dwave-system Documentation

Release 1.18.0

D-Wave Systems Inc

May 12, 2023

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, `LeapHybridSampler`, for Leap's hybrid solvers, and other. 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 in either a hybrid quantum-classical solution, using `LeapHybridSampler()`, for example, or dwave-hybrid samplers such as `KerberosSampler`, or directly using `DWaveSampler()`.

Note: For applications that require detailed control on communication with the remote compute resource (a D-Wave QPU or Leap's hybrid solvers), see dwave-cloud-client.

D-Wave System Documentation describes D-Wave quantum computers and Leap hybrid solvers, including features, parameters, and properties. It also provides guidance on programming the D-Wave system, including how to formulate problems and configure parameters.

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. 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
```

(continues on next page)

(continued from previous page)

```
>>> from dwave.system import DWaveSampler, EmbeddingComposite
...
>>> s5 = nx.star_graph(4)  # a star graph where node 0 is hub to four other nodes
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> print(dnx.min_vertex_cover(s5, sampler))
[0]
```

1.2 Reference Documentation

1.2.1 Samplers

A sampler accepts a problem in binary quadratic model (BQM) or discrete quadratic model (DQM) format and returns variable assignments. Samplers generally try to find minimizing values but can also sample from distributions defined by the problem.

- *DWaveSampler*
- *DWaveCliqueSampler*
- *LeapHybridSampler*
- *LeapHybridCQMSampler*
- *LeapHybridDQMSampler*

These samplers are non-blocking: the returned `SampleSet` is constructed from a `Future`-like object that is resolved on the first read of any of its properties; for example, by printing the results. Your code can query its status with the `done()` method or ensure resolution with the `resolve()` method.

Other Ocean packages provide additional samplers; for example, dimod provides samplers for testing your code.

DWaveSampler

class DWaveSampler(*failover=False*, *retry_interval=-1*, ***config*)

A class for using the D-Wave system as a sampler for binary quadratic models.

You can configure your solver selection and usage by setting parameters, hierarchically, in a configuration file, as environment variables, or explicitly as input arguments. For more information, see D-Wave Cloud Client `get_solvers()`. By default, online D-Wave systems are returned ordered by highest number of qubits.

Inherits from `dimod.Sampler` and `dimod.Structured`.

Parameters

- **failover** (*bool, optional, default=False*) – Signal a failover condition if a sampling error occurs. When `True`, raises `FailoverCondition` or `RetryCondition` on sampleset resolve to signal failover.

  Actual failover, i.e. selection of a new solver, has to be handled by the user. A convenience method `trigger_failover()` is available for this. Note that hardware graphs vary between QPUs, so triggering failover results in regenerated *nodelist*, *edgelist*, *properties* and *parameters*.

Changed in version 1.16.0: In the past, the *sample()* method was blocking and `failover=True` caused a solver failover and sampling retry. However, this failover implementation broke when *sample()* became non-blocking (asynchronous), Setting `failover=True` had no effect.

- **retry_interval** (*number, optional, default=-1*) – Ignored, but kept for backward compatibility.

  Changed in version 1.16.0: Ignored since 1.16.0. See note for `failover` parameter above.

- ****config** – Keyword arguments passed to `dwave.cloud.client.Client.from_config()`.

Note: Prior to version 1.0.0, *DWaveSampler* used the `base` client, allowing non-QPU solvers to be selected. To reproduce the old behavior, instantiate *DWaveSampler* with `client='base'`.

Examples

This example submits a two-variable Ising problem mapped directly to two adjacent qubits on a D-Wave system. `qubit_a` is the first qubit in the QPU's indexed list of qubits and `qubit_b` is one of the qubits coupled to it. Other required parameters for communication with the system, such as its URL and an authentication token, are implicitly set in a configuration file or as environment variables, as described in Configuring Access to D-Wave Solvers. Given sufficient reads (here 100), the quantum computer should return the best solution, $1, -1$ on `qubit_a` and `qubit_b`, respectively, as its first sample (samples are ordered from lowest energy).

```
>>> from dwave.system import DWaveSampler
...
>>> sampler = DWaveSampler()
...
>>> qubit_a = sampler.nodelist[0]
>>> qubit_b = next(iter(sampler.adjacency[qubit_a]))
>>> sampleset = sampler.sample_ising({qubit_a: -1, qubit_b: 1},
...                                  {},
...                                  num_reads=100)
>>> sampleset.first.sample[qubit_a] == 1 and sampleset.first.sample[qubit_b] == -1
True
```

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

Properties

For parameters and properties of D-Wave systems, see D-Wave System Documentation.

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 *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.

Continued on next page

Table 1 – continued from previous page

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()` 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 import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.properties    # doctest: +SKIP
{'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 *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 import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.parameters    # doctest: +SKIP
{'anneal_offsets': ['parameters'],
 'anneal_schedule': ['parameters'],
 'annealing_time': ['parameters'],
 'answer_mode': ['parameters'],
 '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

First 5 entries of the node list for one Advantage system.

```
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.nodelist[:5]    # doctest: +SKIP
[30, 31, 32, 33, 34]
```

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

First 5 entries of the coupler list for one Advantage system.

```
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.edgelist[:5]    # doctest: +SKIP
[(30, 31), (30, 45), (30, 2940), (30, 2955), (30, 2970)]
```

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.

dwave.system.samplers.DWaveSampler.structure

DWaveSampler.**structure**

Structure of the structured sampler formatted as a `namedtuple()` where the 3-tuple values are the *`nodelist`*, *`edgelist`* and *`adjacency`* attributes.

Methods

DWaveSampler.sample(bqm[, warnings])	Sample from the specified binary quadratic model.
DWaveSampler.sample_ising(h, *args, **kwargs)	Sample from an Ising model using the implemented sample method.
DWaveSampler.sample_qubo(Q, Hashable], ...)	Sample from a QUBO using the implemented sample method.
DWaveSampler.validate_anneal_schedule(...)	Raise an exception if the specified schedule is invalid for the sampler.
DWaveSampler.to_networkx_graph()	Converts DWaveSampler's structure to a Chimera, Pegasus or Zephyr NetworkX graph.

dwave.system.samplers.DWaveSampler.sample

DWaveSampler.**sample**(*bqm*, *warnings=None*, ***kwargs*)

Sample from the specified binary quadratic model.

Parameters

- **bqm** (BinaryQuadraticModel) – The binary quadratic model. Must match *nodelist* and *edgelist*.
- **warnings** (*WarningAction*, optional) – Defines what warning action to take, if any. See *Warnings*. The default behaviour is to ignore warnings.
- ****kwargs** – Optional keyword arguments for the sampling method, specified per solver in *parameters*. D-Wave System Documentation's solver guide describes the parameters and properties supported on the D-Wave system.

Returns Sample set constructed from a (non-blocking) Future-like object. In it this sampler also provides timing information in the *info* field as described in the D-Wave System Documentation's QPU Timing Information from SAPI.

Return type SampleSet

Examples

This example submits a two-variable Ising problem mapped directly to two adjacent qubits on a D-Wave system. qubit_a is the first qubit in the QPU's indexed list of qubits and qubit_b is one of the qubits coupled to it. Given sufficient reads (here 100), the quantum computer should return the best solution, $1, -1$ on qubit_a and qubit_b, respectively, as its first sample (samples are ordered from lowest energy).

```
>>> from dwave.system import DWaveSampler
...
>>> sampler = DWaveSampler()
...
>>> qubit_a = sampler.nodelist[0]
>>> qubit_b = next(iter(sampler.adjacency[qubit_a]))
>>> sampleset = sampler.sample_ising({qubit_a: -1, qubit_b: 1},
...                                  {},
...                                  num_reads=100)
>>> sampleset.first.sample[qubit_a] == 1 and sampleset.first.sample[qubit_b] == -1
True
```

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

dwave.system.samplers.DWaveSampler.sample_ising

DWaveSampler.**sample_ising**(*h*, **args*, ***kwargs*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.samplers.DWaveSampler.sample_qubo

DWaveSampler.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a BinaryQuadraticModel and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

See also:

sample(), *sample_ising()*

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 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
>>>
```

dwave.system.samplers.DWaveSampler.to_networkx_graph

`DWaveSampler.`**`to_networkx_graph`**`()`

Converts DWaveSampler's structure to a Chimera, Pegasus or Zephyr NetworkX graph.

Returns Either a Chimera lattice of shape [m, n, t], a Pegasus lattice of shape [m] or a Zephyr lattice of size [m,t].

Return type `networkx.Graph`

Examples

This example converts a selected D-Wave system solver to a graph and verifies it has over 5000 nodes.

```
>>> from dwave.system import DWaveSampler
...
>>> sampler = DWaveSampler()
>>> g = sampler.to_networkx_graph()       # doctest: +SKIP
>>> len(g.nodes) > 5000                   # doctest: +SKIP
True
```

DWaveCliqueSampler

`class DWaveCliqueSampler``(*, failover: bool = False, retry_interval: numbers.Number = -1, **config)`

A sampler for solving clique binary quadratic models on the D-Wave system.

This sampler wraps `find_clique_embedding()` to generate embeddings with even chain length. These embeddings work well for dense binary quadratic models. For sparse models, using *EmbeddingComposite* with *DWaveSampler* is preferred.

Configuration such as solver selection is similar to that of *DWaveSampler*.

Parameters

- **failover** (*bool, optional, default=False*) – Signal a failover condition if a sampling error occurs. When `True`, raises `FailoverCondition` or `RetryCondition` on sampleset resolve to signal failover.

  Actual failover, i.e. selection of a new solver, has to be handled by the user. A convenience method `trigger_failover()` is available for this. Note that hardware graphs vary between QPUs, so triggering failover results in regenerated *nodelist*, *edgelist*, *properties* and *parameters*.

  Changed in version 1.16.0: In the past, the *sample()* method was blocking and `failover=True` caused a solver failover and sampling retry. However, this failover implementation broke when *sample()* became non-blocking (asynchronous), Setting `failover=True` had no effect.

- **retry_interval** (*number, optional, default=-1*) – Ignored, but kept for backward compatibility.

  Changed in version 1.16.0: Ignored since 1.16.0. See note for `failover` parameter above.

- ****config** – Keyword arguments, as accepted by *DWaveSampler*

Examples

This example creates a BQM based on a 6-node clique (complete graph), with random ± 1 values assigned to nodes, and submits it to a D-Wave system. Parameters for communication with the system, such as its URL and an authentication token, are implicitly set in a configuration file or as environment variables, as described in Configuring Access to D-Wave Solvers.

```
>>> from dwave.system import DWaveCliqueSampler
>>> import dimod
...
>>> bqm = dimod.generators.ran_r(1, 6)
...
>>> sampler = DWaveCliqueSampler()    # doctest: +SKIP
>>> sampler.largest_clique_size > 5  # doctest: +SKIP
True
>>> sampleset = sampler.sample(bqm, num_reads=100)    # doctest: +SKIP
```

Properties

DWaveCliqueSampler.largest_clique_size	The maximum number of variables that can be embedded.
DWaveCliqueSampler.qpu_linear_range	Range of linear biases allowed by the QPU.
DWaveCliqueSampler.qpu_quadratic_range	Range of quadratic biases allowed by the QPU.
DWaveCliqueSampler.properties	Properties as a dict containing any additional information about the sampler.

