D-Wave Hybrid

Release 0.4.0

Aug 27, 2019

Contents

A general, minimal Python framework for building hybrid asynchronous decomposition samplers for quadratic unconstrained binary optimization (QUBO) problems.

dwave-hybrid facilitates three aspects of solution development:

- Hybrid approaches to combining quantum and classical compute resources
- Evaluating a portfolio of algorithmic components and problem-decomposition strategies
- Experimenting with workflow structures and parameters to obtain the best application results

The framework enables rapid development and insight into expected performance of productized versions of its experimental prototypes.

Your optimized algorithmic components and other contributions to this project are welcome!

CHAPTER 1

Example

```
import dimod
import hybrid

# Construct a problem
bqm = dimod.BinaryQuadraticModel({}, {'ab': 1, 'bc': -1, 'ca': 1}, 0, dimod.SPIN)

# Define the workflow
iteration = hybrid.RacingBranches(
    hybrid.InterruptableTabuSampler(),
    hybrid.EnergyImpactDecomposer(size=2)
    | hybrid.QPUSubproblemAutoEmbeddingSampler()
    | hybrid.SplatComposer()
) | hybrid.ArgMin()
workflow = hybrid.LoopUntilNoImprovement(iteration, convergence=3)

# Solve the problem
init_state = hybrid.State.from_problem(bqm)
final_state = workflow.run(init_state).result()

# Print results
print("Solution: sample={.samples.first}".format(final_state))
```

CHAPTER 2

Documentation

2.1 Introduction

dwave-hybrid provides a framework for iterating arbitrary-sized sets of samples through parallel solvers to find an optimal solution.

For the documentation of a particular code element, see the *Reference Documentation* section. This introduction gives an overview of the package; steps you through using it, starting with running a provided hybrid solver that handles arbitrary-sized QUBOs; and points out the way to developing your own components in the framework.

- *Overview* presents the framework and explains key concepts.
- *Using the Framework* shows how to use the framework. You can quickly get started by using a provided reference sampler built with this framework, Kerberos, to solve a problem too large to minor-embed on a D-Wave system. Next, use the framework to build (hybrid) workflows; for example, a solver similar to qbsolv, which can employ tabu search on a whole problem while submitting parts of the problem to a D-Wave system.
- *Developing New Components* guides you to developing your own hybrid components.
- *Reference Examples* describes some workflow examples included in the code.

2.1.1 Overview

The *dwave-hybrid* framework enables you to quickly design and test workflows that iterate sets of samples through samplers to solve arbitrary QUBOs. Large problems can be decomposed and two or more solution techniques can run in parallel.

The *Schematic Representation* figure below shows an example configuration. Samples are iterated over four parallel solvers. The top **branch** represents a classical tabu search that runs on the entire problem until interrupted by another branch completing. These use different decomposers to parcel out parts of the current sample set (iteration i) to samplers such as a D-Wave system (second-highest branch) or another structure of parallel simulated annealing and tabu search. A generic representation of a branch's components—decomposer, sampler, and composer—is shown in the lowest branch. A user-defined criterion selects from current samples and solver outputs a sample set for iteration $i + 1$.

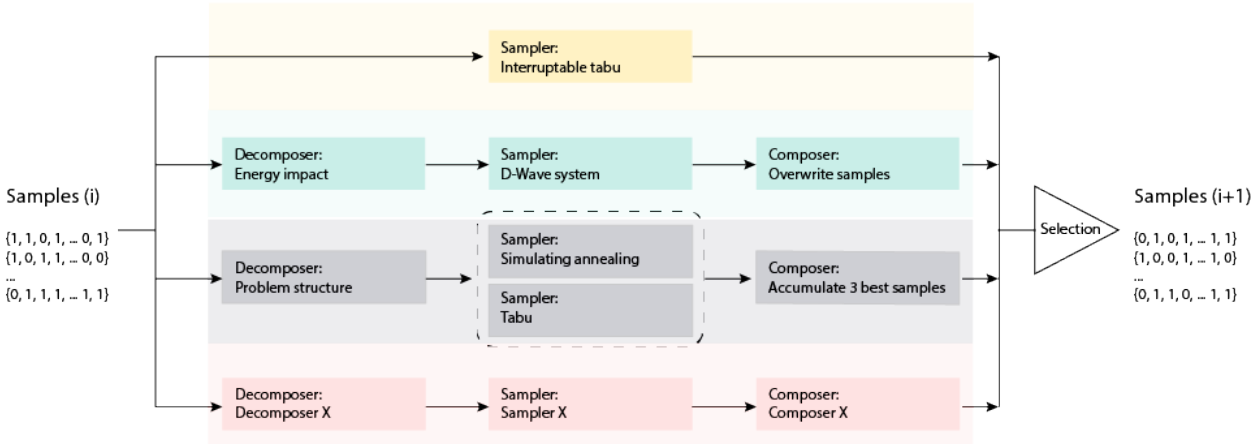


Fig. 1: Schematic Representation

You can use the framework to run a provided hybrid solver or to configure workflows using provided components such as tabu samplers and energy-based decomposers.

You can also use the framework to build your own components to incorporate into your workflow.

2.1.2 Using the Framework

This section helps you quickly use a provided reference sampler to solve arbitrary-sized problems and then shows you how to build (hybrid) workflows using provided components.

Reference Hybrid Sampler: Kerberos

dwave-hybrid includes a reference example sampler built using the framework: Kerberos is a dimod-compatible hybrid asynchronous decomposition sampler that enables you to solve problems of arbitrary structure and size. It finds best samples by running in parallel tabu search, simulated annealing, and D-Wave subproblem sampling on problem variables that have high-energy impact.

The example below uses Kerberos to solve a large QUBO.

```
>>> import dimod
>>> from hybrid.reference.kerberos import KerberosSampler
>>> with open('../problems/random-chimera/8192.01.qubo') as problem:
...     bqm = dimod.BinaryQuadraticModel.from_coo(problem)
>>> len(bqm)
8192
>>> solution = KerberosSampler().sample(bqm, max_iter=10, convergence=3)    # doctest:␣
↪+SKIP
>>> solution.first.energy     # doctest: +SKIP
-4647.0
```

Building Workflows

As shown in the *Overview* section, you build hybrid solvers by arranging components such as samplers in a workflow.

Building Blocks

The basic components—building blocks—you use are based on the *Runnable* class: decomposers, samplers, and composers. Such components input a set of samples, a `SampleSet`, and output updated samples. A `State` associated with such an iteration of a component holds the problem, samples, and optionally additional information.

The following example demonstrates a simple workflow that uses just one *Runnable*, a sampler representing the classical tabu search algorithm, to solve a problem (fully classically, without decomposition). The example solves a small problem of a triangle graph of nodes identically coupled. An initial *State* of all-zero samples is set as a starting point. The solution, *new_state*, is derived from a single iteration of the *TabuProblemSampler* *Runnable*.

```
>>> import dimod
>>> # Define a problem
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {'ab': 0.5, 'bc': 0.5, 'ca': 0.5})
>>> # Set up the sampler with an initial state
>>> sampler = TabuProblemSampler(tenure=2, timeout=5)
>>> state = State.from_sample({'a': 0, 'b': 0, 'c': 0}, bqm)
>>> # Sample the problem
>>> new_state = sampler.run(state).result()
>>> print(new_state.samples)                    # doctest: +SKIP
    a   b   c  energy  num_occ.
0  +1  -1  -1    -0.5         1
['SPIN', 1 rows, 1 samples, 3 variables]
```

Flow Structuring

The framework provides classes for structuring workflows that use the "building-block" components. As shown in the *Overview* section, you can create a *branch* of `Runnable` classes; for example `decomposer | sampler | composer`, which delegates part of a problem to a sampler such as the D-Wave system.

The following example shows a branch comprising a decomposer, local Tabu solver, and a composer. A 10-variable binary quadratic model is decomposed by the energy impact of its variables into a 6-variable subproblem to be sampled twice. An initial state of all -1 values is set using the utility function *min_sample()*.

```
>>> import dimod           # Create a binary quadratic model
>>> bqm = dimod.BinaryQuadraticModel({t: 0 for t in range(10)},
...                                  {(t, (t+1) % 10): 1 for t in range(10)},
...                                  0, 'SPIN')
>>> branch = (EnergyImpactDecomposer(size=6, min_gain=-10) |
...           TabuSubproblemSampler(num_reads=2) |
...           SplatComposer())
>>> new_state = branch.next(State.from_sample(min_sample(bqm), bqm))
>>> print(new_state.subsamples)      # doctest: +SKIP
    4   5   6   7   8   9  energy  num_occ.
0  +1  -1  -1  +1  -1  +1    -5.0         1
1  +1  -1  -1  +1  -1  +1    -5.0         1
['SPIN', 2 rows, 2 samples, 6 variables]
```

Such *Branch* classes can be run in parallel using the *RacingBranches* class. From the outputs of these parallel branches, *ArgMin* selects a new current sample. And instead of a single iteration on the sample set, you can use the *Loop* to iterate a set number of times or until a convergence criteria is met.

This example of *Racing Branches* solves a binary quadratic model by iteratively producing best samples. Similar to qbsolv, it employs both tabu search on the entire problem and a D-Wave system on subproblems. In addition to building-block components such as employed above, this example also uses infrastructure classes to manage the decomposition and parallel running of branches.

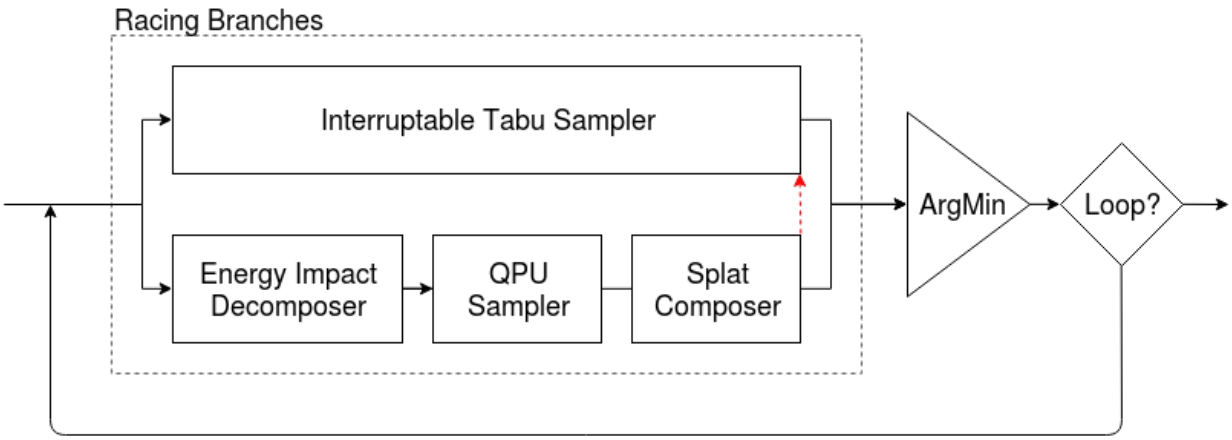


Fig. 2: Racing Branches

```
import dimod
import hybrid

# Construct a problem
bqm = dimod.BinaryQuadraticModel({}, {'ab': 1, 'bc': -1, 'ca': 1}, 0, dimod.SPIN)

# Define the workflow
iteration = hybrid.RacingBranches(
    hybrid.InterruptableTabuSampler(),
    hybrid.EnergyImpactDecomposer(size=2)
    | hybrid.QPUSubproblemAutoEmbeddingSampler()
    | hybrid.SplatComposer()
) | hybrid.ArgMin()
workflow = hybrid.LoopUntilNoImprovement(iteration, convergence=3)

# Solve the problem
init_state = hybrid.State.from_problem(bqm)
final_state = workflow.run(init_state).result()

# Print results
print("Solution: sample={.samples.first}".format(final_state))
```

Flow Refining

The framework enables quick modification of work flows to improve solutions and performance. For example, after verifying the *Racing Branches* workflow above on its small problem, you might make a series of modifications such as the examples below to better fit it to problems with large numbers of variables.

1. Configure a decomposition window that moves down a fraction of problem variables, ordered from highest to lower energy impact, and submit those subproblems to the D-Wave system while tabu searches globally. This example submits 50-variable subproblems on up to 15% of the total variables.

```
# Redefine the workflow: a rolling decomposition window
subproblem = hybrid.EnergyImpactDecomposer(size=50, rolling_history=0.15)
subsampler = hybrid.QPUSubproblemAutoEmbeddingSampler() | hybrid.SplatComposer()
```

(continues on next page)

(continued from previous page)

```
iteration = hybrid.RacingBranches(
    hybrid.InterruptableTabuSampler(),
    subproblem | subsampler
) | hybrid.ArgMin()

workflow = hybrid.LoopUntilNoImprovement(iteration, convergence=3)
```

2. Instead of sequentially producing a sample per subproblem, a further modification might be to process all the subproblems in parallel and merge the returned samples. Here the *EnergyImpactDecomposer* is iterated until it raises a *EndOfStream()* exception when it reaches 15% of the variables, and then all the 50-variable subproblems are submitted to the D-Wave system.

```
# Redefine the workflow: parallel subproblem solving for a single sample
subproblem = hybrid.Unwind(
    hybrid.EnergyImpactDecomposer(size=50, rolling_history=0.15)
)

subsampler = hybrid.Map(
    hybrid.QPUSubproblemAutoEmbeddingSampler()
) | hybrid.Reduce(
    hybrid.Lambda(merge_substates)
) | hybrid.SplatComposer()
```

3. Change the criterion for selecting subproblems. By default, the variables are selected by maximal energy impact but selection can be better tailored to a problem's structure.

   For example, for binary quadratic model representing the problem graph shown in the *Traversal by Energy Impact* graphic, if you select a subproblem size of four, these nodes selected by descending energy impact are not directly connected (no shared edges, and might not represent a local structure of the problem).

