The Quantum Computing Company™

Programming the D-Wave QPU: Setting the Chain Strength

WHITEPAPER

2020-04-14

Overview

We explore the importance of the chain strength in programming the D-Wave quantum processing unit (QPU). The need to adjust the chain strength may arise when minor embedding problems onto the D-Wave QPU hardware.

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

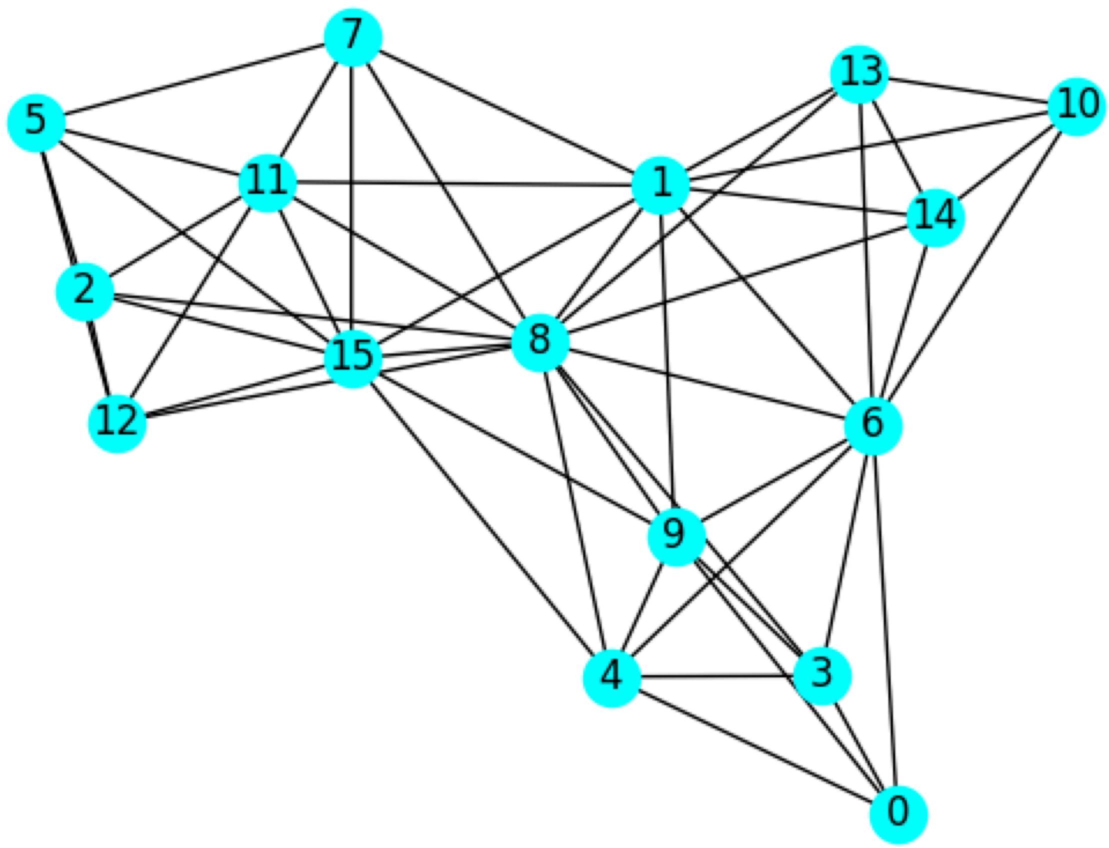
14-1041A-A

Notice and Disclaimer

Contents

1 Introduction

In this document, we choose a small problem to explore the importance of finding a good value for chain strength. We randomly place N = 16 objects into a square, and we 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** [1], and a typical random geometric graph looks like this:



Our goal is to divide the graph into two subsets (also known as **partitions**), which we want to have equal size, and we also want to minimize the number of links between the partitions. This problem is known as **graph partitioning**, and the general problem is NP-hard. [2]

To run this problem on the D-Wave quantum processing unit (QPU), we need to express the problem as a quadratic unconstrained binary optimization (QUBO) problem or Ising model. We choose the QUBO approach, and 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*, which 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 γ = 60 will be acceptable.