Continued on next page

Table 3 – continued from previous page

DWaveCliqueSampler.parameters	Parameters as a dict, where keys are keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
DWaveCliqueSampler.target_graph	The QPU topology.

dwave.system.samplers.DWaveCliqueSampler.largest_clique_size

DWaveCliqueSampler.**largest_clique_size**
The maximum number of variables that can be embedded.

dwave.system.samplers.DWaveCliqueSampler.qpu_linear_range

DWaveCliqueSampler.**qpu_linear_range**
Range of linear biases allowed by the QPU.

dwave.system.samplers.DWaveCliqueSampler.qpu_quadratic_range

DWaveCliqueSampler.**qpu_quadratic_range**
Range of quadratic biases allowed by the QPU.

dwave.system.samplers.DWaveCliqueSampler.properties

DWaveCliqueSampler.**properties**
Properties as a dict containing any additional information about the sampler.

dwave.system.samplers.DWaveCliqueSampler.parameters

DWaveCliqueSampler.**parameters**
Parameters as a dict, where keys are keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

dwave.system.samplers.DWaveCliqueSampler.target_graph

DWaveCliqueSampler.**target_graph**
The QPU topology.

Methods

DWaveCliqueSampler.largest_clique()	The clique embedding with the maximum number of source variables.
DWaveCliqueSampler.sample(bqm[, chain_strength])	Sample from the specified binary quadratic model.
DWaveCliqueSampler.sample_ising(h, ...)	Sample from an Ising model using the implemented sample method.

Continued on next page

Table 4 – continued from previous page

DWaveCliqueSampler.sample_qubo(Q, Hashable], ...)	Sample from a QUBO using the implemented sample method.

dwave.system.samplers.DWaveCliqueSampler.largest_clique

DWaveCliqueSampler.**largest_clique**()

The clique embedding with the maximum number of source variables.

Returns The clique embedding with the maximum number of source variables.

Return type dict

dwave.system.samplers.DWaveCliqueSampler.sample

DWaveCliqueSampler.**sample**(*bqm*, *chain_strength=None*, ***kwargs*)

Sample from the specified binary quadratic model.

Parameters

- **bqm** (BinaryQuadraticModel) – Any binary quadratic model with up to *largest_clique_size* variables. This BQM is embedded using a clique embedding.
- **chain_strength** (*float/mapping/callable, optional*) – Sets the coupling strength between qubits representing variables that form a chain. Mappings should specify the required chain strength for each variable. Callables should accept the BQM and embedding and return a float or mapping. By default, *chain_strength* is calculated with *uniform_torque_compensation()*.
- ****kwargs** – Optional keyword arguments for the sampling method, specified per solver in *parameters*. D-Wave System Documentation's solver guide describes the parameters and properties supported on the D-Wave system. Note that *auto_scale* is not supported by this sampler, because it scales the problem as part of the embedding process.

Returns Sample set constructed from a (non-blocking) Future-like object.

Return type SampleSet

dwave.system.samplers.DWaveCliqueSampler.sample_ising

DWaveCliqueSampler.**sample_ising**(*h: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.samplers.DWaveCliqueSampler.sample_qubo

DWaveCliqueSampler.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

See also:

sample(), *sample_ising()*

LeapHybridSampler

class LeapHybridSampler(***config*)

A class for using Leap's cloud-based hybrid BQM solvers.

Leap's quantum-classical hybrid BQM solvers are intended to solve arbitrary application problems formulated as binary quadratic models (BQM).

You can configure your solver selection and usage by setting parameters, hierarchically, in a configuration file, as environment variables, or explicitly as input arguments, as described in D-Wave Cloud Client.

dwave-cloud-client's `get_solvers()` method filters solvers you have access to by solver properties `category=hybrid` and `supported_problem_type=bqm`. By default, online hybrid BQM solvers are returned ordered by latest `version`.

The default specification for filtering and ordering solvers by features is available as *default_solver* property. Explicitly specifying a solver in a configuration file, an environment variable, or keyword arguments overrides this specification. See the example below on how to extend it instead.

Parameters ****config** – Keyword arguments passed to `dwave.cloud.client.Client.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
```

This example specializes the default solver selection by filtering out bulk BQM solvers. (Bulk solvers are throughput-optimal for heavy/batch workloads, have a higher start-up latency, and are not well suited for live workloads. Not all Leap accounts have access to bulk solvers.)

```
>>> from dwave.system import LeapHybridSampler
...
>>> solver = LeapHybridSampler.default_solver
>>> solver.update(name__regex=".*(?<!bulk)$")        # name shouldn't end with
↪"bulk"
>>> sampler = LeapHybridSampler(solver=solver)       # doctest: +SKIP
>>> sampler.solver         # doctest: +SKIP
BQMSolver(id='hybrid_binary_quadratic_model_version2')
```

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 *properties* for each key.
LeapHybridSampler.default_solver	

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.

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 *properties* for each key.

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

dwave.system.samplers.LeapHybridSampler.default_solver

```
LeapHybridSampler.default_solver = {'order_by':  '-properties.version', 'supported_problem_
```

Methods

LeapHybridSampler.sample(bqm[, time_limit])	Sample from the specified binary quadratic model.
LeapHybridSampler.sample_ising(h, ...)	Sample from an Ising model using the implemented sample method.
LeapHybridSampler.sample_qubo(Q, Hashable], ...)	Sample from a QUBO using the implemented sample method.
LeapHybridSampler.min_time_limit(bqm)	Return the minimum *time_limit* accepted for the given problem.

dwave.system.samplers.LeapHybridSampler.sample

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

Sample from the specified binary quadratic model.

Parameters

- **bqm** (dimod.BinaryQuadraticModel) – 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.

  min_time_limit() calculates (and describes) the minimum time for your problem.
- ****kwargs** – Optional keyword arguments for the solver, specified in *parameters*.

Returns Sample set constructed from a (non-blocking) Future-like object.

Return type 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: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.samplers.LeapHybridSampler.sample_qubo

LeapHybridSampler.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

See also:

sample(), *sample_ising()*

dwave.system.samplers.LeapHybridSampler.min_time_limit

LeapHybridSampler.**min_time_limit**(*bqm*)

Return the minimum *time_limit* accepted for the given problem.

The minimum time for a hybrid BQM solver is specified as a piecewise-linear curve defined by a set of floating-point pairs, the *minimum_time_limit* field under *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 number of variables is a linear interpolation calculated on two pairs that represent the relevant range for the given number of variables.

Examples

For a solver where *LeapHybridSampler().properties["minimum_time_limit"]* returns *[[1, 0.1], [100, 10.0], [1000, 20.0]]*, the minimum time for a problem 50 variales is 5 seconds (the linear interpolation of the first two pairs that represent problems with between 1 to 100 variables).

LeapHybridCQMSampler

class LeapHybridCQMSampler(***config*)

A class for using Leap's cloud-based hybrid CQM solvers.

Leap's quantum-classical hybrid CQM solvers are intended to solve application problems formulated as constrained quadratic models (CQM).

You can configure your solver selection and usage by setting parameters, hierarchically, in a configuration file, as environment variables, or explicitly as input arguments, as described in D-Wave Cloud Client.

dwave-cloud-client's `get_solvers()` method filters solvers you have access to by solver properties `category=hybrid` and `supported_problem_type=cqm`. By default, online hybrid CQM solvers are returned ordered by latest `version`.

Parameters ****config** – Keyword arguments passed to `dwave.cloud.client.Client.from_config()`.

Examples

This example solves a simple problem of finding the rectangle with the greatest area when the perimeter is limited. In this example, the perimeter of the rectangle is set to 8 (meaning the largest area is for the $2X2$ square).

A CQM is created that will have two integer variables, i, j, each limited to half the maximum perimeter length of 8, to represent the lengths of the rectangle's sides:

```
>>> from dimod import ConstrainedQuadraticModel, Integer
>>> i = Integer('i', upper_bound=4)
>>> j = Integer('j', upper_bound=4)
>>> cqm = ConstrainedQuadraticModel()
```

The area of the rectangle is given by the multiplication of side i by side j. The goal is to maximize the area, $i * j$. Because D-Wave samplers minimize, the objective should have its lowest value when this goal is met. Objective $-i * j$ has its minimum value when $i * j$, the area, is greatest:

```
>>> cqm.set_objective(-i*j)
```

Finally, the requirement that the sum of both sides must not exceed the perimeter is represented as constraint $2i + 2j <= 8$:

```
>>> cqm.add_constraint(2*i+2*j <= 8, "Max perimeter")
'Max perimeter'
```

Instantiate a hybrid CQM sampler and submit the problem for solution by a remote solver provided by the Leap quantum cloud service:

```
>>> from dwave.system import LeapHybridCQMSampler    # doctest: +SKIP
>>> sampler = LeapHybridCQMSampler()                 # doctest: +SKIP
>>> sampleset = sampler.sample_cqm(cqm)              # doctest: +SKIP
```

(continues on next page)

(continued from previous page)

```
>>> print(sampleset.first)                          # doctest: +SKIP
Sample(sample={'i': 2.0, 'j': 2.0}, energy=-4.0, num_occurrences=1,
...            is_feasible=True, is_satisfied=array([ True]))
```

The best (lowest-energy) solution found has $i = j = 2$ as expected, a solution that is feasible because all the constraints (one in this example) are satisfied.

Properties

LeapHybridCQMSampler.properties	Solver properties as returned by a SAPI query.
LeapHybridCQMSampler.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 *properties* for each key.

dwave.system.samplers.LeapHybridCQMSampler.properties

LeapHybridCQMSampler.**properties**
Solver properties as returned by a SAPI query.

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

dwave.system.samplers.LeapHybridCQMSampler.parameters

LeapHybridCQMSampler.**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 *properties* for each key.

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

Methods

LeapHybridCQMSampler.sample_cqm(cqm, ...)	Sample from the specified constrained quadratic model.
LeapHybridCQMSampler.min_time_limit(cqm)	Return the minimum *time_limit* accepted for the given problem.

dwave.system.samplers.LeapHybridCQMSampler.sample_cqm

LeapHybridCQMSampler.**sample_cqm**(*cqm: dimod.constrained.constrained.ConstrainedQuadraticModel*, *time_limit: Optional[float] = None*, ***kwargs*)
Sample from the specified constrained quadratic model.

Parameters

- **cqm** (*dimod.ConstrainedQuadraticModel*) – Constrained quadratic model (CQM).
- **time_limit** (*int, optional*) – Maximum run time, in seconds, to allow the solver to work on the problem. Must be at least the minimum required for the problem, which is calculated and set by default.

min_time_limit() calculates (and describes) the minimum time for your problem.

- ****kwargs** – Optional keyword arguments for the solver, specified in *parameters*.

Returns Sample set constructed from a (non-blocking) Future-like object.

Return type SampleSet

Examples

See the example in *LeapHybridCQMSampler*.

dwave.system.samplers.LeapHybridCQMSampler.min_time_limit

LeapHybridCQMSampler.**min_time_limit**(*cqm: dimod.constrained.constrained.ConstrainedQuadraticModel*) → float

Return the minimum *time_limit* accepted for the given problem.

LeapHybridDQMSampler

class LeapHybridDQMSampler(***config*)

A class for using Leap's cloud-based hybrid DQM solvers.

Leap's quantum-classical hybrid DQM solvers are intended to solve arbitrary application problems formulated as **discrete** quadratic models (DQM).

You can configure your solver selection and usage by setting parameters, hierarchically, in a configuration file, as environment variables, or explicitly as input arguments, as described in D-Wave Cloud Client.

dwave-cloud-client's `get_solvers()` method filters solvers you have access to by solver properties `category=hybrid` and `supported_problem_type=dqm`. By default, online hybrid DQM solvers are returned ordered by latest `version`.

The default specification for filtering and ordering solvers by features is available as *default_solver* property. Explicitly specifying a solver in a configuration file, an environment variable, or keyword arguments overrides this specification. See the example in *LeapHybridSampler* on how to extend it instead.

Parameters ****config** – Keyword arguments passed to `dwave.cloud.client.Client.from_config()`.

Examples

This example solves a small, illustrative problem: a game of rock-paper-scissors. The DQM has two variables representing two hands, with cases for rock, paper, scissors. Quadratic biases are set to produce a lower value of the DQM for cases of variable `my_hand` interacting with cases of variable `their_hand` such that the former wins over the latter; for example, the interaction of `rock-scissors` is set to -1 while `scissors-rock` is set to +1.