Additional Examples

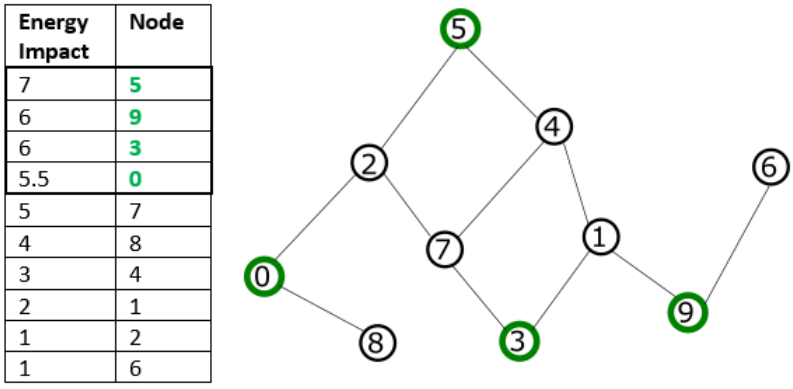
Tailoring State Selection

The next example tailors a state selector for a sampler that does some post-processing and can alert upon suspect samples. Sampler output modified by ellipses ("…") for readability is shown below for an Ising model of a triangle problem with zero biases and interactions all equal to 0.5. The first of three *State* classes is flagged as problematic using the *info* field:

```
[{...,'samples': SampleSet(rec.array([([0, 1, 0], 0., 1)], ..., ['a', 'b', 'c'], {
↪'Postprocessor': 'Excessive chain breaks'}, 'SPIN')},
{...,'samples': SampleSet(rec.array([([1, 1, 1], 1.5, 1)], ..., ['a', 'b', 'c'], {},
↪'SPIN')},
{...,'samples': SampleSet(rec.array([([0, 0, 0], 0., 1)], ..., ['a', 'b', 'c'], {},
↪'SPIN')}]
```

This code snippet defines a metric for the key argument in *ArgMin*:

```
def preempt(si):
    if 'Postprocessor' in si.samples.info:
        return(math.inf)
    else:
        return(si.samples.first.energy)
```



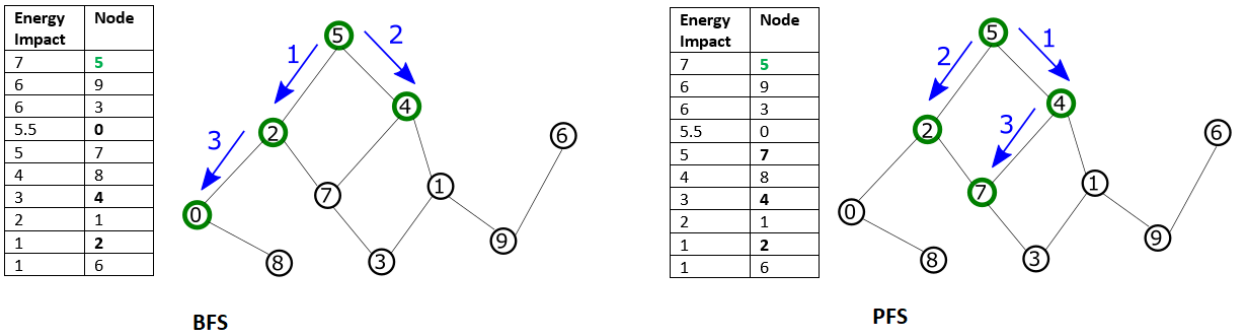
Energy Impact	Node
7	5
6	9
6	3
5.5	0
5	7
4	8
3	4
2	1
1	2
1	6

Fig. 3: Traversal by Energy Impact

Configuring a mode of traversal such as breadth-first (BFS) or priority-first selection (PFS) can capture features that represent local structures within a problem.

```
# Redefine the workflow: subproblem selection
subproblem = hybrid.Unwind(
    hybrid.EnergyImpactDecomposer(size=50, rolling_history=0.15,␣
↪traversal='bfs'))
```

These two selection modes are shown in the *Traversal by BFS or PFS* graphic. BFS starts with the node with maximal energy impact, from which its graph traversal proceeds to directly connected nodes, then nodes directly connected to those, and so on, with graph traversal ordered by node index. In PFS, graph traversal selects the node with highest energy impact among unselected nodes directly connected to any already selected node.



Energy Impact	Node
7	5
6	9
6	3
5.5	**0**
5	7
4	8
3	**4**
2	1
1	**2**
1	6

Energy Impact	Node
7	5
6	9
6	3
5.5	0
5	**7**
4	8
3	**4**
2	1
1	**2**
1	6

Fig. 4: Traversal by BFS or PFS

Using the defined key on the above input, *ArgMin* finds the state with the lowest energy (zero) excluding the flagged state (which also has energy of zero):

```
>>> ArgMin(key=preempt).next(states)      # doctest: +SKIP
{'problem': BinaryQuadraticModel({'a': 0.0, 'b': 0.0, 'c': 0.0}, {('a', 'b'): 0.5, ('b
↪', 'c'): 0.5, ('c', 'a'): 0.5},
0.0, Vartype.SPIN), 'samples': SampleSet(rec.array([([0, 0, 0], 0., 1)],
dtype=[('sample', 'i1', (3,)), ('energy', '<f8'), ('num_occurrences', '<i4')]), ['a',
↪'b', 'c'], {}, 'SPIN')}
```

Parallel Sampling

The code snippet below uses *Map* to run a tabu search on two states in parallel.

```
>>> Map(TabuProblemSampler()).run(States(                    # doctest: +SKIP
        State.from_sample({'a': 0, 'b': 0, 'c': 1}, bqm1),
        State.from_sample({'a': 1, 'b': 1, 'c': 0}, bqm2)))
>>> _.result()                 # doctest: +SKIP
[{'samples': SampleSet(rec.array([([-1, -1,  1], -0.5, 1)], dtype=[('sample', 'i1',␣
↪(3,)),
  ('energy', '<f8'), ('num_occurrences', '<i4')]), ['a', 'b', 'c'], {}, 'SPIN'),
  'problem': BinaryQuadraticModel({'a': 0.0, 'b': 0.0, 'c': 0.0}, {('a', 'b'): 0.5, ('b
↪', 'c'): 0.5,
  ('c', 'a'): 0.5}, 0.0, Vartype.SPIN)},
  {'samples': SampleSet(rec.array([([ 1,  1, -1], -1., 1)], dtype=[('sample', 'i1', (3,
↪)),
  ('energy', '<f8'), ('num_occurrences', '<i4')]), ['a', 'b', 'c'], {}, 'SPIN'),
  'problem': BinaryQuadraticModel({'a': 0.0, 'b': 0.0, 'c': 0.0}, {('a', 'b'): 1, ('b',
↪ 'c'): 1,
  ('c', 'a'): 1}, 0.0, Vartype.SPIN)}]
```

Logging and Execution Information

You can see detailed execution information by setting the level of logging.

The package supports logging levels TRACE, DEBUG, INFO, WARNING, ERROR, and CRITICAL in ascending order of severity. By default, logging level is set to ERROR. You can select the logging level with environment variable `DWAVE_HYBRID_LOG_LEVEL`.

For example, on a Windows operating system, set this environment variable to INFO level as:

```
set DWAVE_HYBRID_LOG_LEVEL=INFO
```

or on a Unix-based system as:

```
DWAVE_HYBRID_LOG_LEVEL=INFO
```

The previous example above might output something like the following:

```
>>> print("Solution: sample={s.samples.first}".format(s=solution))    # doctest: +SKIP
```

```
2018-12-10 15:18:30,634 hybrid.flow INFO Loop Iteration(iterno=0, best_state_quality=-
↪3.0)
2018-12-10 15:18:31,511 hybrid.flow INFO Loop Iteration(iterno=1, best_state_quality=-
↪3.0)
```

(continues on next page)

(continued from previous page)

```
2018-12-10 15:18:35,889 hybrid.flow INFO Loop Iteration(iterno=2, best_state_quality=-
↪3.0)
2018-12-10 15:18:37,377 hybrid.flow INFO Loop Iteration(iterno=3, best_state_quality=-
↪3.0)
Solution: sample=Sample(sample={'a': 1, 'b': -1, 'c': -1}, energy=-3.0, num_
↪occurrences=1)
```

2.1.3 Developing New Components

The *dwave-hybrid* framework enables you to build your own components to incorporate into your workflow.

The key superclass is the `Runnable` class: all basic components—samplers, decomposers, composers—and flow-structuring components such as branches inherit from this class. A `Runnable` is run for an iteration in which it updates the `State` it receives. Typical methods are *run* or *next* to execute an iteration and *stop* to terminate the `Runnable`.

The *Primitives* and *Flow Structuring* sections describe, respectively, the basic `Runnable` classes (building blocks) and flow-structuring ones and their methods. If you are implementing these methods for your own `Runnable` class, see comments in the code.

The *Racing Branches* graphic below shows the top-down composition (tree structure) of a hybrid loop.

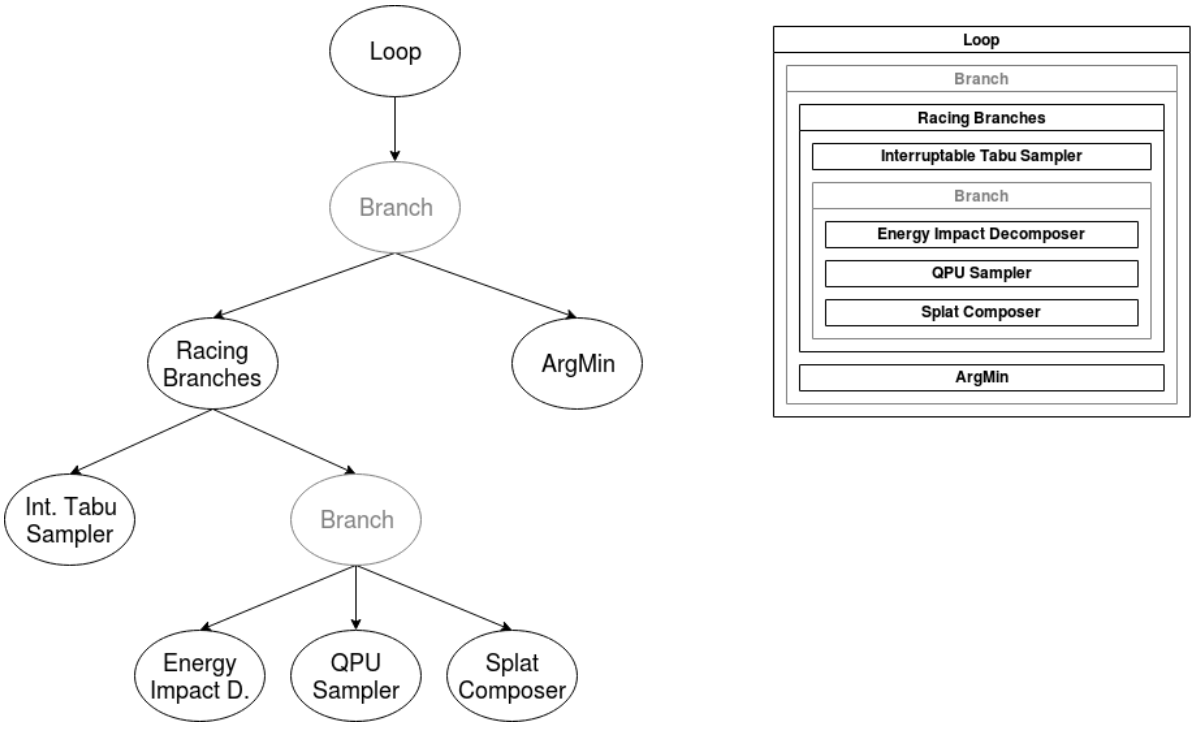


Fig. 5: Top-Down Composition

State traits are verified for all `Runnable` objects that inherit from `StateTraits` or its subclasses. Verification includes:

(1) Minimal checks of workflow construction (composition of `Runnable` classes)

(2) Runtime checks

All built-in *Runnable* classes declare state traits requirements that are either independent (for simple ones) or derived from a child workflow. Traits of a new *Runnable* must be expressed and modified at construction time by its parent. When developing new *Runnable* classes, constructing composite traits can be nontrivial for some advanced flow-control runnables.

The *Dimod Conversion* section describes the *HybridRunnable* class you can use to produce a *Runnable* sampler based on a dimod sampler.

The *Utilities* section provides a list of useful utility methods.

2.1.4 Reference Examples

The examples directory of the code includes implementations of some *Reference Workflows* you can incorporate as provided into your application and also use to jumpstart your development of custom workflows.

A typical first use of dwave-hybrid might be to simply use the Kerberos reference sampler to solve a QUBO, as shown in *Using the Framework*. Next, you might tune its configurable parameters, described under *Reference Workflows*.

To further improve performance, you can step up from using a generic workflow to one tailored for your application and its problem. As a first step you can modify a reference workflow with existing components. After that, you can implement your own components as described in *Developing New Components*.

2.2 Reference Documentation

2.2.1 Primitives

Basic building-block classes and superclasses for hybrid workflows.

Classes

class Present(*result=None*, *exception=None*)
Already resolved Future object.

Users should treat this class as just another Future, the difference being an implementation detail: *Present* is "resolved" at construction time.

See the example of the *run()* method.

class Runnable(***runopts*)
Components such as samplers and branches that can be run for an iteration.

Parameters ****runopts** (*dict*) – Keyword arguments passed down to each *Runnable.run* call.

Note: The base class *Runnable* does not enforce traits validation. To enable validation, derive your subclass from one of the state structure, I/O dimensionality, or I/O validation mixins in *traits*.

