The Quantum Computing Company™

Developer Guide for Python

USER MANUAL

2018-03-21

Overview

This document describes how to interact with D-Wave solvers using the Solver API (SAPI) Python client library.

CONTACT

Corporate Headquarters
3033 Beta Ave
Burnaby, BC V5G 4M9
Canada
Tel. 604-630-1428

US Office
2650 E Bayshore Rd
Palo Alto, CA 94303

Email: info@dwavesys.com

www.dwavesys.com

Contents

CHAPTER 1

INTRODUCTION

The D-Wave QPU is based on a physical lattice of qubits and the couplers that connect them. Together, these qubits and couplers are referred to as the Chimera graph. The lattice structure is a set of connected unit cells, each comprising four horizontal qubits connected to four vertical qubits via couplers. Unit cells are tiled vertically and horizontally with adjacent qubits connected, creating a lattice of sparsely connected qubits. Within a given system, certain qubits or couplers may not function as desired. In such cases, the devices are eliminated from the programmable fabric available. The subgraph available to solve a problem is called the *working graph*.

A given *logical problem* defined on a general graph can be mapped to a *physical problem* defined on the working graph using *chains*. A chain is a collection of qubits bound together to represent a single logical node. The association between the logical problem and the physical problem is carried out by minor *embedding*.

The qubits, denoted q_i, implement the Ising spins. Their physical connectivity determines which couplings, $J_{i,j}$, can be set to nonzero values. The allowed connectivity is described with a Chimera graph; see Figure 1.1. An $M \times N \times L$ Chimera graph consists of an $M \times N$ two-dimensional lattice of blocks, with each block consisting of $2L$ variables, for a total of $2MNL$ variables.

Any discrete optimization problem can be cast as a Chimera-structured Ising problem given a large enough Chimera lattice. Methods are available to reduce higher-order interactions in the optimization objective to pairwise, and to address the connectivity mismatches between the problem and the fixed qubit connectivity of Chimera. In this document, we provide an overview of software tools (available in Python[1] packs) that solve these problems using hardware or software solvers and make formulating and solving problems simpler. Further information is available from the command line.

The functionalities of the utility packs include:

1. Managing connections to solvers, and in particular, solving Ising/QUBO problems by quantum annealing or simulated quantum annealing.
2. Simplifying Ising/QUBO problems to equivalent, but easier to solve problems.
3. Solving non-Chimera-structured problems in solvers using embeddings (special mappings of problem variables to qubits and find embeddings).
4. Reducing objective functions with high-order interactions to QUBO problems.
5. Using QSage quantum accelerator tool.
6. Utilities that help convert between Ising and QUBO formats and get information about qubit connectivity.

[1] Python is a trademark or registered trademark of the Python Software Foundation ("PSF").

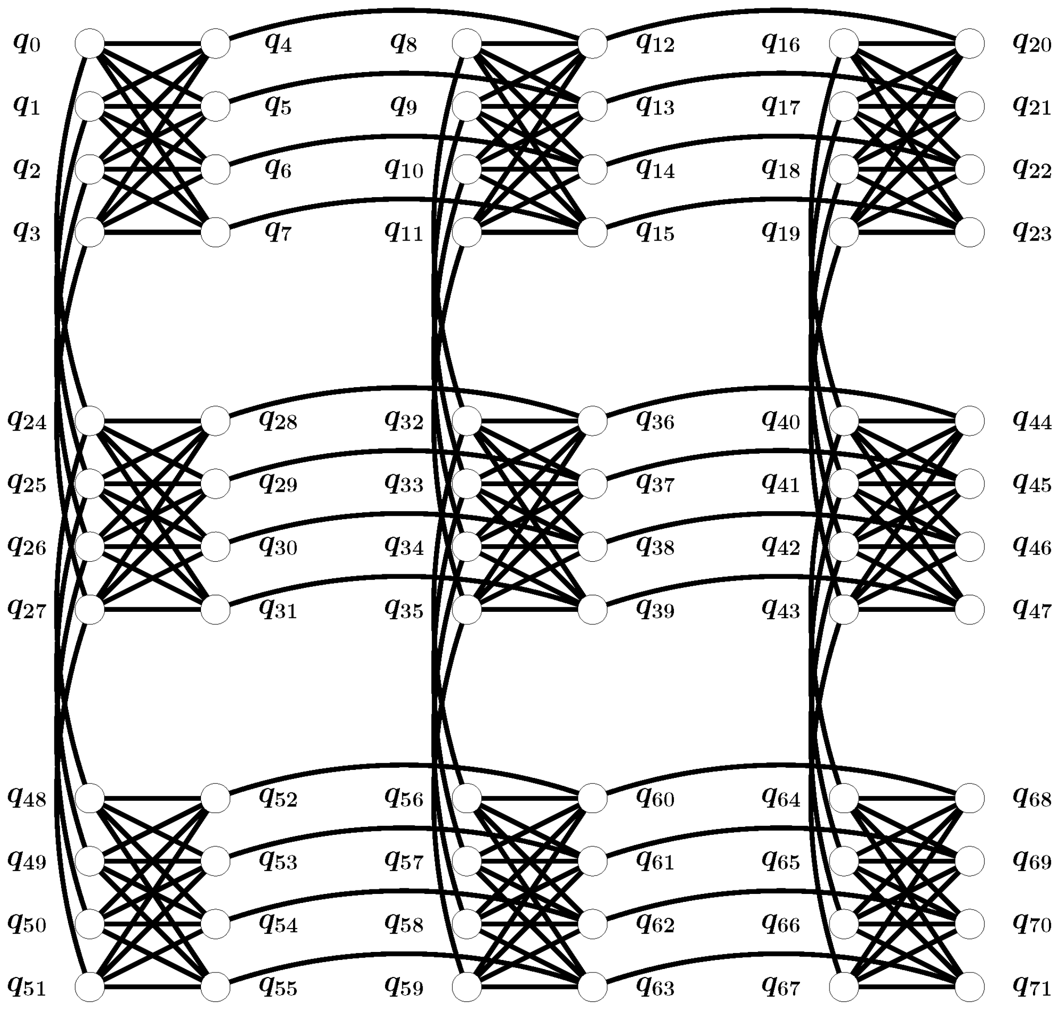

Figure 1.1: A $3 \times 3 \times 4$ Chimera graph. Nodes in an $M \times N \times L$ Chimera graph represent each of the $2MNL$ qubits, labeled as q_i. Edges (connections between nodes) in the graph, $J_{i,j}$, indicate couplings that may be nonzero. As an example, $J_{3,4}$ may be nonzero because an edge connects qubits 3 and 4, but $J_{2,3}$ must always be zero because no edge connects qubits 2 and 3. The basic repeating block of Chimera (a block of $2L$ variables with complete bipartite connectivity) may be tiled into an $M \times N$ lattice. The left-side variables within each block connect vertically; the right-side variables, horizontally.

Note: The results shown in the examples of this document may differ from the results you would get from running the same examples depending on the target solver and its parameters.

CHAPTER 2

CONNECTING TO THE SOLVER

2.1 Connecting to a Remote or Local Solver

In the D-Wave system, a solver is simply a resource that runs a problem. Some solvers interface to the D-Wave QPU; others leverage CPU and GPU resources to solve problems. Software solvers, like optimizing and sampling emulators, mimic the problem-solving behaviors of the QPU and are useful for prototyping algorithms that make multiple calls to the hardware.

Before you submit a problem to the D-Wave system, you need to know the properties of the solver and what parameters it accepts with the problem submission—how many reads of the problem, you want, for example. This section explains how to connect to the Solver API (SAPI) to retrieve the list of available solvers and their properties.

The types of solvers available are shown below.

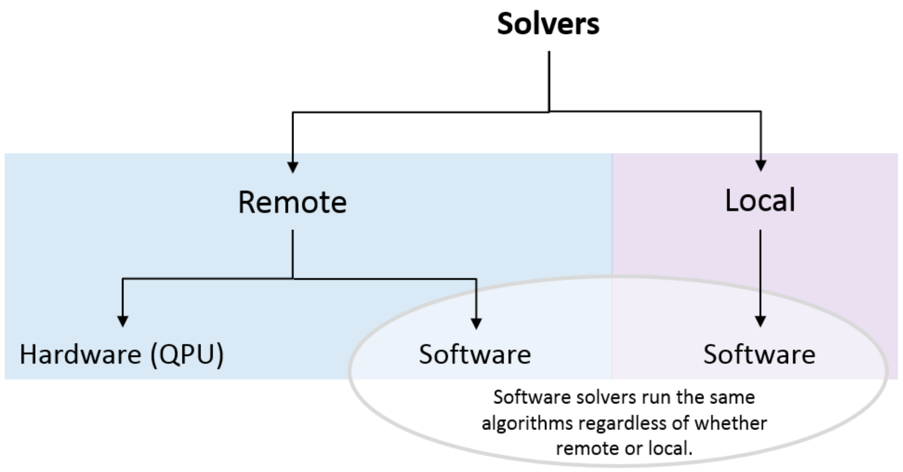
Solvers
Remote
Local
Hardware (QPU)
Software
Software
Software solvers run the same algorithms regardless of whether remote or local.

Figure 2.1: Solver tree diagram

You can connect to either a remote solver, which runs remotely, or a local solver, which runs on your own machine. A remote solver can be either a hardware solver (the QPU) or a software solver. Regardless of whether local or remote, the software solvers run the same algorithms.

To connect to SAPI, you need the know the system URL and have an active API token. Obtain these through Qubist. The sample code below uses the URL and token to connect to SAPI, and then prints the names of all solvers accessible by the token.

```
from dwave_sapi2.remote import RemoteConnection

url = 'system-url'
token = 'your-api-token'
conn = RemoteConnection(url, token)
print conn.solver_names()
```

For more code examples, see the `/examples` directory in the client download package.

The remainder of this chapter describes the other methods you need to establish a connection and send a problem to a solver for solution.

2.1.1 RemoteConnection

If using a remote solver, a connection to the solver must be established through *RemoteConnection*.

This creates a remote connection handle to the available software solvers and hardware solvers. Error conditions are raised if any of the inputs are invalid.

```
from dwave_sapi2.remote import RemoteConnection

remote_connection = RemoteConnection(url, token)
remote_connection = RemoteConnection(url, token, proxy_url)
```

Parameters

url: SAPI URL.

token: API token.

proxy_url: Proxy URL.

Return Value

remote_connection — Remote connection object. Retrieves available solver names and creates a remote solver object through the *solver_names* and *get_solver* methods.

To retrieve all the available remote solver names in *remote_connection*:

```
solver_names = remote_connection.solver_names()
```

To retrieve a remote solver object, identify it by the name retrieved through the *solver_names* call:

```
solver = remote_connection.get_solver("name")
```

Access solver properties through the *properties* attribute. See the *Solver Properties and Parameters Reference* for more information on solver properties.

Example

Example 1

This basic example shows how to establish a connection to a remote solver. This example assumes that you have already obtained the proper URL and have a token.

```
from dwave_sapi2.remote import RemoteConnection

# define the url and a valid token
url = "http://myURL"
token = "myToken001"

# create a remote connection using url and token
remote_connection = RemoteConnection(url, token)
```

Example 2

This example shows how to retrieve the available solvers, create a solver object, and retrieve the properties of the solver.

```
from dwave_sapi2.remote import RemoteConnection

# define the url and a valid token
url = "http://myURL"
token = "myToken001"

solver_name = "solver_name"

# create a remote connection using url and token
remote_connection = RemoteConnection(url, token)
# get a solver
solver = remote_connection.get_solver(solver_name)

# get solver's properties
print solver.properties
```

2.1.2 local_connection

```
from dwave_sapi2.local import local_connection
```

If you choose to use a local solver instead of a remote solver as in *RemoteConnection*, establish a connection to the local solver through *local_connection*. This call retrieves available solver names and creates a local solver object through the *solver_names* and *get_solver* methods.

To retrieve all the available remote solver names available through the local connection:

```
solver_names = local_connection.solver_names()
```

To retrieve a local solver object, identify it by the name retrieved through the *solver_names* call:

```
solver = local_connection.get_solver("name")
```

Access solver properties through the *properties* attribute. See the *Solver Properties and Parameters Reference* for more information on solver properties.