```
>>> import dimod
>>> from dwave.system import LeapHybridDQMSampler
...
>>> cases = ["rock", "paper", "scissors"]
>>> win = {"rock": "scissors", "paper": "rock", "scissors": "paper"}
...
>>> dqm = dimod.DiscreteQuadraticModel()
```

(continues on next page)

(continued from previous page)

```
>>> dqm.add_variable(3, label='my_hand')
'my_hand'
>>> dqm.add_variable(3, label='their_hand')
'their_hand'
>>> for my_idx, my_case in enumerate(cases):
...     for their_idx, their_case in enumerate(cases):
...         if win[my_case] == their_case:
...             dqm.set_quadratic('my_hand', 'their_hand',
...                               {(my_idx, their_idx): -1})
...         if win[their_case] == my_case:
...             dqm.set_quadratic('my_hand', 'their_hand',
...                               {(my_idx, their_idx): 1})
...
>>> dqm_sampler = LeapHybridDQMSampler()       # doctest: +SKIP
...
>>> sampleset = dqm_sampler.sample_dqm(dqm)    # doctest: +SKIP
>>> print("{} beats {}".format(cases[sampleset.first.sample['my_hand']],
...                            cases[sampleset.first.sample['their_hand']]))    #␣
↪doctest: +SKIP
rock beats scissors
```

Properties

LeapHybridDQMSampler.properties	Solver properties as returned by a SAPI query.
LeapHybridDQMSampler.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 *properties* for each key.
LeapHybridDQMSampler.default_solver	

dwave.system.samplers.LeapHybridDQMSampler.properties

LeapHybridDQMSampler.**properties**

Solver properties as returned by a SAPI query.

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

dwave.system.samplers.LeapHybridDQMSampler.parameters

LeapHybridDQMSampler.**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 *properties* for each key.

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

dwave.system.samplers.LeapHybridDQMSampler.default_solver

LeapHybridDQMSampler.**default_solver = {'order_by': '-properties.version', 'supported_probl**

Methods

LeapHybridDQMSampler.sample_dqm(dqm[, ...])	Sample from the specified discrete quadratic model.
LeapHybridDQMSampler.min_time_limit(dqm)	Return the minimum *time_limit* accepted for the given problem.

dwave.system.samplers.LeapHybridDQMSampler.sample_dqm

LeapHybridDQMSampler.**sample_dqm**(*dqm*, *time_limit=None*, *compress=False*, *compressed=None*, ***kwargs*)

Sample from the specified discrete quadratic model.

Parameters

- **dqm** (dimod.DiscreteQuadraticModel) – Discrete quadratic model (DQM).

  Note that if *dqm* is a dimod.CaseLabelDQM, then map_sample() will need to be used to restore the case labels in the returned sample set.

- **time_limit** (*int, optional*) – 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.

  min_time_limit() calculates (and describes) the minimum time for your problem.

- **compress** (*binary, optional*) – Compresses the DQM data when set to True. Use if your problem somewhat exceeds the maximum allowed size. Compression tends to be slow and more effective on homogenous data, which in this case means it is more likely to help on DQMs with many identical integer-valued biases than ones with random float-valued biases, for example.

- **compressed** (*binary, optional*) – Deprecated; please use compress instead.

- ****kwargs** – Optional keyword arguments for the solver, specified in *parameters*.

Returns Sample set constructed from a (non-blocking) Future-like object.

Return type SampleSet

Examples

See the example in *LeapHybridDQMSampler*.

dwave.system.samplers.LeapHybridDQMSampler.min_time_limit

LeapHybridDQMSampler.**min_time_limit**(*dqm*)

Return the minimum *time_limit* accepted for the given problem.

The minimum time for a hybrid DQM solver is specified as a piecewise-linear curve defined by a set of floating-point pairs, the *minimum_time_limit* field under *properties*. The first element in each pair is a combination of the numbers of interactions, variables, and cases that reflects the "density" of connectivity between the problem's variables; the second is the minimum required time. The minimum time for any particular problem size is a linear interpolation calculated on two pairs that represent the relevant range for the given problem.

Examples

For a solver where *LeapHybridDQMSampler().properties["minimum_time_limit"]* returns *[[1, 0.1], [100, 10.0], [1000, 20.0]]*, the minimum time for a problem of "density" 50 is 5 seconds (the linear interpolation of the first two pairs that represent problems with "density" between 1 to 100).

1.2.2 Composites

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

Other Ocean packages provide additional composites; for example, dimod provides composites that operate on the problem (e.g., scaling values), track inputs and outputs for debugging, and other useful functionality relevant to generic samplers.

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 unchanged {'a': 1, 'b': -1, 'c': -1} and this solution is found. However, the sample is attributed the energy appropriate to the bqm without thresholding.

```
>>> import dimod
>>> sampler = DWaveSampler(solver={'qpu': True})
>>> bqm = dimod.BinaryQuadraticModel({'a': -1, 'b': 1, 'c': 1},
...                                  {'ab': 0.8, 'ac': 0.7, 'bc': -1},
...                                  0,
...                                  dimod.SPIN)
>>> CutOffComposite(AutoEmbeddingComposite(sampler), 0.75).sample(bqm,
...                 num_reads=1000).first.energy
-5.5
```

Properties

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

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.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.

dwave.system.composites.CutOffComposite.properties

`CutOffComposite.`**`properties`**
A dict containing any additional information about the sampler.

Methods

`CutOffComposite.sample`(bqm, **parameters)	Cut off interactions and sample from the provided binary quadratic model.
`CutOffComposite.sample_ising`(h, Union[float, ...)	Sample from an Ising model using the implemented sample method.
`CutOffComposite.sample_qubo`(Q, Hashable], ...)	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 value 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: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.composites.CutOffComposite.sample_qubo

CutOffComposite.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

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.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.
PolyCutOffComposite.properties	A dict containing any additional information about the sampler.

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.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.

dwave.system.composites.PolyCutOffComposite.properties

`PolyCutOffComposite.`**`properties`**
A dict containing any additional information about the sampler.

Methods

PolyCutOffComposite.sample_poly(poly, **kwargs)	Cutoff and sample from the provided binary polynomial.
PolyCutOffComposite.sample_hising(h, ...)	Sample from a higher-order Ising model.
PolyCutOffComposite.sample_hubo(H, ...)	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: Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **kwargs*) → dimod.sampleset.SampleSet
Sample from a higher-order Ising model.

Converts the given higher-order Ising model to a `BinaryPolynomial` and calls *sample_poly()*.

Parameters

- **h** – Variable biases of the Ising problem.
- **J** – Interaction biases of the Ising problem.
- ****kwargs** – See *sample_poly()* for additional keyword definitions.

Returns Samples from the higher-order Ising model.

See also:

sample_poly(), *sample_hubo()*

dwave.system.composites.PolyCutOffComposite.sample_hubo

PolyCutOffComposite.**sample_hubo**(*H: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]]*, ***kwargs*) → dimod.sampleset.SampleSet

Sample from a higher-order unconstrained binary optimization problem.

Converts the given higher-order unconstrained binary optimization problem to a `BinaryPolynomial` and then calls *sample_poly()*.

Parameters

- **H** – Coefficients of the HUBO.
- ****kwargs** – See *sample_poly()* for additional keyword definitions.

Returns Samples from a higher-order unconstrained binary optimization problem.

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

- **child_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, ...)	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/mapping/callable, optional*) – Sets the coupling strength between qubits representing variables that form a chain. Mappings should spec-

ify the required chain strength for each variable. Callables should accept the BQM and embedding and return a float or mapping. By default, *chain_strength* is calculated with *uniform_torque_compensation()*.

- **chain_break_method** (*function/list, optional*) – Method or methods used to resolve chain breaks. If multiple methods are given, the results are concatenated and a new field called "chain_break_method" specifying the index of the method is appended to the sample set. See *unembed_sampleset()* and *dwave.embedding.chain_breaks*.
- **chain_break_fraction** (*bool, optional, default=True*) – Add a *chain_break_fraction* field to the unembedded response with the fraction of chains broken before unembedding.
- **embedding_parameters** (*dict, optional*) – If provided, parameters are passed to the embedding method as keyword arguments. Overrides any *embedding_parameters* passed to the constructor.
- **return_embedding** (*bool, optional*) – If True, the embedding, chain strength, chain break method and embedding parameters are added to `dimod.SampleSet.info` of the returned sample set. The default behaviour is defined by `return_embedding_default`, which itself defaults to False.
- **warnings** (*WarningAction*, optional) – Defines what warning action to take, if any. See `warnings`. The default behaviour is defined by `warnings_default`, which itself defaults to `IGNORE`
- ****parameters** – Parameters for the sampling method, specified by the child sampler.

Returns `dimod.SampleSet`

Examples

See the example in *EmbeddingComposite*.

dwave.system.composites.AutoEmbeddingComposite.sample_ising

`AutoEmbeddingComposite.`**sample_ising**(*h: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.composites.AutoEmbeddingComposite.sample_qubo

AutoEmbeddingComposite.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

See also:

sample(), *sample_ising()*

EmbeddingComposite

class EmbeddingComposite(*child_sampler*, *find_embedding=<function find_embedding>*, *embedding_parameters=None*, *scale_aware=False*, *child_structure_search=<function child_structure_dfs>*)

Maps problems to a structured sampler.

Automatically minor-embeds a problem into a structured sampler such as a D-Wave system. A new minor-embedding is calculated each time one of its sampling methods is called.

Parameters

- **child_sampler** (`dimod.Sampler`) – A dimod sampler, such as a *DWaveSampler*, that accepts only binary quadratic models of a particular structure.
- **find_embedding** (*function, optional*) – A function *find_embedding(S, T, **kwargs)* where *S* and *T* are edgelists. The function can accept additional keyword arguments. Defaults to *minorminer.find_embedding()*.
- **embedding_parameters** (*dict, optional*) – If provided, parameters are passed to the embedding method as keyword arguments.
- **scale_aware** (*bool, optional, default=False*) – Pass chain interactions to child samplers that accept an *ignored_interactions* parameter.
- **child_structure_search** (*function, optional*) – A function *child_structure_search(sampler)* that accepts a sampler and returns the `dimod.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, num_reads=100)
>>> sampleset.first.energy
-4.5
```

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 behavior 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 behavior for *sample()*'s *warnings* kwarg.

