docs Documentation

Release 0.0.1

D-Wave Systems Inc

Nov 20, 2019

Contents

Ocean software is a suite of tools D-Wave Systems provides on the D-Wave GitHub repository for solving hard problems with quantum computers.

CHAPTER 1

Getting Started

See how to install and use Ocean tools.

CHAPTER 2

Tools

Access a particular tool's documentation and repository.

CHAPTER 3

Glossary

Understand the terminology.

3.1 Getting Started

New to Ocean? The following sections describe how to install Ocean tools, what they are and how they fit together, and give examples of using them to solve hard problems on a D-Wave quantum computer.

3.1.1 Initial Set Up

Install the tools and configure for running on a D-Wave system (QPU) or locally (CPU/GPU).

Installing Ocean Tools

Ocean software is supported on the following operating systems:

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

Ocean software requires a *Python environment*. Supported Python versions are:

- 2.7.x
- 3.5 and higher

> **Attention:** D-Wave's Ocean software will stop supporting Python 2 at the end of 2019.
>
> For information on why many in the Python development community are requiring Python 3, see the Python 3 statement.

This section explains how to *install Ocean software*, either the entire suite of tools or particular tools from the D-Wave GitHub repositories listed under *Tools*.

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

Python Virtual Environment

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

1. Download Python describes how to install Python on your local machine for supported operating system.

   For Unix-based systems, which often have Python pre-installed, installation might be as simple as:

   ```
   sudo apt-get install python<version>
   ```

   Attention: For Windows systems, note that only **64-bit** Python is supported.

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

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

   ```
   sudo pip install virtualenv
   ```

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

   ```
   virtualenv ocean
   source ocean/bin/activate
   ```

   (On Windows operating system, activating a virtual environment might be done with the `Scripts\activate` command instead.)

Your machine is now ready to install Ocean software.

Install Ocean Software

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

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

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

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

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

Note: To install a particular tool within the SDK only, follow the link to the GitHub repository for the tool, as listed under *Tools*, and follow the installation instructions on the README file.

Configure a Solver

Most Ocean tools solve problems on a *solver*, which is a compute resources such as a D-Wave system or CPU, and might require that you configure a default solver.

- *Using a D-Wave System* describes how to configure your system to solve problems on a D-Wave system.
- *Using a Classical Solver* describes how to configure your system to solve problems classically on your local CPU/GPU.

Using a D-Wave System

To use a D-Wave system as your *solver* (the compute resource for solving problems), you access it through the D-Wave Solver API (SAPI).

Interacting with SAPI

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

1. API endpoint URL

   A URL to the remote D-Wave system. By default, `https://cloud.dwavesys.com/sapi` is used to connect to resources provided by D-Wave's Leap Quantum Application Environment.

2. API Token

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

3. Solver

   A D-Wave resource to be used to solve your submitted problems.

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

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

Configuring a D-Wave System as a Solver

The simplest way to configure a solver is to use the dwave-cloud-client interactive CLI, which is installed as part of the dwave-ocean-sdk (or D-Wave Cloud Client tool installation).

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

```
$ dwave config create
Configuration file not found; the default location is: /home/jane/.config/dwave/dwave.
↪conf
Confirm configuration file path [/home/jane/.config/dwave/dwave.conf]:
Profile (create new) [prod]:
API endpoint URL [skip]:
Authentication token [skip]: ABC-1234567890abcdef1234567890abcdef
```

```
Default client class (qpu or sw) [qpu]:
Default solver [skip]:
Configuration saved.
```

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

Alternatively, you can create and edit a D-Wave Cloud Client configuration file manually or set the solver, API token, and URL directly in your code or as local environment variables. For more information, see the examples in this document or D-Wave Cloud Client.

Verifying Your Solver Configuration

You can test that your solver is configured correctly and that you have access to a D-Wave solver with the same dwave-cloud-client interactive CLI installed as part of the SDK or D-Wave Cloud Client tool installation.

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

```
$ dwave ping
Using endpoint: https://my.dwavesys.url/
Using solver: My_DWAVE_2000Q

Wall clock time:
 * Solver definition fetch: 2007.239 ms
 * Problem submit and results fetch: 1033.931 ms
 * Total: 3041.171 ms

QPU timing:
 * total_real_time = 10493 us
 * anneal_time_per_run = 20 us
 * post_processing_overhead_time = 128 us
 * qpu_anneal_time_per_sample = 20 us
 # Snipped for brevity
```

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

```
$ dwave sample --random-problem
Using endpoint: https://my.dwavesys.url/
Using solver: My_DWAVE_2000Q
Using qubit biases: {0: -1.0345257941434953, 1: -0.5795618633919246, 2: 0.
↪9721956399428491, 3: 1....
Using qubit couplings: {(1634, 1638): 0.721736584181423, (587, 590): 0.
↪9611623181258304, (642, 64...
Number of samples: 1
Samples: [[1, 1, -1, -1, -1, -1, 1, -1, -1, 1, -1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -
↪1, 1, 1, -1,...
Occurrences: [1]
Energies: [-2882.197791239335]
```

Querying Available Solvers

The dwave-cloud-client interactive CLI can also show you the available solvers, their parameters, and properties.

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

```
$ dwave solvers
Solver: My_DWAVE_2000Q
   Parameters:
      anneal_offsets: A list of anneal offsets for each working qubit (NaN if u...
      anneal_schedule: A piecewise linear annealing schedule specified by a list...
      annealing_time: A positive integer that sets the duration (in microsecond...

      <Output snipped for brevity>

   Properties:
      anneal_offset_ranges: [[-0.18627387668142237, 0.09542224439071689], [-0.1836548.
↪..
      anneal_offset_step: 0.00426679499507194
      anneal_offset_step_phi0: 0.0002716837027763096
      annealing_time_range: [1, 150000]
      chip_id: W7-1_C16_4724854-02-G4_C5R9-device-cal-data-18-05-27-14:27
      couplers: [[0, 4], [1, 4], [2, 4], [3, 4], [0, 5], [1, 5], [2, 5], ...

      <Output snipped for brevity>
```

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

```
>>> from dwave.cloud import Client
>>> client = Client.from_config(token='ABC-1234567891234567891234567789')
>>> client.get_solvers()
[Solver(id='2000Q_ONLINE_SOLVER1'),
 Solver(id='2000Q_ONLINE_SOLVER2')]
```

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

```
>>> from dwave.system.samplers import DWaveSampler
>>> sampler = DWaveSampler()
>>> sampler.parameters
{u'anneal_offsets': ['parameters'],
u'anneal_schedule': ['parameters'],
u'annealing_time': ['parameters'],
u'answer_mode': ['parameters'],
u'auto_scale': ['parameters'],
# Snipped above response for brevity
```

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

Submitting Problems to a D-Wave System

Once you have configured a D-Wave Cloud Client configuration file your default solver configuration is used when you submit a problem without explicitly overriding it. For example, the following code uses a dwave-system structured sampler, `EmbeddingComposite(DWaveSampler())`, as the sampler, which uses a D-Wave system for the compute resource. Because no parameters (e.g., SAPI endpoint URL) are set explicitly, the line `sampler = EmbeddingComposite(DWaveSampler())` uses your default solver.

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> response = sampler.sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b'): -1})
>>> response.data_vectors['energy']
array([-1.5])
```

The examples under *Getting Started* demonstrate solving problems on the D-Wave system, starting from very simple and gradually increasing the complexity.

Using a Classical Solver

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

Examples

Among several samplers provided in the dimod tool for testing your code locally, is the `ExactSolver()` that calculates the energy of all possible samples for a given problem. This example solves a two-variable Ising model classically on your local machine.

```
>>> import dimod
>>> solver = dimod.ExactSolver()
>>> response = solver.sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b'): -1})
>>> response.data_vectors['energy']
array([-1.5, -0.5, -0.5,  2.5])
```

This example solves the previous problem using the dwave_neal simulated annealing sampler. The two samples requested and generated by this classical solver on your local machine vary by execution.

```
>>> import neal
>>> solver = neal.SimulatedAnnealingSampler()
>>> response = solver.sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b'): -1}, num_
↪reads=2)
>>> response.data_vectors['energy']
array([-1.5, -0.5])
```

3.1.2 Overview of Ocean Software

Learn how problems are formulated for solution on D-Wave systems using Ocean tools.

Solving Problems on a D-Wave System

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

How a D-Wave System Solves Problems

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

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

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

$$\sum_{i}^{N} q_i x_i + \sum_{i<j}^{N} q_{i,j} x_i x_j$$

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

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

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

1. *Formulate the Problem as a BQM*.
2. *Solve the BQM with a Sampler*.
3. *Improve the Solutions*, if needed, using advanced features.

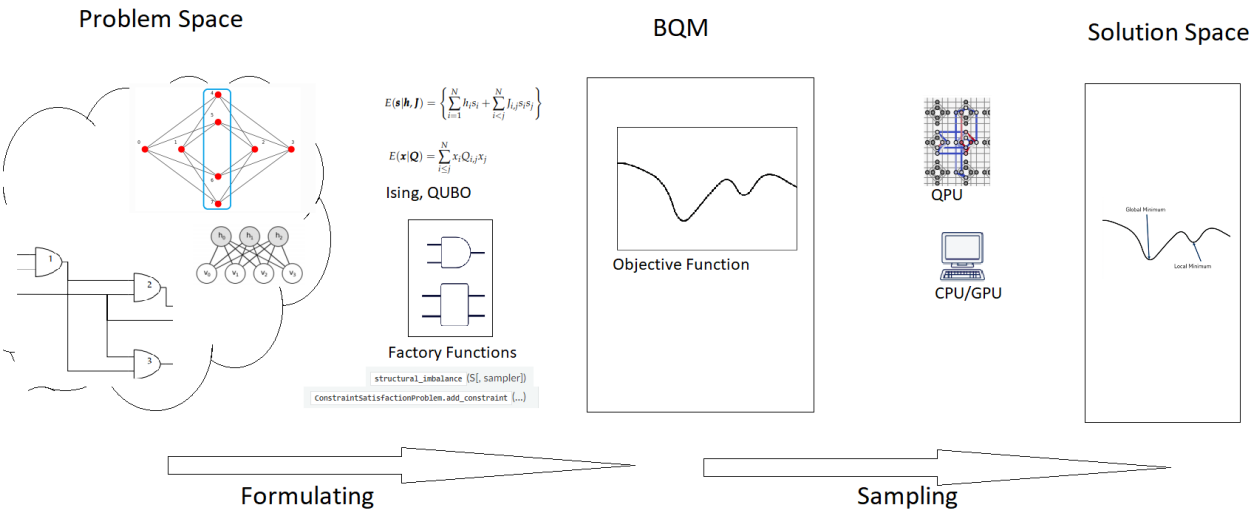

Fig. 3.1: Solution steps: (1) a problem known in "problem space" (a circuit of Boolean gates, a graph, a network, etc) is formulated as a BQM, mathematically or using Ocean functionality and (2) the BQM is sampled for solutions.

Formulate the Problem as a BQM

There are different ways of mapping between a problem—chains of amino acids forming 3D structures of folded proteins, traffic in the streets of Beijing, circuits of binary gates—and a BQM to be solved (by sampling) with a

[1] The "native" forms of BQM programmed into a D-Wave system are the *Ising* model traditionally used in statistical mechanics and its computer-science equivalent, shown here, the *QUBO*.

D-Wave system or locally on your CPU/GPU.

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

The following are two example formulations.

1. The *Boolean NOT Gate* example, takes a NOT gate represented symbolically as $x_2 \Leftrightarrow \neg x_1$ and formulates it mathematically as the following BQM:

   $$-x_1 - x_2 + 2x_1x_2$$

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

   Table 3.1: Boolean NOT Operation Formulated as a BQM.

x_1	x_2	**Valid?**	**BQM Value**
0	1	Yes	0
1	0	Yes	0
0	0	No	1
1	1	No	1

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

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

Once you have a BQM that represents your problem, you sample it for solutions.

Solve the BQM with a Sampler

To solve your problem, now represented as a binary quadratic model, you submit it to a classical or quantum sampler. If you use a classical solver running locally on your CPU, a single sample might provide the optimal solution. When you use a probabilistic sampler like the D-Wave system, you typically program for multiple reads.

Note: To configure access to a D-Wave system, see the *Using a D-Wave System* section.

For example, the BQM of the AND gate created above may look like this:

```
>>> bqm
BinaryQuadraticModel({'x1': 0.0, 'x2': 0.0, 'y1': 6.0}, {('x2', 'x1'): 2.0, ('y1', 'x1
↪'): -4.0, ('y1', 'x2'): -4.0}, -1.5, Vartype.BINARY)
```

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

Ocean's dimod tool provides a reference solver that calculates the values of a BQM (its "energy") for all possible assignments of variables. Such a sampler can solve a small three-variable problem like the AND gate created above.

