dwavebinarycsp

Release 0.1.1

D-Wave Systems Inc

Feb 25, 2020

Contents

Library to construct a binary quadratic model from a constraint satisfaction problem with small constraints over binary variables.

Below is an example usage:

```
import dwavebinarycsp
import dimod

csp = dwavebinarycsp.factories.random_2in4sat(8, 4)  # 8 variables, 4 clauses

bqm = dwavebinarycsp.stitch(csp)

resp = dimod.ExactSolver().sample(bqm)

for sample, energy in resp.data(['sample', 'energy']):
    print(sample, csp.check(sample), energy)
```

CHAPTER 1

Documentation

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

1.1 Introduction

dwavebinarycsp is a library to construct a binary quadratic *model* from a constraint satisfaction problem (CSP) with small constraints over binary variables (represented as either binary values {0, 1} or spin values {-1, 1}).

1.1.1 Constraint Satisfaction Problems

Constraint satisfaction problems require that all a problem's variables be assigned values, out of a finite domain, that result in the satisfying of all constraints.

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

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

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

1.1.2 Binary Constraint Satisfaction Problems

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

Fig. 1: Coloring a map of Canada with four colors.

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

Table 1: Boolean AND Operation

x_1, x_2	y
$0, 0$	0
$0, 1$	0
$1, 0$	0
$1, 1$	1

1.1.3 Binary Quadratic Models

D-Wave systems solve problems that can be mapped onto an Ising model or a quadratic unconstrained binary optimization (QUBO) problem. These can be seen as subsets of a binary quadratic model (BQM).

For example, the Boolean operations of logical gates represented as CSPs can also be represented by a particular type of BQM called a penalty model: penalty functions penalize invalid states; that is, invalid sets of input and output values representing gates have higher penalty values than valid sets.

For example, the AND gate's constraint $y = x_1 x_2$ can be represented as penalty function

$$x_1 x_2 - 2(x_1 + x_2)y + 3y,$$

which penalizes invalid configurations while no penalty is applied to valid configurations.

In Table *Boolean AND Operation as a Penalty*, columns Out_{valid} and $Out_{invalid}$ represent, together with the in column for each row, valid and invalid configurations of an AND gate, with columns P_{valid} and $P_{invalid}$ the respective penalty values.

Table 2: Boolean AND Operation as a Penalty.

in	$\mathrm{out}_{\mathrm{valid}}$	$\mathrm{out}_{\mathrm{invalid}}$	$\mathrm{P}_{\mathrm{valid}}$	$\mathrm{P}_{\mathrm{invalid}}$
$0,0$	0	1	0	3
$0,1$	0	1	0	1
$1,0$	0	1	0	1
$1,1$	1	0	0	1

For example, the state $in = 0, 0; out_{valid} = 0$ of the first row is represented by the penalty function with $x_1 = x_2 = 0$ and $z = 0 = x_1 \wedge x_2$. For this valid configuration, the value of P_{valid} is

$$\begin{aligned} x_1x_2 - 2(x_1 + x_2)z + 3z &= 0 \times 0 - 2 \times (0 + 0) \times 0 + 3 \times 0 \\ &= 0, \end{aligned}$$

not penalizing the valid configuration. In contrast, the state $in = 0, 0; out_{invalid} = 1$ of the same row is represented by the penalty function with $x_1 = x_2 = 0$ and $z = 1 \neq x_1 \wedge x_2$. For this invalid configuration, the value of $P_{invalid}$ is

$$\begin{aligned} x_1x_2 - 2(x_1 + x_2)z + 3z &= 0 \times 0 - 2 \times (0 + 0) \times 1 + 3 \times 1 \\ &= 3, \end{aligned}$$

adding a penalty of 3 to the incorrect configuration.

The samples representing low energy states returned from a sampler such as the D-Wave system correspond to valid configurations, and therefore correctly represent the AND gate.

1.1.4 Example: Solving a Map-Coloring CSP

This example finds a solution to the map-coloring problem for a map of Canada using four colors. Canada's 13 provinces are denoted by postal codes:

Table 3: Canadian Provinces' Postal Codes

Code	Province	Code	Province
AB	Alberta	BC	British Columbia
MB	Manitoba	NB	New Brunswick
NL	Newfoundland and Labrador	NS	Nova Scotia
NT	Northwest Territories	NU	Nunavut
ON	Ontario	PE	Prince Edward Island
QC	Quebec	SK	Saskatchewan
YT	Yukon		

The workflow for solution is as follows:

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

The following sample code creates a graph of the map with provinces as nodes and shared borders between provinces as edges (e.g., "('AB', 'BC')" is an edge representing the shared border between British Columbia and Alberta). It creates a binary constraint satisfaction problem based on two types of constraints:

- `csp.add_constraint(one_color_configurations, variables)` represents the constraint that each node (province) select a single color, as represented by valid configurations `one_color_configurations = {(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)}`
- `csp.add_constraint(not_both_1, variables)` represents the constraint that two nodes (provinces) with a shared edge (border) not both select the same color.

```
import dwavebinarycsp
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import EmbeddingComposite
import networkx as nx
import matplotlib.pyplot as plt

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

# Function for the constraint that two nodes with a shared edge not both select one␣
↪color
def not_both_1(v, u):
    return not (v and u)

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

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

# Create a binary constraint satisfaction problem
csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)

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

# Add constraint that each pair of nodes with a shared edge not both select one color
```

(continues on next page)

(continued from previous page)

```
for neighbor in neighbors:
    v, u = neighbor
          for i in range(colors):
        variables = [v+str(i), u+str(i)]
                    csp.add_constraint(not_both_1, variables)

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

# Set up a solver using the local system's default D-Wave Cloud Client configuration␣
↪file
# and sample 50 times
sampler = EmbeddingComposite(DWaveSampler())         # doctest: +SKIP
response = sampler.sample(bqm, num_reads=50)         # doctest: +SKIP

# Plot the lowest-energy sample if it meets the constraints
sample = next(response.samples())      # doctest: +SKIP
if not csp.check(sample):              # doctest: +SKIP
    print("Failed to color map")
else:
    plot_map(sample)
```