Methods

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

dwave.system.composites.EmbeddingComposite.sample

EmbeddingComposite.**sample**(*bqm*, *chain_strength=None*, *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/mapping/callable, optional*) – Sets the coupling strength between qubits representing variables that form a chain. Mappings should specify the required chain strength for each variable. Callables should accept the BQM and embedding and return a float or mapping. By default, *chain_strength* is calculated with *uniform_torque_compensation()*.
- **chain_break_method** (*function/list, optional*) – Method or methods used to resolve chain breaks. If multiple methods are given, the results are concatenated and a new field called "chain_break_method" specifying the index of the method is appended to the sample set. See *unembed_sampleset()* and *dwave.embedding.chain_breaks*.
- **chain_break_fraction** (*bool, optional, default=True*) – Add a *chain_break_fraction* field to the unembedded response with the fraction of chains broken before unembedding.
- **embedding_parameters** (*dict, optional*) – If provided, parameters are passed to the embedding method as keyword arguments. Overrides any *embedding_parameters* passed to the constructor.
- **return_embedding** (*bool, optional*) – If True, the embedding, chain strength, chain break method and embedding parameters are added to dimod.SampleSet.info of the returned sample set. The default behaviour is defined by *return_embedding_default*, which itself defaults to False.
- **warnings** (*WarningAction*, optional) – Defines what warning action to take, if any. See warnings. The default behaviour is defined by *warnings_default*, which itself defaults to IGNORE
- ****parameters** – Parameters for the sampling method, specified by the child sampler.

Returns dimod.SampleSet

Examples

See the example in *EmbeddingComposite*.

dwave.system.composites.EmbeddingComposite.sample_ising

EmbeddingComposite.**sample_ising**(*h: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.composites.EmbeddingComposite.sample_qubo

EmbeddingComposite.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

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

- **child_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 as *{node: {node neighbours}}*.
- **kwargs** – See the *EmbeddingComposite* class for additional keyword arguments. Note that `find_embedding` and `embedding_parameters` keyword arguments are ignored.

Examples

To embed a triangular problem (a problem with a three-node complete graph, or clique) in the Chimera topology, you need to chain two qubits. This example maps triangular problems to a composed sampler (based on the unstructured `ExactSolver`) with a Chimera unit-cell structure.

```
>>> import dimod
>>> import dwave_networkx as dnx
>>> from dwave.system import FixedEmbeddingComposite
...
>>> c1 = dnx.chimera_graph(1)
>>> embedding = {'a': [0, 4], 'b': [1], 'c': [5]}
>>> structured_sampler = dimod.StructureComposite(dimod.ExactSolver(),
...                                               c1.nodes, c1.edges)
>>> sampler = FixedEmbeddingComposite(structured_sampler, embedding)
>>> sampler.edgelist
[('a', 'b'), ('a', 'c'), ('b', 'c')]
```

Properties

FixedEmbeddingComposite.adjacency	Adjacency structure for the composed sampler.
FixedEmbeddingComposite.child	The child sampler.
FixedEmbeddingComposite.children	
FixedEmbeddingComposite.edgelist	Edges available to the composed sampler.
FixedEmbeddingComposite.nodelist	Nodes available to the composed sampler.
FixedEmbeddingComposite.parameters	
FixedEmbeddingComposite.properties	
FixedEmbeddingComposite.structure	Structure of the structured sampler formatted as a `namedtuple()` where the 3-tuple values are the *nodelist*, *edgelist* and *adjacency* attributes.

dwave.system.composites.FixedEmbeddingComposite.adjacency

`FixedEmbeddingComposite.`**`adjacency`**

Adjacency structure for the composed sampler.

Type dict[variable, set]

dwave.system.composites.FixedEmbeddingComposite.child

FixedEmbeddingComposite.**child**
The child sampler. First sampler in Composite.children.

Type Sampler

dwave.system.composites.FixedEmbeddingComposite.children

FixedEmbeddingComposite.**children = None**

dwave.system.composites.FixedEmbeddingComposite.edgelist

FixedEmbeddingComposite.**edgelist**
Edges available to the composed sampler.

Type list

dwave.system.composites.FixedEmbeddingComposite.nodelist

FixedEmbeddingComposite.**nodelist**
Nodes available to the composed sampler.

Type list

dwave.system.composites.FixedEmbeddingComposite.parameters

FixedEmbeddingComposite.**parameters = None**

dwave.system.composites.FixedEmbeddingComposite.properties

FixedEmbeddingComposite.**properties = None**

dwave.system.composites.FixedEmbeddingComposite.structure

FixedEmbeddingComposite.**structure**
Structure of the structured sampler formatted as a namedtuple() where the 3-tuple values are the *nodelist*, *edgelist* and *adjacency* attributes.

Methods

FixedEmbeddingComposite.sample(bqm, **parameters)	Sample the binary quadratic model.
FixedEmbeddingComposite.sample_ising(h, ...)	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/mapping/callable, optional*) – Sets the coupling strength between qubits representing variables that form a chain. Mappings should specify the required chain strength for each variable. Callables should accept the BQM and embedding and return a float or mapping. By default, *chain_strength* is calculated with *uniform_torque_compensation()*.
- **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: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.composites.FixedEmbeddingComposite.sample_qubo

FixedEmbeddingComposite.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

See also:

sample(), *sample_ising()*

LazyFixedEmbeddingComposite

class LazyFixedEmbeddingComposite(*child_sampler*, *find_embedding=<function find_embedding>*, *embedding_parameters=None*, *scale_aware=False*, *child_structure_search=<function child_structure_dfs>*)

Maps problems to the structure of its first given problem.

This composite reuses the minor-embedding found for its first given problem without recalculating a new minor-embedding for subsequent calls of its sampling methods.

Parameters

- **child_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})
>>> sampler.nodelist  # has structure based on given problem
['a', 'b']
```

Properties

LazyFixedEmbeddingComposite.adjacency	Adjacency structure for the composed sampler.
LazyFixedEmbeddingComposite.edgelist	Edges available to the composed sampler.
LazyFixedEmbeddingComposite.nodelist	Nodes available to the composed sampler.
LazyFixedEmbeddingComposite.parameters	
LazyFixedEmbeddingComposite.properties	
LazyFixedEmbeddingComposite.structure	Structure of the structured sampler formatted as a `namedtuple()` where the 3-tuple values are the *nodelist*, *edgelist* and *adjacency* attributes.

dwave.system.composites.LazyFixedEmbeddingComposite.adjacency

LazyFixedEmbeddingComposite.**adjacency**
Adjacency structure for the composed sampler.

Type dict[variable, set]

dwave.system.composites.LazyFixedEmbeddingComposite.edgelist

LazyFixedEmbeddingComposite.**edgelist**
Edges available to the composed sampler.

Type list

dwave.system.composites.LazyFixedEmbeddingComposite.nodelist

LazyFixedEmbeddingComposite.**nodelist**
Nodes available to the composed sampler.

Type list

dwave.system.composites.LazyFixedEmbeddingComposite.parameters

LazyFixedEmbeddingComposite.**parameters = None**

dwave.system.composites.LazyFixedEmbeddingComposite.properties

LazyFixedEmbeddingComposite.**properties = None**

dwave.system.composites.LazyFixedEmbeddingComposite.structure

LazyFixedEmbeddingComposite.**structure**
Structure of the structured sampler formatted as a `namedtuple()` 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/mapping/callable, optional*) – Sets the coupling strength between qubits representing variables that form a chain. Mappings should specify the required chain strength for each variable. Callables should accept the BQM and embedding and return a float or mapping. By default, *chain_strength* is calculated with *uniform_torque_compensation()*.
- **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: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]]*, ***parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.composites.LazyFixedEmbeddingComposite.sample_qubo

LazyFixedEmbeddingComposite.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]]*, ***parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

See also:

sample(), *sample_ising()*

TilingComposite

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

Composite to tile a small problem across a structured sampler.

Enables parallel sampling on Chimera or Pegasus structured samplers of small problems. The small problem should be defined on a Chimera graph of dimensions `sub_m`, `sub_n`, `t`, or minor-embeddable to such a graph.

Notation *PN* referes to a Pegasus graph consisting of a 3x(N-1)x(N-1) grid of cells, where each unit cell is a bipartite graph with shore of size t, supplemented with odd couplers (see `nice_coordinate` definition in the `dwave_networkx.pegasus_graph()` function). The Advantage QPU supports a P16 Pegasus graph: its qubits may be mapped to a 3x15x15 matrix of unit cells, each of 8 qubits. This code supports tiling of Chimera-structured problems, with an option of additional odd-couplers, onto Pegasus. See also the `dwave_networkx.pegasus_graph()` function.

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. (An earlier quantum computer, the D-Wave 2000Q, supported a C16 Chimera graph: its 2048 qubits were logically mapped into a 16x16 matrix of unit cells of 8 qubits (t=4). See also the `dwave_networkx.chimera_graph()` function.)

A problem that can be minor-embedded in a single chimera unit cell, for example, can therefore be tiled as 3x15x15 duplicates across an Advantage QPU (or, previously, over the unit cells of a D-Wave 2000Q as 16x16

duplicates), subject to solver yield. This enables over 600 (256 for the D-Wave 2000Q) parallel samples per read.

Parameters

- **sampler** (*dimod.Sampler*) – Structured dimod sampler such as a *DWaveSampler()*.
- **sub_m** (*int*) – Minimum number of Chimera unit cell rows required for minor-embedding a single instance of the problem.
- **sub_n** (*int*) – Minimum number of Chimera unit cell columns required for minor-embedding a single instance of the problem.
- **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 import DWaveSampler, EmbeddingComposite
>>> from dwave.system import TilingComposite
...
>>> sampler = EmbeddingComposite(TilingComposite(DWaveSampler(), 1, 1, 4))
>>> Q = {(1, 1): -1, (1, 2): 2, (2, 1): 0, (2, 2): -1}
>>> sampleset = sampler.sample_qubo(Q)
>>> len(sampleset)> 1
True
```

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

Properties

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.child	The child sampler.
TilingComposite.children	The single wrapped structured sampler.
TilingComposite.edgelist	List of active couplers for the structured solver.
TilingComposite.embeddings	Embeddings into each available tile on the structured solver.
TilingComposite.nodelist	List of active qubits for the structured solver.
TilingComposite.num_tiles	Number of tiles available for replicating the problem.
TilingComposite.parameters	Parameters in the form of a dict.
TilingComposite.properties	Properties in the form of a dict.
TilingComposite.structure	Structure of the structured sampler formatted as a namedtuple() where the 3-tuple values are the *nodelist*, *edgelist* and *adjacency* attributes.

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.

dwave.system.composites.TilingComposite.child

TilingComposite.**child**
The child sampler. First sampler in Composite.children.

Type Sampler

dwave.system.composites.TilingComposite.children

TilingComposite.**children = None**
The single wrapped structured sampler.

Type list

dwave.system.composites.TilingComposite.edgelist

TilingComposite.**edgelist = None**
List of active couplers for the structured solver.

Type list

dwave.system.composites.TilingComposite.embeddings

TilingComposite.**embeddings = []**
Embeddings into each available tile on the structured solver.

Type list

dwave.system.composites.TilingComposite.nodelist

TilingComposite.**nodelist = None**
List of active qubits for the structured solver.

Type list

dwave.system.composites.TilingComposite.num_tiles

TilingComposite.**num_tiles**
Number of tiles available for replicating the problem.

dwave.system.composites.TilingComposite.parameters

`TilingComposite.`**`parameters = None`**
Parameters in the form of a dict.

