D:Wave
The Quantum Computing Company™

Programming the D-Wave QPU: Parameters for Beginners

WHITEPAPER

2020-09-17

Overview

This document explores the importance of the required parameters that are involved in programming the D-Wave quantum processing unit (QPU).

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

14-1045A-A
D-Wave Whitepaper Series

Notice and Disclaimer

Contents

1 Introduction

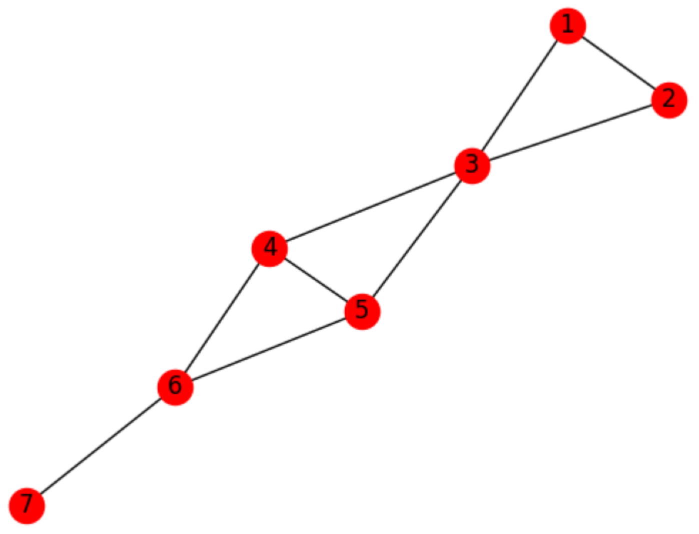
The D-Wave quantum processing unit (QPU) accepts many parameters that you can specify when submitting a problem. Many of these parameters are not required for basic use of the system; they are associated with advanced research into performance tuning of quantum annealing. This document focuses on the small set of parameters that are required for almost every problem. For a full list of supported parameters, see the D-Wave Solver Properties and Parameters Guide [1].

2 Number of Reads

The `num_reads` parameter tells the QPU how many times to run a problem, once the problem has been programmed onto the QPU hardware. Each run of a problem is also known as a *read*, or an annealing cycle. The default, and the maximum, vary from system to system. For the D-Wave 2000Q, the default is 10, and the maximum is 10000. Each system's values can be obtained programmatically from the system [2].

The typical timescale for individual reads of a problem is microseconds, so it is inexpensive to increase the number of reads. Even with 1000 reads, the timescale is milliseconds, and you won't use much time on the system at that rate. There are two main reasons for using larger values for `num_reads`. First, the QPU is probabilistic, and there is no guarantee that a particular solution will emerge on a given read of the QPU. A larger number of reads will increase the probability that diverse solutions will be returned.

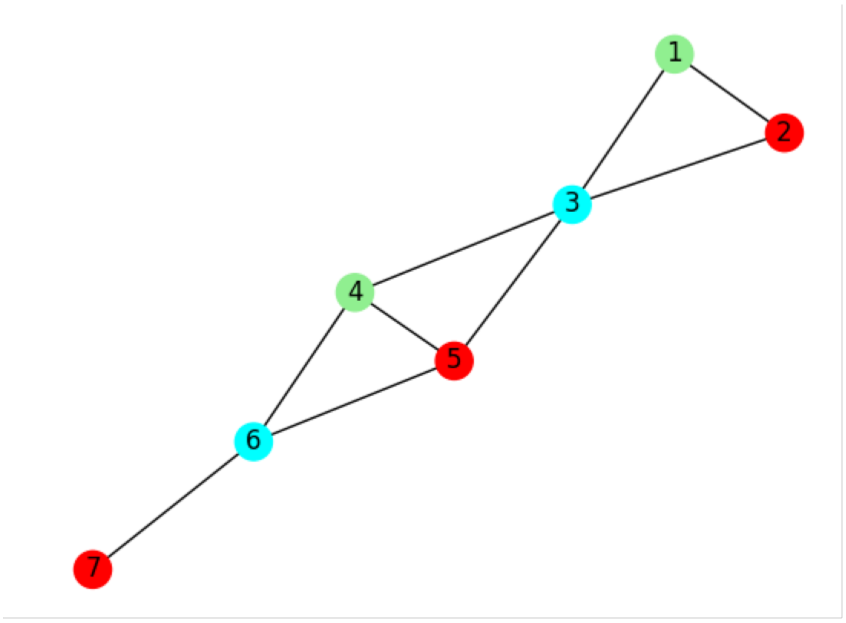
To see this, let's look at a small graph coloring problem. This is the graph of the problem:



The goal of this program is to assign colors to each node in the graph so that no neighboring node has the same color. Here is a Python program to solve this problem, written using D-Wave's Ocean package [3]:

```
import networkx as nx
import dwave_networkx as dnx
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import EmbeddingComposite
from dwave_networkx.algorithms.coloring import min_vertex_color_qubo

sampler = EmbeddingComposite(DWaveSampler())

G = nx.Graph()
G.add_edges_from([(1, 2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 5),
                 (4, 6), (5, 6), (6, 7)])

Q = min_vertex_color_qubo(G)

sampleset = sampler.sample_qubo(Q, num_reads=100)

for sample, energy, num_occ in sampleset.data(['sample', 'energy',
     'num_occurrences']):
    colors_chosen = [i for i in sample if sample[i] == 1]
    print(colors_chosen, energy, num_occ)
```