Examples

This example runs a tabu search on a binary quadratic model. An initial state is manually set to $x = y = 0, z = 1; a = b = 1, c = 0$ and an updated state is created by running the sampler for one iteration.

```
>>> import dimod            # Create a binary quadratic model
>>> bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b
↪': 0.0, 'c': 6.0},
...                                  {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y
↪'): -4.0,
...                                  ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b
↪'): -4.0, ('a', 'z'): -4.0},
...                                  -1.0, 'BINARY')
>>> # Set up the sampler runnable
>>> sampler = TabuProblemSampler(tenure=2, timeout=5)
>>> # Run one iteration of the sampler
>>> new_state = sampler.next(State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b
↪': 1, 'c': 0}, bqm))
>>> print(new_state.samples)      # doctest: +SKIP
      a  b  c  x  y  z  energy  num_occ.
   0  1  1  1  1  1  1    -1.0         1
   [ 1 rows, 6 variables ]
```

class State(**args*, ***kwargs*)

Computation state passed along a branch between connected components.

State is a `dict` subclass and usually contains at least two keys: `samples` and `problem`.

Examples

```
>>> import dimod            # Create a binary quadratic model
>>> bqm = dimod.BinaryQuadraticModel({0: -1, 1: -1}, {(0, 1): 2}, 0.0, dimod.
↪BINARY)
>>> hybrid.core.State.from_sample(hybrid.utils.min_sample(bqm), bqm)    # doctest:␣
↪+SKIP
{'problem': BinaryQuadraticModel({0: -1, 1: -1}, {(0, 1): 2}, 0.0, Vartype.
↪BINARY),
 'samples': SampleSet(rec.array([([0, 0], 0., 1)],
    dtype=[('sample', 'i1', (2,)), ('energy', '<f8'), ('num_occurrences', '<i4
↪')]), [0, 1], {}, 'BINARY')}
```

class States(**args*)

List of states.

Properties

Runnable.name	Return the *Runnable* class name.
SampleSet.first	Sample with the lowest-energy.

hybrid.core.Runnable.name

Runnable.**name**

Return the *Runnable* class name.

hybrid.core.SampleSet.first

SampleSet.**first**

Sample with the lowest-energy.

Raises ValueError – If empty.

Example

```
>>> sampleset = dimod.ExactSolver().sample_ising({'a': 1}, {('a', 'b'): 1})
>>> sampleset.first
Sample(sample={'a': -1, 'b': 1}, energy=-2.0, num_occurrences=1)
```

Methods

Runnable.dispatch(future, **kwargs)	Dispatch state from resolving *future* to either *next* or *error* methods.
Runnable.error(exc)	Execute one blocking iteration of an instantiated *Runnable* with an exception as input.
Runnable.init(state, **runopts)	Run prior to the first next/run, with the first state received.
Runnable.halt()	Called by *stop()*.
Runnable.next(state, **runopts)	Execute one blocking iteration of an instantiated *Runnable* with a valid state as input.
Runnable.run(state, **kwargs)	Execute the next step/iteration of an instantiated *Runnable*.
Runnable.stop()	Terminate an iteration of an instantiated *Runnable*.
State.updated(**kwargs)	Return a (deep) copy of the state, updated from *kwargs*.
State.from_sample(sample, bqm, **kwargs)	Convenience method for constructing a state from a raw (dict) sample.
State.from_samples(samples, bqm, **kwargs)	Convenience method for constructing a state from raw (dict) samples.

hybrid.core.Runnable.dispatch

Runnable.**dispatch**(*future*, ***kwargs*)

Dispatch state from resolving *future* to either *next* or *error* methods.

Parameters **state** (concurrent.futures.Future-like object) – *State* future.

Returns state from *next()* or *error()*, or passes through an exception raised there.

Blocks on state resolution and execution of *next()* or *error()*.

hybrid.core.Runnable.error

Runnable.**error**(*exc*)

Execute one blocking iteration of an instantiated *Runnable* with an exception as input.

Called when the previous component raised an exception instead of generating a new state.

The default implementation raises again the input exception. Runnable errors must be explicitly silenced.

hybrid.core.Runnable.init

Runnable.**init**(*state*, ***runopts*)
Run prior to the first next/run, with the first state received.

Default to NOP.

hybrid.core.Runnable.halt

Runnable.**halt**()
Called by *stop()*. Override this method (instead of *stop*) to handle stopping of one blocking call of *next*. Defaults to NOP.

hybrid.core.Runnable.next

Runnable.**next**(*state*, ***runopts*)
Execute one blocking iteration of an instantiated *Runnable* with a valid state as input.

Parameters **state** (*State*) – Computation state passed between connected components.

Returns The new state.

Return type *State*

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

hybrid.core.Runnable.run

Runnable.**run**(*state*, ***kwargs*)
Execute the next step/iteration of an instantiated *Runnable*.

Accepts a state in a Future-like object and returns a new state in a Future-like object.

Parameters

- **state** (*State*) – Computation state future-like object passed between connected components.
- **executor** (Executor, optional, default=None) – The Executor to which the execution of this block is scheduled. By default *hybrid.concurrency.thread_executor* is used.

Examples

These two code snippets run one iteration of a sampler to produce a new state. The first is an asynchronous call and the second a blocking call.

```
>>> sampler.run(State.from_sample(min_sample(bqm), bqm))    # doctest: +SKIP
<Future at 0x20cbe22ea20 state=running>
```

```
>>> sampler.run(State.from_sample(min_sample(bqm), bqm),
...             executor=hybrid.immediate_executor)   # doctest: +SKIP
<Present at 0x20ca68cd2b0 state=finished returned State>
```

hybrid.core.Runnable.stop

Runnable.**stop**()
Terminate an iteration of an instantiated *Runnable*.

hybrid.core.State.updated

State.**updated**(***kwargs*)
Return a (deep) copy of the state, updated from *kwargs*.

This method has *dict.update* semantics with immutability of *sorted*. Currently an exception is the *debug* key, if it exists, for which a depth-unlimited recursive merge is executed.

Example

```
>>> state = State()
>>> state
{}
>>> newstate = state.updated(problem="test")
>>> newstate
{'problem': 'test'}
```

hybrid.core.State.from_sample

classmethod State.**from_sample**(*sample*, *bqm*, ***kwargs*)
Convenience method for constructing a state from a raw (dict) sample.

Energy is calculated from the binary quadratic model (BQM), and *State.problem* is also set to that BQM.

Example

```
>>> import dimod
>>> bqm = dimod.BQM.from_ising({}, {'ab': 0.5, 'bc': 0.5, 'ca': 0.5})
>>> state = State.from_sample({'a': -1, 'b': -1, 'c': -1}, bqm)
```

hybrid.core.State.from_samples

classmethod State.**from_samples**(*samples*, *bqm*, ***kwargs*)
Convenience method for constructing a state from raw (dict) samples.

Per-sample energy is calculated from the binary quadratic model (BQM), and *State.problem* is set to the BQM.

Example

```
>>> import dimod
>>> bqm = dimod.BQM.from_ising({}, {'ab': 0.5, 'bc': 0.5, 'ca': 0.5})
>>> state = State.from_samples([{'a': -1, 'b': -1, 'c': -1},
...                            {'a': -1, 'b': -1, 'c': 1}], bqm)
```

2.2.2 Samplers

Classical and quantum *Runnable* dimod samplers for problems and subproblems.

Classes

class InterruptableTabuSampler(*max_time=None*, ***tabu*)

An interruptable tabu sampler for a binary quadratic problem.

Parameters

- **num_reads** (*int, optional, default=1*) – Number of states (output solutions) to read from the sampler.
- **tenure** (*int, optional*) – Tabu tenure, which is the length of the tabu list, or number of recently explored solutions kept in memory. Default is a quarter of the number of problem variables up to a maximum value of 20.
- **timeout** (*int, optional, default=20*) – Timeout for non-interruptable operation of tabu search. At the completion of each loop of tabu search through its problem variables, if this time interval has been exceeded, the search can be stopped by an interrupt signal or expiration of the *timeout* parameter.
- **initial_states_generator** (*str, 'none'/'tile'/'random', optional, default='random'*) – Defines the expansion of input state samples into *initial_states* for the Tabu search, if fewer than *num_reads* samples are present. See `sample()`.
- **max_time** (*float, optional, default=None*) – Total running time in milliseconds.

See *Examples*.

class QPUSubproblemExternalEmbeddingSampler(*num_reads=100*, *qpu_sampler=None*, *sampling_params=None*, ***runopts*)

A quantum sampler for a subproblem with a defined minor-embedding.

Note: Externally supplied embedding must be present in the input state.

Parameters

- **num_reads** (*int, optional, default=100*) – Number of states (output solutions) to read from the sampler.
- **qpu_sampler** (`dimod.Sampler`, optional, default=`DWaveSampler`()) – Quantum sampler such as a D-Wave system.
- **sampling_params** (*dict*) – Dictionary of keyword arguments with values that will be used on every call of the (external-embedding-wrapped QPU) sampler.

See *Examples*.

class QPUSubproblemAutoEmbeddingSampler(*num_reads=100*, *qpu_sampler=None*, *sampling_params=None*, *auto_embedding_params=None*, ***runopts*)

A quantum sampler for a subproblem with automated heuristic minor-embedding.

Parameters

- **num_reads** (*int, optional, default=100*) – Number of states (output solutions) to read from the sampler.
- **qpu_sampler** (dimod.Sampler, optional, default=DWaveSampler()) – Quantum sampler such as a D-Wave system. Subproblems that do not fit the sampler's structure are minor-embedded on the fly with AutoEmbeddingComposite.
- **sampling_params** (*dict*) – Dictionary of keyword arguments with values that will be used on every call of the (embedding-wrapped QPU) sampler.
- **auto_embedding_params** (*dict, optional*) – If provided, parameters are passed to the AutoEmbeddingComposite constructor as keyword arguments.

See *Examples*.

class RandomSubproblemSampler

A random sample generator for a subproblem.

class ReverseAnnealingAutoEmbeddingSampler(*num_reads=100*, *anneal_schedule=None*, *qpu_sampler=None*, *sampling_params=None*, *auto_embedding_params=None*, ***runopts*)

A quantum reverse annealing sampler for a subproblem with automated heuristic minor-embedding.

Parameters

- **num_reads** (*int, optional, default=100*) – Number of states (output solutions) to read from the sampler.
- **anneal_schedule** (*list(list), optional, default=[[0, 1], [0.5, 0.5], [1, 1]]*) – An anneal schedule defined by a series of pairs of floating-point numbers identifying points in the schedule at which to change slope. The first element in the pair is time t in microseconds; the second, normalized persistent current s in the range [0,1]. The resulting schedule is the piecewise-linear curve that connects the provided points. For more details, see validate_anneal_schedule().
- **qpu_sampler** (dimod.Sampler, optional, default=DWaveSampler()) – Quantum sampler such as a D-Wave system. Subproblems that do not fit the sampler's structure are minor-embedded on the fly with AutoEmbeddingComposite.
- **sampling_params** (*dict*) – Dictionary of keyword arguments with values that will be used on every call of the (embedding-wrapped QPU) sampler.
- **auto_embedding_params** (*dict, optional*) – If provided, parameters are passed to the AutoEmbeddingComposite constructor as keyword arguments.

class SimulatedAnnealingProblemSampler(*num_reads=None*, *num_sweeps=1000*, *beta_range=None*, *beta_schedule_type='geometric'*, *initial_states_generator='random'*, ***runopts*)

A simulated annealing sampler for a complete problem.

Parameters

- **num_reads** (*int, optional, default=len(state.samples) or 1*) – Number of states (output solutions) to read from the sampler.

- **num_sweeps** (*int, optional, default=1000*) – Number of sweeps or steps.
- **beta_range** (*tuple, optional*) – A 2-tuple defining the beginning and end of the beta schedule, where beta is the inverse temperature. The schedule is applied linearly in beta. Default range is set based on the total bias associated with each node.
- **beta_schedule_type** (*string, optional, default='geometric'*) – Beta schedule type, or how the beta values are interpolated between the given 'beta_range'. Supported values are: linear and geometric.
- **initial_states_generator** (*str, 'none'/'tile'/'random', optional, default='random'*) – Defines the expansion of input state samples into *initial_states* for the simulated annealing, if fewer than *num_reads* samples are present. See `sample()`.

class SimulatedAnnealingSubproblemSampler(*num_reads=None, num_sweeps=1000, beta_range=None, beta_schedule_type='geometric', initial_states_generator='random', **runopts*)

A simulated annealing sampler for a subproblem.

Parameters

- **num_reads** (*int, optional, default=len(state.subsamples) or 1*) – Number of states (output solutions) to read from the sampler.
- **num_sweeps** (*int, optional, default=1000*) – Number of sweeps or steps.
- **beta_range** (*tuple, optional*) – A 2-tuple defining the beginning and end of the beta schedule, where beta is the inverse temperature. The schedule is applied linearly in beta. Default range is set based on the total bias associated with each node.
- **beta_schedule_type** (*string, optional, default='geometric'*) – Beta schedule type, or how the beta values are interpolated between the given 'beta_range'. Supported values are: linear and geometric.
- **initial_states_generator** (*str, 'none'/'tile'/'random', optional, default='random'*) – Defines the expansion of input state subsamples into *initial_states* for the simulated annealing, if fewer than *num_reads* subsamples are present. See `sample()`.

See *Examples*.

class TabuProblemSampler(*num_reads=None, tenure=None, timeout=100, initial_states_generator='random', **runopts*)

A tabu sampler for a binary quadratic problem.

Parameters

- **num_reads** (*int, optional, default=len(state.samples) or 1*) – Number of states (output solutions) to read from the sampler.
- **tenure** (*int, optional*) – Tabu tenure, which is the length of the tabu list, or number of recently explored solutions kept in memory. Default is a quarter of the number of problem variables up to a maximum value of 20.
- **timeout** (*int, optional, default=100*) – Total running time in milliseconds.
- **initial_states_generator** (*str, 'none'/'tile'/'random', optional, default='random'*) – Defines the expansion of input state samples into *initial_states* for the Tabu search, if fewer than *num_reads* samples are present. See `sample()`.

See *Examples*.

class TabuSubproblemSampler(*num_reads=None*, *tenure=None*, *timeout=100*, *initial_states_generator='random'*, ***runopts*)

A tabu sampler for a subproblem.