Type dict[str, list]

dwave.system.composites.TilingComposite.properties

`TilingComposite.`**`properties = None`**
Properties in the form of a dict.

Type dict

dwave.system.composites.TilingComposite.structure

`TilingComposite.`**`structure`**
Structure of the structured sampler formatted as a `namedtuple()` where the 3-tuple values are the *`nodelist`*, *`edgelist`* and *`adjacency`* attributes.

Methods

`TilingComposite.sample`(bqm, **kwargs)	Sample from the specified binary quadratic model.
`TilingComposite.sample_ising`(h, Union[float, ...)	Sample from an Ising model using the implemented sample method.
`TilingComposite.sample_qubo`(Q, Hashable], ...)	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 import DWaveSampler, EmbeddingComposite
>>> from dwave.system import TilingComposite
...
>>> sampler = EmbeddingComposite(TilingComposite(DWaveSampler(), 1, 1, 4))
```

(continues on next page)

(continued from previous page)

```
>>> sampleset = sampler.sample_ising({},{('a', 'b'): 1})
>>> len(sampleset) > 1
True
```

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

dwave.system.composites.TilingComposite.sample_ising

TilingComposite.**sample_ising**(*h: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.composites.TilingComposite.sample_qubo

TilingComposite.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

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 an Ising problem to a D-Wave solver. This simple three-variable problem is manually minor-embedded such that variables `a` and `b` are represented by single qubits while variable `c` is represented by a four-qubit chain. The chain strength is set to the maximum allowed found from querying the solver's extended J range. The minor embedding shown below was for an execution of this example on a particular Advantage system; select a suitable embedding for the QPU you use.

```
>>> from dwave.system import DWaveSampler, VirtualGraphComposite
...
>>> h = {'a': 1, 'b': -1}
>>> J = {('b', 'c'): -1, ('a', 'c'): -1}
...
>>> qpu = DWaveSampler()
```

(continues on next page)

(continued from previous page)

```
>>> embedding = {'a': [2656], 'c': [2641, 2642, 2643, 2644], 'b': [2659]}
>>> qpu.properties['extended_j_range']
[-2.0, 1.0]
>>> # Sample using VirtualGraphComposite
>>> sampler = VirtualGraphComposite(qpu, embedding, chain_strength=2) # doctest:␣
↪+SKIP
>>> sampleset = sampler.sample_ising(h, J, num_reads=100) # doctest: +SKIP
>>> print(sampleset)    # doctest: +SKIP
    a  b  c energy num_oc. chain_.
0 +1 +1 +1   -2.0      21     0.0
1 -1 +1 +1   -2.0      66     0.0
2 -1 -1 -1   -2.0       8     0.0
3 -1 +1 -1   -2.0       5     0.0
['SPIN', 4 rows, 100 samples, 3 variables]
```

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

Properties

VirtualGraphComposite.adjacency	Adjacency structure for the composed sampler.
VirtualGraphComposite.child	The child sampler.
VirtualGraphComposite.children	
VirtualGraphComposite.edgelist	Edges available to the composed sampler.
VirtualGraphComposite.nodelist	Nodes available to the composed sampler.
VirtualGraphComposite.parameters	
VirtualGraphComposite.properties	
VirtualGraphComposite.structure	Structure of the structured sampler formatted as a `namedtuple()` where the 3-tuple values are the *nodelist*, *edgelist* and *adjacency* attributes.

dwave.system.composites.VirtualGraphComposite.adjacency

VirtualGraphComposite.**adjacency**

Adjacency structure for the composed sampler.

Type dict[variable, set]

dwave.system.composites.VirtualGraphComposite.child

VirtualGraphComposite.**child**

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

Type `Sampler`

dwave.system.composites.VirtualGraphComposite.children

VirtualGraphComposite.**children = None**

dwave.system.composites.VirtualGraphComposite.edgelist

VirtualGraphComposite.**edgelist**
Edges available to the composed sampler.

Type list

dwave.system.composites.VirtualGraphComposite.nodelist

VirtualGraphComposite.**nodelist**
Nodes available to the composed sampler.

Type list

dwave.system.composites.VirtualGraphComposite.parameters

VirtualGraphComposite.**parameters = None**

dwave.system.composites.VirtualGraphComposite.properties

VirtualGraphComposite.**properties = None**

dwave.system.composites.VirtualGraphComposite.structure

VirtualGraphComposite.**structure**
Structure of the structured sampler formatted as a `namedtuple()` 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, ...)	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.

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

dwave.system.composites.VirtualGraphComposite.sample_ising

VirtualGraphComposite.**sample_ising**(*h: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.composites.VirtualGraphComposite.sample_qubo

VirtualGraphComposite.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

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.

Examples

This example runs 100 reverse anneals each from two initial states on a problem constructed by setting random ±1 values on a clique (complete graph) of 15 nodes, minor-embedded on a D-Wave system using the *DWaveCliqueSampler* sampler.

```
>>> import dimod
>>> from dwave.system import DWaveCliqueSampler, ReverseBatchStatesComposite
...
>>> sampler = DWaveCliqueSampler()      # doctest: +SKIP
>>> sampler_reverse = ReverseBatchStatesComposite(sampler)    # doctest: +SKIP
>>> schedule = [[0.0, 1.0], [10.0, 0.5], [20, 1.0]]
...
>>> bqm = dimod.generators.ran_r(1, 15)
>>> init_samples = [{i: -1 for i in range(15)}, {i: 1 for i in range(15)}]
>>> sampleset = sampler_reverse.sample(bqm,
...                                    anneal_schedule=schedule,
...                                    initial_states=init_samples,
...                                    num_reads=100,
...                                    reinitialize_state=True)    # doctest: +SKIP
```

Properties

ReverseBatchStatesComposite.child	The child sampler.
ReverseBatchStatesComposite.children	List of child samplers that that are used by this composite.
ReverseBatchStatesComposite.parameters	Parameters as a dict, where keys are keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
ReverseBatchStatesComposite.properties	Properties as a dict containing any additional information about the sampler.

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.parameters

ReverseBatchStatesComposite.**parameters**
Parameters as a dict, where keys are keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

dwave.system.composites.ReverseBatchStatesComposite.properties

ReverseBatchStatesComposite.**properties**
Properties as a dict containing any additional information about the sampler.

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*, *initial_states=None*, *initial_states_generator='random'*, *num_reads=None*, *seed=None*, ***parameters*)
Sample the binary quadratic model using reverse annealing from multiple initial states.

Parameters

- **bqm** (dimod.BinaryQuadraticModel) – Binary quadratic model to be sampled from.
- **initial_states** (*samples-like, optional, default=None*) – One or more samples, each defining an initial state for all the problem variables. If fewer than *num_reads* initial states are defined, additional values are generated as specified by *initial_states_generator*. See dimod.as_samples() for a description of "samples-like".
- **initial_states_generator** (*{'none', 'tile', 'random'}, optional, default='random'*) – Defines the expansion of *initial_states* if fewer than *num_reads* are specified:
  - **"none":** If the number of initial states specified is smaller than *num_reads*, raises ValueError.
  - **"tile":** Reuses the specified initial states if fewer than *num_reads* or truncates if greater.
  - **"random":** Expands the specified initial states with randomly generated states if fewer than *num_reads* or truncates if greater.
- **num_reads** (*int, optional, default=len(initial_states) or 1*) – Equivalent to number of desired initial states. If greater than the number of provided initial states, additional states will be generated. If not provided, it is selected to match the length of *initial_states*. If *initial_states* is not provided, *num_reads* defaults to 1.
- **seed** (*int (32-bit unsigned integer), optional*) – Seed to use for the PRNG. Specifying a particular seed with a constant set of parameters produces identical

results. If not provided, a random seed is chosen.

- ****parameters** – Parameters for the sampling method, specified by the child sampler.

Returns dimod.SampleSet that has initial_state field.

Examples

This example runs 100 reverse anneals each from two initial states on a problem constructed by setting random ±1 values on a clique (complete graph) of 15 nodes, minor-embedded on a D-Wave system using the *DWaveCliqueSampler* sampler.

```
>>> import dimod
>>> from dwave.system import DWaveCliqueSampler, ReverseBatchStatesComposite
...
>>> sampler = DWaveCliqueSampler()        # doctest: +SKIP
>>> sampler_reverse = ReverseBatchStatesComposite(sampler)    # doctest: +SKIP
>>> schedule = [[0.0, 1.0], [10.0, 0.5], [20, 1.0]]
...
>>> bqm = dimod.generators.ran_r(1, 15)
>>> init_samples = [{i: -1 for i in range(15)}, {i: 1 for i in range(15)}]
>>> sampleset = sampler_reverse.sample(bqm,
...                                    anneal_schedule=schedule,
...                                    initial_states=init_samples,
...                                    num_reads=100,
...                                    reinitialize_state=True)  # doctest: +SKIP
```

dwave.system.composites.ReverseBatchStatesComposite.sample_ising

ReverseBatchStatesComposite.**sample_ising**(*h: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.composites.ReverseBatchStatesComposite.sample_qubo

ReverseBatchStatesComposite.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]]*, ***parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

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.

Examples

This example runs 100 reverse anneals each for three schedules on a problem constructed by setting random ±1 values on a clique (complete graph) of 15 nodes, minor-embedded on a D-Wave system using the *DWaveCliqueSampler* sampler.

```
>>> import dimod
>>> from dwave.system import DWaveCliqueSampler, ReverseAdvanceComposite
...
>>> sampler = DWaveCliqueSampler()      # doctest: +SKIP
>>> sampler_reverse = ReverseAdvanceComposite(sampler)     # doctest: +SKIP
>>> schedule = [[[0.0, 1.0], [t, 0.5], [20, 1.0]] for t in (5, 10, 15)]
...
>>> bqm = dimod.generators.ran_r(1, 15)
>>> init_samples = {i: -1 for i in range(15)}
>>> sampleset = sampler_reverse.sample(bqm,
...                                    anneal_schedules=schedule,
...                                    initial_state=init_samples,
...                                    num_reads=100,
...                                    reinitialize_state=True)      # doctest:␣
↪+SKIP
```

Properties

ReverseAdvanceComposite.child	The child sampler.
ReverseAdvanceComposite.children	List of child samplers that that are used by this composite.
ReverseAdvanceComposite.parameters	Parameters as a dict, where keys are keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
ReverseAdvanceComposite.properties	Properties as a dict containing any additional information about the sampler.

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.parameters

ReverseAdvanceComposite.**parameters**
Parameters as a dict, where keys are keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

dwave.system.composites.ReverseAdvanceComposite.properties

ReverseAdvanceComposite.**properties**
Properties as a dict containing any additional information about the sampler.

Methods

ReverseAdvanceComposite.sample(bqm[, ...])	Sample the binary quadratic model using reverse annealing along a given set of anneal schedules.
ReverseAdvanceComposite.sample_ising(h, ...)	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, optional, default=[[[0, 1], [1, 0.35], [9, 0.35], [10, 1]]]*) – Anneal schedules in order of submission. Each schedule is formatted as a list of [time, s] pairs, in which time is in microseconds and s is the normalized persistent current in the range [0,1].
- **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.

Examples

This example runs 100 reverse anneals each for three schedules on a problem constructed by setting random ±1 values on a clique (complete graph) of 15 nodes, minor-embedded on a D-Wave system using the *DWaveCliqueSampler* sampler.