2 Writing an Initial Python Program for the QPU

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:

- Import the D-Wave problem inspector, so that we can graphically display the results.
- Arrange for the QPU to solve the problem, using the DWaveSampler object.
- Embed the problem [4] onto the graph structure of the QPU with FixedEmbeddingComposite, using an embedding that we found using a different Ocean package minorminer.
- Solve the problem, and run 1000 samples, using dwave_sampler.sample.
- Display the problem in the D-Wave problem inspector.

3 Running the program on the QPU

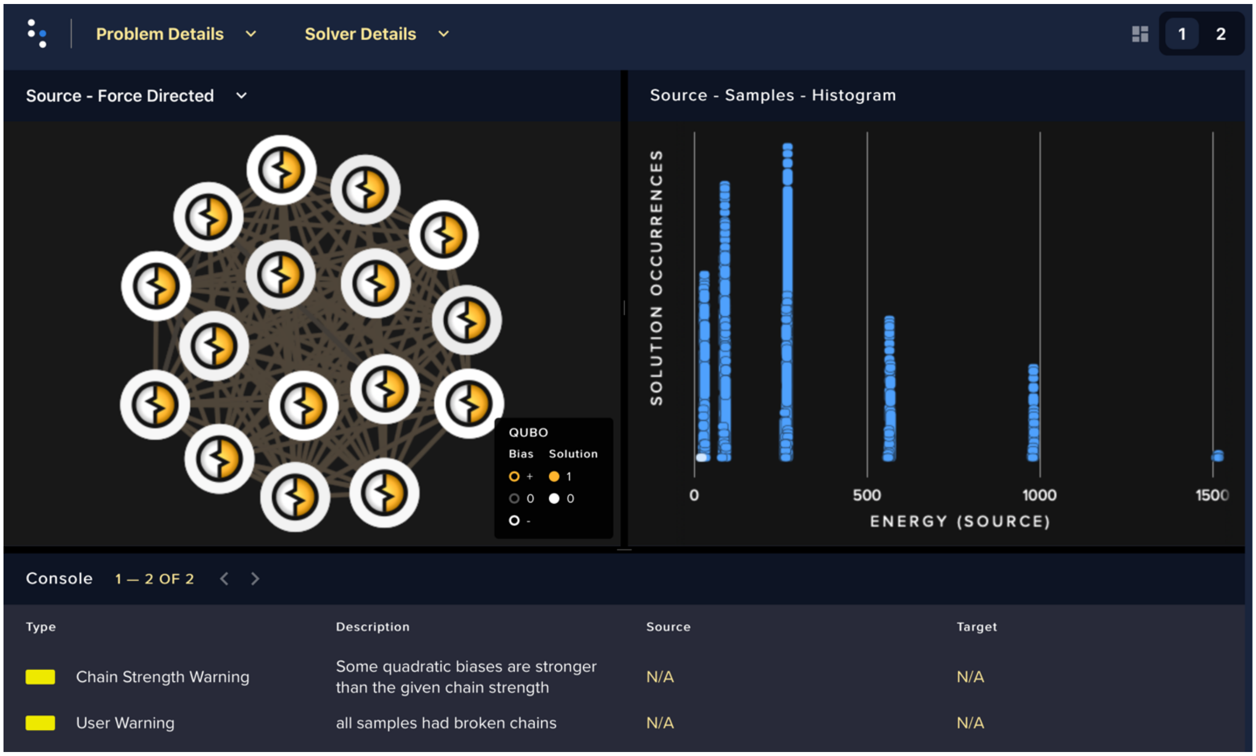
Here are the results from an initial run of the above Python program, including the command line output and the view from the problem inspector.

```
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.
```



