], [1]]
# [[2], [1], [0, 3]]
# And more, but all of them start with [2]

This minorminer execution of the example suggests that source variable 0 be assigned to target node 2 as a starting
point for finding an embedding.

embedding = find_embedding(triangle, square, initial_chains={0: [2]})
print(embedding)
# [[2], [0, 3], [1]]
# [[0], [3], [1, 2]]
# Output where source variable 0 has switched to a different target node is possible.

This example minor embeds a fully connected K6 graph into a 30-node random regular graph of degree 3.

import networkx as nx

clique = nx.complete_graph(6).edges()
target_graph = nx.random_regular_graph(d=3, n=30).edges()
embedding = find_embedding(clique, target_graph)
print(embedding)
# There are many possible outputs, and sometimes it might fail

(continues on next page)
Fig. 27: Embedding a $K_6$ source graph (upper left) into a 30-node random target graph of degree 3 (upper right) by chaining several target nodes to represent one source node (bottom). The graphic of the embedding clusters representing nodes in the source graph: the cluster of red nodes is a chain of target nodes that represent source node 0, the orange nodes represent source node 1, and so on.

6.10.2 Reference Documentation

Python Interface

```python
find_embedding(S, T, **params)
```

Heuristically attempt to find a minor-embedding of a graph representing an Ising/QUBO into a target graph.
Args:

S: an iterable of label pairs representing the edges in the source graph, or a NetworkX Graph

T: an iterable of label pairs representing the edges in the target graph, or a NetworkX Graph

**params (optional): see below

Returns:

When return_overlap = False (the default), returns a dict that maps labels in S to lists of labels in T.

If the heuristic fails to find an embedding, an empty dictionary is returned

When return_overlap = True, returns a tuple consisting of a dict that maps labels in S to lists of labels in T and a bool indicating whether or not a valid embedding was found

When interrupted by Ctrl-C, returns the best embedding found so far

Note that failure to return an embedding does not prove that no embedding exists

Optional parameters:

max_no_improvement: Maximum number of failed iterations to improve the current solution, where each iteration attempts to find an embedding for each variable of S such that it is adjacent to all its neighbours. Integer >= 0 (default = 10)

random_seed: Seed for the random number generator that find_embedding uses. Integer >=0 (default is randomly set)

timeout: Algorithm gives up after timeout seconds. Number >= 0 (default is approximately 1000 seconds, stored as a double)

max_beta: Qubits are assigned weight according to a formula (beta^n) where n is the number of chains containint that qubit. This value should never be less than or equal to 1. (default is effectively infinite, stored as a double)

tries: Number of restart attempts before the algorithm stops. On D-WAVE 2000Q, a typical restart takes between 1 and 60 seconds. Integer >= 0 (default = 10)

inner_rounds: the algorithm takes at most this many iterations between restart attempts; restart attempts are typically terminated due to max_no_improvement. Integer >= 0 (default = effectively infinite)

chainlength_patience: Maximum number of failed iterations to improve chainlengths in the current solution, where each iteration attempts to find an embedding for each variable of S such that it is adjacent to all its neighbours. Integer >= 0 (default = 10)

max_fill: Restricts the number of chains that can simultaneously incorporate the same qubit during the search. Integer >= 0, values above 63 are treated as 63 (default = effectively infinite)
threads: Maximum number of threads to use. Note that the
parallelization is only advantageous where the expected degree of
variables is significantly greater than the number of threads.
Integer >= 1 (default = 1)

return_overlap: This function returns an embedding whether or not qubits
are used by multiple variables. Set this value to 1 to capture both
return values to determine whether or not the returned embedding is
valid. Logical 0/1 integer (default = 0)

skip_initialization: Skip the initialization pass. Note that this only
works if the chains passed in through initial_chains and
fixed_chains are semi-valid. A semi-valid embedding is a collection
of chains such that every adjacent pair of variables (u,v) has a
coupler (p,q) in the hardware graph where p is in chain(u) and q is
in chain(v). This can be used on a valid embedding to immediately
skip to the chainlength improvement phase. Another good source of
semi-valid embeddings is the output of this function with the
return_overlap parameter enabled. Logical 0/1 integer (default = 0)

verbose: Level of output verbosity. Integer < 4 (default = 0).
When set to 0, the output is quiet until the final result.
When set to 1, output looks like this:

initialized
max qubit fill 3; num maxfull qubits=3
embedding trial 1
max qubit fill 2; num maxfull qubits=21
embedding trial 2
embedding trial 3
embedding trial 4
embedding trial 5
embedding found.
max chain length 4; num max chains=1
reducing chain lengths
max chain length 3; num max chains=5

When set to 2, outputs the information for lower levels and also
reports progress on minor statistics (when searching for an
embedding, this is when the number of maxfull qubits decreases;
when improving, this is when the number of max chains decreases)
When set to 3, report before each before each pass. Look here when
tweaking ‘tries’, ‘inner_rounds’, and ‘chainlength_patience’
When set to 4, report additional debugging information. By default,
this package is built without this functionality. In the c++
headers, this is controlled by the CPPDEBUG flag

Detailed explanation of the output information:
max qubit fill: largest number of variables represented in a qubit
num maxfull: the number of qubits that has max overfill
max chain length: largest number of qubits representing a single variable
num max chains: the number of variables that has max chain size

interactive: If ‘logging’ is None or False, the verbose output will be printed
to stdout/stderr as appropriate, and keyboard interrupts will stop the
embedding
process and the current state will be returned to the user. Otherwise, output
(continues on next page)
will be directed to the logger `logging.getLogger(minorminer.__name__)` and
keyboard interrupts will be propagated back to the user. Errors will use
`logger.error()`, verbosity levels 1 through 3 will use `logger.info()` and,
level 4 will use `logger.debug()`.

bool, default False

initial_chains: Initial chains inserted into an embedding before
fixed_chains are placed, which occurs before the initialization
pass. These can be used to restart the algorithm in a similar state
to a previous embedding; for example, to improve chainlength of a
valid embedding or to reduce overlap in a semi-valid embedding (see
skip_initialization) previously returned by the algorithm. Missing
or empty entries are ignored. A dictionary, where initial_chains[i]
is a list of qubit labels.

fixed_chains: Fixed chains inserted into an embedding before the
initialization pass. As the algorithm proceeds, these chains are not
allowed to change, and the qubits used by these chains are not used by
other chains. Missing or empty entries are ignored. A dictionary, where
fixed_chains[i] is a list of qubit labels.

restrict_chains: Throughout the algorithm, we maintain the condition
that chain[i] is a subset of restrict_chains[i] for each i, except
those with missing or empty entries. A dictionary, where
restrict_chains[i] is a list of qubit labels.

suspend_chains: This is a metafeature that is only implemented in the Python
interface. suspend_chains[i] is an iterable of iterables; for example
suspend_chains[i] = [blob_1, blob_2],
with each blob_j an iterable of target node labels.
this enforces the following:
for each suspended variable i,
  for each blob_j in the suspension of i,
    at least one qubit from blob_j will be contained in the
    chain for i

we accomplish this through the following problem transformation
for each iterable blob_j in suspend_chains[i],
  * add an auxiliary node Zij to both source and target graphs
  * set fixed_chains[Zij] = [Zij]
  * add the edge (i,Zij) to the source graph
  * add the edges (q,Zij) to the target graph for each q in blob_j

C++ Library

6.11 penaltymodel

One approach to solve a constraint satisfaction problem (CSP) using an Ising model or a QUBO, is to map each
individual constraint in the CSP to a ‘small’ Ising model or QUBO. This mapping is called a penalty model.

Imagine that we want to map an AND clause to a QUBO. In other words, we want the solutions to the QUBO (the
solutions that minimize the energy) to be exactly the valid configurations of an AND gate. Let \( z = AND(x_1, x_2) \).

Before anything else, let’s import that package we will need.
import penaltymodel.core as pm
import dimod
import networkx as nx

Next, we need to determine the feasible configurations that we wish to target (by making the energy of these configuration in the binary quadratic low). Below is the truth table representing an AND clause.