D-Wave's Ocean package `dwave_networkx` has a method, `min_vertex_color_qubo`, that generates the problem's quantum unconstrained binary optimization model (QUBO), and then we solve the problem on the QPU. One solution looks like this:



A recent run of this problem with `num_reads = 10` produced 5 different solutions; another, with `num_reads = 100`, produced 18 different solutions. For a problem like this, with many valid solutions, it is clear that increasing num_reads provides more diversity in the returned solutions.

Second, larger values of num_reads may make statistical trends apparent. Consider a very small QUBO problem involving 2 qubits, which are constrained to favor solutions in which the qubits have the same value ((0, 0) and (1, 1)) as opposed to solutions in which the qubits have opposite values ((0, 1) and (1, 0)). Here is an Ocean program for this problem:

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

Q = {(0, 0): 1, (1, 1): 1, (0, 1): -2}

sampler = EmbeddingComposite(DWaveSampler())

bqm = dimod.BinaryQuadraticModel.from_qubo(Q)

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

For two equivalent ground state solutions (both with energy zero), we expect that the QPU will find 500 solutions of (0, 0) and 500 solutions of (1, 1). Because of the probabilistic nature of the QPU, however, on any individual run, we do not expect precisely 500 of each. Rather, we expect that, on average, we will get 500 of each.

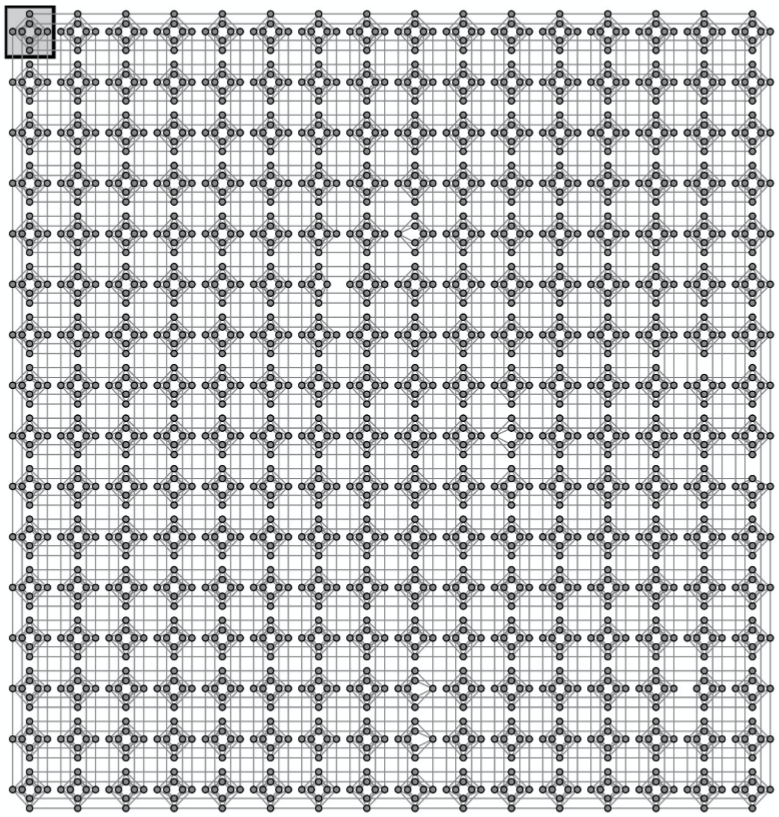
Repeated runs of this program can show substantial variability – totals like 750 and 250, for instance. If a problem runs only once, you might make unwarranted conclusions about the trending of the results. Over many samples, though, we expect the totals to average out to 500.