Parameters

- **num_reads** (*int, optional, default=len(state.subsamples) or 1*) – Number of states (output solutions) to read from the sampler.
- **tenure** (*int, optional*) – Tabu tenure, which is the length of the tabu list, or number of recently explored solutions kept in memory. Default is a quarter of the number of problem variables up to a maximum value of 20.
- **timeout** (*int, optional, default=100*) – Total running time in milliseconds.
- **initial_states_generator** (*str, 'none'/'tile'/'random', optional, default='random'*) – Defines the expansion of input state subsamples into *initial_states* for the Tabu search, if fewer than *num_reads* subsamples are present. See `sample()`.

See *Examples*.

Examples

QPUSubproblemExternalEmbeddingSampler

This example works on a binary quadratic model of two AND gates in series by sampling a BQM representing just one of the gates. Output z of gate $z = x \wedge y$ connects to input a of gate $c = a \wedge b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated by sampling the subproblem 100 times on a D-Wave system. The execution results shown here were three valid solutions to the subproblem; for example, $x = 0, y = 1, z = 0$ occurred 22 times.

```
import dimod
import minorminer
from dwave.system.samplers import DWaveSampler
from hybrid.samplers import QPUSubproblemExternalEmbeddingSampler
from hybrid.core import State

# Define a problem and a subproblem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c
↪': 6.0},
                                 {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0,
                                 ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0,␣
↪('a', 'z'): -4.0},
                                 -1.0, 'BINARY')
sub_bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0},
                                     {('x', 'y'): 2.0, ('x', 'z'): -4.0, ('y', 'z'): -
↪4.0},
                                     -1.0, dimod.Vartype.BINARY)

# Find a minor-embedding for the subproblem
qpu_sampler = DWaveSampler()
sub_embedding = minorminer.find_embedding(list(sub_bqm.quadratic.keys()), qpu_sampler.
↪edgelist)

# Set up the sampler with an initial state
```

(continues on next page)

(continued from previous page)

```
sampler = QPUSubproblemExternalEmbeddingSampler(num_reads=100)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
state.update(subproblem=sub_bqm, embedding=sub_embedding)
```

```
# Sample the subproblem on the QPU (REPL)
>>> new_state = sampler.run(state).result()
>>> print(new_state.subsamples.record)
[([0, 1, 0], -1., 22) ([0, 0, 0], -1., 47) ([1, 0, 0], -1., 31)]
```

QPUSubproblemAutoEmbeddingSampler

This example works on a binary quadratic model of two AND gates in series by sampling a BQM representing just one of the gates. Output z of gate $z = x \wedge y$ connects to input a of gate $c = a \wedge b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated by sampling the subproblem 100 times on a D-Wave system. The execution results shown here were four valid solutions to the subproblem; for example, $x = 0, y = 0, z = 0$ occurred 53 times.

```
import dimod
from hybrid.samplers import QPUSubproblemAutoEmbeddingSampler
from hybrid.core import State

# Define a problem and a subproblem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c
↪': 6.0},
                                 {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0,
                                  ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0,␣
↪('a', 'z'): -4.0},
                                 -1.0, 'BINARY')
sub_bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0},
                                     {('x', 'y'): 2.0, ('x', 'z'): -4.0, ('y', 'z'): -
↪4.0},
                                     -1.0, dimod.Vartype.BINARY)

# Set up the sampler with an initial state
sampler = QPUSubproblemAutoEmbeddingSampler(num_reads=100)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
state.update(subproblem=sub_bqm)
```

```
# Sample the subproblem on the QPU (REPL)
>>> new_state = sampler.run(state).result()
>>> print(new_state.subsamples.record)
[([0, 0, 0], -1., 53) ([0, 1, 0], -1., 15) ([1, 0, 0], -1., 31) ([1, 1, 1],  1.,  1)]
```

SimulatedAnnealingSubproblemSampler

This example works on a binary quadratic model of two AND gates in series by sampling a BQM representing just one of the gates. Output z of gate $z = x \wedge y$ connects to input a of gate $c = a \wedge b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated by sampling the subproblem 10 times. The execution results shown here were valid solutions to the subproblem; for example, $x = 0, y = 1, z = 0$.

```
import dimod
from hybrid.samplers import SimulatedAnnealingSubproblemSampler
```

(continues on next page)

(continued from previous page)

```
from hybrid.core import State

# Define a problem and a subproblem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c
↪': 6.0},
                                 {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0,
                                 ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0, (
↪'a', 'z'): -4.0},
                                 -1.0, 'BINARY')
sub_bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0},
                                     {('x', 'y'): 2.0, ('x', 'z'): -4.0, ('y', 'z'): -
↪4.0},
                                     -1.0, dimod.Vartype.BINARY)

# Set up the sampler with an initial state
sampler = SimulatedAnnealingSubproblemSampler(num_reads=10)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
state.update(subproblem=sub_bqm)
```

```
# Sample the subproblem (REPL)
>>> new_state = sampler.run(state).result()
>>> print(new_state.subsamples.record)
[([0, 1, 0], -1., 1) ([0, 1, 0], -1., 1) ([0, 0, 0], -1., 1)
([0, 0, 0], -1., 1) ([0, 0, 0], -1., 1) ([1, 0, 0], -1., 1)
([1, 0, 0], -1., 1) ([0, 0, 0], -1., 1) ([0, 1, 0], -1., 1)
([1, 0, 0], -1., 1)]
```

TabuSubproblemSampler

This example works on a binary quadratic model of two AND gates in series by sampling a BQM representing just one of the gates. Output z of gate $z = x \wedge y$ connects to input a of gate $c = a \wedge b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated by a tabu search on the subproblem. The execution results shown here was a valid solution to the subproblem: example, $x = 0, y = 1, z = 0$.

```
import dimod
from hybrid.samplers import TabuSubproblemSampler
from hybrid.core import State

# Define a problem and a subproblem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c
↪': 6.0},
                                 {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0,
                                  ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0,
↪ ('a', 'z'): -4.0},
                                 -1.0, 'BINARY')
sub_bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0},
                                     {('x', 'y'): 2.0, ('x', 'z'): -4.0, ('y', 'z'): -
↪4.0},
                                     -1.0, dimod.Vartype.BINARY)

# Set up the sampler with an initial state
sampler = TabuSubproblemSampler(tenure=2, timeout=5)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
state.update(subproblem=sub_bqm)
```

```
>>> # Sample the subproblem (REPL)
>>> print(new_state.subsamples.record)
[([0, 1, 0], -1., 1)]
```

TabuProblemSampler

This example works on a binary quadratic model of two AND gates in series, where output z of gate $z = x \wedge y$ connects to input a of gate $c = a \wedge b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated by a tabu search. The execution results shown here was a valid solution to the problem: example, $x = y = z = a = b = c = 1$.

```
import dimod
from hybrid.samplers import TabuProblemSampler
from hybrid.core import State

# Define a problem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c
↪': 6.0},
                                 {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0,
                                  ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0,
↪ ('a', 'z'): -4.0},
                                 -1.0, 'BINARY')

# Set up the sampler with an initial state
sampler = TabuProblemSampler(tenure=2, timeout=5)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
```

```
# Sample the problem (REPL)
>>> new_state = sampler.run(state).result()
>>> print(new_state.samples)
Response(rec.array([([1, 1, 1, 1, 1, 1], -1., 1)],
    dtype=[('sample', 'i1', (6,)), ('energy', '<f8'), ('num_occurrences', '<i4')]),
    ['a', 'b', 'c', 'x', 'y', 'z'], {}, 'BINARY')
```

InterruptableTabuSampler

This example works on a binary quadratic model of two AND gates in series, where output z of gate $z = x \wedge y$ connects to input a of gate $c = a \wedge b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated by a tabu search. The execution results shown here was a valid solution to the problem: example, $x = y = z = a = b = c = 1$.

```
import dimod
from hybrid.samplers import InterruptableTabuSampler
from hybrid.core import State

# Define a problem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c
↪': 6.0},
                                 {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0,
                                  ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0,
↪ ('a', 'z'): -4.0},
                                 -1.0, 'BINARY')
```

(continues on next page)

(continued from previous page)

```
# Set up the sampler with an initial state
sampler = InterruptableTabuSampler(tenure=2, quantum_timeout=30, timeout=5000)
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
```

```
>>> # Sample the problem (REPL)
>>> new_state = sampler.run(state)
>>> new_state
<Future at 0x179eae59898 state=running>
>>> sampler.stop()
>>> new_state
<Future at 0x179eae59898 state=finished returned State>
>>> print(new_state.result())
State(samples=Response(rec.array([([1, 1, 1, 1, 1, 1], -1., 1)],
    dtype=[('sample', 'i1', (6,)), ('energy', '<f8'), ('num_occurrences', '<i4')]),
    ['a', 'b', 'c', 'x', 'y', 'z'], {}, 'BINARY'))
```

RandomSubproblemSampler

This example works on a binary quadratic model of two AND gates in series by sampling a BQM representing just one of the gates. Output z of gate $z = x \wedge y$ connects to input a of gate $c = a \wedge b$. An initial state is manually set with invalid solution $x = y = 0, z = 1; a = b = 1, c = 0$. The state is updated with a random sample..

```
import dimod
from hybrid.samplers import RandomSubproblemSampler
from hybrid.core import State

# Define a problem and a subproblem
bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0, 'a': 2.0, 'b': 0.0, 'c
↪': 6.0},
                                 {('y', 'x'): 2.0, ('z', 'x'): -4.0, ('z', 'y'): -4.0,
                                  ('b', 'a'): 2.0, ('c', 'a'): -4.0, ('c', 'b'): -4.0,
↪ ('a', 'z'): -4.0},
                                 -1.0, 'BINARY')
sub_bqm = dimod.BinaryQuadraticModel({'x': 0.0, 'y': 0.0, 'z': 8.0},
                                     {('x', 'y'): 2.0, ('x', 'z'): -4.0, ('y', 'z'): -
↪4.0},
                                     -1.0, dimod.Vartype.BINARY)

# Set up the sampler with an initial state
sampler = RandomSubproblemSampler()
state = State.from_sample({'x': 0, 'y': 0, 'z': 1, 'a': 1, 'b': 1, 'c': 0}, bqm)
state.update(subproblem=sub_bqm)
```

```
# Sample the subproblem a couple of times (REPL)
>>> new_state = sampler.run(state).result()
>>> print(new_state.subsamples.record)
[([0, 0, 0], -1., 1)]
>>> new_state = sampler.run(state).result()
>>> print(new_state.subsamples.record)
[([1, 1, 1], 1., 1)]
```

2.2.3 Composers

Class

class IdentityComposer
Copy *subsamples* to *samples* verbatim.

class SplatComposer
A composer that overwrites current samples with subproblem samples.

See *Examples*.

class GreedyPathMerge
Dialectic-search merge operation [KS]. Generates a path from one input state, representing the thesis, to another input state, representing the antithesis, using a greedy method of single bit flips selected by decreasing energy.

Returns the best sample on the path, which represents the synthesis.

Note: only the lowest-energy sample, is considered from either input state.

See *Examples*.

References

Examples

SplatComposer

This example runs one iteration of a *SplatComposer* composer, overwriting an initial solution to a 6-variable binary quadratic model of all zeros with a solution to a 3-variable subproblem that was manually set to all ones.

```
import dimod
from hybrid.composers import SplatComposer
from hybrid.core import State, SampleSet
from hybrid.utils import min_sample

bqm = dimod.BinaryQuadraticModel({t: 0 for t in range(6)},
                                 {(t, (t+1) % 6): 1 for t in range(6)},
                                 0, 'BINARY')

composer = SplatComposer()
state0 = State.from_sample(min_sample(bqm), bqm)
state1 = state0.updated(subsamples=SampleSet.from_samples({3: 1, 4: 1, 5: 1}, 'BINARY
↪', 0.0))

composed_state = composer.run(state1).result()
```

```
>>> print(composed_state.samples)
Response(rec.array([([0, 0, 0, 1, 1, 1], 1, 2)],
        dtype=[('sample', 'i1', (6,)), ('num_occurrences', '<i8'), ('energy', '<i8
↪')]), [0, 1, 2, 3, 4, 5], {}, 'BINARY')
```

GreedyPathMerge

This example runs one iteration of a *GreedyPathMerge* composer on a thesis and antithesis `State` to find a ground state of a square graph. By inverting the state of variable d and c in *samples_d* and then variable a of the lowest energy sample of *samples_a* (second sample), the composer finds a path between these two samples that contains the ground state shown on the right of the top figure.

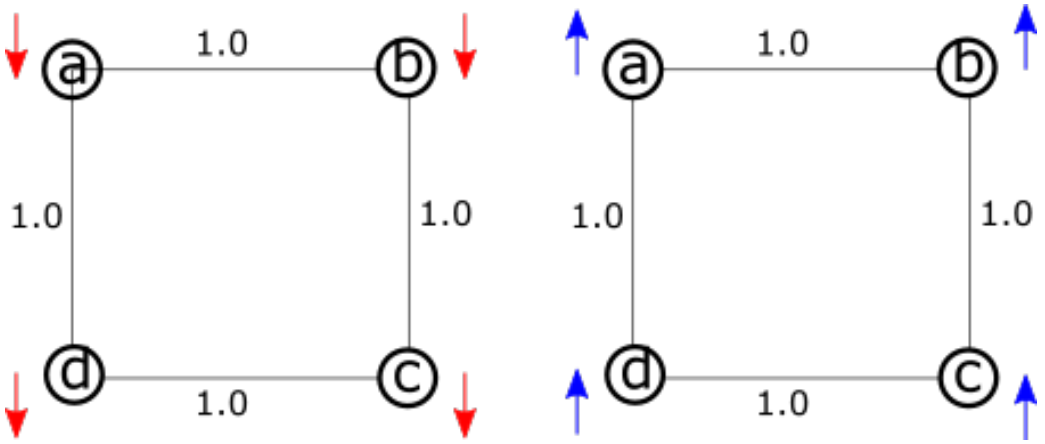


Fig. 6: Square problem with two ground states.