Table 172: AND Gate

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The rows of the truth table are exactly the feasible configurations.

feasible_configurations = {(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1)}

We also need a target graph and to label the decision variables. We create a node in the graph for each variable in the problem, and we add an edge between each node, representing the interactions between the variables. In this case we allow an interaction between every variable, but more sparse interactions are possible. The labels of the nodes and the decision variables match.

```python
graph = nx.Graph()
graph.add_edges_from([('x1', 'x2'), ('x1', 'z'), ('x2', 'z')])
decision_variables = ['x1', 'x2', 'z']
```

We now have everything needed to build our Specification. We have binary variables so we select the appropriate variable type.

```python
spec = pm.Specification(graph, decision_variables, feasible_configurations, dimod.BINARY)
```

At this point, if we have any factories installed, we could use the factory interface to get an appropriate penalty model for our specification.

```python
p_model = pm.get_penalty_model(spec)
```

However, if we know the QUBO, we can build the penalty model ourselves. We observe that for the equation:

$$E(x_1, x_2, z) = x_1x_2 - 2(x_1 + x_2)z + 3z + 0$$

We get the following energies for each row in our truth table.
We can see that the energy is minimized on exactly the desired feasible configurations. So we encode this energy function as a QUBO. We make the offset 0.0 because there is no constant energy offset.

\[
\text{qubo} = \text{dimod.BinaryQuadraticModel}({'x1': 0., 'x2': 0., 'z': 3.},
\{(\text{x1}, \text{x2}): 1., (\text{x1}, \text{z}): 2., (\text{x2}, \text{z}): 2.},
0.0,
\text{dimod.BINARY})
\]

We know from the table that our ground energy is 0, but we can calculate it using the qubo to check that this is true for the feasible configuration (0, 1, 0).

\[
\text{ground_energy} = \text{qubo.energy}({'x1': 0, 'x2': 1, 'z': 0})
\]

The last value that we need is the classical gap. This is the difference in energy between the lowest infeasible state and the ground state.

With all of the pieces, we can now build the penalty model.

```
classical_gap = 1
p_model = pm.PenaltyModel.from_specification(spec, qubo, classical_gap, ground_energy)
```

### 6.11.1 Packages

**Release** 2.5.0  
**Date** Aug 12, 2020
penaltymodel

The core penalty model package. Contains the logic shared in the penalty model ecosystem.

Included Classes

PenaltyModel

class PenaltyModel (graph, decision_variables, feasible_configurations, vartype, model, classical_gap, ground_energy, ising_linear_ranges=None, ising_quadratic_ranges=None)

Container class for the components that make up a penalty model.

A penalty model is a small Ising problem or QUBO that has ground states that match the feasible configurations and excited states that have a classical energy greater than the ground energy by at least the classical gap.

PenaltyModel is a subclass of Specification.

Parameters

- graph (networkx.Graph/iterable[edge]) – Defines the structure of the desired binary quadratic model. Each node in the graph represents a variable and each edge defines an interaction between two variables. If given as an iterable of edges, the graph will be constructed by adding each edge to an (initially) empty graph.

- decision_variables (iterable) – The labels of the penalty model’s decision variables. Each variable label in decision_variables must correspond to a node in graph. Should be an ordered iterable of hashable labels.

- feasible_configurations (dict[tuple[int], number]/iterable[tuple[int]]) – The set of feasible configurations. Defines the allowed configurations of the decision variables allowed by the constraint. Each feasible configuration should be a tuple, each element of which must be of a value matching vartype. If given as a dict, the key is the feasible configuration and the value is the desired relative energy. If given as an iterable, it will be case to a dict where the relative energies are all 0.

- vartype (Vartype/str/set) – The variable type desired for the penalty model. Accepted input values: Vartype.SPIN, 'SPIN', {-1, 1} Vartype.BINARY, 'BINARY', (0, 1)

- model (dimod.BinaryQuadraticModel) – A binary quadratic model that has ground states that match the feasible_configurations.

- classical_gap (numeric) – The difference in classical energy between the ground state and the first excited state. Must be positive.

- ground_energy (numeric) – The minimum energy of all possible configurations.

- ising_linear_ranges (dict[node, [number, number]], optional, default=None) – When the penalty model is spin-valued, specifies the allowed range for each of the linear biases. If a dict, should be of the form {v: [min, max], ...} where v is a variable in the desired penalty model and (min, max) defines the acceptable range for the linear bias associated with v. If None, the default will be set to {v: [-1, 1], ...} for each v in graph. A partial assignment is allowed.

- ising_quadratic_ranges (dict[node, dict[node, [number, number]]], optional, default=None) – When the penalty model is spin-valued, specifies the allowed range for each of the quadratic biases. If a dict, should be of the form {v: {u: [min, max], ...}, ...} where u and v are variables in the desired penalty model.
model and u, v have an interaction - there is an edge between nodes u, v in graph. (min, max) the acceptable range for the quadratic bias associated with u, v. If None, the default will be set to {v: {u: [min, max], ...}, u: {v: [min, max], ...}, ...} for each edge u, v in graph. A partial assignment is allowed.

Examples

The penalty model can be created from its component parts:

```python
>>> import networkx as nx
>>> import dimod

>>> graph = nx.path_graph(3)

>>> decision_variables = (0, 2)  # the ends of the path
>>> feasible_configurations = {(-1, -1), (1, 1)}  # we want the ends of the path to agree

>>> model = dimod.BinaryQuadraticModel({0: 0, 1: 0, 2: 0}, {(0, 1): -1, (1, 2): -1}, 0.0, dimod.SPIN)

>>> classical_gap = 2.0
>>> ground_energy = -2.0

>>> widget = pm.PenaltyModel(graph, decision_variables, feasible_configurations, model, classical_gap, ground_energy)
```

Or it can be created from a specification:

```python
>>> spec = pm.Specification(graph, decision_variables, feasible_configurations, dimod.SPIN)

>>> widget = pm.PenaltyModel.from_specification(spec, model, classical_gap, ground_energy)
```

decision_variables
Maps the feasible configurations to the graph.

Type tuple

classical_gap
The difference in classical energy between the ground state and the first excited state. Must be positive.

Type numeric

feasible_configurations
The set of feasible configurations. The value is the (relative) energy of each of the feasible configurations.

Type dict[tuple[int], number]

graph
The graph that defines the relation between variables in the penalty model. The node labels will be used as the variable labels in the binary quadratic model.

Type networkx.Graph

ground_energy
The minimum energy of all possible configurations.

Type numeric

ising_linear_ranges
Defines the energy ranges available for the linear biases of the penalty model.

Type dict[node, (number, number)]
model
A binary quadratic model that has ground states that match the feasible configurations.

Type `dimod.BinaryQuadraticModel`

ising_quadratic_ranges
Defines the energy ranges available for the quadratic biases of the penalty model.

Type `dict[edge, (number, number)]`

vartype
The variable type.

Type `dimod.Vartype`

classmethod from_specification(specification, model, classical_gap, ground_energy)
Construct a PenaltyModel from a Specification.

Parameters

• `specification (Specification)` – A specification that was used to generate the model.

• `model (dimod.BinaryQuadraticModel)` – A binary quadratic model that has ground states that match the feasible configurations.

• `classical_gap (numeric)` – The difference in classical energy between the ground state and the first excited state. Must be positive.

• `ground_energy (numeric)` – The minimum energy of all possible configurations.

Returns `PenaltyModel`

relabel_variables(mapping, inplace=True)
Relabel the variables and nodes according to the given mapping.

Parameters

• `mapping (dict[hashable, hashable])` – A dict with the current variable labels as keys and new labels as values. A partial mapping is allowed.

• `inplace (bool, optional, default=True)` – If True, the penalty model is updated in-place; otherwise, a new penalty model is returned.

Returns A PenaltyModel with the variables relabeled according to mapping.