```
>>> from dimod.reference.samplers import ExactSolver
>>> sampler = ExactSolver()
>>> response = sampler.sample(bqm)
>>> for datum in response.data(['sample', 'energy']):
...     print(datum.sample, datum.energy)
...
{'x1': 0, 'x2': 0, 'y1': 0} -1.5
{'x1': 1, 'x2': 0, 'y1': 0} -1.5
{'x1': 0, 'x2': 1, 'y1': 0} -1.5
{'x1': 1, 'x2': 1, 'y1': 1} -1.5
{'x1': 1, 'x2': 1, 'y1': 0} 0.5
{'x1': 0, 'x2': 1, 'y1': 1} 0.5
{'x1': 1, 'x2': 0, 'y1': 1} 0.5
{'x1': 0, 'x2': 0, 'y1': 1} 4.5
```

Note that the first four samples are the valid states of the AND gate and have lower values than the second four, which represent invalid states.

Ocean's dwave-system tool enables you to use a D-Wave system as a sampler. In addition to *DWaveSampler()*, the tool provides a *EmbeddingComposite()* composite that maps unstructured problems to the graph structure of the selected sampler, a process known as *minor-embedding*. In our case, the problem is defined on alphanumeric variables $x1, x2, y1$, that must be mapped to the QPU's numerically indexed qubits.

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

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> response = sampler.sample(bqm, num_reads=1000)
>>> for datum in response.data(['sample', 'energy', 'num_occurrences']):
...     print(datum.sample, datum.energy, "Occurrences: ", datum.num_occurrences)
...
{'x1': 0, 'x2': 1, 'y1': 0} -1.5 Occurrences:  92
{'x1': 1, 'x2': 1, 'y1': 1} -1.5 Occurrences:  256
{'x1': 0, 'x2': 0, 'y1': 0} -1.5 Occurrences:  264
{'x1': 1, 'x2': 0, 'y1': 0} -1.5 Occurrences:  173
{'x1': 1, 'x2': 0, 'y1': 1} 0.5 Occurrences:  215
```

Note that the first four samples are the valid states of the AND gate and have lower values than invalid state $x1 = 1, x2 = 0, y1 = 1$.

Improve the Solutions

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

The mapping from problem variables to qubits, *minor-embedding*, can significantly affect performance. Ocean tools perform this mapping heuristically so simply rerunning a problem might improve results. Advanced users may customize the mapping by directly using the minorminer tool or setting a minor-embedding themselves (or some combination).

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

You can see the parameters and properties a sampler supports. For example, Ocean's dwave-system lets you use the D-Wave's *virtual graphs* feature to simplify minor-embedding. The following example maps a problem's variables x,

y to qubits 1, 5 and variable z to two qubits 0 and 4, and checks some features supported on the D-Wave system used as a sampler.

> **Attention:** D-Wave's *virtual graphs* feature can require many seconds of D-Wave system time to calibrate qubits to compensate for the effects of biases. If your account has limited D-Wave system access, consider using *FixedEmbeddingComposite()* instead.

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import VirtualGraphComposite
>>> DWaveSampler().properties['extended_j_range']
[-2.0, 1.0]
>>> embedding = {'x': {1}, 'y': {5}, 'z': {0, 4}}
>>> sampler = VirtualGraphComposite(DWaveSampler(), embedding)
>>> sampler.parameters
{u'anneal_offsets': ['parameters'],
 u'anneal_schedule': ['parameters'],
 u'annealing_time': ['parameters'],
 u'answer_mode': ['parameters'],
 'apply_flux_bias_offsets': [],
 u'auto_scale': ['parameters'],
>>>  # Snipped above response for brevity
```

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

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

Ocean Software Stack

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

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

Abstraction Layers

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

- Compute Resources

  The hardware on which the problem is solved. This might be a D-Wave quantum processor but it can also be the CPU of your laptop computer.

- Samplers

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

- Sampler API

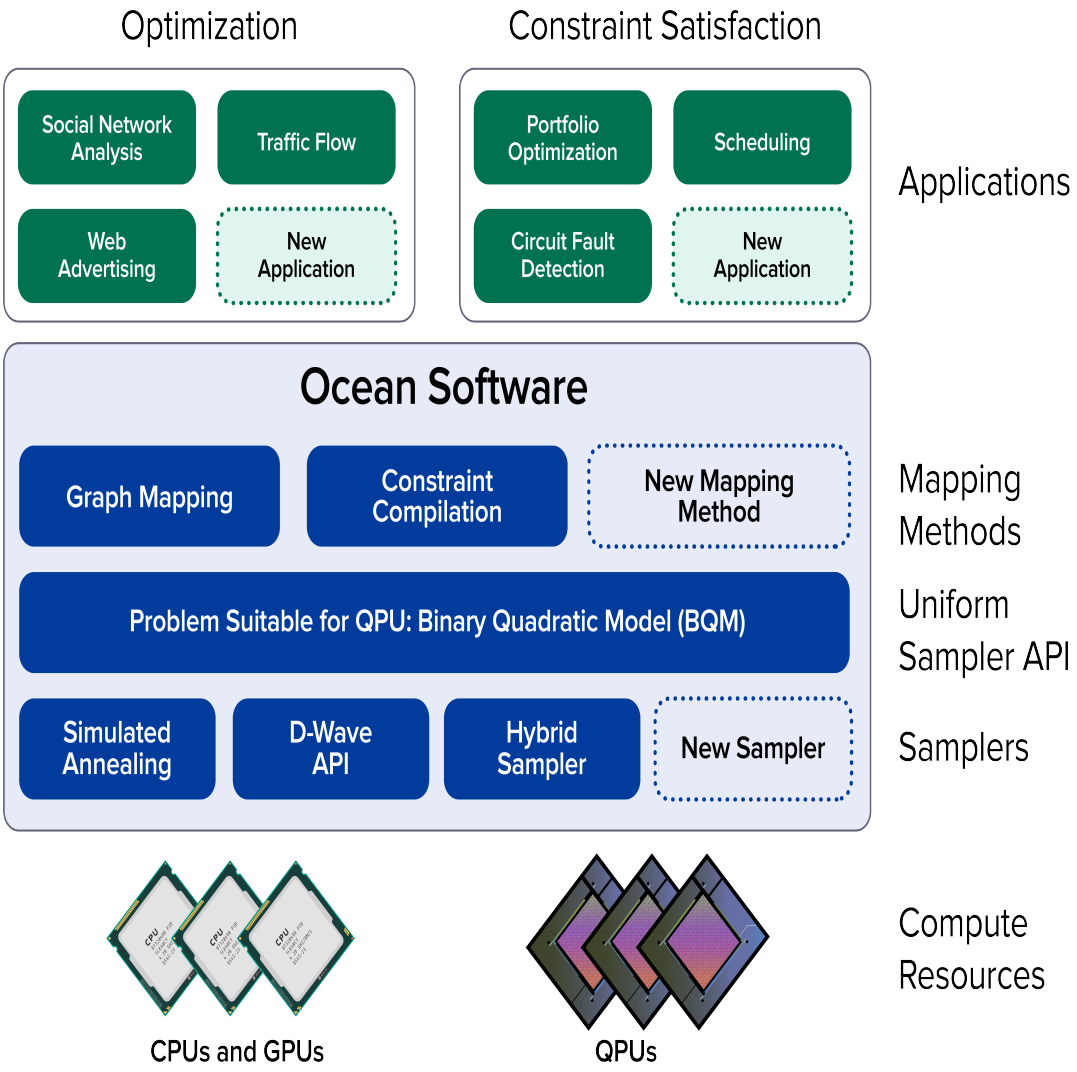

Fig. 3.2: Ocean Software Stack

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

- Methods

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

- Application

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

Problem-to-Solution Tool Chain

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

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

Bottom-Up Approach

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

1. **Compute resource**

   You will likely use some combination of both local classical resources and a D-Wave system in your work with Ocean software. When would you use which?

   - CPU/GPU: for offline testing, small problems that can be solved exactly or heuristically in a reasonable amount of time.
   - QPU: hard problems or for learning how to use quantum resources to solve such problems.
   - Hybrid of both QPU and CPU/GPU: large, complex problems that need to run classically but may benefit from having some parts allocated to a quantum computer for solution.

2. **Sampler**

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

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

Table 3.2: Ocean Samplers

Computation	Tool & Sampler	Usage	Notes
Classical	dimod `ExactSampler()`	Find all states for small (<20 variables) problems.	For code-development testing.
Classical	dimod `RandomSampler()`	Random sampler for testing.	For code-development testing.
Classical	dimod `SimulatedAnnealingSampler()`	Simulated annealing sampler for testing.	For code-development testing.
Classical	dwave-neal `SimulatedAnnealingSampler()`	Simulated annealing sampler.	
Quantum	dwave-system `DWaveSampler()`	Quick incorporation of the D-Wave system as a sampler.	Typically part of a composite that handles *minor-embedding*.
Quantum	dwave-cloud-client `Solver()`	D-Wave system as a sampler.[1]	For low-level control of problem submission.
	dimod custom	Write a custom sampler for special cases.	See examples in dimod.

3. **Pre- and Post-Processing**

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

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

Table 3.3: Ocean Composites

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

In addition to composites that provide pre- and post-processing, Ocean also provides stand-alone tools to handle complex or large problems. For example:

[1] This sampler is for low-level work on communicating with SAPI and is not a dimod sampler.

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

4. **Map to a Supported Format**

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

   Ocean provides tools for formulating the BQM:

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

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

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

5. **Formulate**

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

   The selection function must express the problem's constraints:

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

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

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

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

   Ocean demo applications, which formulate known problems, include:

   - Structural Imbalance.
   - Circuit-Fault Diagnosis.

Top-Down Approach

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

Table 3.4: Ocean Software

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

3.1.3 Examples

See how Ocean tools are used with these end-to-end examples.

Beginner-Level Examples

Large Map Coloring

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

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

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

Example Requirements

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

- The requisite information for problem submission through SAPI, as described in *Using a D-Wave System*
- Ocean tools dwave-hybrid and dwave_networkx.

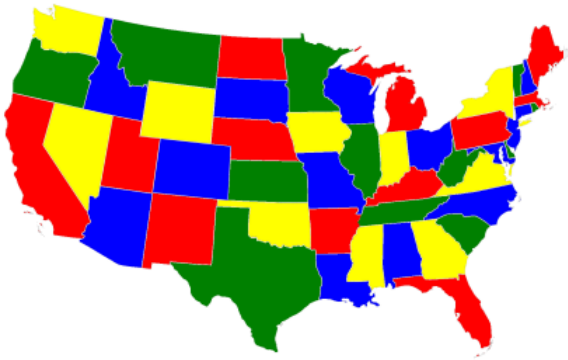

Fig. 3.3: Coloring a map of the USA.

If you installed dwave-ocean-sdk and ran `dwave config create`, your installation should meet these requirements.

Solution Steps

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

Formulate the Problem

This example uses the NetworkX *read_adjlist* function to read a text file, *usa.adj*, containing the states of the USA and their adjacencies (states with a shared border) into a graph. The original map information was found here on write-only blog of Gregg Lind and looks like this:

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

# Snipped here for brevity
```

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

```
>>> import networkx as nx
>>> G = nx.read_adjlist('usa.adj', delimiter = ',')
```

Graph G now represents states as vertices and each state's neighbors as shared edges. Ocean's dwave_networkx can return a minimum vertex coloring for a graph, which assigns a color to the vertices of a graph in a way that no adjacent

vertices have the same color, using the minimum number of colors. Given a graph representing a map and a *sampler*, the *min_vertex_coloring* function tries to solve the map coloring problem.