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

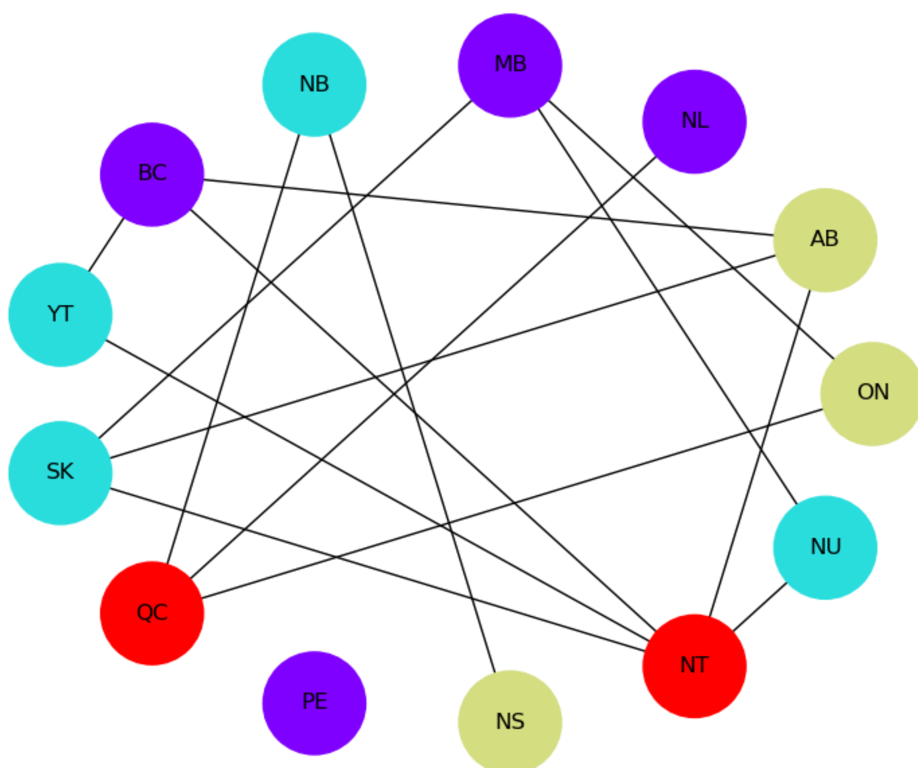

Fig. 2: Solution for a map of Canada with four colors. The graph comprises 13 nodes representing provinces connected by edges representing shared borders. No two nodes connected by an edge share a color.

1.1.5 Terminology

model A collection of variables with associated linear and quadratic biases.

sampler A process that samples from low energy states of a problem's objective function. A binary quadratic model (BQM) sampler samples from low energy states in models such as those defined by an Ising equation or a Quadratic Unconstrained Binary Optimization (QUBO) problem and returns an iterable of samples, in order of increasing energy. A dimod sampler provides 'sample_qubo' and 'sample_ising' methods as well as the generic BQM sampler method.

1.2 Reference Documentation

Release 0.1.1

Date Feb 25, 2020

1.2.1 Defining Constraint Satisfaction Problems

Constraint satisfaction problems require that all a problem's variables be assigned values, out of a finite domain, that result in the satisfying of all constraints. The `ConstraintSatisfactionProblem` class aggregates all constraints and variables defined for a problem and provides functionality to assist in problem solution, such as verifying whether a candidate solution satisfies the constraints.

Class

class ConstraintSatisfactionProblem(*vartype*)

A constraint satisfaction problem.

Parameters **vartype** (`Vartype`/str/set) – Variable type for the binary quadratic model. Supported values are:

- `SPIN`, `'SPIN'`, `{-1, 1}`
- `BINARY`, `'BINARY'`, `{0, 1}`

constraints

Constraints that together constitute the constraint satisfaction problem. Valid solutions satisfy all of the constraints.

Type list[*`Constraint`*]

variables

Variables of the constraint satisfaction problem as a dict, where keys are the variables and values a list of all of constraints associated with the variable.

Type dict[variable, list[*`Constraint`*]]

vartype

Enumeration of valid variable types. Supported values are `SPIN` or `BINARY`. If *vartype* is SPIN, variables can be assigned -1 or 1; if BINARY, variables can be assigned 0 or 1.

Type `dimod.Vartype`

Example

This example creates a binary-valued constraint satisfaction problem, adds two constraints, $a = b$ and $b \neq c$, and tests $a, b, c = 1, 1, 0$.

```
>>> import dwavebinarycsp
>>> import operator
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem('BINARY')
>>> csp.add_constraint(operator.eq, ['a', 'b'])
>>> csp.add_constraint(operator.ne, ['b', 'c'])
>>> csp.check({'a': 1, 'b': 1, 'c': 0})
True
```

Methods

Adding variables and constraints

ConstraintSatisfactionProblem.add_constraint(…)	Add a constraint.
ConstraintSatisfactionProblem.add_variable(v)	Add a variable.

dwavebinarycsp.ConstraintSatisfactionProblem.add_constraint

`ConstraintSatisfactionProblem.`**`add_constraint`**(*constraint*, *variables=()*)

Add a constraint.

Parameters

- **constraint** (function/iterable/*Constraint*) – Constraint definition in one of the supported formats:
  1. Function, with input arguments matching the order and *vartype* type of the *variables* argument, that evaluates True when the constraint is satisfied.
  2. List explicitly specifying each allowed configuration as a tuple.
  3. *Constraint* object built either explicitly or by `dwavebinarycsp.factories`.
- **variables** (*iterable*) – Variables associated with the constraint. Not required when *constraint* is a *Constraint* object.

Examples

This example defines a function that evaluates True when the constraint is satisfied. The function's input arguments match the order and type of the *variables* argument.

```
>>> import dwavebinarycsp
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> def all_equal(a, b, c):  # works for both dwavebinarycsp.BINARY and␣
↪dwavebinarycsp.SPIN
...     return (a == b) and (b == c)
>>> csp.add_constraint(all_equal, ['a', 'b', 'c'])
>>> csp.check({'a': 0, 'b': 0, 'c': 0})
```

(continues on next page)

(continued from previous page)

```
True
>>> csp.check({'a': 0, 'b': 0, 'c': 1})
False
```

This example explicitly lists allowed configurations.

```
>>> import dwavebinarycsp
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.SPIN)
>>> eq_configurations = {(-1, -1), (1, 1)}
>>> csp.add_constraint(eq_configurations, ['v0', 'v1'])
>>> csp.check({'v0': -1, 'v1': +1})
False
>>> csp.check({'v0': -1, 'v1': -1})
True
```

This example uses a *Constraint* object built by `dwavebinarycsp.factories`.

```
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.and_gate(['a', 'b', 'c']))  # add an AND gate
>>> csp.add_constraint(gates.xor_gate(['a', 'c', 'd']))  # add an XOR gate
>>> csp.check({'a': 1, 'b': 0, 'c': 0, 'd': 1})
True
```

dwavebinarycsp.ConstraintSatisfactionProblem.add_variable

`ConstraintSatisfactionProblem.`**`add_variable`**`(v)`

Add a variable.