```
>>> import dimod
>>> from dwave.system import DWaveCliqueSampler, ReverseAdvanceComposite
...
>>> sampler = DWaveCliqueSampler()          # doctest: +SKIP
>>> sampler_reverse = ReverseAdvanceComposite(sampler)     # doctest: +SKIP
>>> schedule = [[[0.0, 1.0], [t, 0.5], [20, 1.0]] for t in (5, 10, 15)]
...
>>> bqm = dimod.generators.ran_r(1, 15)
>>> init_samples = {i: -1 for i in range(15)}
>>> sampleset = sampler_reverse.sample(bqm,
...                                    anneal_schedules=schedule,
...                                    initial_state=init_samples,
...                                    num_reads=100,
...                                    reinitialize_state=True)  # doctest: +SKIP
```

dwave.system.composites.ReverseAdvanceComposite.sample_ising

ReverseAdvanceComposite.**sample_ising**(*h: Union[Mapping[Hashable, Union[float, numpy.floating, numpy.integer]], Sequence[Union[float, numpy.floating, numpy.integer]]], J: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from an Ising model using the implemented sample method.

This method is inherited from the Sampler base class.

Converts the Ising model into a BinaryQuadraticModel and then calls *sample()*.

Parameters

- **h** – Linear biases of the Ising problem. If a list, indices are the variable labels.
- **J** – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from the Ising model.

See also:

sample(), *sample_qubo()*

dwave.system.composites.ReverseAdvanceComposite.sample_qubo

ReverseAdvanceComposite.**sample_qubo**(*Q: Mapping[Tuple[Hashable, Hashable], Union[float, numpy.floating, numpy.integer]], **parameters*) → dimod.sampleset.SampleSet

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the quadratic unconstrained binary optimization (QUBO) into a `BinaryQuadraticModel` and then calls *sample()*.

Parameters

- **Q** – Coefficients of a QUBO problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns: Samples from a QUBO.

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.

For an introduction to minor-embedding, see Minor-Embedding.

Generators

Tools for finding embeddings.

Generic

minorminer is a heuristic tool for minor embedding: given a minor and target graph, it tries to find a mapping that embeds the minor into the target.

minorminer.find_embedding	Heuristically attempt to find a minor-embedding of source graph S into a target graph T.

minorminer.find_embedding

find_embedding()

Heuristically attempt to find a minor-embedding of source graph S into a target graph T.

Parameters

- **S** (*iterable/NetworkX Graph*) – The source graph as an iterable of label pairs representing the edges, or a NetworkX Graph.
- **T** (*iterable/NetworkX Graph*) – The target graph as an iterable of label pairs representing the edges, or a NetworkX Graph.
- ****params** (*optional*) – See below.

Returns

When the optional parameter `return_overlap` is False (the default), the function 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` is True, the function 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, the function 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 (int, optional, default=10): 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.

random_seed (int, optional, default=None): Seed for the random number generator. If None, seed is set by `os.urandom()`.

timeout (int, optional, default=1000): Algorithm gives up after timeout seconds.

max_beta (double, optional, max_beta=None): Qubits are assigned weight according to a formula (beta^n) where n is the number of chains containing that qubit. This value should never be less than or equal to 1. If None, `max_beta` is effectively infinite.

tries (int, optional, default=10): Number of restart attempts before the algorithm stops. On D-WAVE 2000Q, a typical restart takes between 1 and 60 seconds.

inner_rounds (int, optional, default=None): The algorithm takes at most this many iterations between restart attempts; restart attempts are typically terminated due to `max_no_improvement`. If None, `inner_rounds` is effectively infinite.

chainlength_patience (int, optional, default=10): Maximum number of failed iterations to improve chain lengths 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.

max_fill (int, optional, default=None): Restricts the number of chains that can simultaneously incorporate the same qubit during the search. Values above 63 are treated as 63. If None, `max_fill` is effectively infinite.

threads (int, optional, default=1): 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. Value must be greater than 1.

return_overlap (bool, optional, default=False): This function returns an embedding, regardless of whether or not qubits are used by multiple variables. `return_overlap` determines the function's return value. If True, a 2-tuple is returned, in which the first element is the embedding and the second element is a bool representing the embedding validity. If False, only an embedding is returned.

skip_initialization (bool, optional, default=False): 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 chain length improvement phase. Another good source of semi-valid embeddings is the output of this function with the `return_overlap` parameter enabled.

verbose (int, optional, default=0): Level of output verbosity.

When set to 0: 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: Output the information for lower levels and also report 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 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: Number of qubits that have max overfill.

max chain length: Largest number of qubits representing a single variable.

num max chains: Number of variables that have max chain size.

interactive (bool, optional, default=False): If *logging* is None or False, the verbose output will be printed to stdout/stderr as appropriate, and keyboard interrupts will stop the embedding process and the current state will be returned to the user. Otherwise, output will be directed to the logger `logging.getLogger(minorminer.__name__)` and keyboard interrupts will be propagated back to the user. Errors will use `logger.error()`, verbosity levels 1 through 3 will use `logger.info()` and level 4 will use `logger.debug()`.

initial_chains (dict, optional): 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 chain length of a valid embedding or to reduce overlap in a semi-valid embedding (see `skip_initialization`) previously returned

by the algorithm. Missing or empty entries are ignored. Each value in the dictionary is a list of qubit labels.

fixed_chains (dict, optional): Fixed chains inserted into an embedding before the initialization pass. As the algorithm proceeds, these chains are not allowed to change, and the qubits used by these chains are not used by other chains. Missing or empty entries are ignored. Each value in the dictionary is a list of qubit labels.

restrict_chains (dict, optional): Throughout the algorithm, it is guaranteed that chain[i] is a subset of `restrict_chains[i]` for each i, except those with missing or empty entries. Each value in the dictionary is a list of qubit labels.

suspend_chains (dict, optional): This is a metafeature that is only implemented in the Python interface. `suspend_chains[i]` is an iterable of iterables; for example, `suspend_chains[i] = [blob_1, blob_2]`, with each *blob_j* an iterable of target node labels.

This enforces the following:

```
for each suspended variable i,
    for each blob_j in the suspension of i,
        at least one qubit from blob_j will be contained in the chain for␣
↪i
```

We accomplish this through the following problem transformation for each iterable *blob_j* in `suspend_chains[i]`,

- Add an auxiliary node *Zij* to both source and target graphs
- Set *fixed_chains[Zij] = [Zij]*
- Add the edge *(i,Zij)* to the source graph
- Add the edges *(q,Zij)* to the target graph for each *q* in *blob_j*

Chimera

Minor-embedding in Chimera-structured target graphs.

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=None*, *n=None*, *t=None*, *target_edges=None*, *target_graph=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, optional, default=None*) – 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()`.
- **target_graph** (*networkx.Graph*) – A Chimera graph constructed by `chimera_graph()`.

Returns An embedding mapping a clique to the Chimera lattice.

Return type dict

Note: Either target_edges or target_graph must be None. If both are None, a graph with perfect yield is assumed from the parameters m, n, t. If target_edges is not None, at least m must not be None.

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=None*, *n=None*, *t=None*, *target_edges=None*, *target_graph=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, optional, default=None*) – 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()`.
- **target_graph** (*networkx.Graph*) – A Chimera graph constructed 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

Note: Either target_edges or target_graph must be None. If both are None, a graph with perfect yield is assumed from the parameters m, n, t. If target_edges is not None, at least m must not be None.

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.
pegasus.find_biclique_embedding(a, b[, m, ...])	Find an embedding for a biclique 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]}
```

dwave.embedding.pegasus.find_biclique_embedding

find_biclique_embedding(*a*, *b*, *m=None*, *target_graph=None*)

Find an embedding for a biclique in a Pegasus 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 Pegasus 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 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

A 2-tuple containing:

dict: An embedding mapping the left shore of the biclique to the Pegasus lattice.

dict: An embedding mapping the right shore of the biclique to the Pegasus lattice.

Return type tuple

Examples

This example finds an embedding for an alphanumerically labeled biclique in a small Pegasus graph

```
>>> from dwave.embedding.pegasus import find_biclique_embedding
...
>>> left, right = find_biclique_embedding(['a', 'b', 'c'], ['d', 'e'], 2)
>>> print(left, right)  # doctest: +SKIP
{'a': [40], 'b': [41], 'c': [42]} {'d': [4], 'e': [5]}
```

Zephyr

Minor-embedding in Zephyr-structured target graphs.

zephyr.find_clique_embedding(k[, m, ...])	Find an embedding for a clique in a Zephyr graph.
zephyr.find_biclique_embedding(a, b[, m, ...])	Find an embedding for a biclique in a Zephyr graph.

dwave.embedding.zephyr.find_clique_embedding

find_clique_embedding(*k*, *m=None*, *target_graph=None*)

Find an embedding for a clique in a Zephyr graph.

Given a clique (fully connected graph) and target Zephyr graph, attempts to find an embedding.

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 Zephyr graph. Required to generate an `m`-by-`m` Zephyr graph when `target_graph` is None.
- **target_graph** (`networkx.Graph`) – A Zephyr 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 zephyr coordinates.

Return type dict

Examples

This example finds an embedding for a K_5 complete graph in a 2-by-2 Zephyr graph.

```
>>> from dwave.embedding.zephyr import find_clique_embedding
>>> find_clique_embedding(5, 2)
{0: (16, 96), 1: (18, 98), 2: (20, 100), 3: (22, 102), 4: (24, 104)}
```

dwave.embedding.zephyr.find_biclique_embedding

find_biclique_embedding(*a*, *b*, *m=None*, *target_graph=None*)

Find an embedding for a biclique in a Zephyr 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 Zephyr graph, attempts to find an embedding.

Parameters

- **a** (*int/iterable*) – Describes the left shore of the biclique to embed. If `a` is an integer, the left shore will be labelled [0, a-1]. If `a` is an iterable, the left shore will be labelled by `a`.
- **b** (*int/iterable*) – Describes the right shore of the biclique to embed. If `b` is an integer and `a` is an iterable, the right shore will be labelled [0, b-1]. If both `a` and `b` are integers, the right shore will be labelled [a, a+b-1]. If `b` is an iterable, the right shore will be labelled by `b`.
- **m** (*int*) – Number of tiles in a row of a square Zephyr graph. Required to generate an `m`-by-`m` Zephyr graph when `target_graph` is None.
- **target_graph** (`networkx.Graph`) – A Zephyr graph. Required when `m` is None.

Returns

A 2-tuple containing:

dict: An embedding mapping the left shore of the biclique to the Zephyr lattice.

dict: An embedding mapping the right shore of the biclique to the Zephyr lattice.

Return type tuple

Examples

This example finds an embedding for an alphanumerically labeled biclique in a 2x2 Zephyr graph.

```
>>> from dwave.embedding.zephyr import find_biclique_embedding
>>> left, right = find_biclique_embedding(['a', 'b', 'c'], ['d', 'e'], 2)
>>> print(left, right)
{'a': (0,), 'b': (4,), 'c': (8,)} {'d': (80,), 'e': (84,)}
```

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 sample set.

dwave.embedding.embed_bqm

embed_bqm(*source_bqm*, *embedding=None*, *target_adjacency=None*, *chain_strength=None*, *smear_vartype=None*)

Embed a binary quadratic model onto a target graph.