3 Chain Strength

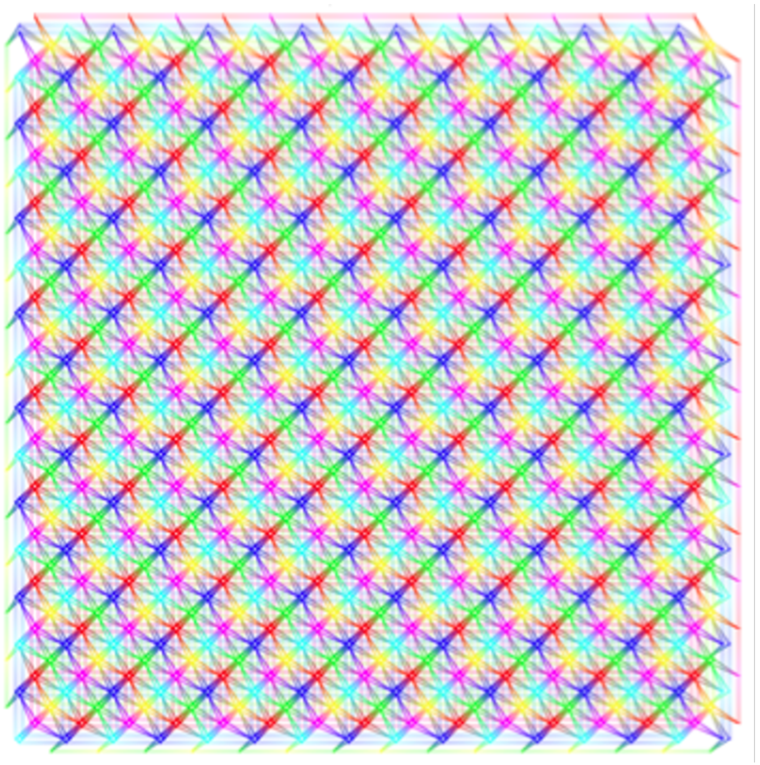
The chain_strength parameter specifies the relative strength of chains embedded on the QPU, which become important when a problem's graph does not map one-to-one, problem variable to physical qubit, onto the QPU. Since most problem graphs have different topology than the QPU's architecture, the chain_strength parameter is required for most problems.

3.1 What is a chain?

The D-Wave QPU is comprised of a grid of qubits and couplers. The D-Wave 2000Q implements the Chimera graph:

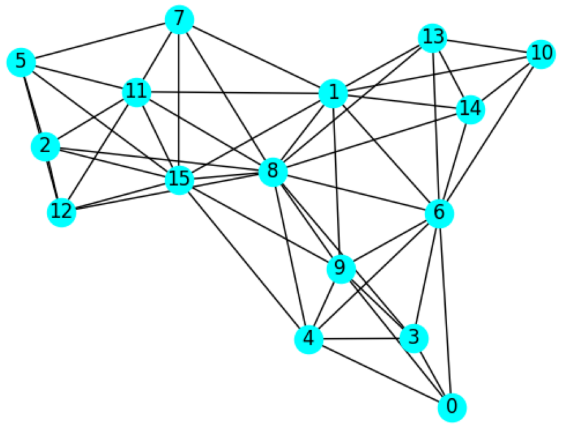


Advantage implements the Pegasus graph structure:



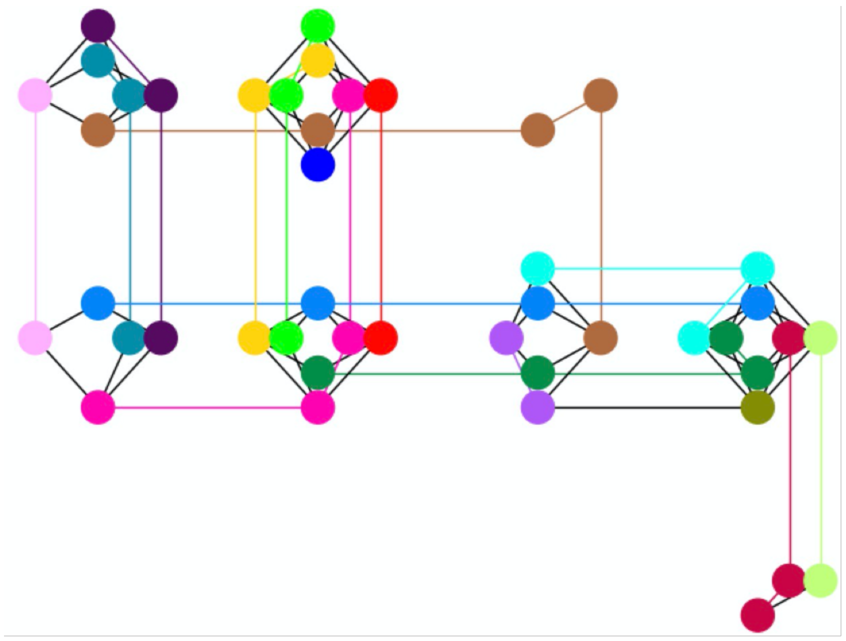
The Advantage graph is more complex; please see [4].

A small problem helps us understand how to map a problem onto these architectures. Consider a problem in which we randomly place $N = 16$ objects into a square, and connect the objects with an edge only if the distance between them is smaller than a certain radius, which we choose to be 0.5, half of the distance across the square. The resulting graph is known as a random geometric graph [5], and a typical random geometric graph looks like this:



The question is whether this graph would “fit” onto either D-Wave architecture: that is, whether we could map each node in the graph (for example, 12, or 15) onto a single physical qubit.