Parameters **v** (*variable*) – Variable in the constraint satisfaction problem. May be of any type that can be a dict key.

Examples

This example adds two variables, one of which is already used in a constraint of the constraint satisfaction problem.

```
>>> import dwavebinarycsp
>>> import operator
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.SPIN)
>>> csp.add_constraint(operator.eq, ['a', 'b'])
>>> csp.add_variable('a')  # does nothing, already added as part of the constraint
>>> csp.add_variable('c')
>>> csp.check({'a': -1, 'b': -1, 'c': 1})
True
>>> csp.check({'a': -1, 'b': -1, 'c': -1})
True
```

Satisfiability

ConstraintSatisfactionProblem.check(solution)	Check that a solution satisfies all of the constraints.

dwavebinarycsp.ConstraintSatisfactionProblem.check

`ConstraintSatisfactionProblem.`**`check`**(*solution*)

Check that a solution satisfies all of the constraints.

Parameters **solution** (*container*) – An assignment of values for the variables in the constraint satisfaction problem.

Returns True if the solution satisfies all of the constraints; False otherwise.

Return type bool

Examples

This example creates a binary-valued constraint satisfaction problem, adds two logic gates implementing Boolean constraints, $c = a \wedge b$ and $d = a \oplus c$, and verifies that the combined problem is satisfied for a given assignment.

```
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.and_gate(['a', 'b', 'c']))  # add an AND gate
>>> csp.add_constraint(gates.xor_gate(['a', 'c', 'd']))  # add an XOR gate
>>> csp.check({'a': 1, 'b': 0, 'c': 0, 'd': 1})
True
```

Transformations

ConstraintSatisfactionProblem.fix_variable(v, ...)	Fix the value of a variable and remove it from the constraint satisfaction problem.

dwavebinarycsp.ConstraintSatisfactionProblem.fix_variable

`ConstraintSatisfactionProblem.`**`fix_variable`**(*v*, *value*)

Fix the value of a variable and remove it from the constraint satisfaction problem.

Parameters

- **v** (*variable*) – Variable to be fixed in the constraint satisfaction problem.
- **value** (*int*) – Value assigned to the variable. Values must match the *vartype* of the constraint satisfaction problem.

Examples

This example creates a spin-valued constraint satisfaction problem, adds two constraints, $a = b$ and $b \neq c$, and fixes variable b to +1.

```
>>> import dwavebinarycsp
>>> import operator
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.SPIN)
>>> csp.add_constraint(operator.eq, ['a', 'b'])
>>> csp.add_constraint(operator.ne, ['b', 'c'])
>>> csp.check({'a': +1, 'b': +1, 'c': -1})
True
>>> csp.check({'a': -1, 'b': -1, 'c': +1})
True
>>> csp.fix_variable('b', +1)
>>> csp.check({'a': +1, 'b': +1, 'c': -1})  # 'b' is ignored
True
>>> csp.check({'a': -1, 'b': -1, 'c': +1})
False
>>> csp.check({'a': +1, 'c': -1})
True
>>> csp.check({'a': -1, 'c': +1})
False
```

1.2.2 Converting to a Binary Quadratic Model

Constraint satisfaction problems can be converted to binary quadratic models to be solved on samplers such as the D-Wave system.

Compilers

Compilers accept a constraint satisfaction problem and return a `dimod.BinaryQuadraticModel`.

stitch(csp[, min_classical_gap, max_graph_size])	Build a binary quadratic model with minimal energy levels at solutions to the specified constraint satisfaction problem.

dwavebinarycsp.stitch

stitch (*csp*, *min_classical_gap=2.0*, *max_graph_size=8*)

Build a binary quadratic model with minimal energy levels at solutions to the specified constraint satisfaction problem.

Parameters

- **csp** (*ConstraintSatisfactionProblem*) – Constraint satisfaction problem.
- **min_classical_gap** (*float, optional, default=2.0*) – Minimum energy gap from ground. Each constraint violated by the solution increases the energy level of the binary quadratic model by at least this much relative to ground energy.
- **max_graph_size** (*int, optional, default=8*) – Maximum number of variables in the binary quadratic model that can be used to represent a single constraint.

Returns `BinaryQuadraticModel`

Notes

For a *min_classical_gap* > 2 or constraints with more than two variables, requires access to factories from the penaltymodel ecosystem to construct the binary quadratic model.

Examples

This example creates a binary-valued constraint satisfaction problem with two constraints, $a = b$ and $b \neq c$, and builds a binary quadratic model with a minimum energy level of -2 such that each constraint violation by a solution adds the default minimum energy gap.

```
>>> import dwavebinarycsp
>>> import operator
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(operator.eq, ['a', 'b'])  # a == b
>>> csp.add_constraint(operator.ne, ['b', 'c'])  # b != c
>>> bqm = dwavebinarycsp.stitch(csp)
>>> bqm.energy({'a': 0, 'b': 0, 'c': 1})  # satisfies csp
-2.0
>>> bqm.energy({'a': 0, 'b': 0, 'c': 0})  # violates one constraint
0.0
>>> bqm.energy({'a': 1, 'b': 0, 'c': 0}) # violates two constraints
2.0
```

This example creates a binary-valued constraint satisfaction problem with two constraints, $a = b$ and $b \neq c$, and builds a binary quadratic model with a minimum energy gap of 4. Note that in this case the conversion to binary quadratic model adds two ancillary variables that must be minimized over when solving.

```
>>> import dwavebinarycsp
>>> import operator
>>> import itertools
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(operator.eq, ['a', 'b'])  # a == b
>>> csp.add_constraint(operator.ne, ['b', 'c'])  # b != c
>>> bqm = dwavebinarycsp.stitch(csp, min_classical_gap=4.0)
>>> list(bqm)   # # doctest: +SKIP
['a', 'aux1', 'aux0', 'b', 'c']
>>> min([bqm.energy({'a': 0, 'b': 0, 'c': 1, 'aux0': aux0, 'aux1': aux1}) for
... aux0, aux1 in list(itertools.product([0, 1], repeat=2))])  # satisfies csp
-6.0
>>> min([bqm.energy({'a': 0, 'b': 0, 'c': 0, 'aux0': aux0, 'aux1': aux1}) for
... aux0, aux1 in list(itertools.product([0, 1], repeat=2))])  # violates one␣
↪constraint
-2.0
>>> min([bqm.energy({'a': 1, 'b': 0, 'c': 0, 'aux0': aux0, 'aux1': aux1}) for
... aux0, aux1 in list(itertools.product([0, 1], repeat=2))])  # violates two␣
↪constraints
2.0
```

This example finds for the previous example the minimum graph size.

```
>>> import dwavebinarycsp
>>> import operator
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(operator.eq, ['a', 'b'])  # a == b
>>> csp.add_constraint(operator.ne, ['b', 'c'])  # b != c
```

(continues on next page)

(continued from previous page)

```
>>> for n in range(8, 1, -1):
...     try:
...         bqm = dwavebinarycsp.stitch(csp, min_classical_gap=4.0, max_graph_
↪size=n)
...     except dwavebinarycsp.exceptions.ImpossibleBQM:
...         print(n+1)
...
3
```

1.2.3 Other CSP Formats

DIMACS

The DIMACS format is used to encode boolean satisfiability problems in conjunctive normal form.