Parameters

- **source_bqm** (BinaryQuadraticModel) – Binary quadratic model to embed.
- **embedding** (dict/*EmbeddedStructure*) – 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. Alternately, an EmbeddedStructure object produced by, for example, EmbeddedStructure(target_adjacency.edges(), embedding). If embedding is a dict, target_adjacency must be provided.
- **target_adjacency** (dict/networkx.Graph, optional) – 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. This should be omitted if and only if embedding is an EmbeddedStructure object.
- **chain_strength** (*float/mapping/callable, optional*) – Sets the coupling strength between qubits representing variables that form a chain. Mappings should specify the required chain strength for each variable. Callables should accept the BQM and embedding and return a float or mapping. By default, *chain_strength* is calculated with *uniform_torque_compensation()*.
- **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)]
-1.9996979771955565
>>> print(target_bqm.quadratic)  # doctest: +SKIP
{(0, 1): 1.0, (0, 3): 1.0, (1, 2): 1.0, (2, 3): -1.9996979771955565}
```

See also:

embed_ising(), *embed_qubo()*

dwave.embedding.embed_ising

embed_ising(*source_h*, *source_J*, *embedding*, *target_adjacency*, *chain_strength=None*)

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/mapping/callable, optional*) – Sets the coupling strength between qubits representing variables that form a chain. Mappings should specify the required chain strength for each variable. Callables should accept the BQM and embedding and return a float or mapping. By default, *chain_strength* is calculated with *uniform_torque_compensation()*.

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=None*)

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/mapping/callable, optional*) – Sets the coupling strength between qubits representing variables that form a chain. Mappings should specify the required chain strength for each variable. Callables should accept the BQM and embedding and return a float or mapping. By default, *chain_strength* is calculated with *uniform_torque_compensation()*.

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 sample 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** (BinaryQuadraticModel) – Source BQM.
- **chain_break_method** (*function/list, optional*) – Method or methods used to resolve chain breaks. If multiple methods are given, the results are concatenated and a new field called "chain_break_method" specifying the index of the method is appended to the sample set. Defaults to *majority_vote()*. 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
...
```

(continues on next page)

(continued from previous page)

```
>>> 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.

```
>>> from dwave.embedding import diagnose_embedding
>>> import networkx as nx
>>> source = nx.complete_graph(3)
>>> target = nx.cycle_graph(4)
>>> embedding = {0: [2], 1: [1, 'a'], 2: [2, 3]}
>>> diagnosis = diagnose_embedding(embedding, source, target)
>>> for problem in diagnosis:  # doctest: +SKIP
...     print(problem)
```

(continues on next page)

(continued from previous page)

```
(<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 Strength

Utility functions for calculating chain strength.

Examples

This example uses *uniform_torque_compensation()*, given a prefactor of 2, to calculate a chain strength that `EmbeddingComposite` then uses.

```
>>> from functools import partial
>>> from dwave.system import EmbeddingComposite, DWaveSampler
>>> from dwave.embedding.chain_strength import uniform_torque_compensation
...
>>> Q = {(0,0): 1, (1,1): 1, (2,3): 2, (1,2): -2, (0,3): -2}
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> # partial() can be used when the BQM or embedding is not accessible
>>> chain_strength = partial(uniform_torque_compensation, prefactor=2)
>>> sampleset = sampler.sample_qubo(Q, chain_strength=chain_strength, return_
↪embedding=True)
>>> sampleset.info['embedding_context']['chain_strength']
1.224744871391589
```

chain_strength.uniform_torque_compensation(bqm[, embedding, ...])		Chain strength that attempts to compensate for torque that would break the chain.
chain_strength.scaled(bqm[, embedding, ...])		Chain strength that is scaled to the problem bias range.

dwave.embedding.chain_strength.uniform_torque_compensation

uniform_torque_compensation(*bqm*, *embedding=None*, *prefactor=1.414*)

Chain strength that attempts to compensate for torque that would break the chain.

The RMS of the problem's quadratic biases is used for calculation.

Parameters

- **bqm** (`BinaryQuadraticModel`) – A binary quadratic model.
- **embedding** (dict/*EmbeddedStructure*, default=None) – Included to satisfy the *chain_strength* callable specifications for *embed_bqm*.
- **prefactor** (*float, optional, default=1.414*) – Prefactor used for scaling. For non-pathological problems, the recommended range of prefactors to try is [0.5, 2].

Returns The chain strength, or 1 if chain strength is not applicable.

Return type float

dwave.embedding.chain_strength.scaled

scaled(*bqm*, *embedding=None*, *prefactor=1.0*)

Chain strength that is scaled to the problem bias range.

Parameters

- **bqm** (`BinaryQuadraticModel`) – A binary quadratic model.
- **embedding** (dict/*EmbeddedStructure*, default=None) – Included to satisfy the *chain_strength* callable specifications for *embed_bqm*.
- **prefactor** (*float, optional, default=1.0*) – Prefactor used for scaling.

Returns The chain strength, or 1 if chain strength is not applicable.

Return type float

Chain-Break Resolution

Unembedding samples with broken chains.

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)
>>> print(idx)
[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)
>>> print(unembedded)
[[1 0]
 [1 1]]
>>> print(idx)
[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])
```

__init__(*bqm*, *embedding*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(bqm, embedding)	Initialize self.

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.

Classes

class EmbeddedStructure(*target_edges*, *embedding*)
Processes an embedding and a target graph to collect target edges into those within individual chains, and those that connect chains. This is used elsewhere to embed binary quadratic models into the target graph.

Parameters

- **target_edges** (*iterable[edge]*) – An iterable of edges in the target graph. Each edge should be an iterable of 2 hashable objects.

- **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.

This class is a dict, and acts as an immutable duplicate of embedding.

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 part of a particular Advantage quantum computer's working graph.

```
>>> import dwave_networkx as dnx
>>> from dwave.system import DWaveSampler, common_working_graph
...
>>> sampler = DWaveSampler(solver={'topology__type': 'pegasus'})
>>> P3 = dnx.pegasus_graph(3)
>>> p3_working_graph = common_working_graph(P3, sampler.adjacency)
```

coupling_groups(hardware_graph)	Generate groups of couplers for which a limit on total coupling applies for each group.

dwave.system.coupling_groups.coupling_groups

coupling_groups (*hardware_graph*)

Generate groups of couplers for which a limit on total coupling applies for each group.

Parameters **hardware_graph** (networkx.Graph) – The hardware graph of a QPU. Note that only Zephyr graphs have coupling groups.

Yields Lists of tuples, where each list is a group of couplers in `hardware_graph`.

Temperature Utilities

The following effective temperature estimators are provided:

- Maximum pseudo-likelihood is an efficient estimator for the temperature describing a classical Boltzmann distribution P(x) = exp(-H(x)/T)/Z(T) given samples from that distribution, where H(x) is the classical energy function. The following links describe features of the estimator in application to equilibrium distribution drawn from binary quadratic models and non-equilibrium distributions generated by annealing: https://www.jstor.org/stable/25464568 https://doi.org/10.3389/fict.2016.00023
- An effective temperature can be inferred assuming freeze-out during the anneal at s=t/t_a, an annealing schedule, and a device physical temperature. Necessary device-specific properties are published for online solvers: https://docs.dwavesys.com/docs/latest/doc_physical_properties.html

effective_field(bqm[, samples, ...])	Returns the effective field for all variables and all samples.
maximum_pseudolikelihood_temperature([bqm, ...])	Returns a sampling-based temperature estimate.
freezeout_effective_temperature(freezeout_B, ...)	Provides an effective temperature as a function of freezeout information.
fast_effective_temperature([sampler, ...])	Provides an estimate to the effective temperature, T, of a sampler.

dwave.system.temperatures.effective_field

effective_field(*bqm*, *samples=None*, *current_state_energy=False*) -> (*<class 'numpy.ndarray'>*, *<class 'list'>*)

Returns the effective field for all variables and all samples.

The effective field with `current_state_energy = False` is the energy attributable to setting a variable to value 1, conditioned on fixed values for all neighboring variables (relative to exclusion of the variable, and associated energy terms, from the problem).

The effective field with `current_state_energy = True` is the energy gained by flipping the variable state against its current value (from say -1 to 1 in the Ising case, or 0 to 1 in the QUBO case). A positive value indicates that the energy can be decreased by flipping the variable, hence the variable is in a locally excited state. If all values are negative (positive) within a sample, that sample is a local minima (maxima).

Any BQM can be converted to an Ising model with

$$H(s) = Constant + \sum_i h_i s_i + 0.5 \sum_{i,j} J_{i,j} s_i s_j$$

with unique values of J (symmetric) and h. The sample dependent effect field on variable i, $f_i(s)$, is then defined

if current_state_energy == False:

$$f_i(s) = h_i + \sum_j J_{i,j} s_j$$

else:

$$f_i(s) = 2 s_i [h_i + \sum_j J_{i,j} s_j]$$

Parameters

- **bqm** (dimod.BinaryQuadraticModel) – Binary quadratic model.
- **samples** (samples_like or SampleSet,optional) – A collection of raw samples. *samples_like* is an extension of NumPy's array like structure. See dimod.sampleset.as_samples(). By default, a single sample with all +1 assignments is used.
- **current_state_energy** (*bool, optional, default=False*) – By default, returns the effective field (the energy contribution associated to a state assignment of 1). When set to True, returns the energy lost in flipping the value of each variable. Note current_state_energy is typically negative for positive temperature samples, meaning energy is not decreased by flipping the spin against its current assignment.

Returns A Tuple of the effective_fields, and the variable labels. Effective fields are returned as a numpy.ndarray. Rows index samples, and columns index variables in the order returned by variable labels.

Return type samples_like

Examples

For a ferromagnetic Ising chain $H = -0.5 \sum_i s_i s_{i+1}$ and for a ground state sample (all +1), the energy lost when flipping any spin is equal to the number of couplers frustrated: -2 in the center of the chain (variables 1,2,..,N-2), and -1 at the end (variables 0 and N-1).