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

```
>>> import dwave_networkx as dnx
>>> from hybrid.reference.kerberos import KerberosSampler
>>> coloring = dnx.min_vertex_coloring(G, sampler=KerberosSampler(), chromatic_ub=4,␣
↪max_iter=10, convergence=3)
>>> set(coloring.values())
{0, 1, 2, 3}
```

Note: The next code requires Matplotlib.

Plot the solution, if valid.

```
>>> import matplotlib.pyplot as plt
>>> node_colors = [coloring.get(node) for node in G.nodes()]
>>> if dnx.is_vertex_coloring(G, coloring):  # adjust the next line if using a␣
↪different map
...     nx.draw(G, pos=nx.shell_layout(G, nlist = [list(G.nodes)[x:x+10] for x in␣
↪range(0, 50, 10)] + [[list(G.nodes)[50]]]), with_labels=True, node_color=node_
↪colors, node_size=400, cmap=plt.cm.rainbow)
>>> plt.show()
```

The graphic below shows the result of one such run.

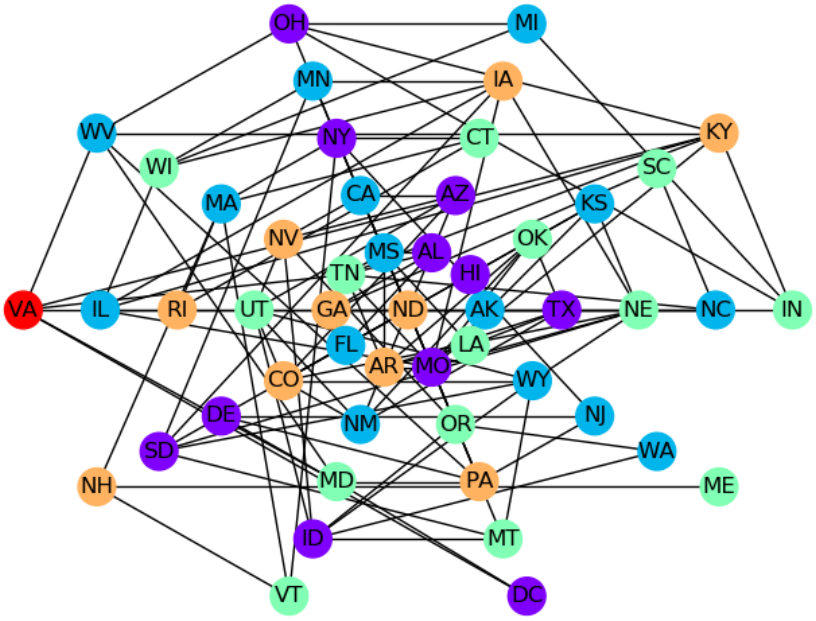

Fig. 3.4: One solution found for the USA map-coloring problem.

Vertex Cover

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

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

Example Requirements

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

- The requisite information for problem submission through SAPI, as described in *Using a D-Wave System*.
- Ocean tools dwave-system, dimod, and dwave_networkx.

If you installed dwave-ocean-sdk and ran `dwave config create`, your installation should meet these requirements.

Solution Steps

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

Formulate the Problem

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

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

```
>>> import networkx as nx
>>> s5 = nx.star_graph(4)
```

Solve the Problem by Sampling

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

Solving Classically on a CPU

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

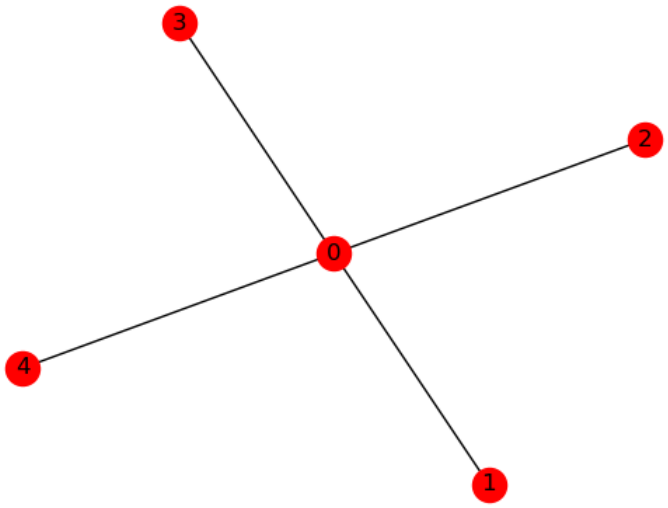

Fig. 3.5: A five-node star graph.

```
>>> from dimod.reference.samplers import ExactSolver
>>> sampler = ExactSolver()
```

The next code lines use Ocean's dwave_networkx to produce a BQM for our `s5` graph and solve it on our selected sampler. In other examples the BQM is explicitly created but the Ocean tool used here abstracts the BQM: given the problem graph it returns a solution to a BQM it creates internally.

```
>>> import dwave_networkx as dnx
>>> print(dnx.min_vertex_cover(s5, sampler))
[0]
```

Solving on a D-Wave System

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

Note: In the code below, replace sampler parameters in the third line. If you configured a default solver, as described in *Using a D-Wave System*, you should be able to set the sampler without parameters as `sampler = EmbeddingComposite(DWaveSampler())`. You can see this information by running `dwave config inspect` in your terminal.

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler(endpoint='https://URL_to_my_D-Wave_
↪system/', token='ABC-1234567890123456789012345678890', solver='My_D-Wave_Solver'))
>>> print(dnx.min_vertex_cover(s5, sampler))
[0]
```

Additional Problem Graphs

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

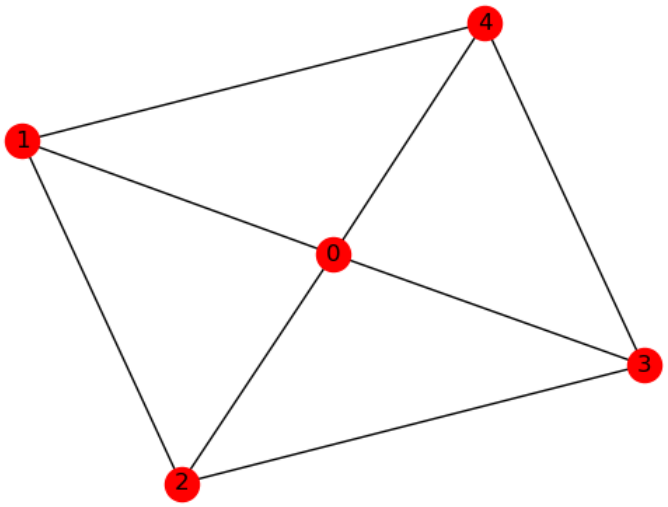

Fig. 3.6: A five-node wheel graph.

The code snippet below creates a new graph and solves on a D-Wave system.

```
>>> w5 = nx.wheel_graph(5)
>>> print(dnx.min_vertex_cover(w5, sampler))
[0, 1, 3]
```

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

```
>>> print(dnx.min_vertex_cover(w5, sampler))
[0, 2, 4]
```

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

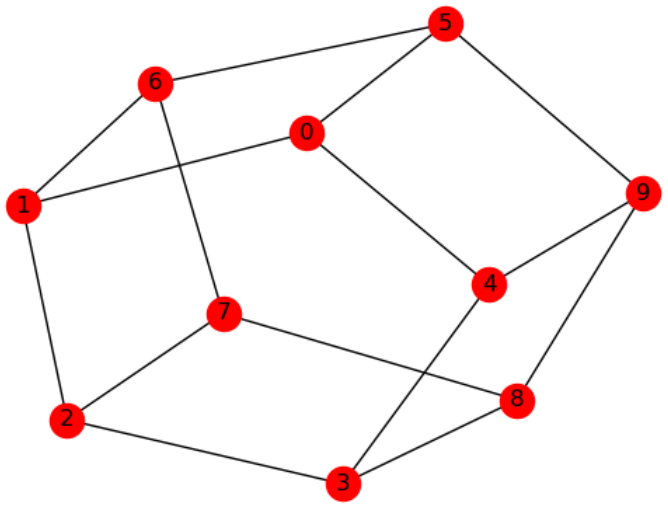

Fig. 3.7: A ten-node circular-ladder graph.

The code snippet below replaces the problem graph and submits twice to the D-Wave system for solution, producing two of the possible valid solutions.

```
>>> c5 = nx.circular_ladder_graph(5)
>>> print(dnx.min_vertex_cover(c5, sampler))
[0, 2, 3, 6, 8, 9]
>>> print(dnx.min_vertex_cover(c5, sampler))
[1, 3, 4, 5, 7, 9]
```

Summary

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

- Application: an example application might be placing limited numbers of traffic-monitoring equipment on routers in a telecommunication network. Such problems can be posed as graphs.
- Method: graph mapping. Many different real-world problems can be formulated as instances of classified graph problems. Some of these are hard and the best currently known algorithms for solution may not scale well. Quantum computing might provide better solutions. In this example, vertex cover is a hard problem that can be solved on D-Wave systems.
- Sampler API: the Ocean tool internally builds a BQM with lowest values ("ground states") that correspond to a minimum vertex cover and uses our selected sampler to solve it.
- Sampler: classical *ExactSolver()* and then *DWaveSampler()*.
- Compute resource: first a local CPU then a D-Wave system.

Constrained Scheduling

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

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

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

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

Example Requirements

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

- The requisite information for problem submission through SAPI, as described in *Using a D-Wave System*.
- Ocean tools dwave-binarycsp, dwave-system, and dimod.

If you installed dwave-ocean-sdk and ran `dwave config create`, your installation should meet these requirements.

Solution Steps

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

Formulate the Problem

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

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

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

Table 3.5: All Possible Meeting Options.

Time of Day	Venue	Duration	Participation	Valid?
Business hours	Office	Short	Mandatory	Yes
Business hours	Office	Short	Optional	No (violates 2)
Business hours	Office	Long	Mandatory	Yes
Business hours	Office	Long	Optional	No (violates 2)
Business hours	Teleconference	Short	Mandatory	No (violates 1)
Business hours	Teleconference	Short	Optional	No (violates 1, 2)
Business hours	Teleconference	Long	Mandatory	No (violates 1)
Business hours	Teleconference	Long	Optional	No (violates 1, 2)
Non-business hours	Office	Short	Mandatory	No (violates 3)
Non-business hours	Office	Short	Optional	No (violates 3)
Non-business hours	Office	Long	Mandatory	No (violates 3, 4)
Non-business hours	Office	Long	Optional	No (violates 3, 4)
Non-business hours	Teleconference	Short	Mandatory	Yes
Non-business hours	Teleconference	Short	Optional	Yes
Non-business hours	Teleconference	Long	Mandatory	No (violates 4)
Non-business hours	Teleconference	Long	Optional	No (violates 4)

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

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

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

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

This tool, dwavebinarycsp, can also convert the binary CSP to a BQM. The following code does so and displays the BQM's linear and quadratic coefficients, q_i and $q_{i,j}$ respectively in $\sum_i^N q_i x_i + \sum_{i<j}^N q_{i,j} x_i x_j$, which are the inputs for programming the quantum computer.