CNF

load_cnf(fp)	Load a constraint satisfaction problem from a .cnf file.

dwavebinarycsp.io.cnf.load_cnf

load_cnf (*fp*)

Load a constraint satisfaction problem from a .cnf file.

Parameters fp (*file, optional*) – *.write()*-supporting file object DIMACS CNF formatted file.

Returns *ConstraintSatisfactionProblem* a binary-valued SAT problem.

Examples

```
>>> import dwavebinarycsp as dbcsp
...
>>> with open('test.cnf', 'r') as fp: # doctest: +SKIP
...     csp = dbcsp.cnf.load_cnf(fp)
```

1.2.4 Reducing Constraints

Constraints can sometimes be reduced into several smaller constraints.

Functions

irreducible_components(constraint)	Determine the sets of variables that are irreducible.

dwavebinarycsp.irreducible_components

irreducible_components(*constraint*)

Determine the sets of variables that are irreducible.

Let V(C) denote the variables of constraint C. For a configuration x, let x|A denote the restriction of the configuration to the variables of A. Constraint C is reducible if there is a partition of V(C) into nonempty subsets A and B, and two constraints C_A and C_B, with V(C_A) = A and C_B V(C_B) = B, such that a configuration x is feasible in C if and only if x|A is feasible in C_A and x|B is feasible in C_B. A constraint is irreducible if it is not reducible.

Parameters **constraint** (*Constraint*) – Constraint to attempt to reduce.

Returns List of tuples in which each tuple is a set of variables that is irreducible.

Return type list[tuple]

Examples

This example reduces a constraint, created by specifying its valid configurations, to two constraints. The original constraint, that valid configurations for a,b,c are 0,0,1 and 1,1,1, can be represented by two reduced constraints, for example, (c=1) & (a=b). For comparison, an attempt to reduce a constraint representing an AND gate fails to find a valid reduction.

```
>>> import dwavebinarycsp
>>> const = dwavebinarycsp.Constraint.from_configurations([(0, 0, 1), (1, 1, 1)],
...                                                      ['a', 'b', 'c'],␣
↪dwavebinarycsp.BINARY)
>>> dwavebinarycsp.irreducible_components(const)
[('c',), ('a', 'b')]
>>> const_and = dwavebinarycsp.Constraint.from_configurations([(0, 0, 0), (0, 1,␣
↪0), (1, 0, 0), (1, 1, 1)],
...                                                          ['a', 'b', 'c'],␣
↪dwavebinarycsp.BINARY)
>>> dwavebinarycsp.irreducible_components(const_and)
[('a', 'b', 'c')]
```

1.2.5 Defining Constraints

Solutions to a constraint satisfaction problem must satisfy certains conditions, the constraints of the problem, such as equality and inequality constraints. The `Constraint` class defines constraints and provides functionality to assist in constraint definition, such as verifying whether a candidate solution satisfies a constraint.

Class

class Constraint(*func*, *configurations*, *variables*, *vartype*, *name=None*)

A constraint.

variables

Variables associated with the constraint.

Type tuple

func

Function that returns True for configurations of variables that satisfy the constraint. Inputs to the function are ordered by *variables*.

Type function

configurations
Valid configurations of the variables. Each configuration is a tuple of variable assignments ordered by `variables`.

Type frozenset[tuple]

vartype
Variable type for the constraint. Accepted input values:

- `SPIN`, `'SPIN'`, `{-1, 1}`
- `BINARY`, `'BINARY'`, `{0, 1}`

Type `dimod.Vartype`

name
Name for the constraint. If not provided on construction, defaults to 'Constraint'.

Type str

Examples

This example defines a constraint, named "plus1", based on a function that is True for $(y1, y0) = (x1, x0) + 1$ on binary variables, and demonstrates some of the constraint's functionality.

```
>>> import dwavebinarycsp
>>> def plus_one(y1, y0, x1, x0):  # y=x++ for two bit binary numbers
...     return (y1, y0, x1, x0) in [(0, 1, 0, 0), (1, 0, 0, 1), (1, 1, 1, 0)]
...
>>> const = dwavebinarycsp.Constraint.from_func(
...                 plus_one,
...                 ['out1', 'out0', 'in1', 'in0'],
...                 dwavebinarycsp.BINARY,
...                 name='plus1')
>>> print(const.name)    # Check constraint defined as intended
plus1
>>> len(const)
4
>>> in0, in1, out0, out1 = 0, 0, 1, 0
>>> const.func(out1, out0, in1, in0)    # Order matches variables
True
```

This example defines a constraint based on specified valid configurations that represents an AND gate for spin variables, and demonstrates some of the constraint's functionality.

```
>>> import dwavebinarycsp
>>> const = dwavebinarycsp.Constraint.from_configurations(
...             [(-1, -1, -1), (-1, -1, 1), (-1, 1, -1), (1, 1, 1)],
...             ['y', 'x1', 'x2'],
...             dwavebinarycsp.SPIN)
>>> print(const.name)    # Check constraint defined as intended
Constraint
>>> isinstance(const, dwavebinarycsp.core.constraint.Constraint)
True
>>> (-1, 1, -1) in const.configurations    # Order matches variables: y,x1,x2
True
```

Methods

Construction

`Constraint.from_configurations`(…[, name])	Construct a constraint from valid configurations.
`Constraint.from_func`(func, variables, vartype)	Construct a constraint from a validation function.

dwavebinarycsp.Constraint.from_configurations

classmethod `Constraint.`**`from_configurations`**(*configurations*, *variables*, *vartype*, *name=None*)

Construct a constraint from valid configurations.