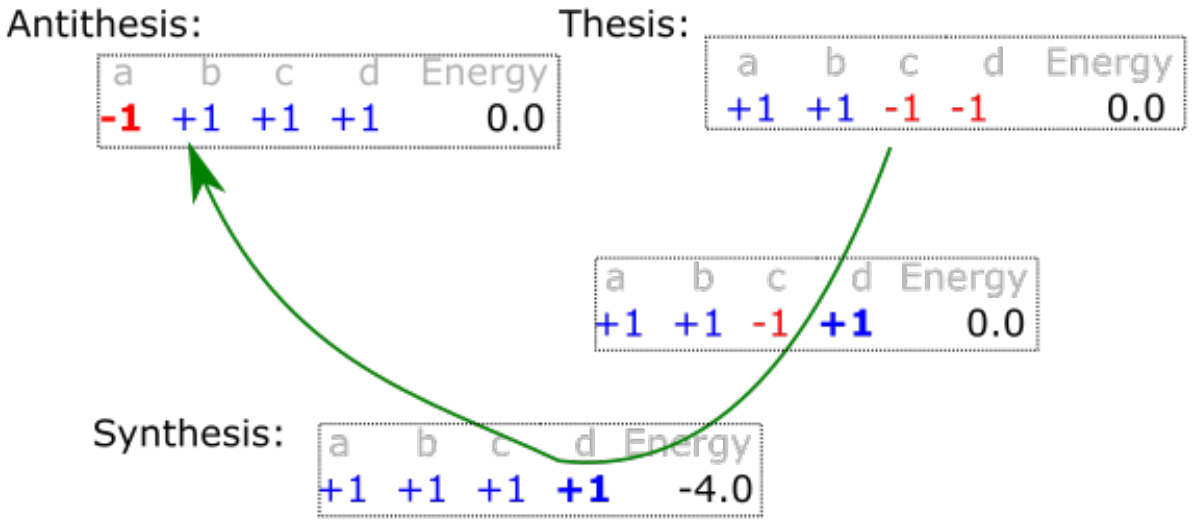


Fig. 7: Path from thesis to antithesis.

```
import dimod
bqm = dimod.BinaryQuadraticModel({}, {'ab': 1.0, 'bc': 1.0, 'cd': 1.0, 'da': 1}, 0,
↪'SPIN')
samples_d = {'a': 1, 'b': 1, 'c': -1, 'd': -1}
samples_a = [{'a': -1, 'b': -1, 'c': 1, 'd': 1}, {'a': -1, 'b': 1, 'c': 1, 'd': 1}]
states = [hybrid.State.from_samples(samples_d, bqm),
          hybrid.State.from_samples(samples_a, bqm)]
synthesis = GreedyPathMerge().next(states)
```

```
>>> print(synthesis.samples)
       a   b   c   d  energy  num_occ.
   0  +1  +1  +1  +1    -4.0         1
   [ 1 rows, 4 variables ]
```

2.2.4 Decomposers

Classes

class EnergyImpactDecomposer(*size*, *min_gain=None*, *rolling=True*, *rolling_history=1.0*, *silent_rewind=True*, *traversal='energy'*, ***runopts*)

Selects a subproblem of variables maximally contributing to the problem energy.

The selection currently implemented does not ensure that the variables are connected in the problem graph.

Parameters

- **size** (*int*) – Nominal number of variables in the subproblem. Actual subproblem can be smaller, depending on other parameters (e.g. *min_gain*).
- **min_gain** (*int, optional, default=-inf*) – Minimum reduction required to BQM energy, given the current sample. A variable is included in the subproblem only if inverting its sample value reduces energy by at least this amount.
- **rolling** (*bool, optional, default=True*) – If True, successive calls for the same problem (with possibly different samples) produce subproblems on different variables, selected by rolling down the list of all variables sorted by decreasing impact.
- **rolling_history** (*float, optional, default=1.0*) – Fraction of the problem size, as a float in range 0.0 to 1.0, that should participate in the rolling selection. Once reached, subproblem unrolling is reset.
- **silent_rewind** (*bool, optional, default=True*) – If False, raises `EndOfStream` when resetting/rewinding the subproblem generator upon the reset condition for unrolling.
- **traversal** (*str, optional, default='energy'*) – Traversal algorithm used to pick a subproblem of *size* variables. Options are:

  energy: Use the next *size* variables in the list of variables ordered by descending energy impact.

  bfs: Breadth-first traversal seeded by the next variable in the energy impact list.

  pfs: Priority-first traversal seeded by variables from the energy impact list, proceeding with the variable on the search boundary that has the highest energy impact.

See *Examples*.

class IdentityDecomposer
Selects a subproblem that is a full copy of the problem.

class RandomConstraintDecomposer(*size*, *constraints*, ***runopts*)
Selects variables randomly as constrained by groupings.

By grouping related variables, the problem's structure can guide the random selection of variables so subproblems are related to the problem's constraints.

Parameters

- **size** (*int*) – Number of variables in the subproblem.
- **constraints** (*list[set]*) – Groups of variables in the BQM, as a list of sets, where each set is associated with a constraint.

See *Examples*.

class RandomSubproblemDecomposer(*size*, ***runopts*)
Selects a subproblem of *size* random variables.

The selection currently implemented does not ensure that the variables are connected in the problem graph.

Parameters **size** (*int*) – Number of variables in the subproblem.

See *Examples*.

class RoofDualityDecomposer(*sampling_mode=True*, ***runopts*)
Selects a subproblem with variables that cannot be fixed by roof duality.

Roof duality finds a lower bound for the minimum of a quadratic polynomial. It can also find minimizing assignments for some of the polynomial's variables; these fixed variables take the same values in all optimal solutions [BHT] [BH]. A quadratic pseudo-Boolean function can be represented as a network to find the lower bound through network-flow computations. This decomposer can also use maximum flow in the implication network to fix variables. Consequently, you can find an assignment for the remaining variables that attains the optimal value.

Parameters **sampling_mode** (*bool, optional, default=True*) – In sampling mode, only roof-duality is used. When *sampling_mode* is false, strongly connected components are used to fix more variables, but in some optimal solutions these variables may take different values.

class TilingChimeraDecomposer(*size=(4, 4, 4)*, *loop=True*, ***runopts*)
Returns sequential Chimera lattices that tile the initial problem.

A Chimera lattice is an m-by-n grid of Chimera tiles, where each tile is a bipartite graph with shores of size t. The problem is decomposed into a sequence of subproblems with variables belonging to the Chimera lattices that tile the problem Chimera lattice. For example, a 2x2 Chimera lattice could be tiled 64 times (8x8) on a fully-yielded D-Wave 2000Q system (16x16).

Parameters

- **size** (*int, optional, default=(4,4,4)*) – Size of the Chimera lattice as (m, n, t), where m is the number of rows, n the columns, and t the size of shore in the Chimera lattice.
- **loop** (*Bool, optional, default=True*) – Cycle continually through the tiles.

See *Examples*.

Examples

EnergyImpactDecomposer

This example iterates twice on a 10-variable binary quadratic model with a random initial sample set. *size* configuration limits the subproblem in the first iteration to the first 4 variables shown in the output of *flip_energy_gains*.

```
import dimod
from hybrid.decomposers import EnergyImpactDecomposer
from hybrid.core import State
from hybrid.utils import min_sample, flip_energy_gains

bqm = dimod.BinaryQuadraticModel({t: 0 for t in range(10)},
                                 {(t, (t+1) % 10): 1 for t in range(10)},
                                 0, 'BINARY')

decomposer = EnergyImpactDecomposer(size=4, rolling=True, rolling_history=1.0)
state0 = State.from_sample(min_sample(bqm), bqm)
```

```
>>> flip_energy_gains(bqm, state0.samples.first.sample)
[(0, 9), (0, 8), (0, 7), (0, 6), (0, 5), (0, 4), (0, 3), (0, 2), (0, 1), (0, 0)]
>>> state1 = decomposer.run(state0).result()
>>> list(state1.subproblem.variables)
[8, 7, 9, 6]
>>> state2 = decomposer.run(state1).result()
>>> list(state2.subproblem.variables)
[2, 3, 4, 5]
```

RandomSubproblemDecomposer

This example decomposes a 6-variable binary quadratic model with a random initial sample set to create a 3-variable subproblem.

```
import dimod
from hybrid.decomposers import RandomSubproblemDecomposer
from hybrid.core import State
from hybrid.utils import random_sample

bqm = dimod.BinaryQuadraticModel(
    {t: 0 for t in range(6)}, {(t, (t+1) % 6): 1 for t in range(6)}, 0, 'BINARY')

decomposer = RandomSubproblemDecomposer(bqm, size=3)
state0 = State.from_sample(random_sample(bqm), bqm)
state1 = decomposer.run(state0).result()
```

```
>>> print(state1.subproblem)
BinaryQuadraticModel({2: 1.0, 3: 0.0, 4: 0.0}, {(2, 3): 1.0, (3, 4): 1.0}, 0.0,␣
↪Vartype.BINARY)
```

TilingChimeraDecomposer

This example decomposes a 2048-variable Chimera structured binary quadratic model read from a file into 2x2x4-lattice subproblems.

```
import dimod
from hybrid.decomposers import TilingChimeraDecomposer
from hybrid.core import State
from hybrid.utils import random_sample

with open('problems/random-chimera/2048.09.qubo', 'r') as fp:
    bqm = dimod.BinaryQuadraticModel.from_coo(fp)

decomposer = TilingChimeraDecomposer(size=(2,2,4))
state0 = State.from_sample(random_sample(bqm), bqm)
state1 = decomposer.run(state0).result()
```

```
>>> print(state1.subproblem)
BinaryQuadraticModel({0: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: -3.0, 1: 0.0, 2: 0.0, 3: -4.
↪0, 1024: -7.0, 1028: 0.0,
>>> # Snipped above response for brevity
>>> state1 = decomposer.run(state0).result()
>>> print(state1.subproblem)
BinaryQuadraticModel({8: 3.0, 12: 0.0, 13: 2.0, 14: -11.0, 15: -3.0, 9: 4.0, 10: 0.0,␣
↪11: 0.0, 1032: 0.0,
>>> # Snipped above response for brevity
```

RandomConstraintDecomposer

This example decomposes a 4-variable binary quadratic model that represents three serial NOT gates into 2-variable subproblems. The expected decomposition should use variables that represent one of the NOT gates rather than two arbitrary variables.

```
import dimod
from hybrid.decomposers  RandomConstraintDecomposer
from hybrid.core import State
from hybrid.utils import random_sample

bqm = dimod.BinaryQuadraticModel({'w': -2.0, 'x': -4.0, 'y': -4.0, 'z': -2.0},
                                 {('w', 'x'): 4.0, ('x', 'y'): 4.0, ('y', 'z'): 4.0},
                                 3.0, 'BINARY')

decomposer = RandomConstraintDecomposer(2, [{'w', 'x'}, {'x', 'y'}, {'y', 'z'}])
state0 = State.from_sample(random_sample(bqm), bqm)
state1 = decomposer.run(state0).result()
```

```
>>> print(state1.subproblem)
BinaryQuadraticModel({'z': -2.0, 'y': 0.0}, {('z', 'y'): 4.0}, 0.0, Vartype.BINARY)
```

2.2.5 Flow Structuring

Classes that structure (hybrid) workflows.

Classes

class ArgMin(*key=None*, ***runopts*)

Selects the best state from a sequence of *States*.

Parameters **key** (*callable/str*) – Best state is judged according to a metric defined with a *key*. The *key* can be a *callable* with a signature:

```
key :: (State s, Ord k) => s -> k
```

or a string holding a key name/path to be extracted from the input state with *operator.attrgetter* method.

By default, *key == operator.attrgetter('samples.first.energy')*, thus favoring states containing a sample with the minimal energy.

Examples

This example runs two branches—a classical tabu search interrupted by samples of subproblems returned from a D-Wave system— and selects the state with the minimum-energy sample:

```
RacingBranches(
    InterruptableTabuSampler(),
    EnergyImpactDecomposer(size=2)
    | QPUSubproblemAutoEmbeddingSampler()
    | SplatComposer()
) | ArgMin()
```

class Branch(*components=()*, ***runopts*)

Sequentially executed *Runnable* components.

Parameters **components** (iterable of *Runnable*) – Complete processing sequence to update a current set of samples, such as: `decomposer | sampler | composer`.

Input: Defined by the first branch component.

Output: Defined by the last branch component.

Examples

This example runs one iteration of a branch comprising a decomposer, local Tabu solver, and a composer. A 10-variable binary quadratic model is decomposed by the energy impact of its variables into a 6-variable subproblem to be sampled twice with a manually set initial state of all -1 values.

```
>>> import dimod           # Create a binary quadratic model
>>> bqm = dimod.BQM({t: 0 for t in range(10)},
...                 {(t, (t+1) % 10): 1 for t in range(10)},
...                 0, 'SPIN')
>>> # Run one iteration on a branch
>>> branch = (EnergyImpactDecomposer(size=6, min_gain=-10) |
...           TabuSubproblemSampler(num_reads=2) |
...           SplatComposer())
>>> new_state = branch.next(State.from_sample(min_sample(bqm), bqm))
>>> print(new_state.subsamples)      # doctest: +SKIP
       4   5   6   7   8   9  energy  num_occ.
   0  +1  -1  -1  +1  -1  +1    -5.0         1
   1  +1  -1  -1  +1  -1  +1    -5.0         1
   [ 2 rows, 6 variables ]
```

class Branches(**branches*, ***runopts*)

Runs multiple workflows of type *Runnable* in parallel, blocking until all finish.

Branches operates similarly to *ParallelBranches*, but each branch runs on a separate input *State* (while parallel branches all use the same input state).

Parameters ***branches** ([*Runnable*]) – Runnable branches listed as positional arguments.

Input: *States*

Output: *States*

Note: *Branches* is also available via implicit parallelization binary operator *&*.