Example

This example shows how to retrieve the available solvers, create a solver object to be used and retrieve the properties of the solver to be used from a local connection.

```
from dwave_sapi2.local import local_connection

# get a solver
solver = local_connection.get_solver("c4-sw_sample")

# get solver's properties
print solver.properties

Output:

{'qubits': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,␣
 ↪20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,␣
 ↪42,
43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,␣
 ↪64,
65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,␣
 ↪86,
87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105,␣
 ↪106,
107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123,
124, 125, 126, 127], 'couplers': [[0, 4], [0, 5], [0, 6], [0, 7], [0, 32], [1, 4],
[1, 5], [1, 6], [1, 7], [1, 33], [2, 4], [2, 5], [2, 6], [2, 7], [2, 34], [3, 4],
[3, 5], [3, 6], [3, 7], [3, 35], [4, 12], [5, 13], [6, 14], [7, 15], [8, 12], [8,␣
 ↪13],
[8, 14], [8, 15], [8, 40], [9, 12], [9, 13], [9, 14], [9, 15], [9, 41], [10, 12],
[10, 13], [10, 14], [10, 15], [10, 42], [11, 12], [11, 13], [11, 14], [11, 15],
[11, 43], [12, 20], [13, 21], [14, 22], [15, 23], [16, 20], [16, 21], [16, 22],
[16, 23], [16, 48], [17, 20], [17, 21], [17, 22], [17, 23], [17, 49], [18, 20],
[18, 21], [18, 22], [18, 23], [18, 50], [19, 20], [19, 21], [19, 22], [19, 23],
[19, 51], [20, 28], [21, 29], [22, 30], [23, 31], [24, 28], [24, 29], [24, 30],
[24, 31], [24, 56], [25, 28], [25, 29], [25, 30], [25, 31], [25, 57], [26, 28],
[26, 29], [26, 30], [26, 31], [26, 58], [27, 28], [27, 29], [27, 30], [27, 31],
[27, 59], [32, 36], [32, 37], [32, 38], [32, 39], [32, 64], [33, 36], [33, 37],
[33, 38], [33, 39], [33, 65], [34, 36], [34, 37], [34, 38], [34, 39], [34, 66],
[35, 36], [35, 37], [35, 38], [35, 39], [35, 67], [36, 44], [37, 45], [38, 46],
[39, 47], [40, 44], [40, 45], [40, 46], [40, 47], [40, 72], [41, 44], [41, 45],
[41, 46], [41, 47], [41, 73], [42, 44], [42, 45], [42, 46], [42, 47], [42, 74],
[43, 44], [43, 45], [43, 46], [43, 47], [43, 75], [44, 52], [45, 53], [46, 54],
[47, 55], [48, 52], [48, 53], [48, 54], [48, 55], [48, 80], [49, 52], [49, 53],
[49, 54], [49, 55], [49, 81], [50, 52], [50, 53], [50, 54], [50, 55], [50, 82],
[51, 52], [51, 53], [51, 54], [51, 55], [51, 83], [52, 60], [53, 61], [54, 62],
[55, 63], [56, 60], [56, 61], [56, 62], [56, 63], [56, 88], [57, 60], [57, 61],
[57, 62], [57, 63], [57, 89], [58, 60], [58, 61], [58, 62], [58, 63], [58, 90],
[59, 60], [59, 61], [59, 62], [59, 63], [59, 91], [64, 68], [64, 69], [64, 70],
[64, 71], [64, 96], [65, 68], [65, 69], [65, 70], [65, 71], [65, 97], [66, 68],
[66, 69], [66, 70], [66, 71], [66, 98], [67, 68], [67, 69], [67, 70], [67, 71],
[67, 99], [68, 76], [69, 77], [70, 78], [71, 79], [72, 76], [72, 77], [72, 78],
[72, 79], [72, 104], [73, 76], [73, 77], [73, 78], [73, 79], [73, 105], [74, 76],
[74, 77], [74, 78], [74, 79], [74, 106], [75, 76], [75, 77], [75, 78], [75, 79],
[75, 107], [76, 84], [77, 85], [78, 86], [79, 87], [80, 84], [80, 85], [80, 86],
[80, 87], [80, 112], [81, 84], [81, 85], [81, 86], [81, 87], [81, 113], [82, 84],
[82, 85], [82, 86], [82, 87], [82, 114], [83, 84], [83, 85], [83, 86], [83, 87],
[83, 115], [84, 92], [85, 93], [86, 94], [87, 95], [88, 92], [88, 93], [88, 94],
[88, 95], [88, 120], [89, 92], [89, 93], [89, 94], [89, 95], [89, 121], [90, 92],
[90, 93], [90, 94], [90, 95], [90, 122], [91, 92], [91, 93], [91, 94], [91, 95],
[91, 123], [96, 100], [96, 101], [96, 102], [96, 103], [97, 100], [97, 101], [97,␣
↪102],
[97, 103], [98, 100], [98, 101], [98, 102], [98, 103], [99, 100], [99, 101], [99,␣
↪102],
[99, 103], [100, 108], [101, 109], [102, 110], [103, 111], [104, 108], [104, 109],
[104, 110], [104, 111], [105, 108], [105, 109], [105, 110], [105, 111], [106, 108],
[106, 109], [106, 110], [106, 111], [107, 108], [107, 109], [107, 110], [107, 111],
[108, 116], [109, 117], [110, 118], [111, 119], [112, 116], [112, 117], [112, 118],
[112, 119], [113, 116], [113, 117], [113, 118], [113, 119], [114, 116], [114, 117],
[114, 118], [114, 119], [115, 116], [115, 117], [115, 118], [115, 119], [116, 124],
[117, 125], [118, 126], [119, 127], [120, 124], [120, 125], [120, 126], [120, 127],
[121, 124], [121, 125], [121, 126], [121, 127], [122, 124], [122, 125], [122, 126],
[122, 127], [123, 124], [123, 125], [123, 126], [123, 127]],
'supported_problem_types': ['ising', 'qubo'], 'num_qubits': 128}
```

See also:

RemoteConnection

2.2 Sending Problems to a Solver

2.2.1 Solver Parameters: Keywords

Solver parameters (keywords) vary by solver. Before you send a problem to a solver, check its properties to determine what parameters it accepts. See the the *Solver Properties and Parameters Reference* for a description of each parameter, for the range of permitted values, and for default settings.

2.2.2 solve_ising

```
from dwave_sapi2.core import solve_ising

answer = solve_ising(solver, h, J)
answer = solve_ising(solver, h, J, param_name=value, ...)
```

Solve an Ising problem synchronously.

Note: We call a working qubit i *active* if either h_i is non-zero, or there is another working qubit j and a working coupler between i and j for which J_{ij} is non-zero. Non-working qubits must always be inactive.

Parameters

solver — Solver object.

h — List or tuple of the linear Ising coefficients. The *h* value of a non-working qubit must be zero or an exception will be raised. Inactive qubits are disabled during annealing and cannot distinguish their states.

J — Dictionary of Ising coupling coefficients. Diagonal entries must be zero. Only entries indexed by working couplers may be nonzero. Both upper- and lower-triangular values can be used; (i, j) and (j, i) entries are added together. If a *J* value is assigned to a coupler not present in the processor, an exception will be raised.

param_name, *value* — Keyword parameter names and values.

Return Value

answer — Dictionary with keys:

> *solutions* — List of lists. Each row represents a readout. The entry is ±1 only for active qubits. If *answer_mode* = 'histogram', the states (rows) are unique and sorted in increasing-energy order. If *answer_mode* = 'raw', all the output states are in the order that they were generated (number of rows = *num_reads*).
>
> *energies* — List containing the energies of the corresponding solutions.

num_occurrences — List indicating how many times each solution appeared. (optional, only appears if *answer_mode* is not set or set as ‘histogram’)

timing — Dictionary containing the time taken (in microseconds) at each step of the routine such as *qpu_anneal_time_per_sample*, *preprocessing_time*, etc. (Optional, only hardware solvers return the *timing* structure.)

Example 1

This example solves an Ising problem using a local solver. Specified parameters include 10 reads.

```
from dwave_sapi2.local import local_connection
from dwave_sapi2.core import solve_ising

# get a solver
solver = local_connection.get_solver("c4-sw_sample")

# solve ising problem
h = [1, -1, 1, 1, -1, 1, 1]
J = {(0, 6): -10}

params = {"num_reads": 10}
answer_1 = solve_ising(solver, h, J, **params)
print "answer_1:", answer_1

answer_2 = solve_ising(solver, h, J, num_reads=10)
print "answer_2:", answer_2

Output:

answer_1: {'energies': [-17.0], 'num_occurrences': [10], 'solutions': [[-1, 1, -1, -
 ↪1,
1, -1, -1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3]]}
answer_2: {'energies': [-17.0], 'num_occurrences': [10], 'solutions': [[-1, 1, -1, -
 ↪1,
1, -1, -1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3]]}
```

Note: The solutions you get may not be exactly the same as above.

See also:

solve_qubo | *async_solve_ising* | *async_solve_qubo* | *await_completion* | *RemoteConnection* | *local_connection*

Example 2

This example defines an anneal schedule; in this case, a reverse schedule.

The following rules apply to the set of anneal schedule points provided:

- Time t must increase for all points in the schedule.
- For forward annealing, the first point must be $(0,0)$ and the anneal fraction s must increase monotonically.
- For reverse annealing, the anneal fraction s must start and end at $s = 1$.
- In the final point, anneal fraction s must equal 1 and time t must not exceed the maximum value in the *annealing_time_range* property.
- The number of points must be ≥ 2.
  The upper bound is system-dependent; check the *max_anneal_schedule_points* property. For reverse annealing, the maximum number of points allowed is one *more* than the number given by this property.
- Additional rules that govern maximum slope vary by product; check the QPU properties document for your system.

For more information about defining an anneal schedule, see *Technical Description of the D-Wave Quantum Processing Unit*.

The following example shows how to:

- Define a reverse anneal schedule for a two-qubit problem. This schedule is shape of the waveform to be used in the reverse anneal.
- Set the initial state of both qubits to 1.
- Set the reinitialize state to `false`. This state is most likely of interest if the returned time sequences are an important part of your solution data. For this reason, use this setting with `answer_mode="raw"`.
- Send the Ising problem to the solver for solution.

It assumes that you already have a connection to a remote solver and have defined your problem.

```
from dwave_sapi2.remote import RemoteConnection
from dwave_sapi2.core import solve_ising

# define the url and a valid token
url = "http://myURL"
token = "myToken001"

solver_name = "solver_name"
```

```
# create a remote connection using url and token
remote_connection = RemoteConnection(url, token)

anneal_schedule = [[0, 1], [10, 0.5], [110, 0.5], [120, 1]]

initial_state = [3] * 2048
initial_state[6] = 1
initial_state[101] = 1

# run problem
answer1 = solve_ising(solver, h, J, initial_state=initial_state,
  anneal_schedule=anneal_schedule)
answer2 = solve_ising(solver, h, J, initial_state=initial_state,
  anneal_schedule=anneal_schedule, reinitialize_state="false", answer_mode="raw")
```

See also:

RemoteConnection

2.2.3 solve_qubo

```
from dwave_sapi2.core import solve_qubo

answer = solve_qubo(solver, Q)
answer = solve_qubo(solver, Q, param_name=value, ...)
```

Solve a QUBO problem synchronously.

Note: We call a working qubit i *active* if either Q_{ii} is non-zero, or there is another working qubit j and a working coupler between i and j for which Q_{ij} is non-zero. Non-working qubits must always be inactive.

Parameters

solver: the solver object.

Q — Dictionary of QUBO coefficients. Only entries indexed by working couplers may be nonzero. Both upper- and lower-triangular values can be used; (i, j) and (j, i) entries are added together. If a Q value is assigned to a coupler not present in the processor, an exception will be raised.

param_name, *value*: keyword parameter names and values.

Return Value

answer — Dictionary with keys:

solutions — List of lists. Each row represents a readout. The entry is 0/1 only for active qubits. If *answer_mode* = 'histogram', the states (rows) are unique and sorted in increasing-energy order. If *answer_mode* = 'raw', all the output states are in the order that they were generated (number of rows = *num_reads*).

energies — List containing the energies of the corresponding solutions.

num_occurrences — List indicating how many times each solution appeared. (optional, only appears if *answer_mode* is not set or set as 'histogram')

timing — Dictionary containing the time taken (in microseconds) at each step of the routine such as *qpu_anneal_time_per_sample*, *preprocessing_time*, etc. (Optional, only hardware solvers return the *timing* structure.)