Parameters

- **configurations** (*iterable[tuple]*) – Valid configurations of the variables. Each configuration is a tuple of variable assignments ordered by `variables`.
- **variables** (*iterable*) – Iterable of variable labels.
- **vartype** (`Vartype`/str/set) – Variable type for the constraint. Accepted input values:
  - `SPIN`, `'SPIN'`, `{-1, 1}`
  - `BINARY`, `'BINARY'`, `{0, 1}`
- **name** (*string, optional, default='Constraint'*) – Name for the constraint.

Examples

This example creates a constraint that variables *a* and *b* are not equal.

```
>>> import dwavebinarycsp
>>> const = dwavebinarycsp.Constraint.from_configurations([(0, 1), (1, 0)],
...                   ['a', 'b'], dwavebinarycsp.BINARY)
>>> print(const.name)
Constraint
>>> (0, 0) in const.configurations   # Order matches variables: a,b
False
```

This example creates a constraint based on specified valid configurations that represents an OR gate for spin variables.

```
>>> import dwavebinarycsp
>>> const = dwavebinarycsp.Constraint.from_configurations(
...           [(-1, -1, -1), (1, -1, 1), (1, 1, -1), (1, 1, 1)],
...           ['y', 'x1', 'x2'],
...           dwavebinarycsp.SPIN, name='or_spin')
>>> print(const.name)
or_spin
>>> (1, 1, -1) in const.configurations   # Order matches variables: y,x1,x2
True
```

dwavebinarycsp.Constraint.from_func

classmethod Constraint.**from_func**(*func*, *variables*, *vartype*, *name=None*)
Construct a constraint from a validation function.

Parameters

- **func** (*function*) – Function that evaluates True when the variables satisfy the constraint.
- **variables** (*iterable*) – Iterable of variable labels.
- **vartype** (Vartype/str/set) – Variable type for the constraint. Accepted input values:
  - SPIN, 'SPIN', {-1, 1}
  - BINARY, 'BINARY', {0, 1}
- **name** (*string, optional, default='Constraint'*) – Name for the constraint.

Examples

This example creates a constraint that binary variables *a* and *b* are not equal.

```
>>> import dwavebinarycsp
>>> import operator
>>> const = dwavebinarycsp.Constraint.from_func(operator.ne, ['a', 'b'], 'BINARY')
>>> print(const.name)
Constraint
>>> (0, 1) in const.configurations
True
```

This example creates a constraint that $out = NOT(x)$ for spin variables.

```
>>> import dwavebinarycsp
>>> def not_(y, x):  # y=NOT(x) for spin variables
...     return (y == -x)
...
>>> const = dwavebinarycsp.Constraint.from_func(
...                 not_,
...                 ['out', 'in'],
...                 {1, -1},
...                 name='not_spin')
>>> print(const.name)
not_spin
>>> (1, -1) in const.configurations
True
```

Satisfiability

Constraint.check(solution)	Check that a solution satisfies the constraint.

dwavebinarycsp.Constraint.check

Constraint.**check**(*solution*)
Check that a solution satisfies the constraint.

Parameters **solution** (*container*) – An assignment for the variables in the constraint.

Returns True if the solution satisfies the constraint; otherwise False.

Return type bool

Examples

This example creates a constraint that $a \neq b$ on binary variables and tests it for two candidate solutions, with additional unconstrained variable c.

```
>>> import dwavebinarycsp
>>> const = dwavebinarycsp.Constraint.from_configurations([(0, 1), (1, 0)],
...                 ['a', 'b'], dwavebinarycsp.BINARY)
>>> solution = {'a': 1, 'b': 1, 'c': 0}
>>> const.check(solution)
False
>>> solution = {'a': 1, 'b': 0, 'c': 0}
>>> const.check(solution)
True
```

Transformations

Constraint.fix_variable(v, value)	Fix the value of a variable and remove it from the constraint.
Constraint.flip_variable(v)	Flip a variable in the constraint.

dwavebinarycsp.Constraint.fix_variable

Constraint.**fix_variable**(*v*, *value*)

Fix the value of a variable and remove it from the constraint.

Parameters

- **v** (*variable*) – Variable in the constraint to be set to a constant value.
- **val** (*int*) – Value assigned to the variable. Values must match the Vartype of the constraint.

Examples

This example creates a constraint that $a \neq b$ on binary variables, fixes variable a to 0, and tests two candidate solutions.

```
>>> import dwavebinarycsp
>>> const = dwavebinarycsp.Constraint.from_func(operator.ne,
...                 ['a', 'b'], dwavebinarycsp.BINARY)
>>> const.fix_variable('a', 0)
>>> const.check({'b': 1})
True
>>> const.check({'b': 0})
False
```

dwavebinarycsp.Constraint.flip_variable

`Constraint.`**`flip_variable`**`(v)`

Flip a variable in the constraint.

Parameters **v** (*variable*) – Variable in the constraint to take the complementary value of its construction value.

Examples

This example creates a constraint that $a = b$ on binary variables and flips variable a.

```
>>> import dwavebinarycsp
>>> const = dwavebinarycsp.Constraint.from_func(operator.eq,
...             ['a', 'b'], dwavebinarycsp.BINARY)
>>> const.check({'a': 0, 'b': 0})
True
>>> const.flip_variable('a')
>>> const.check({'a': 1, 'b': 0})
True
>>> const.check({'a': 0, 'b': 0})
False
```

Copies and projections

Constraint.copy()	Create a copy.
Constraint.projection(variables)	Create a new constraint that is the projection onto a subset of the variables.

dwavebinarycsp.Constraint.copy

`Constraint.`**`copy`**`()`

Create a copy.

Examples

This example copies constraint $a \neq b$ and tests a solution on the copied constraint.

```
>>> import dwavebinarycsp
>>> import operator
>>> const = dwavebinarycsp.Constraint.from_func(operator.ne,
...             ['a', 'b'], 'BINARY')
>>> const2 = const.copy()
>>> const2 is const
False
>>> const2.check({'a': 1, 'b': 1})
False
```

dwavebinarycsp.Constraint.projection

`Constraint.projection(variables)`
Create a new constraint that is the projection onto a subset of the variables.

Parameters **variables** (*iterable*) – Subset of the constraint's variables.

Returns A new constraint over a subset of the variables.

Return type *Constraint*

Examples

```
>>> import dwavebinarycsp
...
>>> const = dwavebinarycsp.Constraint.from_configurations([(0, 0), (0, 1)],
...                                                        ['a', 'b'],
...                                                        dwavebinarycsp.BINARY)
>>> proj = const.projection(['a'])
>>> proj.variables
['a']
>>> proj.configurations
{(0,)}
```

1.2.6 Factories

dwavebinarycsp currently provides factories for constraints representing Boolean gates and satisfiability problems and CSPs for circuits and satisfiability problems.

