The Quantum Computing Company™

Ocean Programs for Beginners

WHITEPAPER

2021-09-28

Overview

This guide provides a gentle introduction to Ocean programs for beginners new to D-Wave's Ocean SDK. Here you will find the building blocks and basic components required in an Ocean program in order to successfully run problems using D-Wave's hardware and software tools.

CONTACT

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

US Office
2650 E Bayshore Rd
Palo Alto, CA 94303

Email: info@dwavesys.com

www.dwavesys.com

14-1043A-B
D-Wave Whitepaper Series

Notice and Disclaimer

Contents

1 Introduction

D-Wave's Ocean software development kit (SDK) allows users to interact with the D-Wave quantum processing units (QPUs) and hybrid solvers through Python programs.

The purpose of this guide is to step new Ocean users through the basic components of an Ocean program. Before working through this guide, please review our introduction to binary quadratic models (BQMs) [1].

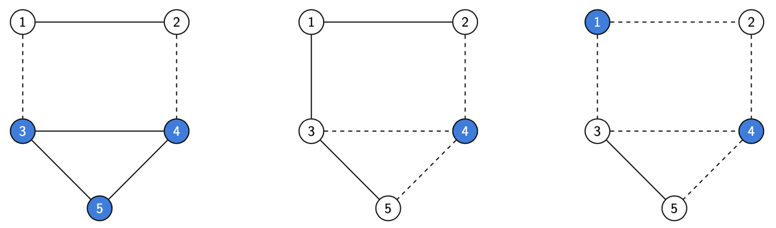
A user interacts with D-Wave solvers by formulating a quadratic model (QM) for their problem, writing a Python program that uses the Ocean SDK, running that Python program, and reviewing the results returned. The Python program introduces a QM and submits it to the selected solver to find the minimum energy value for that model. For example, if the solver is the QPU, the Ocean SDK provides the proper inputs to the physical QPU so that the energy landscape matches the BQM provided.

2 Core Components

An Ocean program has several core components. First, we must build a quadratic model (QM) that will be provided to the solver. This can be done in a number of ways, such as either quadratic unconstrained binary optimization (QUBO) or Ising form. Second, we need to select a sampler to run our problem and provide results. In this section we will step you through each of these components and demonstrate using the maximum cut problem from D-Wave's collection of code examples [2].

2.1 Setting Up The Problem

For this guide, we will work with a simple example called the maximum cut problem. In the maximum cut problem our objective is to maximize the number of cut edges in a graph. In other words, we want to divide the set of nodes in the graph into two subsets so that we have as many edges as possible that go between the two sets. In the image below, the two sets are denoted using blue and white nodes and the cut edges are represented with dashed lines. For this example, the maximum cut size is 5.



Cut Size: 2 Cut Size: 3 Cut Size: 5

We can represent this problem as a binary quadratic model (BQM) with the objective:

$$\min \sum_{(i,j)\in E} \left(-x_i - x_j + 2x_i x_j\right)$$

For the full details on how this formulation is derived, see the full description in the Collection of Code Examples [2]. This particular problem consists only of one single objective and no constraints.

2.2 Building a Quadratic Model

We need to define a quadratic model (QM) that represents our problem. To learn about how to formulate a QM for your problem as a QUBO or Ising model, check out our "Learning to Formulate Problems" guide [1]. That guide will take us through the steps of how to formulate your problem as a QM and how to represent it in mathematical and matrix form.

Once we have our QM, we need to build it in our Python program.

Binary Quadratic Models: The simplest way to build a binary quadratic model (BQM) is using Ocean's symbolic variables. For each mathematical variable in your BQM, we define a symbolic binary variable using Ocean.

```
x = {n: Binary(n) for n in G.nodes}
```

This defines a Python dictionary in which we have a symbolic binary variable defined for each node in our graph. In this dictionary, each key is a node in the graph while the value is the corresponding symbolic binary variable.