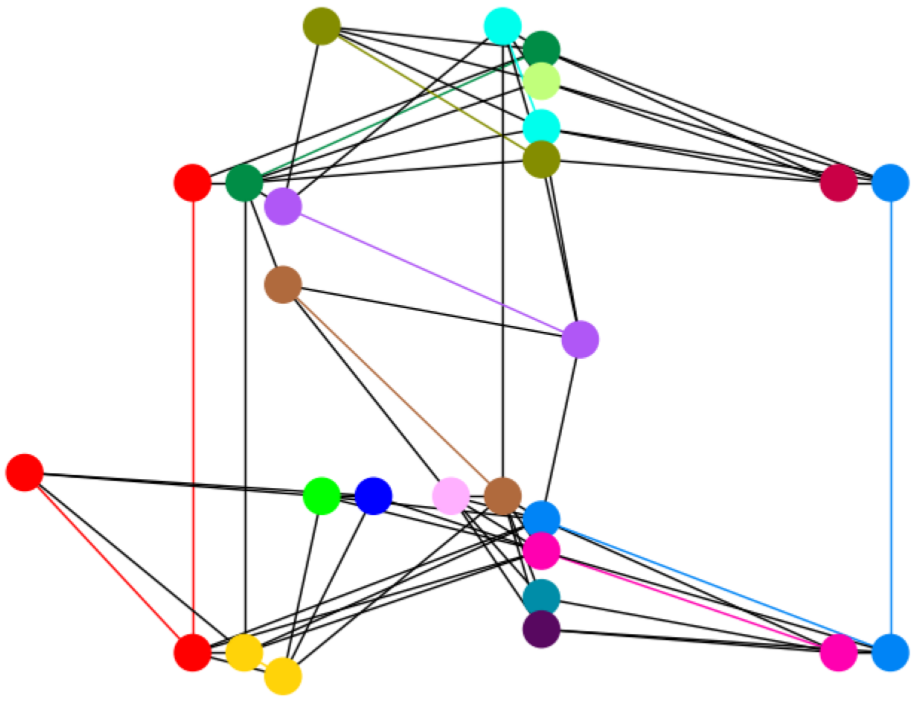
D-Wave provides embedding software in its Ocean toolkit [3]. If we run that software, and embed this problem onto the Chimera graph of the D-Wave 2000Q QPU, we are likely to get an embedding that looks like this:



Notice the different colors, and how each color may have a few physical qubits (circles) associated with it. For example, consider the three bright green qubits in the center of the picture. These three qubits were selected by the D-Wave embedding algorithm to together represent one of the nodes (0-15) in the problem graph above. Notice that the three qubits are connected; this connection is known as a chain. We say that the three qubits have been chained together, to represent a single logical node. This allows the logical node to reach (have a coupler in common with) other qubits, in the graph of the overall problem.

Looking at the picture, we can see that the D-Wave embedding algorithm used chains of length 2, 3, 4 and 5, to be able to represent the problem graph. (For example, there is a chain of length 3 represented by the color cyan; there is a chain of length 2 represented by the color violet.)

Now we can also embed this problem on the Advantage architecture:

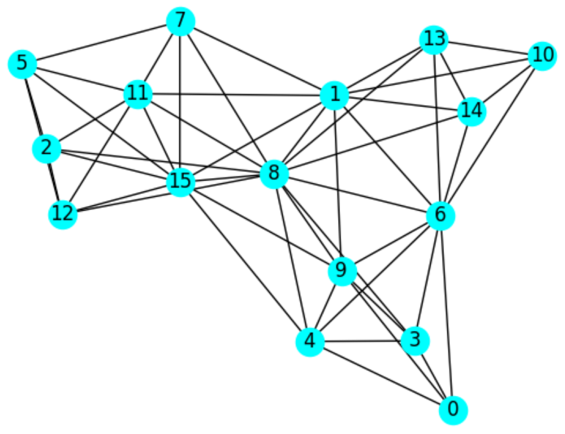


The embedding on the Advantage architecture uses fewer qubits, and the chains are shorter. This will be discussed further in the next section.

3.2 What is a chain break?

This section introduces the notion of a chain break and introduces the `chain_strength` parameter.

First, let's return to the problem in Section 3.1, the random geometric graph:



Our goals are to divide the graph into two subsets (also known as partitions), which we want to have equal size, and to minimize the number of links between the partitions. This problem is known as *graph partitioning*, and the general problem is NP-hard [6].

To run this problem on the QPU, we need to express the problem either as a QUBO problem or as an Ising model. We choose the QUBO approach; to do this, we write the graph partitioning problem as a sum of two terms. The first term is something that we want to minimize – in this case, the number of links between the partitions. The second term is composed of constraints, which are mathematical expressions on binary variables (0 or 1). We want the quantum computer to adjust the binary variables and return a solution that minimizes the first term and satisfies all the constraints.

The QUBO for graph partitioning has the following form:

$$\min \sum_{(i,j)\in E} (x_i + x_j - 2x_ix_j) + \gamma \left(-N \cdot \sum_i^N x_i + \sum_i^N x_i + \sum_i^N \sum_{j>i} 2x_ix_j \right)$$

where x_i is the value of the ith binary variable, $N = 16$ objects, and γ is the *Lagrange parameter*. The Lagrange parameter is adjusted depending on the relative strengths of the first term and the constraints, which are represented by the term in the parenthesis. A good value for the Lagrange parameter can be found by trial-and-error, but for this example, we know that $\gamma = 60$ will be acceptable.