Constraints

Gates

gates.and_gate(variables[, vartype, name])	AND gate.
gates.or_gate(variables[, vartype, name])	OR gate.
gates.xor_gate(variables[, vartype, name])	XOR gate.
gates.halfadder_gate(variables[, vartype, name])	Half adder.
gates.fulladder_gate(variables[, vartype, name])	Full adder.

dwavebinarycsp.factories.constraint.gates.and_gate

and_gate(*variables*, *vartype=<Vartype.BINARY: frozenset({0, 1})>*, *name='AND'*)
AND gate.

Parameters

- **variables** (*list*) – Variable labels for the and gate as *[in1, in2, out]*, where *in1, in2* are inputs and *out* the gate's output.
- **vartype** (*Vartype, optional, default='BINARY'*) – Variable type. Accepted

input values:

- Vartype.SPIN, 'SPIN', {-1, 1}
- Vartype.BINARY, 'BINARY', {0, 1}

- **name** (*str, optional, default='AND'*) – Name for the constraint.

Returns Constraint that is satisfied when its variables are assigned values that match the valid states of an AND gate.

Return type Constraint(*Constraint*)

Examples

```
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.and_gate(['a', 'b', 'c'], name='AND1'))
>>> csp.check({'a': 1, 'b': 0, 'c': 0})
True
```

dwavebinarycsp.factories.constraint.gates.or_gate

or_gate(*variables, vartype=<Vartype.BINARY: frozenset({0, 1})>, name='OR'*)

OR gate.

Parameters

- **variables** (*list*) – Variable labels for the and gate as *[in1, in2, out]*, where *in1, in2* are inputs and *out* the gate's output.
- **vartype** (*Vartype, optional, default='BINARY'*) – Variable type. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **name** (*str, optional, default='OR'*) – Name for the constraint.

Returns Constraint that is satisfied when its variables are assigned values that match the valid states of an OR gate.

Return type Constraint(*Constraint*)

Examples

```
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.SPIN)
>>> csp.add_constraint(gates.or_gate(['x', 'y', 'z'], {-1,1}, name='OR1'))
>>> csp.check({'x': 1, 'y': -1, 'z': 1})
True
```

dwavebinarycsp.factories.constraint.gates.xor_gate

xor_gate(*variables*, *vartype=<Vartype.BINARY: frozenset({0, 1})>*, *name='XOR'*)

XOR gate.

Parameters

- **variables** (*list*) – Variable labels for the and gate as *[in1, in2, out]*, where *in1, in2* are inputs and *out* the gate's output.
- **vartype** (*Vartype, optional, default='BINARY'*) – Variable type. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **name** (*str, optional, default='XOR'*) – Name for the constraint.

Returns Constraint that is satisfied when its variables are assigned values that match the valid states of an XOR gate.

Return type Constraint(Constraint)

Examples

```
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.xor_gate(['x', 'y', 'z'], name='XOR1'))
>>> csp.check({'x': 1, 'y': 1, 'z': 1})
False
```

dwavebinarycsp.factories.constraint.gates.halfadder_gate

halfadder_gate(*variables*, *vartype=<Vartype.BINARY: frozenset({0, 1})>*, *name='HALF_ADDER'*)

Half adder.

Parameters

- **variables** (*list*) – Variable labels for the and gate as *[in1, in2, sum, carry]*, where *in1, in2* are inputs to be added and *sum* and 'carry' the resultant outputs.
- **vartype** (*Vartype, optional, default='BINARY'*) – Variable type. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **name** (*str, optional, default='HALF_ADDER'*) – Name for the constraint.

Returns Constraint that is satisfied when its variables are assigned values that match the valid states of a Boolean half adder.

Return type Constraint(Constraint)

Examples

```
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.halfadder_gate(['a', 'b', 'total', 'carry'], name=
↪'HA1'))
>>> csp.check({'a': 1, 'b': 1, 'total': 0, 'carry': 1})
True
```

dwavebinarycsp.factories.constraint.gates.fulladder_gate

fulladder_gate (*variables*, *vartype=<Vartype.BINARY: frozenset({0, 1})>*, *name='FULL_ADDER'*)

Full adder.

Parameters

- **variables** (*list*) – Variable labels for the and gate as *[in1, in2, in3, sum, carry]*, where *in1, in2, in3* are inputs to be added and *sum* and 'carry' the resultant outputs.
- **vartype** (*Vartype, optional, default='BINARY'*) – Variable type. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **name** (*str, optional, default='FULL_ADDER'*) – Name for the constraint.

Returns Constraint that is satisfied when its variables are assigned values that match the valid states of a Boolean full adder.

Return type Constraint(*Constraint*)

Examples

```
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories.constraint.gates as gates
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(gates.fulladder_gate(['a', 'b', 'c_in', 'total', 'c_out'],␣
↪name='FA1'))
>>> csp.check({'a': 1, 'b': 0, 'c_in': 1, 'total': 0, 'c_out': 1})
True
```

Satisfiability Problems

sat.sat2in4(pos[, neg, vartype, name])	Two-in-four (2-in-4) satisfiability.

dwavebinarycsp.factories.constraint.sat.sat2in4

sat2in4 (*pos*, *neg=()*, *vartype=<Vartype.BINARY: frozenset({0, 1})>*, *name='2-in-4'*)

Two-in-four (2-in-4) satisfiability.

Parameters

- **pos** (*iterable*) – Variable labels, as an iterable, for non-negated variables of the constraint. Exactly four variables are specified by *pos* and *neg* together.
- **neg** (*tuple*) – Variable labels, as an iterable, for negated variables of the constraint. Exactly four variables are specified by *pos* and *neg* together.
- **vartype** (*Vartype, optional, default='BINARY'*) – Variable type. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **name** (*str, optional, default='2-in-4'*) – Name for the constraint.

Returns Constraint that is satisfied when its variables are assigned values that satisfy a two-in-four satisfiability problem.