Return type `PenaltyModel`

Examples

```python
>>> spec = pm.Specification(nx.path_graph(3), (0, 2), {(-1, -1), (1, 1)},
←dimod.SPIN)
>>> model = dimod.BinaryQuadraticModel({0: 0, 1: 0, 2: 0}, {(0, 1): -1, (1, 2): -1}, 0.0, dimod.SPIN)
>>> penalty_model = pm.PenaltyModel.from_specification(spec, model, 2., -2.)
>>> relabeled_penalty_model = penalty_model.relabel_variables({0: 'a'},
←inplace=False)
>>> relabeled_penalty_model.decision_variables
('a', 2)
```
>>> spec = pm.Specification(nx.path_graph(3), (0, 2), {(-1, -1), (1, 1)},
    →dimod.SPIN)
>>> model = dimod.BinaryQuadraticModel({0: 0, 1: 0, 2: 0}, {(0, 1): -1, (1,
    →2): -1}, 0.0, dimod.SPIN)
>>> penalty_model = pm.PenaltyModel.from_specification(spec, model, 2., -2.)
>>> ___ = penalty_model.relabel_variables({0: 'a'}, inplace=True)
>>> penalty_model.decision_variables
('a', 2)

## Specification

class Specification(graph, decision_variables, feasible_configurations, vartype,
    ising_linear_ranges=None, ising_quadratic_ranges=None,
    min_classical_gap=2)

Specification for a PenaltyModel.

See PenaltyModel documentation for a fuller description of the different components. A specification can be thought of as an incomplete penalty model.

### Parameters

- **graph** ([networkx.Graph/iterable[edge]]) – Defines the structure of the desired binary quadratic model. Each node in the graph represents a variable and each edge defines an interaction between two variables. If given as an iterable of edges, the graph will be constructed by adding each edge to an (initially) empty graph.

- **decision_variables** ([iterable]) – The labels of the penalty model’s decision variables. Each variable label in decision_variables must correspond to a node in graph. Should be an ordered iterable of hashable labels.

- **feasible_configurations** ([dict[tuple[int], number]/iterable[tuple[int]]]) – The set of feasible configurations. Defines the allowed configurations of the decision variables allowed by the constraint. Each feasible configuration should be a tuple, each element of which must be of a value matching vartype. If given as a dict, the key is the feasible configuration and the value is the desired relative energy. If given as an iterable, it will be case to a dict where the relative energies are all 0.

- **vartype** ([dimod.Vartype/str/set]) – The variable type desired for the penalty model. Accepted input values: Vartype.SPIN, 'SPIN', {-1, 1} Vartype.BINARY, 'BINARY', (0, 1)

- **ising_linear_ranges** ([dict[node, [number, number]], optional, default=None]) – When the penalty model is spin-valued, specifies the allowed range for each of the linear biases. If a dict, should be of the form `{v: [min, max], ...}` where v is a variable in the desired penalty model and (min, max) defines the acceptable range for the linear bias associated with v. If None, the default will be set to `{v: [-1, 1], ...}` for each v in graph. A partial assignment is allowed.

- **ising_quadratic_ranges** ([dict[node, dict[node, [number, number]]], optional, default=None]) – When the penalty model is spin-valued, specifies the allowed range for each of the quadratic biases. If a dict, should be of the form `{v: {u: [min, max], ...}, ...}` where u and v are variables in the desired penalty model and u, v have an interaction - there is an edge between nodes u, v in graph. (min, max) the acceptable range for the quadratic bias associated with u, v. If None, the default will be set to `{v: [min, max], ...}, u: {v: [min, max], ...}, ...` for each edge u, v in graph. A partial assignment is allowed.
Examples

```python
>>> import networkx as nx
>>> import dimod

>>> graph = nx.path_graph(5)  # the ends of the path
>>> decision_variables = (0, 4)  # we want the ends of the path to agree
>>> feasible_configurations = {(-1, -1), (1, 1)}

>>> vartype = dimod.Vartype.SPIN

>>> spec = pm.Specification(graph, decision_variables, feasible_configurations, vartype)
```

If we want to make the interaction between (0, 1) ferromagnetic (negative):

```python
>>> ising_quadratic_ranges = {0: {1: (-1, 0)}}

>>> spec = pm.Specification(graph, decision_variables, feasible_configurations, vartype)
```

decision_variables

The labels of the penalty model’s decision variables. Each variable label in decision_variables must correspond to a node in graph.

Type tuple

feasible_configurations

The set of feasible configurations. Defines the allowed configurations of the decision variables allowed by the constraint. The key is the allowed configuration, the value is the relative energy of each configuration.

Type dict[tuple[int], number]

graph

Defines the structure of the desired binary quadratic model. Each node in the graph represents a variable and each edge defines an interaction between two variables.

Type networkx.Graph

ising_linear_ranges

When the penalty model is spin-valued, specifies the allowed range for each of the linear biases. A dict of the form \{v: [min, max], \ldots \} where v is a variable in the desired penalty model and [min, max] defines the acceptable range for the linear bias associated with v.

Type dict[node, [number, number], optional, default=None

ising_quadratic_ranges

When the penalty model is spin-valued, specifies the allowed range for each of the quadratic biases. A dict of the form \{v: \{u: [min, max], \ldots \}, u: \{v: [min, max], \ldots \}, \ldots \} where u and v are variables in the desired penalty model and u, v have an interaction - there is an edge between nodes u, v in graph.

Type dict[node, dict[node, [number, number]]], optional, default=None

min_classical_gap

This is a threshold value for the classical gap. It describes the minimum energy gap between the highest feasible state and the lowest infeasible state. Default value is 2.

Type float

relabel_variables (mapping, inplace=True)

Relabel the variables and nodes according to the given mapping.

Parameters

- mapping (dict) – a dict mapping the current variable/node labels to new ones.
• `inplace(bool, optional, default=True)` – If True, the specification is updated in-place; otherwise, a new specification is returned.

**Returns** A Specification with the variables relabeled according to mapping. If copy=False returns itself, if copy=True returns a new Specification.

**Return type** `Specification`

### Using PenaltyModel Factories

penaltymodel provides functionality for accessing PenaltyModel factories.

#### Accessing Factories

Any factories that have been identified through the `FACTORY_ENTRYPOINT` entrypoint and installed on the python path can be accessed through the `get_penalty_model()` function.

#### Examples

```python
>>> import networkx as nx
>>> import dimod

>>> graph = nx.path_graph(5)
>>> decision_variables = (0, 4)  # the ends of the path
>>> feasible_configurations = {(-1, -1), (1, 1)}  # we want the ends of the path to agree

>>> spec = pm.Specification(graph, decision_variables, feasible_configurations, dimod.SPIN)
>>> widget = pm.get_penalty_model(spec)
```

### Functions and Utilities

`FACTORY_ENTRYPOINT = 'penaltymodel_factory'`
constant used when assigning entrypoints for factories.

**Type** `str`

`CACHE_ENTRYPOINT = 'penaltymodel_cache'`
constant used when assigning entrypoints for caches.

**Type** `str`

`get_penalty_model(specification)`
Retrieve a PenaltyModel from one of the available factories.

**Parameters** `specification(Specification)` – The specification for the desired Penalty-Model.

**Returns** A PenaltyModel as returned by the highest priority factory, or None if no factory could produce it.

**Return type** `PenaltyModel/None`

**Raises** ImpossiblePenaltyModel – If the specification describes a penalty model that cannot be built by any factory.
penaltymodel_factory (priority)
Decorator to assign a priority attribute to the decorated function.

**Parameters**

- **priority** *(int)* — The priority of the factory. Factories are queried in order of decreasing priority.

**Examples**

Decorate penalty model factories like:

```python
>>> @pm.penaltymodel_factory(105)
... def factory_function(spec):
...     pass
>>> factory_function.priority
105
```

iter_factories()
Iterate through all factories identified by the factory entrypoint.

**Yields**

- **function** — A function that accepts a Specification and returns a PenaltyModel.

iter_caches()
Iterator over the PenaltyModel caches.

**Yields**

- **function** — A function that accepts a PenaltyModel and caches it.

**Exceptions**

- **FactoryException**
  General exception for a factory being not able to produce a penalty model.

- **ImpossiblePenaltyModel**
  PenaltyModel is impossible to build.

- **MissingPenaltyModel**
  PenaltyModel is missing from the cache or otherwise unavailable.

**License**

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

**TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION**

1. Definitions.

   - “License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.
   - “Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.
   - “Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such
entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination with their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.
penaltymodel-cache

A local cache for penalty models. Serves as a factory and cache for penaltymodel.

On install, penaltymodel_cache registers an entry point that can be read by penaltymodel. By identifying itself as both a cache and a factory, it will be used automatically by any project that uses penaltymodel’s get_penalty_model function. It will also be automatically populated.

Interface

This module has the primary public-facing methods for the project.

get_penalty_model(specification, database=None)

Factory function for penaltymodel_cache.