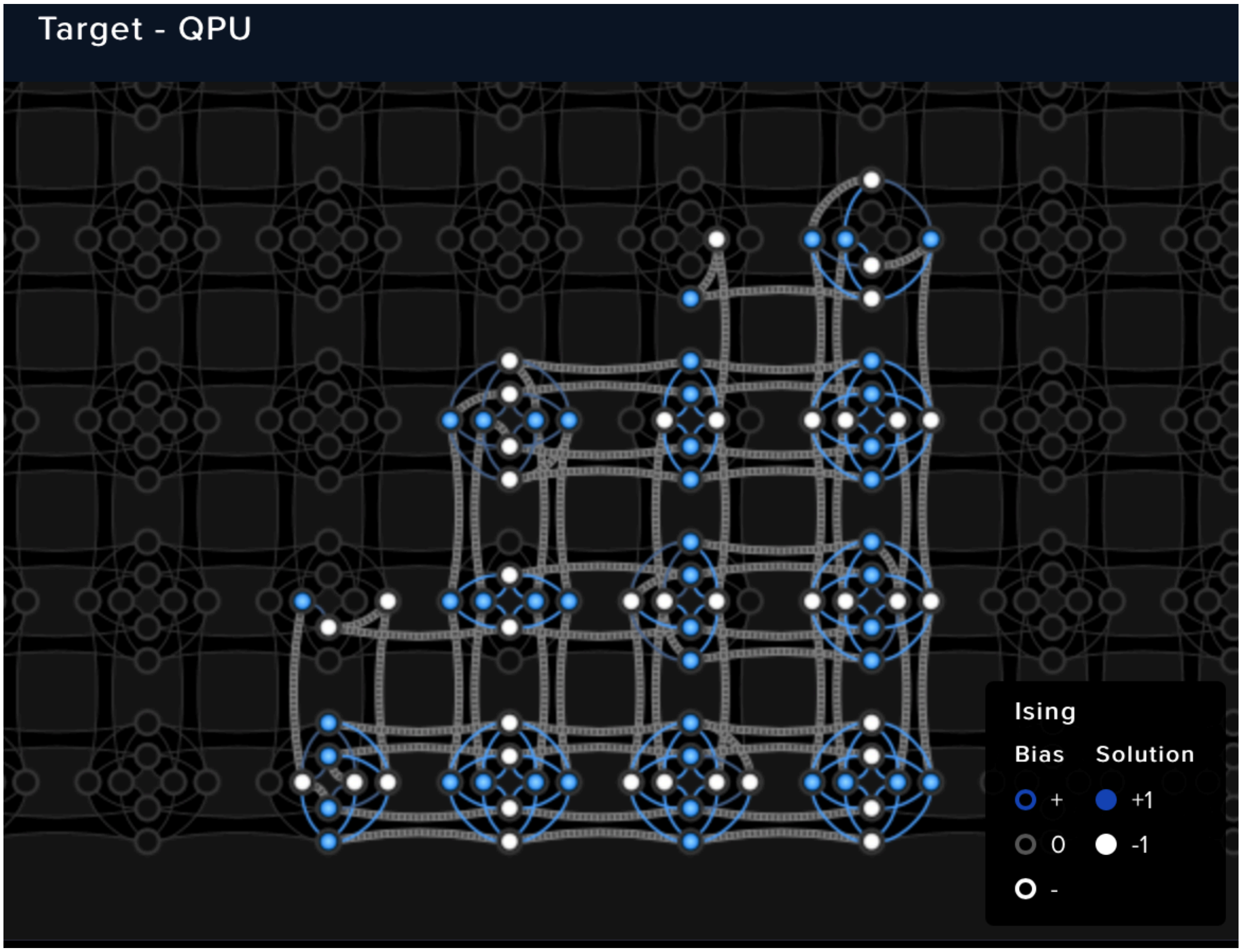
Here are our observations from these results:

- The partitions have equal size (8 nodes in each).
- There are 19 links between the partitions.
- There was 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.

