D:Wave
The Quantum Computing Company™

Getting Started with D-Wave Solvers

USER MANUAL

2021-10-20

Overview

This document introduces the D-Wave quantum computer, describes the basics of how it works, and explains with simple examples how to use it to solve problems.

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

D-Wave User Manual 09-1076D-C

Notice and Disclaimer

D-Wave Systems Inc. (D-Wave), its subsidiaries and affiliates, makes commercially reasonable efforts to ensure that the information in this document is accurate and up to date, but errors may occur. NONE OF D-WAVE SYSTEMS INC., its subsidiaries and affiliates, OR ANY OF ITS RESPECTIVE DIRECTORS, EMPLOYEES, AGENTS, OR OTHER REPRESENTATIVES WILL BE LIABLE FOR DAMAGES, CLAIMS, EXPENSES OR OTHER COSTS (INCLUDING WITHOUT LIMITATION LEGAL FEES) ARISING OUT OF OR IN CONNECTION WITH THE USE OF THIS DOCUMENT OR ANY INFORMATION CONTAINED OR REFERRED TO IN IT. THIS IS A COMPREHENSIVE LIMITATION OF LIABILITY THAT APPLIES TO ALL DAMAGES OF ANY KIND, INCLUDING (WITHOUT LIMITATION) COMPENSATORY, DIRECT, INDIRECT, EXEMPLARY, PUNITIVE AND CONSEQUENTIAL DAMAGES, LOSS OF PROGRAMS OR DATA, INCOME OR PROFIT, LOSS OR DAMAGE TO PROPERTY, AND CLAIMS OF THIRD PARTIES.

D-Wave reserves the right to alter this document and other referenced documents without notice from time to time and at its sole discretion. D-Wave reserves its intellectual property rights in and to this document and its proprietary technology, including copyright, trademark rights, industrial design rights, and patent rights. D-Wave trademarks used herein include D-Wave®, Leap™ quantum cloud service, Ocean™, Advantage™ quantum system, D-Wave 2000Q™, D-Wave 2X™, and the D-Wave logos (the D-Wave Marks). Other marks used in this document are the property of their respective owners. D-Wave does not grant any license, assignment, or other grant of interest in or to the copyright of this document, the D-Wave Marks, any other marks used in this document, or any other intellectual property rights used or referred to herein, except as D-Wave may expressly provide in a written agreement. This document may refer to other documents, including documents subject to the rights of third parties. Nothing in this document constitutes a grant by D-Wave of any license, assignment, or any other interest in the copyright or other intellectual property rights of such other documents. Any use of such other documents is subject to the rights of D-Wave and/or any applicable third parties in those documents.

Contents

CHAPTER 1

WELCOME TO D-WAVE

I'm not happy with all the analyses that go with just the classical theory, because Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy.

It's not a Turing machine, but a machine of a different kind.

—Richard Feynman, 1981

1.1 What We Do

Despite the incredible power of today's supercomputers, many complex computing problems cannot be addressed by conventional systems. The huge growth of data and our need to better understand everything from the universe to our own DNA leads us to seek new tools that can help provide answers. Quantum computing is the next frontier in computing, providing an entirely new approach to solving the world's most difficult problems.

While certainly not easy, much progress has been made in the field of quantum computing since 1981, when Feynman gave his famous lecture at the California Institute of Technology. Still a relatively young field, quantum computing is complex and different approaches are being pursued around the world. Today, there are two leading candidate architectures for quantum computers: gate model (also known as circuit model) and quantum annealing.

Gate-model quantum computing implements compute algorithms with quantum gates, analogously to the use of Boolean gates in classical computers.

With quantum annealers you initialize the system in a low-energy state and gradually introduce the parameters of a problem you wish to solve. The slow change makes it likely that the system ends in a low-energy state of the problem, which corresponds to an optimal solution. This technique is explained in more detail in the *What is Quantum Annealing?* chapter.

Quantum annealing is implemented in D-Wave's generally available quantum computers, such as the Advantage™ and D-Wave 2000Q, as a single quantum algorithm, and this scalable approach to quantum computing has enabled us to create quantum processing units (QPUs) with more than 5000 quantum bits (*qubits*)—far beyond the state of the art for gate-model quantum computing.

D-Wave has been developing various generations of our "machine of a different kind," to use Feynman's words, since 1999. We are the world's first commercial quantum computer company.

1.2 D-Wave's Quantum Computer Systems

The D-Wave system contains a QPU that must be kept at a temperature near absolute zero and isolated from the surrounding environment in order to behave quantum mechanically. The system achieves these requirements as follows:

- Cryogenic temperatures, achieved using a closed-loop cryogenic dilution refrigerator system. The QPU operates at temperatures below 15 mK.
- Shielding from electromagnetic interference, achieved using a radio frequency (RF)–shielded enclosure and a magnetic shielding subsystem.

Figure 1.1: Advantage™ system.

The D-Wave QPU (Figure 1.2) is a lattice of tiny metal loops, each of which is a qubit or a coupler. Below temperatures of 9.2 kelvin, these loops become superconductors and exhibit quantum-mechanical effects.

The QPU in D-Wave's newest system, Advantage™, has more than 5,000 qubits and 35,000 couplers. To reach this scale, it uses over 1,000,000 Josephson junctions, which makes the Advantage QPU by far the most complex superconducting integrated circuit ever built.

For details on the topology of the QPU, see the *D-Wave QPU Architecture: Topologies* section.

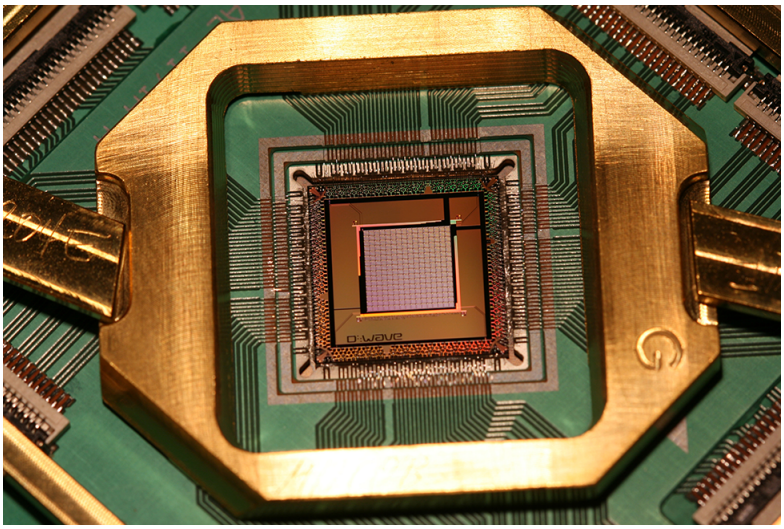


Figure 1.2: D-Wave QPU.

Note: For more details on the physical system, including specifications and essential safety information required for anyone who accesses the hardware directly, see the *D-Wave Quantum Computer Operations* manual, available from D-Wave.

1.3 D-Wave's Software Environment

Users interact with the D-Wave quantum computer through a web user interface (UI), and through open-source tools that communicate with the Solver API (SAPI).[1] The SAPI components are responsible for user interaction, user authentication, and work scheduling. In turn, SAPI connects to back-end servers that send problems to and return results from QPUs and additional solvers, which are located in different geographical regions (for example, North America or Europe).[2]

See Figure 1.3 for a simplified view of the D-Wave software environment.

[1] A *solver* is simply a resource that runs a problem. Some solvers interface to the QPU; others leverage CPU and GPU resources.

[2] Solvers are available by region. To view the supported regions and solvers that are available in each one, go to your **Leap Dashboard**.

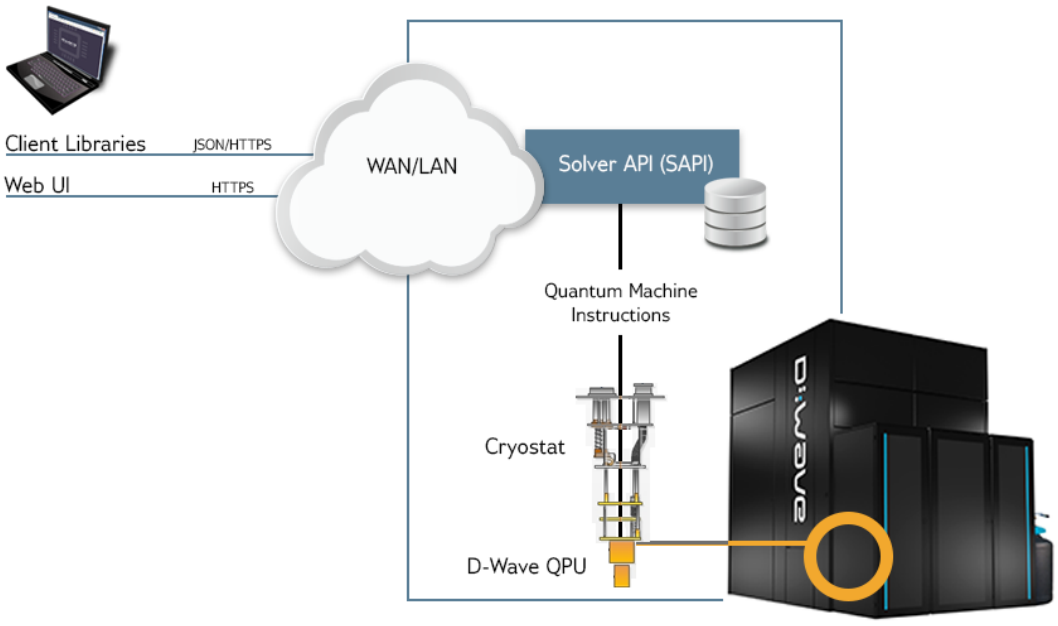


Figure 1.3: D-Wave software environment.

1.3.1 Leap™ Quantum Cloud Service

Leap™ is the quantum cloud service from D-Wave Systems.

Leap brings quantum computing to the real world by providing real-time cloud access to our systems. Through Leap, you can

- connect to D-Wave QPUs
- access *hybrid* solvers, which use a combination of classical and quantum resources, and can accept extremely large problems
- run demos and interactive coding examples
- contribute your ideas to our GitHub repositories of open-source code
- use the prebuilt cloud-based IDE
- join the growing conversation in our community of like-minded users
- administer projects, which are logical or organizational groupings of solvers and users

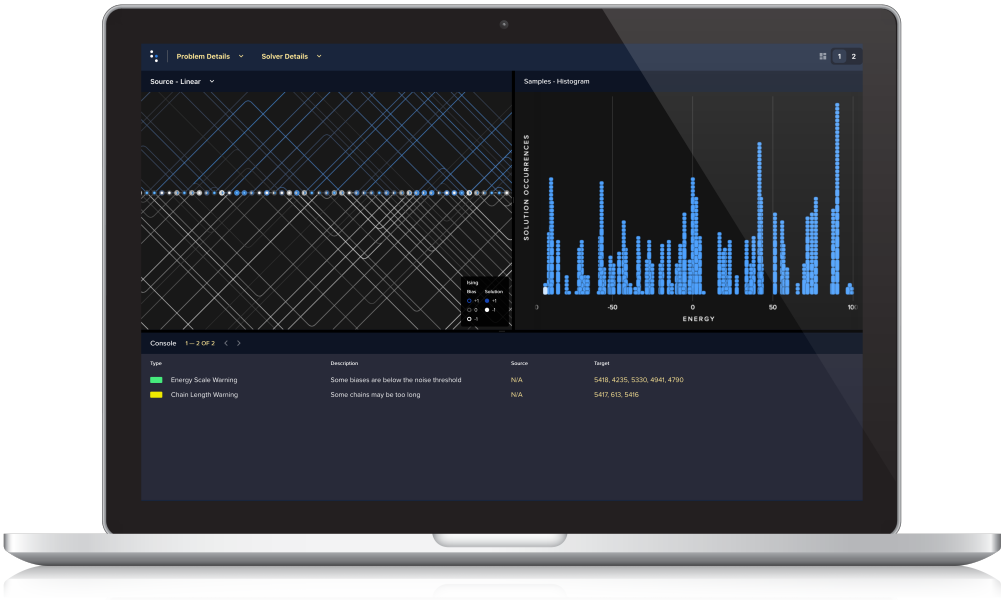
---

Note: To administer projects, you must be a project administrator.

Sign up for Leap here: Leap signup.

1.3.2 Ocean SDK

D-Wave's Python-based open-source software development kit (SDK), Ocean, makes application development for quantum computers rapid and efficient and facilitates collaborative projects. See Ocean SDK on GitHub to access the Ocean SDK, and Ocean documentation for the associated documentation.



CHAPTER 2

WORKFLOW: FORMULATION AND SAMPLING

This section provides a high-level description of how you solve problems using quantum computers.

The two main steps of solving problems using quantum computers are:

1. **Formulate your problem as an objective function**

   Objective (cost) functions are mathematical expressions of the problem to be optimized; for quantum computing, these are *quadratic models* that have lowest values (energy) for good solutions to the problems they represent.

2. **Find good solutions by sampling**

   Samplers are processes that sample from low-energy states of objective functions. Find good solutions by submitting your quadratic model to one of a variety of D-Wave's quantum, classical, and hybrid quantum-classical samplers.

Note: Samplers run—either remotely (for example, in D-Wave's Leap environment) or locally on your CPU—on compute resources known as *solvers*. (Note that some classical samplers actually brute-force solve small problems rather than sample, and these are also referred to as solvers.)

2.1 Objective Functions

To express a problem for a D-Wave solver in a form that enables solution by minimization, you need an *objective function*, a mathematical expression of the energy of a system. When the solver is a QPU, the energy is a function of binary variables representing its qubits; for quantum-classical hybrid solvers, the objective function might be more abstract.

For most problems, the lower the energy of the objective function, the better the solution. Sometimes any low-energy state is an acceptable solution to the original problem; for other problems, only optimal solutions are acceptable. The best solutions typically correspond to the *global minimum* energy in the solution space; see Figure 2.1.

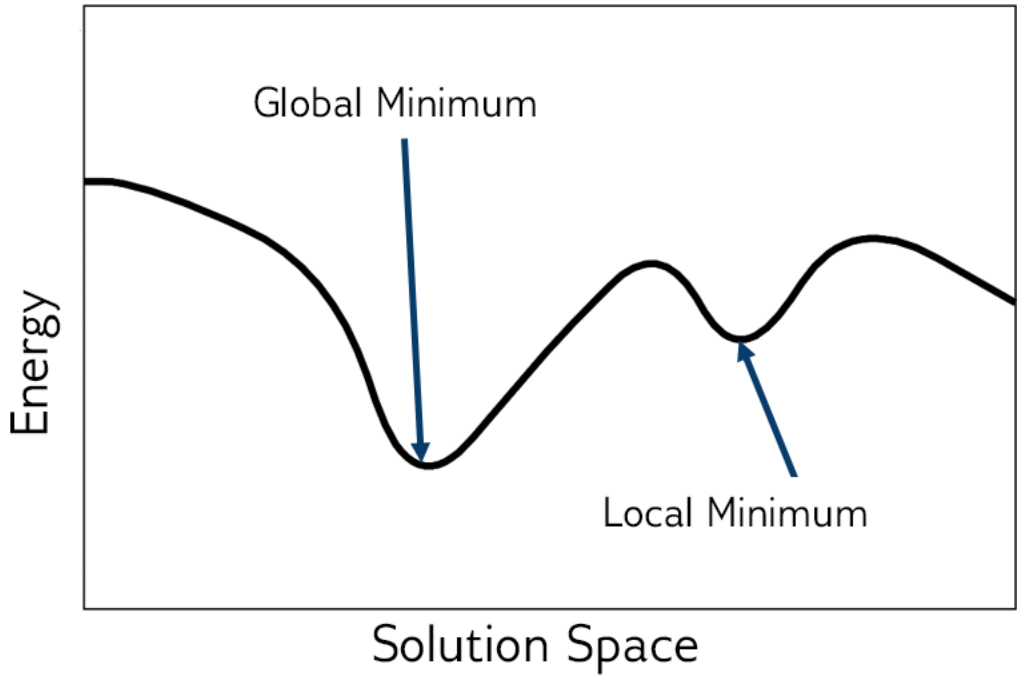


Figure 2.1: Energy of the objective function.