Parameters

• specification (penaltymodel.Specification) – The specification for the desired penalty model.

• database (str, optional) – The path to the desired sqlite database file. If None, will use the default.

• priority (int) – 100

Returns Penalty model with the given specification.

Return type penaltymodel.PenaltyModel

Raises penaltymodel.MissingPenaltyModel – If the penalty model is not in the cache.

cache_penalty_model(penalty_model, database=None)

Caching function for penaltymodel_cache.

Parameters

• penalty_model (penaltymodel.PenaltyModel) – Penalty model to be cached.

• database (str, optional) – The path to the desired sqlite database file. If None, will use the default.

Database Utilities

Utilities for access to the sqlite cache.

cache_connect(database=None)

Returns a connection object to a sqlite database.

Parameters database (str, optional) – The path to the database the user wishes to connect to. If not specified, a default is chosen using cache_file(). If the special database name ‘`:memory:`’ is given, then a temporary database is created in memory.

Returns sqlite3.Connection

insert_graph(cur, nodelist, edgelist, encoded_data=None)

Insert a graph into the cache.

A graph is stored by number of nodes, number of edges and a json-encoded list of edges.

Parameters
• **cur** (*sqlite3.Cursor*) – An sqlite3 cursor. This function is meant to be run within a `with` statement.

• **nodelist** (*list*) – The nodes in the graph.

• **edgelist** (*list*) – The edges in the graph.

• **encoded_data** (*dict, optional*) – If a dictionary is provided, it will be populated with the serialized data. This is useful for preventing encoding the same information many times.

### Notes

This function assumes that the nodes are index-labeled and range from 0 to `num_nodes - 1`.

In order to minimize the total size of the cache, it is a good idea to sort the nodelist and edgelist before inserting.

### Examples

```python
>>> nodelist = [0, 1, 2]
>>> edgelist = [(0, 1), (1, 2)]
>>> with pmc.cache_connect(':memory:') as cur:
...    pmc.insert_graph(cur, nodelist, edgelist)

>>> nodelist = [0, 1, 2]
>>> edgelist = [(0, 1), (1, 2)]
>>> encoded_data = {}
>>> with pmc.cache_connect(':memory:') as cur:
...    pmc.insert_graph(cur, nodelist, edgelist, encoded_data)
...    encoded_data['num_nodes']
3
...    encoded_data['num_edges']
2
...    encoded_data['edges']
'[0,1],[1,2]'
```

### iter_graph(*cur*)

Iterate over all graphs in the cache.

**Parameters**

- **cur** (*sqlite3.Cursor*) – An sqlite3 cursor. This function is meant to be run within a `with` statement.

**Yields**

- **tuple** –
  
  A 2-tuple containing:

  - list: The nodelist for a graph in the cache.
  - list: the edgelist for a graph in the cache.

### Examples

```python
>>> nodelist = [0, 1, 2]
>>> edgelist = [(0, 1), (1, 2)]
>>> with pmc.cache_connect(':memory:') as cur:
...    pmc.insert_graph(cur, nodelist, edgelist)
```

(continues on next page)
... list(pmc.iter_graph(cur))
[(0, 1, 2), (0, 1, 1, 2), (0, 1, 1, 2, 2)]

**insert_feasible_configurations** *(cur, feasible_configurations, encoded_data=None)*

Insert a group of feasible configurations into the cache.

**Parameters**

- **cur** *(sqlite3.Cursor)* – An sqlite3 cursor. This function is meant to be run within a with statement.
- **feasible_configurations** *(dict[tuple[int]])* – The set of feasible configurations. Each key should be a tuple of variable assignments. The values are the relative energies.
- **encoded_data** *(dict, optional)* – If a dictionary is provided, it will be populated with the serialized data. This is useful for preventing encoding the same information many times.

**Examples**

```python
given feasible_configurations = {(-1, -1): 0.0, (+1, +1): 0.0}
with pmc.cache_connect(':memory:') as cur:
... pmc.insert_feasible_configurations(cur, feasible_configurations)
```

**iter_feasible_configurations** *(cur)*

Iterate over all of the sets of feasible configurations in the cache.

**Parameters**

- **cur** *(sqlite3.Cursor)* – An sqlite3 cursor. This function is meant to be run within a with statement.

**Yields** *(dict[tuple(int) – number])*: The feasible_configurations.

**insert_ising_model** *(cur, nodelist, edgelist, linear, quadratic, offset, encoded_data=None)*

Insert an Ising model into the cache.

**Parameters**

- **cur** *(sqlite3.Cursor)* – An sqlite3 cursor. This function is meant to be run within a with statement.
- **nodelist** *(list)* – The nodes in the graph.
- **edgelist** *(list)* – The edges in the graph.
- **linear** *(dict)* – The linear bias associated with each node in nodelist.
- **quadratic** *(dict)* – The quadratic bias associated with each edge in edgelist.
- **offset** *(float)* – The constant offset applied to the ising problem.
- **encoded_data** *(dict, optional)* – If a dictionary is provided, it will be populated with the serialized data. This is useful for preventing encoding the same information many times.

**iter_ising_model** *(cur)*

Iterate over all of the Ising models in the cache.

**Parameters**

- **cur** *(sqlite3.Cursor)* – An sqlite3 cursor. This function is meant to be run within a with statement.
Yields `tuple` –

A 5-tuple consisting of:

- list: The nodelist for a graph in the cache.
- list: the edgelist for a graph in the cache.
- dict: The linear biases of an Ising Model in the cache.
- dict: The quadratic biases of an Ising Model in the cache.
- float: The constant offset of an Ising Model in the cache.

`insert_penalty_model(cur, penalty_model)`

Insert a penalty model into the database.

Parameters

- `cur (sqlite3.Cursor)` – An sqlite3 cursor. This function is meant to be run within a `with` statement.
- `penalty_model (penaltymodel.PenaltyModel)` – A penalty model to be stored in the database.

Examples

```python
>>> import networkx as nx
>>> import penaltymodel.core as pm
>>> import dimod
>>> graph = nx.path_graph(3)
>>> decision_variables = (0, 2)
>>> feasible_configurations = {(-1, -1): 0., (+1, +1): 0.}
>>> spec = pm.Specification(graph, decision_variables, feasible_configurations, dimod.SPIN)
>>> linear = {v: 0 for v in graph}
>>> quadratic = {edge: -1 for edge in graph.edges}
>>> model = dimod.BinaryQuadraticModel(linear, quadratic, 0.0, vartype=dimod.SPIN)
>>> widget = pm.PenaltyModel.from_specification(spec, model, 2., -2)
>>> with pmc.cache_connect(':memory:') as cur:
...     pmc.insert_penalty_model(cur, widget)
```

`iter_penalty_model_from_specification(cur, specification)`

Iterate through all penalty models in the cache matching the given specification.

Parameters

- `cur (sqlite3.Cursor)` – An sqlite3 cursor. This function is meant to be run within a `with` statement.
- `specification (penaltymodel.Specification)` – A specification for a penalty model.

Yields `penaltymodel.PenaltyModel`
Cache Information

**APPNAME = 'dwave-penaltymodel-cache'**
The application name is used to determine the cache location.

**APPAUTHOR = 'dwave-systems'**
The application author is used to determine the cache location.

**DATABASENAME = 'penaltymodel_cache_v0.4.1.db'**
The name for the sqlite database itself. Based on the version of the package.

```python
cache_file(app_name='dwave-penaltymodel-cache', app_author='dwave-systems', file_name='penaltymodel_cache_v0.4.1.db')
```

Returns the filename (including path) for the data cache.

The path will depend on the operating system, certain environmental variables and whether it is being run inside a virtual environment. See homebase.

**Parameters**

- **app_name (str, optional)** – The application name. Default is given by `APPNAME`.
- **app_author (str, optional)** – The application author. Default is given by `APPAUTHOR`.
- **filename (str, optional)** – The name of the database file. Default is given by `DATABASENAME`.

**Returns**
The full path to the file that can be used as a cache.

**Return type**
str

**Notes**

Creates the directory if it does not already exist.

If run inside of a virtual environment, the cache will be stored in `/path/to/virtualenv/data/app_name`

**License**

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

**TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION**

1. Definitions.

   “License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   “Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   “Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such
entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the
outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this
License.