In the previous section, when we discussed chains, we said that the member qubits are chained together. This means that they are mathematically constrained to have the same value: 0 or 1 (QUBO); or -1 or 1 (Ising). To accomplish this, the parameter `chain_strength` is used, which gives the chain its ability to compete with the relative strengths of the first term in the QUBO and the constraints. If the `chain_strength` parameter is too small, the qubits in each chain won't have the same value, and the required relationships between qubits won't work as intended. A chain in which qubits take different values is called a *broken chain*, and the problem won't be quite the same as the graph that the programmer wants to solve.

Conversely, if the `chain_strength` parameter is too large, the embedding terms will dominate the graph partitioning problem details, and we might lose the ability to study the intended problem. It is clear that the `chain_strength` parameter must be chosen carefully: not too small and not too large.

Furthermore, when chains are long, it is challenging to find a value for the `chain_strength` parameter that is strong enough to balance the other terms in the problem. Because the Pegasus architecture in Advantage requires shorter chains than does the Chimera architecture to embed the same problem, the impact of this effect is reduced.

In the next section, we will write a program to solve this problem, and will explore broken chains and their effects.

4 Writing an Initial Python Program for the QPU

In this section, we write a program using the Ocean software [3], D-Wave's open-source Python SDK. The program includes the following lines of code:

```
import dwave.inspector

dwave_sampler = FixedEmbeddingComposite(DWaveSampler(solver={'qpu': True}),
    embedding)

bqm = dimod.BinaryQuadraticModel.from_qubo(Q, offset=offset)

sampleset = dwave_sampler.sample(bqm, num_reads=1000)

dwave.inspector.show(sampleset)
```

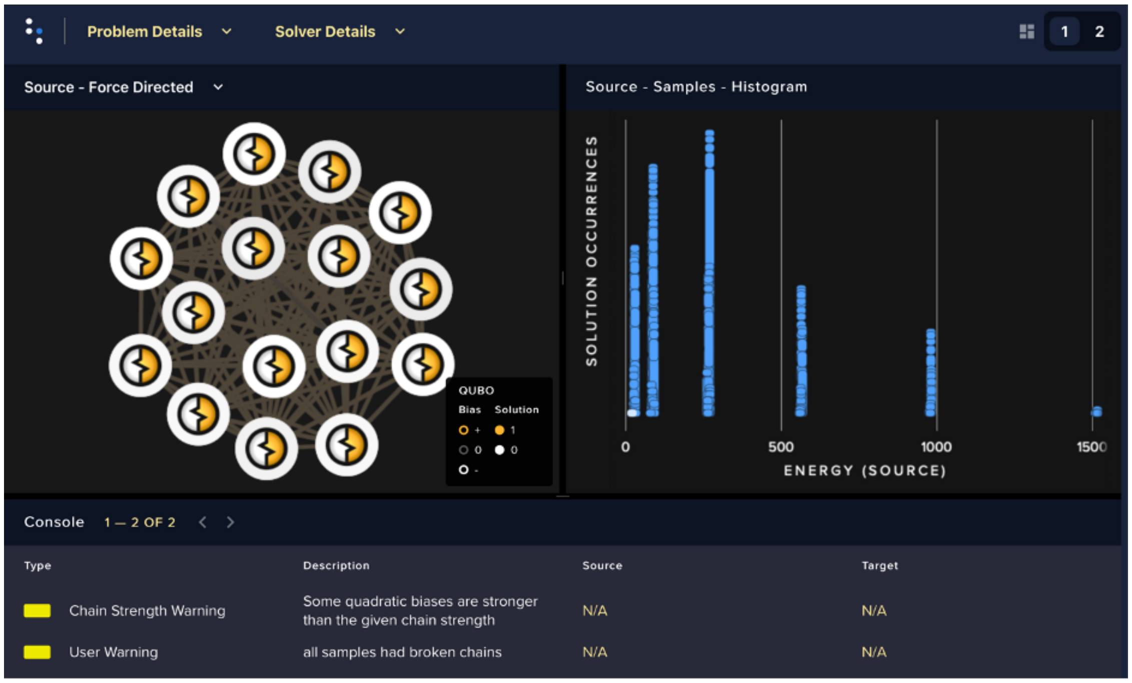
This code fragment performs the following steps:

1. Import the D-Wave problem inspector [7], so that we can graphically display the results.
2. Arrange for the QPU to solve the problem, using the DWaveSampler object [2] in Ocean [3]
3. Embed [8] the problem onto the graph structure of the QPU with FixedEmbedding-Composite [9], using an embedding that we found using a different Ocean package, `minorminer` [10].
4. Solve the problem, with 1000 reads for each, using `dwave_sampler.sample` [11].
5. Display the problem in the D-Wave problem inspector.

