dwave-system Documentation

Release 0.9.1

D-Wave Systems Inc

Feb 25, 2020

Contents

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.

CHAPTER 1

Documentation

Note: This documentation is for the latest version of dwave-system. Documentation for the version currently installed by dwave-ocean-sdk is here: dwave-system.

1.1 Introduction

dwave-system enables easy incorporation of the D-Wave system as a sampler—the component used to find variable values that minimize the binary quadratic model (BQM) representing a problem—in the typical Ocean problem-solving procedure:

1. Formulate the problem as a BQM.
2. Solve the BQM with a sampler.

You can incorporate the D-Wave system in either a hybrid quantum-classical solution, using *LeapHybridSampler()* or dwave-hybrid samplers such as `KerberosSampler()`, or directly using *DWaveSampler()*.

1.1.1 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. (Access to a D-Wave system has been set up in a configuration file that is used implicitly.) 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.

```
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import EmbeddingComposite
```

(continues on next page)

(continued from previous page)

```
...
>>> 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]
```

1.1.2 Using the D-Wave System as a Sampler

The dimod API makes it possible to easily interchange samplers in your code. For example, you might develop code using dwave_neal, Ocean's simulated annealing sampler, and then swap in a D-Wave system composed sampler.

Using a D-Wave System explains how you set up access to a D-Wave system.

D-Wave System Documentation describes the D-Wave system, its features, parameters, and properties. The documentation provides guidance on programming the D-Wave system, including how to formulate problems and configure parameters.

1.1.3 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.

1.1.4 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()`, used in the example above, and `VirtualGraphComposite()`, both of which handle the mapping known as minor-embedding.

1.1.5 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.)

See the *Embedding* section for more information on minor-embedding and the provided functionality.

1.2 Reference Documentation

1.2.1 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.

DWaveSampler

class DWaveSampler (*failover=False*, *retry_interval=-1*, ***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 propogate the *SolverNotFoundError* to the user.
- **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 autentication token, are implicitly set in a configuration file or as environment variables, as described in Configuring a D-Wave System.

```
>>> from dwave.system.samplers import DWaveSampler
>>> sampler = DWaveSampler(solver={'qubits__issuperset': {0, 1}})
>>> sampleset = sampler.sample_ising({0: -1, 1: 1}, {})
>>> for sample in sampleset.samples():  # doctest: +SKIP
...    print(sample)
...
{0: 1, 1: -1}
```

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.

DWaveSampler.properties	D-Wave solver properties as returned by a SAPI query.
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.
DWaveSampler.nodelist	List of active qubits for the D-Wave solver.
DWaveSampler.edgelist	List of active couplers for the D-Wave solver.
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.
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.

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

```
>>> from dwave.system.samplers import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.properties    # doctest: +SKIP
{u'anneal_offset_ranges': [[-0.2197463755538704, 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

```
>>> from dwave.system.samplers 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

```
>>> from dwave.system.samplers import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.nodelist    # doctest: +SKIP
[0,
 1,
 2,
# Snipped above response for brevity
```

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

```
>>> from dwave.system.samplers import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.edgelist    # doctest: +SKIP
[(0, 4),
 (0, 5),
 (0, 6),
 (0, 7),
 (0, 128),
 (1, 4),
# Snipped above response for brevity
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

Type list

dwave.system.samplers.DWaveSampler.adjacency

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]

dwave.system.samplers.DWaveSampler.structure

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

DWaveSampler.sample(bqm, **parameters)	Sample from a binary quadratic model.
DWaveSampler.sample_ising(h, J[, warnings])	Sample from the specified Ising model.
DWaveSampler.sample_qubo(Q[, warnings])	Sample from the specified QUBO.
DWaveSampler.validate_anneal_schedule(...)	Raise an exception if the specified schedule is invalid for the sampler.

dwave.system.samplers.DWaveSampler.sample

DWaveSampler.**sample**(*bqm*, ***parameters*)

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()*).

:param `BinaryQuadraticModel`: A binary quadratic model. :param **kwargs: See the implemented sampling for additional keyword definitions.

Returns `SampleSet`

See also:

sample_ising(), *sample_qubo()*

dwave.system.samplers.DWaveSampler.sample_ising

DWaveSampler.**sample_ising**(*h*, *J*, *warnings=None*, ***kwargs*)

Sample from the specified Ising model.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising model. 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, except in the case of missing qubits in which case 0 biases are ignored while a non-zero bias set on a missing qubit raises an error.
- **J** (*dict[(int, int)* – float]): Quadratic biases of the Ising model.
- **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.