“Source” form shall mean the preferred form for making modifications, including but not limited to
software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a
Source form, including but not limited to compiled object code, generated documentation, and
conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available
under the License, as indicated by a copyright notice that is included in or attached to the work (an
example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or
derived from) the Work and for which the editorial revisions, annotations, elaborations, or other
modifications represent, as a whole, an original work of authorship. For the purposes of this Li-
cense, Derivative Works shall not include works that remain separable from, or merely link (or bind
by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and
any modifications or additions to that Work or Derivative Works thereof, that is intentionally sub-
mitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal
Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition,
“submitted” means any form of electronic, verbal, or written communication sent to the Licensor or
its representatives, including but not limited to communication on electronic mailing lists, source
code control systems, and issue tracking systems that are managed by, or on behalf of, the Licen-
sor for the purpose of discussing and improving the Work, but excluding communication that is
conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contri-
bution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contri-
bution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor
hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform,
sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor
hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable
(except as stated in this section) patent license to make, have made, use, offer to sell, sell, import,
and otherwise transfer the Work, where such license applies only to those patent claims licensable
by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination
of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute
patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging
that the Work or a Contribution incorporated within the Work constitutes direct or contributory
patent infringement, then any patent licenses granted to You under this License for that Work shall
terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof
in any medium, with or without modifications, and in Source or Object form, provided that You
meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement You may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.
penaltymodel-maxgap

Generates penalty models using smt solvers. Serves as a factory and cache for penaltymodel.

On install, penaltymodel_maxgap registers an entry point that can be read by penaltymodel. It will be used automatically by any project that uses penaltymodel’s get_penalty_model function.

Interface

get_penalty_model(specification)

Factory function for penaltymodel_maxgap.

Parameters

• specification (penaltymodel.Specification) – The specification for the desired penalty model.
• priority (int) -- 100

Returns Penalty model with the given specification.

Return type penaltymodel.PenaltyModel

Raises penaltymodel.ImpossiblePenaltyModel – If the penalty cannot be built.

SMT Generation

generate(graph, feasible_configurations, decision_variables, linear_energy_ranges, quadratic_energy_ranges, min_classical_gap, smt_solver_name=None)

Generates the Ising model that induces the given feasible configurations. The code is based on the papers\(^1\) and\(^2\).

Parameters

• graph (nx.Graph) – The target graph on which the Ising model is to be built.
• feasible_configurations (dict) – The set of feasible configurations of the decision variables. The key is a feasible configuration as a tuple of spins, the values are the associated energy.
• decision_variables (list/tuple) – Which variables in the graph are assigned as decision variables.
• linear_energy_ranges (dict, optional) – A dict of the form {v: (min, max), ...} where min and max are the range of values allowed to v.
• quadratic_energy_ranges (dict) – A dict of the form {(u, v): (min, max), ...} where min and max are the range of values allowed to (u, v).
• min_classical_gap (float) – The minimum energy gap between the highest feasible state and the lowest infeasible state.
• smt_solver_name (str/None) – The name of the smt solver. Must be a solver available to pysmt. If None, uses the pysmt default.

Returns

A 4-tuple containing:

---


dimod.BinaryQuadraticModel

float: The classical energy gap between ground and the first excited state.

Return type: tuple

Raises: ImpossiblePenaltyModel – If the penalty model cannot be built. Normally due to a non-zero infeasible gap.

License

Apache License
Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or
its representatives, including but not limited to communication on electronic mailing lists, source
code control systems, and issue tracking systems that are managed by, or on behalf of, the Licen-
sor for the purpose of discussing and improving the Work, but excluding communication that is
conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contri-
bution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contri-
bution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor
hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform,
sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor
hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable
(except as stated in this section) patent license to make, have made, use, offer to sell, sell, import,
and otherwise transfer the Work, where such license applies only to those patent claims licensable
by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination
of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute
patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging
that the Work or a Contribution incorporated within the Work constitutes direct or contributory
patent infringement, then any patent licenses granted to You under this License for that Work shall
terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof
in any medium, with or without modifications, and in Source or Object form, provided that You
meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License;
and
(b) You must cause any modified files to carry prominent notices stating that You changed the
files; and
(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright,
patent, trademark, and attribution notices from the Source form of the Work, excluding those
notices that do not pertain to any part of the Derivative Works; and
(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works
that You distribute must include a readable copy of the attribution notices contained within
such NOTICE file, excluding those notices that do not pertain to any part of the Derivative
Works, in at least one of the following places: within a NOTICE text file distributed as part
of the Derivative Works; within the Source form or documentation, if provided along with the
Derivative Works; or, within a display generated by the Derivative Works, if and wherever such
third-party notices normally appear. The contents of the NOTICE file are for informational
purposes only and do not modify the License. You may add Your own attribution notices
within Derivative Works that You distribute, alongside or as an addendum to the NOTICE
text from the Work, provided that such additional attribution notices cannot be construed as
modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or
different license terms and conditions for use, reproduction, or distribution of Your modifications,
or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of
the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally
submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions
of this License, without any additional terms or conditions. Notwithstanding the above, nothing
herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

penaltymodel-mip

Generates penalty models using Google Optimization Tools’ Mixed-Integer Programming capability. Serves as a factory and cache for penaltymodel.

On install, penaltymodel-mip registers an entry point that can be read by penaltymodel. It will be used automatically by any project that uses penaltymodel’s `get_penalty_model` function.

Interface

`get_penalty_model(specification)`

Factory function for penaltymodel-mip.

Parameters

- `specification` (`penaltymodel.Specification`) – The specification for the desired penalty model.
- `priority` (`int`) – 100

Returns Penalty model with the given specification.

Return type `penaltymodel.PenaltyModel`

Raises `penaltymodel.ImpossiblePenaltyModel` – If the penalty cannot be built.
Mixed Integer (Linear) Programming Generation

```
generate_bqm(graph, table, decision, linear_energy_ranges=None, quadratic_energy_ranges=None, min_classical_gap=2, precision=7, max_decision=8, max_variables=10, return_auxiliary=False)
```

Get a binary quadratic model with specific ground states.

**Parameters**

- **graph (Graph)** – Defines the structure of the generated binary quadratic model.
- **table (iterable)** – Iterable of valid configurations (of spin-values). Each configuration is a tuple of variable assignments ordered by `decision`.
- **decision (list/tuple)** – The variables in the binary quadratic model which have specified configurations.
- **linear_energy_ranges (dict, optional)** – Dict of the form `{v: (min, max, ...)` where min and max are the range of values allowed to v. The default range is [-2, 2].
- **quadratic_energy_ranges (dict, optional)** – Dict of the form `{(u, v): (min, max), ...}` where min and max are the range of values allowed to (u, v). The default range is [-1, 1].
- **min_classical_gap (float)** – The minimum energy gap between the highest feasible state and the lowest infeasible state.
- **precision (int, optional, default=7)** – Values returned by the optimization solver are rounded to precision digits of precision.
- **max_decision (int, optional, default=4)** – Maximum number of decision variables allowed. The algorithm is valid for arbitrary sizes of problem but can be extremely slow.
- **max_variables (int, optional, default=4)** – Maximum number of variables allowed. The algorithm is valid for arbitrary sizes of problem but can be extremely slow.
- **return_auxiliary (bool, optional, False)** – If True, the auxiliary configurations are returned for each configuration in table.

**Returns**

- `float`: The classical gap.

If return_auxiliary is True:

- `float`: The classical gap.
- `dict`: The auxiliary configurations, keyed on the configurations in table.

**Return type** If return_auxiliary is False

**Raises** ImpossiblePenaltyModel – If the penalty model cannot be built. Normally due to a non-zero infeasible gap.
License

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without
limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

penalty-model-lp

Generates penalty models using scipy.optimize’s Linear Programming capability. Serves as a factory and cache for penaltymodel.

On install, penalty-model-lp registers an entry point that can be read by penaltymodel. It will be used automatically by any project that uses penaltymodel’s get_penalty_model function.

Interface

get_penalty_model (specification)

Factory function for penalty-model-lp.

Parameters

- specification (penaltymodelSpecification) – The specification for the desired penalty model.
- priority (int) – -100

Returns Penalty model with the given specification.

Return type penaltymodel.PenaltyModel

Raises penaltymodel.ImpossiblePenaltyModel – If the penalty cannot be built.