2.1.1 Simple Objective Example

As an illustrative example, consider solving a simple equation, $x + 1 = 2$, not by the standard algebraic techniques but by formulating an objective that at its minimum value assigns a value to the variable, x, that is also a good solution to the original equation.

Taking the square of the subtraction of one side from another, you can formulate the following objective function to minimize:

$$\begin{aligned} \mathrm{E}(x) &= [2 - (x + 1)]^2 \\ &= (1 - x)^2 \end{aligned}$$

In this case minimization, $\min_x (1 - x)^2$, seeks the shortest distance between the sides of the original equation, which occurs at equality (with the square eliminating negative distance).

The *Simple Sampling Example* example below shows an equally simple solution by sampling.

2.2 Quadratic Models

To express your problem as an objective function and submit to a D-Wave sampler for solution, you typically use one of the quadratic models[1] provided by Ocean software:

- Binary Quadratic Models are unconstrained[2] and have binary variables.

  BQMs are typically used for applications that optimize over decisions that could either be true (or yes) or false (no); for example, should an antenna transmit, or did a network node experience failure?

- Constrained Quadratic Models can be constrained and have integer and binary variables.

  CQMs are typically used for applications that optimize problems that might include integer and/or binary variables and one or more constraints.

- Discrete Quadratic Models are unconstrained and have discrete variables.

  DQMs are typically used for applications that optimize over several distinct options; for example, which shift should employee X work, or should the state on a map be colored red, blue, green or yellow?

2.3 Samplers

Samplers are processes that sample from low-energy states of objective functions. Having formulated an *objective function* that represents your problem (typically as a *quadratic model*), you sample it for solutions.

D-Wave provides quantum, classical, and quantum-classical hybrid samplers that run either remotely (for example, in D-Wave's Leap environment) or locally on your CPU.

- QPU Solvers

  D-Wave currently supports Advantage and D-Wave 2000Q quantum computers.

  This guide focuses on QPU solvers and provides examples of using quantum computers to solve problems.

- Quantum-Classical Hybrid Solvers

  Quantum-classical hybrid is the use of both classical and quantum resources to solve problems, exploiting the complementary strengths that each provides. For an overview of, and motivation for, hybrid computing, see: Medium Article on Hybrid Computing.

  D-Wave provides two types of hybrid solvers: Leap's hybrid solvers, which are cloud-based hybrid compute resources, and hybrid solvers developed in Ocean's

[1] Quadratic functions have one or two variables per term. A simple example of a quadratic function is,

$$D = Ax + By + Cxy$$

where A, B, and C are constants. Single variable terms—Ax and By here—are linear with the constant biasing the term's variable. Two-variable terms—Cxy here—are quadratic with a relationship between the variables.
Ocean also provides support for higher order models, which are typically reduced to quadratic for sampling.

[2] Constraints for such models are typically represented by adding penalty models to the objective, as shown in the *Constraints Example: Problem Formulation* section.

dwave-hybrid tool.

- Classical Solvers

  You might use a classical solver while developing your code or on a small version of your problem to verify your code.

  For information on classical solvers, see the Ocean SDK documentation.

2.3.1 Simple Sampling Example

As an illustrative example, consider solving by sampling the objective, $E(x) = (1-x)^2$ found in the *Simple Objective Example* example above to represent equation, $x + 1 = 2$.

This example creates a simple sampler that generates 10 random values of the variable x and selects the one that produces the lowest value of the objective:

```
>>> import random
>>> x = [random.uniform(-10, 10) for i in range(10)]
>>> e = list(map(lambda x: (1-x)**2, x))
>>> best_found = x[e.index(min(e))]
```

One particular execution found this best solution:

```
>>> print('x_i = ' + ' , '.join(f'{x_i:.2f}' for x_i in x))
x_i = 7.87, 1.79, 9.61, 2.37, 0.68, -2.93, 3.96, 1.30, -3.85, -0.13
>>> print('e_i = ' + ', '.join(f'{e_i:.2f}' for e_i in e))
e_i = 47.23, 0.63, 74.19, 1.89, 0.10, 15.44, 8.77, 0.09, 23.50, 1.28
>>> print("Best solution found is {:.2f}".format(best_found))
Best solution found is 1.30
```

Figure 2.2 shows the value of the objective function for the random values of x chosen in the execution above. The minimum distance between the sides of the original equation, which occurs at equality, has the lowest value (energy) of $E(x)$.

2.4 Simple Workflow Example

This example uses Ocean Software tools to demonstrate the solution workflow described in this section on a simple problem of finding the rectangle with the greatest area when the perimeter is limited.

In this example, the perimeter of the rectangle is set to 8 (meaning the largest area is for the 2*X*2 square).

A CQM is created that will have two integer variables, i, j, each limited to half the maximum perimeter length of 8, to represent the lengths of the rectangle's sides:

```
>>> from dimod import ConstrainedQuadraticModel, Integer
>>> i = Integer('i', upper_bound=4)
>>> j = Integer('j', upper_bound=4)
>>> cqm = ConstrainedQuadraticModel()
```

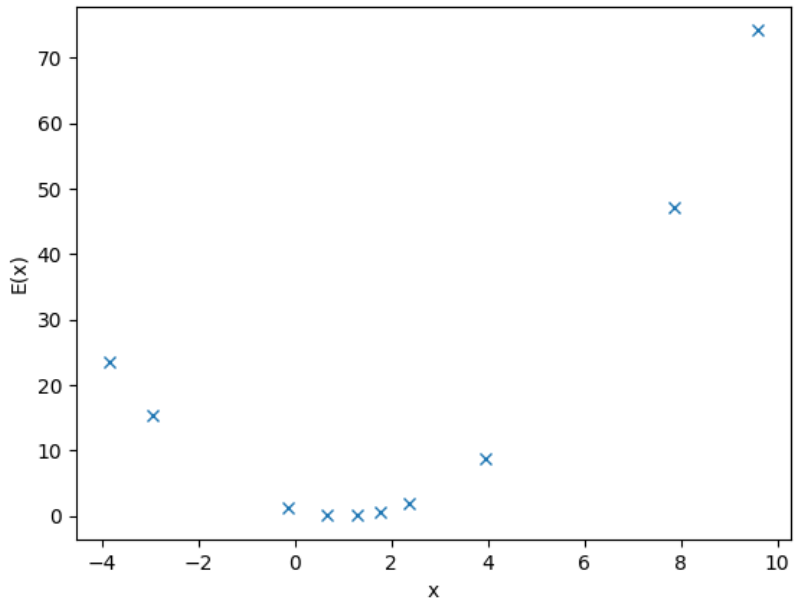


Figure 2.2: Values of the objective function, $\mathrm{E}(x) = (1 - x)^2$, versus random values of x selected by a simple random sampler.

The area of the rectangle is given by the multiplication of side i by side j. The goal is to maximize the area, $i * j$. Because D-Wave samplers minimize, the objective should have its lowest value when this goal is met. Objective $-i * j$ has its minimum value when $i * j$, the area, is greatest:

```
>>> cqm.set_objective(-i*j)
```

Finally, the requirement that the sum of both sides must not exceed the perimeter is represented as constraint $2i + 2j <= 8$:

```
>>> cqm.add_constraint(2*i+2*j <= 8, "Max perimeter")
'Max perimeter'
```

Instantiate a hybrid CQM sampler and submit the problem for solution by a remote solver provided by the Leap quantum cloud service:

```
>>> from dwave.system import LeapHybridCQMSampler
>>> sampler = LeapHybridCQMSampler()
>>> sampleset = sampler.sample_cqm(cqm)
>>> print(sampleset.first)
Sample(sample={'i': 2.0, 'j': 2.0}, energy=-4.0, num_occurrences=1,
...            is_feasible=True, is_satisfied=array([ True]))
```

The best (lowest-energy) solution found has $i = j = 2$ as expected, a solution that is feasible because all the constraints (one in this example) are satisfied.

2.5 Quantum Samplers: What and How?

The *Simple Workflow Example* example above uses a Leap hybrid CQM solver, which can accept integer variables and handle constraints natively. Hybrid samplers are also less restrictive on other aspects of the quadratic models they can accept; for example size (e.g., the number of variables) and structure (e.g., how interconnected the variables are).

This guide focuses on QPU solvers and provides examples of using quantum computers to solve problems:

- Chapter *What is Quantum Annealing?* explains the physics of how these quantum computers work as samplers.
- Chapter *D-Wave QPU Architecture: Topologies* describes QPU topology, which can be helpful when mapping problem variables to qubits.
- Chapter *Solving Problems with Quantum Samplers* describes the particular workflow used for QPU solvers.

The guide's other chapters walk you through simple examples of solving problems using this section's workflow. Although a QPU solver is used, the introduction provided is useful for classical and hybrid solvers.

CHAPTER 3

WHAT IS QUANTUM ANNEALING?

This section explains what quantum annealing is and how it works, and introduces the underlying quantum physics that governs its behavior. For more in-depth information on quantum annealing in D-Wave quantum computers, see *QPU Solver Datasheet*.

3.1 Applicable Problems

Quantum annealing processors naturally return low-energy solutions; some applications require the real minimum energy (optimization problems) and others require good low-energy samples (probabilistic sampling problems).

Optimization problems. In an optimization problem, you search for the best of many possible combinations. Optimization problems include scheduling challenges, such as "Should I ship this package on this truck or the next one?" or "What is the most efficient route a traveling salesperson should take to visit different cities?"

Physics can help solve these sorts of problems because you can frame them as energy minimization problems. A fundamental rule of physics is that everything tends to seek a minimum energy state. Objects slide down hills; hot things cool down over time. This behavior is also true in the world of quantum physics. Quantum annealing simply uses quantum physics to find low-energy states of a problem and therefore the optimal or near-optimal combination of elements.

Sampling problems. Sampling from many low-energy states and characterizing the shape of the energy landscape is useful for machine learning problems where you want to build a probabilistic model of reality. The samples give you information about the model state for a given set of parameters, which can then be used to improve the model.

Probabilistic models explicitly handle uncertainty by accounting for gaps in knowledge and errors in data sources. Probability distributions represent the unobserved quantities in a model (including noise effects) and how they relate to the data. The distribution of the data is approximated based on a finite set of samples. The model infers from the observed data, and learning occurs as it transforms the prior distribution, defined before observing the data, into the posterior distribution, defined afterward. If the training process is successful, the learned distribution resembles the distribution that generated the data, allowing predictions to be made on unobserved data. For example, when training on the famous MNIST dataset of handwritten digits, such a model can generate images resembling handwritten digits that are consistent with the training set.

Sampling from energy-based distributions is a computationally intensive task that is an excellent match for the way that the D-Wave quantum computer solves problems; that is,

by seeking low-energy states.

You can see a variety of example problems in the *D-Wave Problem-Solving Handbook* guide, in D-Wave's code examples repository on GitHub, and the many user-developed early quantum applications on D-Wave systems shown on the D-Wave website.

3.2 How Quantum Annealing Works in D-Wave QPUs

The quantum bits—also known as *qubits*—are the lowest energy states of the superconducting loops that make up the D-Wave QPU. These states have a circulating current and a corresponding magnetic field. As with classical bits, a qubit can be in state of 0 or 1; see Figure 3.1. But because the qubit is a quantum object, it can also be in a superposition of the 0 state and the 1 state at the same time. At the end of the quantum annealing process, each qubit collapses from a superposition state into either 0 or 1 (a classical state).

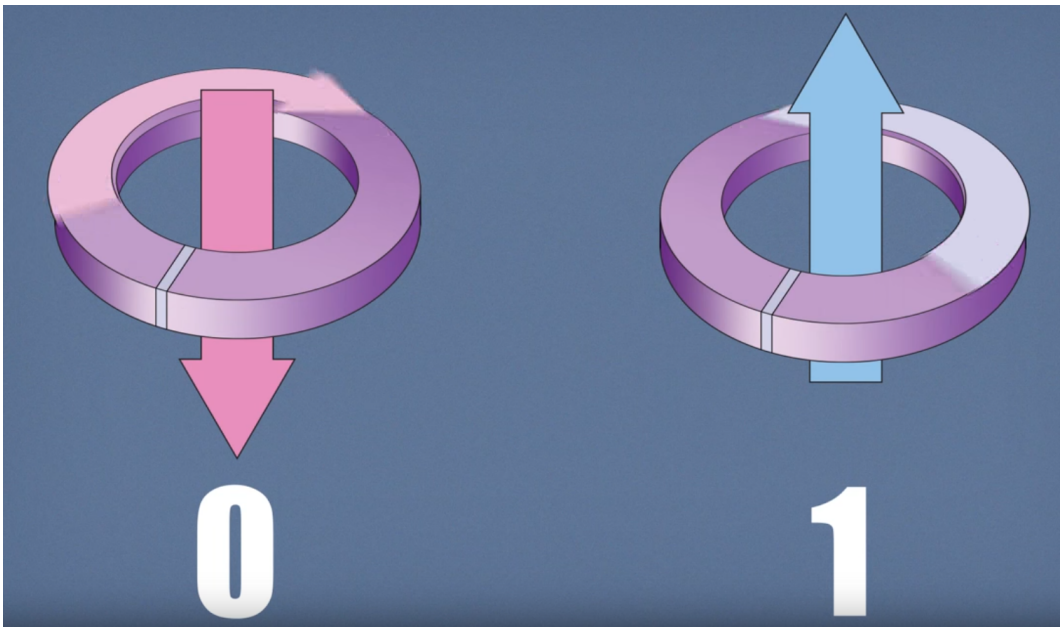


Figure 3.1: A qubit's state is implemented as a circulating current, shown clockwise for 0 and counter clockwise for 1, with a corresponding magnetic field.

The physics of this process can be shown (visualized) with an energy diagram as in Figure 3.2. This diagram changes over time, as shown in (a), (b), and (c). To begin, there is just one valley (a), with a single minimum. The quantum annealing process runs, the barrier is raised, and this turns the energy diagram into what is known as a *double-well potential* (b). Here, the low point of the left valley corresponds to the 0 state, and the low point of the right valley corresponds to the 1 state. The qubit ends up in one of these valleys at the end of the anneal.

Everything else being equal, the probability of the qubit ending in the 0 or the 1 state is equal (50 percent). You can, however, control the probability of it falling into the 0 or the

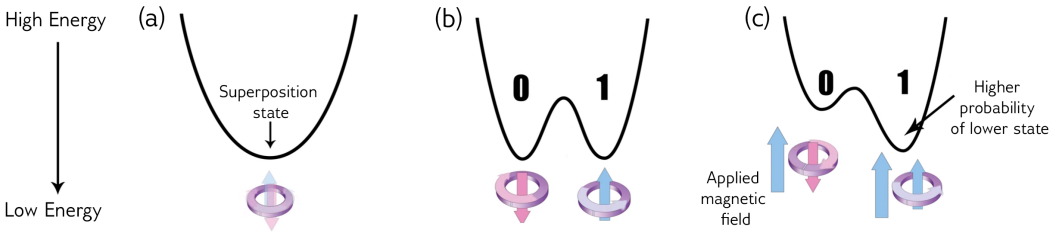


Figure 3.2: Energy diagram changes over time as the quantum annealing process runs and a bias is applied.

1 state by applying an external magnetic field to the qubit (c). This field tilts the double-well potential, increasing the probability of the qubit ending up in the lower well. The programmable quantity that controls the external magnetic field is called a *bias*, and the qubit minimizes its energy in the presence of the bias.

The bias term alone is not useful, however. The real power of the qubits comes when you link them together so they can influence each other. This is done with a device called a *coupler*. A coupler can make two qubits tend to end up in the same state—both 0 or both 1—or it can make them tend to be in opposite states. Like a qubit bias, the correlation weights between coupled qubits can be programmed by setting a coupling strength. Together, the programmable biases and weights are the means by which a problem is defined in the D-Wave quantum computer.