```
>>> from dwave.system.samplers import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampleset = sampler.sample_ising({0: -1, 1: 1}, {})
>>> for sample in sampleset.samples():    # doctest: +SKIP
...    print(sample)
...
{0: 1, 1: -1}
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

dwave.system.samplers.DWaveSampler.sample_qubo

DWaveSampler.**sample_qubo**(*Q*, *warnings=None*, ***kwargs*)

Sample from the specified QUBO.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) model.
- **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 QUBO mapped directly to qubits 0 and 4 on a D-Wave system.

```
>>> from dwave.system.samplers import DWaveSampler
>>> sampler = DWaveSampler()
>>> Q = {(0, 0): -1, (4, 4): -1, (0, 4): 2}
>>> sampleset = sampler.sample_qubo(Q)
>>> for sample in sampleset.samples():    # doctest: +SKIP
...    print(sample)
...
{0: 0, 4: 1}
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

dwave.system.samplers.DWaveSampler.validate_anneal_schedule

`DWaveSampler.`**`validate_anneal_schedule`**`(`*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, 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.

```
>>> from dwave.system.samplers 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
>>>
```

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.

```
>>> 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.
`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.

dwave.system.samplers.LeapHybridSampler.properties

`LeapHybridSampler.`**`properties`**

solver properties as returned by a SAPI query.

Solver properties are dependent on the selected solver and subject to change.

Type dict

dwave.system.samplers.LeapHybridSampler.parameters

`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.

Solver parameters are dependent on the selected solver and subject to change.

Type dict[str, list]

Methods

`LeapHybridSampler.sample`(bqm[, time_limit])	Sample from the specified binary quadratic model.
`LeapHybridSampler.sample_ising`(h, J, ...)	Sample from an Ising model using the implemented sample method.
`LeapHybridSampler.sample_qubo`(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dwave.system.samplers.LeapHybridSampler.sample

`LeapHybridSampler.`**`sample`**(*bqm*, *time_limit=None*, ***kwargs*)

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 *LeapHybrid-*

Sampler().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.

```
>>> 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.samplers.LeapHybridSampler.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

- **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.LeapHybridSampler.sample_qubo

LeapHybridSampler.**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()*

1.2.2 Composites

dimod composites that provide layers of pre- and post-processing (e.g., minor-embedding) when using the D-Wave system.

Contents

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.

```
>>> import dimod
>>> sampler = DWaveSampler(solver={'qpu': True})
>>> bqm = dimod.BinaryQuadraticModel({'a': -1, 'b': 1, 'c': 1},    # doctest:␣
↪+SKIP
...                                  {'ab': -0.8, 'ac': -0.7, 'bc': -1},
...                                  0,
...                                  dimod.SPIN)
>>> CutOffComposite(AutoEmbeddingComposite(sampler), 0.75).sample(bqm,
...                 num_reads=1000).first.sample # doctest: +SKIP
{'a': -1, 'b': -1, 'c': -1}
```

Properties

CutOffComposite.child	The child sampler.
CutOffComposite.children	List of child samplers that that are used by this composite.
CutOffComposite.properties	A dict containing any additional information about the sampler.
CutOffComposite.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.

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 relevent to each parameter.

Methods

CutOffComposite.sample(bqm, **parameters)	Cut off interactions and sample from the provided binary quadratic model.
CutOffComposite.sample_ising(h, J, **parameters)	Sample from an Ising model using the implemented sample method.
CutOffComposite.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

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.
- **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.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.

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().

```
>>> 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.