Note: Prior to Release 2.4 of the Solver API, the timing field names were different. For more information about timing, see *Measuring Computation Time on D-Wave Systems*.

Example

This example solves a QUBO problem using a local solver.

```
from dwave_sapi2.local import local_connection
from dwave_sapi2.core import solve_qubo

# get a solver
solver = local_connection.get_solver("c4-sw_sample")

# solve qubo problem
Q = {(0, 5): -10}

params = {"num_reads": 10}
answer_1 = solve_qubo(solver, Q, **params)
print "answer_1:", answer_1

answer_2 = solve_qubo(solver, Q, num_reads=10)
print "answer_2:", answer_2

Output:

answer_1: {'energies': [-10.0], 'num_occurrences': [10], 'solutions': [[1, 3, 3, 3,␣
 ↪3,
1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3]]}
answer_2: {'energies': [-10.0], 'num_occurrences': [10], 'solutions': [[1, 3, 3, 3,␣
 ↪3,
1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
↪3,
3, 3, 3, 3, 3, 3, 3]]}
```

Note: The solutions you get may not be exactly the same as above.

See also:

solve_ising | *async_solve_ising* | *async_solve_qubo* | *await_completion* | *RemoteConnection* | *local_connection*

2.2.4 async_solve_ising

```
from dwave_sapi2.core import async_solve_ising

submitted_problem = async_solve_ising(solver, h, J)
submitted_problem = async_solve_ising(solver, h, J, param_name=value, ...)
```

Solve an Ising problem asynchronously.

Note: We call a working qubit i *active* if either h_i is non-zero, or there is another working qubit j and a working coupler between i and j for which J_{ij} is non-zero. Non-working qubits must always be inactive.

When submitting a large number of problems, it can often take a long time to solve all the problems. *async_solve_ising* lets you submit Ising problems and continue working on other tasks.

Note: When solver is a local solver, the submitted_problem.done() method will return True immediately, and the problem is actually solved when the answer is requested when using submitted_problem.result() method.

Parameters

solver — Solver object.

h — List or tuple of the linear Ising coefficients. The *h* value of a non-working qubit must be zero or an exception will be raised. Inactive qubits are disabled during annealing and cannot distinguish their states.

J - Dictionary of Ising coupling coefficients. Diagonal entries must be zero. Only entries indexed by working couplers may be nonzero. Both upper- and lower-triangular values can be used; (i, j) and (j, i) entries are added together. If a J value is assigned to a coupler not present in the processor, an exception will be raised.

param_name, *value* — Keyword parameter names and values.

Return Value

submitted_problem — Asynchronously submitted problem object. It has methods *done*, *result*, and *cancel*.

- submitted_problem.done() determines whether the submitted problem has finished or not. Once the problem has been solved or has failed, it returns True, otherwise it returns False.
- submitted_problem.status() returns a dictionary containing information about the progress of the problem. For remote problems, the keys are:
  - *'state'*: describes the state of the problem as seen by the client. This key is always present. Possible values are:
    - *'SUBMITTING'*: the problem is in the process of being submitted.
    - *'SUBMITTED'*: the problem has been submitted but is not done yet.
    - *'DONE'*: the problem is done, meaning either that it was successfully completed or that solving failed.
    - *'FAILED'*: an error occurred while determining the actual state. This condition does not imply anything about whether the problem itself has succeeded or not.
    - *'RETRYING'*: similar to *'FAILED'* but the client is actively trying to fix the problem.
  - *'remote_status'*: describes the state of the problem as reported by the server. This key is always present. Possible values are:
    - *'PENDING'*: the problem is waiting in a queue.
    - *'IN_PROGRESS'*: processing has started for the problem.
    - *'COMPLETED'*: the problem has completed successfully.
    - *'FAILED'*: an error occurred while solving the problem.
    - *'CANCELED'*: the problem was cancelled by the user.
    - *'UNKNOWN'*: the client has not yet received information from the server (i.e. state has not reached *'SUBMITTED'*).
  - *'problem_id'*: the remote problem ID for this problem. This key is always present. It is an empty string until the *'SUBMITTED'* state is reached.
  - *'last_good_state'*: last good value of the state key. The *'SUBMITTING'*, *'SUBMITTED'*, and *'DONE'* states are "good," while *'FAILED'*, and *'RETRYING'* are "bad." This key is present only when the state key is bad and its value is the last good value of the state key.
  - *'error_type'*: machine-readable error category. This key is present only when either:

        * state is *'FAILED'*, or
        * state is *'RETRYING'*, or
        * state is *'DONE'* and remote_status is not *'COMPLETED'*.

      Possible values are:

        * *'NETWORK'*: network communication failed.
        * *'PROTOCOL'*: client couldn't understand a response from the server. Possible causes include communication errors between intermediate servers, client or server bugs.
        * *'AUTH'*: authentication failed.
        * *'SOLVE'*: solving failed.
        * *'MEMORY'*: out of memory.
        * *'INTERNAL'*: catch-all value for unexpected errors.

    - *'error_message'*: human-readable error message.
    - *'time_received'*: time at which the server received the problem. Present once the problem has been received.
    - *'time_solved'*: time at which the problem was solved. Present once the problem has been solved.

  For local problems, the status is always *{'state': 'DONE'}*.

- submitted_problem.result() can be called to retrieve the solution of the problem.
- submitted_problem.retry() attempts to retry the problem if it has failed for non-solving reasons (e.g. network or authentication errors). It is only useful when *submitted_problem.status['state'] == 'FAILED'*.
- submitted_problem.cancel() cancels a submitted problem. Cancellation is not guaranteed; problems may still complete successfully.

Example

This example solves an Ising problem asynchronously.

```
from dwave_sapi2.local import local_connection
from dwave_sapi2.core import async_solve_ising, await_completion

# get a solver
solver = local_connection.get_solver("c4-sw_sample")

h = [1, -1, 1, 1, -1, 1, 1]
J = {(0, 6): -10}

submitted_problem = async_solve_ising(solver, h, J, num_reads=10)

# Wait until solved
await_completion([submitted_problem], 1, float('inf'))

# display result
```

```
print "answer:", submitted_problem.result()

Output:

answer:  {'energies': [-17.0], 'num_occurrences': [10], 'solutions': [[-1, 1, -1, -
 ↪1,
1, -1, -1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3]]}
```

Note: The solutions you get may not be exactly the same as above.

See also:

solve_ising | *solve_qubo* | *async_solve_qubo* | *await_completion* | *RemoteConnection* | *local_connection*

2.2.5 async_solve_qubo

```
from dwave_sapi2.core import async_solve_qubo, await_completion

submitted_problem = async_solve_qubo(solver, Q)
submitted_problem = async_solve_qubo(solver, Q, param_name=value, ...)
```

Solve a QUBO problem asynchronously.

Note: We call a working qubit i *active* if either Q_{ii} is non-zero, or there is another working qubit j and a working coupler between i and j for which Q_{ij} is non-zero. Non-working qubits must always be inactive.

When submitting a large number of problems, it can often take a long time to solve all the problems. *async_solve_qubo* lets you submit QUBO problems and continue working on other tasks.

Note: When solver is a local solver, the submitted_problem.done() method will return True immediately, and the problem is actually solved when the answer is requested when using submitted_problem.result() method.

Parameters

solver — Solver object.

Q — Dictionary of QUBO coefficients. Only entries indexed by working couplers may be nonzero. Both upper- and lower-triangular values can be used; (*i*, *j*) and (*j*, *i*) entries are added together. If a *Q* value is assigned to a coupler not present in the processor, an exception will be raised.

param_name, *value* — Keyword parameter names and values.

Return Value

submitted_problem — Asynchronously submitted problem object.

See also:

async_solve_ising

Example

This example solves a QUBO problem asynchronously.

```
from dwave_sapi2.local import local_connection
from dwave_sapi2.core import async_solve_qubo

# get a solver
solver = local_connection.get_solver("c4-sw_sample")

Q = {(0, 5): -10}

submitted_problem = async_solve_qubo(solver, Q, num_reads=10)

# Wait until solved
await_completion([submitted_problem], 1, float('inf'))

# display result
print "answer:", submitted_problem.result()

Output:

answer: {'energies': [-10.0], 'num_occurrences': [10], 'solutions': [[1, 3, 3, 3, 3,
 ↪ 1,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3]]}
```

Note: The solutions you get may not be exactly the same as above.

See also:

solve_ising | *solve_qubo* | *async_solve_ising* | *await_completion* | *RemoteConnection* | *local_connection*

2.2.6 await_completion

```
from dwave_sapi2.core import await_completion

done = await_completion(submitted_problems, min_done, timeout)
```

Waits for problems to complete.

Parameters

submitted_problems — List of asynchronous problems (returned by *async_solve_ising* or *async_solve_qubo*).

min_done — Minimum number of problems that must be completed before returning (without timeout).

timeout — Maximum time to wait (in seconds).

Return Value

done — True if returning because enough problems completed, False if returning because of timeout.

Example

```
from dwave_sapi2.local import local_connection
from dwave_sapi2.core import async_solve_ising, await_completion

# get a solver
solver = local_connection.get_solver("c4-sw_sample")

h = [1, -1, 1, 1, -1, 1, 1]
J = {(0, 6): -10}

p1 = async_solve_ising(solver, h, J, num_reads=10)
p2 = async_solve_ising(solver, h, J, num_reads=20)

min_done = 2
timeout = 1.0
done = await_completion([p1, p2], min_done, timeout)
```

```
if done:
    print "answer_1:", p1.result()
    print "answer_2:", p2.result()

Output:

answer_1: {'energies': [-17.0], 'num_occurrences': [10], 'solutions': [[-1, 1, -1, -
 ↪1,
1, -1, -1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3]]}
answer_2: {'energies': [-17.0], 'num_occurrences': [20], 'solutions': [[-1, 1, -1, -
 ↪1,
1, -1, -1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,␣
 ↪3,
3, 3, 3, 3, 3, 3, 3, 3, 3]]}
```

Note: The solutions you get may not be exactly the same as above.

See also:

solve_ising | *solve_qubo* | *async_solve_ising* | *async_solve_qubo* | *RemoteConnection* | *local_connection*

CHAPTER 3

POSTPROCESSING

3.1 Postprocessing Overview

3.1.1 Available Methods

The D-Wave system enables users to run postprocessing optimization and sampling algorithms on solutions obtained through the quantum processing unit (QPU). Postprocessing provides local improvements to these solutions with minimal overhead.

When submitting a problem to the QPU, users choose from:

- No postprocessing (default for hardware solvers)
- Optimization postprocessing
- Sampling postprocessing (default for VFYC solvers)

For optimization problems, the goal is to find the state vector with the lowest energy. For sampling problems, the goal is to produce samples from a specific probability distribution. In both cases, a logical graph structure is defined and embedded into the QPU's Chimera topology. Postprocessing methods are applied to solutions defined on this logical graph structure.

For more information about the postprocessing methods available, see *Postprocessing Methods on D-Wave Systems*.

3.1.2 Trade-offs

Mapping most real-world problems onto a Chimera graph requires increasing the connectivity on the QPU, which is done by introducing so-called *chains*: groups of qubits that are strongly coupled together to represent a single problem variable. In a perfect world, where infinite precision on h and J values is available, one could force the qubits on the chain to take the same spin by assigning a large enough (problem-dependent) coupling strength between the qubits on the chain. In reality, however, chains can (and often do) break.

When a chain breaks, the corresponding sample could either be discarded or mapped to a close feasible state with no breakage. The first option wastes samples; the second introduces some overhead to postprocess the samples. D-Wave's postprocessing algorithms perform majority voting on the chain to map the broken chains to their closest[1] feasible

[1] Closest in terms of Hamming distance; i.e., the fewest number of positions at which the corresponding values

state.

Moreover, for optimization problems, where we are looking for the global optima or at least good local optima, states that are not locally optimum are not interesting before further postprocessings. The simplest postprocessing to map a non-locally optimum state to a candidate solution is to run a local search to find a close local optimum state. In practice, some of the samples returned by the QPU are not locally optimum and, like broken chains, leave us with two options on to treat them: discard them, or run a local search to fix them. Again, the trade-off is between spending time sampling new states or spending time running local search on such samples.

3.2 Parameters

Use the following parameters to control postprocessing:

- *beta*
- *chains*
- *postprocess*

See also:

RemoteConnection

are different.

