Quantum Computing HW 6
Carl Klier and Nachiket Patel
Github Link: https://github.ncsu.edu/nnpatel5/Project-B2-GraphColoring
11/17/2020

Combinatorial Optimization for Graph Coloring Report 3

Introduction to the Graph Coloring Problem

In graph theory, the idea of graph coloring, specifically a vertex coloring, entails assigning colors to nodes of the graph so that no two adjacent nodes are the same color. A coloring of the vertices that uses at most k colors is a k-coloring and the graph is said to be k-colorable. The motivation to be able to solve the k-coloring problem in the most efficient way comes from the fact that there are a lot of practical applications for this problem. One such application is in scheduling a set of conflicting jobs that interfere with each other and need to be executed at different times. If we use a graph where the vertices correspond to jobs, and edges exist between two jobs if the two jobs can't be run at the same time, then a k-coloring tells us that all the jobs can be completed in k time slots. The minimum value for k such that a k-coloring exists, called the chromatic number, is the minimum time required to finish all the jobs [1].

Another application for graph coloring is in compiler theory with register allocation. The goal of register allocation is to efficiently assign many variables to a limited number of cpu registers to optimize the speed of applications. This problem can be converted into a k-coloring problem just as the scheduling problem and solved to conclude that the set of variables needed at one time can be stored in just k number of registers [2].

Although the k-coloring graph problem is NP-complete for any integer $k \geq 3$, there have been many algorithms and heuristics developed to solve the problem on classical computers. One obvious exact algorithm is brute force search which iterates through k values starting at 1 until a valid coloring is found. However, this is impractical for large graphs. Since no known polynomial time algorithm exists, it is necessary to use heuristics to arrive at a suboptimal solution in polynomial time. One common algorithm is the greedy coloring algorithm which takes a list of vertices in some order and simply assigns each vertex the smallest possible color [3]. While the greedy algorithm succeeds in finding a valid coloring, the order of the vertices determines whether the algorithm finds the chromatic number and finding this optimal ordering is non-trivial [2].

Project Goals

The goal of our project is to solve the graph coloring problem on a quantum computer by implementing the problem in two different ways, first on a quantum annealing machine such as

D-wave's system and secondly on a gate based quantum computer such as IBM Q's machine. We want to compare and contrast these two implementations analyzing how the problem was programmed, specifically the effort involved in deriving the BQM for the annealing machine compared to designing a circuit for the gate based machine. We also want to compare the quality and accuracy of the solutions and the computational speedup over classical approaches. Lastly, we want to explore the limitations of current quantum computers in solving the graph coloring problem.

Quantum Annealing for Graph Coloring

With the graph coloring problem being a combinatorial optimization problem, it can be represented as constraint satisfaction problem and converted into a binary quadratic model with unary encoding to represent the colors. The first step would be to formulate the problem as a graph, where each node is represented with c variables, which correspond to the number of possible different colors in the problem. From the initial description of the D-Wave implementation, it is obvious that for large graphs, the problem becomes infeasible for embedding into a unit cell of the D-Wave system because most QUBOs for such problems are too large [4]. To avoid this problem, either the problem size has to be restricted or a decomposition technique will have to be implemented to allow problems to be scaled up [5]. The techniques to allow for solving larger problems involve minor embedding and cloning. Minor embedding is the process of mapping logical qubits to physical qubits. This is necessary because the structure of the unit cell limits what physical connections between qubits are possible [6]. Cloning is analogous to chaining of physical qubits to represent a single logical qubit. However, in this case of the graph coloring problem, a node is cloned so that multiple unit cells are connected and used to represent a color for a node. The motivation for cloning is that we can extend the footprint of a single node in the unit cell array to provide more neighbors to that node. This allows us to enforce more neighbor constraints that otherwise would not transfer directly to the unit cell array. [7]

An interesting property of the graph coloring problem is that it exhibits symmetry, this can be utilized as the number of colors in the graph increases to simplify the problem. Due to this symmetry, scaling up from c colors to c+1 colors can be done systematically without the need for complete reconstruction of the problem [7].