5 Running the program on the QPU

Here are the results from an initial run of the above Python program, including the command line output and the view from the problem inspector.

```
Number of nodes in one set is 8, in the other, 8.
The number of links between partitions is 19.0.
Percentage of valid solutions is 0.1.
Percentage of samples with high rates of breaks is 84.8.
```



Type	Description	Source	Target
Chain Strength Warning	Some quadratic biases are stronger than the given chain strength	N/A	N/A
User Warning	all samples had broken chains	N/A	N/A

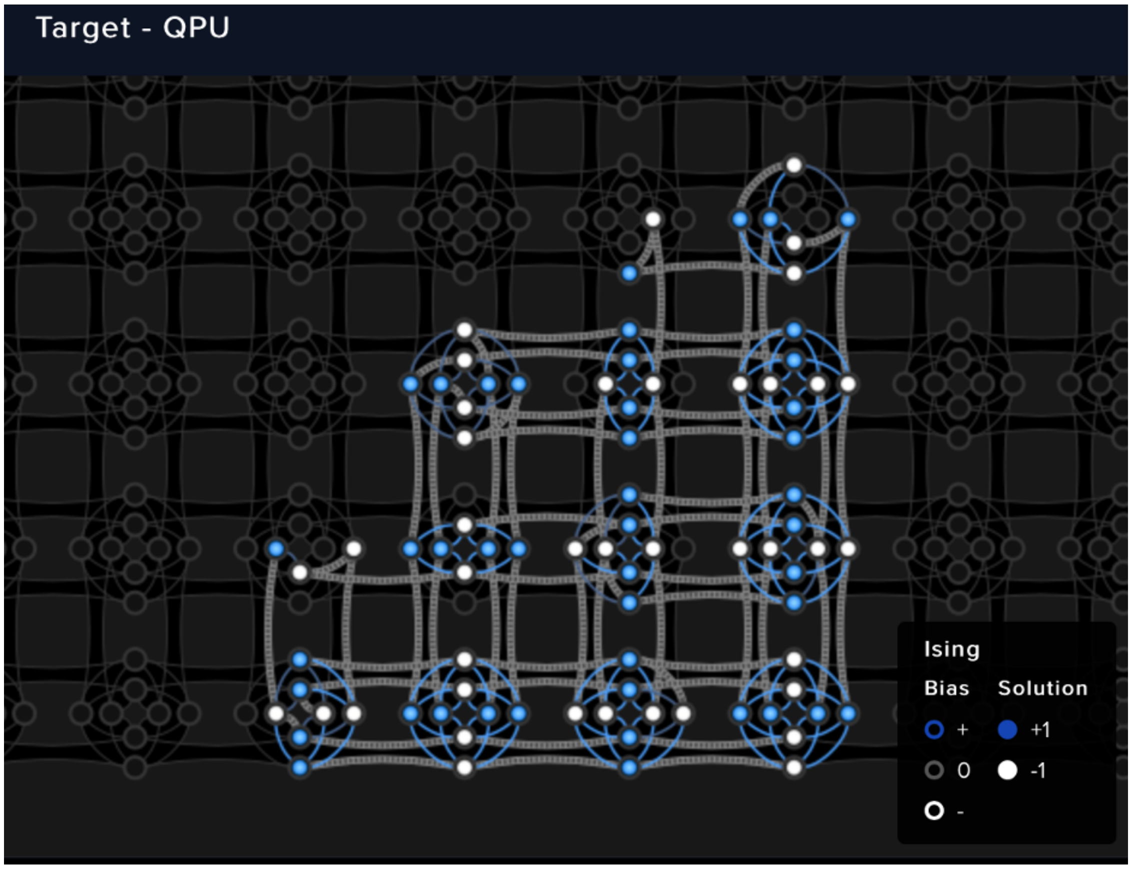
Here are our observations from these results:

- The partitions have equal size (8 nodes in each).
- There are 19 links between the partitions.
- There is one valid solution (0.1% for 1000 samples).
- All chains in the problem are broken. A broken chain is indicated by the white and orange zig-zag symbol on a problem variable.

Let's focus on the last observation. We discussed earlier that broken chains may be fixed by increasing the chain strength. First, we will set up the problem inspector to visualize the problem further.