When you use a coupler, you are using another phenomenon of quantum physics called entanglement. When two qubits are entangled, they can be thought of as a single object with four possible states. Figure 3.3 illustrates this idea, showing a potential with four states, each corresponding to a different combination of the two qubits: (0,0), (0,1), (1,1), and (1,0). The relative energy of each state depends on the biases of qubits and the coupling between them. During the anneal, the qubit states are potentially delocalized in this landscape before finally settling into (1,1) at the end of the anneal.

As stated, each qubit has a bias and qubits interact via the couplers. When formulating a problem, users choose values for the biases and couplers. The biases and couplings define an energy landscape, and the D-Wave quantum computer finds the minimum energy of that landscape: this is quantum annealing.

Systems get increasingly complex as qubits are added: two qubits have four possible states over which to define an energy landscape; three qubits have eight. Each additional qubit doubles the number of states over which you can define the energy landscape: the number of states goes up exponentially with the number of qubits.

In summary, the systems starts with a set of qubits, each in a superposition state of 0 and 1. They are not yet coupled. When they undergo quantum annealing, the couplers and biases are introduced and the qubits become entangled. At this point, the system is in an entangled state of many possible answers. By the end of the anneal, each qubit is in a classical state that represents the minimum energy state of the problem, or one very close to it. All of this happens in D-Wave quantum computers in a matter of microseconds.

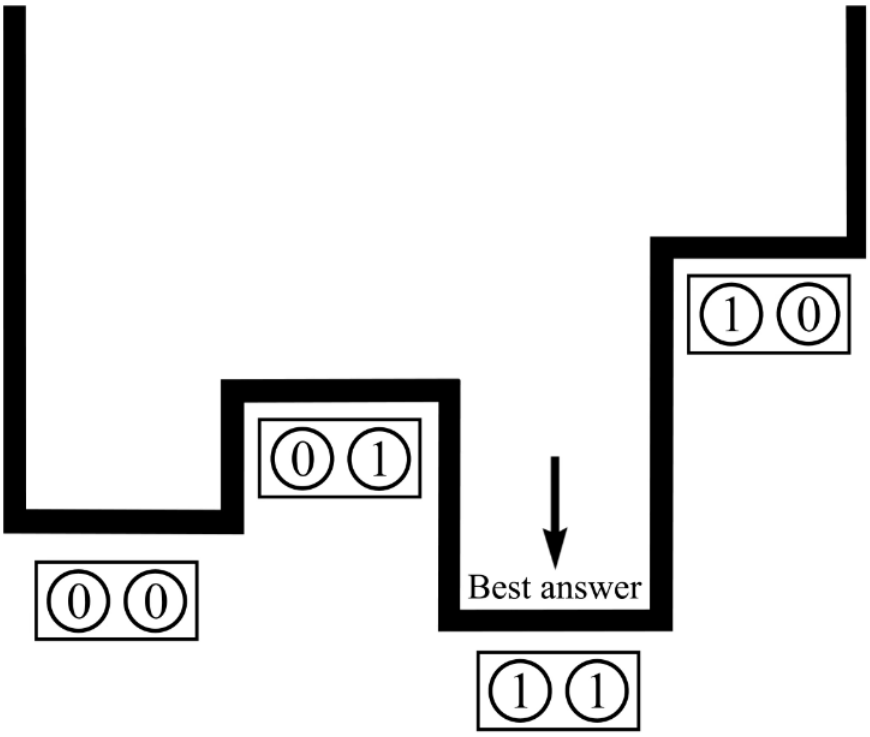


Figure 3.3: Energy diagram showing the best answer.

3.3 Underlying Quantum Physics

This section discusses some concepts essential to understanding the quantum physics that governs the D-Wave quantum annealing process.

3.3.1 The Hamiltonian and the Eigenspectrum

A classical Hamiltonian is a mathematical description of some physical system in terms of its energies. You can input any particular state of the system, and the Hamiltonian returns the energy for that state. For most non-convex Hamiltonians, finding the minimum energy state is an NP-hard problem that classical computers cannot solve efficiently.

As an example of a classical system, consider an extremely simple system of a table and an apple. This system has two possible states: the apple on the table, and the apple on the floor. The Hamiltonian tells you the energies, from which you can discern that the state with the apple on the table has a higher energy than that when the apple is on the floor.

For a quantum system, a Hamiltonian is a function that maps certain states, called *eigenstates*, to energies. Only when the system is in an eigenstate of the Hamiltonian is its energy well defined and called the *eigenenergy*. When the system is in any other state, its energy is uncertain. The collection of eigenstates with defined eigenenergies make up the *eigenspectrum*.

For the D-Wave quantum computer, the Hamiltonian may be represented as

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

where $\hat{\sigma}_{x,z}^{(i)}$ are Pauli matrices operating on a qubit q_i, and h_i and $J_{i,j}$ are the qubit biases and coupling strengths.[1]

The Hamiltonian is the sum of two terms, the *initial Hamiltonian* and the *final Hamiltonian*:

- Initial Hamiltonian (first term)—The lowest-energy state of the initial Hamiltonian is when all qubits are in a superposition state of 0 and 1. This term is also called the *tunneling Hamiltonian*.
- Final Hamiltonian (second term)—The lowest-energy state of the final Hamiltonian is the answer to the problem that you are trying to solve. The final state is a classical state, and includes the qubit biases and the couplings between qubits. This term is also called the *problem Hamiltonian*.

In quantum annealing, the system begins in the lowest-energy eigenstate of the initial Hamiltonian. As it anneals, it introduces the problem Hamiltonian, which contains the biases and couplers, and it reduces the influence of the initial Hamiltonian. At the end of the anneal, it is in an eigenstate of the problem Hamiltonian. Ideally, it has stayed in the minimum energy state throughout the quantum annealing process so that—by the end—it is in the minimum energy state of the problem Hamiltonian and therefore has an answer to the problem you want to solve. By the end of the anneal, each qubit is a classical object.

[1] Nonzero values of h_i and $J_{i,j}$ are limited to those available in the working graph; see the *D-Wave QPU Architecture: Topologies* chapter.

3.3.2 Annealing in Low-Energy States

A plot of the eigenenergies versus time is a useful way to visualize the quantum annealing process. The lowest energy state during the anneal—the *ground state*—is typically shown at the bottom, and any higher excited states are above it; see Figure 3.4.

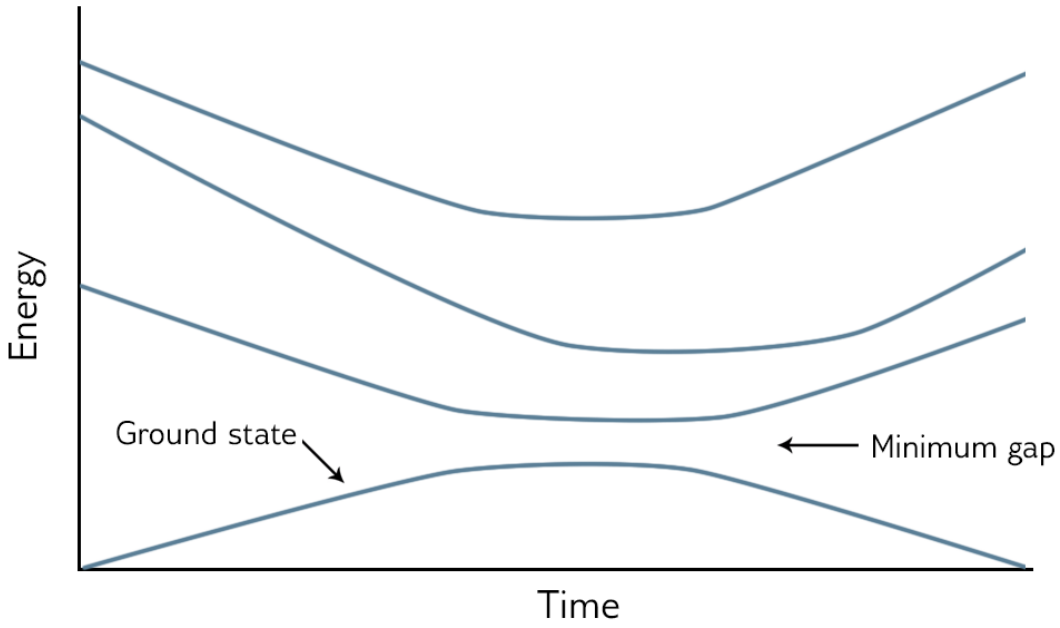


Figure 3.4: Eigenspectrum, where the ground state is at the bottom and the higher excited states are above.

As an anneal begins, the system starts in the lowest energy state, which is well separated from any other energy level. As the problem Hamiltonian is introduced, other energy levels may get closer to the ground state. The closer they get, the higher the probability that the system will jump from the lowest energy state into one of the excited states. There is a point during the anneal where the first excited state—that with the lowest energy apart from the ground state—approaches the ground state closely and then diverges away again. The minimum distance between the ground state and the first excited state throughout any point in the anneal is called the *minimum gap*.

Certain factors may cause the system to jump from the ground state into a higher energy state. One is thermal fluctuations that exist in any physical system. Another is running the annealing process too quickly. An annealing process that experiences no interference from outside energy sources and evolves the Hamiltonian slowly enough is called an *adiabatic* process, and this is where the name *adiabatic quantum computing* comes from. Because no real-world computation can run in perfect isolation, quantum annealing may be thought of as the real-world counterpart to adiabatic quantum computing, a theoretical ideal. In reality, for some problems, the probability of staying in the ground state can sometimes be small; however, the low-energy states that are returned are still very useful.

For every different problem that you specify, there is a different Hamiltonian and a different corresponding eigenspectrum. The most difficult problems, in terms of quantum annealing, are generally those with the smallest minimum gaps.

3.3.3 Evolution of Energy States

Figure 3.5 shows the dependence of the A and B parameters in the *Hamiltonian* on s, the normalized anneal fraction, an abstract parameter ranging from 0 to 1. The $A(s)$ curve is the tunneling energy and the $B(s)$ curve is the problem Hamiltonian energy at s. Both are expressed as energies in units of Joules as is standard for a Hamiltonian. A linear anneal sets $s = t/t_f$, where t is time and t_f is the total time of the anneal. At $t = 0$, $A(0) \gg B(0)$, which leads to the quantum ground state of the system where each spin is in a delocalized combination of its classical states. As the system is annealed, A decreases and B increases until t_f, when the final state of the qubits represents a low-energy solution.

At the end of the anneal, the Hamiltonian contains the only $B(s)$ term. It is a classical Hamiltonian where every possible classical bitstring (that is, list of qubit states that are either 0 or 1) corresponds to an eigenstate and the eigenenergy is the classical energy objective function you have input into the system.

3.4 Annealing Controls

D-Wave continues to pursue a deeper understanding of the fine details of quantum annealing and devise better controls for it. The quantum computer includes features that give users programmable control over the annealing schedule, which enable a variety of searches through the energy landscape. These controls can improve both optimization and sampling performance for certain types of problems, and can help investigate what is happening partway through the annealing process.

For more information about the available annealing controls, see *QPU Solver Datasheet*.

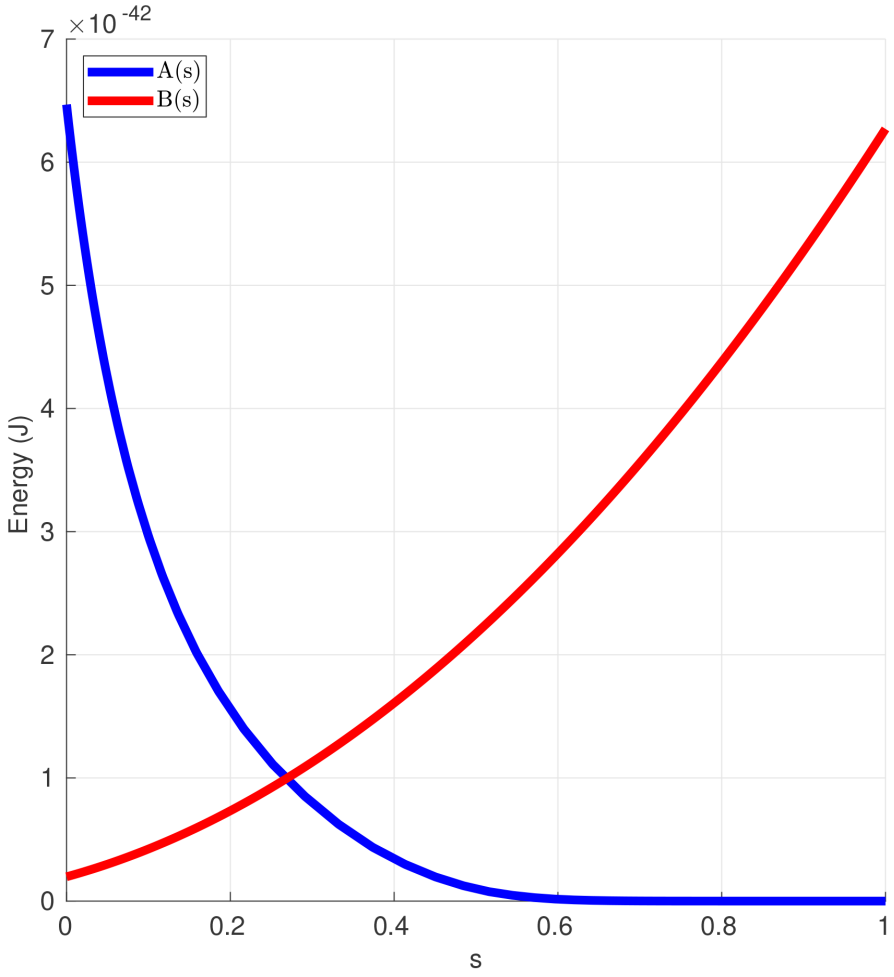


Figure 3.5: Annealing functions $A(s)$, $B(s)$. Annealing begins at $s = 0$ with $A(s) \gg B(s)$ and ends at $s = 1$ with $A(s) \ll B(s)$. Data shown are representative of D-Wave 2X systems.

CHAPTER 4

D-WAVE QPU ARCHITECTURE: TOPOLOGIES

The layout of the D-Wave QPU is critical to formulating an objective function in a format that a D-Wave quantum computer can solve; see the *Solving Problems with Quantum Samplers* section for more information.

Note: Although Ocean software automates the mapping from the linear and quadratic coefficients of a quadratic model to qubit bias and coupling values set on the QPU, you should understand it if you are using QPU solvers because it has implications for the problem-graph size and solution quality. If you are sending your problem to a Leap quantum-classical hybrid solver, the solver handles all interactions with the QPU.

The D-Wave quantum processing unit (QPU) is a lattice of interconnected qubits. While some qubits connect to others via couplers, the D-Wave QPU is not fully connected. Instead, the qubits of D-Wave 2000Q and earlier generations of QPUs interconnect in a topology known as *Chimera* while Advantage QPUs incorporate the *Pegasus* topology.

Some small number of qubits and couplers in a QPU may not meet the specifications to function as desired. These are therefore removed from the programmable fabric that users can access. The subset of the Pegasus or Chimera graph available to users is the *working graph*. The yield[1] of the working graph is the percentage of working qubits that are present.

4.1 Chimera Graph

In the D-Wave 2000Q and earlier systems, qubits are "oriented" on the QPU vertically or horizontally as shown in Figure 4.1.

For QPUs with the Chimera topology it is conceptually useful to categorize couplers as follows:

- **Internal couplers**.

  Internal couplers connect pairs of orthogonal (with opposite orientation) qubits as shown in Figure 4.2. The Chimera topology has a recurring structure of four horizontal qubits coupled to four vertical qubits in a $K_{4,4}$ bipartite graph, called a *unit*

[1] Manufacturing variations and the need to prepare the QPU to operate at cryogenic temperatures in a low–magnetic field environment limits the yield. These variations are minimized through an extensive calibration process that attempts to bring all of these analog devices into a consistent parametric regime.
For example, each D-Wave 2000Q QPU is fabricated with 2048 qubits and 6016 couplers in a Chimera topology. Of this total, the number and specific set of qubits and couplers that can be made available in the working graph changes with each system cooldown and calibration cycle. Calibrated commercial systems typically have more than 97% of fabricated qubits available in their working graphs.