CHAPTER 4

SIMPLIFYING OPTIMIZATION PROBLEMS

For some QUBO/Ising problems we can infer (in polynomial time) the value that certain variables take in the lowest energy states. If certain variables can be set, this reduces the size of the problem that needs to be sent to the processor, and may help alleviate precision problems. For certain problems (submodular problems where all $J_{ij} < 0$ or $Q_{ij} < 0$) all variables may be inferred. For other problems, no variables may be inferred.

4.1 fix_variables

```
from dwave_sapi2.fix_variables import fix_variables

result = fix_variables(q, method="optimized")
```

Fix variables for solving Ising problems.

4.1.1 Parameters

q: QUBO problem dictionary. Pairs of variables indices (use repeated indices for linear terms) map to coefficient values.

method: fix variables method, it determines the algorithm used to infer values. (must be 'optimized' or 'standard', default = 'optimized')

- 'optimized': uses roof-duality & strongly connected components (default value)
- 'standard': uses roof-duality only

fix_variables uses maximum flow in the implication network to correctly fix variables (that is, one can find an assignment for the other variables that attains the optimal value). The variables that roof duality fixes will take the same values in all optimal solutions.

Using strongly connected components can fix more variables, but in some optimal solutions these variables may take different values.

In summary:

- All the variables fixed by *method* = 'standard' will also be fixed by *method* = 'optimized' (reverse is not true)
- All the variables fixed by *method* = 'standard' will take the same value in every optimal solution

- There exists at least one optimal solution that has the fixed values as given by '*method*' = 'optimized'

Thus, *method* = 'standard' is a subset of *method* = 'optimized' as any variable that is fixed by *method* = 'standard' will also be fixed by *method* = 'optimized' and additionally, *method* = 'optimized' may fix some variables that *method* = 'standard' could not. For this reason, *method* = 'optimized' takes longer than *method* = 'standard'.

4.1.2 Return Value

result: a dictionary that has keys: "fixed_variables", "new_Q", "offset".

"fixed_variables": dictionary mapping variable indices to their fixed values.

"new_Q": simplified problem in the same format as input *q*. Fixed variable indices do not appear.

"offset": energy offset. Add this value to the energy of any state for the *new_Q* problem to get the energy of the same state with fixed variables for the original *q* problem.

4.1.3 Example

```
from dwave_sapi2.fix_variables import fix_variables

q = {(0, 5): 100}
result = fix_variables(q)
print result

Output:

{'offset': -100.0, 'new_Q': {}}, 'fixed_variables': {0: 0, 5: 0}}
```

CHAPTER 5

SOLVING NON-CHIMERA STRUCTURED PROBLEMS

In this section, we consider Ising/QUBO problems with variable interactions that do not match those of the current working Chimera graph. It is assumed that:

1. These problems have fewer variables than the current working Chimera graph, and
2. The user provides an *embedding* of the problem variables into the current working Chimera graph.

Suppose that $G = (V, E)$ is the current working Chimera graph, where V is the set of vertices (i.e., working qubits) and E is the set edges (i.e., working couplers). Consider the Ising problem P defined by $min_x(h' \times x + x' \times J \times x)$, where the dimension of x (i.e., the number of variables) is t. We assume that $t \leq |V|$. Our goal is to define a problem in the current working Chimera graph whose solution will result in a solution to the original problem P.

An *embedding* of the Ising problem P into graph G is a mapping that assigns to each variable x_i a subset of nodes $T(x_i) \subset V$ of G such that:

- The subsets $T(x_i)$ are disjoint, that is, $T(x_i) \cap T(x_j) = \emptyset$ for $i \neq j$,
- For each i the subset $T(x_i)$ is connected (usually a path),
- If there is an interaction between x_i and x_j, that is, $J_{ij} \neq 0$, then there is at least one edge $e \in E$ (i.e., a working coupler) between the subsets $T(x_i)$ and $T(x_j)$.

An embedding *splits* the original coefficients h and J and create new coefficients h_0 and J_0 as follows:

If, for variable x_i, the set $T(x_i)$ contains more than one node, the weight h_i is distributed evenly across the qubits in $T(x_i)$.

The next step is to solve the problem defined by h_0 and a J_0 in the Chimera graph. However, we must also make sure that for each i, all the qubits in $T(x_i)$ are aligned (i.e., all the variables in $T(x_i)$ take the same value). We enforce these constraints by penalizing the qubit configurations that violate them using a penalty term JF_m (called *ferromagnetic coupling*). Essentially we use the following. Suppose that qubits q_a and q_b have to take the same value and have a common coupler. We can accomplish this by setting $(JF_m)_{ab}$ to be -1 and scaling up JF_m as necessary.

Finally, to solve the original problem P, we solve $\min_y(h_0' \times y + y' \times J_0 \times y + \lambda\, y' \times JF_m \times y)$ in the hardware graph, where $\lambda > 0$ needs to be adjusted to make sure that all the qubit constraints are satisfied.

5.1 find_embedding

```
from dwave_sapi2.embedding import find_embedding

embeddings = find_embedding(S, A)
embeddings = find_embedding(S, A, param_name=value, ...)
```

Attempts to find an embedding of a QUBO/Ising problem in a graph. This function is entirely heuristic: failure to return an embedding does not prove that no embedding exists.

It can be interrupted by Ctrl-C, will return the best embedding found so far.

5.1.1 Parameters

S: edge structures of a problem. The embedder only cares about the edge structure (i.e. which variables have nontrivial interactions), not the coefficient values. (must be an iterable object containing pairs of integers representing edges)

A: adjacency matrix of the graph (as returned by *get_chimera_adjacency* or *get_hardware_adjacency*). (must be an iterable object containing pairs of integers representing edges)

param_name, *value*: keyword parameter names and values.

find_embedding can take the following keyword parameters when it is solving the embedding problem:

fast_embedding: tries to get an embedding quickly, without worrying about chain length. (must be a boolean, default = False)

max_no_improvement: number of rounds of the algorithm to try from the current solution with no improvement. Each round consists of an attempt to find an embedding for each variable of S such that it is adjacent to all its neighbours. (must be an integer >= 0, default = 10)

random_seed: seed for random number generator that *find_embedding* uses. (must be an integer >= 0, default is randomly set)

timeout: algorithm gives up after timeout seconds. (must be a number >= 0, default is approximately 1000.0 seconds)

tries: the algorithm stops after this number of restart attempts. (must be an integer >= 0, default = 10)

Note: The algorithm stops when either of *timeout* or *tries* is reached, whichever comes first.

verbose: control the output information. (must be an integer [0 1], default = 0) When verbose is 1, the output information will be like:

component ..., try ...:

max overfill = ..., num max overfills = ...

Embedding found. Minimizing chains...

max chain size = ..., num max chains = ..., qubits used = ...

Detailed explanation of the output information:

- "component": process ith (0-based) component, the algorithm tries to embed larger strongly connected components first, then smaller components
- "try": jth (0-based) try
- "max overfill": largest number of variables represented in a qubit
- "num max overfills": the number of qubits that has max overfill
- "max chain size": largest number of qubits representing a single variable
- "num max chains": the number of variables that has max chain size
- "qubits used": the total number of qubits used to represent variables

The acceptable range and the default value of each field are given in the table below:

Field	Range	Default value
fast_embedding	True or False	False
max_no_improvement	>= 0	10
random_seed	>= 0	randomly set
timeout	>= 0.0	1000.0
tries	>= 0	10
verbose	[0 1]	0

5.1.2 Return Value

embeddings: embeddings[i] is the list of qubits representing logical variable i. embeddings can be used in *embed_problem*. If the algorithm fails, the output is an empty list.

5.1.3 Example

This example find the embedding for a size of 30 in a full chimera graph.

```
from dwave_sapi2.util import get_chimera_adjacency
from dwave_sapi2.embedding import find_embedding

S_size = 30
S = {}
for i in range(S_size):
    for j in range(S_size):
        S[(i, j)] = 1

M = 8
N = M
L = 4
A = get_chimera_adjacency(M, N, L)

embeddings = find_embedding(S, A, verbose=1)

Output:
```

```
component 0, try 0:
max overfill = 2, num max overfills = 17
max overfill = 1, num max overfills = 355
max overfill = 1, num max overfills = 366
max overfill = 1, num max overfills = 366
Embedding found. Minimizing chains...
max chain size = 15, num max chains = 1, qubits used = 366
max chain size = 15, num max chains = 1, qubits used = 366
max chain size = 15, num max chains = 1, qubits used = 366
max chain size = 15, num max chains = 1, qubits used = 366
```

This example find the embedding for a size of 17 in one of the solver chimera graph.

```
from dwave_sapi2.local import local_connection
from dwave_sapi2.util import get_hardware_adjacency
from dwave_sapi2.embedding import find_embedding

S_size = 17
S = {}
for i in range(S_size):
    for j in range(S_size):
        S[(i, j)] = 1

# get a solver
solver = local_connection.get_solver("c4-sw_sample")
A = get_hardware_adjacency(solver)

embeddings = find_embedding(S, A, verbose=1)

Output:

component 0, try 0:
max overfill = 2, num max overfills = 19
max overfill = 1, num max overfills = 111
max overfill = 1, num max overfills = 115
max overfill = 1, num max overfills = 115
Embedding found. Minimizing chains...
max chain size = 9, num max chains = 1, qubits used = 115
max chain size = 8, num max chains = 7, qubits used = 115
max chain size = 8, num max chains = 7, qubits used = 115
max chain size = 8, num max chains = 6, qubits used = 116
max chain size = 8, num max chains = 6, qubits used = 116
```

Note: The solutions you get may not be exactly the same as above.

See also:

embed_problem | *unembed_answer*

5.2 embed_problem

```
from dwave_sapi2.embedding import embed_problem

[h0, j0, jc, embeddings] = embed_problem(h, j, embeddings, adj, clean, smear, h_
↪range, j_range)
```

Embed an Ising problem into a graph.

5.2.1 Parameters

h: list of linear Ising coefficients.

j: quadratic Ising coefficients, a dictionary mapping variable pairs to values.

embeddings: a list describing the embedding. Element k is a list of the output variables to which problem variable k is mapped. Elements (of the list) must be mutually disjoint.

adj: a set describing the output adjacency structure. Output variables i and j are adjacent if (i, j) is in adj.

clean: logical value indicating whether or not to "clean" the embedding. Cleaning means iteratively removing any physical variables from each chain that are:

- adjacent to a single variable in the same chain
- not adjacent to any variables in other chains

(default=False)

smear: logical value indicating whether or not to "smear" the embedding. Smearing attempts to increase chain sizes so that the scale of h values (relative to h_range) does not exceed the scale of J values (relative to j_range). Smearing is performed after cleaning, so enabled both is potentially useful. (default=False)

h_range, *j_range*: valid ranges of h and J values, respectively. Each is a two-element list of the form [min, max] (just like *h_range* and *j_range* solver properties, which is the likely source for these parameters). These values are only used when *smear* is true. There is no actual enforcement of h or J values in this function. Both have default value [-1, 1].

5.2.2 Return Value

h0: embedded linear Ising coefficients.

j0: embedded quadratic Ising coefficients.

jc: strong output variable couplings. jc[i, j] == -1 precisely when i < j (jc is upper triangular), (i, j) in A, and i and j correspond to the same problem variable.

embeddings: possibly modified embeddings. This will differ from input embeddings only if clean or smear is True.

The reason that jc is returned in addition to j0 is that Ising problems have no constraints. A problem variable may map to multiple output variable, so there is a real possibility that the optimal solution of the output problem has inconsistent values for those output variables. In this case, h0 and j0 can be scaled down relative to jc to produce a new Ising problem (h1, j1):

h1 = s * h0; j1 = s * j0 + jc;

for some 0 < s < 1. Finding the right value for s is hard. It's easy to find a tiny value that will make every consistent solution better than any inconsistent solution, but this may raise the precision requirements beyond the capabilities of the quantum processor. Also, you may need only the best one or two consistent solutions, allowing a larger value of s. Bottom line: try s = 1 first and decrease until desired consistency is achieved.