```
>>> import dimod
>>> import numpy as np
>>> from dwave.system.temperatures import effective_field
>>> N = 5
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {(i,i+1) : -0.5 for i in␣
↪range(N-1)})
>>> var_labels = list(range(N))
>>> samples = (np.ones(shape=(1,N)), var_labels)
>>> E = effective_field(bqm,samples,current_state_energy=True)
>>> print('Cost to flip spin against current assignment', E)
Cost to flip spin against current assignment (array([[-1., -2., -2., -2., -1.]]),␣
↪[0, 1, 2, 3, 4])
```

dwave.system.temperatures.maximum_pseudolikelihood_temperature

maximum_pseudolikelihood_temperature(*bqm=None, sampleset=None, site_energy=None, num_bootstrap_samples=0, seed=None, T_guess=None, optimize_method='bisect', T_bracket=(0.001, 1000)*) → Tuple[float, numpy.ndarray]

Returns a sampling-based temperature estimate.

The temperature T parameterizes the Boltzmann distribution as $P(x) = \exp(-H(x)/T)/Z(T)$, where $P(x)$ is a probability over a state space, $H(x)$ is the energy function (BQM) and $Z(T)$ is a normalization. Given a sample set (S), a temperature estimate establishes the temperature that is most likely to have produced the sample set. An effective temperature can be derived from a sample set by considering the rate of excitations only. A maximum-pseudo-likelihood (MPL) estimator considers local excitations only, which are sufficient to establish a temperature efficiently (in compute time and number of samples). If the BQM consists of uncoupled variables then the estimator is equivalent to a maximum likelihood estimator.

The effective MPL temperature is defined by the solution T to

$$0 = \sum_i \sum_{s \in S} f_i(s) \exp(f_i(s)/T),$$

where f is the energy lost in flipping spin i against its current assignment (the effective field).

The problem is a convex root solving problem, and is solved with SciPy optimize.

If the distribution is not Boltzmann with respect to the BQM provided, as may be the case for heuristic samplers (such as annealers), the temperature estimate can be interpreted as characterizing only a rate of local excitations. In the case of sample sets obtained from D-Wave annealing quantum computers the temperature can be identified with a physical temperature via a late-anneal freeze-out phenomena.

Parameters

- **bqm** (dimod.BinaryQuadraticModel, optional) – Binary quadratic model describing sample distribution. If bqm and site_energy are both None, then by default 100 samples are drawn using *DWaveSampler*, with bqm defaulted as described.
- **sampleset** (dimod.SampleSet, optional) – A set of samples, assumed to be fairly sampled from a Boltzmann distribution characterized by bqm.
- **site_energy** (*samples_like, optional*) – A Tuple of effective fields and site labels. Derived from the bqm and sampleset if not provided.
- **num_bootstrap_samples** (*int, optional, default=0*) – Number of bootstrap estimators to calculate.
- **seed** (*int, optional*) – Seeds the bootstrap method (if provided) allowing reproducibility of the estimators.
- **T_guess** (*float, optional*) – User approximation to the effective temperature, must be a positive scalar value. Seeding the root-search method can enable faster convergence. By default, T_guess is ignored if it falls outside the range of T_bracket.
- **optimize_method** (*str,optional,default='bisect'*) – SciPy method used for optimization. Options are 'bisect' and None (the default SciPy optimize method).
- **T_bracket** (*list or Tuple of 2 floats, optional, default=(0.001,1000)*) – If excitations are absent, temperature is defined as zero, otherwise this defines the range of Temperatures over which to attempt a fit when using the 'bisect' optimize_method (the default).

Returns

(T_estimate,T_bootstrap_estimates)

T_estimate: a temperature estimate *T_bootstrap_estimates*: a numpy array of bootstrap estimators

Return type Tuple of float and NumPy array

Examples

Draw samples from a D-Wave Quantum Computer for a large spin-glass problem (random couplers J, zero external field h). Establish a temperature estimate by maximum pseudo-likelihood.

Note that due to the complicated freeze-out properties of hard models, such as large scale spin-glasses, deviation from a classical Boltzmann distribution is anticipated. Nevertheless, the T estimate can always be interpreted as an estimator of local excitations rates. For example T will be 0 if only local minima are returned (even if some of the local minima are not ground states).

```
>>> import dimod
>>> from dwave.system.temperatures import maximum_pseudolikelihood_temperature
>>> from dwave.system import DWaveSampler
>>> from random import random
>>> sampler = DWaveSampler()
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {e : 1-2*random() for e in␣
↪sampler.edgelist})
>>> sampleset = sampler.sample(bqm, num_reads=100, auto_scale=False)
>>> T,T_bootstrap =  maximum_pseudolikelihood_temperature(bqm,sampleset)
>>> print('Effective temperature ',T)     # doctest: +SKIP
Effective temperature  0.24066488780293813
```

See also:

https://doi.org/10.3389/fict.2016.00023

https://www.jstor.org/stable/25464568

dwave.system.temperatures.freezeout_effective_temperature

freezeout_effective_temperature (*freezeout_B*, *temperature*, *units_B='GHz'*, *units_T='mK'*) → float

Provides an effective temperature as a function of freezeout information.

See https://docs.dwavesys.com/docs/latest/c_qpu_annealing.html for a complete summary of D-Wave annealing quantum computer operation.

A D-Wave annealing quantum computer is assumed to implement a Hamiltonian $H(s) = B(s)/2H_P - A(s)/2H_D$, where: H_P is the unitless diagonal problem Hamiltonian, H_D is the unitless driver Hamiltonian, $B(s)$ is the problem energy scale; A(s) is the driver energy scale, amd s is the normalized anneal time $s = t/t_a$ (in [0,1]). Diagonal elements of H_P, indexed by the spin state x, are equal to the energy of a classical Ising spin system

$$E_{Ising}(x) = \sum_i h_i x_i + \sum_{i>j} J_{i,j} x_i x_j$$

If annealing achieves a thermally equilibrated distribution over decohered states at large s where $A(s) \ll B(s)$, and dynamics stop abruptly at $s = s^*$, the distribution of returned samples is well described by a Boltzmann distribution:

$$P(x) = \exp(-B(s^*)RE_{Ising}(x)/2k_BT)$$

where T is the physical temperature, and k_B is the Boltzmann constant. R is a Hamiltonain rescaling factor, if a QPU is operated with auto_scale=False, then R=1. The function calculates the unitless effective temperature as $T_{eff} = 2k_BT/B(s^*)$.

Device temperature T, annealing schedules $\{A(s), B(s)\}$ and single-qubit freeze-out (s^*, for simple uncoupled Hamltonians) are reported device properties: https://docs.dwavesys.com/docs/latest/doc_physical_properties.html These values (typically specified in mK and GHz) allows the calculation of an effective temperature for simple Hamiltonians submitted to D-Wave quantum computers. Complicated problems exploiting embeddings, or with many coupled variables, may freeze out at different values of s or piecemeal). Large problems may have slow dynamics at small values of s, so $A(s)$ cannot be ignored as a contributing factor to the distribution.

Note that for QPU solvers this temperature estimate applies to problems submitted with no additional scaling factors (sampling with `auto_scale = False`). If `auto_scale=True` (default) additional scaling factors must be accounted for.

Parameters

- **freezeout_B** (*float*) – $B(s^*)$, the problem Hamiltonian energy scale at freeze-out.
- **temperature** (*float*) – T, the physical temperature of the quantum computer.
- **units_B** (*string, optional, 'GHz'*) – Units in which `freezeout_B` is specified. Allowed values: 'GHz' (Giga-Hertz) and 'J' (Joules).
- **units_T** (*string, optional, 'mK'*) – Units in which the `temperature` is specified. Allowed values: 'mK' (milli-Kelvin) and 'K' (Kelvin).

Returns The effective (unitless) temperature.

Return type float

Examples

This example uses the published parameters <https://docs.dwavesys.com/docs/latest/doc_physical_properties.html> for the Advantage_system4.1 QPU solver as of November 22nd 2021: $B(s = 0.612) = 3.91$ GHz , :math:'T = 15.4'mK.

```
>>> from dwave.system.temperatures import freezeout_effective_temperature
>>> T = freezeout_effective_temperature(freezeout_B = 3.91, temperature = 15.4)
>>> print('Effective temperature at single qubit freeze-out is', T)  # doctest:␣
↪+ELLIPSIS
Effective temperature at single qubit freeze-out is 0.164...
```

See also:

The function *fast_effective_temperature* estimates the temperature for single-qubit Hamiltonians, in approximate agreement with estimates by this function at reported single-qubit freeze-out values s^* and device physical parameters.

dwave.system.temperatures.fast_effective_temperature

fast_effective_temperature(*sampler=None, num_reads=None, seed=None, h_range=(-0.1639344262295082, 0.1639344262295082), sampler_params=None, optimize_method=None, num_bootstrap_samples=0*) → Tuple[numpy.float64, numpy.float64]

Provides an estimate to the effective temperature, T, of a sampler.

This function submits a set of single-qubit problems to a sampler and uses the rate of excitations to infer a maximum-likelihood estimate of temperature.

Parameters

- **sampler** (`dimod.Sampler`, optional, default=*DWaveSampler*) – A dimod sampler.
- **num_reads** (*int, optional*) – Number of reads to use. Default is 100 if not specified in `sampler_params`.
- **seed** (*int, optional*) – Seeds the problem generation process. Allowing reproducibility from pseudo-random samplers.
- **h_range** (*float, optional, default = [-1/6.1,1/6.1]*) – Determines the range of external fields probed for temperature inference. Default is based on a D-Wave Advantage processor, where single-qubit freeze-out implies an effective temperature of 6.1 (see *freezeout_effective_temperature*). The range should be chosen inversely proportional to the anticipated temperature for statistical efficiency, and to accomodate precision and other nonidealities such as precision limitations.

- **sampler_params** (*dict, optional*) – Any additional non-defaulted sampler parameterization. If `num_reads` is a key, must be compatible with `num_reads` argument.
- **optimize_method** (*str, optional*) – Optimize method used by SciPy `root_scalar` method. The default method works well under default operation, 'bisect' can be numerically more stable when operated without defaults.
- **num_bootstrap_samples** (*int, optional, default=0*) – Number of bootstrap samples to use for estimation of the standard error. By default no bootstrapping is performed and the standard error is defaulted to 0.

Returns The effective temperature describing single qubit problems in an external field, and a standard error (+/- 1 sigma). By default the confidence interval is set as 0.

Return type Tuple[float, float]

See also:

https://doi.org/10.3389/fict.2016.00023

https://www.jstor.org/stable/25464568

Examples

Draw samples from a *DWaveSampler*, and establish the temperature

```
>>> from dwave.system.temperatures import fast_effective_temperature
>>> from dwave.system import DWaveSampler
>>> sampler = DWaveSampler()
>>> T, _ = fast_effective_temperature(sampler)
>>> print('Effective temperature at freeze-out is',T)     # doctest: +SKIP
0.21685104745347336
```

See also:

The function *freezeout_effective_temperature* may be used in combination with published device values to estimate single-qubit freeze-out, in approximate agreement with empirical estimates of this function.

https://doi.org/10.3389/fict.2016.00023

https://www.jstor.org/stable/25464568

1.2.5 Warnings

Settings for raising warnings may be configured by tools such as composites or dwave-inspector.

This example configures warnings for an instance of the *EmbeddingComposite()* class used on a sampler structured to represent variable `a` with a long chain.

```
>>> import networkx as nx
>>> import dimod
>>> import greedy
...
>>> G = nx.Graph()
>>> G.add_edges_from([(n, n + 1) for n in range(10)])
>>> sampler = dimod.StructureComposite(greedy.SteepestDescentSampler(), G.nodes, G.
↪edges)
>>> sampleset = EmbeddingComposite(sampler).sample_ising({}, {("a", "b"): -1},
...     return_embedding=True,
```

(continues on next page)

(continued from previous page)

```
...     embedding_parameters={"fixed_chains": {"a": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}},
...     warnings=dwave.system.warnings.SAVE)
>>> "warnings" in sampleset.info
True
```

class ChainBreakWarning
Raised if a chain's qubits are in different states for lowest-energy samples.

class ChainLengthWarning
Raised if the number of qubits forming a chain is high.

class ChainStrengthWarning
Base category for warnings about the embedding chain strength.

class EnergyScaleWarning
Base category for warnings about the relative bias strengths.

class TooFewSamplesWarning
Raised if lowest-energy samples are a small fraction of the total samples.

class WarningAction
Settings for raising warnings.

An enum with values `IGNORE` and `SAVE`.

class WarningHandler(*action=None*)

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

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

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

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

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

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

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

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

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

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

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

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

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

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly

display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

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

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

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

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

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

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

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

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

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

NOTICE file.

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

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

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

END OF TERMS AND CONDITIONS

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

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

Copyright [yyyy] [name of copyright owner]

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

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

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

Python Module Index

d

Index

Symbols

A

C

D

E

F

I

L

M

N

P

Q

R

S

T

U

V

W