To solve the graph coloring problem on a D-Wave annealing machine, we first created a constraint satisfaction problem with two constraints. The first constraint was that a node could only be labeled with a single color and the second constraint was that a node could not have the same color as one of its neighbors. We then converted the constraint satisfaction problem into a binary quadratic model. Initially, we coded the graph coloring problem to use a quantum simulator, specifically dimod's ExactSolver. Once the code was working properly, we transitioned to running on actual quantum hardware. Using dwave.system's EmbedingComposite to create an implicit embedding, we then let the DWaveSampler perform 8000 reads. What we get back from the sampler is a list of node and color combinations with certain combinations "turned on." We then plot the "turned on" color and node combinations to visually examine if we arrived at a valid coloring or not. In addition, we also check that the list of variables (node and color combinations) satisfies the constraints of our constraint satisfaction problem.

The initial problem we aimed to solve was finding a 3-coloring for the graph below:

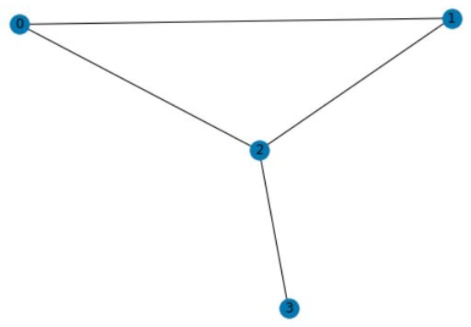


Figure 1: The graph used for 3 coloring

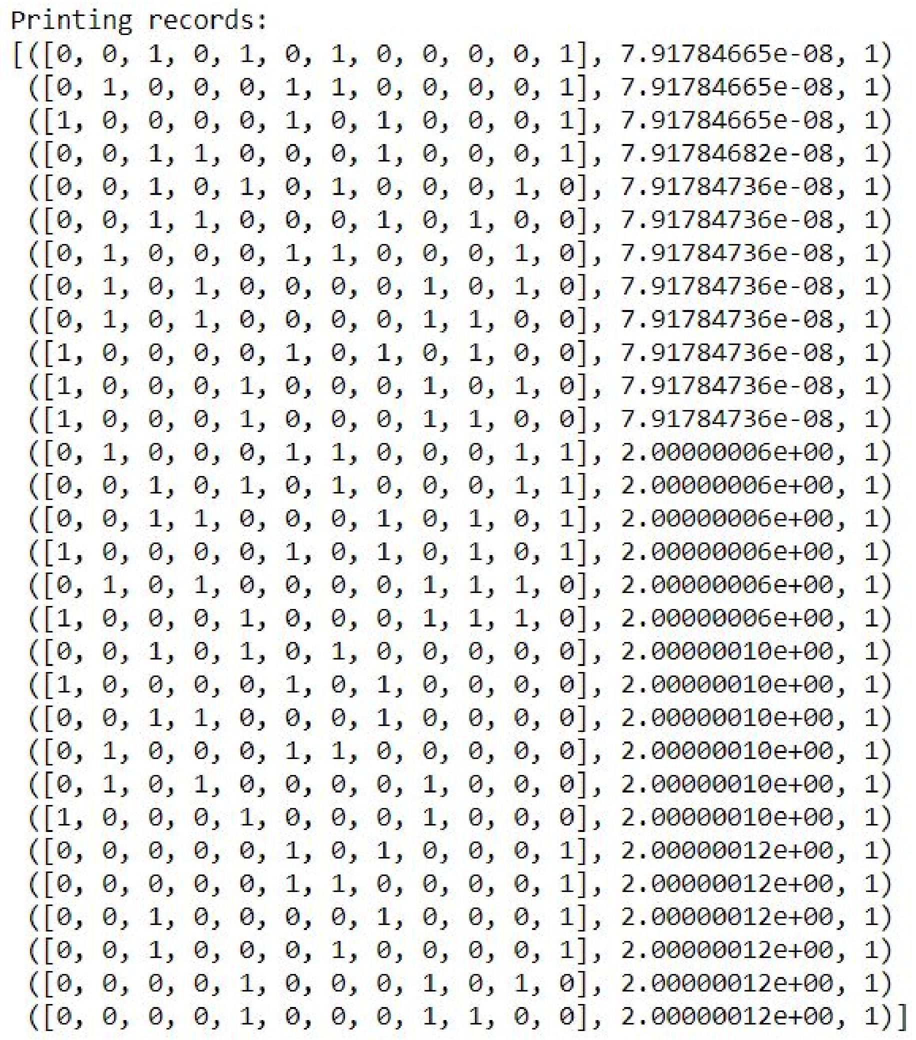
To start out, we used one of D-Wave's provided examples for solving the graph coloring problem. This example showed how to use a constraint satisfaction problem to solve a 4-coloring for a graph of Canada's provinces. We then made modifications to allow the code to handle any arbitrary graph and to find any k-coloring. This involved generating the possible color configurations based upon the k value to create one of the constraints for the constraint satisfaction problem. We then used this code which finds k-colorings on any graph to find a 3-coloring for the graph seen in Figure 1.