5.2.3 Example

```
from dwave_sapi2.util import get_chimera_adjacency
from dwave_sapi2.embedding import embed_problem

h = [-200.0, 300, 600, -600]
j = {(0, 1): 200.0, (0, 2): -700.0, (0, 3): 600.0, (1, 2): -600.0, (1, 3): -200.0,
     (2, 3): 300.0}
embeddings = [[48, 52], [49, 53], [50, 54], [51, 55]]
A = get_chimera_adjacency(4, 4, 4)
(h0, j0, jc, new_emb) = embed_problem(h, j, embeddings, A)
print h0
print j0
print jc

Output:

[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 -0.2857142857142857, 0.42857142857142855, 0.8571428571428571, -0.8571428571428571,
 -0.2857142857142857, 0.42857142857142855, 0.8571428571428571, -0.8571428571428571,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

{(50, 55): 0.42857142857142855, (49, 52): 0.2857142857142857, (48, 54): -1.0,
 (49, 54): -0.8571428571428571, (48, 55): 0.8571428571428571,
 (51, 54): 0.42857142857142855, (49, 55): -0.2857142857142857,
 (48, 53): 0.2857142857142857, (50, 53): -0.8571428571428571,
 (51, 52): 0.8571428571428571, (50, 52): -1.0, (51, 53): -0.2857142857142857}

 {(49, 53): -1.0, (51, 55): -1.0, (50, 54): -1.0, (48, 52): -1.0}
```

Note: The solutions you get may not be exactly the same as above.

See also:

find_embedding | *unembed_answer*

5.3 unembed_answer

```
from dwave_sapi2.embedding import unembed_answer

result = unembed_answer(solutions, embeddings, broken_chains=None, h=None, j=None)
```

unembed answer from solver to reconstruct answer for the original problem.

5.3.1 Parameters

solutions: solution bits from solving embedded problem (e.g. "solutions" value of answer returned by *solve_ising*).

embeddings: embeddings used to embed.

broken_chains: parameter that controls how to deal with broken chains, can be:

'minimize_energy': repair chains one at a time, minimizing energy at each step (default)

'vote': do a majority vote for broken chains, break the tie randomly.

'discard': discard solution that has broken chains.

'weighted_random': choose +1 or -1 with probability equal to the fraction of +1 or -1 values in the chain

h, *j*: original (pre-embedding) problem. Required when broken_chains='minimize_energy', ignored otherwise.

5.3.2 Return Value

new_answer: the answer for the original problem.

5.3.3 Example

```
from dwave_sapi2.util import get_hardware_adjacency
from dwave_sapi2.embedding import embed_problem, unembed_answer
from dwave_sapi2.core import solve_ising
from dwave_sapi2.local import local_connection

solver = local_connection.get_solver("c4-sw_sample")
h = [-200.0, 300, 600, -600]
j = {(0, 1): 200.0, (0, 2): -700.0, (0, 3): 600.0, (1, 2): -600.0, (1, 3): -200.0,
     (2, 3): 300.0}
embeddings = [[48, 52], [49, 53], [50, 54], [51, 55]]
A = get_hardware_adjacency(solver)
```

```
(h0, j0, jc, new_emb) = embed_problem(h, j, embeddings, A)
emb_j = j0.copy()
emb_j.update(jc)
result = solve_ising(solver, h0, emb_j, num_reads=6)
new_answer = unembed_answer(result['solutions'], new_emb, 'minimize_energy', h, j)
print new_answer

Output:

[[-1, -1, -1, 1]]
```

Note: The solutions you get may not be exactly the same as above.

See also:

find_embedding | *solve_ising* | *embed_problem*

CHAPTER 6

REDUCING ORDER INTERACTION

Many problems involve interactions between groups of 3 or more variables, and thus, cannot be directly modeled within the Ising/QUBO model due to limitations of those models to pairwise interactions. Functions having higher-order interactions are conveniently represented as a weighted sum of products of literals, for example, $f(x_0, x_1, x_2, x_3) = 5x_0x_1 - 3x_1x_2x_3$. Each product of variables is called a term. A function expressed in this manner can be stored in the computer as a list of terms where each term is itself a list of the variables in the product. The weighting of each term can be stored as a separate list, e.g. $[5, -3]$. For example, the terms in the above function are t_0 and t_1 where $t_0 = 0, 1$ and $t_1 = 1, 2, 3$ indicate that x_0 and x_1 appear in term t_0, and x_1, x_2, x_3 appear in term t_1. The function f cannot be represented in hardware because of the third-order interactions in term t_1. However, by introducing a new variable $y = x_1x_2$ we can write f as $5x_0x_1 - 3yx_3$. Fortunately, the constraint $y = x_1x_2$ can be represented using a penalty function $P(x_1, x_2; y)$ having only pairwise interactions so that $\tilde{f}(x_0, x_1, x_2, x_3, y) = 5x_0x_1 - 3yx_3 + P(x_1, x_2; y)$ when minimized over y represents f. The routines in this chapter facilitate these kinds of reductions to QUBOs.

6.1 reduce_degree

```
from dwave_sapi2.util import reduce_degree

(new_terms, vars_rep) = reduce_degree(terms)
```

Reduce the degree of a set of objectives specified by terms to have maximum two degrees via the introduction of ancillary variables.

6.1.1 Parameters

terms: a list, element in the list represents each term's variables in the expression, the index in terms must be a non-negative integer.

6.1.2 Return Value

new_terms: a list of the new pairwise and lower terms written as a function of the extended variable set.

vars_rep: a $n \times 3$ list where n is the number of new variables needed. Each row of *vars_rep* indicated as [v, x, y] introduces a new variable v as the product of previously introduced variables x and y.

6.1.3 Example

This example reduces the greater than pairwise terms to pairwise by introducing new variables.

```
from dwave_sapi2.util import reduce_degree

# f =   x0 * x2 * x3 * x4 * x5 * x8
#     + x3 * x6 * x8
#     + x1 * x6 * x7 * x8
#     + x0 * x2 * x3 * x5 * x6 * x7
#     + x1 * x3 * x6
#     + x1 * x6 * x8 * x10 * x12

terms = [[0, 2, 3, 4, 5, 8], [3, 6, 8], [1, 6, 7, 8], [0, 2, 3, 5, 6, 7], [1, 3, 6],
         [1, 6, 8, 10, 12]]
(new_terms, vars_rep) = reduce_degree(terms)
print "new_terms:", new_terms
print "vars_rep:", vars_rep

Output:

new_terms: [[17, 18], [8, 21], [7, 14], [7, 19], [3, 13], [14, 20]]
vars_rep: [[13, 1, 6], [14, 8, 13], [15, 2, 5], [16, 3, 15], [17, 0, 16], [18, 4,␣
 ↪8],
           [19, 6, 17], [20, 10, 12], [21, 3, 6]]
```

The returned *vars_rep* indicates that the routine has introduced new variables numbered 13 to 21 and for example, variable 13 is the product of variable x_1 and x_6. *new_terms* are the terms in the new variables. No term in new terms will have more than pair-wise interactions. For example,

$$new\ term_0 = Term_{17} \times Term_{18}$$
$$Term_{17} = x_0 \times Term_{16}$$
$$Term_{18} = x_4 x_8$$
$$Term_{16} = x_3 \times Term_{15}$$
$$Term_{15} = x_2 x_5;\text{ therefore}$$
$$Term_{16} = x_2 x_3 x_5;\text{ therefore}$$
$$Term_{17} = x_0 x_2 x_3 x_5;\text{ therefore}$$
$$new\ term_0 = x_0 x_2 x_3 x_4 x_5 x_8$$

Therefore, as can be seen from above, the non-pair-wise term in the original problem has been replaced with a new pair-wise term comprising the product of $Term_{17}$ and $Term_{18}$.

See also:

make_quadratic

6.2 make_quadratic

```
from dwave_sapi2.util import make_quadratic

(Q, new_terms, vars_rep) = make_quadratic(f, penalty_weight=None)
```

If an objective function f is represented explicitly as a vector of numbers (e.g. [f_{000}, f_{001}, f_{010}, f_{011}, f_{100}, ..., f_{111}]), we may not know the representation as sums of terms. *make_quadratic* function performs similar optimization as in *reduce_degree* when the input is a vector of numbers instead of an array of terms. For example, if we define a problem with 3 variables x_0, x_1 and x_2, then f_{000} represents the value of f with $x_0 = 0$, $x_1 = 0$ and $x_2 = 0$. Similarly, the second term of f (f_{001}) will have $x_0 = 1$, $x_1 = 0$ and $x_2 = 0$. Then, for a problem defined as $f = 8x_0x_2 - x_0x_1x_2$:

f_{000}	f_{001}	f_{010}	f_{011}	f_{100}	f_{101}	f_{110}	f_{111}
0	0	0	0	0	8	0	7

The function looks at the length of the problem submitted and determines if the length is equal to a power of 2. If it is so, it then acts similar to *reduce_degree* by replacing the variables with equivalent pair-wise interactions.

make_quadratic takes an explicit function indicated by the f, and generates an equivalent QUBO representation specified by the Q.

6.2.1 Parameters

f: a function defined over binary variables represented as an array stored in decimal order.

penalty_weight: the strength of the penalty used to define the product constraints on the new ancillary variables. If not provided, it will use the default value, the default value is usually sufficiently large, but may be larger than necessary.

6.2.2 Return Value

Q: quadratic coefficients.

new_terms: the terms in the QUBO arising from quadraticization of the interactions present in f.

variables_rep: the definition of the new ancillary variables.

6.2.3 Example

This example generates an equivalent QUBO representation of the input specified.

```
from dwave_sapi2.util import make_quadratic

# A function f defined over binary variables represented as an
# array stored in decimal order
#
#         f(x3, x2, x1, x0) =    a
#                              + b * x0
#                              + c * x1
#                              + d * x2
#                              + e * x3
#                              + g * x0 * x1
#                              + h * x0 * x2
#                              + i * x0 * x3
#                              + j * x1 * x2
#                              + k * x1 * x3
#                              + l * x2 * x3
#                              + m * x0 * x1 * x2
#                              + n * x0 * x1 * x3
#                              + o * x0 * x2 * x3
#                              + p * x1 * x2 * x3
#                              + q * x0 * x1 * x2 * x3
#
#         f(0000) means when x3 = 0, x2 = 0, x1 = 0, x0 = 0, so f(0000) = a
#         f(0001) means when x3 = 0, x2 = 0, x1 = 0, x0 = 1, so f(0001) = a + b
#         f(0010) means when x3 = 0, x2 = 0, x1 = 1, x0 = 0, so f(0010) = a + c
#         etc.

f = [0, -1, 2, 1, 4, -1, 0, 0, -1, -3, 0, -1, 0, 3, 2, 2]

(Q, new_terms, vars_rep) = make_quadratic(f)

print "Q:", Q
print "new_terms:", new_terms
print "vars_rep:", vars_rep

Output:

Q: {(1, 3): -0.5, (3, 0): -0.5, (5, 4): -4.5, (2, 1): -3.0, (5, 1): 3.5, (0, 3): -0.
↪5,
    (2, 5): -45.0, (4, 0): -45.0, (1, 2): -3.0, (5, 5): 135.0, (4, 4): 135.0, (1,␣
↪5): 3.5,
    (5, 0): 4.5, (0, 4): -45.0, (3, 3): -1.0, (3, 5): -45.0, (4, 1): -45.0, (1, 1):␣
↪2.0,
    (3, 2): 21.0, (0, 0): -1.0, (4, 5): -4.5, (2, 2): 4.0, (1, 4): -45.0, (0, 5): 4.
↪5,
    (4, 2): 2.5, (1, 0): 22.5, (5, 3): -45.0, (0, 1): 22.5, (5, 2): -45.0, (3, 1): -
↪0.5,
    (0, 2): -2.0, (2, 0): -2.0, (4, 3): 0.5, (2, 3): 21.0, (3, 4): 0.5, (2, 4): 2.5}
new_terms: [[0], [1], [2], [0, 2], [1, 2], [2, 4], [3], [0, 3], [1, 3], [3, 4], [2,␣
↪3],
            [0, 5], [1, 5], [4, 5]]
vars_rep: [[4, 0, 1], [5, 2, 3]]
```

Note: The length of f has to be a power of 2. i.e., len(f) = 2^m where m is the number of variables in the given problem.

See also:

reduce_degree

CHAPTER 7

QSAGE

This chapter describes the QSage algorithm and how it can help achieve better results for problems submitted to the D-Wave QPU. At the end of the chapter is a description of the *solve_qsage* Python function.