Linear Programming Generation

get_item (dictionary, tuple_key, default_value)

Grab values from a dictionary using an unordered tuple as a key.

Dictionary should not contain None, 0, or False as dictionary values.

Parameters

- dictionary – Dictionary that uses two-element tuple as keys
• **tuple_key** – Unordered tuple of two elements

• **default_value** – Value that is returned when the tuple_key is not found in the dictionary

```python
generate_bqm(graph, table, decision_variables, linear_energy_ranges=None, quadratic_energy_ranges=None, min_classical_gap=2, catch_warnings=True)
```

**Parameters**

• **graph** – A networkx.Graph

• **table** – An iterable of valid spin configurations. Each configuration is a tuple of variable assignments ordered by decision.

• **decision_variables** – An ordered iterable of the variables in the binary quadratic model.

• **linear_energy_ranges** – Dictionary of the form `{v: (min, max), ...}` where min and max are the range of values allowed to `v`. The default range is `[-2, 2]`.

• **quadratic_energy_ranges** – Dict of the form `{(u, v): (min, max), ...}` where min and max are the range of values allowed to `(u, v)`. The default range is `[-1, 1]`.

• **min_classical_gap** – A float. The minimum energy gap between the highest feasible state and the lowest infeasible state.

---

**License**

Apache License

Version 2.0, January 2004

[http://www.apache.org/licenses/](http://www.apache.org/licenses/)

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   “License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   “Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   “Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   “You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

   “Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   “Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.
“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the
6.12 qbsolv

A decomposing solver that finds a minimum value of a large quadratic unconstrained binary optimization (QUBO) problem by splitting it into pieces. The pieces are solved using a classical solver running the tabu algorithm. qbsolv also enables configuring a D-Wave system as the solver.

For more information, see qbsolv documentation.

It is strongly recommended that you use dwave-hybrid instead.
6.13 Indices and tables

- genindex
- search


Python Module Index

d
dimod.bqm.adjarraybqm, 137
dimod.core.composite, 260
dimod.core.initialized, 264
dimod.core.polysampler, 265
dimod.core.sampler, 257
dimod.core.structured, 262
dimod.decorators, 289
dimod.exceptions, 307
dimod.higherorder.polynomial, 283
dimod.higherorder.utils, 287
dimod.reference.composites.clipcomposite, 222
dimod.reference.composites.connectedcomponent, 220
dimod.reference.composites.fixedvariable, 225
dimod.reference.composites.roofduality, 228
dimod.reference.composites.scalecomposite, 230
dimod.reference.composites.spin_transform, 233
dimod.reference.composites.structure, 236
dimod.reference.composites.tracking, 239
dimod.reference.composites.truncatecomposite, 242
dimod.reference.samplers, 209
dimod.reference.samplers.identity_sampler, 211
dimod.reference.samplers.null_sampler, 213
dimod.reference.samplers.randomSampler, 215
dimod.reference.samplers.simulated_annealing, 217
dimod.serialization.coo, 293
dimod.serialization.fileview, 295
dimod.serialization.json, 298
dimod.testing, 301
dimod.vartypes, 306
dwave.cloud.client, 346
dwave.cloud.computation, 364
dwave.cloud.config, 339
dwave.cloud.exceptions, 372
dwave.cloud.gpu, 355
dwave.cloud.solver, 357
dwave.system.composites.embedding, 497
dwave.system.utilities, 541
dwave_networkx, 473
dwave_networkx.algorithms.canonicalization, 435
dwave_networkx.algorithms.clique, 435
dwave_networkx.algorithms.coloring, 438
dwave_networkx.algorithms.cover, 441
dwave_networkx.algorithms.elimination_ordering, 444
dwave_networkx.algorithms.independent_set, 456
dwave_networkx.algorithms.markov, 450
dwave_networkx.algorithms.matching, 451
dwave_networkx.algorithms.max_cut, 454
dwave_networkx.algorithms.social, 459
dwave_networkx.algorithms.tsp, 462
dwave_networkx.default_sampler, 477
dwave_networkx.drawing.chimera_layout, 464
dwave_networkx.exceptions, 476
dwave_networkx.utils, 471
dwave_networkx.utils.decorators, 471
dwavebinarycsp.core.constraint, 318
dwavebinarycsp.core.csp, 311
dwavebinarycsp.factories.constraint, 324
dwavebinarycsp.factories.csp, 328
dwavebinarycsp.io.cnf, 317
dwavebinarycsp.reduction, 317
hybrid.composers, 394
hybrid.core, 381
hybrid.decomposers, 396
hybrid.exceptions, 417
hybrid.flow, 400
hybrid.reference.kerberos, 417
hybrid.reference.pa, 421
hybrid.reference.pt, 419
hybrid.reference.qbsolv, 423
hybrid.samplers, 386
hybrid.traits, 415
hybrid.utils, 406

minorminer, 543

neal.sampler, 429

penaltymodel.cache.cache_manager, 564
penaltymodel.cache.database_manager, 560
penaltymodel.cache.interface, 560
penaltymodel.core.classes.penaltymodel, 551
penaltymodel.core.classes.specification, 554
penaltymodel.core.exceptions, 557
penaltymodel.core.interface, 556
penaltymodel.lp.generation, 574
penaltymodel.lp.interface, 574
penaltymodel.maxgap.generation, 567
penaltymodel.maxgap.interface, 567
penaltymodel.mip.generation, 571
penaltymodel.mip.interface, 570