However, it was then suggested that a k value that is a power of 2 would potentially give better results, so we also decided to run our code to try and find a 4-coloring for the same graph above and compare our results. After this, we also explored other extensions to our graph coloring problem which are mentioned in a below section. These involve further exploration of the coloring of Canada's provinces and a solution to finding the chromatic number.

Quantum Annealing for Graph Coloring Results

Dimod's ExactSolver's Results for 3-coloring

For the graph seen in Figure 1, there are 12 valid 3-colorings. The simulator using our derived BQM was able to find all 12 valid colors and they had significantly lower energy values then invalid colorings. This can be seen in the below figure of the output of the simulator.



```
Printing records:
[([0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1], 7.91784665e-08, 1)
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1], 7.91784665e-08, 1)
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1], 7.91784665e-08, 1)
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1], 7.91784682e-08, 1)
 ([0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0], 7.91784736e-08, 1)
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0], 7.91784736e-08, 1)
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0], 7.91784736e-08, 1)
 ([0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0], 7.91784736e-08, 1)
 ([0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0], 7.91784736e-08, 1)
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0], 7.91784736e-08, 1)
 ([1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0], 7.91784736e-08, 1)
 ([1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0], 7.91784736e-08, 1)
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1], 2.00000006e+00, 1)
 ([0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1], 2.00000006e+00, 1)
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1], 2.00000006e+00, 1)
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1], 2.00000006e+00, 1)
 ([0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0], 2.00000006e+00, 1)
 ([1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0], 2.00000006e+00, 1)
 ([0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0], 2.00000010e+00, 1)
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0], 2.00000010e+00, 1)
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0], 2.00000010e+00, 1)
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0], 2.00000010e+00, 1)
 ([0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0], 2.00000010e+00, 1)
 ([1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0], 2.00000010e+00, 1)
 ([0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1], 2.00000012e+00, 1)
 ([0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1], 2.00000012e+00, 1)
 ([0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1], 2.00000012e+00, 1)
 ([0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1], 2.00000012e+00, 1)
 ([0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0], 2.00000012e+00, 1)
 ([0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0], 2.00000012e+00, 1)]
```

Figure 2: First 30 outputs of the simulator for 3-coloring

Looking at the lowest energy value result, we see that when the sample is graphed, it is indeed a valid 3-coloring.

```
Sample:  {'a0': 0, 'a1': 0, 'a2': 1, 'b0': 0, 'b1': 1, 'b2': 0, 'c0': 1, 'c1': 0, 'c2': 0, 'd0': 0, 'd1': 0, 'd2': 1}
Energy Value:  7.917846645000282e-08
```