```
>>> bqm = dwavebinarycsp.stitch(csp)
>>> bqm.linear
{'length': -2.0, 'location': 2.0, 'mandatory': 0.0, 'time': 2.0}
>>> bqm.quadratic
{('location', 'length'): 2.0,
 ('mandatory', 'length'): 0.0,
 ('mandatory', 'location'): -2.0,
 ('time', 'length'): 0.0,
 ('time', 'location'): -4.0,
 ('time', 'mandatory'): 0.0}
```

Solve the Problem by Sampling

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

Solving Classically on a CPU

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

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

Valid solutions—assignments of variables that do not violate any constraint—should have the lowest value of the BQM, and *ExactSolver()* orders its assignments of variables by ascending order, so the first solution has the lowest value (lowest energy state). The code below sets variable `min_energy` to the BQM's lowest value, which is in the first record of the returned result.

```
>>> min_energy = next(solution.data(['energy']))[0]
>>> print(min_energy)
-2.0
```

The code below prints all those solutions (assignments of variables) for which the BQM has its minimum value.

```
>>> for sample, energy in solution.data(['sample', 'energy']):
...     if energy == min_energy:
...         time = 'business hours' if sample['time'] else 'evenings'
...         location = 'office' if sample['location'] else 'home'
...         length = 'short' if sample['length'] else 'long'
...         mandatory = 'mandatory' if sample['mandatory'] else 'optional'
```

```
...         print("During {} at {}, you can schedule a {} meeting that is {}".
↪format(time, location, length, mandatory))
...
During evenings at home, you can schedule a short meeting that is optional
During evenings at home, you can schedule a short meeting that is mandatory
During business hours at office, you can schedule a short meeting that is mandatory
During business hours at office, you can schedule a long meeting that is mandatory
```

Solving on a D-Wave System

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

Note: In the code below, replace sampler parameters in the third line. If you configured a default solver, as described in *Using a D-Wave System*, you should be able to set the sampler without parameters as `sampler = EmbeddingComposite(DWaveSampler())`. You can see this information by running `dwave config inspect` in your terminal.

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler(endpoint='https://URL_to_my_D-Wave_
↪system/', token='ABC-1234567890123456789012345678890', solver='My_D-Wave_Solver'))
```

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

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

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

```
>>> total = 0
... for sample, energy, occurrences in response.data(['sample', 'energy', 'num_
↪occurrences']):
...     total = total + occurrences
...     if energy == min_energy:
...         time = 'business hours' if sample['time'] else 'evenings'
...         location = 'office' if sample['location'] else 'home'
...         length = 'short' if sample['length'] else 'long'
...         mandatory = 'mandatory' if sample['mandatory'] else 'optional'
...         print("{}: During {} at {}, you can schedule a {} meeting that is {}".
↪format(occurrences, time, location, length, mandatory))
... print("Total occurrences: ", total)
...
1676: During business hours at office, you can schedule a long meeting that is␣
↪mandatory
1229: During business hours at office, you can schedule a short meeting that is␣
↪mandatory
1194: During evenings at home, you can schedule a short meeting that is optional
898: During evenings at home, you can schedule a short meeting that is mandatory
Total occurrences:  5000
```

Summary

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

- Application: scheduling under constraints. There exist many CSPs that are computationally hard problems; for example, the map-coloring problem is to color all regions of a map such that any two regions sharing a border have different colors. The job-shop scheduling problem is to schedule multiple jobs done on several machines with constraints on the machines' execution of tasks.
- Method: constraint compilation.
- Sampler API: the Ocean tool builds a BQM with lowest values ("ground states") that correspond to assignments of variables that satisfy all constraints.
- Sampler: classical *ExactSolver()* and then *DWaveSampler()*.
- Compute resource: first a local CPU then a D-Wave system.

Boolean NOT Gate

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

Example Requirements

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

- The requisite information for problem submission through SAPI, as described in *Using a D-Wave System*
- Ocean tools dwave-system and dimod.

If you installed dwave-ocean-sdk and ran `dwave config create`, your installation should meet these requirements.

Solution Steps

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

Formulate the NOT Gate as a BQM

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

$$\sum_{i}^{N} q_i x_i + \sum_{i<j}^{N} q_{i,j} x_i x_j,$$

[1] The "native" forms of BQM programmed into a D-Wave system are the *Ising* model traditionally used in statistical mechanics and its computer-science equivalent, shown here, the *QUBO*.

(in1) at (0, 0) x; (out1) at (2, 0) z ;
(1, 0) node[not port] (mynot1)
(0.1, 0) – (mynot1.in) (mynot1.out) – (1.9, 0);

Fig. 3.8: NOT gate

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

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

A NOT gate is shown in Figure 3.8.

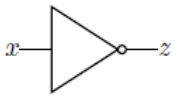

Fig. 3.9: A NOT gate.

Representing the Problem With a Penalty Function

This example demonstrates a mathematical formulation of the BQM. We can represent a NOT gate, $z \Leftrightarrow \neg x$, where x is the gate's input and z its output, using a *penalty function*:

$$2xz - x - z + 1.$$

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

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

Table 3.6: Boolean NOT Operation Represented by a Penalty Function.

x	z	Valid?	P
0	1	Yes	0
1	0	Yes	0
0	0	No	1
1	1	No	1

For example, the state $x, z = 0, 1$ of the first row represents valid assignments, and the value of P is

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

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

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

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

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

Formulating the Problem as a QUBO

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

$$2xz - x - z,$$

are easily reordered in standard *QUBO* formulation:

$$-x_1 - x_2 + 2x_1x_2$$

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

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

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

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

Solve the Problem by Sampling

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

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

Note: In the code below, replace sampler parameters in the third line. If you configured a default solver, as described in *Using a D-Wave System*, you should be able to set the sampler without parameters as `sampler = EmbeddingComposite(DWaveSampler())`. You can see this information by running `dwave config inspect` in your terminal.

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler(endpoint='https://URL_to_my_D-Wave_
↪system/', token='ABC-1234567890123456789012345678 90', solver='My_D-Wave_Solver'))
```

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

```
>>> Q = {('x', 'x'): -1, ('x', 'z'): 2, ('z', 'x'): 0, ('z', 'z'): -1}
>>> response = sampler.sample_qubo(Q, num_reads=5000)
>>> for datum in response.data(['sample', 'energy', 'num_occurrences']):
...     print(datum.sample, "Energy: ", datum.energy, "Occurrences: ", datum.num_
↪occurrences)
...
{'x': 0, 'z': 1} Energy:  -1.0 Occurrences:  2062
{'x': 1, 'z': 0} Energy:  -1.0 Occurrences:  2937
{'x': 1, 'z': 1} Energy:  0.0 Occurrences:  1
```

Almost all the returned samples represent valid value assignments for a NOT gate, and minimize (are low-energy states of) the BQM.

Summary

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

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

Boolean AND Gate

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

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

Example Requirements

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

- The requisite information for problem submission through SAPI, as described in *Using a D-Wave System*
- Ocean tools dwave-system.

If you installed dwave-ocean-sdk and ran `dwave config create`, your installation should meet these requirements.

Solution Steps

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

Formulate the AND Gate as a BQM

Ocean tools can automate the representation of logic gates as a BQM, as demonstrated in the *Multiple-Gate Circuit* example. The *Boolean NOT Gate* example presents a mathematical formulation of a BQM for a Boolean gate in detail.

Here we briefly repeat the steps of mathematically formulating a BQM while adding details on the underlying physical processes.

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

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

The biases and couplings define an energy landscape, and the D-Wave quantum computer seeks the minimum energy of that landscape. Once you express your problem in a formulation[1] such that desired outcomes have low energy values and undesired outcomes high energy values, the D-Wave system solves your problem by finding the low-energy states.

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

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

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

AND as a Penalty Function

This example represents the AND operation, $z \Leftrightarrow x_1 \wedge x_2$, where x_1, x_2 are the gate's inputs and z its output, using a *penalty function*:

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

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

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

Formulating the Problem as a QUBO

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

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

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

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

where h_i are biases and $J_{i,j}$ couplings between spins.

where $z = x_3$ is the AND gate's output, x_1, x_2 the inputs, linear coefficients are $q_1 = 3$, and quadratic coefficients are $q_{1,2} = 1, q_{1,3} = -2, q_{2,3} = -2$. The coefficients matrix is,

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

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

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

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

Solve the Problem by Sampling: Automated Minor-Embedding

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

Again we use sampler *DWaveSampler()* from Ocean software's dwave-system and its *EmbeddingComposite()* composite to *minor-embed* our unstructured problem (variables x1, x2, and z) on the sampler's graph structure (the QPU's numerically indexed qubits).

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

Note: In the code below, replace sampler parameters in the third line. If you configured a default solver, as described in *Using a D-Wave System*, you should be able to set the sampler without parameters as `sampler = DWaveSampler()`. You can see this information by running `dwave config inspect` in your terminal.

```
>>> from dwave.system.samplers import DWaveSampler
>>> from dwave.system.composites import EmbeddingComposite
>>> sampler = DWaveSampler(endpoint='https://URL_to_my_D-Wave_system/', token='ABC-
↪123456789012345678901234567890', solver='My_D-Wave_Solver')
>>> sampler_embedded = EmbeddingComposite(sampler)
```

As before, we ask for 5000 samples.

```
>>> response = sampler_embedded.sample_qubo(Q, num_reads=5000)
>>> for datum in response.data(['sample', 'energy', 'num_occurrences']):
...     print(datum.sample, "Energy: ", datum.energy, "Occurrences: ", datum.num_
↪occurrences)
...
{'x1': 1, 'x2': 0, 'z': 0} Energy:  0.0 Occurrences:  1009
{'x1': 1, 'x2': 1, 'z': 1} Energy:  0.0 Occurrences:  1452
{'x1': 0, 'x2': 0, 'z': 0} Energy:  0.0 Occurrences:  1292
{'x1': 0, 'x2': 1, 'z': 0} Energy:  0.0 Occurrences:  1246
{'x1': 0, 'x2': 1, 'z': 0} Energy:  0.0 Occurrences:  1
```

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

Note that the last line of output from this execution shows a single sample that seems identical to the line above it. The next section addresses that.

Solve the Problem by Sampling: Non-automated Minor-Embedding

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

Minor-Embedding a NOT Gate

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

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

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

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

```
>>> print(sampler.adjacency[sampler.nodelist[0]])
{128, 4, 5, 6, 7}
```

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

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

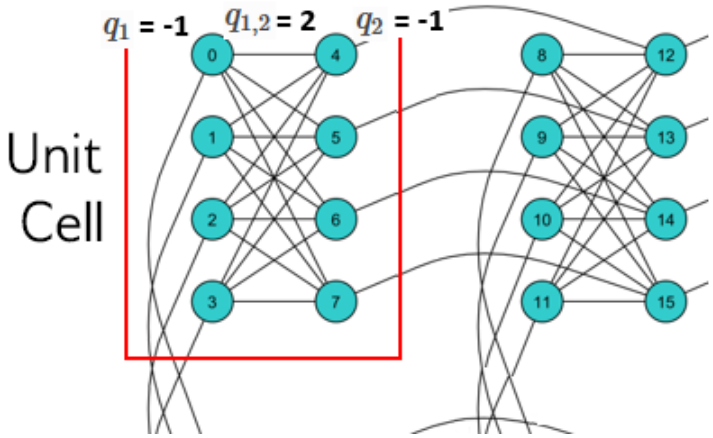

Fig. 3.10: A NOT gate minor embedded into the topmost left unit cell of a D-Wave 2000Q QPU. Variables x_1, x_2 are minor embedded as qubits 0 and 4 (blue circles). Biases $q_1, q_2 = -1, -1$ and coupling strength $q_{1,2} = 2$ are also shown.

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

```
>>> from dwave.system.composites import FixedEmbeddingComposite
>>> sampler_embedded = FixedEmbeddingComposite(sampler, {'x': [0], 'z': [4]})
>>> print(sampler_embedded.adjacency)
{'x': {'z'}, 'z': {'x'}}
```

As before, we ask for 5000 samples.