Now that we have binary variables that we can use in our Python program, we can build our BQM using these binary variables. Recall that our BQM is represented by the following mathematical expression.

$$\min \sum_{(i,j)\in E} \left(-x_i - x_j + 2x_i x_j\right)$$

Using our symbolic binary variables, we define a QM that matches this mathematical expression.

```
bqm = sum(-x[i]-x[j]+2*x[i]*x[j] for i,j in G.edges)
```

Once the QM is defined, it is stored as a `BinaryQuadraticModel` object. This object stores the linear and quadratic coefficients of the mathematical expression, any constant term or offset, and the type of variables used to build the model. In this case, printing out the object `bqm` that we have constructed reveals the following.

```
BinaryQuadraticModel({1: -2.0, 2: -2.0, 3: -3.0, 4: -3.0, 5: -2.0},
  {(2, 1): 2.0, (3, 1): 2.0, (4, 2): 2.0, (4, 3): 2.0, (5, 3): 2.0, (5, 4): 2.0},
  0.0, 'BINARY')
```

For the 5-node example graph, simplifying the mathematical expression for our objective produces linear and quadratic terms that match the constructed `BinaryQuadraticModel` object.

Constrained Quadratic Models: A constrained quadratic model (CQM) allows us to explicitly set an objective and constraints to model our problem. We begin in the same way as the BQM example by defining our variables. For the maximum cut problem, these are all binary.

```
x = {n: Binary(n) for n in G.nodes}
```

Next, we initialize our CQM object and set the objective for our CQM to represent the minimization problem that we are looking to solve.

```
# Initialize a CQM object
cqm = ConstrainedQuadraticModel()

# Set an objective function for the CQM
cqm.set_objective(sum(-x[i]-x[j]+2*x[i]*x[j] for i,j in G.edges))
```

2.3 Interacting with a Sampler or Solver

To find the minimum energy state for a QM (the assignment of variable values that gives us the minimum energy value for our QM), the Ocean SDK provides samplers and solvers. A **solver** is a resource that runs a problem. **Samplers** are processes that run a problem many times to obtain a collection of samples, each of which is a possible solution to our problem. For convenience, we will generally refer to Ocean's samplers as a whole, to include solvers as well.

The Ocean SDK provides a variety of different samplers that we can use to examine our problem. These range from the D-Wave QPU (`DWaveSampler`) to classical algorithms like tabu search (`TabuSampler`) and even hybrid tools (`LeapHybridSampler`). More information on samplers can be found in the full Ocean documentation [3].

Defining a Sampler. To define a sampler, we need to understand which package it belongs to in the Ocean SDK. In Figure 1 is a list of some commonly used samplers for beginners. Each sampler obtains samples in a different way and can be useful at different stages of your application development.

To define a sampler for our program, we first import the package that contains the sampler using the same syntax as we would for any other Python package import. Then we can instantiate our sampler object in our program so that it is ready to be called.

Example. To use the D-Wave QPU, we might use the following lines of code.

```
# Import the packages required
from dwave.system.samplers import DWaveSampler, EmbeddingComposite

# Define the sampler
sampler = EmbeddingComposite(DWaveSampler())
```

In these lines we see a few different things taking place.

Sampler	Description	Ocean Package	Useful for:
`DWaveSampler`	D-Wave QPU	`dwave-system`	Running problems directly on the QPU
`ExactSolver`	Considers all possible answers to find the optimal solution	`dimod`	Running very small problems classically (<10 variables)
`SimulatedAnnealingSampler`	Classical algorithm for simulated annealing	`dwave-neal`	Running medium-sized problems classically
`LeapHybridSampler`	Quantum-Classical hybrid sampler	`dwave-system`	Running large problems across a portfolio of quantum and classical hardware

Figure 1: List of commonly used Ocean samplers

First, `DWaveSampler` tells the system that we want to use the D-Wave quantum computer. Wrapped around our call to `DWaveSampler` we see `EmbeddingComposite`. This tool lets the Ocean SDK find the best way to map, or embed, our logical problem (our QM) onto the physical QPU. It decides which qubits to map our variables onto and will unembed solutions so that they are returned to us in terms of our variables.

2.4 Calling a Sampler

Once we have established our sampler in our program, we can call it for our QM. Each type of QM model has its own method for interacting with the sampler, whether it be QUBO, BinaryQuadrticModel, or any other QM. We call the sampler to sample our QM using one of Ocean's `sample` functions, depending on what type of QM we are using. For example, the code snippet below demonstrates how we can sample a `BinaryQuadraticModel` object named `bqm` using the QPU.

```
# Define the sampler
sampler = EmbeddingComposite(DWaveSampler())

# Sample the BQM and store the results in the SampleSet object
sampleset = sampler.sample(bqm, num_reads = 100)
```

Note that each sampler has its own set of parameters, or additional settings, in the sample methods available. In the previous code snippet, the parameter `num_reads` is used to run the BQM 100 times on the QPU. A list of the properties and parameters specific to the QPU (`DWaveSampler`) is available here [4]. Beginners should pay particular attention to the `chainstrength` and `num_-reads` (number of reads) parameters, as discussed in the documentation.