7.1 Motivation

The restricted connectivity between qubits limits the ability to directly solve arbitrarily structured problems. To solve a problem directly in the QPU if there is an interaction between problem variables s_1 and s_2, then a physical connection (edge) must exist between the qubits representing the values of these variables. For most problems, the interactions between variables do not match the qubit connectivity. This limitation can be circumvented using embedding. However, this solution requires you to find an embedding or mapping of problem variables to qubits. Finding such embeddings is itself a hard optimization problem.

Moreover, the native D-Wave QPU is limited to the minimization of Ising or QUBO objective functions:

A graph with edge set E defines the allowed interactions between variables. This functional form is restricting in two ways. First, your problem may involve interactions between more than pairs of variables. Though this problem can be addressed using the methods of reducing higher-order interactions, this costs qubits, and requires additional programming. Second, the function you want to minimize may not have a mathematical description. The objective function you want to minimize may be represented as a computer program, which when input with a bit string, returns a number representing the value of the objective function. The optimization of problems not expressible mathematically in terms of $\boldsymbol{h}$ and $\boldsymbol{J}$ is not possible directly on the D-Wave 2X QPU.

In this chapter, we show how all of these problems can be addressed using a method called *QSage*, which relies on quantum annealing to heuristically minimize (*i.e.* minimize without a guarantee of optimality) arbitrary objective functions. The method can be applied to any objective function. As a user, all you need to do is supply an objective function that returns the objective value of any configuration $\boldsymbol{s}$. We will give an overview of how the QSage method works and provide a description of the parameters needed by the function. In cases where the objective function that you wish to minimize is computationally expensive we also show how the QSage method allows for parallelization across function evaluations.

It is important to stress that even though the QSage method can be applied to any objective function, we cannot expect good results on all problems. Due to the generality of the method, an optimization expert who studies the details of a particular problem is likely to develop a better, specifically-tailored optimization approach. Our goal is to provide a method that will typically yield good results, and that only requires the user to code an objective function evaluator. As you define your objective functions to be solved by the QSage method, take note that there are many ways to do this, and that some objectives may be easier to solve using this method than others. While there is as much art as science in crafting objective functions, we offer one important guiding principle:

Hint: As much as possible keep your objective function smooth so that small changes in input cause small changes in objective value.

Smooth objectives result in fewer local optima and make the problem more solvable by quantum annealing. If you can think of multiple ways to represent your optimization task it may be worthwhile coding many of them.

In the remainder of this chapter we provide an overview describing how the QSage optimizer works and its interface.

7.2 Algorithm Overview

The QSage method attempts to minimize an objective function $G(\boldsymbol{s})$ defined over array $\boldsymbol{s}$ of length n consisting of -1/1 or 0/1 values. We assume that the user has written code which evaluates $G(\boldsymbol{s})$, and returns a number indicating the objective value. The QSage method attempts to find the configuration $\boldsymbol{s}$ that has the lowest objective value. Note that number of variables n may be larger than, smaller than, or equal to the number of qubits available in the QPU. If n is larger than the number of qubits then the larger problem is solved by large neighbourhood local search *[Ahuja2000]* whereby random subsets of problem variables are optimized in the context of fixed values of the remaining variables. If n is smaller than the number of qubits then the extra qubits are exploited to create additional edge interactions between variables through embedding. Since QSage is unaware of any problem structure in the objective the additional edge interactions are assigned randomly, and fixed through the course of the algorithm.

At the highest level, QSage works by extending a very successful heuristic, tabu search, for discrete optimization problems. Tabu is a local search algorithm. In tabu search an initially random configuration $\boldsymbol{s}_0$ is perturbed to generate many different, but similar, variants. Amongst these variants we hope to discover a new configuration having an objective value lower than $G(\boldsymbol{s}_0)$. Most commonly, the variants of a configuration $\boldsymbol{s}_0$ are obtained by flipping the sign of one of the n spin values. In this way, we generate n variants with each variant differing from $\boldsymbol{s}_0$ in a single spin variable. Like all local search algorithms tabu search usually adopts one of the variants if it has a lower objective value. This new and improved configuration becomes the new configuration $\boldsymbol{s}_1$. We now iterate this procedure and look at all the variants of $\boldsymbol{s}_1$ in the hopes of finding yet another improvement.

As the algorithm runs we accumulate improvements, but eventually this iterative improvement becomes stuck after t steps in a configuration $\boldsymbol{s}_t$ that has lower objective than all of its variants. Such a configuration is called a *local minimum* as $\boldsymbol{s}_t$ is lower than all the locally perturbed variants. To make further progress, configurations that worsen the objective value must be adopted. However, there is no point in moving to a new configuration $\boldsymbol{s}_{t+1}$ only at the next time step $t+2$ to move right back to the previously considered configuration $\boldsymbol{s}_t$.

Tabu search adopts a simple, but effective, short-term memory strategy to prevent such behaviour. Every time a bit is flipped when moving from configuration $\boldsymbol{s}_t$ to $\boldsymbol{s}_{t+1}$ the flipped bit is marked as tabu which indicates that it cannot be altered again until after a certain number of iterations have elapsed. A parameter called the tabu tenure determines this length of time. The QSage algorithm automatically sets this parameter to a reasonable value. This tabu mechanism prevents the algorithm from rapidly returning to configurations it has already visited. The tabu tenure affects the adoption of new configurations with a simple new rule. Amongst all the variants generated around a particular configuration $\boldsymbol{s}$ we adopt the move to the variant which has the lowest objective value *and* which is not prevented by a tabu restriction. We note that this move to the lowest non-tabued variant may increase the objective value. This mechanism allows for escape from local minima in order to explore new and potentially more fruitful regions. Further details regarding refinements of tabu search not discussed here can be found in [Glover90].

While often effective, tabu search can be slow to explore the search space. Moreover, the optimal setting of the tabu tenure depends on the representation of the problem being solved and optimization performance can be quite sensitive to the tenure setting.

Here we improve upon tabu search by generating additional, and hopefully promising, targeted variants beyond the single-bit-flip variants of standard tabu. These additional variants are added to the pool of single-bit-flip variants and the best (lowest objective value) non-tabued variant within this larger pool is selected as the next candidate configuration.

Expressed as pseudocode, the important high-level steps of the algorithm are as follows:

```
1. Create a random initial configuration and determine its objective value.
2. Initialize the tabu tenure of all bits to 0.
3. While an outer loop termination condition is not met.
    (a) Generate all single bit flips of the current configuration.
    (b) Generate additional targeted variants of the current configuration by building a
        hardware-compatible surrogate model and sampling low energy configurations
        of the model.
    (c) Evaluate the objective value of all variants.
    (d) Update the current configuration to the variant with lowest objective value and
        whose bit-flips are not tabued.
    (e) Update the tabu list by setting the tabu tenure of the just flipped bits to
        tabuTenure, and by decrementing the tenure of all other bits.
4. Return the best configuration seen and its objective value.
```

Next, we drill down into the mechanism by which targeted variants are generated, and how these variants are tabued to prevent trapping in poor local minima.

7.3 Models for Targeted Variation

Tabu search relies on small changes to the current configuration to generate new variants. Small, rather than large, changes are critical to the success of this incremental approach. Once the algorithm has accumulated a number of improvements, the resulting configuration is significantly better than a randomly chosen configuration. To find another configuration which improves even further, a large change is unlikely to be successful unless the alteration is very carefully designed. Thus, making small changes is key to the success of the local search approach. Unfortunately, the small scale incremental approach means that when larger scale changes are necessary, the algorithm may fail to identify these larger changes through iterative local improvement. Thus, we develop a mechanism by which larger, but targeted, alterations can be proposed.

We first observe that if the problem you are trying to solve was in fact a hardware-structured Ising model we could use the QPU to propose variants that were very good solutions by running quantum annealing. Of course, most problems do not have the structure of the hardware. We can, however, build a model compatible with a hardware-structured Ising model, and which at least locally around the current configuration, approximates the true objective function $G(\boldsymbol{s})$. This approximation, when minimized in hardware, yields variants that do a good job at minimizing the model. Thus, if the model is moderately accurate (at least locally), then these model minimizers may be very useful variants for tabu to consider.

Model building can be made very flexible, and adaptable to any hardware qubit connectivity. All that is required to build a model is training data consisting of some spin configurations $\{\boldsymbol{s}_i\}$ and corresponding objective function values $\{G(\boldsymbol{s}_i)\}$. Within the machine learning literature, a great deal is known about building models from training data like this (the construction of such models is called supervised learning), and all of this insight can be brought to bear. As one simple example, linear regression, can be used to build a model. The parameters available in the linear model are the $\boldsymbol{h}$ values of all qubits, and the $\boldsymbol{J}$ values of all edges in the hardware connectivity graph. Values for $\boldsymbol{h}$ and $\boldsymbol{J}$ are set by minimizing the squared error between the hardware Chimera energies at $\{\boldsymbol{s}_i\}$ and the true values $\{G(\boldsymbol{s}_i)\}$.

However, in spite of the similarity to other supervised learning applications, there is one significant difference in the present case. In this application it is not the model itself that matters, but rather the minimizers of the model as these are proposed as variants for tabu search. Once a model is built, and some of its minimizers identified, it is important that these minimizers be local to the current configuration around which the model was constructed. If the minimizers are far from the current configuration then they will be large extrapolations from the region in which the model is likely to be valid. Consequently, distant minimizers are likely to be artifacts without statistical validity, and thus poor variants to suggest to tabu search. In building models then we want to bias towards those models having minimizers which are nearby to the current configuration around which the model has been trained.

In addition, models are built at every step in the tabu search, and because this happens so frequently the model building process must be fast. To minimize the time spent model building we use the training data we have available at hand, and very fast learning algorithms to construct the model. Fortunately, there is a nice supply of training data available in the configuration itself and all of its single bit flip variants which have been generated for tabu search. These configurations and the objective function values are used as training data. With more training data better models might be constructed, and the proposed variants might be better targeted. However, more training data requires evaluation of the objective at these new data points, and if the objective function is computationally expensive this may be costly. Consequently, we have adopted a conservative approach, and based the model only on the configuration and all its single bit flip variants. We rely on fast linear programming solvers to construct models. The linear programming approach strikes a balance between speed and accurate models.

We rely on the quantum annealing to perform approximate minimization of the local model. The fact that the QPU returns a diversity of answers is useful in this setting because the model is only an approximation to the true local objective. Generating multiple samples from the QPU is useful in supplying additional variants, but too many samples can become costly if evaluating the objective function on these samples is expensive. We address ways to work with costly objective functions in *Parallelization*.

7.4 Tabuing Variants

The tabu mechanism applied when moving to a single flip variant is simple—the single bit that was flipped is made tabu and prevented from flipping again for tabuTenure iterations. However, a variant proposed from the modeling process will differ from the current configuration in two or more bits. If such a configuration is adopted because it has lower objective value, then a new tabu mechanism must be specified. We might, for example, tabu all the flipped bits to prevent them moving again for tabuTenure iterations. However, such a move is too drastic and can rapidly lead to all bits being made tabu so that no moves are permitted. Moreover, the goal of the tabu mechanism is to prevent revisiting previously examined configurations, and some of these bits could be flipped and still not return to a previous state. Consequently, when making a multi-spin-flip move we randomly select one of the flipped bits and make that bit tabu. None of the other flipped bits are tabued.

When determining whether a multispin flip move can be adopted a similar issue arises. In the current QSage algorithm we allow the new state to be adopted as long as at least one of the flipped spins is not tabued. So even though some of the flipped spins may be tabued it is unlikely the move will be returning to a recently visited configuration because a non-tabued spin has also flipped.

While the choice adopted here in QSage works well for many problems, but alternative tabu mechanisms for variants at multiple bit-flip distances is an open research question, and future versions of QSage may change the current tabu mechanism.

7.5 Parallelization

The QSage algorithm itself is very lightweight, and typically the vast majority of run time is spent in evaluating the objective function $G(\boldsymbol{s})$ at configurations proposed as candidate variants. Thus the QSage routine offers the user the ability to parallelize these objective function evaluations in cases where the objective is computationally expensive. QSage generates batches of proposed variants consisting of single bit flips and hardware-proposed variants consisting of multiple bit flip variants. The number of variants within a batch is controlled by the solver parameter *num_reads*. QSage calls the objective function by passing in the entire batch of configurations, and not configuration by configuration. This allows the user to define a function returning objective values at all configurations within the batch. The user may then parallelize across these evaluations by spawning processes or threads to evaluate the objective at each constituent configuration, or at subsets of configurations.