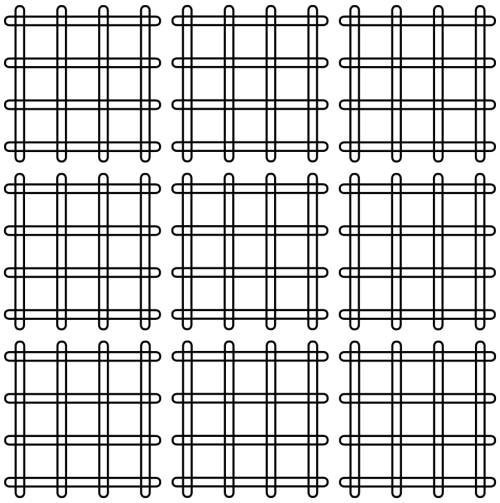


Figure 4.1: Qubits represented as horizontal and vertical loops. This graphic shows three rows of 12 vertical qubits and three columns of 12 horizontal qubits for a total of 72 qubits, 36 vertical and 36 horizontal.

cell.

A unit cell is typically rendered as either a cross or a column as shown in Figure 4.3.

- **External couplers**.

  External couplers connect colinear pairs of qubits—pairs of parallel qubits in the same row or column—as shown in Figure 4.4.

The $K_{4,4}$ unit cells formed by internal couplers are connected by external couplers as a lattice: this is the Chimera topology. Figure 4.5 shows two unit cells that form part of a larger Chimera graph.

Chimera qubits are characterized as having:

- nominal length 4—each qubit is connected to 4 orthogonal qubits through internal couplers
- degree 6—each qubit is coupled to 6 different qubits

The notation CN refers to a Chimera graph consisting of an NxN grid of unit cells. The D-Wave 2000Q QPU supports a C16 Chimera graph: its more than 2000 qubits are logically mapped into a 16x16 matrix of unit cells of 8 qubits. The 2x2 Chimera graph of Figure 4.2 is denoted C2.

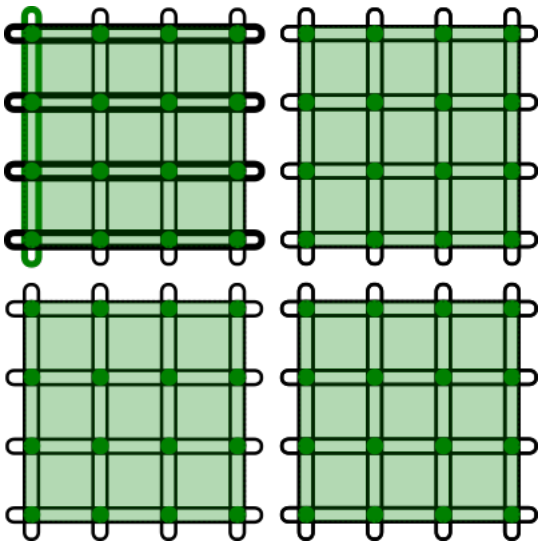


Figure 4.2: Green circles at the intersections of qubits signify internal couplers; for example, the upper leftmost vertical qubit, highlighted in green, internally couples to four horizontal qubits, shown bolded. The translucent green squares provide a helpful way to envision a recurring structure of this topology: a division of couplings into *unit cells* of $K_{4,4}$ bipartite graphs.

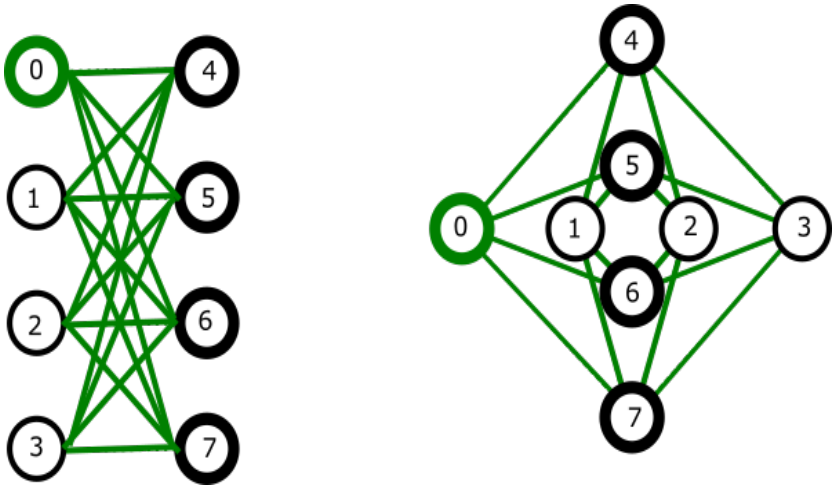


Figure 4.3: Chimera unit cell. In each of these renderings there are two sets of four qubits. Each qubit connects to all qubits in the other set but to none in its own, forming a $K_{4,4}$ graph; for example, the green qubit labeled 0 connects to bolded qubits 4 to 7.

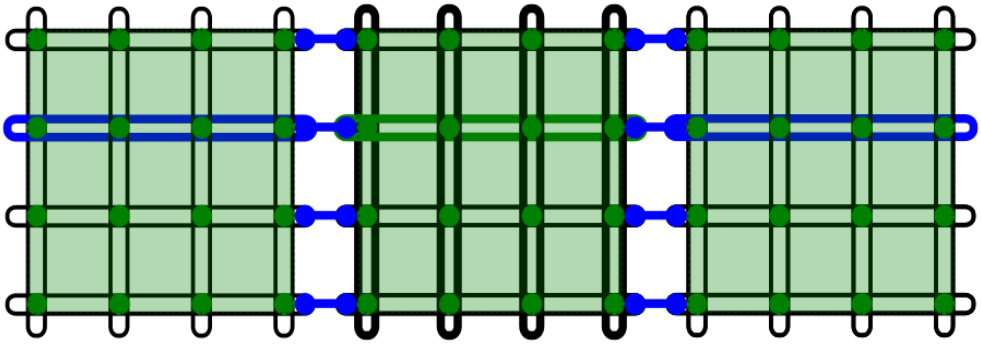


Figure 4.4: External couplers, shown as connected blue circles, couple vertical qubits to adjacent vertical qubits and horizontal qubits to adjacent horizontal qubits; for example, the green horizontal qubit in the center couples to the two blue horizontal qubits in adjacent unit cells. (It is also coupled to the bolded qubits in its own unit cell by internal couplers.)

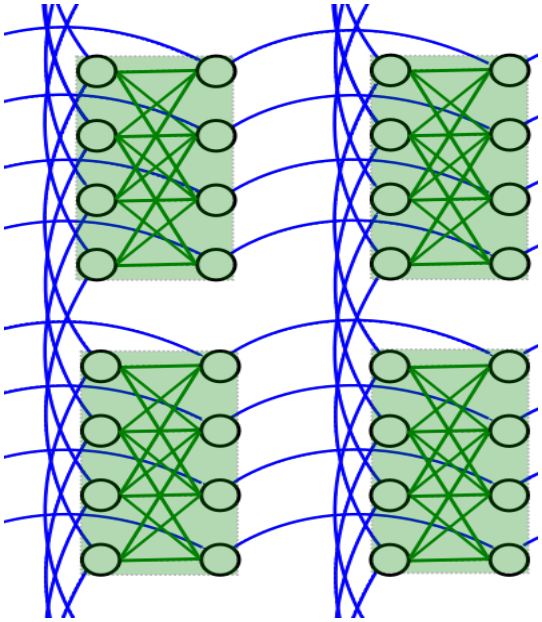


Figure 4.5: A cropped view of two unit cells of a Chimera graph. Qubits are arranged in 4 unit cells (translucent green squares) interconnected by external couplers (blue lines).

4.2 Pegasus Graph

In the Pegasus topology, qubits are "oriented" vertically or horizontally, as in Chimera, but similarly aligned qubits are also shifted, as illustrated in Figure 4.6.

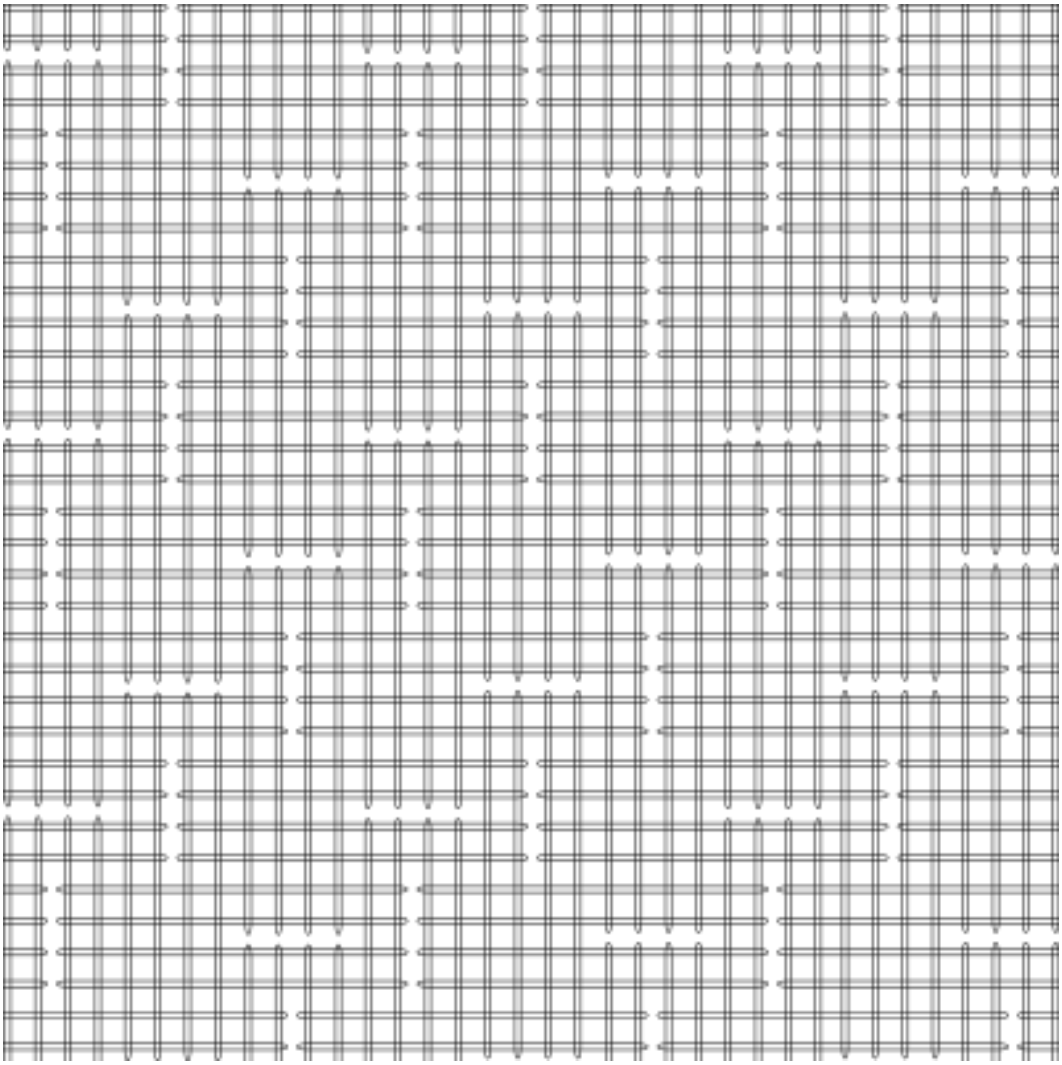


Figure 4.6: A cropped view of the Pegasus topology with qubits represented as horizontal and vertical loops. This graphic shows approximately three rows of 12 vertical qubits and three columns of 12 horizontal qubits for a total of 72 qubits, 36 vertical and 36 horizontal.

For QPUs with the Pegasus topology it is conceptually useful to categorize couplers as internal, external, and odd. Figure 4.7 and Figure 4.8 show two views of the coupling of qubits in this topology.

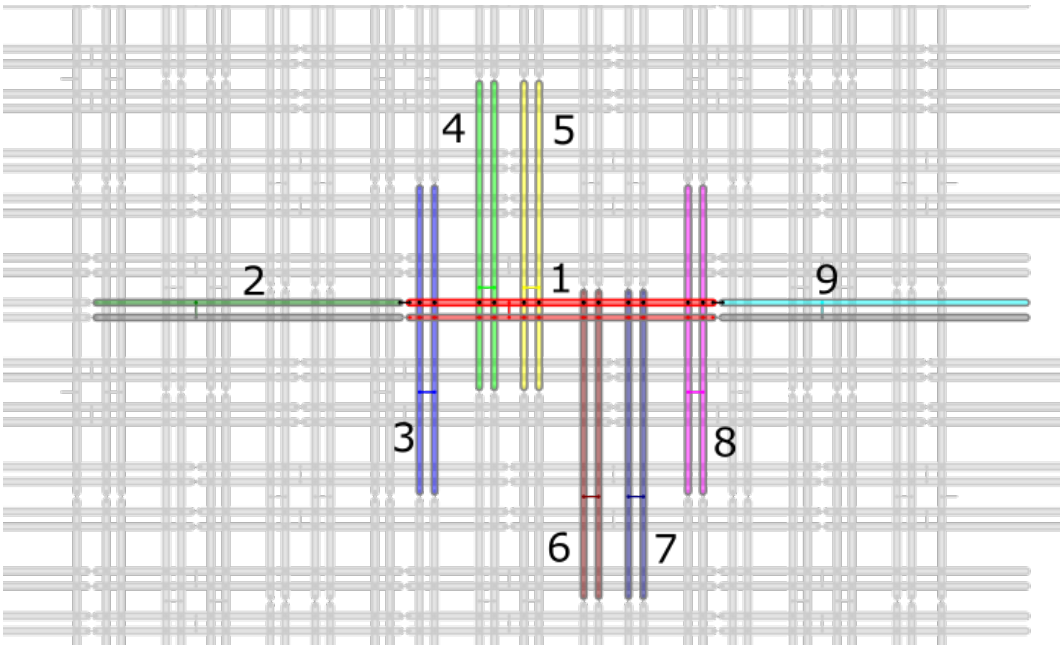


Figure 4.7: Coupled qubits (represented as horizontal and vertical loops): the horizontal qubit in the center, shown in red and numbered 1, with its odd coupler and paired qubit also in red, is internally coupled to vertical qubits, in pairs 3 through 8, each pair and its odd coupler shown in a different color, and externally coupled to horizontal qubits 2 and 9, each shown in a different color.

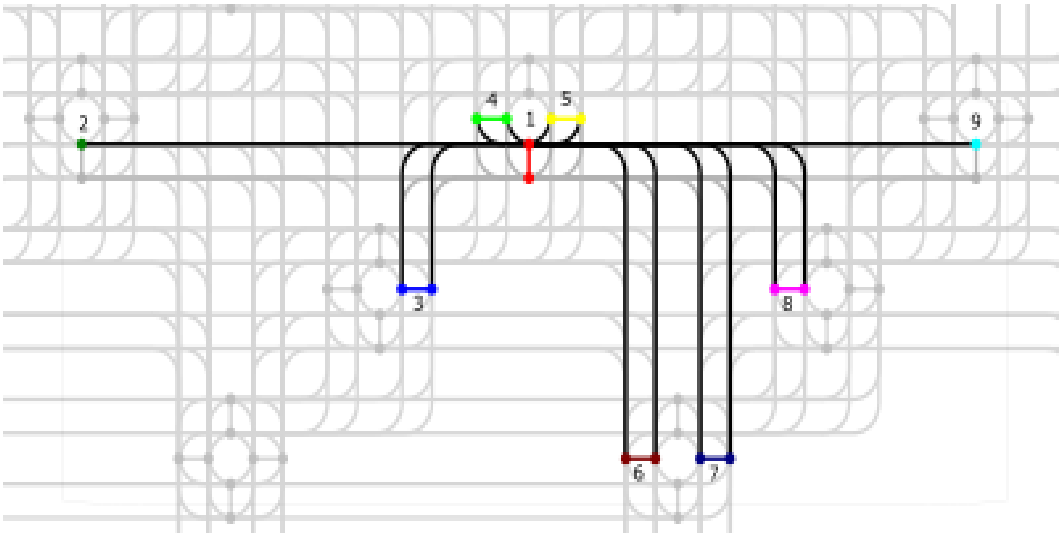


Figure 4.8: Coupled qubits "roadway" graphic (qubits represented as dots and couplers as lines): the qubit in the upper center, shown in red and numbered 1, is oddly coupled to the (red) qubit shown directly below it, internally coupled to vertical qubits, in pairs 3 through 8, each pair and its odd coupler shown in a different color, and externally coupled to horizontal qubits 2 and 9, each shown in a different color.