Continued on next page

Table 7 – 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 relevent 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 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 relevent 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_hubo(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 hueristic 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

`AutoEmbeddingComposite.child`	The child sampler.
`AutoEmbeddingComposite.parameters`	
`AutoEmbeddingComposite.properties`	

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`(bqm, **parameters)	Sample from the provided binary quadratic model.
`AutoEmbeddingComposite.sample_ising`(h, J, ...)	Sample from an Ising model using the implemented sample method.
`AutoEmbeddingComposite.sample_qubo`(Q, ...)	Sample from a QUBO using the implemented sample method.

dwave.system.composites.AutoEmbeddingComposite.sample

`AutoEmbeddingComposite.`**`sample`**(*bqm*, ***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 * *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.
- **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.AutoEmbeddingComposite.sample_ising

`AutoEmbeddingComposite.`**`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.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=<built-in 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.Structured.structure`. Defaults to `dimod.child_structure_dfs()`.

Examples

```
>>> from dwave.system import DWaveSampler, EmbeddingComposite
...
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> h = {'a': -1., 'b': 2}
>>> J = {('a', 'b'): 1.5}
>>> sampleset = sampler.sample_ising(h, J)
```

Properties

EmbeddingComposite.child	The child sampler.
EmbeddingComposite.parameters	Parameters in the form of a dict.
EmbeddingComposite.properties	Properties in the form of a dict.
EmbeddingComposite.return_embedding_default	Defines the default behaviour for *sample()*'s *return_embedding* kwarg.
EmbeddingComposite.warnings_default	Defines the default behabior for *sample()*'s *warnings* kwarg.

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]

dwave.system.composites.EmbeddingComposite.properties

`EmbeddingComposite.`**properties = None**

Properties in the form of a dict.

Contains the properties of the child sampler.

Type dict

dwave.system.composites.EmbeddingComposite.return_embedding_default

EmbeddingComposite.**return_embedding_default = False**

Defines the default behaviour for `sample()`'s *return_embedding* kwarg.

dwave.system.composites.EmbeddingComposite.warnings_default

EmbeddingComposite.**warnings_default = 'ignore'**

Defines the default behabior for `sample()`'s *warnings* kwarg.

Methods

`EmbeddingComposite.sample`(bqm[, ...])	Sample from the provided binary quadratic model.
`EmbeddingComposite.sample_ising`(h, J, ...)	Sample from an Ising model using the implemented sample method.
`EmbeddingComposite.sample_qubo`(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dwave.system.composites.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 * *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.
- **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()*

dwave.system.composites.EmbeddingComposite.sample_qubo

`EmbeddingComposite.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()*

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

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import 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})
```

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.
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.

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]

dwave.system.composites.FixedEmbeddingComposite.structure

`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

`FixedEmbeddingComposite.sample`(bqm, **parameters)	Sample the binary quadratic model.

Continued on next page

Table 14 – continued from previous page

FixedEmbeddingComposite.sample_ising(h, J, . . .)	Sample from an Ising model using the implemented sample method.
FixedEmbeddingComposite.sample_qubo(Q, . . .)	Sample from a QUBO using the implemented sample method.

dwave.system.composites.FixedEmbeddingComposite.sample

`FixedEmbeddingComposite.`**`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`

dwave.system.composites.FixedEmbeddingComposite.sample_ising

`FixedEmbeddingComposite.`**`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.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=<built-in 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

```
>>> 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})
```

(continues on next page)

(continued from previous page)