Even if the objective is not parallelized, evaluating the objective at batches of configurations can prove beneficial. For some objectives required state can be calculated once and then shared across all configuration evaluations rather than being recalculated for each configuration. Depending on the form of the objective this savings can prove to be beneficial. The user should also keep in mind that the objective function may contain static variables which maintain their state between calls. This may also allow for faster evaluation if the objective function incrementally updates certain data structures which allow for faster evaluation.

7.6 solve_qsage

```
from dwave_sapi2.qsage import solve_qsage

answer = solve_qsage(objfunc, num_vars, solver, solver_params, qsage_params)
```

Note:

1. *solve_qsage* can be interrupted by Ctrl-C, it will return the best solution found so far.
2. For *num_vars* <= 10, *solve_qsage* will do a brute-force search.

7.6.1 Parameters

objfunc: the implementation of the objective function *f*. *solve_qsage* assumes the following signature:

> *y = f(states)* where *states* is a list of lists of states, *y* is the returned objective value for each state (*y* can be a list or tuple).

num_vars: number of variables.

solver: a solver handle.

solver_params: a dictionary of solver parameters.

qsage_params: a dictionary of QSage parameters. *qsage_params* can have the following fields:

> *draw_sample*: if False, *solve_qsage* will not draw samples, will only do tabu search. (must be a boolean, default = True)
>
> *exit_threshold_value*: if best value found by *solve_qsage* <= *exit_threshold_value* then exit. (can be any number, default = float('-inf'))
>
> *init_solution*: the initial solution for the problem. (must be a matrix containing only -1/1 if *ising_qubo* parameter is not set or set as *'ising'*; or 0/1 if *ising_qubo* parameter is set as *'qubo'*. The number of elements of *init_solution* must be equal to *num_vars*, default is randomly set)
>
> *ising_qubo*: if set as *'ising'*, the return best solution will be -1/1; if set as *'qubo'*, the return best solution will be 0/1. (must be a string *'ising'* or *'qubo'*, default = *'ising'*)
>
> *lp_solver*: a solver that can solve linear programming problems. Finds the min-

imum of a problem specified by

```
min      f * x
st.      Aineq * x <= bineq
           Aeq * x = beq
           lb <= x <= ub
```

solve_qsage assumes the following signature for *lp_solver*:

```
x = solve(f, Aineq, bineq, Aeq, beq, lb, ub)
```

input arguments (suppose *f*'s size is f_size, *Aineq*'s size is Aineq_size by f_size, Aeq's size is Aeq_size by f_size):

f: linear objective function, a list. (size: f_size)

Aineq: linear inequality constraints, a list of lists. (size: Aineq_size by f_size)

bineq: righthand side for linear inequality constraints, a list. (size: Aineq_size)

Aeq: linear equality constraints, a list of lists. (size: Aeq_size by f_size)

beq: righthand side for linear equality constraints, a list. (size: Aeq_size)

lb: lower bounds, a list. (size: f_size)

ub: upper bounds, a list. (size: f_size)

output arguments:

x: solution found by the optimization function. (size: f_size)

(default uses Coin-or Linear Programming solver)

max_num_state_evaluations: the maximum number of state evaluations, if the total number of state evaluations >= *max_num_state_evaluations* then exit. (must be an integer >= 0, default = 50,000,000)

random_seed: seed for random number generator that *solve_qsage* uses. (must be an integer >= 0, default is randomly set)

timeout: timeout for *solve_qsage*. (seconds, must be a number >= 0, default is approximately 10.0 seconds)

verbose: control the output information. (must be an integer [0 2], default = 0)

0: quiet

1, 2: different levels of verbosity

when *verbose* is 1, the output information will be like:

[num_state_evaluations = ..., num_obj_func_calls = ..., num_solver_calls = ..., num_lp_solver_calls = ...],

best_energy = ..., distance to best_energy = ...

detailed explanation of the output information:

- "num_state_evaluations": the current total number of state evaluations

- "num_obj_func_calls": the current total number of objective function calls
- "num_solver_calls": the current total number of solver calls
- "num_lp_solver_calls": the current total number of lp solver calls
- "best_energy": the global best energy found so far
- "distance to best_energy": the hamming distance between the global best state found so far and the current state found by tabu search

when *verbose* is 2, in addition to the output information when verbose is 1, the following output information will also be shown:

sample_num = ...

min_energy = ...

move_length = ...

detailed explanation of the output information:

- "sample_num": the number of unique samples returned by sampler
- "min_energy": minimum energy found during the current phase of tabu search
- "move_length": the length of the move (the hamming distance between the current state and the new state)

The acceptable range and the default value of each field are given in the table below:

Field	Range	Default value
draw_sample	True or False	True
exit_threshold_value	any number	float('-inf')
initial_solution	N/A	randomly set
ising_qubo	*'ising'* or *'qubo'*	*'ising'*
lp_solver	N/A	uses Coin-or Linear Programming solver
max_num_state_evaluations	>= 0	50,000,000
random_seed	>= 0	randomly set
timeout	>= 0.0	10.0
verbose	[0 2]	0

7.6.2 Return Value

answer: answer is a dictionary which has the following keys:

- "best_solution": A list of +1/-1 or 1/0 values giving the best solution.
- "best_energy": best energy found for the given objective function.
- "num_state_evaluations": number of state evaluations.

- “num_obj_func_calls”: number of objective function calls.
- “num_solver_calls”: number of solver (local/remote) calls.
- “num_lp_solver_calls”: number of lp solver calls.
- “state_evaluations_time”: state evaluations time (seconds).
- “solver_calls_time”: solver (local/remote) calls time (seconds).
- “lp_solver_calls_time”: lp solver calls time (seconds).
- “total_time”: total running time of *solve_qsage* (seconds).
- “history”: A list of lists of number of state evaluations, number of objective function calls, number of solver (local/remote) calls, number of lp solver calls, time (seconds) and objective value.

7.6.3 Example

```
from dwave_sapi2.local import local_connection
from dwave_sapi2.qsage import solve_qsage

# objective function can be a class or a function

class ObjClass:
    def __call__(self, states):
        num_states = len(states)
        state_len = len(states[0])
        ret = []
        for i in range(num_states):
            d1 = 0
            for j in range(state_len / 2):
                d1 = d1 + states[i][j]

            d2 = 0
            for j in range(state_len / 2, state_len):
                d2 = d2 + states[i][j]

            ret.append(d1 - d2)

        return ret

def obj_function(states):
    num_states = len(states)
    state_len = len(states[0])
    ret = []
    for i in range(num_states):
        d1 = 0
        for j in range(state_len / 2):
            d1 = d1 + states[i][j]

        d2 = 0
        for j in range(state_len / 2, state_len):
            d2 = d2 + states[i][j]
        ret.append(d1 - d2)

    return ret


solver = local_connection.get_solver("c4-sw_sample")
num_vars = 12
answer_1 = solve_qsage(ObjClass(), num_vars, solver, {"num_reads": 100},
                                                     {"verbose": 2})
print "answer_1:", answer_1
print "\n\n"
answer_2 = solve_qsage(obj_function, num_vars, solver, {"num_reads": 100},
                                                       {"verbose": 2})
print "answer_2:", answer_2

Output:

sample_num = 1
min_energy = 2.000000
sample_num = 1
min_energy = 0.000000
sample_num = 1
min_energy = -2.000000
sample_num = 1
min_energy = -4.000000
sample_num = 1
min_energy = -6.000000
sample_num = 1
min_energy = -8.000000
sample_num = 1
min_energy = -10.000000
sample_num = 1
min_energy = -12.000000
sample_num = 1
min_energy = -12.000000
move_length = 1
[num_state_evaluations = 127, num_obj_func_calls = 28, num_solver_calls = 9,
 num_lp_solver_calls = 9], best_energy = -12.000000, distance to best_energy = 1
sample_num = 1
min_energy = -12.000000
move_length = 1
sample_num = 1
min_energy = -10.000000
move_length = 1
sample_num = 1
min_energy = -8.000000
move_length = 1
sample_num = 1
min_energy = -6.000000
move_length = 1
sample_num = 1
min_energy = -8.000000
move_length = 1
sample_num = 1
min_energy = -10.000000
```

```
move_length = 1
sample_num = 1
min_energy = -12.000000
move_length = 1
sample_num = 1
min_energy = -12.000000
move_length = 1
[num_state_evaluations = 239, num_obj_func_calls = 52, num_solver_calls = 17,
 num_lp_solver_calls = 17], best_energy = -12.000000, distance to best_energy = 1
sample_num = 1
min_energy = -12.000000
move_length = 1
sample_num = 1
min_energy = -10.000000
move_length = 1

10.000000 seconds timeout has been reached.
answer_1: {'num_solver_calls': 19L, 'total_time': 10.366,
           'best_solution': [-1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1],
           'state_evaluations_time': 0.002,
           'lp_solver_calls_time': 0.15500000000000005, 'best_energy': -12.0,
           'num_obj_func_calls': 58L, 'num_lp_solver_calls': 19L,
           'history': [[1L, 1L, 0L, 0L, 0.001, 4.0], [127L, 28L, 9L, 9L, 4.853, -12.
 ↪0]],
           'solver_calls_time': 9.825, 'num_state_evaluations': 267L}



sample_num = 1
min_energy = 2.000000
sample_num = 1
min_energy = 0.000000
sample_num = 1
min_energy = -2.000000
sample_num = 1
min_energy = -4.000000
sample_num = 1
min_energy = -6.000000
sample_num = 1
min_energy = -8.000000
sample_num = 1
min_energy = -10.000000
sample_num = 1
min_energy = -12.000000
sample_num = 1
min_energy = -12.000000
move_length = 1
[num_state_evaluations = 127, num_obj_func_calls = 28, num_solver_calls = 9,
 num_lp_solver_calls = 9], best_energy = -12.000000, distance to best_energy = 1
sample_num = 1
min_energy = -12.000000
move_length = 1
sample_num = 1
min_energy = -10.000000
move_length = 1
```

```
sample_num = 1
min_energy = -8.000000
move_length = 1
sample_num = 1
min_energy = -6.000000
move_length = 1
sample_num = 1
min_energy = -4.000000
move_length = 1
sample_num = 1
min_energy = -6.000000
move_length = 1
sample_num = 1
min_energy = -8.000000
move_length = 1
sample_num = 1
min_energy = -10.000000
move_length = 1
sample_num = 1
min_energy = -12.000000
move_length = 1
sample_num = 1
min_energy = -12.000000
move_length = 1
[num_state_evaluations = 267, num_obj_func_calls = 58, num_solver_calls = 19,
 num_lp_solver_calls = 19], best_energy = -12.000000, distance to best_energy = 1
sample_num = 1
min_energy = -12.000000
move_length = 1

10.000000 seconds timeout has been reached.
answer_2: {'num_solver_calls': 19L, 'total_time': 10.366,
           'best_solution': [-1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1],
           'state_evaluations_time': 0.002,
           'lp_solver_calls_time': 0.15500000000000005, 'best_energy': -12.0,
           'num_obj_func_calls': 58L, 'num_lp_solver_calls': 19L,
           'history': [[1L, 1L, 0L, 0L, 0.001, 4.0], [127L, 28L, 9L, 9L, 4.853, -12.
 ↪0]],
           'solver_calls_time': 9.825, 'num_state_evaluations': 267L}
```

Note: The solutions you get may not be exactly the same as above.

CHAPTER 8

UTILITIES

8.1 ising_to_qubo

```
from dwave_sapi2.util import ising_to_qubo

(Q, qubo_offset) = ising_to_qubo(h, J)
```

As we have observed, qubo and ising problems are two different representations for the same problem. In certain applications, one or the other may be more convenient.

ising_to_qubo converts an ising problem to a qubo equivalent.

Explicitly, $x'Qx + qubo\_offset = s'Js + h's$ where x is the column vector of boolean values, Q is the matrix of qubo coefficients, s is the column vector of ± 1 values, and h, J are the ising parameters. *qubo_offset* is a constant value that shifts all qubo energies.

8.1.1 Parameters

h: a list of ising coefficients.

J: a dictionary of ising coupling coefficients.

8.1.2 Return Value

Q: a dictionary of equivalent qubo coefficients of an ising problem specified by h and J.

qubo_offset: a constant value that shifts all qubo energies.