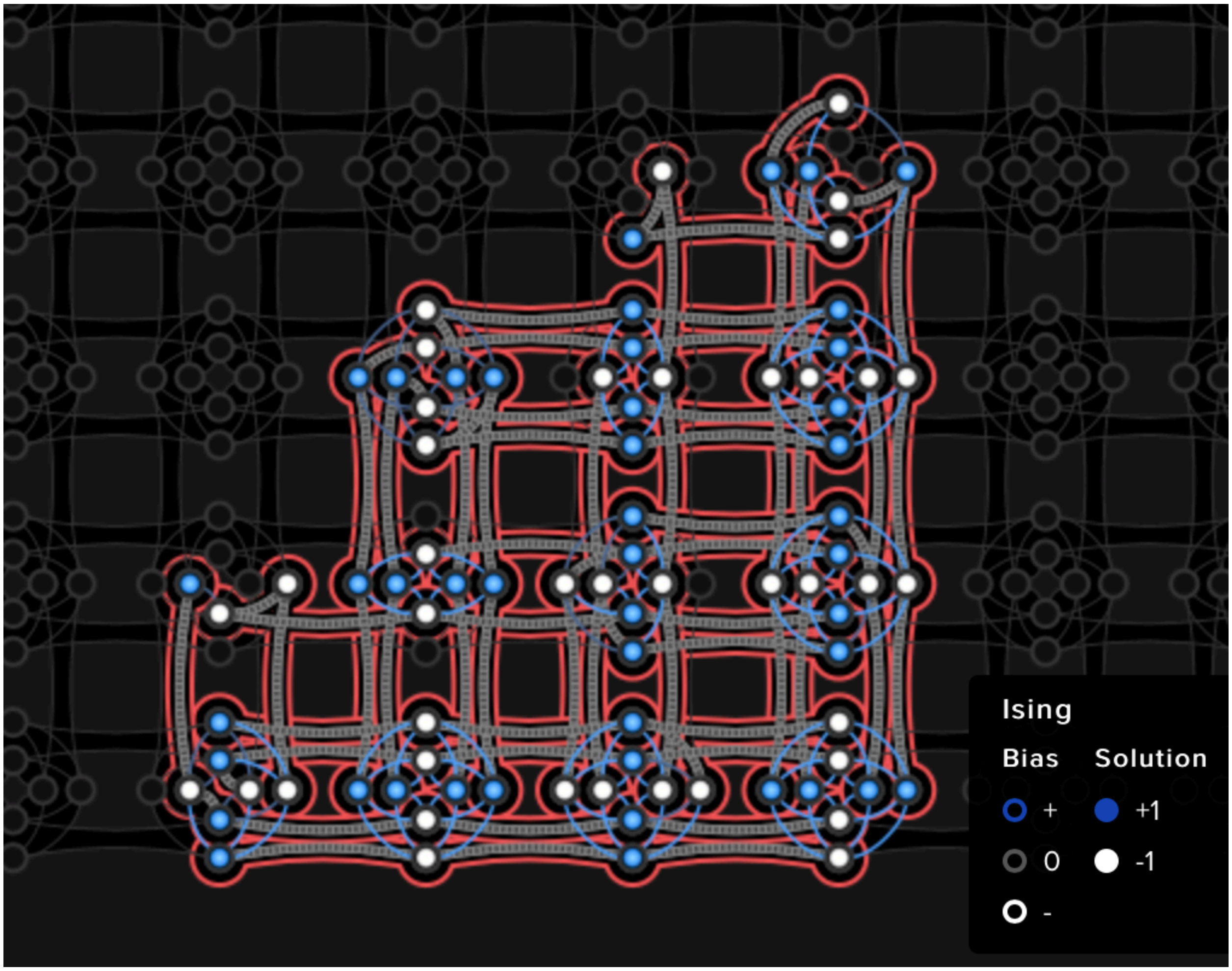
What does the last item mean, and what can we do about it?

4 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 all the same color, that is, either all 0 or all 1. 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 embedding 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 in 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 the 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 which is the correct assignment to the variable.
- The qubit chains are constrained to have the same value, 0 or 1, by a single parameter, the *chain strength*. This is the weight that is assigned to the gray edges.
- If the chain strength is not large enough, the physical qubits in the chain will not take the same value at the end of the annealing process, and the chain will break.
- If the chains break, the solutions returned from the processor may be degraded and are less likely to be optimal.

There is another question that we need to ask. 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 to look for valid solutions without chain breaks.