4.2.1 Pegasus Couplers

- **Internal couplers**.

  Internal couplers connect pairs of orthogonal (with opposite orientation) qubits as shown in Figure 4.9. Each qubit is connected via internal coupling to 12 other qubits.

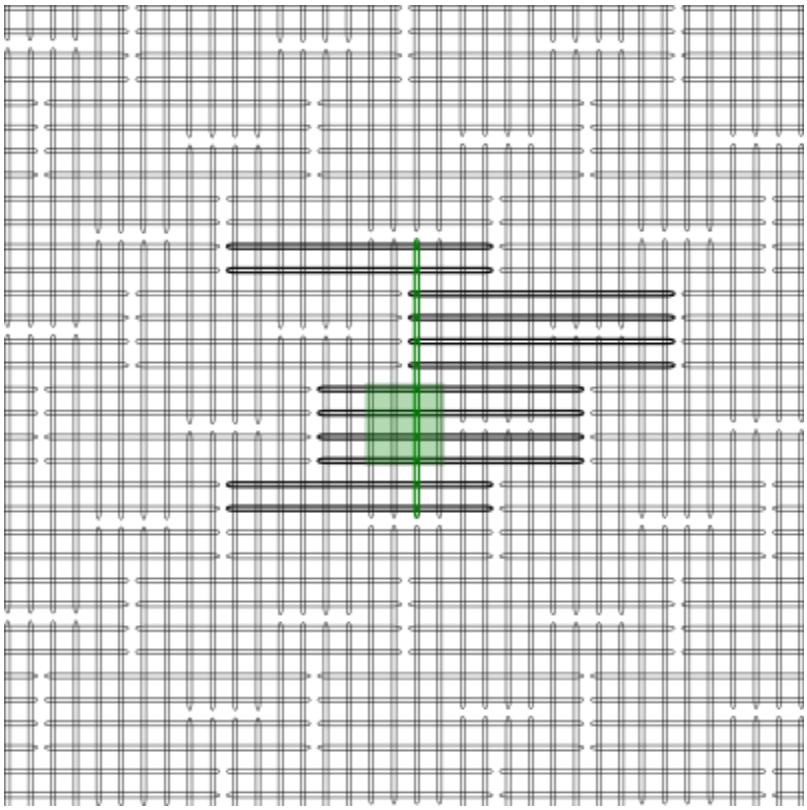


Figure 4.9: Junctions of horizontal and vertical loops signify internal couplers; for example, the green vertical qubit is coupled to 12 horizontal qubits, shown bolded. The translucent green square represents a Chimera unit cell structure (a $K_{4,4}$ bipartite graph of internal couplings).

- **External couplers**.

  External couplers connect vertical qubits to adjacent vertical qubits and horizontal qubits to adjacent horizontal qubits as shown in Figure 4.10.

- **Odd couplers**.

  Odd couplers connect similarly aligned pairs of qubits as shown in Figure 4.11.

Pegasus features qubits of degree 15 and native K_4 and $K_{6,6}$ subgraphs. Pegasus qubits are considered to have a nominal length of 12 (each qubit is connected to 12 orthogonal qubits through internal couplers) and degree of 15 (each qubit is coupled to 15 different qubits).

As the notation C_n refers to a Chimera graph with size parameter N, P_n refers to instances of Pegasus topologies; for example, P_3 is a graph with 144 nodes. A Pegasus unit cell contains twenty-four qubits, with each qubit coupled to one similarly aligned qubit in the cell and two similarly aligned qubits in adjacent cells, as shown in Figure 4.12. An Advantage QPU

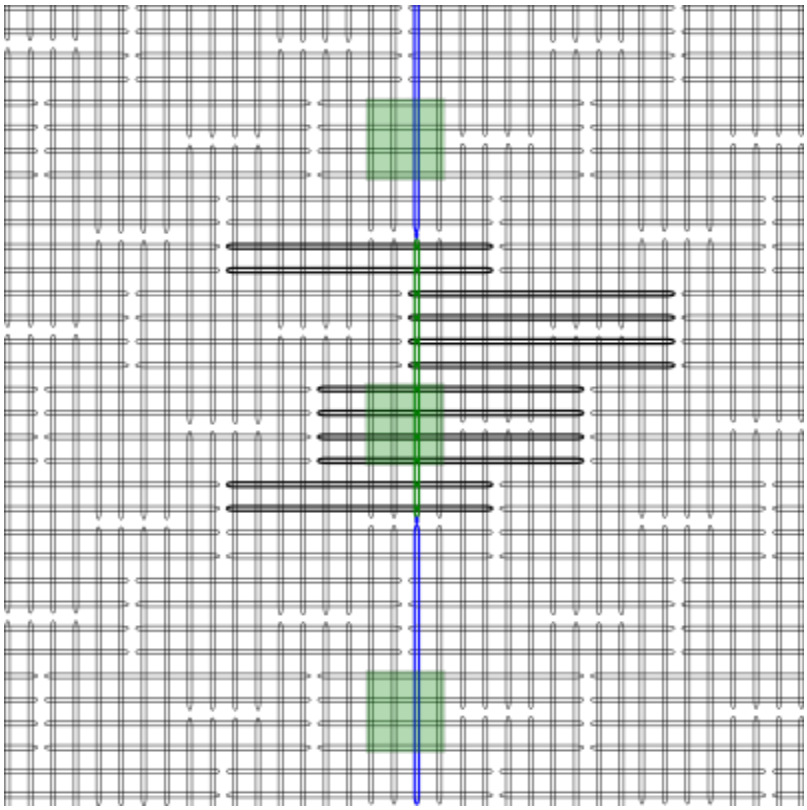


Figure 4.10: External couplers connect similarly aligned adjacent qubits; for example, the green vertical qubit is coupled to the two adjacent vertical qubits, highlighted in blue.

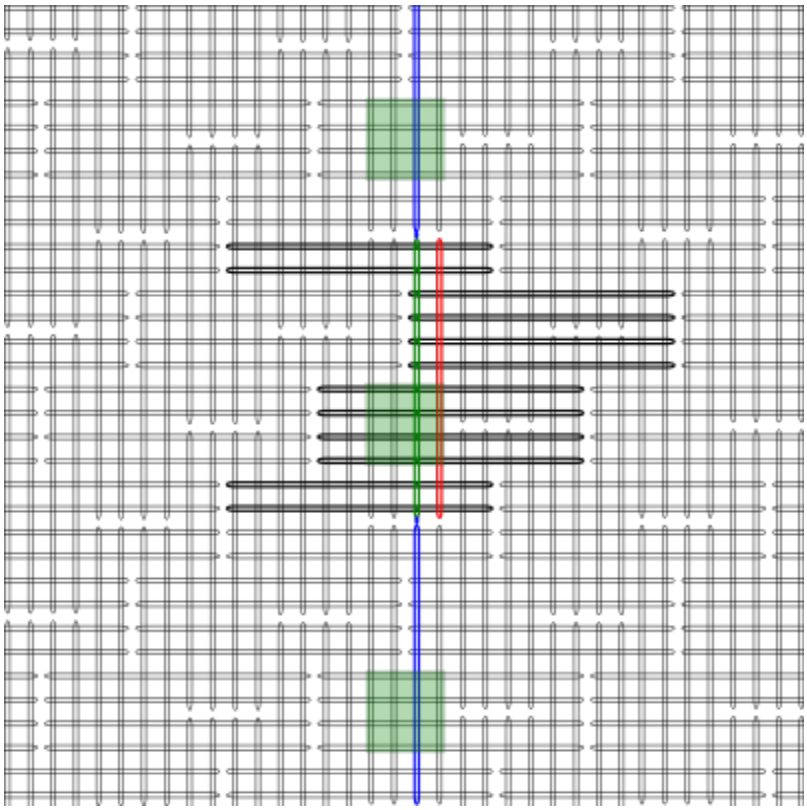


Figure 4.11: Odd couplers connect similarly aligned pairs of qubits; for example, the green vertical qubit is coupled to the red vertical qubit by an odd coupler.

is a lattice of 16x16 such unit cells, denoted as a P_{16} Pegasus graph.

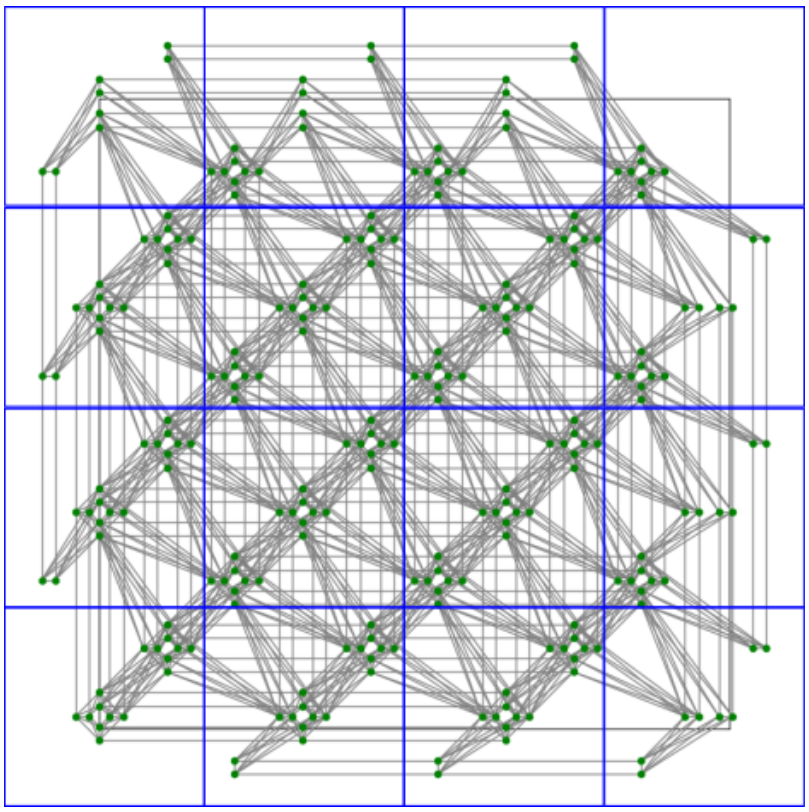


Figure 4.12: Pegasus unit cells in a P_4 graph, with qubits represented as green dots and couplers as gray lines.

More formally, the Pegasus unit cell consists of 48 halves of qubits that are divided between adjacent such unit cells, as shown in Figure 4.13.

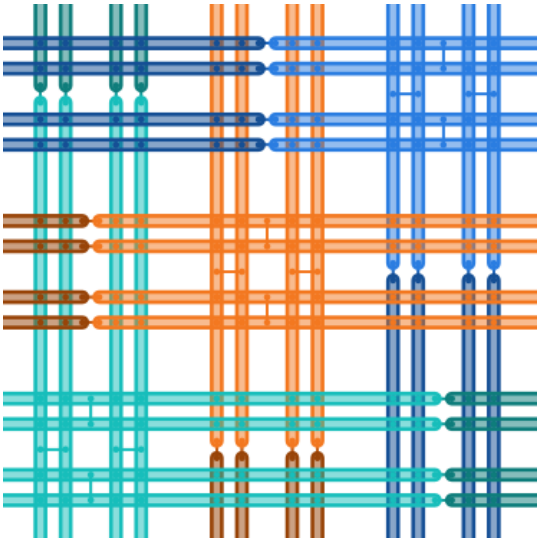


Figure 4.13: Pegasus unit cell shown as 48 halves of qubits from adjacent unit cells, with qubits represented as truncated loops (double lines), internal couplers as dots, and external and odd couplers as dots connected by short lines.

CHAPTER 5

SOLVING PROBLEMS WITH QUANTUM SAMPLERS

The *What is Quantum Annealing?* chapter explained how the D-Wave QPU uses quantum annealing to find the minimum of an energy landscape defined by the biases and couplings applied to its qubits in the form of a problem Hamiltonian. As described in the *Workflow: Formulation and Sampling* chapter, to solve a problem by sampling, you formulate an objective function such that when the solver finds its minimum, it is finding solutions to your problem.

This chapter shows how you formulate your objective as the problem Hamiltonian of a D-Wave quantum computer by defining the linear and quadratic coefficients of a binary quadratic model (BQM) that maps those values to the qubits and couplers of the QPU.

5.1 Binary Quadratic Models

For the QPU, two formulations for objective functions are the *Ising Model* and *QUBO*. Both these formulations are binary quadratic models and conversion between them is trivial[1].

5.1.1 Ising Model

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

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

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

[1] Chapter *Appendix: Next Learning Steps* provides information on the differences and conversion between the two formulations.

5.1.2 QUBO

QUBO problems are traditionally used in computer science, with variables taking values 1 (TRUE) and 0 (FALSE).

A QUBO problem is defined using an upper-diagonal matrix Q, which is an N x N upper-triangular matrix of real weights, and x, a vector of binary variables, as minimizing the function

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

where the diagonal terms $Q_{i,i}$ are the linear coefficients and the nonzero off-diagonal terms $Q_{i,j}$ are the quadratic coefficients.

This can be expressed more concisely as

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

In scalar notation, used throughout most of this document, the objective function expressed as a QUBO is as follows:

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

Note: Quadratic unconstrained binary optimization problems—QUBOs—are *unconstrained* in that there are no constraints on the variables other than those expressed in Q.

5.2 Minor Embedding

A graph comprises a collection of nodes and edges, which can be used to represent an objective function's variables and the connections between them, respectively.

For example, to represent a quadratic equation,

$$H(a, b) = 5a + 7ab - 3b,$$

you need two nodes, a and b, with biases of 5 and -3, and an edge between them with a strength of 7, as shown in Figure 5.1.

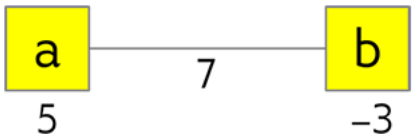


Figure 5.1: Two-variable objective function.

This graphic representation means you can map a BQM representing your objective function to the QPU:

- Nodes that represent the objective function's variables such as s_i (Ising) or q_i (QUBO) are mapped to qubits on the QPU.
- Edges that represent the objective function's quadratic coefficients such as $J_{i,j}$ (Ising) and $b_{i,j}$ (QUBO) are mapped to couplers.

The process of mapping variables in the problem formulation to qubits on the QPU is known as *minor embedding*.

The *Constraints Example: Minor-Embedding* chapter demonstrates minor embedding with an example; typically Ocean software handles it automatically.

5.3 Problem-Solving Process

In summary, to solve a problem on quantum samplers, you formulate the problem as an objective function, usually in Ising or QUBO format. Low energy states of the objective function represent good solutions to the problem. Because you can represent the objective function as a graph, you can map it to the QPU:[2] linear coefficients to qubit biases and quadratic coefficients to coupler strengths. The QPU uses quantum annealing to seek the minimum of the resulting energy landscape, which corresponds to the solution of your problem.

[2] Classical solvers might not require minor-embedding and quantum-classical hybrid solvers might embed parts of the problem on the QPU while solving other parts with classical algorithms on CPUs or GPUs.

CHAPTER 6

UNCONSTRAINED EXAMPLE: SOLVING A SAT PROBLEM

The D-Wave quantum computer is well suited to solving optimization and satisfiability (SAT) problems with binary variables. Binary variables can only have values 0 (NO, or FALSE) or 1 (YES, or TRUE), which are easily mapped to qubit final states of "spin up" (↑) and "spin down" (↓). These are problems that answer questions such as, "Should a package be shipped on this truck?"