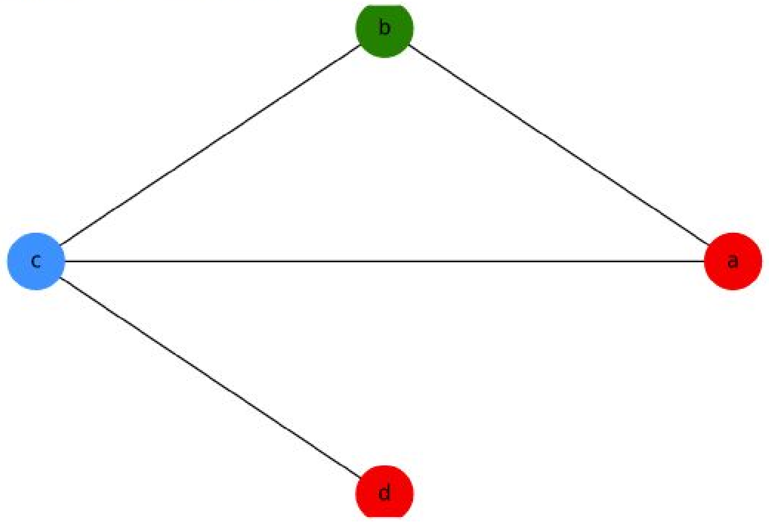


Figure 3: Valid 3-coloring from simulator

Looking at the 13th result, a result that has an energy value greater than 2, we see that it is not a valid 3-coloring. Here, the color grey means that more than one color was assigned to the node which violates one of the constraints of our constraint satisfaction problem.

```
Sample:  {'a0': 0, 'a1': 1, 'a2': 0, 'b0': 0, 'b1': 0, 'b2': 1, 'c0': 1, 'c1': 0, 'c2': 0, 'd0': 0, 'd1': 1, 'd2': 1}
Energy Value:  2.0000000592409197
Failed to color graph validly
```

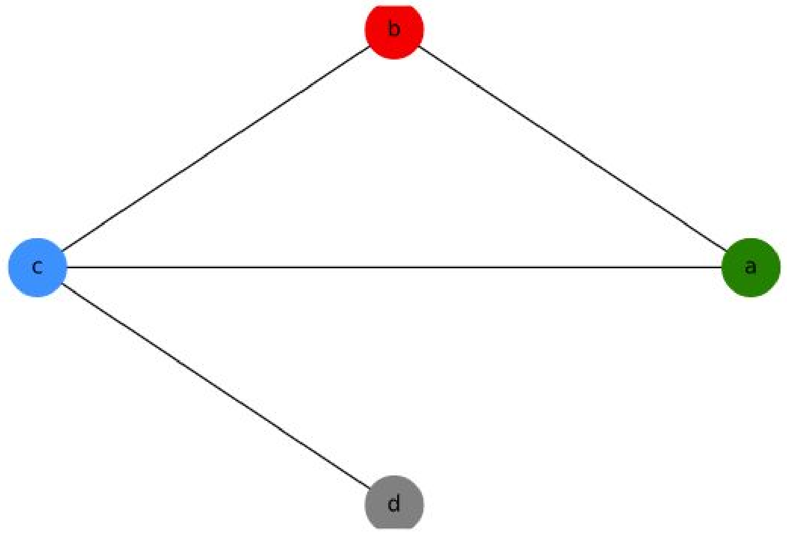
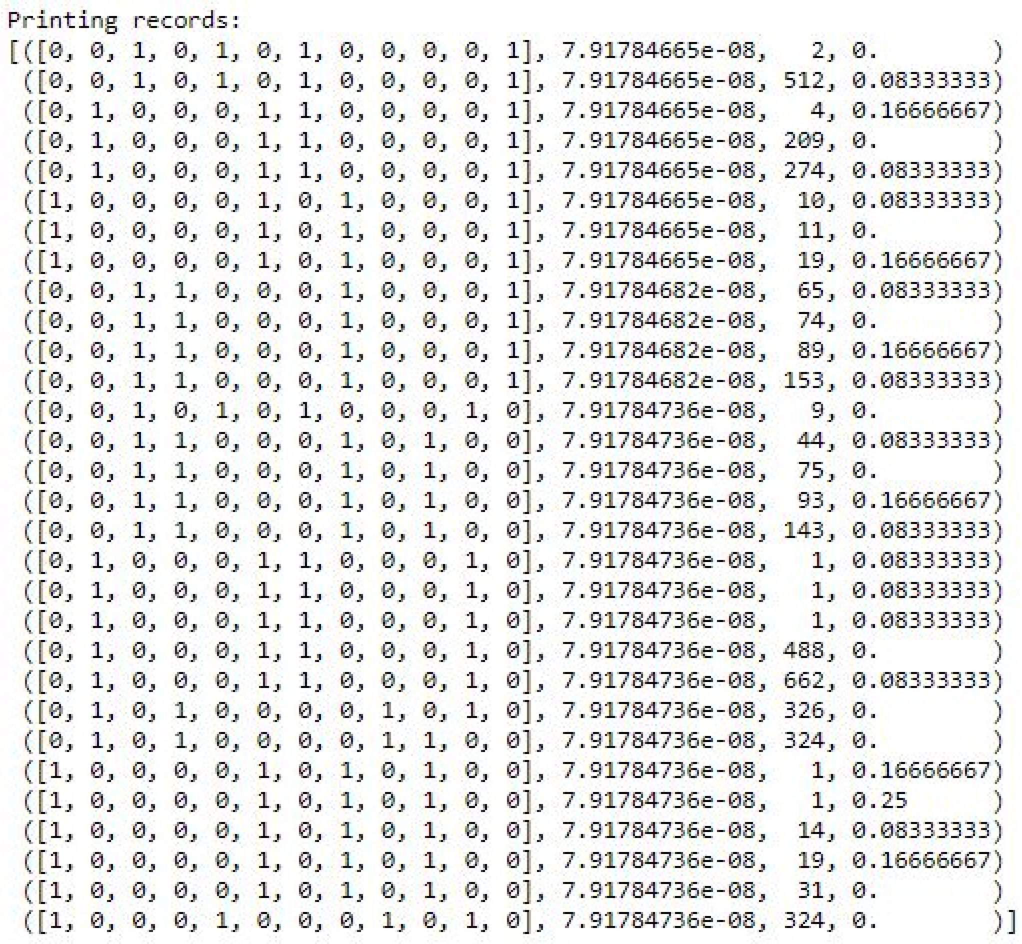