```
>>> sampler.nodelist  # has structure based on given problem
['a', 'b']
```

Properties

LazyFixedEmbeddingComposite.parameters	
LazyFixedEmbeddingComposite.properties	
LazyFixedEmbeddingComposite.nodelist	Nodes available to the composed sampler.
LazyFixedEmbeddingComposite.edgelist	Edges available to the composed sampler.
LazyFixedEmbeddingComposite.adjacency	Adjacency structure for the composed sampler.
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.

dwave.system.composites.LazyFixedEmbeddingComposite.parameters

`LazyFixedEmbeddingComposite.`**`parameters = None`**

dwave.system.composites.LazyFixedEmbeddingComposite.properties

`LazyFixedEmbeddingComposite.`**`properties = None`**

dwave.system.composites.LazyFixedEmbeddingComposite.nodelist

`LazyFixedEmbeddingComposite.`**`nodelist`**

Nodes available to the composed sampler.

Type list

dwave.system.composites.LazyFixedEmbeddingComposite.edgelist

`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

`LazyFixedEmbeddingComposite.sample`(bqm, . . .)	Sample the binary quadratic model.
`LazyFixedEmbeddingComposite.sample_ising`(h, . . .)	Sample from an Ising model using the implemented sample method.
`LazyFixedEmbeddingComposite.sample_qubo`(Q, . . .)	Sample from a QUBO using the implemented sample method.

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`

dwave.system.composites.LazyFixedEmbeddingComposite.sample_ising

LazyFixedEmbeddingComposite.**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.composites.LazyFixedEmbeddingComposite.sample_qubo

`LazyFixedEmbeddingComposite.`**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()*

TilingComposite

class TilingComposite(*sampler*, *sub_m*, *sub_n*, *t=4*)

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.

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import EmbeddingComposite
>>> from dwave.system.composites import TilingComposite
...
>>> sampler = EmbeddingComposite(TilingComposite(DWaveSampler(), 1, 1, 4))
>>> Q = {(1, 1): -1, (1, 2): 2, (2, 1): 0, (2, 2): -1}
>>> response = sampler.sample_qubo(Q)
>>> response.first    # doctest: +SKIP
Sample(sample={1: 0, 2: 1}, energy=-1.0, num_occurrences=1, chain_break_
↪fraction=0.0)
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

Properties

`TilingComposite.properties`	Properties in the form of a dict.
`TilingComposite.parameters`	Parameters in the form of a dict.
`TilingComposite.children`	The single wrapped structured sampler.
`TilingComposite.child`	The child sampler.
`TilingComposite.nodelist`	List of active qubits for the structured solver.
`TilingComposite.edgelist`	List of active couplers for the D-Wave solver.
`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.
`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.

dwave.system.composites.TilingComposite.properties

`TilingComposite.`**`properties = None`**

Properties in the form of a dict.

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

`TilingComposite.sample`(bqm, **kwargs)	Sample from the specified binary quadratic model.

Continued on next page

Table 18 – continued from previous page

TilingComposite.sample_ising(h, J, **parameters)	Sample from an Ising model using the implemented sample method.
TilingComposite.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dwave.system.composites.TilingComposite.sample

TilingComposite.**sample**(*bqm*, ***kwargs*)

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).

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import EmbeddingComposite
>>> from dwave.system.composites import EmbeddingComposite, 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

TilingComposite.**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.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.

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import VirtualGraphComposite
>>> embedding = {'x': {1}, 'y': {5}, 'z': {0, 4}}
>>> DWaveSampler().properties['extended_j_range']   # doctest: +SKIP
[-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}
>>> response = sampler.sample_qubo(Q, num_reads=10) # doctest: +SKIP
>>> for sample in response.samples():     # doctest: +SKIP
...     print(sample)
...
{'y': 0, 'x': 1, 'z': 0}
{'y': 1, 'x': 0, 'z': 0}
{'y': 1, 'x': 0, 'z': 0}
{'y': 1, 'x': 1, 'z': 1}
{'y': 0, 'x': 1, 'z': 0}
{'y': 1, 'x': 0, 'z': 0}
{'y': 0, 'x': 1, 'z': 0}
{'y': 0, 'x': 1, 'z': 0}
{'y': 0, 'x': 0, 'z': 0}
{'y': 1, 'x': 0, 'z': 0}
```

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.

Continued on next page

Table 19 – continued from previous page

`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

`VirtualGraphComposite.sample`(bqm[, ...])	Sample from the given Ising model.
`VirtualGraphComposite.sample_ising`(h, J, ...)	Sample from an Ising model using the implemented sample method.
`VirtualGraphComposite.sample_qubo`(Q, ...)	Sample from a QUBO using the implemented sample method.

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.