Examples

This example runs two branches, a classical tabu search and a random sampler, until both terminate:

```
Branches(TabuSubproblemSampler(), RandomSubproblemSampler())
```

Alternatively:

```
TabuSubproblemSampler() & RandomSubproblemSampler()
```

class Const(***consts*)

Set state variables to constant values.

Parameters ****consts** (*dict, optional*) – Mapping of state variables to constant values, as keyword arguments.

Example

This example defines a workflow that resets the set of samples before a Tabu sampler call in order to avoid using existing samples as initial states. Instead, Tabu will use randomly generated initial states:

```
random_tabu = Const(samples=None) | TabuProblemSampler(initial_states_generator=
↪'random')
```

class Dup(*n*, **args*, ***kwargs*)

Duplicates input *State*, n times, into output *States*.

class Identity

Trivial identity runnable. The output is a direct copy of the input.

class BlockingIdentity(**args*, ***kwargs*)

Trivial identity runnable that blocks indefinitely before producing output, but is interruptable. The output is a direct copy of the input, but to receive the output, the block has to be explicitly stopped (useful for example in *RacingBranches* to prevent short-circuiting of racing branches with the identity branch).

```
BlockingIdentity := Identity | Wait
```

Due to nature of *Identity*, *BlockingIdentity* is functionally equivalent to *Wait*.

class Lambda(*next*, *error=None*, *init=None*, ***runopts*)

Creates a runnable on fly, given just its *next* function (optionally *init* and *error* functions can be specified too).

Parameters

- **next** (*callable*) – Implementation of runnable's *next* method, provided as a callable (usually a lambda expression for simple operations). Signature of the callable has to match the signature of *next()*; i.e., it accepts two arguments: runnable instance and state instance.
- **error** (*callable*) – Implementation of runnable's *error* method. See *error()*.
- **init** (*callable*) – Implementation of runnable's *init* method. See *init()*.

Note: Traits are not enforced, apart from the SISO requirement. Also, note *Lambda* runnables can only implement SISO systems.

Examples

This example creates and runs a simple runnable that multiplies state variables *a* and *b*, storing them in *c*.

```
>>> Lambda(lambda _, s: s.updated(c=s.a * s.b)).run(State(a=2, b=3)).result()
↪# doctest: +SKIP
{'a': 2, 'b': 3, 'c': 6}
```

This example applies *x += 1* to a sequence of input states.

```
>>> Map(Lambda(lambda _, s: s.updated(x=s.x + 1))).run(States(State(x=0),␣
↪State(x=1))).result()
[{'x': 1}, {'x': 2}]
```

class Loop(**args*, ***kwargs*)
Alias for *LoopUntilNoImprovement*.

class LoopUntilNoImprovement(**args*, ***kwargs*)
Iterates *Runnable* for up to *max_iter* times, or until a state quality metric, defined by the *key* function, shows no improvement for at least *convergence* number of iterations. Alternatively, maximum allowed runtime can be defined with *max_time*, or a custom termination Boolean function can be given with *terminate* (a predicate on *key*). Loop is always terminated on `EndOfStream` raised by body runnable.

Parameters

- **runnable** (*Runnable*) – A runnable that's looped over.
- **max_iter** (*int/None, optional, default=None*) – Maximum number of times the *runnable* is run, regardless of other termination criteria. This is the upper bound. By default, an upper bound on the number of iterations is not set.
- **convergence** (*int/None, optional, default=None*) – Terminates upon reaching this number of iterations with unchanged output. By default, convergence is not checked, so the only termination criteria is defined with *max_iter*. Setting neither creates an infinite loop.
- **max_time** (*float/None, optional, default=None*) – Wall clock runtime termination criterion. Unlimited by default.
- **key** (*callable/str*) – Best state is judged according to a metric defined with a *key*. *key* can be a *callable* with a signature:

```
key :: (State s, Ord k) => s -> k
```

or a string holding a key name/path to be extracted from the input state with *operator.attrgetter* method.

By default, *key* == *operator.attrgetter('samples.first.energy')*, thus favoring states containing a sample with the minimal energy.

- **terminate** (*callable, optional, default=None*) – Loop termination Boolean function (a predicate on *key* value):

```
terminate :: (Ord k) => k -> Bool
```

class LoopWhileNoImprovement(*runnable*, *max_iter=None*, *max_tries=None*, *max_time=None*, *key=None*, *terminate=None*, ***runopts*)

Iterates *Runnable* until a state quality metric, defined by the *key* function, shows no improvement for at least *max_tries* number of iterations or until *max_iter* number of iterations is exceeded. Alternatively, maximum allowed runtime can be defined with *max_time*, or a custom termination Boolean function can be given with *terminate* (a predicate on *key*).

Note: Unlike *LoopUntilNoImprovement/Loop*, *LoopWhileNoImprovement* will run the loop body runnable with the **same input** if output shows no improvement (up to *max_tries* times), and it will use the new output if it's better than the input.

Parameters

- **runnable** (*Runnable*) – A runnable that's looped over.
- **max_iter** (*int/None, optional, default=None*) – Maximum number of times the *runnable* is run, regardless of other termination criteria. This is the upper bound. By default, an upper bound on the number of iterations is not set.
- **max_tries** (*int, optional, default=None*) – Maximum number of times the *runnable* is run for the **same** input state. On each improvement, the better state is used for the next input state, and the try/trial counter is reset. Defaults to an infinite loop (unbounded number of tries).
- **max_time** (*float/None, optional, default=None*) – Wall clock runtime termination criterion. Unlimited by default.
- **key** (*callable/str*) – Best state is judged according to a metric defined with a *key*. *key* can be a *callable* with a signature:

```
key :: (State s, Ord k) => s -> k
```

or a string holding a key name/path to be extracted from the input state with *operator.attrgetter* method.

By default, *key* == *operator.attrgetter('samples.first.energy')*, thus favoring states containing a sample with the minimal energy.

- **terminate** (*callable, optional, default=None*) – Loop termination Boolean function (a predicate on *key* value):

```
terminate :: (Ord k) => k -> Bool
```

class Map(*runnable*, ***runopts*)

Runs a specified *Runnable* in parallel on all input states.

Parameters **runnable** (*Runnable*) – A runnable executed for every input state.

Examples

This example runs *TabuProblemSampler* on two input states in parallel, returning when both are done.

```
>>> states = States(State(problem=bqm1), State(problem=bqm2))    # doctest: +SKIP
>>> Map(TabuProblemSampler()).run(states).result()              # doctest: +SKIP
[<state_1_with_solution>, <state_2_with_solution>]
```

Parallel
alias of *hybrid.flow.ParallelBranches*

class ParallelBranches(**branches*, ***runopts*)
Runs multiple workflows of type *Runnable* in parallel, blocking until all finish.

Parallel/ParallelBranches operates similarly to *Branches*, but every branch re-uses the same input *State*.

Parameters ***branches** ([*Runnable*]) – Comma-separated branches.

Input: *State*

Output: *States*

Note: *Parallel* is implemented as:

```
Parallel(*branches) := Dup(len(branches)) | Branches(*branches)
```

Note: *ParallelBranches* is also available as *Parallel*.

Examples

This example runs two branches, a classical tabu search and a random sampler, until both terminate:

```
Parallel(
    TabuSubproblemSampler(),
    RandomSubproblemSampler()
) | ArgMin()
```

Race
alias of *hybrid.flow.RacingBranches*

class RacingBranches(**branches*, ***runopts*)
Runs (races) multiple workflows of type *Runnable* in parallel, stopping all once the first finishes. Returns the results of all, in the specified order.

Parameters ***branches** ([*Runnable*]) – Comma-separated branches.

Note: Each branch runnable is called with run option `racing_context=True`, so it can adapt its behaviour to the context.

Note: *RacingBranches* is also available as *Race*.

Examples

This example runs two branches: a classical tabu search interrupted by samples of subproblems returned from a D-Wave system.

```
RacingBranches(
    InterruptableTabuSampler(),
    EnergyImpactDecomposer(size=2)
    | QPUSubproblemAutoEmbeddingSampler()
    | SplatComposer()
) | ArgMin()
```

class Reduce(*runnable*, *initial_state=None*, ***runopts*)

Fold-left using the specified *Runnable* on a sequence of input states, producing a single output state.

Parameters

- **runnable** (*Runnable*) – A runnable used as the fold-left operator. It should accept a 2-State input and produce a single State on output.
- **initial_state** (*State*, optional, default=None) – Optional starting state into which input states will be folded in. If undefined, the first input state is used as the *initial_state*.

class TrackMin(*key=None*, *output=False*, *input_key='samples'*, *output_key='samples'*, ***runopts*)

Tracks and records the best *State* according to a metric defined with a *key* function; typically this is the minimal state.

Parameters

- **key** (*callable/str, optional, default=None*) – Best state is judged according to a metric defined with a *key*. *key* can be a *callable* with a signature:

  ```
  key :: (State s, Ord k) => s -> k
  ```

  or a string holding a key name/path to be extracted from the input state with *operator.attrgetter* method.

  By default, *key* == *operator.attrgetter('samples.first.energy')*, thus favoring states containing a sample with the minimal energy.

- **output** (*bool, optional, default=False*) – Update the output state's *output_key* with the *input_key* of the best state seen so far.
- **input_key** (*str, optional, default='samples'*) – If *output=True*, then this defines the variable/key name in the input state that shall be included in the output state.
- **output_key** (*str, optional, default='samples'*) – If *output=True*, then the key under which the *input_key* from the best state seen so far is stored in the output state.

Note: If *output* option is turned on, and *output_key* is not changed, the output will by default change the state's *samples* on output.

---

class Unwind(*runnable*, ***runopts*)

Iterates *Runnable* until *EndOfStream* is raised, collecting all output states along the way.

Note: the child runnable is called with run option `silent_rewind=False`, and it is expected to raise *EndOfStream* on unwind completion.

---

class Wait (**args*, ***kwargs*)
Run indefinitely (effectively blocking branch execution). Has to be explicitly stopped.

Example

To effectively exclude one branch from the race, i.e. prevent premature stopping of the race between the remaining branches, use *Wait* as the last element in a (fast-executing) racing branch:

```
Race(
    Identity() | Wait(),
    InterruptableTabuSampler(),
    SimulatedAnnealingProblemSampler()
)
```

This is functionally identical to:

```
Parallel(
    Identity(),
    Race(
        InterruptableTabuSampler(),
        SimulatedAnnealingProblemSampler()
    )
)
```

Methods

See *Primitives* for methods inherited from the *Runnable* superclass.

2.2.6 Utilities

Methods

bqm_edges_between_variables(bqm, variables)	Return edges connecting specified variables of a binary quadratic model.
bqm_induced_by(bqm, variables, sample)	Induce a binary quadratic model by fixing values of boundary variables.
bqm_reduced_to(bqm, variables, sample[, ...])	Reduce a binary quadratic model by fixing values of some variables.
chimera_tiles(bqm, m, n, t)	Map a binary quadratic model to a set of Chimera tiles.
flip_energy_gains(bqm, sample[, variables, ...])	Order variable flips by descending contribution to energy changes in a BQM.
max_sample(bqm)	Return a sample with all variables set to the maximal value for a binary quadratic model.
min_sample(bqm)	Return a sample with all variables set to the minimal value for a binary quadratic model.
random_sample(bqm)	Return a random sample for a binary quadratic model.
random_sample_seq(size, vartype)	Return a random sample.
sample_as_dict(sample)	Return sample object in dict format.
sample_as_list(sample)	Return sample object in list format.

Continued on next page

Table 3 – continued from previous page

select_localsearch_adversaries(bqm, sample)	Find variable flips that contribute high energy changes to a BQM.
select_random_subgraph(bqm, n)	Select randomly *n* variables of the specified binary quadratic model.
updated_sample(sample, replacements)	Update a copy of a sample with replacement values.

hybrid.utils.bqm_edges_between_variables

bqm_edges_between_variables (*bqm*, *variables*)

Return edges connecting specified variables of a binary quadratic model.

Parameters

- **bqm** (dimod.BinaryQuadraticModel) – Binary quadratic model (BQM).
- **variables** (*list/set*) – Subset of variables in the BQM.

Returns All edges connecting *variables* as tuples plus the variables themselves as tuples (v, v).

Return type list

Examples

This example returns connecting edges between 3 nodes of a BQM based on a 4-variable path graph.

```
>>> import dimod
>>> bqm = dimod.BQM({}, {(0, 1): 1, (1, 2): 1, (2, 3): 1}, 0, 'BINARY')
>>> bqm_edges_between_variables(bqm, {0, 1, 3})
[(0, 1), (0, 0), (1, 1), (3, 3)]
```

hybrid.utils.bqm_induced_by

bqm_induced_by (*bqm*, *variables*, *sample*)

Induce a binary quadratic model by fixing values of boundary variables.

The function is optimized for `len(variables) << len(bqm)`, that is, for fixing the majority of variables.

Parameters

- **bqm** (dimod.BinaryQuadraticModel) – Binary quadratic model (BQM).
- **variables** (*list/set*) – Subset of variables to keep in the reduced BQM, typically a subgraph.
- **sample** (*dict/list*) – Mapping of variable labels to values or a list when labels are sequential integers. Values are required only for boundary variables, that is, for variables with interactions with *variables* (having edges with non-zero quadratic biases connected to the subgraph).

Returns A BQM induced by fixing values of those variables adjacent to its subset of variables and setting the energy offset to zero.

Return type dimod.BinaryQuadraticModel

Examples

This example induces a 2-variable BQM from a 6-variable path graph—the subset of nodes 2 and 3 of nodes 0 to 5—by fixing values of boundary variables 1 and 4.

```
>>> import dimod
>>> import networkx as nx
>>> bqm = dimod.BinaryQuadraticModel({},
...     {edge: edge[0] + 0.5 for edge in set(nx.path_graph(6).edges)}, 0, 'BINARY
↪')
>>> sample = {1: 3, 4: 3}
>>> bqm_induced_by(bqm, [2, 3], sample)
BinaryQuadraticModel({2: 4.5, 3: 10.5}, {(2, 3): 2.5}, 0.0, Vartype.BINARY)
```

hybrid.utils.bqm_reduced_to

bqm_reduced_to(*bqm*, *variables*, *sample*, *keep_offset=True*)

Reduce a binary quadratic model by fixing values of some variables.