Figure 4: Invalid 3-coloring from simulator

DWaveSampler’s Results for 3-coloring

Running our code on actual hardware using DWaveSampler also resulted in finding valid 3-colorings. See below the first 30 outputs of the sampler sorted by increasing energy value.



```
Printing records:
[([0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1], 7.91784665e-08,   2, 0.        )
 ([0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1], 7.91784665e-08, 512, 0.08333333)
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1], 7.91784665e-08,   4, 0.16666667)
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1], 7.91784665e-08, 209, 0.        )
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1], 7.91784665e-08, 274, 0.08333333)
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1], 7.91784665e-08,  10, 0.08333333)
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1], 7.91784665e-08,  11, 0.        )
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1], 7.91784665e-08,  19, 0.16666667)
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1], 7.91784682e-08,  65, 0.08333333)
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1], 7.91784682e-08,  74, 0.        )
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1], 7.91784682e-08,  89, 0.16666667)
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1], 7.91784682e-08, 153, 0.08333333)
 ([0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0], 7.91784736e-08,   9, 0.        )
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0], 7.91784736e-08,  44, 0.08333333)
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0], 7.91784736e-08,  75, 0.        )
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0], 7.91784736e-08,  93, 0.16666667)
 ([0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0], 7.91784736e-08, 143, 0.08333333)
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0], 7.91784736e-08,   1, 0.08333333)
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0], 7.91784736e-08,   1, 0.08333333)
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0], 7.91784736e-08,   1, 0.08333333)
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0], 7.91784736e-08, 488, 0.        )
 ([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0], 7.91784736e-08, 662, 0.08333333)
 ([0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0], 7.91784736e-08, 326, 0.        )
 ([0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0], 7.91784736e-08, 324, 0.        )
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0], 7.91784736e-08,   1, 0.16666667)
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0], 7.91784736e-08,   1, 0.25      )
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0], 7.91784736e-08,  14, 0.08333333)
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0], 7.91784736e-08,  19, 0.16666667)
 ([1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0], 7.91784736e-08,  31, 0.        )
 ([1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0], 7.91784736e-08, 324, 0.        )]
```

Figure 5: First 30 outputs of the real D-Wave hardware for 3-coloring

You can see from the graph below the transition between the valid 3-colorings and the invalid 3-colorings returned by the sampler by the spike in energy values around the 30th sample.