```
>>> response = sampler_embedded.sample_qubo(Q_not, num_reads=5000)
>>> for datum in response.data(['sample', 'energy', 'num_occurrences']):
...     print(datum.sample, "Energy: ", datum.energy, "Occurrences: ", datum.num_
↪occurrences)
...
{'x': 0, 'z': 1} Energy:  -1.0 Occurrences:  2520
{'x': 1, 'z': 0} Energy:  -1.0 Occurrences:  2474
{'x': 0, 'z': 0} Energy:  0.0 Occurrences:  4
{'x': 1, 'z': 1} Energy:  0.0 Occurrences:  2
```

From NOT to AND: an Important Difference

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

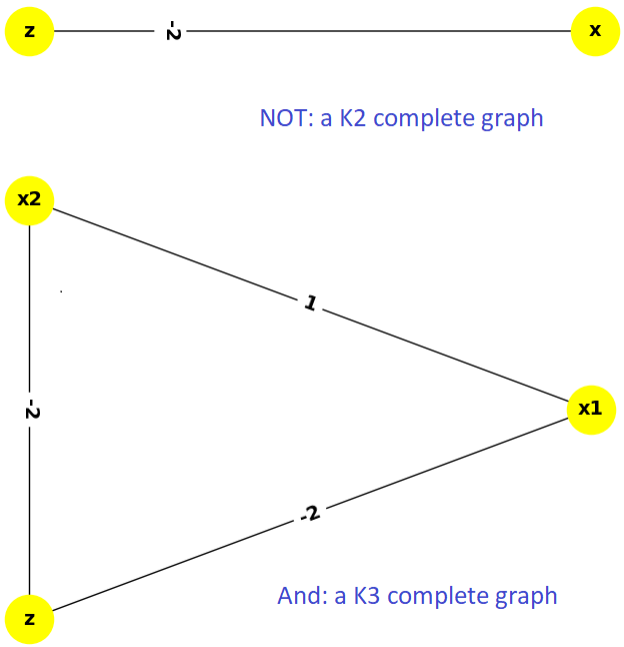

Fig. 3.11: NOT gate K_2 complete graph (top) versus AND gate K_3 complete graph (bottom.)

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

Minor-Embedding an AND Gate

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

To fit the 3-qubit loop into a 4-sided structure, create a chain of 2 qubits to represent a single variable. For example, chain qubit 0 and qubit 4 to represent variable z.

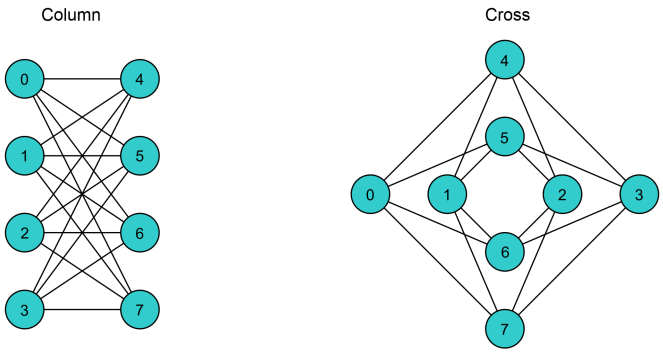

Fig. 3.12: Chimera unit cell illustrated in two layouts.

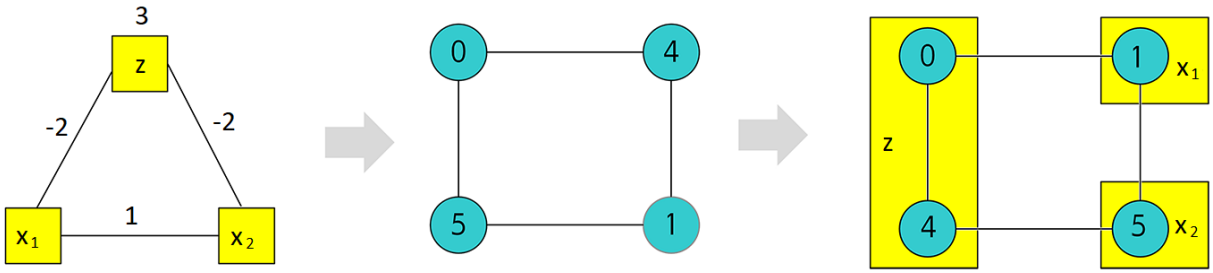

Fig. 3.13: Embedding a K_3 graph into Chimera by using a chain.

The strength of the coupler between qubits 0 and 4, which represents variable z, must be set to correlate the qubits strongly, so that in most solutions they have a single value for z. (Remember the output in the *Solve the Problem by Sampling: Automated Minor-Embedding* section with its identical two last lines? This was likely due to the qubits in a chain taking different values.)

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

```
>>> from dwave.system.composites import FixedEmbeddingComposite
>>> embedding = {'x1': {1}, 'x2': {5}, 'z': {0, 4}}
>>> sampler_embedded = FixedEmbeddingComposite(sampler, embedding)
>>> print(sampler_embedded.adjacency)
{'x1': {'x2', 'z'}, 'x2': {'x1', 'z'}, 'z': {'x1', 'x2'}}
```

We ask for 5000 samples.

```
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'): 3}
>>> response = sampler_embedded.sample_qubo(Q, num_reads=5000)
>>> for datum in response.data(['sample', 'energy', 'num_occurrences']):
...     print(datum.sample, "Energy: ", datum.energy, "Occurrences: ", datum.num_
↪occurrences)
...
{'z': 0, 'x1': 1, 'x2': 0} Energy:  0.0 Occurrences:  1088
{'z': 0, 'x1': 0, 'x2': 1} Energy:  0.0 Occurrences:  1806
{'z': 1, 'x1': 1, 'x2': 1} Energy:  0.0 Occurrences:  1126
{'z': 0, 'x1': 0, 'x2': 0} Energy:  0.0 Occurrences:  977
{'z': 1, 'x1': 0, 'x2': 1} Energy:  1.0 Occurrences:  2
{'z': 1, 'x1': 0, 'x2': 1} Energy:  1.0 Occurrences:  1
```

For comparison, the following code purposely weakens the chain strength (strength of the coupler between qubits 0

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

```
>>> print(sampler.properties['extended_j_range'])
[-2.0, 1.0]
>>> sampler_embedded = FixedEmbeddingComposite(sampler, embedding)
>>> response = sampler_embedded.sample_qubo(Q, num_reads=5000, chain_strength=0.25)
>>> for datum in response.data(['sample', 'energy', 'num_occurrences']):
...     print(datum.sample, "Energy: ", datum.energy, "Occurrences: ", datum.num_
↪occurrences)
...
{'z': 0, 'x1': 1, 'x2': 0} Energy:  0.0 Occurrences:  690
{'z': 0, 'x1': 0, 'x2': 1} Energy:  0.0 Occurrences:  936
{'z': 1, 'x1': 1, 'x2': 1} Energy:  0.0 Occurrences:  573
{'z': 0, 'x1': 0, 'x2': 0} Energy:  0.0 Occurrences:  984
{'z': 1, 'x1': 1, 'x2': 1} Energy:  0.0 Occurrences:  1
{'z': 1, 'x1': 1, 'x2': 0} Energy:  1.0 Occurrences:  525
{'z': 1, 'x1': 0, 'x2': 1} Energy:  1.0 Occurrences:  1289
{'z': 1, 'x1': 1, 'x2': 0} Energy:  1.0 Occurrences:  1
{'z': 0, 'x1': 1, 'x2': 1} Energy:  1.0 Occurrences:  1
```

- *Large Map Coloring* demonstrates out-of-the-box solving of an arbitrary-sized problem.
- *Vertex Cover* solves a small graph problem.
- *Constrained Scheduling* solves a small constraint satisfaction problem.
- *Boolean NOT Gate* mathematically formulates a BQM for a two-variable problem.
- *Boolean AND Gate* demonstrates programming the QPU more directly (*minor-embedding*).

Intermediate-Level Examples

Map Coloring

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

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

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

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

Example Requirements

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

- The requisite information for problem submission through SAPI, as described in *Using a D-Wave System*.
- Ocean tools dwavebinarycsp and dwave-system. For graphics, you will also need NetworkX.

Fig. 3.14: Coloring a map of Canada with four colors.

If you installed dwave-ocean-sdk and ran `dwave config create`, your installation should meet these requirements.

Solution Steps

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

The full workflow is as follows:

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

Four-Color Canadian Map

This example finds a solution to the map-coloring problem for a map of Canada using four colors (the sample code can easily be modified to change the number of colors or use different maps). Canada's 13 provinces are denoted by postal codes:

Table 3.7: Canadian Provinces' Postal Codes

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

Map Coloring: Full Code

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- `csp.add_constraint(one_color_configurations, variables)` represents the con-

straint that each node (province) select a single color, as represented by valid configurations `one_color_configurations = {(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)}`

- `csp.add_constraint(not_both_1, variables)` represents the constraint that two nodes (provinces) with a shared edge (border) not both select the same color.

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

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

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

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

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

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

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

The next code sets up a D-Wave system as the sampler and requests 50 samples.

Note: In the code below, replace sampler parameters as needed. If you configured a default solver, as described in *Using a D-Wave System*, you should be able to set the sampler without parameters as `sampler = EmbeddingComposite(DWaveSampler())`. You can see this information by running `dwave config inspect` in your terminal.

```
# Sample 50 times
sampler = EmbeddingComposite(DWaveSampler(endpoint='https://URL_to_my_D-Wave_system/',
↪ token='ABC-12345678901234567890123456789O', solver='My_D-Wave_Solver'))
response = sampler.sample(bqm, num_reads=50)
```

Note: The next code requires Matplotlib.

Plot a valid solution.