The function is optimized for `len(variables) ~ len(bqm)`, that is, for small numbers of fixed variables.

Parameters

- **bqm** (`dimod.BinaryQuadraticModel`) – Binary quadratic model (BQM).
- **variables** (*list/set*) – Subset of variables to keep in the reduced BQM.
- **sample** (*dict/list*) – Mapping of variable labels to values or a list when labels are sequential integers. Must include all variables not specified in *variables*.
- **keep_offset** (*bool, optional, default=True*) – If false, set the reduced binary quadratic model's offset to zero; otherwise, uses the caluclated energy offset.

Returns A reduced BQM.

Return type `dimod.BinaryQuadraticModel`

Examples

This example reduces a 3-variable BQM to two variables.

```
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': -1, 'bc': -1, 'ca': -1}, 0, 'BINARY')
>>> sample = {'a': 1, 'b': 1, 'c': 0}
>>> subbqm = bqm_reduced_to(bqm, ['a', 'b'], sample)
>>> len(subbqm)
2
```

hybrid.utils.chimera_tiles

chimera_tiles(*bqm*, *m*, *n*, *t*)

Map a binary quadratic model to a set of Chimera tiles.

A Chimera lattice is an m-by-n grid of Chimera tiles, where each tile is a bipartite graph with shores of size t.

Parameters

- **bqm** (BinaryQuadraticModel) – Binary quadratic model (BQM).
- **m** (*int*) – Rows.
- **n** (*int*) – Columns.
- **t** (*int*) – Size of shore.

Returns Map as a dict where keys are tile coordinates (row, column, aisle) and values are partial embeddings of part of the BQM to a Chimera tile. Embeddings are those that would be generated by dwave_networkx's chimera_graph() function.

Return type dict

Examples

This example maps a 1-by-2 Chimera-derived BQM to 2 side-by-side tiles.

```
>>> import dwave_networkx as dnx
>>> import dimod
>>> G = dnx.chimera_graph(1, 2)      # Create a Chimera-based BQM
>>> bqm = dimod.BinaryQuadraticModel({}, {edge: edge[0] for edge in G.edges}, 0,
↪'BINARY')
>>> chimera_tiles(bqm, 1, 1, 4)      # doctest: +SKIP
{(0, 0, 0): {0: [0], 1: [1], 2: [2], 3: [3], 4: [4], 5: [5], 6: [6], 7: [7]},
 (0, 1, 0): {8: [0], 9: [1], 10: [2], 11: [3], 12: [4], 13: [5], 14: [6], 15: [7]}
↪}
```

hybrid.utils.flip_energy_gains

flip_energy_gains(*bqm*, *sample*, *variables=None*, *min_gain=None*)

Order variable flips by descending contribution to energy changes in a BQM.

Parameters

- **bqm** (dimod.BinaryQuadraticModel) – Binary quadratic model (BQM).
- **sample** (*list/dict*) – Sample values as returned by dimod samplers (0 or 1 values for *dimod.BINARY* and -1 or +1 for *dimod.SPIN*)
- **variables** (*sequence, optional, default=None*) – Consider only flips of these variables. If undefined, consider all variables in *sample*.
- **min_gain** (*float, optional, default=None*) – Minimum required energy increase from flipping a sample value to return its corresponding variable.

Returns

Energy changes in descending order, in the format of tuples (energy_gain, variable), for flipping the given sample value for each variable.

Return type list

Examples

This example returns connecting edges between 3 nodes of a BQM based on a 4-variable path graph.

```
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': 0, 'bc': 1, 'cd': 2}, 0, 'SPIN')
>>> flip_energy_gains(bqm, {'a': -1, 'b': 1, 'c': 1, 'd': -1})
[(4, 'd'), (2, 'c'), (0, 'a'), (-2, 'b')]
```

hybrid.utils.max_sample

max_sample(*bqm*)

Return a sample with all variables set to the maximal value for a binary quadratic model.

Parameters **bqm** (BinaryQuadraticModel) – Binary quadratic model (BQM).

Returns A sample with maximal values for all variables of the BQM.

Return type dict

Examples

```
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': -1, 'bc': -1, 'ca': -1}, 0, 'BINARY')
>>> max_sample(bqm)      # doctest: +SKIP
{'a': 1, 'b': 1, 'c': 1}
```

hybrid.utils.min_sample

min_sample(*bqm*)

Return a sample with all variables set to the minimal value for a binary quadratic model.

Parameters **bqm** (BinaryQuadraticModel) – Binary quadratic model (BQM).

Returns A sample with minimal values for all variables of the BQM.

Return type dict

Examples

```
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': -1, 'bc': -1, 'ca': -1}, 0, 'BINARY')
>>> min_sample(bqm)      # doctest: +SKIP
{'a': 0, 'b': 0, 'c': 0}
```

hybrid.utils.random_sample

random_sample(*bqm*)

Return a random sample for a binary quadratic model.

Parameters **bqm** (BinaryQuadraticModel) – Binary quadratic model (BQM).

Returns A sample with random values for the BQM.

Return type dict

Examples

```
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': -1, 'bc': -1, 'ca': -1}, 0, 'BINARY')
>>> random_sample(bqm)      # doctest: +SKIP
{'a': 0, 'b': 1, 'c': 1}
```

hybrid.utils.random_sample_seq

random_sample_seq(*size*, *vartype*)
Return a random sample.

Parameters

- **size** (*int*) – Sample size (number of variables).
- **vartype** (dimod.Vartype) – Variable type; for example, *Vartype.SPIN*, *BINARY*, or *{-1, 1}*.

Returns Random sample of *size* in length, with values from *vartype*.

Return type dict

Examples

```
>>> random_sample_seq(4, dimod.BINARY)       # doctest: +SKIP
{0: 0, 1: 1, 2: 0, 3: 0}
```

hybrid.utils.sample_as_dict

sample_as_dict(*sample*)
Return sample object in dict format.

Parameters

- **sample** (*list/dict/dimod.SampleView*) – Sample object formatted as a list,
- **array, dict, or as returned by dimod samplers.** (*Numpy*) –

Returns Copy of *sample* formatted as a dict, with variable indices as keys.

Return type list

Examples

```
>>> sample = [1, 2, 3]
>>> sample_as_dict(sample)      # doctest: +SKIP
{0: 1, 1: 2, 2: 3}
```

hybrid.utils.sample_as_list

sample_as_list(*sample*)
Return sample object in list format.

Parameters

- **sample** (*list/dict/dimod.SampleView*) – Sample object formatted as a list,
- **array, dict, or as returned by dimod samplers. Variable labeling** (*Numpy*) –
- **be numerical.** (*must*) –

Returns Copy of *sample* formatted as a list.

Return type list

Examples

```
>>> sample = {0: 1, 1: 1}
>>> sample_as_list(sample)
[1, 1]
```

hybrid.utils.select_localsearch_adversaries

select_localsearch_adversaries(*bqm*, *sample*, *max_n=None*, *min_gain=None*)

Find variable flips that contribute high energy changes to a BQM.

Parameters

- **bqm** (dimod.BinaryQuadraticModel) – Binary quadratic model (BQM).
- **sample** (*list/dict*) – Sample values as returned by dimod samplers (0 or 1 values for *dimod.BINARY* and -1 or +1 for *dimod.SPIN*)
- **max_n** (*int, optional, default=None*) – Maximum contributing variables to return. By default, returns any variable for which flipping its sample value results in an energy gain of *min_gain*.
- **min_gain** (*float, optional, default=None*) – Minimum required energy increase from flipping a sample value to return its corresponding variable.

Returns Up to *max_n* variables for which flipping the corresponding sample value increases the BQM energy by at least *min_gain*.

Return type list

Examples

This example returns 2 variables (out of up to 3 allowed) for which flipping sample values changes BQM energy by 1 or more. The BQM has energy gains of 0, -2, 2, 4 for variables a, b, c, d respectively for the given sample.

```
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': 0, 'bc': 1, 'cd': 2}, 0, 'SPIN')
>>> select_localsearch_adversaries(
...     bqm, {'a': -1, 'b': 1, 'c': 1, 'd': -1}, max_n=3, min_gain=1)
['d', 'c']
```

hybrid.utils.select_random_subgraph

select_random_subgraph(*bqm*, *n*)
Select randomly *n* variables of the specified binary quadratic model.

Parameters

- **bqm** (dimod.BinaryQuadraticModel) – Binary quadratic model (BQM).
- **n** (*int*) – Number of requested variables. Must be between 0 and *len(bqm)*.

Returns *n* variables selected randomly from the BQM.

Return type list

Examples

This example returns 2 variables of a 4-variable BQM.

```
>>> import dimod
>>> bqm = dimod.BQM({}, {'ab': 0, 'bc': 1, 'cd': 2}, 0, 'BINARY')
>>> select_random_subgraph(bqm, 2)       # doctest: +SKIP
['d', 'b']
```

hybrid.utils.updated_sample

updated_sample(*sample*, *replacements*)
Update a copy of a sample with replacement values.

Parameters

- **sample** (*list/dict*) – Sample values as returned by dimod samplers to be copied.
- **replacements** (*list/dict*) – Sample values to replace in the copied *sample*.

Returns Copy of *sample* overwritten by specified values.

Return type list/dict

Examples

```
>>> sample = {'a': 1, 'b': 1}
>>> updated_sample(sample, {'b': 2})        # doctest: +SKIP
{'a': 1, 'b': 2}
```

2.2.7 Dimod Conversion

These classes handle conversion between *dwave-hybrid* *Runnable* classes and dimod samplers.

Classes

class HybridSampler(*workflow*)
Produces a *dimod.Sampler* from a *hybrid.Runnable*-based sampler.

Parameters **workflow** (*Runnable*) – Hybrid workflow, likely composed, that accepts a binary quadratic model in the input state and produces sample(s) in the output state.

Example

This example produces a dimod sampler from *TabuProblemSampler* and uses its *sample_ising* mixin to solve a simple Ising problem.

```
>>> hybrid_sampler = TabuProblemSampler()
>>> dimod_sampler = HybridSampler(hybrid_sampler)
>>> solution = dimod_sampler.sample_ising({}, {'ab': 0.5, 'bc': 0.5, 'ca': 0.5})
>>> solution.first.energy
-0.5
```

class HybridRunnable(*sampler*, *fields*, ***sample_kwargs*)

Produces a *hybrid.Runnable* from a *dimod.Sampler* (dual of *HybridSampler*).

The runnable samples from a problem defined in a state field named *fields[0]* and populates the state field referred to by *fields[1]*.

Parameters

- **sampler** (*dimod.Sampler*) – dimod-compatible sampler which is run on every iteration of the runnable.
- **fields** (*tuple(str, str)*) – Input and output state field names.
- ****sample_kwargs** (*dict*) – Sampler-specific parameters passed to sampler on every call.

Example

This example creates a *Runnable* from dimod sampler TabuSampler, runs it on an Ising model, and finds the lowest energy.

```
>>> from tabu import TabuSampler
>>> import dimod
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {'ab': 0.5, 'bc': 0.5, 'ca':␣
↪0.5})
>>> runnable = HybridRunnable(TabuSampler(), fields=('subproblem', 'subsamples'),␣
↪timeout=100)
>>> state0 = State(subproblem=bqm, subsamples=SampleSet.from_samples_bqm(min_
↪sample(bqm), bqm))
>>> state = runnable.run(state0)
>>> state.result()['subsamples'].first.energy     # doctest: +SKIP
-0.5
```

class HybridProblemRunnable(*sampler*, ***sample_kwargs*)

Produces a *hybrid.Runnable* from a *dimod.Sampler* (dual of *HybridSampler*).

The runnable that samples from *state.problem* and populates *state.samples*.

See an example in *hybrid.core.HybridRunnable*. An example of the duality with *HybridSampler* is:

```
HybridProblemRunnable(HybridSampler(TabuProblemSampler())) == TabuProblemSampler()
```

class HybridSubproblemRunnable(*sampler*, ***sample_kwargs*)

Produces a *hybrid.Runnable* from a *dimod.Sampler* (dual of *HybridSampler*).

The runnable that samples from *state.subproblem* and populates *state.subsamples*.

See an example in `hybrid.core.HybridRunnable`.

2.2.8 Traits

State traits are verified for all `Runnable` objects that inherit from `StateTraits` or its subclasses. Verification includes:

(1) Minimal checks of workflow construction (composition of `Runnable` classes)

(2) Runtime checks

All built-in `Runnable` classes declare state traits requirements that are either independent (for simple ones) or derived from a child workflow. Traits of a new `Runnable` must be expressed and modified at construction time by its parent. When developing new `Runnable` classes, constructing composite traits can be nontrivial for some advanced flow-control runnables. State traits validation base class and related state validation mixins (i/o validation toggle, i/o dimensionality, state structure).

When subclassing (combining with `Runnable`), list them in the following order, left to right:

- structure mixins (e.g. *SubsamplesIntaking* and *SubproblemSampler*)
- dimensionality mixins (e.g. *MultiInputStates* and *MISO*)
- validation toggles (e.g. *InputValidated* and *NotValidated*)
- StateTraits base class (not required if any of the above is used)
- Runnable base class

For example:

class MyRunnable(hybrid.traits.SubsamplesIntaking, hybrid.traits.MISO, hybid.Runnable): pass

class EmbeddingIntaking

class EmbeddingProducing

class InputNotValidated

class InputValidated

class MIMO
Multiple Inputs, Multiple Outputs.

class MISO
Multiple Inputs, Single Output.

class MultiInputStates

class MultiOutputStates

class NotValidated
Input state(s) and output state(s) are not validated.

class OutputNotValidated

class OutputValidated

class ProblemDecomposer

class ProblemIntaking

class ProblemProducing

class ProblemSampler

class SIMO
Single Input, Multiple Outputs.

class SISO
Single Input, Single Output.

class SamplesIntaking

class SamplesProcessor

class SamplesProducing

class SingleInputState

class SingleOutputState

class StateTraits
Set of traits imposed on State. By default, **not validated**.

validate_state_trait (*state*, *trait*, *io*)
Validate single input/output (*io*) *state trait*.

class SubproblemIntaking

class SubproblemProducing

class SubproblemSampler

class SubsamplesComposer

class SubsamplesIntaking

class SubsamplesProcessor

class SubsamplesProducing

class Validated
Validated input state(s) and output state(s).