This section uses an example problem of a very simple SAT:

$$(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2)$$

This is a 2-SAT in *conjunctive normal form* (CNF), meaning it has disjunctions of two Boolean variables (X OR Y) joined by an AND, and is satisfied only if all its disjunctions are satisfied; that is, to make the SAT true, you must find and assign appropriate values of 0 or 1 to all its variables.

You can easily verify that the solution to this example problem is $x_1 = x_2$. For example:

$$\begin{aligned} x_1 = x_2 = 0 \quad &\rightarrow \quad (0 \vee 1) \wedge (1 \vee 0) = 1 \wedge 1 = 1 \\ x_1{=}0 \neq x2{=}1 \quad &\rightarrow \quad (0 \vee 0) \wedge (1 \vee 1) = 0 \wedge 1 = 0 \end{aligned}$$

Note: The simple program below poses and solves this SAT in an alternative format familiar to anyone with some programming experience:

```
>>> import itertools
>>> def sat(x1, x2):
...     return (x1 or not x2) and (not x1 or x2)
>>> for (x1, x2) in list(itertools.product([False, True], repeat=2)):
...     print(x1, x2, '-->', sat(x1, x2))
False False --> True
False True --> False
True False --> False
True True --> True
```

6.1 Formulating an Objective Function

The first step in solving problems on QPU solvers is to formulate an objective function. Such an objective, usually in Ising or QUBO format, represents good solutions to the problem as low-energy states of the system. This subsection shows an intuitive approach to formulating such a QUBO.

For two variables, the *QUBO* formulation reduces to,

$$\mathrm{E}(a_i, b_{i,j}; q_i) = a_1 q_1 + a_2 q_2 + b_{1,2} q_1 q_2,$$

where a_1 and a_2, the linear coefficients, and $b_{1,2}$, the quadratic coefficient, are the programmable parameters you need to set so that q_1 and q_2, the binary variables (values of $\{0, 1\}$), represent solutions to your problem when the objective function is minimized. For this simple example, it's easy to work out values for these parameters.

A two-variable QUBO can have four possible values of its variables, representing four possible states of the system, as shown here:

State	q_1	q_2
1	0	0
2	0	1
3	1	0
4	1	1

For this objective function to represent the example SAT (i.e., to reformulate the original problem as an objective function that can be solved through minimization by a D-Wave solver), it needs to favor states 1 and 4 over states 2 and 3. You do this by *penalizing* states that do not satisfy the SAT; that is, by formulating the objective such that those states have higher energy.

First, notice that when q_1 and q_2 both equal 0—state 1—the value of the objective function is 0 for any value of the coefficients. To favor this state, you should formulate the objective function to have a global minimum energy (the *ground state* energy of the system) equal to 0. Doing so ensures that state 1 is a good solution.

Second, you penalize states 2 and 3 relative to state 1. One way to do this is to set both a_1 and a_2 to a positive value such as 0.1[1]. Doing so sets the the value of the objective function for those two states to 0.1.

Third, you also favor state 4 along with state 1. Given that for state 4, your objective function so far is

$$\mathrm{E}(a_i = 0.1, b_{i,j}; q_i = 1) = 0.2 + b_{1,2},$$

you can do this by setting the quadratic coefficient $b_{1,2} = -0.2$. The resulting objective function is

$$\mathrm{E}(q_i) = 0.1 q_1 + 0.1 q_2 - 0.2 q_1 q_2,$$

and the table of possible outcomes is shown below.

[1] Why not 0.2 or 0.5? The *Appendix: Next Learning Steps* chapter looks at scaling the problem values.

State	q_1	q_2	Objective Value
1	0	0	0
2	0	1	0.1
3	1	0	0.1
4	1	1	0

6.2 Minor Embedding

You can represent this QUBO as the graph shown in Figure 6.1.

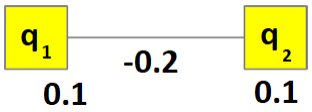


Figure 6.1: Objective function for the example SAT problem.

This graph can be mapped to any two QPU qubits with a shared coupler. The *Constraints Example: Problem Formulation* chapter shows minor-embedding for less simple graphs.

6.3 Solving on a QPU

To program a D-Wave quantum computer is to set values for its qubit biases and coupler strengths. Configuring qubit biases of 0.1 for the two qubits found by minor embedding and a strength of -0.2 for the shared coupler, and submitting to a QPU solver with a request for many anneals—also known as *samples* or *reads*—should strongly favor ground states 1 and 4 over excited states 2 and 3 in the returned results.

Below are results from running this problem on a D-Wave 2000Q system with the number of requested anneals set to 1000:

Energy	State	Occurrences
0	1	555
0	4	443
0.1	2	1
0.1	3	1

If you run this problem again, you can expect the numbers associated with energy 0 to vary, but to stay near the number 500 (50% of the samples). In a perfect system, neither of the ground states should dominant over the other in a statistical sense; however, each run yields different numbers.

Notice that calling the QPU enough times occasionally returns excited states 2 and 3.

The next example shows how, exactly, you submit your problem to a D-Wave solver.

CHAPTER 7

CONSTRAINTS EXAMPLE: PROBLEM FORMULATION

The example of the *Unconstrained Example: Solving a SAT Problem* chapter formulated an objective function for a simple SAT problem. As mentioned, the D-Wave quantum computer is also well suited to solving optimization problems with binary variables. Real-world optimization problems often come with *constraints*: conditions of the problem that any valid solution must satisfy.

For example, when optimizing a traveling salesperson's route through a series of cities, you need a constraint forcing the salesperson to be in exactly one city at each stage of the trip: a solution that puts the salesperson in two or more places at once is invalid.

Figure 7.1: The traveling salesperson problem is an optimization problem that can be solved using exactly-one-true constraints. Map data © 2017 GeoBasis-DE/BKG (© 2009), Google.

In fact, the example of the *Unconstrained Example: Solving a SAT Problem* chapter is actually also an example of a simple constraint, the equality (or XNOR) constraint. The example of this section is slightly more complex. The *exactly-one-true* constraint is the Boolean satisfiability problem of finding, given a set of variables, when exactly one variable is TRUE (equals 1). This chapter looks at an exactly-one-true constraint for three variables.

The problem-solving process is the same as described in the *Solving Problems with Quantum Samplers* chapter.

Note: Quantum samplers optimize objective functions that represent problems in formats that are *unconstrained* (e.g., QUBOs, or unconstrained binary optimization problems). As is shown in this chapter, any constraints in the original problem are represented as part of the objective function; this technique is known as penalty models.

Some Leap quantum-classical hybrid solvers accept only unconstrained objective functions; for example, hybrid BQM solvers. For those too any constraints must be added to the objective function, typically as a penalty. However, some Leap hybrid solvers can

handle constraints natively, as shown in the *Simple Workflow Example* example. For more information, see the Using Leap's Hybrid Solvers section.

7.1 Formulating an Objective Function

For problems with a small number of binary variables (in this example three: a, b, and c), you can tabulate all the possible permutations and identify the states when exactly one variable is 1 and the other two are 0. You do so with a truth table:

a	b	c	Exactly One is True
0	0	0	FALSE
1	0	0	TRUE
0	1	0	TRUE
1	1	0	FALSE
0	0	1	TRUE
1	0	1	FALSE
0	1	1	FALSE
1	1	1	FALSE

Notice that the three constraint-satisfying states in the truth table have in common that the sum of variables equals 1, so you can express the constraint mathematically as the equation,

$$a + b + c = 1.$$

As noted in the *Workflow: Formulation and Sampling* chapter, you can solve some equations by minimization if you formulate an objective function that takes the square[1] of the subtraction of one side from another:

$$E(a, b, c) = (a + b + c - 1)^2.$$

Expanding the squared term—while remembering that for binary variables with values 0 or 1 the square of a variable is itself, $X^2 = X$—shows in explicit form the objective function's quadratic and linear terms.

$$\begin{aligned} E(a, b, c) &= a^2 + ab + ac - a + ba + b^2 + bc - b + ca + cb + c^2 - c - a - b - c + 1 \\ &= a^2 + b^2 + c^2 + 2ab + 2ac + 2bc - 2a - 2b - 2c + 1 \\ &= 2ab + 2ac + 2bc - a - b - c + 1, \end{aligned}$$

Notice that this objective formula matches the *QUBO format* for three variables,

$$E_{qubo}(a_i, b_{i,j}; q_i) = \sum_i a_i q_i + \sum_{i<j} b_{i,j} q_i q_j$$

$$E_{qubo}(a_i, b_{i,j}; q_1, q_2, q_3) = a_0 q_0 + a_1 q_1 + a_2 q_2 + b_{0,1} q_0 q_1 + b_{0,2} q_0 q_2 + b_{1,2} q_1 q_2,$$

where $a_i = -1$ and $b_{i,j} = 2$, with a difference of the $+1$ term.[2]

[1] If you do not square, the minimum for $\tilde{E}(a, b, c) = a + b + c - 1$ is $\tilde{E}(a = b = c = 0) = -1$ which is lower than for any of the three constraint-satisfying states, such as $\tilde{E}(a = b = 0; c = 1) = 0$.

[2] A constant term in an objective function does not affect the solutions because it just increases or decreases energies (values of the objective) for all states by the same amount, preserving relative ordering.

Below, the truth table is shown with an additional column of the energy for the objective function found above. The lowest energy states (best solutions) are those that match the exactly-one-true constraint.

a	b	c	Exactly One is True	Energy
0	0	0	FALSE	1
1	0	0	TRUE	0
0	1	0	TRUE	0
1	1	0	FALSE	1
0	0	1	TRUE	0
1	0	1	FALSE	1
0	1	1	FALSE	1
1	1	1	FALSE	4

Clearly, a solver minimizing the objective function $2ab + 2ac + 2bc - a - b - c$ can be expected to return solutions (values of variables a, b, c) that satisfy the original problem of an *exactly-one-true* constraint.

As explained in the *Solving Problems with Quantum Samplers* chapter, to solve a QUBO with a D-Wave quantum computer, you must map (minor embed) it to the QPU. That step is explained in detail in the next chapter.

The *Constraints Example: Submitting to a QPU Solver* chapter then shows how the problem is submitted for solution to a D-Wave quantum computer.

CHAPTER 8

CONSTRAINTS EXAMPLE: MINOR-EMBEDDING

This chapter shows how to minor-embed the QUBO created in the previous chapter onto a QPU, in this case, a D-Wave 2000Q with its Chimera graph.

It is intended to walk you through the minor-embedding and unembedding process for a simple problem so that you understand how it works. D-Wave provides automatic minor embedding tools, and if you are submitting your problem to a Leap quantum-classical hybrid solver, the solver handles all interactions with the QPU. For more information, see the Ocean software documentation.

8.1 Chains

As stated in the *Solving Problems with Quantum Samplers* chapter, objective functions can be represented by graphs and this graphic representation means you can map the objective function to the QPU: nodes that represent variables (sometimes called *logical qubits*) with linear coefficient are mapped to (physical) qubits and their biases, while edges that represent quadratic coefficients are mapped to couplers and their coupling strengths.

The simple example of the *Unconstrained Example: Solving a SAT Problem* chapter noted that its two-node graph can be mapped to any two QPU qubits with a shared coupler. But as you saw in the *D-Wave QPU Architecture: Topologies* chapter, not all qubits share couplers. How do you minor-embed more complex graphs? Minor embedding often requires *chains*, which are described here.

The QUBO developed for an exactly-one-true constraint with three variables in the *Constraints Example: Problem Formulation* chapter can be represented by the triangular graph shown in Figure 8.1.

To see how a triangular graph fits on the Chimera graph, take a closer look at the Chimera structure shown in Figure 8.2. Notice that there is no way to make a triangular closed loop of three qubits and their connecting edges. However, you can make a closed loop of four qubits and their edges using, say, qubits 0, 1, 4, and 5.

To make a three-node loop of a four-node structure, represent a single variable with a chain of two qubits, as shown in Figure 8.3, by chaining qubit 0 and qubit 5 to represent variable b.

Chaining qubits is done by setting the strength of their connecting couplers negative enough to strongly correlate the states of the chained qubits; if at the end of most anneals these qubits are in the same classical state, representing the same binary value in the objective function, they are in effect acting as a single variable.

Here, for qubits 0 and 5 to represent variable b, the strength of the coupler between them

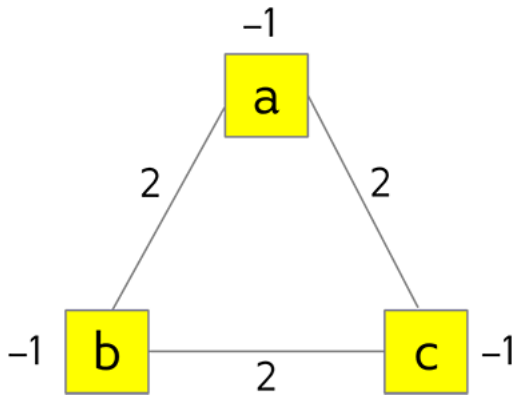


Figure 8.1: Triangular graph for an exactly-one-true constraint with its biased nodes and edges.

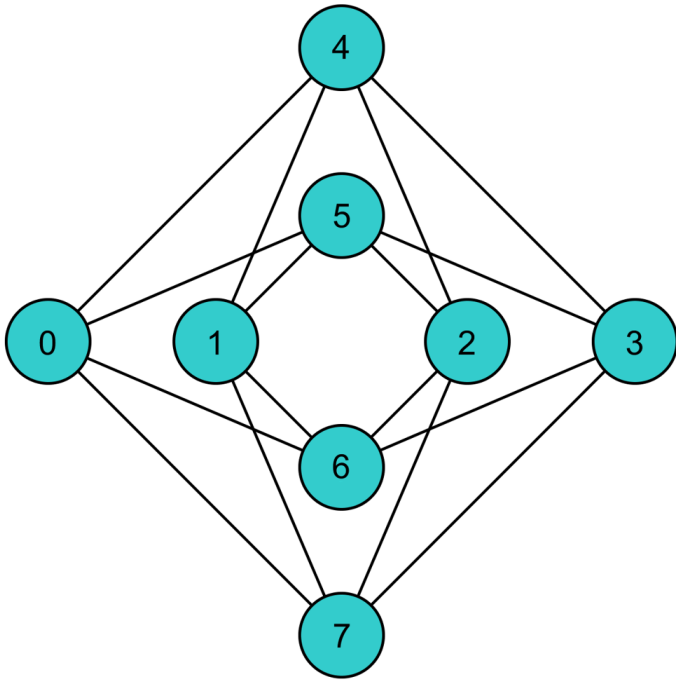


Figure 8.2: Chimera unit cell.

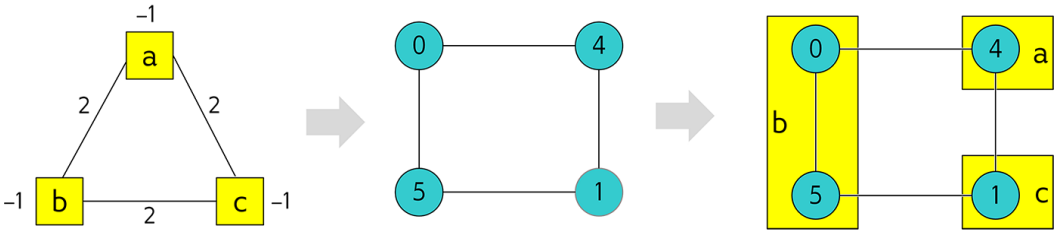


Figure 8.3: Embedding a triangular graph into Chimera by using a chain.

must be set negative enough.

8.2 Manual Minor-Embedding

Manually minor-embedding a problem is typically undertaken only for problems that have either very few variables or a repetitive structure that maps to unit cells of the QPU topology—in both cases you work with one or more unit cells. You are unlikely to manually embed a random 100-variable problem.