```
# Function that plots a returned sample
def plot_map(sample):
    G = nx.Graph()
    G.add_nodes_from(provinces)
```

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

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

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

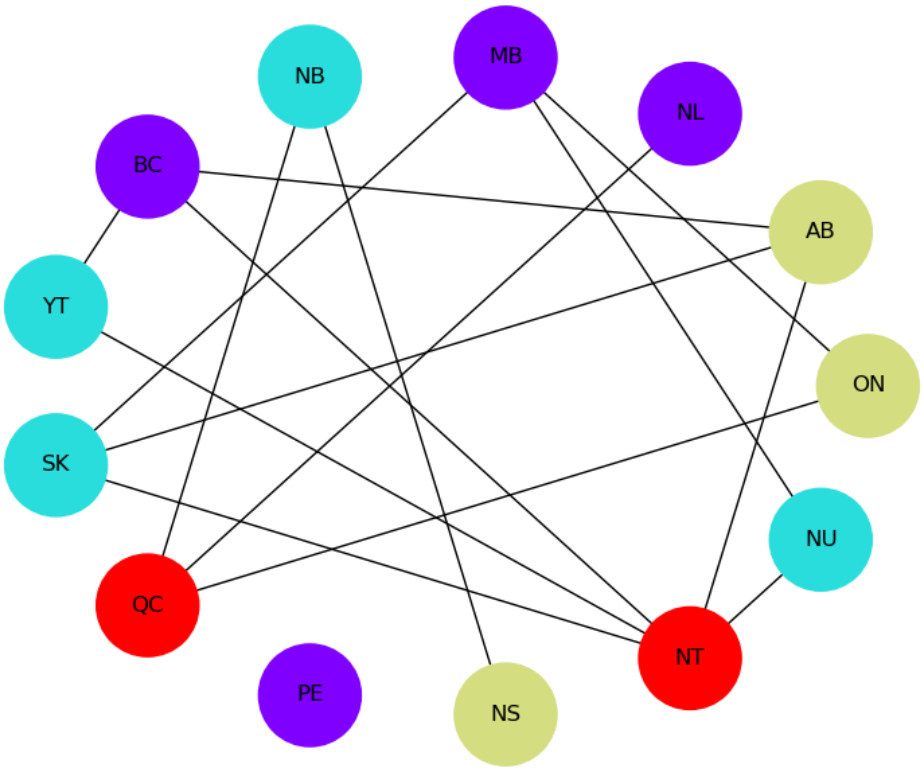

Fig. 3.15: Solution for a map of Canada with four colors. The graph comprises 13 nodes representing provinces connected by edges representing shared borders. No two nodes connected by an edge share a color.

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

(in1) at (0, 6.3) a; (in2) at (0, 5) b; (in3) at (0, 2.7) c; (in4) at (0, 1) d;
(out1) at (10.5, 4.1) z ;
(1.75,5) node[not port] (mynot1) 1 (2.25,3) node[or port] (myor2) 2
(5,6) node[and port] (myand3) 3 (5,1) node[or port] (myor4) 4
(7,5) node[and port] (myand5) 5 (7,1) node[not port] (mynot6) 6
(9.25,4) node[or port] (myor7) 7
(0.1, 6.25) – (myand3.in 1) (0.1, 5) – (mynot1.in) (0.1, 5) -| (myor2.in 1) (0.1, 2.7) – (myor2.in 2) (0.1, 0.75) – (myor4.in 2)
(mynot1.out) |- (myand3.in 2) (myor2.out) |- (myor4.in 1) (myand3.out) |- (myand5.in 1) (myor4.out) |- (myand5.in 2) (myor4.out) |- (mynot6.in) (myand5.out) |- (myor7.in 1) (mynot6.out) |- (myor7.in 2)
(myor7.out) – (10.4, 4.0);

Fig. 3.16: Logic circuit that implements $z = a\overline{b}c$+a $\overline{b}c + b\overline{d} + c\overline{d}$.

Multiple-Gate Circuit

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

A simple circuit is shown in Figure 3.16.

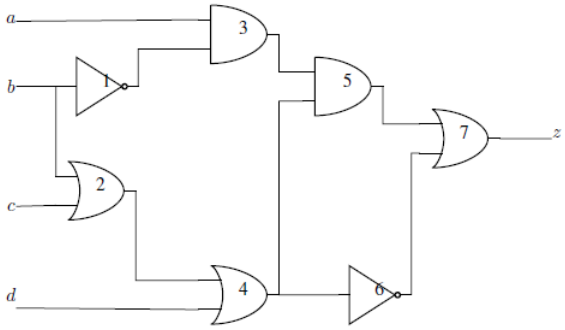

Fig. 3.17: Circuit with 7 logic gates, 4 inputs (a, b, c, d), and 1 output (z). The circuit implements function $z = \overline{b}(ac + ad + \overline{c}\overline{d})$.

Example Requirements

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

- The requisite information for problem submission through SAPI, as described in *Using a D-Wave System*
- Ocean tools dwavebinarycsp and dwave-system. For the optional graphics, you will also need Matplotlib.

If you installed dwave-ocean-sdk and ran `dwave config create`, your installation should meet these requirements.

Formulating the Problem as a CSP

This example demonstrates two formulations of constraints from the problem's logic gates:

1. Single comprehensive constraint:

```
import dwavebinarycsp

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

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

2. Multiple small constraints:

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

csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)
csp.add_constraint(operator.ne, ['b', 'not1'])  # add NOT 1 gate
csp.add_constraint(gates.or_gate(['b', 'c', 'or2']))  # add OR 2 gate
csp.add_constraint(gates.and_gate(['a', 'not1', 'and3']))  # add AND 3 gate
csp.add_constraint(gates.or_gate(['d', 'or2', 'or4']))  # add OR 4 gate
csp.add_constraint(gates.and_gate(['and3', 'or4', 'and5']))  # add AND 5 gate
csp.add_constraint(operator.ne, ['or4', 'not6'])  # add NOT 6 gate
csp.add_constraint(gates.or_gate(['and5', 'not6', 'z']))  # add OR 7 gate
```

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

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

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

Multiple-Gate Circuit: Further Details

Single Comprehensive Constraint

Binary Quadratic Model:

```
BinaryQuadraticModel({'a': 6.0, 'c': -2.0, 'z': 6.0, 'd': 0.0, 'b': -2.0,
'aux0': 0.0, 'aux1': 0.0}, {('aux1', 'a'): -4.0, ('z', 'a'): 0.0, ('aux1', 'aux0'): 0.
↪0,
('z', 'aux1'): -4.0, ('b', 'a'): -2.0, ('d', 'c'): 0.0, ('z', 'c'): -2.0, ('z', 'd'):␣
↪0.0,
('z', 'aux0'): -4.0, ('z', 'b'): 4.0, ('c', 'a'): -2.0, ('c', 'aux0'): 4.0, ('c', 'b
↪'): 2.0,
('d', 'a'): -2.0, ('d', 'aux1'): 4.0, ('d', 'aux0'): -2.0, ('b', 'aux0'): 2.0,
```

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

Embedding 1:

```
{'a': [946, 951, 959],
 'aux0': [954, 957],
 'aux1': [955, 1083, 1086, 1081, 1080],
 'b': [948, 956],
 'c': [944, 958, 950],
 'd': [953, 825, 831, 824],
 'z': [952]}
```

Embedding 2:

```
{'a': [1104],
 'aux0': [1110, 1102],
 'aux1': [1107, 981, 979, 977, 976],
 'b': [1111, 1103, 1098],
 'c': [1105, 1233, 1232, 1238],
 'd': [1108, 1100, 1099],
 'z': [1106, 1109]}
```

Embedding 3:

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

Embedding 4:

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

Multiple Small Constraints

Binary Quadratic Model:

```
BinaryQuadraticModel({'a': 0.0, 'c': 2.0, 'b': 0.0, 'not1': -2.0, 'd': 2.0,
'or4': 0.0, 'or2': 4.0, 'not6': 0.0, 'and5': 8.0, 'z': 2.0, 'and3': 6.0},
{('not6', 'and5'): 2.0, ('or2', 'c'): -4.0, ('or4', 'd'): -4.0, ('or4', 'and3'): 2.0,
('and3', 'a'): -4.0, ('and5', 'and3'): -4.0, ('z', 'and5'): -4.0, ('or4', 'not6'): 4.
↪0,
('or4', 'or2'): -4.0, ('z', 'not6'): -4.0, ('or2', 'd'): 2.0, ('or4', 'and5'): -4.0,
('c', 'b'): 2.0, ('b', 'not1'): 4.0, ('not1', 'and3'): -4.0, ('or2', 'b'): -4.0,
('not1', 'a'): 2.0}, -5.5, Vartype.BINARY)
```

Embedding 1:

```
{'a': [1870, 1737, 1862, 1856, 1865, 1609],
'and3': [1728, 1600],
'and5': [1731, 1735],
'b': [1603],
'c': [1605],
'd': [1606],
'not1': [1604, 1612],
'not6': [1730],
'or2': [1607, 1602],
'or4': [1733, 1729, 1601],
'z': [1734]}
```

Embedding 2:

```
{'a': [578, 450, 452],
'and3': [448, 576],
'and5': [447, 455],
'b': [574, 569],
'c': [572],
'd': [444],
'not1': [582],
'not6': [442, 446, 454],
'or2': [440, 568],
'or4': [441, 445, 453],
'z': [451]}
```

Embedding 3:

```
 {'a': [1051],
'and3': [1058, 1060, 1052],
'and5': [1059],
'b': [921, 924],
'c': [926],
'd': [1054],
'not1': [1049, 1055],
'not6': [1063, 1056],
'or2': [922, 1050],
'or4': [1061, 1053, 1048],
'z': [1062]}
```

Embedding 4:

```
 {'a': [1438],
'and3': [1560, 1432],
'and5': [1688, 1695],
'b': [1561, 1566],
'c': [1564],
'd': [1565],
'not1': [1439, 1433],
'not6': [1691, 1693],
'or2': [1563],
'or4': [1562, 1567, 1690],
'z': [1694]}
```

The first approach, which consolidates the circuit as a single constraint, yields a binary quadratic model with 7 variables: 4 inputs, 1, output, and 2 ancillary variables. The second approach, which creates a constraint satisfaction

problem from multiple small constraints, yields a binary quadratic model with 11 variables: 4 inputs, 1 output, and 6 intermediate outputs of the logic gates.

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

Minor-Embedding and Sampling

Algorithmic minor-embedding is heuristic—solution results vary significantly based on the minor-embedding found.

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

Note: In the code below, replace sampler parameters as needed. If you configured a default solver, as described in *Using a D-Wave System*, you should be able to set the sampler without parameters as `sampler = EmbeddingComposite(DWaveSampler())`. You can see this information by running `dwave config inspect` in your terminal.

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

# Set up a D-Wave system as the sampler
sampler = EmbeddingComposite(DWaveSampler(endpoint='https://URL_to_my_D-Wave_system/',
↪ token='ABC-12345678901234567890123456789O', solver='My_D-Wave_Solver'))
```