tabu.sampler, 542
## Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>()</td>
<td>(MinimizeEnergy method)</td>
<td>539</td>
</tr>
<tr>
<td><strong>init</strong>()</td>
<td>(DimodDecoder method)</td>
<td>300</td>
</tr>
<tr>
<td><strong>init</strong>()</td>
<td>(DimodEncoder method)</td>
<td>299</td>
</tr>
<tr>
<td><strong>init</strong>()</td>
<td>(FileView method)</td>
<td>297</td>
</tr>
<tr>
<td>add_constraint()</td>
<td>(ConstraintSatisfactionProblem method)</td>
<td>312</td>
</tr>
<tr>
<td>add_interaction()</td>
<td>(AdjDictBQM method)</td>
<td>154</td>
</tr>
<tr>
<td>add_interaction()</td>
<td>(AdjMapBQM method)</td>
<td>171</td>
</tr>
<tr>
<td>add_interaction()</td>
<td>(AdjVectorBQM method)</td>
<td>187</td>
</tr>
<tr>
<td>add_interactions_from()</td>
<td>(AdjDictBQM method)</td>
<td>154</td>
</tr>
<tr>
<td>add_interactions_from()</td>
<td>(AdjMapBQM method)</td>
<td>171</td>
</tr>
<tr>
<td>add_interactions_from()</td>
<td>(AdjVectorBQM method)</td>
<td>187</td>
</tr>
<tr>
<td>add_offset()</td>
<td>(AdjArrayBQM method)</td>
<td>141</td>
</tr>
<tr>
<td>add_offset()</td>
<td>(AdjDictBQM method)</td>
<td>154</td>
</tr>
<tr>
<td>add_offset()</td>
<td>(AdjMapBQM method)</td>
<td>171</td>
</tr>
<tr>
<td>add_offset()</td>
<td>(AdjVectorBQM method)</td>
<td>187</td>
</tr>
<tr>
<td>add_variable()</td>
<td>(AdjDictBQM method)</td>
<td>155</td>
</tr>
<tr>
<td>add_variable()</td>
<td>(AdjMapBQM method)</td>
<td>171</td>
</tr>
<tr>
<td>add_variable()</td>
<td>(AdjVectorBQM method)</td>
<td>187</td>
</tr>
<tr>
<td>add_variable()</td>
<td>(ConstraintSatisfactionProblem method)</td>
<td>313</td>
</tr>
<tr>
<td>add_variables_from()</td>
<td>(AdjDictBQM method)</td>
<td>155</td>
</tr>
<tr>
<td>add_variables_from()</td>
<td>(AdjMapBQM method)</td>
<td>172</td>
</tr>
<tr>
<td>add_variables_from()</td>
<td>(AdjVectorBQM method)</td>
<td>188</td>
</tr>
<tr>
<td>adj</td>
<td>(AdjArrayBQM attribute)</td>
<td>139</td>
</tr>
<tr>
<td>adj</td>
<td>(AdjDictBQM attribute)</td>
<td>152</td>
</tr>
<tr>
<td>adj</td>
<td>(AdjMapBQM attribute)</td>
<td>168</td>
</tr>
<tr>
<td>adj</td>
<td>(AdjVectorBQM attribute)</td>
<td>184</td>
</tr>
<tr>
<td>adj</td>
<td>(BinaryQuadraticModel attribute)</td>
<td>199</td>
</tr>
<tr>
<td>adjacency</td>
<td>(DWaveSampler attribute)</td>
<td>483</td>
</tr>
<tr>
<td>adjacency</td>
<td>(FixedEmbeddingComposite attribute)</td>
<td>505</td>
</tr>
<tr>
<td>adjacency</td>
<td>(LazyFixedEmbeddingComposite attribute)</td>
<td>509</td>
</tr>
<tr>
<td>adjacency</td>
<td>(Structured attribute)</td>
<td>264</td>
</tr>
<tr>
<td>adjacency</td>
<td>(TilingComposite attribute)</td>
<td>512</td>
</tr>
<tr>
<td>adjacency</td>
<td>(VirtualGraphComposite attribute)</td>
<td>516</td>
</tr>
<tr>
<td>AdjArrayBQM</td>
<td>(class in dimod)</td>
<td>136</td>
</tr>
<tr>
<td>AdjMapBQM</td>
<td>(class in dimod)</td>
<td>166</td>
</tr>
<tr>
<td>AdjVectorBQM</td>
<td>(class in dimod)</td>
<td>182</td>
</tr>
<tr>
<td>aggregate()</td>
<td>(SampleSet method)</td>
<td>271</td>
</tr>
<tr>
<td>and_gate()</td>
<td>(in module dwavebinarycsp.factories.constraint.gates)</td>
<td>324</td>
</tr>
<tr>
<td>APPAUTHOR</td>
<td>(in module penalty-model.cache.cache_manager)</td>
<td>564</td>
</tr>
<tr>
<td>APPNAME</td>
<td>(in module penalty-model.cache.cache_manager)</td>
<td>564</td>
</tr>
<tr>
<td>ArgMin</td>
<td>(class in hybrid.flow)</td>
<td>400</td>
</tr>
<tr>
<td>as_bqm()</td>
<td>(in module dimod)</td>
<td>200</td>
</tr>
<tr>
<td>as_completed()</td>
<td>(Future static method)</td>
<td>366</td>
</tr>
<tr>
<td>as_samples()</td>
<td>(in module dimod)</td>
<td>267</td>
</tr>
<tr>
<td>as_vartype()</td>
<td>(in module dimod)</td>
<td>307</td>
</tr>
<tr>
<td>assert_bqm_almost_equal()</td>
<td>(in module dimod.testing.asserts)</td>
<td>302</td>
</tr>
<tr>
<td>assert_composite_almost_equal()</td>
<td>(in module dimod.testing.asserts)</td>
<td>302</td>
</tr>
<tr>
<td>assert_sampler_energies()</td>
<td>(in module dimod.testing.asserts)</td>
<td>302</td>
</tr>
<tr>
<td>assert_sampler_api()</td>
<td>(in module dimod.testing.asserts)</td>
<td>301</td>
</tr>
<tr>
<td>assert_sampleset_energies()</td>
<td>(in module dimod.testing.asserts)</td>
<td>303</td>
</tr>
<tr>
<td>assert_structured_api()</td>
<td>(in module dimod.testing.asserts)</td>
<td>302</td>
</tr>
<tr>
<td>AutoEmbeddingComposite</td>
<td>(class in dwave.system.composites)</td>
<td>498</td>
</tr>
<tr>
<td>avg_load</td>
<td>(BaseSolver attribute)</td>
<td>363</td>
</tr>
</tbody>
</table>
children (TilingComposite attribute), 512
children (TruncateComposite attribute), 243
children (VirtualGraphComposite attribute), 516
Chimera, 85
chimera_anticluster() (in module dimod.generators), 201
chimera_coordinates (class in dwave_networkx), 473
chimera_elimination_order() (in module dwave_networkx.algorithms.elimination_ordering), 445
chimera_graph() (in module dwave_networkx), 466
chimera_layout() (in module dwave_networkx.drawing.chimera_layout), 465
chimera_tiles() (in module hybrid.utils), 409
classical_gap (PenaltyModel attribute), 552
clear() (TrackingComposite method), 241
Client (class in dwave.cloud.client), 347
Client (class in dwave.cloud.qpu), 356
Client (class in dwave.cloud.sw), 356
ClipComposite (class in dimod.reference.composites.clipcomposite), 222
clique_number() (in module dwave_networkx), 436
close() (Client method), 355
combinations() (in module dimod.generators), 202
common_working_graph() (in module dwave.system.utilities), 541
Complete graph, 85
Composed sampler, 85
ComposedPolySampler (class in dimod), 267
ComposedSampler (class in dimod), 261
Composite, 85
Composite (class in dimod), 261
concatenate() (in module dimod), 282
ConfigFileError, 372
ConfigFileParseError, 372
ConfigFileReadError, 372
configurations (Constraint attribute), 319
connected_components() (in module dimod.traversal), 206
ConnectedComponentsComposite (class in dimod.reference.composites.connectedcomponent), 220
Const (class in hybrid.flow), 401
Constraint (class in dwavebinarycsp), 318
constraints (ConstraintSatisfactionProblem attribute), 311
ConstraintSatisfactionProblem (class in dwavebinarycsp), 311
contract_variables() (AdjDictBQM method), 156
contract_variables() (AdjMapBQM method), 172
cutOffComposite (class in dwave.system.composites), 492
data() (SampleSet method), 273
DATABASENAME (in module penalty-model.cache.cache_manager), 564
decision_variables (PenaltyModel attribute), 552
decision_variables (Specification attribute), 555
degree (BinaryPolynomial attribute), 283
degree() (AdjArrayBQM method), 141
degree() (AdjDictBQM method), 156
degree() (AdjMapBQM method), 172
degree() (AdjVectorBQM method), 188
degrees() (AdjArrayBQM method), 142
degrees() (AdjDictBQM method), 156
degrees() (AdjMapBQM method), 173
degrees() (AdjVectorBQM method), 189
diagnose_embedding() (in module dwave.embedding), 534
dimod.bqm.adjarraybqm (module), 137
dimod.core.composite (module), 260
dimod.core.initialized (module), 264
dimod.core.polysampler (module), 265
dimod.core.sampler (module), 257
dimod.core.structured (module), 262
dimod.decorators (module), 289
dimod.exceptions (module), 307
dimod.higherorder.polynomial (module), 283
dimod.higherorder.utils (module), 287
dimod.reference.composites.clipcomposite (module), 222
dimod.reference.composites.connectedcomponent (module), 220
dimod.reference.composites.fixedvariable (module), 225
dimod.reference.composites.roofduality (module), 228
dimod.reference.composites.scalecomposite (module), 230
dimod.reference.composites.spin_transform (module), 233
dimod.reference.composites.structure (module), 236
Index
RoofDualityDecomposer (class in hybrid.decomposers), 397
run() (Runnable method), 385
Runnable (class in hybrid.core), 382
RunnableError, 417