Note: It is assumed that ising and qubo models are related to each other through the transformation $s = 2x - 1$.

8.1.3 Example

```
from dwave_sapi2.util import ising_to_qubo

```

```
h = [1, 1, -1]
J = {(0, 1): 15, (0, 2): -5, (1, 0): 30, (1, 2): 5, (2, 0): 5, (2, 1): -20}

(Q, qubo_offset) = ising_to_qubo(h, J)

print "Q:", Q
print "qubo_offset:", qubo_offset

Output:

Q: {(0, 1): 60.0, (1, 2): 20.0, (0, 0): -88, (2, 1): -80.0, (1, 1): -58, (2, 0): 20.
↪0,
    (2, 2): 28, (1, 0): 120.0, (0, 2): -20.0}
qubo_offset: 29
```

See also:

qubo_to_ising

8.2 qubo_to_ising

```
from dwave_sapi2.util import qubo_to_ising

(h, J, ising_offset) = qubo_to_ising(Q)
```

Converts a qubo problem to an ising equivalent.

Explicitly, $s'Js + h's + ising\_offset = x'Qx$ where h is a vector containing the equivalent linear ising coefficients and J is a matrix of equivalent ising coupling coefficients. x is the column vector of boolean values, Q is the matrix of qubo coefficients. *ising_offset* is a constant shifting all ising energies.

8.2.1 Parameters

Q: a dictionary of qubo coefficients.

8.2.2 Return Value

h: a list of ising coefficients.

J: a dictionary of ising coupling coefficients.

ising_offset: a constant shifting all ising energies.

Note: It is assumed that ising and qubo models are related to each other through the transformation $s = 2x - 1$.

8.2.3 Example

```
from dwave_sapi2.util import qubo_to_ising

Q = {(0, 0): -88, (0, 1): 60.0, (0, 2): -20.0,
     (1, 0): 120.0, (1, 1): -58, (1, 2): 20.0,
     (2, 0): 20.0, (2, 1): -80.0, (2, 2): 28}

(h, J, ising_offset) = qubo_to_ising(Q)

print "h:", h
print "J:", J
print "ising_offset:", ising_offset

Output:

h: [1.0, 1.0, -1.0]
J: {(0, 1): 15.0, (1, 2): 5.0, (2, 1): -20.0, (2, 0): 5.0, (1, 0): 30.0, (0, 2): -5.
 ↪0}
ising_offset: -29.0
```

See also:

ising_to_qubo

8.3 get_chimera_adjacency

```
from dwave_sapi2.util import get_chimera_adjacency

A = get_chimera_adjacency(m, n, t)
```

Returns the adjacency matrix A for the Chimera connectivity graph.

Qubits in the processor are connected in a particular architecture called Chimera. Chimera connections are best expressed using a different indexing notation for qubits. To index any qubit, we use four numbers (i, j, u, k). (i, j) indexes the (row, column) of the block. i must be between 0 and $m-1$ inclusive, and j must be between 0 and $n-1$ inclusive. $u = 0$ indicates the left hand qubits in the block, and $u = 1$ indicates the right hand qubits. $k = 0, 1, \cdots, t-1$ indexes qubits within either the left- or right-hand side of a block.

As an example, *Chimera indexing* shows this 4-tuple indexing for a $4 \times 5 \times 4$ Chimera, and highlights the coordinates of two particular qubits. $u = 0$ nodes are indicated in blue, and $u = 1$ nodes in red. In this notation, the qubits indexed by (i, j, u, k) and (i', j', u', k') are neighbours if and only if:

1. $i = i'$ and $j = j'$ and $[(u, u') = (0, 1)$ or $(u, u') = (1, 0)]$ OR
2. $i = i' \pm 1$ and $j = j'$ and $u = u'$ and $u = 0$ and $k = k'$ OR
3. $i = i'$ and $j = j' \pm 1$ and $u = u'$ and $u = 1$ and $k = k'$.

The first rule accounts for connections within a block, the second rule accounts for vertical connections between blocks, and the last rule accounts for horizontal connections between

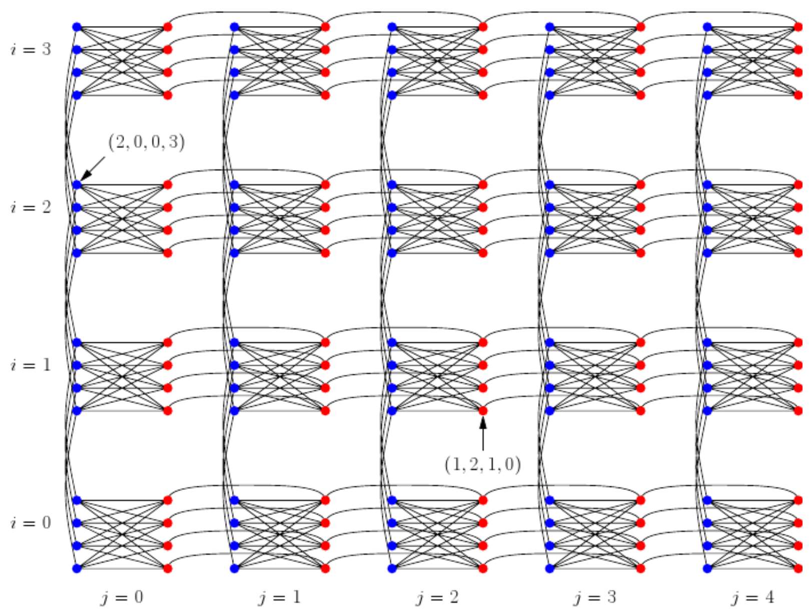

Figure 8.1: Chimera indexing

blocks. Variables indexed by a negative i or j are ignored. [1]

A 4-tuple index (i, j, u, k) (as used in *Chimera indexing*) can be converted to a linear index ℓ (as used in chimeraGraph) according to the formula

$$\ell(i, j, u, k) = 1 + 2nti + 2tj + tu + k$$

where $m \times n \times t$ is the size of the Chimera lattice. We supply functions to make these translations. Refer to linearToChimeraIndex and chimeraToLinearIndex.

8.3.1 Parameters

m, n, t: the Chimera lattice consists of an $m \times n$ array of unit cells each consisting of $2t$ qubits.

8.3.2 Return Value

A: a set which contains the pairs of qubits that interconnect on the Chimera connectivity graph. It is symmetric, i.e., if qubits i and j are connected, both (i, j) and (j, i) exist in A.

[1] The Chimera adjacency matrix $\mathbf{A}$ has matrix elements $A_{i,j,u,k;i',j',u',k'} = \delta_{i,i'}\delta_{j,j'}\delta_{u,u'+1} + \delta_{i,i'\pm1}\delta_{j,j'}\delta_{u,u'}\delta_{u,0}\delta_{k,k} + \delta_{i,i'}\delta_{j,j'\pm1}\delta_{u,u'}\delta_{u,1}\delta_{k,k'}$ where $\delta_{a,b}$ is the Kronecker delta function equal to 1 if $a = b$ and 0 if $a \neq b$. Addition on u is modulo 2. Expressed another way, $\mathbf{A} = \mathbf{I}_M \otimes \mathbf{I}_N \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \mathbf{1}_L + \mathbf{L}_M \otimes \mathbf{I}_N \otimes \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \otimes \mathbf{I}_L + \mathbf{I}_M \otimes \mathbf{L}_N \otimes \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \otimes \mathbf{I}_L$ where $\otimes$ is the matrix tensor product, $\mathbf{I}_n$ is the $n \times n$ identity, $\mathbf{1}_n$ is the $n \times n$ matrix of 1's, and $\mathbf{L}_n$ is the $n \times n$ adjacency matrix for the graph representing a line of n nodes each connected to its left and right neighbours.

8.3.3 Example

The following example returns a 512 × 512 adjacency matrix A (for a fully working Chimera).

```
from dwave_sapi2.util import get_chimera_adjacency

m = 8
n = 8
t = 4

A = get_chimera_adjacency(m, n, t)
```

See also:

get_hardware_adjacency | *linear_index_to_chimera* | *chimera_to_linear_index*

8.4 get_hardware_adjacency

```
from dwave_sapi2.util import get_hardware_adjacency

A = get_hardware_adjacency(solver)
```

Returns adjacency matrix for the solver.

8.4.1 Parameters

solver: the solver handle.

8.4.2 Return Value

A: a set which contains the pairs of qubits that interconnect in the solver. The set A is symmetric, i.e., if qubits i and j are connected, both (i, j) and (j, i) exist in A.

8.4.3 Example

The following example finds the adjacency matrix for a local solver.

```
from dwave_sapi2.local import local_connection
from dwave_sapi2.util import get_hardware_adjacency

solver = local_connection.get_solver("c4-sw_sample")
A = get_hardware_adjacency(solver)
```

See also:

get_chimera_adjacency

8.5 linear_index_to_chimera

```
from dwave_sapi2.util import linear_index_to_chimera

ind = linear_index_to_chimera(linear_index, m, n, t)
```

To indicate connectivity in Chimera, it is convenient to employ a different indexing scheme for qubits such as Chimera indexing. *linear_index_to_chimera* converts the linear index *linear_index* in an $m \times n \times t$ Chimera lattice into a list of indices *ind* according to the scheme outlined in *Chimera indexing*. *linear_index* can be a single index, a list or a tuple.

8.5.1 Parameters

linear_index: a single index or a list of indices.

m, n, t: Chimera dimensions.

8.5.2 Return Value

ind: list of Chimera coordinates.

8.5.3 Example

The following example converts a list of linear indices to its corresponding Chimera index.

```
from dwave_sapi2.util import linear_index_to_chimera

linear_index = [0, 1, 2]
index = linear_index_to_chimera(linear_index, 8, 8, 4)

print "index:", index

Output:

index: [[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 0, 2]]
```

See also:

chimera_to_linear_index | *get_chimera_adjacency* | *get_hardware_adjacency*

8.6 chimera_to_linear_index

```
from dwave_sapi2.util import chimera_to_linear_index

ind = chimera_to_linear_index(i, j, u, k, m, n, t)
```

To indicate connectivity in Chimera, it is convenient to employ a different indexing scheme for qubits. *chimera_to_linear_index* converts a Chimera graph in to a linear index.

It implements the inverse transformation of *linear_index_to_chimera* and converts the list/set/tuple of indices (i, j, u, k) in an $m \times n \times t$ Chimera graph into a linear index. The examples below show two different ways of using *chimera_to_linear_index*.

8.6.1 Example

Example 1

The following example finds the linear index of the input Chimera index 0, 0, 0, 0.

```
from dwave_sapi2.util import chimera_to_linear_index

# calculate the linear index of chimera index: (0, 0, 0, 0)
i = [0]
j = [0]
u = [0]
k = [0]
index = chimera_to_linear_index(i, j, u, k, 8, 8, 4)

print "index:", index

Output:

index: [0]
```

Example 2

The following example finds the linear indices of two Chimera indices.

```
from dwave_sapi2.util import chimera_to_linear_index

# calculate the linear index of chimera index: (0, 0, 0, 0) and (0, 1, 0, 0)
param0 = [[0, 0, 0, 0], [0, 1, 0, 0]]

index = chimera_to_linear_index(param0, 8, 8, 4)

print "index:", index

Output:

index: [0, 8]
```

CHAPTER 9

API TOKENS

To submit a problem to the D-Wave system, you require an API *token*, which the system uses to authenticate the client session when connecting to the remote environment. Because we use tokens for authentication, user names and passwords are not required in code. Furthermore, because tokens are associated with the projects defined in the system, they allow problems to be tracked by project. You can generate as many tokens as you need through Qubist.

To generate a token:

1. From any page, click your user name in the top menu bar and select **API Tokens**.
2. On the Token Management page, click **Create Token**.
3. In the Create New Token dialog box, select your project from the list and click **Create**. The system generates a token associated with the project. The token is a lengthy string; you will cut and paste it when you are ready to submit a problem to the system.

API tokens can be used by anyone who has access to the system and who knows the token ID.

BIBLIOGRAPHY

[Ahuja2000] Ahuja, R. K.; Orlin, J. B.; Sharma, D. *Very large-scale neighborhood search*, International Transactions in Operational Research 7 (4-5): 301-317, (2000).

[Glover90] Glover, F., *Tabu Search: A Tutorial*, Interfaces July/August 1990 20:74-94;