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

```
response = sampler.sample(bqm, num_reads=1000)

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

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

Table 3.8: Single Constraint

Embedding	(valid, invalid)
1	(39, 961)
2	(1000, 0)
3	(998, 2)
4	(316, 684)

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

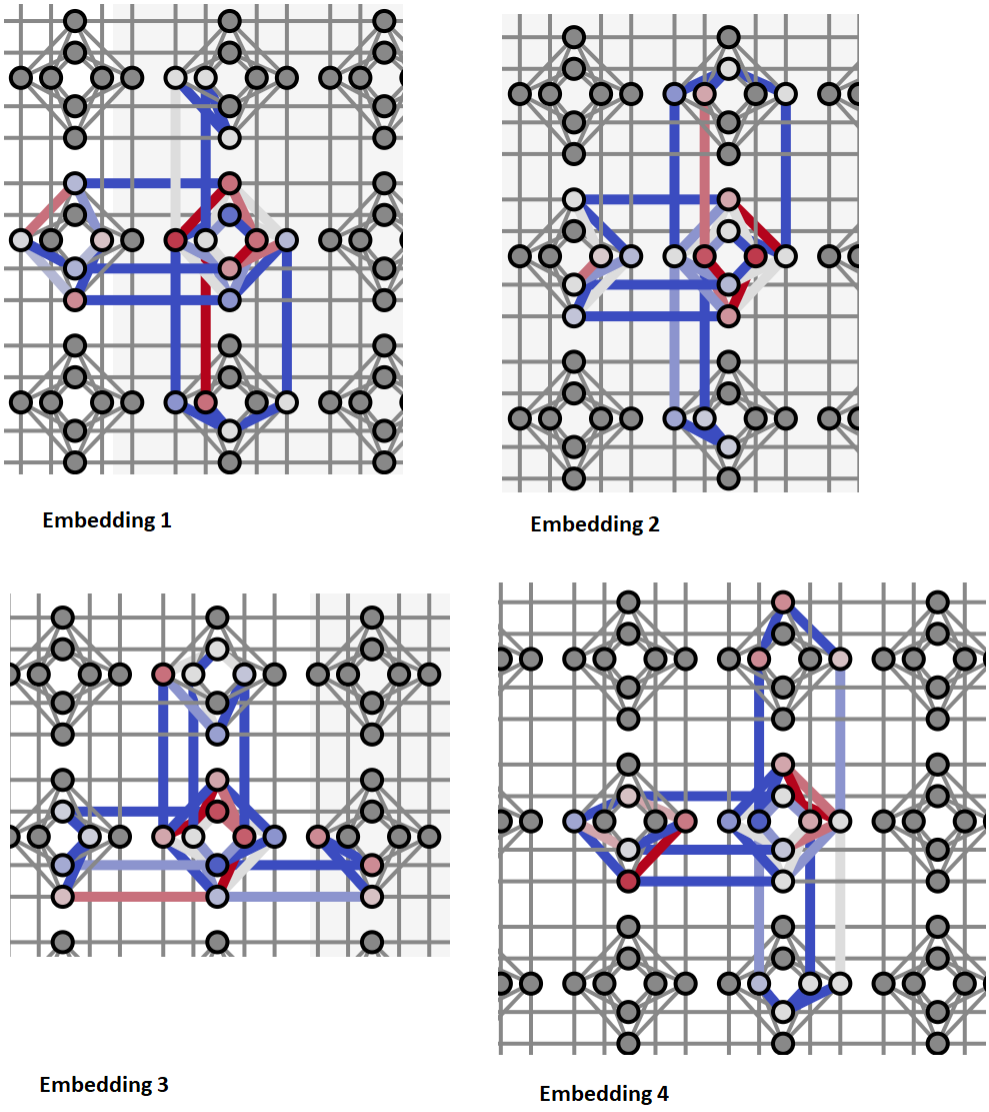

Fig. 3.18: Each of the figure's 4 panels shows a minor-embedding found for one run of the example code above. The panels show part of the Chimera graph representation of a D-Wave QPU, where each unit cell is rendered as a cross of 4 horizontal and 4 vertical dots representing qubits and lines representing couplers between qubit pairs. Color indicates the strengths of linear (qubit) and quadratic (coupler) biases: darker blue for increasingly negative values and darker red for increasingly positive values.

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

Table 3.9: Multiple Constraints

Embedding	Chain Strength	(valid, invalid)
1	1	(7, 993)
2	1	(417, 583)
3	2	(941, 59)
4	2	(923, 77)

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

Looking at the Results

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

```
>>> print(next(response.samples()))
{'a': 1, 'c': 0, 'b': 0, 'not1': 1, 'd': 1, 'or4': 1, 'or2': 0, 'not6': 0,
'and5': 1, 'z': 1, 'and3': 1}
```

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

$$\begin{aligned} z &= \bar{b}(ac + ad + \bar{c}\bar{d}) \\ &= 1(0 + 1 + 0) \\ &= 1 \end{aligned}$$

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

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

You can see in the graph that valid solutions have energy -9.5 and invalid solutions energies of -7.5, -5.5, and -3.5.

```
>>> for datum in response.data(['sample', 'energy', 'num_occurrences', 'chain_break_
↪fraction']):
...     print(datum)
...
Sample(sample={'a': 1, 'c': 0, 'b': 1, 'not1': 0, 'd': 1, 'or4': 1, 'or2': 1, 'not6':␣
↪0, 'and5': 0, 'z': 0, 'and3': 0}, energy=-9.5, num_occurrences=13, chain_break_
↪fraction=0.0)
Sample(sample={'a': 1, 'c': 1, 'b': 1, 'not1': 0, 'd': 0, 'or4': 1, 'or2': 1, 'not6':␣
↪0, 'and5': 0, 'z': 0, 'and3': 0}, energy=-9.5, num_occurrences=14, chain_break_
↪fraction=0.0)
# Snipped this section for brevity
Sample(sample={'a': 1, 'c': 0, 'b': 0, 'not1': 1, 'd': 1, 'or4': 1, 'or2': 0, 'not6':␣
↪1, 'and5': 1, 'z': 1, 'and3': 1}, energy=-7.5, num_occurrences=3, chain_break_
↪fraction=0.09090909090909091)
```

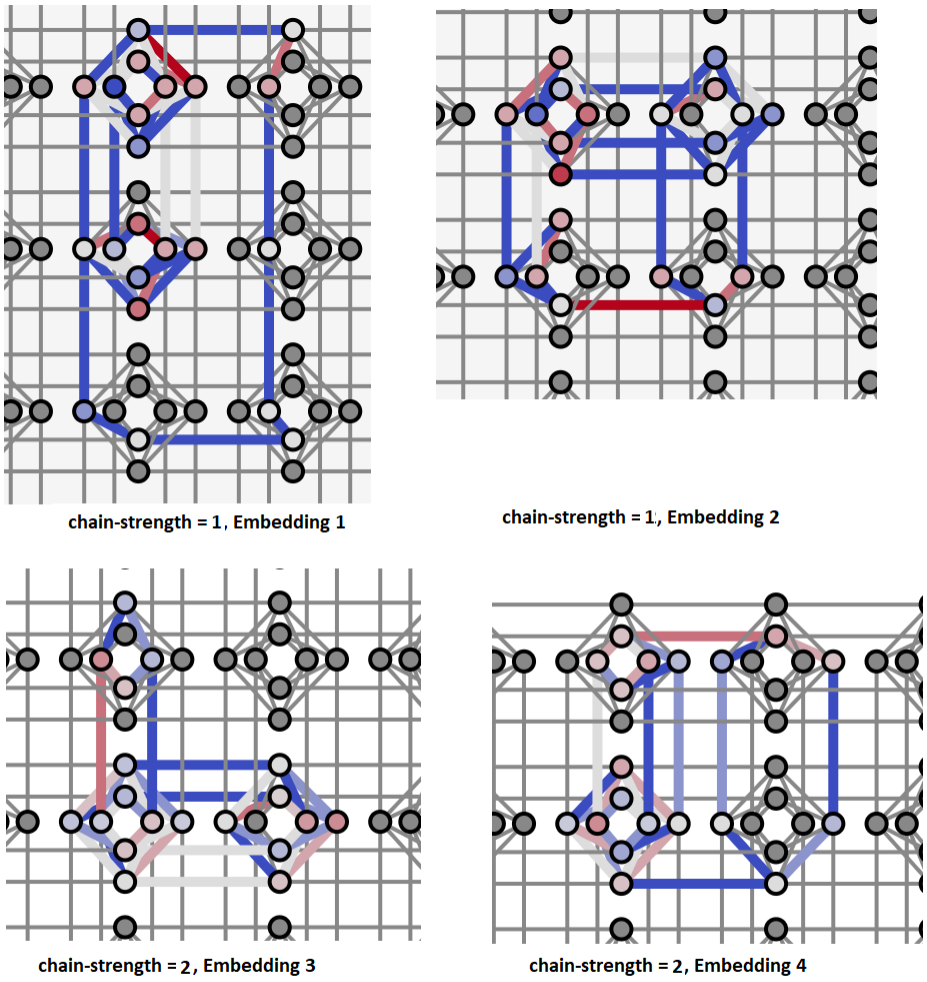

Fig. 3.19: Each of the figure's 4 panels shows a minor-embedding found for one run of the example code above, as described for the previous figure. Here, the top two panels are for runs with the default chain-strength of 1 and the bottom two for chain-strengths of 2.

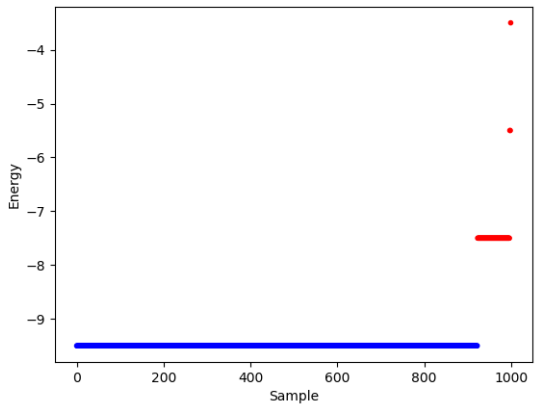

Fig. 3.20: Energies per sample for a 1000-sample problem submission of the logic circuit. Blue points represent valid solutions (solutions that solve the constraint satisfaction problem) and red points the invalid solutions.

```
Sample(sample={'a': 1, 'c': 1, 'b': 0, 'not1': 1, 'd': 1, 'or4': 1, 'or2': 1, 'not6':␣
↪1, 'and5': 1, 'z': 1, 'and3': 1}, energy=-7.5, num_occurrences=1, chain_break_
↪fraction=0.18181818181818182)
Sample(sample={'a': 1, 'c': 1, 'b': 1, 'not1': 1, 'd': 0, 'or4': 1, 'or2': 1, 'not6':␣
↪1, 'and5': 1, 'z': 1, 'and3': 1}, energy=-5.5, num_occurrences=4, chain_break_
↪fraction=0.18181818181818182)
# Snipped this section for brevity
Sample(sample={'a': 1, 'c': 1, 'b': 1, 'not1': 1, 'd': 0, 'or4': 1, 'or2': 1, 'not6':␣
↪1, 'and5': 0, 'z': 1, 'and3': 1}, energy=-3.5, num_occurrences=1, chain_break_
↪fraction=0.2727272727272727)
```

You can see, for example, that sample

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

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

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

on AND gate 3.

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

Problem With Many Variables

This example solves a graph problem with too many variables to fit onto the QPU.

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

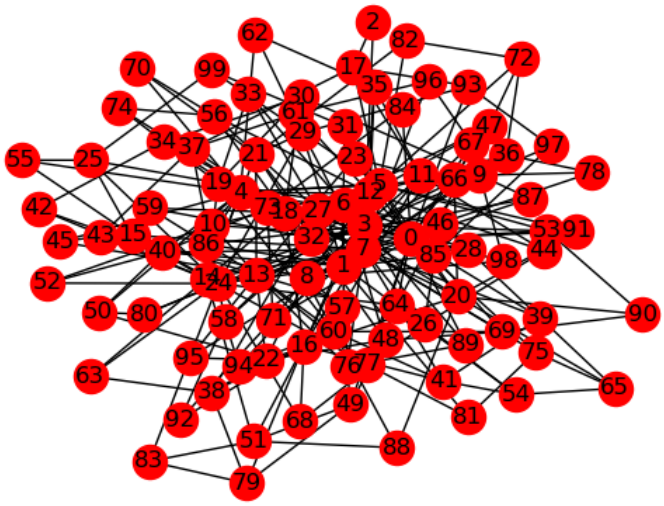

Fig. 3.21: Problem Graph with Many Variables.

Note: Currently Ocean tools are not optimized for very large problems. For fully connected graphs, the number of edges grows very quickly with increased nodes, degrading performance. The current example uses 100 nodes but with a degree of three (each node connects to three other nodes). You can increase the number of nodes substantially as long as you keep the graph sparse.

Example Requirements

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

- The requisite information for problem submission through SAPI, as described in *Using a D-Wave System*.
- Ocean tools dwave-system, dimod, and dwave-hybrid.

If you installed dwave-ocean-sdk and ran `dwave config create`, your installation should meet these requirements.

Solution Steps

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

Formulate the Problem

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

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