S
sample() (AutoEmbeddingComposite method), 499
sample() (ClipComposite method), 224
sample() (ConnectedComponentsComposite method), 221
sample() (CutOffComposite method), 493
sample() (DWaveCliqueSampler method), 486
sample() (DWaveSampler method), 483
sample() (EmbeddingComposite method), 502
sample() (ExactPolySolver method), 246
sample() (ExactSolver method), 210
sample() (FixedEmbeddingComposite method), 506
sample() (FixedVariableComposite method), 226
sample() (IdentitySampler method), 212
sample() (KerberosSampler method), 418
sample() (LazyFixedEmbeddingComposite method), 509
sample() (LeapHybridSampler method), 489
sample() (NullSampler method), 214
sample() (RandomSampler method), 216
sample() (ReverseAdvanceComposite method), 522
sample() (ReverseBatchStatesComposite method), 520
sample() (RoofDualityComposite method), 229
sample() (Sampler method), 259
sample() (ScaleComposite method), 232
sample() (SimulatedAnnealingSampler method), 218, 431
sample() (SpinReversalTransformComposite method), 234
sample() (StructureComposite method), 238
sample() (TabuSampler method), 542
sample() (TilingComposite method), 513
sample() (TrackingComposite method), 241
sample() (TruncateComposite method), 244
sample() (VirtualGraphComposite method), 517
sample_as_dict() (in module hybrid.utils), 411
sample_as_list() (in module hybrid.utils), 412
sample_bqm() (UnstructuredSolver method), 362
sample_hising() (HigherOrderComposite method), 248
sample_hising() (PolyCutOffComposite method), 497
sample_hising() (PolyFixedVariableComposite method), 250
sample_hising() (PolySampler method), 266
sample_hising() (PolyScaleComposite method), 253
sample_hising() (PolyTruncateComposite method), 255
sample_hubo() (HigherOrderComposite method), 248
sample_hubo() (PolyCutOffComposite method), 497
sample_hubo() (PolyFixedVariableComposite method), 251
sample_hubo() (PolySampler method), 266
sample_hubo() (PolyScaleComposite method), 253
sample_hubo() (PolyTruncateComposite method), 256
sample_ising() (AutoEmbeddingComposite method), 499
sample_ising() (ClipComposite method), 224
sample_ising() (ConnectedComponentsComposite method), 499
sample_ising() (CutOffComposite method), 493
sample_ising() (DWaveCliqueSampler method), 486
sample_ising() (DWaveSampler method), 483
sample_ising() (EmbeddingComposite method), 502
sample_ising() (ExactPolySolver method), 246
sample_ising() (ExactSolver method), 210
sample_ising() (FixedEmbeddingComposite method), 506
sample_ising() (FixedVariableComposite method), 226
sample_ising() (IdentitySampler method), 212
sample_ising() (KerberosSampler method), 418
sample_ising() (LazyFixedEmbeddingComposite method), 509
sample_ising() (LeapHybridSampler method), 489
sample_ising() (NullSampler method), 214
sample_ising() (RandomSampler method), 216
sample_ising() (ReverseAdvanceComposite method), 522
sample_ising() (ReverseBatchStatesComposite method), 520
sample_ising() (RoofDualityComposite method), 229
sample_ising() (Sampler method), 259
sample_ising() (ScaleComposite method), 232
sample_ising() (SimulatedAnnealingSampler method), 218, 431
sample_ising() (SpinReversalTransformComposite method), 234
sample_ising() (StructureComposite method), 238
sample_ising() (TabuSampler method), 542
sample_ising() (TilingComposite method), 513
sample_ising() (TrackingComposite method), 241
sample_ising() (TruncateComposite method), 244
sample_ising() (VirtualGraphComposite method), 518
StructuredComposite (class in hybrid.composers), 382

StructuredSubproblemSampler (class in hybrid.samplers), 388

SingleInputState (class in hybrid.traits), 416

SingleOutputState (class in hybrid.traits), 416

SISO (class in hybrid.traits), 416

slice (SampleSet method), 279

software (BaseSolver attribute), 363

Solver, 86

Solver (in module dwave.cloud.solver), 358

SolverAuthenticationError, 373

SolverError, 372

SolverFailureError, 372

SolverNotFoundError, 372

SolverOfflineError, 372

SolverPropertyMissingError, 373

solvers () (Client method), 350

Source, 86

Source graph, 86

Specification (class in penalty-model.core.classes.specification), 554

spin (AdjArrayBQM attribute), 140

spin (AdjDictBQM attribute), 152

spin (AdjMapBQM attribute), 169

spin (AdjVectorBQM attribute), 185

SPIN (BinaryQuadraticModel attribute), 199

SPIN (Vartype attribute), 306

SpinReversalTransformComposite (class in dimod.reference.composites.spin_transform), 233

SplatComposer (class in hybrid.composers), 394

State (class in hybrid.core), 382

StateDimensionalityError, 417

States (class in hybrid.core), 382

StateTraitMissingError, 417

StateTraits (class in hybrid.traits), 415

stitch () (in module dwavebinarycsps), 315

stop () (Runnable method), 385

structural_imbalance () (in module dwave_networkx.algorithms.social), 459

structural_imbalance_ising () (in module dwave_networkx.algorithms.social), 462

structure (DWaveSampler attribute), 483

structure (FixedEmbeddingComposite attribute), 506

structure (LazyFixedEmbeddingComposite attribute), 509

structure (Structured attribute), 264

structure (TilingComposite attribute), 512

structure (VirtualGraphComposite attribute), 516

StructureComposite (class in dimod.reference.composites.structure), 236

Structured (class in dimod), 263

Structured sampler, 87

StructuredSolver (class in dwave.cloud.solver), 358

Subgraph, 87

SubproblemIntaking (class in hybrid.traits), 416

SubproblemProducing (class in hybrid.traits), 416

SubproblemSampler (class in hybrid.traits), 416

SubsamplesComposer (class in hybrid.traits), 416

SubsamplesIntaking (class in hybrid.traits), 416

SubsamplesProcessor (class in hybrid.traits), 416

SubsamplesProducing (class in hybrid.traits), 416

swap_pair () (SwapReplicaPairRandom method), 420

SwapReplicaPairRandom (class in hybrid.reference.pt), 420

SwapReplicasDownsweep (class in hybrid.reference.pt), 420

T

tabu. sampler (module), 542

TabuProblemSampler (class in hybrid.samplers), 389

TabuSampler (class in tabu.sampler), 542

TabuSubproblemSampler (class in hybrid.samplers), 389

Target, 87

Target.graph, 87

TilingChimeraDecomposer (class in hybrid.decomposers), 398

TilingComposite (class in dwave.system.composites), 510

Timeout, 373

timing (Future attribute), 372

to_binary () (BinaryPolynomial method), 287

to_coo () (AdjArrayBQM method), 147

to_coo () (AdjDictBQM method), 163

to_coo () (AdjMapBQM method), 179

to_coo () (AdjVectorBQM method), 195

to_hising () (BinaryPolynomial method), 287

to_hubo () (BinaryPolynomial method), 287

to_ising () (AdjArrayBQM method), 148

to_ising () (AdjDictBQM method), 163

to_ising () (AdjMapBQM method), 179

to_ising () (AdjVectorBQM method), 195

to_networkx_graph () (AdjArrayBQM method), 148

to_networkx_graph () (AdjDictBQM method), 164

to_networkx_graph () (AdjMapBQM method), 180

to_networkx_graph () (AdjVectorBQM method), 196

to_networkx_graph () (in module dimod), 206

to_numpy_matrix () (AdjArrayBQM method), 148

to_numpy_matrix () (AdjDictBQM method), 164

to_numpy_matrix () (AdjMapBQM method), 180

to_numpy_matrix () (AdjVectorBQM method), 196

to_numpy_vectors () (AdjArrayBQM method), 149
to_numpy_vectors() (AdjDictBQM method), 164

variables (ConstraintSatisfactionProblem attribute), 311

variables (Future attribute), 370

variables (SampleSet attribute), 270

vartype (AdjArrayBQM attribute), 139

vartype (AdjDictBQM attribute), 151

vartype (AdjMapBQM attribute), 168

vartype (AdjVectorBQM attribute), 184

vartype (BinaryPolynomial attribute), 283

vartype (BinaryQuadraticModel attribute), 199

Vartype (class in dimod), 306

vartype (Constraint attribute), 319

vartype (ConstraintSatisfactionProblem attribute), 312

vartype (PenaltyModel attribute), 553

vartype (SampleSet attribute), 270

vartype_argument() (in module dimod.decorators), 290

verify_embedding() (in module dwave.embedding), 535

vertex_color() (in module dwave_networkx.algorithms.coloring), 440

vertex_color_qubo() (in module dwave_networkx.algorithms.coloring), 441

VirtualGraphComposite (class in dwave.system.composites), 514

Wait (class in hybrid.flow), 406

wait() (Future method), 367

wait_multiple() (Future static method), 367

warnings_default (EmbeddingComposite attribute), 502

weighted_maximum_cut() (in module dwave_networkx.algorithms.max_cut), 455

weighted_random() (in module dwave.embedding.chain_breaks), 537

Working graph, 87

WriteableError, 307

X

xor_gate() (in module dwavebinarycsp.factories.constraint.gates), 325