2.2.9 Exceptions

exception EndOfStream
Signals end of stream for streaming runnables.

exception InvalidStateError
General state error.

exception RunnableError (*message*, *state*)
Generic Runnable exception error that includes the error context, in particular, the *State* that caused the runnable component to fail.

exception StateDimensionalityError
Single state expected instead of a state sequence, or vice versa.

exception StateTraitMissingError
State missing a trait.

2.2.10 Reference Workflows

The code includes implementations of some reference workflows you can incorporate as provided into your application and also use to jumpstart development of custom workflows.

Kerberos

Kerberos hybrid sampler runs 3 sampling branches in parallel. In each iteration, best results from tabu search and simulated annealing are combined with best results from QPU sampling a subproblem.

Kerberos(*max_iter=100*, *max_time=None*, *convergence=3*, *energy_threshold=None*, *sa_reads=1*, *sa_sweeps=10000*, *tabu_timeout=500*, *qpu_reads=100*, *qpu_sampler=None*, *qpu_params=None*, *max_subproblem_size=50*)

An opinionated hybrid asynchronous decomposition sampler for problems of arbitrary structure and size. Runs Tabu search, Simulated annealing and QPU subproblem sampling (for high energy impact problem variables) in parallel and returns the best samples.

Kerberos workflow is used by *KerberosSampler*.

Termination Criteria Args:

max_iter (int): Number of iterations in the hybrid algorithm.

max_time (float/None, optional, default=None): Wall clock runtime termination criterion. Unlimited by default.

convergence (int): Number of iterations with no improvement that terminates sampling.

energy_threshold (float, optional): Terminate when this energy threshold is surpassed. Check is performed at the end of each iteration.

Simulated Annealing Parameters:

sa_reads (int): Number of reads in the simulated annealing branch.

sa_sweeps (int): Number of sweeps in the simulated annealing branch.

Tabu Search Parameters:

tabu_timeout (int): Timeout for non-interruptable operation of tabu search (time in milliseconds).

QPU Sampling Parameters:

qpu_reads (int): Number of reads in the QPU branch.

qpu_sampler (dimod.Sampler, optional, default=DWaveSampler()): Quantum sampler such as a D-Wave system.

qpu_params (dict): Dictionary of keyword arguments with values that will be used on every call of the QPU sampler.

max_subproblem_size (int): Maximum size of the subproblem selected in the QPU branch.

Returns Workflow (*Runnable* instance).

class KerberosSampler

An opinionated dimod-compatible hybrid asynchronous decomposition sampler for problems of arbitrary structure and size.

Examples

This example solves a two-variable Ising model.

```
>>> import dimod
>>> import hybrid
>>> response = hybrid.KerberosSampler().sample_ising(
...                     {'a': -0.5, 'b': 1.0}, {('a', 'b'): -1})     # doctest:␣
↪+SKIP
```

(continues on next page)

(continued from previous page)

```
>>> response.data_vectors['energy']      # doctest: +SKIP
array([-1.5])
```

sample (*bqm*, *init_sample=None*, *num_reads=1*, ***kwargs*)

Run Tabu search, Simulated annealing and QPU subproblem sampling (for high energy impact problem variables) in parallel and return the best samples.

Sampling Args:

bqm (BinaryQuadraticModel): Binary quadratic model to be sampled from.

init_sample (SampleSet, callable, None): Initial sample set (or sample generator) used for each "read". Use a random sample for each read by default.

num_reads (int): Number of reads. Each sample is the result of a single run of the hybrid algorithm.

Termination Criteria Args:

max_iter (int): Number of iterations in the hybrid algorithm.

max_time (float/None, optional, default=None): Wall clock runtime termination criterion. Unlimited by default.

convergence (int): Number of iterations with no improvement that terminates sampling.

energy_threshold (float, optional): Terminate when this energy threshold is surpassed. Check is performed at the end of each iteration.

Simulated Annealing Parameters:

sa_reads (int): Number of reads in the simulated annealing branch.

sa_sweeps (int): Number of sweeps in the simulated annealing branch.

Tabu Search Parameters:

tabu_timeout (int): Timeout for non-interruptable operation of tabu search (time in milliseconds).

QPU Sampling Parameters:

qpu_reads (int): Number of reads in the QPU branch.

qpu_sampler (dimod.Sampler, optional, default=DWaveSampler()): Quantum sampler such as a D-Wave system.

qpu_params (dict): Dictionary of keyword arguments with values that will be used on every call of the QPU sampler.

max_subproblem_size (int): Maximum size of the subproblem selected in the QPU branch.

Returns A *dimod* `SampleSet` object.

Return type `SampleSet`

Parallel Tempering

Parallel tempering support and a reference workflow implementation.

class FixedTemperatureSampler(*beta=None*, *num_sweeps=10000*, *num_reads=None*, *aggregate=False*, *seed=None*, ***runopts*)

Parallel tempering propagate/update step.

The temperature (*beta*) can be specified upon object construction, and/or given externally (dynamically) in the input state.

On each call, run fixed temperature (~'1/beta') simulated annealing for *num_sweeps* (seeded by input sample(s)), effectively producing a new state by sampling from a Boltzmann distribution at the given temperature.

Parameters

- **beta** (*float, optional*) – Inverse of constant sampling temperature. If not supplied on construction, it must be present in the input state.
- **num_sweeps** (*int, optional, default=10k*) – Number of fixed temperature sampling sweeps.
- **num_reads** (*int, optional, default=len(state.samples)*) – Number of samples produced. If undefined, inferred from the size of the input sample set.
- **aggregate** (*bool, optional, default=False*) – Aggregate samples (duplicity stored in `num_occurrences`).
- **seed** (*int, optional, default=None*) – Pseudo-random number generator seed.

next(*state*, ***runopts*)

Execute one blocking iteration of an instantiated `Runnable` with a valid state as input.

Parameters **state** (`State`) – Computation state passed between connected components.

Returns The new state.

Return type `State`

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

class SwapReplicaPairRandom(*betas=None*, *seed=None*, ***runopts*)

Parallel tempering swap replicas step.

On each call, choose a random input state (replica), and probabilistically accept a swap with the adjacent state (replica). If swap is accepted, **only samples** contained in the selected states are exchanged.

Betas can be supplied in constructor, or otherwise they have to present in the input states.

Parameters

- **betas** (*list(float), optional*) – List of betas (inverse temperature), one for each input state. If not supplied, betas have to be present in the input states.
- **seed** (*int, default=None*) – Pseudo-random number generator seed.

next(*states*, ***runopts*)

Execute one blocking iteration of an instantiated `Runnable` with a valid state as input.

Parameters **state** (`State`) – Computation state passed between connected components.

Returns The new state.

Return type `State`

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

swap_pair (*betas*, *states*, *i*, *j*)
One pair of states' (i, j) samples probabilistic swap.

class SwapReplicasDownsweep (*betas=None*, ***runopts*)
Parallel tempering swap replicas step.

On each call, sweep down and probabilistically swap all adjacent pairs of replicas (input states).

Betas can be supplied in constructor, or otherwise they have to present in the input states.

Parameters **betas** (*list(float), optional*) – List of betas (inverse temperature), one for each input state. If not supplied, betas have to be present in the input states.

next (*states*, ***runopts*)
Execute one blocking iteration of an instantiated `Runnable` with a valid state as input.

Parameters **state** (`State`) – Computation state passed between connected components.

Returns The new state.

Return type `State`

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

ParallelTempering (*num_sweeps=10000*, *num_replicas=10*, *max_iter=None*, *max_time=None*, *convergence=3*)
Parallel tempering workflow generator.

Parameters

- **num_sweeps** (*int, optional*) – Number of sweeps in the fixed temperature sampling.
- **num_replicas** (*int, optional*) – Number of replicas (parallel states / workflow branches).
- **max_iter** (*int/None, optional*) – Maximum number of iterations of the update/swaps loop.
- **max_time** (*int/None, optional*) – Maximum wall clock runtime (in seconds) allowed in the update/swaps loop.
- **convergence** (*int/None, optional*) – Number of times best energy of the coldest replica has to repeat before we terminate.

Returns Workflow (*Runnable* instance).

HybridizedParallelTempering (*num_sweeps=10000*, *num_replicas=10*, *max_iter=None*, *max_time=None*, *convergence=3*)
Parallel tempering workflow generator.

Parameters

- **num_sweeps** (*int, optional*) – Number of sweeps in the fixed temperature sampling.
- **num_replicas** (*int, optional*) – Number of replicas (parallel states / workflow branches).
- **max_iter** (*int/None, optional*) – Maximum number of iterations of the update/swaps loop.
- **max_time** (*int/None, optional*) – Maximum wall clock runtime (in seconds) allowed in the update/swaps loop.
- **convergence** (*int/None, optional*) – Number of times best energy of the coldest replica has to repeat before we terminate.

Returns Workflow (*Runnable* instance).

Population Annealing

Population annealing support and a reference workflow implementation.

class EnergyWeightedResampler(*beta=None*, *seed=None*, ***runopts*)

Sample from the input sample set according to a distribution defined with sample energies (with replacement):

p ~ exp(-sample.energy / temperature) ~ exp(-beta * sample.energy)

Parameters

- **beta** (*float*) – Inverse of sampling temperature. Can be defined on sampler construction, on run method invocation, or in the input state's `beta` variable.
- **seed** (*int, default=None*) – Pseudo-random number generator seed.

Returns Input state with new samples. The lower the energy of an input sample, the higher will be its relative frequency in the output sample set.

next(*state*, ***runopts*)

Execute one blocking iteration of an instantiated `Runnable` with a valid state as input.

Parameters **state** (`State`) – Computation state passed between connected components.

Returns The new state.

Return type `State`

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

class ProgressBetaAlongSchedule(*beta_schedule=None*, ***runopts*)

Sets `beta` state variable to a schedule given on construction or in state at first run.

Parameters **beta_schedule** (*iterable(float)*) – The beta schedule. State's `beta` is iterated according to the beta schedule.

Raises *EndOfStream* when beta schedule is depleted.

init (*state*, ***runopts*)
Run prior to the first next/run, with the first state received.

Default to NOP.

next (*state*, ***runopts*)
Execute one blocking iteration of an instantiated `Runnable` with a valid state as input.

Parameters **state** (`State`) – Computation state passed between connected components.

Returns The new state.

Return type `State`

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

class CalculateAnnealingBetaSchedule (*length=2*, *interpolation='geometric'*, ***runopts*)
Calculate a best-guess beta schedule estimate for annealing methods, based on magnitudes of biases of the input problem, and the requested method of interpolation.

Parameters

- **length** (*int*) – Length of the produced beta schedule.
- **interpolation** (*str, optional, default='geometric'*) – Interpolation used between the hot and the cold beta. Supported values are:
  - linear
  - geometric

See: `neal.default_beta_range()`.

next (*state*, ***runopts*)
Execute one blocking iteration of an instantiated `Runnable` with a valid state as input.

Parameters **state** (`State`) – Computation state passed between connected components.

Returns The new state.

Return type `State`

Examples

This code snippet runs one iteration of a sampler to produce a new state:

```
new_state = sampler.next(core.State.from_sample({'x': 0, 'y': 0}, bqm))
```

PopulationAnnealing (*num_reads=20*, *num_iter=20*, *num_sweeps=1000*)
Population annealing workflow generator.

Parameters

- **num_reads** (*int*) – Size of the population of samples.
- **num_iter** (*int*) – Number of temperatures over which we iterate fixed-temperature sampling / resampling.

- **num_sweeps** (*int*) – Number of sweeps in the fixed temperature sampling step.

Returns Workflow (*Runnable* instance).

HybridizedPopulationAnnealing(*num_reads=20*, *num_iter=20*, *num_sweeps=1000*)
Workflow generator for population annealing initialized with QPU samples.

Parameters

- **num_reads** (*int*) – Size of the population of samples.
- **num_iter** (*int*) – Number of temperatures over which we iterate fixed-temperature sampling / resampling.
- **num_sweeps** (*int*) – Number of sweeps in the fixed temperature sampling step.

Returns Workflow (*Runnable* instance).

qbsolv

QBSolv inspired simple workflows.

SimplifiedQbsolv(*max_iter=10*, *max_time=None*, *convergence=3*, *energy_threshold=None*, *max_subproblem_size=30*)
Races a Tabu solver and a QPU-based sampler of flip-energy-impact induced subproblems.

For arguments description see: *Kerberos*.

2.3 Installation

Install from a package on PyPI:

```
pip install dwave-hybrid
```

or from source:

```
git clone https://github.com/dwavesystems/dwave-hybrid.git
cd dwave-hybrid
pip install -r requirements.txt
python setup.py install
```

2.4 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.

2.5 Indices and tables

- genindex
- modindex
- search
- Glossary

Bibliography

[KS] Kadioglu S., Sellmann M. (2009) Dialectic Search. In: Gent I.P. (eds) Principles and Practice of Constraint Programming - CP 2009. CP 2009. Lecture Notes in Computer Science, vol 5732. Springer, Berlin, Heidelberg

[BHT] Boros, E., P.L. Hammer, G. Tavares. Preprocessing of Unconstraint Quadratic Binary Optimization. Rutcor Research Report 10-2006, April, 2006.

[BH] Boros, E., P.L. Hammer. Pseudo-Boolean optimization. Discrete Applied Mathematics 123, (2002), pp. 155-225

Python Module Index

Index

K

L

M

N

O

P

Q

R

S

T

U

V

W