Return type Constraint(*Constraint*)

Examples

```
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories.constraint.sat as sat
>>> csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
>>> csp.add_constraint(sat.sat2in4(['w', 'x', 'y', 'z'], vartype='BINARY', name=
↪'sat1'))
>>> csp.check({'w': 1, 'x': 1, 'y': 0, 'z': 0})
True
```

CSPs

circuits.multiplication_circuit(nbit[, vartype])	Multiplication circuit constraint satisfaction problem.
sat.random_2in4sat(num_variables, num_clauses)	Random two-in-four (2-in-4) constraint satisfaction problem.
sat.random_xorsat(num_variables, num_clauses)	Random XOR constraint satisfaction problem.

dwavebinarycsp.factories.csp.circuits.multiplication_circuit

multiplication_circuit (*nbit*, *vartype=<Vartype.BINARY: frozenset({0, 1})>*)

Multiplication circuit constraint satisfaction problem.

A constraint satisfaction problem that represents the binary multiplication $ab = p$, where the multiplicands are binary variables of length *nbit*; for example, $2^m a_{nbit} + ... + 4a_2 + 2a_1 + a0$.

The square below shows a graphic representation of the circuit:

```
________________________________________________________________________________
|                                         and20         and10         and00    |
|                                           |             |             |      |
|                           and21         add11──and11  add01──and01    |      |
|                             |┌───────────|┌───────────|             |      |
|             and22         add12──and12  add02──and02    |             |      |
```

(continues on next page)

(continued from previous page)

```
|              |----------|----------|            |            |      |
|            add13--------add03          |            |            |      |
|  -----------|            |            |            |            |      |
| p5           p4           p3           p2           p1           p0     |
---------------------------------------------------------------------------------
```

Parameters

- **nbit** (*int*) – Number of bits in the multiplicands.
- **vartype** (*Vartype, optional, default='BINARY'*) – Variable type. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}

Returns CSP that is satisfied when variables a, b, p are assigned values that correctly solve binary multiplication $ab = p$.

Return type CSP (*ConstraintSatisfactionProblem*)

Examples

This example creates a multiplication circuit CSP that multiplies two 3-bit numbers, which is then formulated as a binary quadratic model (BQM). It fixes the multiplacands as $a = 5, b = 3$ (101 and 011) and uses a simulated annealing sampler to find the product, $p = 15$ (001111).

```
>>> import dwavebinarycsp
>>> from dwavebinarycsp.factories.csp.circuits import multiplication_circuit
>>> import neal
>>> csp = multiplication_circuit(3)
>>> bqm = dwavebinarycsp.stitch(csp)
>>> bqm.fix_variable('a0', 1); bqm.fix_variable('a1', 0); bqm.fix_variable('a2',␣
↪1)
>>> bqm.fix_variable('b0', 1); bqm.fix_variable('b1', 1); bqm.fix_variable('b2',␣
↪0)
>>> sampler = neal.SimulatedAnnealingSampler()
>>> response = sampler.sample(bqm)
>>> p = next(response.samples(n=1, sorted_by='energy'))
>>> print(p['p5'], p['p4'], p['p3'], p['p2'], p['p1'], p['p0'])    # doctest:␣
↪+SKIP
0 0 1 1 1 1
```

dwavebinarycsp.factories.csp.sat.random_2in4sat

random_2in4sat (*num_variables, num_clauses, vartype=<Vartype.BINARY: frozenset({0, 1})>, satisfiable=True*)

Random two-in-four (2-in-4) constraint satisfaction problem.

Parameters

- **num_variables** (*integer*) – Number of variables (at least four).
- **num_clauses** (*integer*) – Number of constraints that together constitute the constraint satisfaction problem.

- **vartype** (*Vartype, optional, default='BINARY'*) – Variable type. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **satisfiable** (*bool, optional, default=True*) – True if the CSP can be satisfied.

Returns CSP that is satisfied when its variables are assigned values that satisfy a two-in-four satisfiability problem.

Return type CSP (*ConstraintSatisfactionProblem*)

Examples

This example creates a CSP with 6 variables and two random constraints and checks whether a particular assignment of variables satisifies it.

```
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories as sat
>>> csp = sat.random_2in4sat(6, 2)
>>> csp.constraints    # doctest: +SKIP
[Constraint.from_configurations(frozenset({(1, 0, 1, 0), (1, 0, 0, 1), (1, 1, 1,
↪1), (0, 1, 1, 0), (0, 0, 0, 0),
 (0, 1, 0, 1)}), (2, 4, 0, 1), Vartype.BINARY, name='2-in-4'),
 Constraint.from_configurations(frozenset({(1, 0, 1, 1), (1, 1, 0, 1), (1, 1, 1,
↪0), (0, 0, 0, 1),
 (0, 1, 0, 0), (0, 0, 1, 0)}), (1, 2, 4, 5), Vartype.BINARY, name='2-in-4')]
>>> csp.check({0: 1, 1: 0, 2: 1, 3: 1, 4: 0, 5: 0})       # doctest: +SKIP
True
```

dwavebinarycsp.factories.csp.sat.random_xorsat

random_xorsat (*num_variables, num_clauses, vartype=<Vartype.BINARY: frozenset({0, 1})>, satisfiable=True*)
Random XOR constraint satisfaction problem.

Parameters

- **num_variables** (*integer*) – Number of variables (at least three).
- **num_clauses** (*integer*) – Number of constraints that together constitute the constraint satisfaction problem.
- **vartype** (*Vartype, optional, default='BINARY'*) – Variable type. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **satisfiable** (*bool, optional, default=True*) – True if the CSP can be satisfied.

Returns CSP that is satisfied when its variables are assigned values that satisfy a XOR satisfiability problem.

Return type CSP (*ConstraintSatisfactionProblem*)

Examples

This example creates a CSP with 5 variables and two random constraints and checks whether a particular assignment of variables satisifies it.

```
>>> import dwavebinarycsp
>>> import dwavebinarycsp.factories as sat
>>> csp = sat.random_xorsat(5, 2)
>>> csp.constraints    # doctest: +SKIP
[Constraint.from_configurations(frozenset({(1, 0, 0), (1, 1, 1), (0, 1, 0), (0, 0,
↪ 1)}), (4, 3, 0),
 Vartype.BINARY, name='XOR (0 flipped)'),
 Constraint.from_configurations(frozenset({(1, 1, 0), (0, 1, 1), (0, 0, 0), (1, 0,
↪ 1)}), (2, 0, 4),
 Vartype.BINARY, name='XOR (2 flipped) (0 flipped)')]
>>> csp.check({0: 1, 1: 0, 2: 0, 3: 1, 4: 1})       # doctest: +SKIP
True
```

1.3 Examples

Release 0.1.1

Date Feb 25, 2020

1.3.1 Circuit Fault Diagnosis

Fault diagnosis is the combinational problem of quickly localizing failures as soon as they are detected in systems. Circuit fault diagnosis (CFD) is the problem of identifying a minimum-sized set of gates that, if faulty, explains an observation of incorrect outputs given a set of inputs.