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import VirtualGraphComposite
>>> embedding = {'x': {0, 4, 1, 5}, 'y': {2, 6, 3, 7}}
>>> DWaveSampler().properties['extended_j_range']   # doctest: +SKIP
[-2.0, 1.0]
>>> sampler = VirtualGraphComposite(DWaveSampler(), embedding, chain_strength=1)
↪# doctest: +SKIP
>>> h = {}
>>> J = {('x', 'y'): 1}
>>> response = sampler.sample_ising(h, J, num_reads=10) # doctest: +SKIP
>>> for sample in response.samples():    # doctest: +SKIP
...     print(sample)
...
```

(continues on next page)

(continued from previous page)

```
{'y': -1, 'x': 1}
{'y': 1, 'x': -1}
{'y': -1, 'x': 1}
{'y': -1, 'x': 1}
{'y': -1, 'x': 1}
{'y': 1, 'x': -1}
{'y': 1, 'x': -1}
{'y': 1, 'x': -1}
{'y': -1, 'x': 1}
{'y': 1, 'x': -1}
```

See Ocean Glossary for explanations of technical terms in descriptions of Ocean tools.

dwave.system.composites.VirtualGraphComposite.sample_ising

`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 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.VirtualGraphComposite.sample_qubo

`VirtualGraphComposite.`**`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

ReverseBatchStatesComposite.child	The child sampler.
ReverseBatchStatesComposite.children	List of child samplers that that are used by this composite.
ReverseBatchStatesComposite.properties	A dict containing any additional information about the sampler.
ReverseBatchStatesComposite.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.

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 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 relevent to each parameter.

Type dict

Methods

ReverseBatchStatesComposite.sample(bqm, …)	Sample the binary quadratic model using reverse annealing from multiple initial states.
ReverseBatchStatesComposite.sample_ising(h, …)	Sample from an Ising model using the implemented sample method.
ReverseBatchStatesComposite.sample_qubo(Q, …)	Sample from a QUBO using the implemented sample method.