5 Increasing the Chain Strength

We change the chain strength in the code:

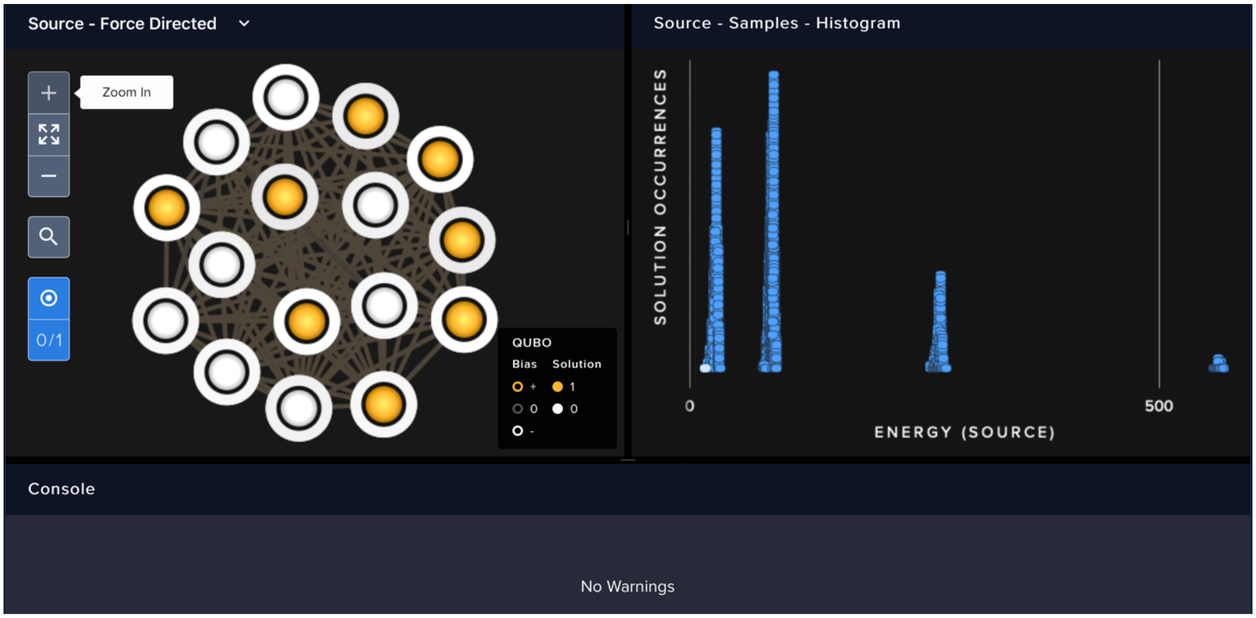
```
sampleset =
          dwave_sampler.sample(bqm,num_reads=1000,
                  chain_strength=chain_strength_value)
```

How do we know what is a reasonable value for the chain strength? A good first estimate is to set the chain strength 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 chain strength is 1000.

6 Results at Chain Strength = 1000

You may have noticed that in our first program, above, we didn't set the chain strength. The default value of chain strength was therefore used, which is 1. Let's see what happens when we submit the problem with increased chain strength:

```
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.
```



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, 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 by chain break fixing.

7 What if the Chain Strength is too Large?

In the previous section, we saw that increasing the chain strength eliminated broken chains and can provide a higher percentage of valid solutions to the problem. However, what would happen if the chain strength were too large?

The auto-scaling feature scales all weights in a given QUBO so that everything lies in the range between [-1, +1]. If chain strengths are larger than the largest absolute value in the QUBO, chain strengths are set to one, and the QUBO weights are proportionally smaller. As chain strengths get larger, 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 with each other. 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 chain strength to 1000 does not show this expected behavior. But we should try to keep chain strength within a reasonable range: large enough to avoid chain breaks, but small enough to maintain the importance of the QUBO terms.

References

1. Random geometric graph, https://en.wikipedia.org/wiki/Random_geometric_graph, accessed April 2020.
2. Andrew Lucas, Ising Formulations of many NP problems, Frontiers in Physics, 12 February 2014 | https://doi.org/10.3389/fphy.2014.00005
3. *D-Wave Ocean*, http://ocean.dwavesys.com, accessed March 2020.
4. Jun Cai, William G Macready, and Aidan Roy. A practical heuristic for finding graph minors. *arXiv preprint arXiv:1406.2741*, 2014.