Note: For simplicity, this example uses the first Chimera unit cell with qubits 0 to 7. As explained in the *D-Wave QPU Architecture: Topologies* chapter, the subset of the graph accessible to users is the *working graph*, and it's possible some qubits or couplers used here are not available in a particular QPU. In such a situation, a different unit cell can be selected.

The mapping of Figure 8.3 is straightforward for non-chained qubits with biases being the linear coefficients of the objective function and coupler strengths the quadratic coefficients:

- Variables a and c, represented by qubits 4 and 1, respectively, have bias -1.
- Edges $(a,b),(a,c),(b,c)$, represented by couplers $(0,4),(1,4),(1,5)$, respectively, have strengths 2.

To chain qubits 0 and 5 to represent variable b requires that you add a strong negative coupling strength between them. This coupling has no corresponding quadratic coefficient in the objective function, so other biases must be adjusted to compensate. This process requires a few steps:

1. Evenly split the bias of -1 from variable b between qubits 0 and 5. Now the bias of these two qubits is -0.5.
2. Choose a strong negative coupling strength for the chain between qubits 0 and 5. This example arbitrarily chooses -3 because it is stronger than the values for couplers around it.[1]
3. Compensate for the -3 added in step 2 by adding $-\frac{-3}{2} = 1.5$ to each bias of qubits 0 and 5. Now the biases for these qubits are 1.

The resulting minor-embedding values are shown in the tables below.

Table 8.1: Minor Embedding: Linear Coefficients.

Node	Linear Coefficient	Qubits	Bias
a	-1	4	-1
b	-1	0, 5	1, 1
c	-1	1	-1

Table 8.2: Minor Embedding: Quadratic Coefficients.

[1] Setting chain strengths is further discussed in the *Appendix: Next Learning Steps* chapter.

Edge	Quadratic Coefficient	Coupler	Strength
(a,b)	2	(0,4)	2
(a,c)	2	(1,4)	2
(b,c)	2	(1,5)	2
		(0,5)	-3

You program the quantum computer to solve this problem by configuring the QPU's qubits with these biases and its couplers with these strengths. The next section shows and makes sense of the results.

Note: When using the QUBO formulation, as in this example, you compensate for the quadratic term a chain introduces into the objective by adding its negative, divided by the number of qubits in the chain, to the biases of the chain's qubits; this compensation is not used for the Ising formulation, where the the energies of valid solutions are simply shifted by the introduced quadratic term.

8.3 Unembedding

Below are results from submitting[2] this problem to a D-Wave 2000Q system for 1000 anneals:

Energy	Qubit				Occurrences
	0	5	4	1	
-1.0	0	0	1	0	206
-1.0	0	0	0	1	526
-1.0	1	1	0	0	267
0.0	1	1	0	1	1

The solutions returned from the QPU express the states of qubits at the end of each anneal. To translate qubit states to values of the problem variables, the solutions must be *unembedded*.

For this simple example with its single chain, unembedding consists of mapping qubits 4, 1 to variables a, c, and qubits 0, 5 to variable b. The results in the table above unembed to:

- Row 1: Solution $(a, b, c) = (1, 0, 0)$ with energy -1 was found 206 times.
- Row 2: Solution $(a, b, c) = (0, 0, 1)$ with energy -1 was found 526 times.
- Row 3: Solution $(a, b, c) = (0, 1, 0)$ with energy -1 was found 267 times.

One anneal ended with result $(a, b, c) = (0, 1, 1)$, which is not a correct solution, and has a higher energy than the correct solutions.

Note: Notice also that the energy of the valid solutions, the ground-state energy, is -1, not the zero calculated in the *Constraints Example: Problem Formulation* chapter's truth table.

[2] The next chapter, *Constraints Example: Submitting to a QPU Solver*, shows how you submit your objective function to a D-Wave solver.

This is because of the constant $+1$ dropped from the objective function, $E(a,b,c) = 2ab + 2ac + 2bc - a - b - c + 1$.

CHAPTER 9

CONSTRAINTS EXAMPLE: SUBMITTING TO A QPU SOLVER

This section shows how you submit a problem to a D-Wave quantum computer. It uses D-Wave's open-source Ocean SDK to submit the exactly-one-true problem formulated in the previous chapters.

Before you can submit a problem to D-Wave solvers, you must have an account and an API token; visit Leap to sign up for an account and get your token.

Note: To run the following steps yourself requires prior configuration of some requisite information for problem submission through SAPI. If you have installed the Ocean SDK, this is typically done as a first step; in D-Wave's Leap IDE, the default workspace meets these requirements.

For more information, including on Ocean SDK installation instructions and detailed examples, see the Ocean software documentation.

The section below submits the problem that the *Constraints Example: Minor-Embedding* chapter manually minor-embedded; the next section uses Ocean tools to minor-embed automatically when submitting the original objective function.

9.1 Manually Minor-Embedded Problem

The Chimera embedding found in the *Constraints Example: Minor-Embedding* chapter can be used only on a D-Wave 2000Q system, and only on a QPU that has the selected qubits and couplers on its working graph.[1]

First, select a D-Wave 2000Q system.

```
from dwave.system import DWaveSampler
sampler_manual = DWaveSampler(solver={'topology__type': 'chimera'})
```

The following checks that the qubits and couplers selected in the *Constraints Example: Minor-Embedding* chapter are available. If not, use others (for example, add 8 to shift to the next Chimera unit cell and try qubits 8, 9, 12, 13).

[1] The yield of a working graph is typically less than the total number of qubits and couplers that are fabricated and physically present on a QPU.

```
>>> all(qubit in sampler_manual.nodelist for qubit in [0, 1, 4, 5])
True
>>> all(coupler in sampler_manual.edgelist for coupler in [(0, 4), (0,␣
↪5), (1, 4), (1, 5)])
True
```

Set values of the minor-embedded QUBO and submit to the selected QPU.

```
qubit_biases = {(0, 0): 1, (1, 1): -1, (4, 4): -1, (5, 5): 1}
coupler_strengths = {(0, 4): 2, (0, 5): -3, (1, 4): 2, (1, 5): 2}
Q = {**qubit_biases, **coupler_strengths}

sampleset = sampler_manual.sample_qubo(Q, num_reads=5000)
```

Below are results from running this problem on a D-Wave 2000Q system:

```
>>> print(sampleset)
   0  1  4  5 energy num_oc.
0  0  0  1  0   -1.0    1274
1  0  1  0  0   -1.0    1726
2  1  0  0  1   -1.0    1997
3  1  1  0  1    0.0       1
4  0  1  1  0    0.0       2
['BINARY', 5 rows, 5000 samples, 4 variables]
```

The solutions returned from the QPU express the states of qubits at the end of each of 5000 anneals (`num_reads=5000`), with the `num_oc` column showing the number of occurrences of a returned state. The results above unembed to:

- Row 0: Solution $(a, b, c) = (1, 0, 0)$ with energy -1 was found 1274 times.
- Row 1: Solution $(a, b, c) = (0, 0, 1)$ with energy -1 was found 1726 times.
- Row 2: Solution $(a, b, c) = (0, 1, 0)$ with energy -1 was found 1997 times.

Although five unique states are returned, the lowest energy occurs for the valid solutions, which are the most frequent solution. Three anneals ended with incorrect solutions, which have higher energy than the correct solutions.

9.2 Automated Minor-Embedded Problem

The Ocean software can heuristically find minor-embeddings for your QUBO or Ising objective functions, as shown here.

First, select a D-Wave 2000Q system.

```
from dwave.system import DWaveSampler, EmbeddingComposite
sampler_auto = EmbeddingComposite(DWaveSampler(solver={'topology__type':
↪'chimera'}))
```

Unlike the previous section, here you do not need to check pre-selected qubits and couplers.

Set values of the original QUBO (i.e., variables, such as a, rather than qubit indices, such as

0, and coefficients without modifications for chains) and submit to the selected QPU.

```
linear = {('a', 'a'): -1, ('b', 'b'): -1, ('c', 'c'): -1}
quadratic = {('a', 'b'): 2, ('b', 'c'): 2, ('a', 'c'): 2}
Q = {**linear, **quadratic}

sampleset = sampler_auto.sample_qubo(Q, num_reads=5000)
```

Below are results from running this problem on a D-Wave 2000Q system:

```
>>> print(sampleset)
   a  b  c energy num_oc. chain_b.
0  0  1  0   -1.0    1465      0.0
1  1  0  0   -1.0    1838      0.0
2  0  0  1   -1.0    1695      0.0
3  0  1  1    0.0       1      0.0
4  1  0  1    0.0       1 0.333333
['BINARY', 5 rows, 5000 samples, 3 variables]
```

The results show values for the objective function's variables, not the qubits they were mapped to, so there is no need for unembedding this time. Again in the results of 5000 reads, you see that the lowest energy occurs for the three valid solutions to the problem.

Note for row 4 that the value in the `chain_b` column, which shows the percentage of variable values that are based on broken chains, is not zero. For this anneal, the qubits in the chain of qubits representing variable b ended in different states. Ocean software chose the value $b = 0$ by fixing the broken chain; it has different methods for dealing with broken chains, for example, selecting the value a majority of the chain's qubits were in at the end of the anneal. This is one of a number of techniques, called *postprocessing*, which are often part of solving problems on quantum computers.

CHAPTER 10

APPENDIX: NEXT LEARNING STEPS

The previous chapters presented a high-level explanation of solving problems on D-Wave solvers, using an intuitive approach. This chapter starts you off on your journey to mastering more methodical techniques.

10.1 Problem Formulation

To map a problem to a quadratic model, you can try the following steps.

1. Write out the objective and constraints.

   This should be written out in terms of your problem domain, not necessarily in math format.

   - The *objective* of a problem is what you are looking to minimize[1].
   - A *constraint* in a problem is a rule that you must follow; an answer that does not satisfy a given constraint is called *infeasible*, it's not a good answer, and might not be usable at all. For example, a travelling salesperson cannot be in two cities at once or the trucks in your problem may not be able to hold more than 100 widgets.

   There may be more than one way to interpret your problem in terms of objectives and constraints.

2. Convert objective and constraints into math expressions.

   Decide on the type of variables to best formulate it:

   - **Binary**

     Does your application optimize over decisions that could either be true (or yes) or false (no)? For example,

     – Should the antenna transmit or no?

     – Did a network node experience failure?

   - **Discrete**

     Does your application optimize over several distinct options? For example,

     – Which shift should employee X work?

     – Should the state be colored red, blue, green or yellow?

[1] For maximization see step 3.

- **Integer**

  Does your application optimize the number of something? For example,

  - How many widgets should be loaded onto the truck?

- **Continuous**

  Does your application optimize over an uncountable set? For example,

  - Where should the sensor be built?

Once you think of a problem in these terms, you can assign a variable for each question.

Next ask about the degree the problem can likely be formulated as:

- **Quadratic**

  Are its relationships defined pairwise? For example,

  - In the structural imbalance example each pair of people in the network is either friendly or hostile.

- **Higher-Order**

  Does your application have relationships between multiple variables at once? For example,

  - Simulating an AND gate

Next, transform objective and constraints into math expressions. For binary variables, this can often be done with truth tables if you can break the problem down into two- or three-variable relationships. For other variables and non-quadratic degrees, you can try techniques such as Non-Quadratic (Higher-Degree) Polynomials and Ocean tools such as Higher-Order Models.

3. Reformulate as a quadratic model.

   Different types of expressions require different strategies. Expressions derived from truth tables may not need any adjustments. The *D-Wave Problem-Solving Handbook* provides a variety of reformulation techniques; some common reformulations are:

   - **Squared terms:** QUBO and Ising models do not have squared binary variables. In QUBO format, its 0 and 1 values remain unchanged when squared, so you can replace any term x_i^2 with x_i. In Ising format, its -1 and +1 values always equal 1 when squared, so you can replace any term s_i^2 with the constant 1.
   - **Maximization to minimization:** if your objective function is a maximization function (for example, maximizing profit), you can convert this to a minimization by multiplying the entire expression by -1. For example:

     $$\arg\max(3v_1 + 2v_1v_2) = \arg\min(-3v_1 - 2v_1v_2)$$

   - **Equality to minimization:** if you have a constraint that is an equality, you can convert it to a minimization expression by moving all arguments and constants to one side of the equality and squaring the non-zero side of the equation. This leaves an expression that is satisfied at its smallest value (0) and unsatisfied at any larger value (>0). For example, the equation $x + 1 = 2$ can be converted to $\min_x[2 - (x + 1)]^2$, which has a minimum

value of zero for the solution of the equation, $x = 1$.

This approach is useful also for $\binom{n}{k}$ constraints (selection of exactly k of n variables), where you disfavor selections of greater or fewer than k variables with the penalty $P = \alpha(\sum_{i=1}^{n} x_i - k)^2$, where α is a weighting coefficient (see penalty method) used to set the penalty's strength.

4. Combine expressions.[2]

   Once you have written all of the components (objective and constraints) as quadratic models, for example BQMs, make a final BQM by adding all of the components. Typically you multiply each expression by a constant that weights the different constraints against each other to best reflect the requirements of your problem. You may need to tune these multipliers through experimentation to achieve good results. You can see examples in Ocean software's collection of code examples on GitHub.

10.1.1 Example: Formulate an Ising Model

This example returns to the simple SAT of the *Unconstrained Example: Solving a SAT Problem* chapter but posed as a constraint and formulated as an Ising model following the steps above.

The problem now is to formulate a BQM that represents "the value of v_1 should be identical to the value of v_2", where v_1 and v_2 are binary-valued variables.[3]

Step 1 is to state just a constraint, v_1 equals v_2.

Step 2 is just to write the constraint as either an equation or a truth table (the problem's variables are already binary valued).

This example shows both: the constraint as an equation is simply, $v_1 = v_2$, and below it is represented as a truth table (notice that for an Ising formulation the values are $\{-1, 1\}$ rather than $\{0, 1\}$).

State	v_1	v_2	$v_1 = v_2$
1	-1	-1	True
2	-1	1	False
3	1	-1	False
4	1	1	True

Step 3 here also demonstrates reformulating for both expressions. First, note that for two variables, the *Ising Model* formulation reduces to,

$$\mathrm{E}(h_i, J_{i,j}; S_i) = h_1 s_1 + h_2 s_2 + J_{1,2} s_1 s_2.$$

Reformulating the equality expression as a minimization can be done as follows:

$$v_1 = v_2 \qquad \rightarrow \qquad \min_{v} [v_1 - v_2]^2$$

[2] When using penalty models for constraints.

[3] This could be one small part of a larger problem, for example, an integer_factorization problem formulated with Boolean gates, as demonstrated in the Leap demo and Factoring Jupyter Notebook. The larger problem is to satisfy a constraint that two variables representing factors be assigned values such that their multiplication equals the factored number. The constraint of this section's example might represent a connection between two gates of such a formulation of the factoring problem.

Expanding the square gives,

$$\begin{aligned}\min_v[v_1 - v_2]^2 &= \min_v[v_1^2 + v_2^2 - 2v_1v_2] \\ &= \min_v[1 + 1 - 2v_1v_2] \\ &= \min_v[2 - 2v_1v_2]\end{aligned}$$

You can now map the minimization directly to $-2s_1s_2$, dropping the constant.

Notice that for this Ising model the energy gap between the ground states (e.g., $E(s_1 = s_2 = -1) = -2$) and the excited states (e.g., $E(s_1 = -1, s_2 = +1) = +2$) is 4. If you want a gap of 1, your Ising model is $E(s_1, s_2) = -0.5s_1s_2$.

Alternatively, if you prefer a truth table, you can reformulate as a penalty function. Here, an energy gap of 1 is chosen.