dwave.system.composites.ReverseBatchStatesComposite.sample

ReverseBatchStatesComposite.**sample**(*bqm*, ***parameters*)

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.

dwave.system.composites.ReverseBatchStatesComposite.sample_ising

ReverseBatchStatesComposite.**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.composites.ReverseBatchStatesComposite.sample_qubo

`ReverseBatchStatesComposite.`**`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()`*

ReverseAdvanceComposite

`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

`ReverseAdvanceComposite.child`	The child sampler.
`ReverseAdvanceComposite.children`	List of child samplers that that are used by this composite.
`ReverseAdvanceComposite.properties`	A dict containing any additional information about the sampler.
`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.

dwave.system.composites.ReverseAdvanceComposite.child

`ReverseAdvanceComposite.`**`child`**

The child sampler. First sampler in `Composite.children`.

Type `Sampler`

dwave.system.composites.ReverseAdvanceComposite.children

ReverseAdvanceComposite.**children**

List of child samplers that that are used by this composite.

Type list[Sampler]

dwave.system.composites.ReverseAdvanceComposite.properties

ReverseAdvanceComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

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

ReverseAdvanceComposite.sample(bqm[, ...])	Sample the binary quadratic model using reverse annealing along a given set of anneal schedules.
ReverseAdvanceComposite.sample_ising(h, J, ...)	Sample from an Ising model using the implemented sample method.
ReverseAdvanceComposite.sample_qubo(Q, ...)	Sample from a QUBO using the implemented sample method.

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()`

dwave.system.composites.ReverseAdvanceComposite.sample_qubo

ReverseAdvanceComposite.**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()`