```
import random

graph = nx.barabasi_albert_graph(100, 3, seed=1)  # Build a quasi-random graph
# Set node and edge values for the problem
h = {v: 0.0 for v in graph.nodes}
J = {edge: random.choice([-1, 1]) for edge in graph.edges}
bqm = dimod.BQM(h, J, offset=0, vartype=dimod.SPIN)
```

Create a Hybrid Workflow

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

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

Solve the Problem Using Hybrid Resources

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

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

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

```
>>> print("Solution: sample={}".format(result.first))
Solution: sample=Sample(sample={0: -1, 1: -1, 2: -1, 3: 1, 4: -1, ... energy=-169.0,␣
↪num_occurrences=1)
```

- *Map Coloring* example solves a more complex constraint satisfaction problem.
- *Multiple-Gate Circuit* looks more deeply at *minor-embedding*.
- *Problem With Many Variables* illustrates solving a large problem using both classical and quantum resources.

Advanced-Level Examples

Working With Different Topologies

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

The code examples below will use the following imports:

```
>>> import neal
>>> import dimod
>>> import dwave_networkx as dnx
>>> import networkx as nx
>>> import dwave.embedding
...
>>> from dwave.system import DWaveSampler, EmbeddingComposite
```

Creating a Chimera Sampler

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

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

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

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

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

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

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

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

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

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

We can even use the sampler with the `dwave.system.EmbeddingComposite`

```
>>> embedding_sampler = EmbeddingComposite(sampler)
```

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

```
>>> qpu_sampler = DWaveSampler(solver={'qpu': True, 'num_active_qubits__within':␣
↪[2000, 2048]})
>>> QPUGraph = nx.Graph(qpu_sampler.edgelist)
>>> all(v in C16.nodes for v in QPUGraph.nodes)
True
>>> all(edge in C16.edges for edge in QPUGraph.edges)
True
```

Creating a Pegasus Sampler

Another topology of interest is the Pegasus topology.

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

```
>>> P6 = dnx.pegasus_graph(6)
>>> classical_sampler = neal.SimulatedAnnealingSampler()
>>> sampler = dimod.StructureComposite(classical_sampler, P6.nodes, P6.edges)
```

Working With Embeddings

The example above using the `EmbeddingComposite` hints that we might be interested in trying *embedding* with different topologies.

One thing we might be interested in is the *chain length* when embedding our problem. Say that we have a *fully connected* problem with 40 variables and we want to know the chain length needed to embed it on a 2048 node *Chimera* graph.

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

```
>>> num_variables = 40
>>> embedding = dwave.embedding.chimera.find_clique_embedding(num_variables, 16)
>>> max(len(chain) for chain in embedding.values())
11
```

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

```
>>> num_variables = 40
>>> embedding = dwave.embedding.pegasus.find_clique_embedding(num_variables, 6)
>>> max(len(chain) for chain in embedding.values())
6
```

- *Working With Different Topologies* running your code on software with different *QPU*-inspired topologies.

3.1.4 Demonstrations

Copy (clone) open-source code to run demos of solving known problems on a D-Wave system.

- Circuit Fault Diagnosis

  Demonstrates the use of the D-Wave system to solve circuit fault diagnosis, the problem of identifying a minimum-sized set of components that, if faulty, explains an observation of incorrect outputs given a set of inputs.

- Factoring

  Demonstrates the use of the D-Wave system to factor numbers in an entirely new way, by turning a multiplication circuit into a constraint satisfaction problem that allows the quantum computer to compute inputs from a predefined output. Essentially, this means running the multiplication circuit in reverse.

- Structural Imbalance

Demonstrates the use of the D-Wave system for analyzing the structural imbalance on a signed social network. This demo calculates and shows structural imbalance for social networks of militant organization based on data from the Stanford Militants Mapping Project.

3.1.5 Additional Tutorials

- Getting Started with the D-Wave System

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

- D-Wave Problem-Solving Handbook

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

- Leap

  Leap includes Jupyter Notebooks that provide additional tutorial examples and a framework to help you develop your own code.

3.2 Tools

D-Wave Ocean tools are documented on *Read the Docs*. Click on a link below for the documentation for each tool (or the link in parentheses for the tool repository located at D-Wave on GitHub).

Table 3.10: Ocean Software

Tool	Description
dimod (repo)	Shared API for binary quadratic *sampler*s. dimod provides a binary quadratic model (BQM) class that contains *Ising* and quadratic unconstrained binary optimization (*QUBO*) models used by samplers such as the D-Wave system. It also provides utilities for constructing new samplers and composed samplers.
dwavebinarycsp (repo)	Library to construct a binary quadratic model from a constraint satisfaction problem with small constraints over binary variables.
dwave-cloud-client (repo)	Minimal implementation of the REST interface used to communicate with D-Wave *Sampler* API (SAPI) servers.
dwave-hybrid (repo)	A general, minimal Python framework for building hybrid asynchronous decomposition samplers for quadratic unconstrained binary optimization (QUBO) problems.
dwave-neal (repo)	An implementation of a simulated annealing sampler.
dwave-networkx (repo)	Extension of NetworkX—a Python language package for exploration and analysis of networks and network algorithms—for users of D-Wave Systems. dwave-networkx provides tools for working with *Chimera* graphs and implementations of graph-theory algorithms on the D-Wave system and other binary quadratic model *sampler*s.
dwave-ocean-sdk (repo)	Installer for D-Wave's Ocean Tools.
dwave-system (repo)	Basic API for easily incorporating the D-Wave system as a *sampler* in the D-Wave Ocean software stack. It includes DWaveSampler, a dimod sampler that accepts and passes system parameters such as system identification and authentication down the stack. It also includes several useful composites—layers of pre- and post-processing—that can be used with DWaveSampler to handle *minor-embedding*, optimize chain strength, etc.
dwave-tabu (repo)	An implementation of the MST2 multistart tabu search algorithm for quadratic unconstrained binary optimization (QUBO) problems with a dimod Python wrapper.
penaltymodel (repo)	An approach to solve a constraint satisfaction problem (CSP) using an *Ising* model or a *QUBO*, is to map each individual constraint in the CSP to a 'small' Ising model or QUBO. Includes a local cache for penalty models and a factory that generates penalty models using SMT solvers.
minorminer (repo)	A tool for finding graph *minor-embedding*s, developed to embed *Ising* problems onto quantum annealers (QA). While it can be used to find minors in arbitrary graphs, it is particularly geared towards the state of the art in QA: problem graphs of a few to a few hundred variables, and hardware graphs of a few thousand qubits.
qbsolv (repo)	A decomposing solver that finds a minimum value of a large quadratic unconstrained binary optimization (*QUBO*) problem by splitting it into pieces. The pieces are solved using a classical solver running the tabu algorithm. qbsolv also enables configuring a D-Wave system as the solver.

3.3 Glossary

binary quadratic model

BQM A collection of binary-valued variables (variables that can be assigned two values, for example -1, 1) with

associated linear and quadratic biases. Sometimes referred to in other tools as a problem.

Chain length The number of qubits in a *Chain*.

Chain One or more nodes or qubits in a target graph that represent a single variable in the source graph. See *embedding*.

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

Complete graph

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

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

Embed

Embedding

Minor-embed

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

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

$$\mathcal{H}_{ising} = \underbrace{\frac{A(s)}{2}\left(\sum_i \hat{\sigma}_x^{(i)}\right)}_{\text{Initial Hamiltonian}} + \underbrace{\frac{B(s)}{2}\left(\sum_i h_i \hat{\sigma}_z^{(i)} + \sum_{i>j} J_{i,j}\hat{\sigma}_z^{(i)}\hat{\sigma}_z^{(j)}\right)}_{\text{Final Hamiltonian}} \tag{3.1}$$

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

Hardware graph See hardware graph. The hardware graph is the physical lattice of interconnected qubits. See also *working graph*.

Ising Traditionally used in statistical mechanics. Variables are "spin up" (↑) and "spin down" (↓), states that correspond to $+1$ and -1 values. Relationships between the spins, represented by couplings, are correlations or anti-correlations. The *objective function* expressed as an Ising model is as follows:

$$\mathrm{E}_{ising}(\boldsymbol{s}) = \sum_{i=1}^{N} h_i s_i + \sum_{i=1}^{N}\sum_{j=i+1}^{N} J_{i,j} s_i s_j \tag{3.2}$$

where the linear coefficients corresponding to qubit biases are h_i, and the quadratic coefficients corresponding to coupling strengths are $J_{i,j}$.

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

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

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

QPU Quantum processing unit

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

$$f(x) = \sum_i Q_{i,i} x_i + \sum_{i<j} Q_{i,j} x_i x_j \tag{3.3}$$

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

$$\min_{x \in \{0,1\}^n} x^T Q x. \tag{3.4}$$

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

$$\mathrm{E}_{qubo}(a_i, b_{i,j}; q_i) = \sum_i a_i q_i + \sum_{i<j} b_{i,j} q_i q_j. \tag{3.5}$$

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

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

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

Subgraph See subgraph on wikipedia.

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

3.4 How to Contribute

The goal of this document is to establish a common understanding among software contributors to D-Wave's Ocean software projects based on the code conventions and best practices used at D-Wave.

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

3.4.1 Testing

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

3.4.2 Submitting Changes

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

Follow the commit conventions described here:

- https://chris.beams.io/posts/git-commit/
- http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html

TL;DR:

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

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

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

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

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

3.4.3 Coding Conventions

- Variable naming should follow the well-known conventions of a language. Avoid uninformative or needlessly terse variable names.
- Code is read more often than written.
- Functions should do one thing.
- Early pull requests and code reviews.
- Early architecting/design. Code reviews can happen before any code has been written.
- Use a consistent character width of 120.
- Use 4 spaces instead of tabs.
- End all files with a newline.

Documentation and Comments

- Do a good job of commenting. Read this Coding Horror article.
- Comments should add, not repeat: avoid repeating in English or pseudo-code what the code does. Rather, discuss what the block is trying to achieve.
- Side effects should be visible on screen; if not in code, then in comments.
- Remember, **the best documentation is clean, simple code with good variable names**. When this is not possible, you must use a comment to explain the purpose of a functional block.

  Example:

  The following code

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

  would be much more informative as

```
# if z is greater than 255, this universe will collapse. See https://url.to.issue.
↪tracker/IS-42
if z > 255:
    raise RuntimeError('z must be <= 255!')
```

  or even better:

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

Python

pep8

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

Python 2/3

All code should be both Python 2 and 3 compatible.

Documentation

- Google docstrings convention (definition, example) on all public-facing functions. The following are exceptions:
  - For D-Wave extensions of third-party projects, we match the existing convention (e.g. the D-Wave NetworkX project follows NumPy conventions).
  - Argument defaults are written "default=x" rather than "default x".
- Private functions should include some sort of docstring.
- If your module has more than one public unit, it should have a module docstring with a table of contents.

- The docstring for the `__init__` method goes on the class.
- All docstrings should be parsable by the Sphinx documentation generation tool (i.e. reStructuredText) The sphinx theme should be readthedocs.

C++

.clang-format

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

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

as opposed to

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

Additional Style

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

Example:

Use the following format

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

as opposed to

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

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

Versioning Scheme

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

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

Readme File

If you are creating a repository, don't forget to include a `README.rst` containing a reasonable description of your project.

CHAPTER 4

Indices and tables

- genindex
- search

Index