2.5 Examining Results from a Sampler

After we have sampled our QM, the sampler returns a `SampleSet` object. This object contains all of the samples returned along with their corresponding energy value, number of occurrences, and more. The additional information varies depending on which sampler is used. As users get more comfortable with the Ocean SDK and the variety of samplers available, it is often useful to take some time to explore the wealth of information provided in the `SampleSet` object.

Some of the key properties and methods of a `SampleSet` that we access are the following.

`SampleSet.record`: The full set of samples.

Each line shows a sample (solution) that was returned, along with the corresponding energy value, number of occurrences, and additional information like chain break fraction (QPU samplers), feasibility (CQM solver), or other sampler-specific information.

Example from a QPU sampler:

```
[([0, 1, 1, 0, 0], -5., 26, 0.)
 ([0, 1, 1, 0, 1], -5., 33, 0.)
 ([1, 0, 0, 1, 0], -5., 24, 0.)
 ([1, 0, 0, 1, 1], -5., 17, 0.)]
```

`SampleSet.first`: The sample with the lowest energy.

Example from a QPU sampler:

```
Sample(sample={1: 0, 2: 1, 3: 1, 4: 0, 5: 0}, energy=-5.0, num_occurrences=26,
    chain_break_fraction=0.0)
```

`SampleSet.data`: The complete information about the solutions and sampler.

Example from a QPU sampler:

```
<bound method SampleSet.data of
SampleSet(rec.array([
  ([0, 1, 1, 0, 0], -5., 26, 0.),
  ([0, 1, 1, 0, 1], -5., 33, 0.),
  ([1, 0, 0, 1, 0], -5., 24, 0.),
  ([1, 0, 0, 1, 1], -5., 17, 0.)],
dtype=[('sample', 'i1', (5,)),
  ('energy', '<f8'),
  ('num_occurrences', '<i8'),
  ('chain_break_fraction', '<f8')]),
[1, 2, 3, 4, 5],
{'timing':
  {'qpu_sampling_time': 2389,
   'qpu_anneal_time_per_sample': 20,
   'qpu_readout_time_per_sample': 198,
   'qpu_access_time': 13078,
   'qpu_access_overhead_time': 4062,
   'qpu_programming_time': 10689,
   'qpu_delay_time_per_sample': 21,
   'total_post_processing_time': 426,
   'post_processing_overhead_time': 426,
   'total_real_time': 13078,
   'run_time_chip': 2389,
   'anneal_time_per_run': 20,
   'readout_time_per_run': 198},
   'problem_id': '9925c084-f2e6-4124-a361-11607a92439c'},
   'BINARY')>
```

3 A More Complex Example

Similar to the maximum cut problem is another problem from graph theory called the graph partitioning problem. In this problem, our objective is to minimize the number of cut edges and our constraint is that both subsets of nodes must have equal size. Mathematically, this can be expressed as the following.

Objective: $\min \sum_{(i,j)\in E} \left(x_i + x_j - 2x_i x_j\right)$

Constraint: $\sum_{v\in G} x_v = |G|/2$

Binary Quadratic Model: We will again begin by defining a binary variable for each node in our graph.

```
x = {n: Binary(n) for n in G.nodes}
```

We begin the construction of our QM by building a `BinaryQuadraticModel` object that consists of our objective expression.

```
bqm = sum(-x[i]-x[j]+2*x[i]*x[j] for i,j in G.edges)
```

Next, we add in our constraint using the `add_linear_equality_constraint` function from Ocean. To use this function we must have a constraint of the form $(\sum_i a_i x_i) + C = 0$. To map our mathematical constraint to this form, we move the constant term $|G|/2$ to the left-hand side of the equation. When using this function, the first parameter is a list of tuples representing (variable, coefficient), or (x_i, a_i). The `constant` parameter is the constant term in the equation, and the `lagrange_multiplier` parameter provides a weighting coefficient to effectively balance the objective and constraint for the problem. Note that for this constraint to be satisfied, we must have an even number of nodes in the graph.

```
bqm.add_linear_equality_constraint([(n, 1) for n in G.nodes],
                                    constant = -G.number_of_nodes()/2,
                                    lagrange_multiplier = 1)
```

Now the `BinaryQuadraticModel` called `bqm` completely models our optimization problem and can be sent over to one of the available samplers.