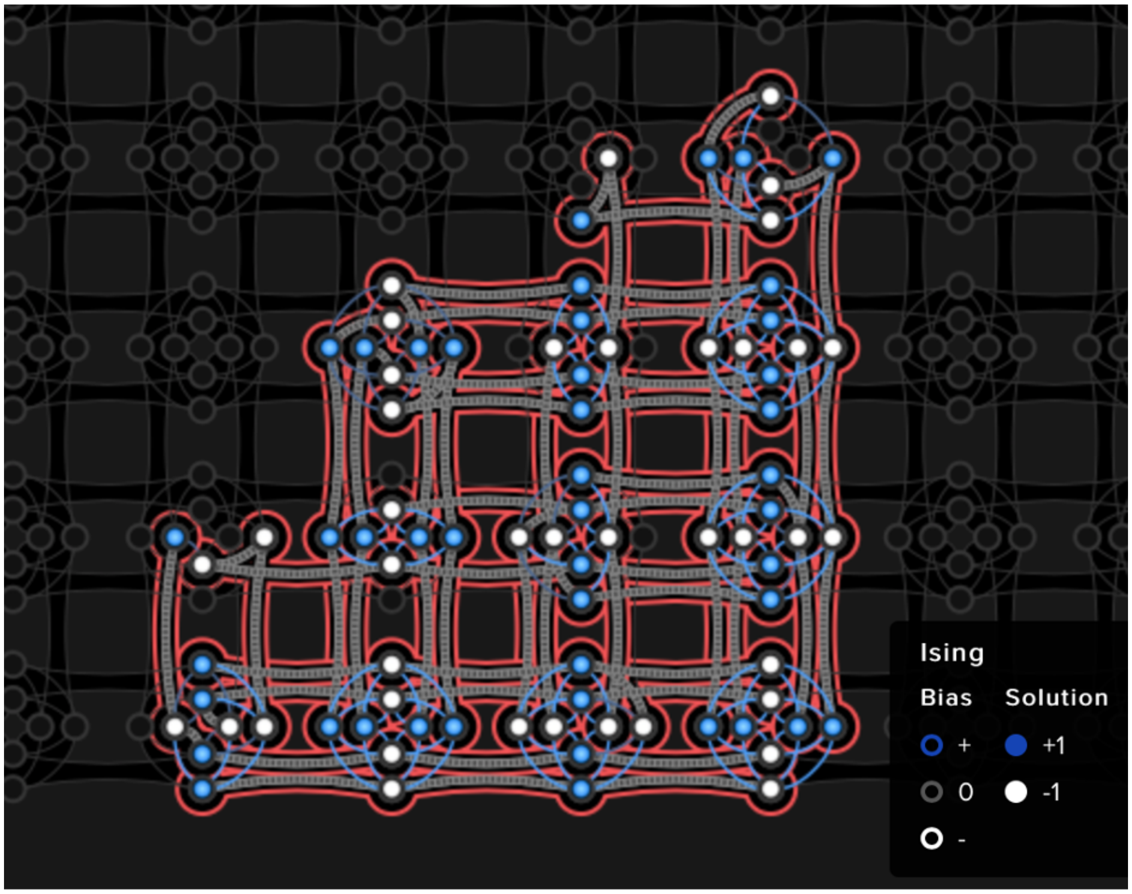
6 Using the Problem Inspector to Show Embedded Chains

The problem we chose is embedded onto the D-Wave QPU, and the problem inspector gives us a view of the embedded graph on the D-Wave QPU's topology:



Note the gray chains connecting groups of qubits. We want each chain to contain qubits that are the same color, that is, either all 0 or all 1. As discussed earlier, a chain is broken if it connects qubits of different colors.

By enabling another option in the problem inspector (on the toolbar on the left of the screen) to show broken chains in red, we can see that all of the chains are broken.



The way the code in this example works is as follows:

- Each logical variable – that is, each binary variable x_i in the original graph – is represented by a chain of physical qubits on the QPU.
- We want all the physical qubits in a given chain to have the same value, all 0 or all 1, at the end of each read. (annealing cycle) If they agree, then it is easy to map the qubit values back to the variable value in the original problem. If not, then postprocessing software (chain break fixing) is used to guess the correct assignment of the variable.
- The qubit chains are constrained to have the same value, 0 or 1, by a single parameter, `chain_strength`. This is the weight that is assigned to the gray edges.
- If the `chain_strength` parameter is too small, the physical qubits in the chain do not take the same value at the end of the annealing process, and the chain breaks.
- If the chains break, the solutions returned from the QPU may be degraded and sub-optimal.

We need to consider another question. If all the chains are broken in the solution, how could the solution be valid, with all the constraints satisfied, and the objective minimized?

The answer is that when we used the `FixedEmbeddingComposite` package in our code, it automatically used chain break fixing software to guess the correct assignment to the variables.

The default algorithm for chain break fixing is majority vote, and in this case, it worked well enough to find a valid solution.

In the next section, we will increase the `chain_strength` parameter to look for valid solutions without chain breaks.

7 Increasing the Chain Strength

We change the `chain_strength` parameter in the code:

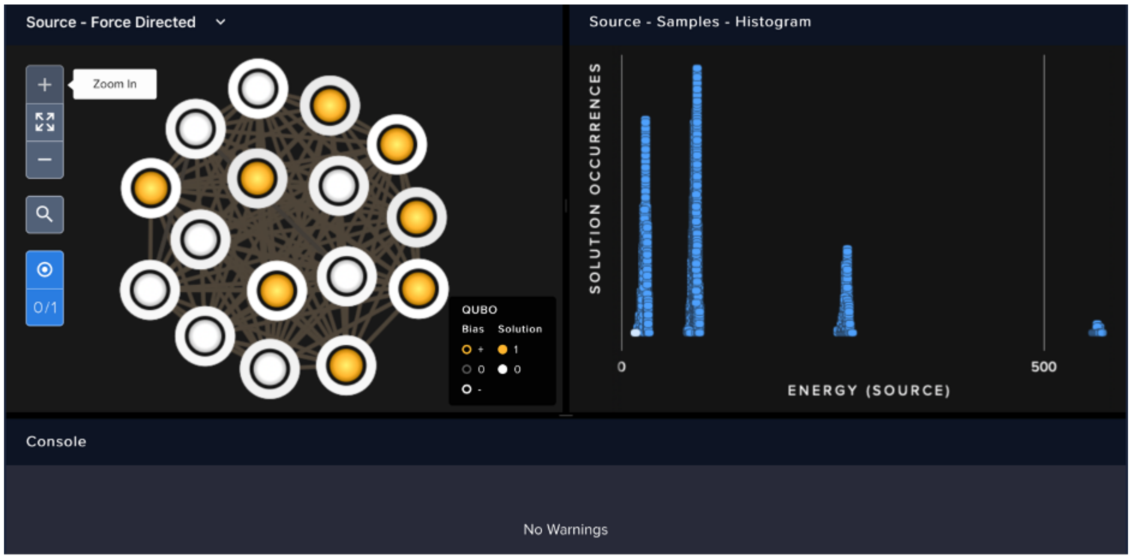
```
sampleset = dwave_sampler.sample(bqm,num_reads=1000,
    chain_strength=chain_strength_value)
```