1.2.3 Embedding

Provides functions that map binary quadratic models and samples between a source graph and a target graph.

The D-Wave 2000Q is Chimera-structured. 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.

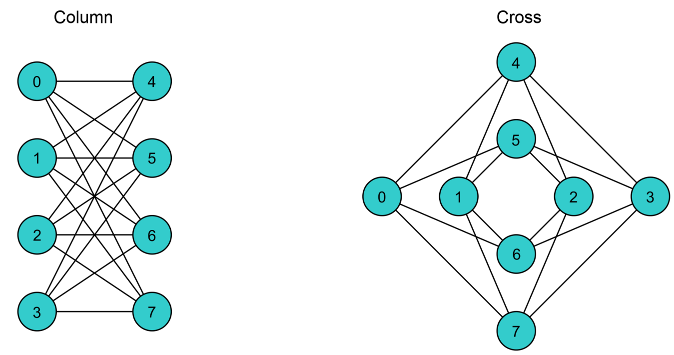

Fig. 1: Chimera unit cell.

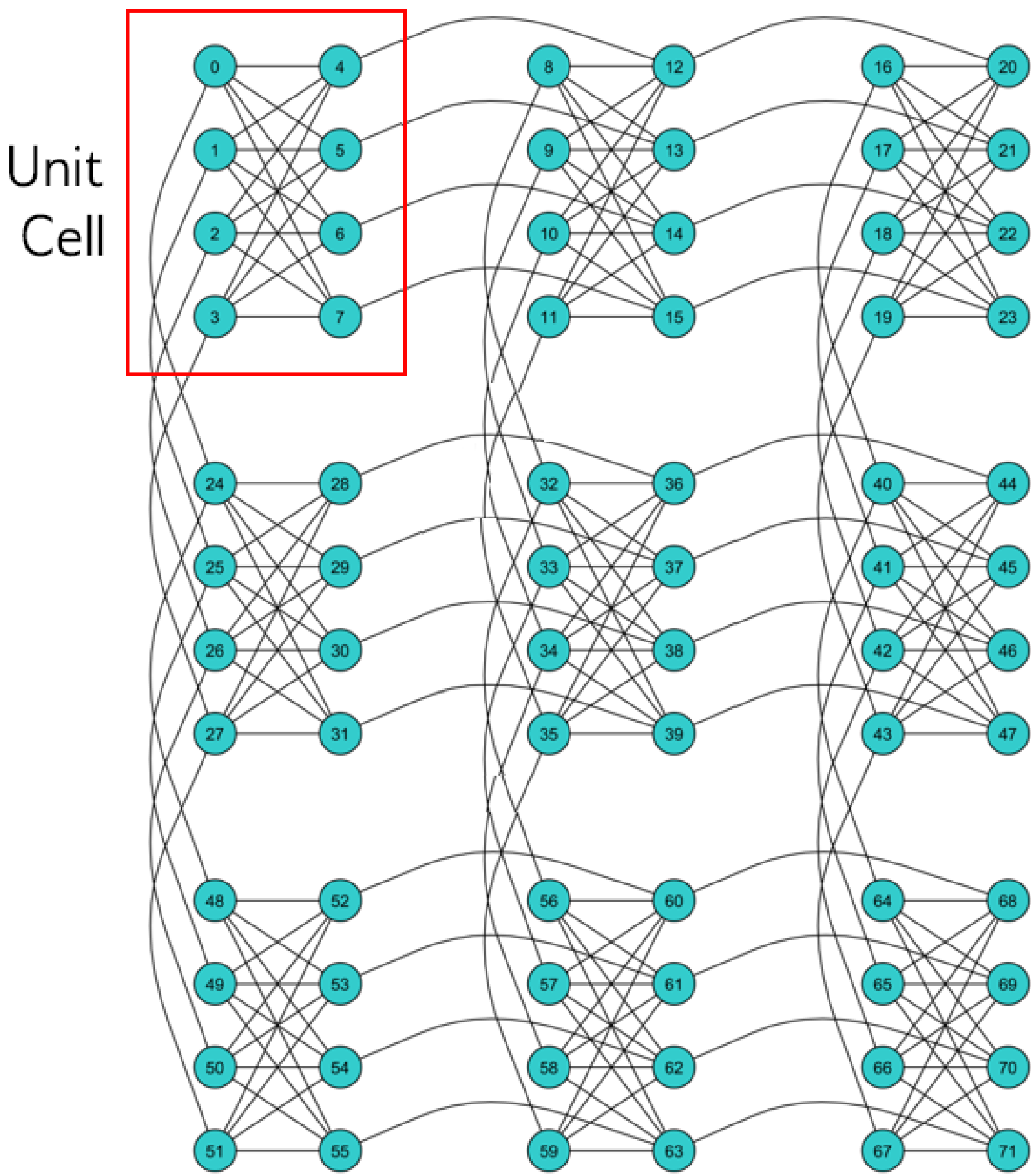

Fig. 2: A 3x3 Chimera graph, denoted C3. Qubits are arranged in 9 unit cells.

Example

As described in more detail in the Boolean AND Gate example, a sampler may not natively support a given problem graph. For example, the D-Wave system does not natively support K_3 graphs. The Boolean AND gate ($x_3 \Leftrightarrow x_1 \wedge x_2$ where x_3 is the AND gate's output and x_1, x_2 the inputs) may be represented as penalty model

$$x_1 x_2 - 2(x_1 + x_2)x_3 + 3x_3.$$

This penalty model can in turn be represented as the QUBO,

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

which is a fully connected K_3 graph.

Sampling this problem on a D-Wave system, therefore, requires minor-embedding. Embedding in this case is accomplished by an edge contraction operation on the target graph: two nodes (qubits) are chained to represent a single node.

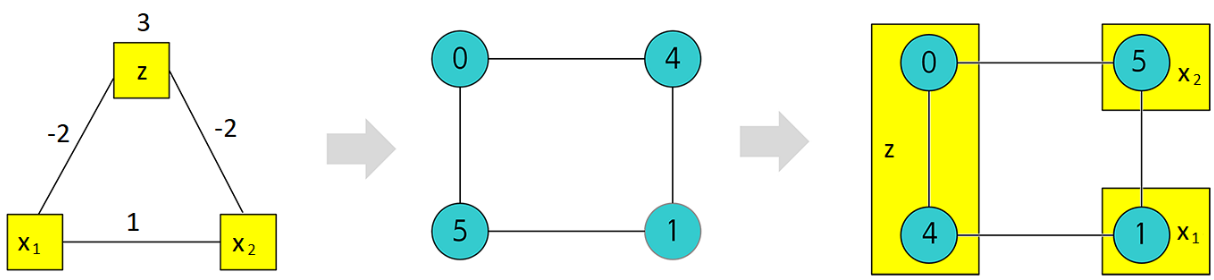

Fig. 3: Embedding an AND gate represented by a K_3 graph onto the D-Wave system's graph. The leftmost graph is the source graph, which is the QUBO representing the AND gate; the middle one is the target graph, representing the D-Wave system; and in the rightmost graph, qubits 0 and 4 of the D-Wave system's graph are chained to represent the single node z of the source graph.

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.

minorminer.find_embedding(S, T, **params)	Heuristically attempt to find a minor-embedding of a graph representing an Ising/QUBO into a target graph.

minorminer.find_embedding

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)
```