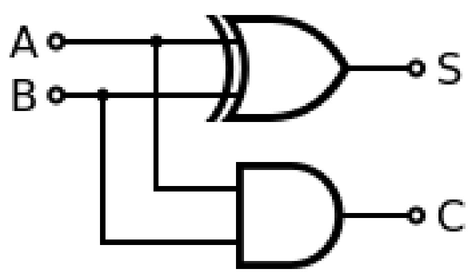

Fig. 3: A Half Adder made up of an XOR gate and an AND gate.

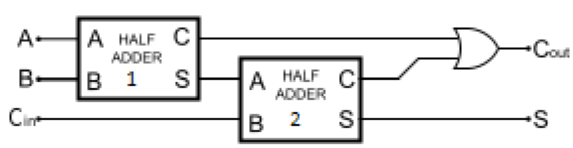

Fig. 4: A Full Adder made up of two Half Adders.

The following example demonstrates some of the techniques available to formulate a given problem so it can be solved on the D-Wave system.

Circuit Fault Diagnosis with Explicit Fault Variables

We can construct the constraints for the circuit fault diagnosis in the following way:

- Each input/output/wire in the circuit is represented by a binary variable in the problem.
- **Each gate can either be:**
  - Healthy, in which case it behaves according to it's normal truth table.
  - Faulty, in which case it does not.

To build these constraints, we start with the truth table for the gate we wish to encode, say an AND gate:

A	B	Output
0	0	0
1	0	0
0	1	0
1	1	1

We then add a new *explicit fault variable*, which encodes whether the gate is faulty or now.

A	B	Output	Faulty
0	0	0	0
1	0	0	0
0	1	0	0
1	1	1	0
0	0	1	1
1	0	1	1
0	1	1	1
1	1	0	1

This new truth table with the explicit fault variable allows the CSP to be satisfied even when the gate is not healthy.

Example Code

The following code demonstrates how to find both fault diagnoses and minimum fault diagnoses for the above Full Adder.

```
import dwavebinarycsp

from dimod import ExactSolver


def xor_fault(a, b, out, fault):
    """Returns True if XOR(a, b) == out and fault == 0 or XOR(a, b) != out and fault
↪== 1."""
    if (a != b) == out:
        return fault == 0
    else:
        return fault == 1


def and_fault(a, b, out, fault):
    """Returns True if AND(a, b) == out and fault == 0 or AND(a, b) != out and fault
↪== 1."""
```

(continues on next page)

(continued from previous page)

```
    if (a and b) == out:
        return fault == 0
    else:
        return fault == 1


def or_fault(a, b, out, fault):
    """Returns True if OR(a, b) == out and fault == 0 or OR(a, b) != out and fault ==
↪1."""
    if (a or b) == out:
        return fault == 0
    else:
        return fault == 1


csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)

# the first half adder
csp.add_constraint(xor_fault, ['A1', 'B1', 'S1/A2', 'xor_fault_1'])
csp.add_constraint(and_fault, ['A1', 'B1', 'C1', 'and_fault_1'])

# the second half adder
csp.add_constraint(xor_fault, ['S1/A2', 'B2', 'S2', 'xor_fault_2'])
csp.add_constraint(and_fault, ['S1/A2', 'B2', 'C2', 'and_fault_2'])

# finally the AND gate
csp.add_constraint(or_fault, ['C1', 'C2', 'ORout', 'or_fault'])


# now, say that the behaviour we witnessed was HA(0, 1, 0) -> 1, 1.
# The 'A' input to the circuit is 'A1'
csp.fix_variable('A1', 0)
# The 'B' input to the circuit is 'B1'
csp.fix_variable('B1', 1)
# the 'Cin' input to the circuit is 'B2'
csp.fix_variable('B2', 0)
# the sum output of the circuit is 'S2'
csp.fix_variable('S2', 1)
# the carry output of the circuit is 'ORout'
csp.fix_variable('ORout', 1)


# convert the csp to a bqm. We specify that the energy gap between the valid
↪configurations and
# the invalid ones must be at least 2.0
bqm = dwavebinarycsp.stitch(csp, min_classical_gap=2.0)


# set up any dimod solver. In this case we use the ExactSolver but any unstructured
↪solver would
# work.
sampler = ExactSolver()


# we can determine the minimum and maximum number of faults that will induce this
↪behavior
response = sampler.sample(bqm)
```

(continues on next page)

(continued from previous page)

```
min_energy = min(response.data_vectors['energy'])

fault_counts = []
for sample, energy in response.data(['sample', 'energy']):
    if csp.check(sample):
        n_faults = sum(sample[v] for v in sample if 'fault' in v)
        fault_counts.append(n_faults)
    else:
        # if the CSP is not satisfied, the energy should be above ground
        assert energy > min_energy

print('Minimum number of faults: ', min(fault_counts))
print('Maximum number of faults: ', max(fault_counts))

# If, instead of the ground states corresponding to all possible fault configurations,
↪ we
# instead only wanted to sample from minimum fault configurations, we need to bias␣
↪against
# higher fault cardinalities. To do this, we add a small linear bias to the fault␣
↪variables.
# We also make sure that the bias we add is less than 2.0, or else we would affect␣
↪the energy
# levels.
bqm.add_variable('xor_fault_1', .5)  # if the variable is present, add_variable adds␣
↪to the linear bias
bqm.add_variable('and_fault_1', .5)
bqm.add_variable('xor_fault_2', .5)
bqm.add_variable('and_fault_2', .5)
bqm.add_variable('or_fault', .5)

# now the samples that satisfy the csp and are minimum energy should be exactly the␣
↪fault
# diagnosis with only a single fault
response = sampler.sample(bqm)
min_energy = min(response.data_vectors['energy'])

min_fault_diagnoses = []
for sample, energy in response.data(['sample', 'energy']):
    if csp.check(sample) and energy == min_energy:
        min_fault_diagnoses.append([v for v in sample if ('fault' in v and␣
↪sample[v])])
    else:
        # if the CSP is not satisfied, the energy should be above ground
        assert energy > min_energy

print('min fault diagnoses: ', min_fault_diagnoses)
```

1.4 Bibliography

1.5 Installation

To install:

```
pip install dwavebinarycsp
```

To build from source:

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

1.6 License

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

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 2

Indices and tables

- genindex
- modindex
- search
- Glossary

Python Module Index

c

f

i

r

Index