Constrained Quadratic Model: As before, we define a binary variable for each node in our graph.

```
x = {n: Binary(n) for n in G.nodes}
```

To build the CQM for the graph partitioning problem, we initialize our `ConstrainedQuadraticModel` object, set our objective, and add in the constraint. This is shown in the code snippet below.

```
# Import the required package
from dwave.system import LeapHybridCQMSampler

# Initialize the CQM
cqm = ConstrainedQuadraticModel()

# Set the objective
cqm.set_objective(sum(-x[i]-x[j]+2*x[i]*x[j] for i,j in G.edges))

# Add a constraint
cqm.add_constraint(sum(x[i] for i in G.nodes) == (G.number_of_nodes()/2),
    label='partition-size')
```

Note that a label is provided for the constraint. This allows the SampleSet returned from the sampler to check whether or not each individual constraint is satisfied. For example, in the following code snippet we define our sampler to be `LeapHybridCQMSampler`, sample the CQM object, and print out the SampleSet object returned.

```
# Define the sampler
sampler = LeapHybridCQMSampler()

# Sample the CQM object
sampleset = sampler.sample_cqm(cqm)

# Print the results
print(sampleset.first)
```

This displays for the user the following SampleSet.

```
Sample(sample={1: 0.0, 2: 1.0, 3: 1.0, 4: 0.0, 5: 0.0, 6: 1.0}, energy=-6.0,
    num_occurrences=1, is_feasible=True, is_satisfied=array([ True]))
```

First we see the sample with binary variable assignments, followed by the energy and number of occurrences. Lastly, we see two fields unique to constrained quadratic models: `is_feasible`: True if all constraints are satisfied, and `is_satisfied`: an array with an entry for each constraint indicating True if the constraint is satisfied.

4 Using Matrices for Binary Quadratic Models

An alternative to building binary quadratic models symbolically is to build them using a matrix representation. A matrix representation of a BQM contains linear coefficients along the diagonal and quadratic coefficients on the off-diagonal. An easy way to think of a BQM in matrix form is to imagine variable names across the rows and columns of our matrix. Using Ocean, we can encode this matrix in a variety of ways such as with a Python dictionary or with a NumPy array. Below we show converting a BQM matrix to a Python dictionary.

$$\begin{array}{c|cc} & x_1 & x_2 \\ \hline x_1 & 0 & 2 \\ x_2 & 0 & 1 \end{array} \quad \text{is equivalent to } \{(x_1, x_2) : 2, (x_2, x_2) : 1\}.$$

When we enter our BQM matrix into a dictionary, we generally only include the non-zero entries, which allows us to save space and run larger problems more efficiently.

Maximum Cut Example. In the maximum cut example, the BQM equation for a graph with edge set E was determined to be:

$$\min \sum_{(i,j)\in E} (-x_i - x_j + 2x_i x_j).$$

In `maximum_cut.py`, the Python dictionary for our matrix is built in the following code snippet.

```
from collections import defaultdict

# Initialize our Q matrix
Q = defaultdict(int)

# Update Q matrix for every edge in the graph
for u, v in G.edges:
    Q[(u,u)]+= -1
    Q[(v,v)]+= -1
    Q[(u,v)]+= 2
```

In this example, we use `defaultdict(int)` to initialize the dictionary `Q`. This allows us to create new dictionary elements that are initialized with the value `0` as they are added to the dictionary [5].

5 Further Example Programs

For some simple Ocean program examples, check out our Collection of Code Examples [2]. These examples can be fully explored through our Leap cloud platform and with our integrated developer environment (IDE), available at `cloud.dwavesys.com/leap`.

References

1 *Learn to formulate problems*, `https://www.dwavesys.com/media/bu0lh5ee/bqm_dev_guide.pdf` (2020).

2 *Collection of code examples*, `https://github.com/dwave-examples` (2020).

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

4 *D-Wave system documentation*, `https://docs.dwavesys.com/docs/latest/c_solver_parameters.html` (2020).

5 *Python documentation: collections*, `https://docs.python.org/3/library/collections.html` (2020).