(continues on next page)

(continued from previous page)

```
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

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
```

(continues on next page)

(continued from previous page)

```
    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. 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 trhough 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.

chimera.find_clique_embedding(k, m[, n, t, ...])	Find an embedding for a clique in a Chimera graph.
chimera.find_biclique_embedding(a, b, m[, ...])	Find an embedding for a biclique in a Chimera graph.
chimera.find_grid_embedding(dim, m[, n, t])	Find an embedding for a grid in a Chimera graph.

dwave.embedding.chimera.find_clique_embedding

find_clique_embedding(*k*, *m*, *n=None*, *t=None*, *target_edges=None*)

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.

```
>>> from dwave.embedding.chimera import find_clique_embedding
...
>>> embedding = find_clique_embedding(4, 1, 1)
>>> embedding  # 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)
>>> embedding  # 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.

```
>>> 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.

```
>>> 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.

pegasus.find_clique_embedding(k[, m, . . .])	Find an embedding for a clique in a Pegasus graph.

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 Pegaus graph.

```
>>> 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

embed_bqm(source_bqm, embedding, . . . [, . . .])	Embed a binary quadratic model onto a target graph.
embed_ising(source_h, source_J, embedding, . . .)	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, . . . [, . . .])	Unembed a samples set.

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.

```
>>> 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.

```
>>> import networkx as nx
...
>>> target = nx.cycle_graph(4)
>>> # 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 * *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.

```
>>> 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()*

dwave.embedding.unembed_sampleset

unembed_sampleset (*target_sampleset, embedding, source_bqm, chain_break_method=None, chain_break_fraction=False, return_embedding=False*)

Unembed a samples set.

Given samples from a target binary quadratic model (BQM), construct a sample set for a source BQM by unembedding.

Parameters

- **target_sampleset** (dimod.SampleSet) – Sample set from the target BQM.
- **embedding** (*dict*) – Mapping from source graph to target graph as a dict of form {s: {t, . . . }, . . . }, where s is a source variable and t is a target variable.
- **source_bqm** (dimod.BinaryQuadraticModel) – Source BQM.
- **chain_break_method** (*function, optional*) – Method used to resolve chain breaks. See `dwave.embedding.chain_breaks`.

- **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.

```
>>> # 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)
>>> samples.record.sample   # doctest: +SKIP
array([[-1, -1, -1],
       [ 1,  1,  1]], dtype=int8)
```

Diagnostics

chain_break_frequency(samples_like, embedding)	Determine the frequency of chain breaks in the given samples.
diagnose_embedding(emb, source, target)	Diagnose a minor embedding.
is_valid_embedding(emb, source, target)	A simple (bool) diagnostic for minor embeddings.
verify_embedding(emb, source, target[, ...])	A simple (exception-raising) diagnostic for minor embeddings.

dwave.embedding.chain_break_frequency

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.

```
>>> 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.

```
>>> 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 represention 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.

```
>>> 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)
>>> idx
array([0])
```

dwave.embedding.chain_breaks.majority_vote

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 represention 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`: Indicies 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.

```
>>> 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 = dwave.embedding.majority_vote(samples, chains)
>>> unembedded
array([[1, 0],
       [1, 1]], dtype=int8)
>>> idx
array([0, 1])
```

dwave.embedding.chain_breaks.weighted_random

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 represention 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`: Indicies 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.

```
>>> 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.

```
>>> 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 represention 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`: Indicies 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(tnode, 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, snode1)	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.

1.2.4 Utilities

Utility functions.

common_working_graph(graph0, graph1)	Creates a graph using the common nodes and edges of two given graphs.

dwave.system.utilities.common_working_graph

common_working_graph(*graph0*, *graph1*)

Creates a graph using the common nodes and edges of two given graphs.

This function finds the edges and nodes with common labels. Note that this not the same as finding the greatest common subgraph with isomorphisms.

Parameters

- **graph0** – (dict[dict]/Graph) A NetworkX graph or a dictionary of dictionaries adjacency representation.
- **graph1** – (dict[dict]/Graph) 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.

```
>>> import dwave_networkx as dnx
>>> from dwave.system import DWaveSampler, common_working_graph
...
>>> sampler = DWaveSampler(solver={'qpu': True}) # doctest: +SKIP
>>> C4 = dnx.chimera_graph(4)  # a 4x4 lattice of Chimera tiles
>>> c4_working_graph = common_working_graph(C4, sampler.adjacency)   # doctest:␣
↪+SKIP
```

1.3 Installation

Installation from PyPI:

```
pip install dwave-system
```

Installation from PyPI with drivers:

Note: Prior to v0.3.0, running `pip install dwave-system` installed a driver dependency called `dwave-drivers` (previously also called `dwave-system-tuning`). This dependency has a restricted license and has been made optional as of v0.3.0, but is highly recommended. To view the license details:

```
from dwave.drivers import __license__
print(__license__)
```

To install with optional dependencies:

```
pip install dwave-system[drivers] --extra-index-url https://pypi.dwavesys.com/simple
```

Installation from source:

```
pip install -r requirements.txt
python setup.py install
```

Note that installing from source installs `dwave-drivers`. To uninstall the proprietary components:

```
pip uninstall dwave-drivers
```

1.4 License

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

Python Module Index

d

Index

Symbols

A

C

D

E

F

I

L

M

N

P

R

S

T

U

V

W