State	v_1	v_2	Penalty
1	-1	-1	p
2	-1	1	p+1
3	1	-1	p+1
4	1	1	p

Substituting the values of the table's variables for variables s_1, s_2 in the two-variable Ising model above, and the desired penalty for the resulting energy, produces for the four rows of the table these four equalities:

$$\begin{aligned}&\text{State 1} \quad h_1(-1) + h_2(-1) + J_{1,2}(-1)(-1) = p \\ &\text{State 2} \quad h_1(-1) + h_2(+1) + J_{1,2}(-1)(+1) = p + 1 \\ &\text{State 3} \quad h_1(+1) + h_2(-1) + J_{1,2}(+1)(-1) = p + 1 \\ &\text{State 4} \quad h_1(+1) + h_2(+1) + J_{1,2}(+1)(+1) = p\end{aligned}$$

Giving the following four equations with four variables:

$$\begin{aligned}-h_1 - h_2 + J_{1,2} &= p \\ -h_1 + h_2 - J_{1,2} &= p + 1 \\ h_1 - h_2 - J_{1,2} &= p + 1 \\ h_1 + h_2 + J_{1,2} &= p\end{aligned}$$

Solving these equations[4] gives $E(s_1, s_2) = -0.5s_1s_2$.

Submitting for solution on a D-Wave quantum computer is similar to the submission shown in the *Constraints Example: Submitting to a QPU Solver* chapter, where it was done for QUBOs:

```
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler())
...
>>> h = {}
```

(continues on next page)

[4] Adding the first and fourth equation immediately gives $J_{1,2} = p$. Adding the second and third, and replacing $J_{1,2}$ for p, gives $J_{1,2} = p = -0.5$. Adding the first two equations, with these now-known values, produces $h_1 = h_2 = 0$.

(continued from previous page)

```
>>> J = {('s1', 's2'): -0.5}
>>> sampleset = sampler.sample_ising(h, J, num_reads=1000)
>>> print(sampleset)
  s1 s2 energy num_oc. chain_.
0 -1 -1   -0.5     372     0.0
1 +1 +1   -0.5     628     0.0
['SPIN', 2 rows, 1000 samples, 2 variables]
```

10.1.2 Next Steps for Learning to Formulate Problems

You can learn more about formulating problems here:

- Problem Formulation Guide on the D-Wave corporate website
- *D-Wave Problem-Solving Handbook*
- Ocean software's collection of code examples on GitHub.

10.2 Ising and QUBO Formulations

Figure 10.1 compares Ising and QUBO notation and related terminology.

Problem Expression	Terms			
	Linear coefficient	Quadratic coefficient	Variable	States
QUBO (scalar)	a_i	$b_{i,j}$	q_i	$\{0, 1\}$
QUBO (matrix)	$Q_{i,i}$	$Q_{i,j}$	x_i	$\{0, 1\}$
Ising	h_i	$J_{i,j}$	s_i	$\{-1, 1\}$
Graph	Node Weight	Edge Strength	Node	
QPU	Qubit Bias	Coupling Strength	Qubit State	{Spin Up, Spin Down}

Figure 10.1: Notation conventions.

Some problems may be easier to formulate or see better results with Ising over QUBO, and vice versa. As demonstrated in the interactive code examples of the Leap demo and Structural Imbalance Jupyter Notebook, the structural balance of signed social networks[5] is easily modeled in Ising format:

$$\sum_{i<j}^{N} J_{i,j} s_i s_j$$

[5] *Social networks* map relationships between people or organizations onto graphs, with the people/organizations as nodes and relationships as edges; for example, Facebook friends form a social network. *Signed social networks* map both friendly and hostile relationships by assigning to edges either positive or negative values. Such networks are said to be structurally balanced when they can be cleanly divided into two sets, with each set containing only friends, and all relations between these sets are hostile.

10.2.1 Transformations Between Ising and QUBO

The transformation between these formats is trivial:

$$s = 2x - 1.$$

Example of Transforming a QUBO to Ising Format

Use $x_i \mapsto \frac{s_i+1}{2}$ to translate a QUBO model to an Ising model, as here:

$$\begin{aligned} f(x) &= -22x_1 - 6x_2 - 14x_3 + 20x_1x_2 + 28x_1x_3 + 9 \\ \tilde{f}(s) &= -22\left(\tfrac{s_1+1}{2}\right) - 6\left(\tfrac{s_2+1}{2}\right) - 14\left(\tfrac{s_3+1}{2}\right) + 20\left(\tfrac{s_1+1}{2}\right)\left(\tfrac{s_2+1}{2}\right) + 28\left(\tfrac{s_1+1}{2}\right)\left(\tfrac{s_3+1}{2}\right) + 9 \\ &= -11s_1 - 11 - 3s_2 - 3 - 7s_3 - 7 + 5s_1s_2 + 5s_1 + 5s_2 + 5 + 7s_1s_3 \\ &\quad +7s_1 + 7s_3 + 7 + 9 \\ &= s_1 + 2s_2 + 5s_1s_2 + 7s_1s_3 \end{aligned}$$

Ocean software can automate such conversion for you:

```
>>> import dimod
>>> dimod.qubo_to_ising({('x1', 'x1'): -22, ('x2', 'x2'): -6, ('x3', 'x3
 ↪'): -14,
...                      ('x1', 'x2'): 20, ('x1', 'x3'): 28},
...                     offset=9)
({'x1': 1.0, 'x2': 2.0, 'x3': 0.0}, {('x1', 'x2'): 5.0, ('x1', 'x3'): 7.
 ↪0}, 0.0)
```

Example of Transforming a Ising to QUBO Format

Use $s_i \mapsto 2x_i - 1$ to translate an Ising model to a QUBO model, as here:

$$\begin{aligned} g(s) &= s_1 + 2s_2 + 5s_1s_2 + 7s_1s_3 \\ \tilde{g}(x) &= (2x_1 - 1) + 2(2x_2 - 1) + 5(2x_1 - 1)(2x_2 - 1) + 7(2x_1 - 1)(2x_3 - 1) \\ &= 2x_1 - 1 + 4x_2 - 2 + 20x_1x_2 - 10x_1 - 10x_2 + 5 + 28x_1x_3 - 14x_1 - 14x_3 + 7 \\ &= -22x_1 - 6x_2 - 14x_3 + 20x_1x_2 + 28x_1x_3 + 9 \end{aligned}$$

Using Ocean software:

```
>>> import dimod
>>> dimod.ising_to_qubo({'s1': 1, 's2': 2}, {('s1', 's2'): 5, ('s1', 's3
 ↪'): 7})
({('s1', 's1'): -22.0, ('s2', 's2'): -6.0, ('s1', 's2'): 20.0,
  ('s1', 's3'): 28.0, ('s3', 's3'): -14.0},
  9.0)
```

10.2.2 Next Steps for Learning about Problem Formats

You can learn more about the QPU's *native* formulations in the *D-Wave Problem-Solving Handbook*.

10.3 Solver Parameters

In previous sections and chapters, you saw the use of parameters such as num_reads in submissions of problems to D-Wave solvers. Setting appropriate parameters can be both hard and crucial to finding good solutions.

10.3.1 num_reads

Consider the outcome of submitting the *exactly-one-true* constraint of the *Constraints Example: Problem Formulation* chapter without specifying the number of anneals.

```
from dwave.system import DWaveSampler, EmbeddingComposite
sampler = EmbeddingComposite(DWaveSampler())
Q = {('a', 'a'): -1, ('b', 'b'): -1, ('c', 'c'): -1,
     ('a', 'b'): 2, ('b', 'c'): 2, ('a', 'c'): 2}
sampleset = sampler.sample_qubo(Q)
```

The single run above returned a solution that is correct but incomplete in that it is one of three possible ground states:

```
>>> print(sampleset)
   a  b  c energy num_oc. chain_.
0  0  1  0   -1.0       1     0.0
['BINARY', 1 rows, 1 samples, 3 variables]
```

For harder problems, the number of anneals you request might determine whether or not you see correct or good solutions at all.

10.3.2 auto_scale

In the *Unconstrained Example: Solving a SAT Problem* chapter, you developed a QUBO for a simple SAT problem,

$$E(q_i) = 0.1q_1 + 0.1q_2 - 0.2q_1q_2,$$

where $a_1 = a_2 = 0.1$ was set to an arbitrary positive number.

Consider now assigning a value of 0.5 to the qubit biases and likewise multiplying the coupler strength fivefold to -1, uniformly scaling the objective function by 5. The scaled QUBO is now,

$$E(q_i) = 0.5q_1 + 0.5q_2 - q_1q_2$$

This objective function also favors states 0 and 4, but the objective value for the excited states is now 0.5 rather than 0.1 previously:

State	q_1	q_2	Objective Value
1	0	0	0
2	0	1	0.5
3	1	0	0.5
4	1	1	0

Recall that the previous objective function returned results that included a small fraction of excited states. One might expect that increasing the value (penalty) of the objective function for the excited states—enlarging the energy gap between the ground state and excited states—would make the excited states harder to reach and therefore less probable.

In fact problems submitted to a QPU solver can benefit from an *autoscaling* feature: each QPU has an allowed range of values for its qubit biases and coupler strengths (a and b in the QUBO format); by default, the system adjusts the a and b values of submitted problems to make use of the entire available range before configuring the QPU's biases and coupler strengths.

You can explicitly disable autoscaling (`auto_scale=False`). Below are results for submitting both QUBOs with and without autoscaling.

```
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler())
...
>>> # Define QUBOs
>>> Q1 = {('q1', 'q1'): 0.1, ('q2', 'q2'): 0.1, ('q1', 'q2'): -0.2}
>>> Q2 = {('q1', 'q1'): 0.5, ('q2', 'q2'): 0.5, ('q1', 'q2'): -1}
...
>>> # Autoscaling on for original QUBO
>>> sampleset = sampler.sample_qubo(Q1, num_reads=5000)
>>> print(sampleset)
  q1 q2 energy num_oc. chain_.
0  0  0    0.0    2764     0.0
1  1  1    0.0    2233     0.0
2  0  1    0.1       1     0.0
3  1  0    0.1       2     0.0
['BINARY', 4 rows, 5000 samples, 2 variables]
...
>>> # Autoscaling on for scaled-up QUBO
>>> sampleset = sampler.sample_qubo(Q2, num_reads=5000)
>>> print(sampleset)
  q1 q2 energy num_oc. chain_.
0  0  0    0.0    3429     0.0
1  1  1    0.0    1570     0.0
2  0  1    0.5       1     0.0
['BINARY', 3 rows, 5000 samples, 2 variables]
...
>>> # Autoscaling off for original QUBO
>>> sampleset = sampler.sample_qubo(Q1, num_reads=5000, auto_scale=False)
>>> print(sampleset)
  q1 q2 energy num_oc. chain_.
0  0  0    0.0    1603     0.0
```

(continues on next page)

(continued from previous page)

```
1  1  1    0.0    1669     0.0
2  0  1    0.1     824     0.0
3  1  0    0.1     904     0.0
['BINARY', 4 rows, 5000 samples, 2 variables]
...
>>> # Autoscaling off for scaled-up QUBO
>>> sampleset = sampler.sample_qubo(Q2, num_reads=5000, auto_scale=False)
>>> print(sampleset)
  q1 q2 energy num_oc. chain_.
0  0  0    0.0    2773     0.0
1  1  1    0.0    2000     0.0
2  1  0    0.5     103     0.0
3  0  1    0.5     124     0.0
['BINARY', 4 rows, 5000 samples, 2 variables]
```

With autoscaling (the default), the two problems are run on the QPU with the same qubit biases and coupling strengths and therefore return similar solutions. (The energies and objective values reported are for the pre-scaling values.) With autoscaling disabled, the first problem, with its smaller energy gap, returns more samples of the excited states.

10.3.3 chain_strength

Although not a parameter of D-Wave solvers, `chain_strength`—a parameter used by some Ocean tools submitting problems to quantum samplers—may also be crucial to successfully solving some problems.

The *Constraints Example: Minor-Embedding* chapter explained that for a chain of qubits to represent a variable, all its constituent qubits must return the same value for a sample, and that this is accomplished by setting a strong coupling to the edges connecting these qubits. It set a value that was a bit stronger than the coupler strength representing edges of the problem.

The last statement might have raised the question, Why not simply maximize the coupling strength for all qubits in all chains? Now, having learnt about the *auto_scale* parameter, you can understand the answer.

In the problem of the *exactly-one-true* constraint of the *Constraints Example: Problem Formulation* chapter, the values set for the problem were:

- qubit biases: -1 and 1,
- coupler strengths between qubits representing variables: 2
- coupler strength between qubits of a chain: -3

Consider a simplified QPU that has a range of -1 to 1 for both biases and coupler strengths. For the maximum value of -3 to fit into the range, it must be scaled down to -1 (i.e., divided by 3). The scaled problem programmed on such a QPU has values:

- qubit biases: $\frac{-1}{3}$ and $\frac{1}{3}$
- coupler strengths between qubits representing variables: $\frac{2}{3}$
- coupler strength between qubits of a chain:-1

If you instead were to use a chain strength of -10, the programmed values are now:

- qubit biases: $\frac{-1}{10}$ and $\frac{1}{10}$
- coupler strengths between qubits representing variables: $\frac{2}{10}$
- coupler strength between qubits of a chain:-1

Notice that the difference between positive and negative qubit biases has shrunk from $\frac{2}{3}$ ($\frac{1}{3}$ - $\frac{-1}{3}$) to just 0.2, and likewise the coupling between qubits representing variables. The QPU is not a high-precision digital computer, it is analog and noisy. For problems with a variety of values for its linear and quadratic coefficients, overly large chain strength degrades the problem definition.

Ocean software tries to set smart default values for your chain strengths. However, complex problems might require "tuning" of chain strengths to reach acceptable solution quality.

Ocean provides tools and information to help you find good values for chain strengths when its default values are inadequate.

For example, you might see information on *broken chains* (chains with qubits that are not all in a single state at the end of the anneal) in returned solutions; if a high percentage of results have broken chains, you might need to increase the coupler strengths; if no or few chains are broken, possibly chain strengths are too strong.

Ocean's problem inspector is a tool for visualizing problems submitted to, and answers received from, D-Wave systems. It helps you see the chains and potential problems.

10.3.4 num_spin_reversal_transforms

Notice that the results shown in the *auto_scale* section above tend to display some asymmetry between the two valid solutions. Qubits on a QPU can be biased to some small degree in one direction or another. The num_spin_reversal_transforms parameter, described in the *Solver Properties and Parameters Reference* guide, can improve results by reducing the impact of analog errors that may exist on the QPU.

```
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler())
...
>>> Q1 = {('q1', 'q1'): 0.1, ('q2', 'q2'): 0.1, ('q1', 'q2'): -0.2}
>>> sampleset = sampler.sample_qubo(Q1, num_reads=5000, num_spin_
 ↪reversal_transforms=50)
>>> print(sampleset)
q1 q2 energy num_oc. chain_.
0  0  0    0.0    2495     0.0
1  1  1    0.0    2505     0.0
['BINARY', 2 rows, 5000 samples, 2 variables]
```

The rerunning of one of the *auto_scale* section's submissions above produced results that are more symmetrical in this case. The use of this parameter has a cost in longer QPU execution time.

10.3.5 Next Steps for Learning about Solver Parameters

Once you have submitted a few first problems of your own to D-Wave solvers, and you are ready to ensure your submissions are configured to produce the best solutions, familiarize yourself with the solver parameters.

You can learn more about solver parameters here:

- *Solver Properties and Parameters Reference*
- *QPU Solver Datasheet* to understand the cost increasing num_reads might have for real-time applications.
- *D-Wave Problem-Solving Handbook* and *QPU Solver Datasheet* guides provide more advanced information on D-Wave systems and best practices.

10.4 Next Learning Steps

Your next learning steps in general depends on how you prefer to learn. Below are some options:

10.4.1 Software

If your preferred learning approach is through programming, consider starting with these resources:

- Ocean software's Getting Started documentation provides a series of code examples, for different levels of experience, that explain various aspects of quantum computing in solving problems.
- Leap demos and Jupyter notebooks.
- Ocean software's collection of code examples on GitHub.

10.4.2 Hardware

If you want to better understand the D-Wave quantum computer system, start with these system-documentation guides:

- Technical details about the system: *QPU Solver Datasheet*.
- The *D-Wave Problem-Solving Handbook* guide's bibliography is a good starting place for papers on quantum computing.
- *Solver Properties and Parameters Reference*

10.4.3 Additional Resources

See also D-Wave corporate website for papers and links to user applications.