How do we know what is a reasonable value for the `chain_strength` parameter? A good first estimate is to set it to equal to something near the largest absolute value in the problem's QUBO. In this graph partitioning problem, the QUBO entries range from -896 to about 120. Therefore, a good first guess for the `chain_strength` parameter is 1000.

8 Results at Chain Strength = 1000

You may have noticed that in our first program above, we didn't set the `chain_strength` parameter. The default value was therefore used, which is 1. Let's see what happens when we submit the problem with a larger value.

```
Number of nodes in one set is 8, in the other, 8.
The number of links between partitions is 16.0.
Percentage of valid solutions is 0.1.
Percentage of samples with high rates of breaks is 0.0.
```



No Warnings

We see a number of improvements:

- No broken chains
- No warnings
- We confirmed that the number of links between partitions is correct

The increased `chain_strength` parameter, in this case, has eliminated the problems that we saw earlier. We may have hoped for a larger percentage of valid solutions. However, it is not unusual in some problems that we do not get many valid solutions. At the higher `chain_strength`, though, we can be confident that the valid solution was not found fortuitously by chain break fixing as it was in Section 6.

9 What if the Chain Strength is too Large?

In the previous section, we saw that increasing the `chain_strength` parameter eliminated broken chains and provided a higher percentage of valid solutions to the problem. However, what would happen if the `chain_strength` is too large?

First, after the problem is submitted to the QPU, the autoscaling feature [12] scales all weights in a given QUBO so that everything lies in the range between [-1, +1]. If the `chain_strength` parameter is smaller than the largest QUBO weight, the autoscaling feature divides all terms by the largest QUBO weight. If it is larger than the largest QUBO weight, the autoscaling will divide by the value of the `chain_strength` parameter. In this manner, all terms in the QUBO, including the chain terms, will be in the range [-1, 1].

Thus, if chain strengths are larger than the largest absolute value in the QUBO, the chain strength-related terms are set to 1, and the QUBO weights are proportionally smaller. As chain strengths increase, the individual QUBO terms, which were introduced to express the terms we want to minimize, and the problem constraints, shrink to near zero. Each chain begins to act like a separate entity, and the QUBO approaches a problem of N independent variables that do not interact. Such a QUBO would have 2^N optimal solutions, all with the same energy. It no longer represents the original problem.

In this example problem, increasing the `chain_strength` parameter to 1000 does not show this behavior. Nonetheless, we should try to keep the `chain_strength` parameter within a reasonable range: large enough to avoid breaks, but small enough to maintain the importance of the QUBO terms.

References

1 *D-Wave system documentation: solver properties and parameters*, https://docs.dwavesys.com/docs/latest/doc_solver_ref.html (2020).

2 *Ocean documentation: samplers*, https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/samplers.html (2020).

3 *Ocean documentation*, https://docs.ocean.dwavesys.com (2020).

4 K. Boothby, P. Bunyk, J. Raymond, and A. Roy, *Next-generation topology of D-Wave quantum processors*, tech. rep. 14-1026A-C (D-Wave Technical Report Series, 2019).

5 Wikipedia contributors, *Random geometric graph — Wikipedia, the free encyclopedia*, [Online; accessed 17-September-2020], 2020.

6 A. Lucas, "Ising formulations of many np problems," Frontiers in Physics **2**, 5 (2014).

7 *Ocean documentation: using the problem inspector*, https://docs.ocean.dwavesys.com/en/latest/examples/inspector_graph_partitioning.html (2020).

8 J. Cai, W. G. Macready, and A. Roy, "A practical heuristic for finding graph minors," arXiv preprint arXiv:1406.2741 (2014).

9 *Ocean documentation: FixedEmbeddingComposite*, https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/composites.html#fixedembeddingcomposite (2020).

10 *Ocean documentation: minorminer*, https://docs.ocean.dwavesys.com/projects/minorminer/en/latest/ (2020).

11 *Ocean documentation: DWaveSampler.sample*, https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/generated/dwave.system.samplers.DWaveSampler.sample.html#dwave.system.samplers.DWaveSampler.sample (2020).

12 *D-Wave system documentation: auto-scale*, https://docs.dwavesys.com/docs/latest/c_solver_1.html#asc (2020).