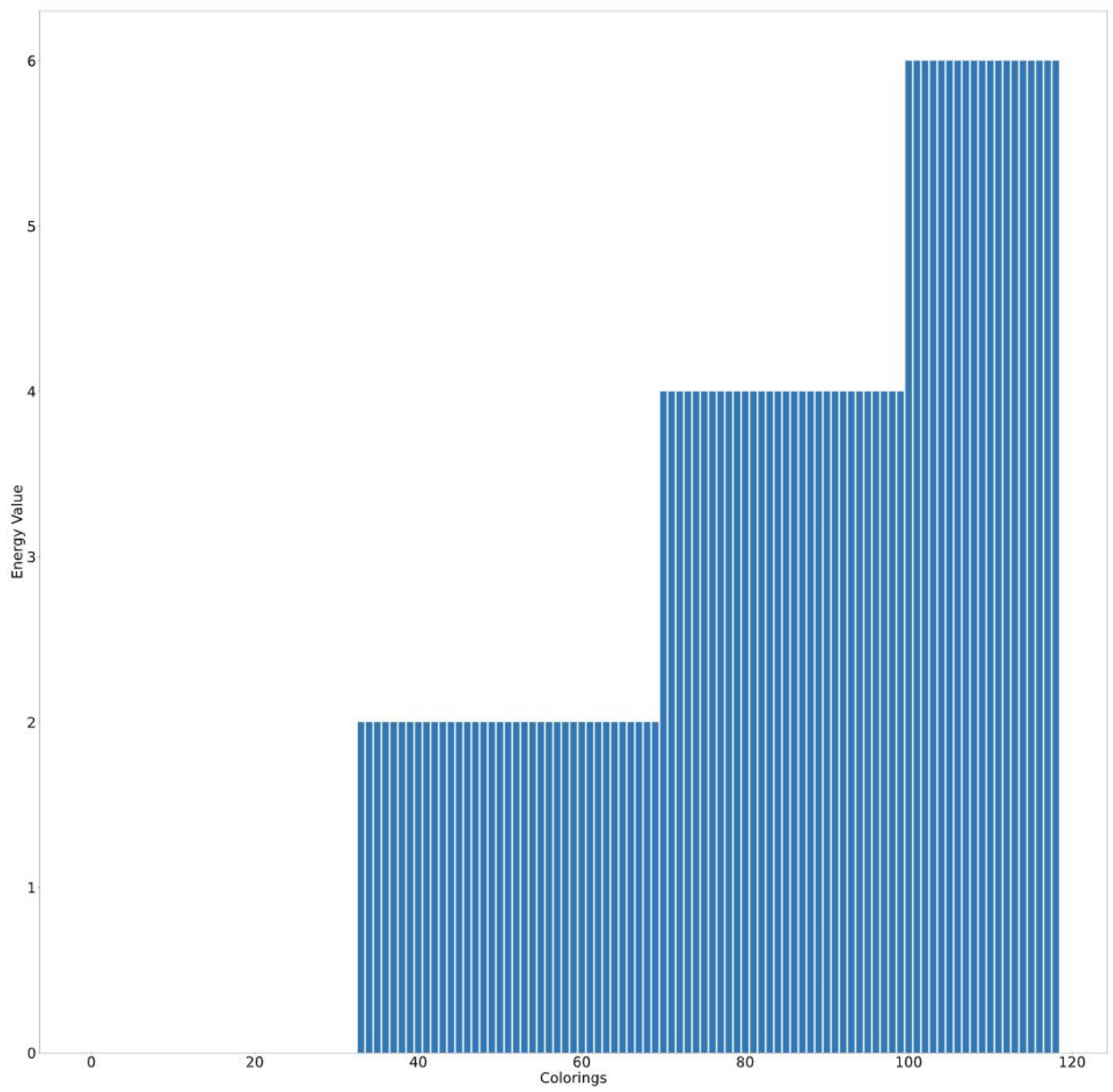


Figure 6: Graph of energy values for the results returned by DWaveSampler for 3-coloring

Looking at the lowest energy value result, we see that when the sample is graphed, it is indeed a valid 3-coloring.

```
Sample:  {'a0': 0, 'a1': 0, 'a2': 1, 'b0': 0, 'b1': 1, 'b2': 0, 'c0': 1, 'c1': 0, 'c2': 0, 'd0': 0, 'd1': 0, 'd2': 1}
Energy Value:  7.917846645000282e-08
```

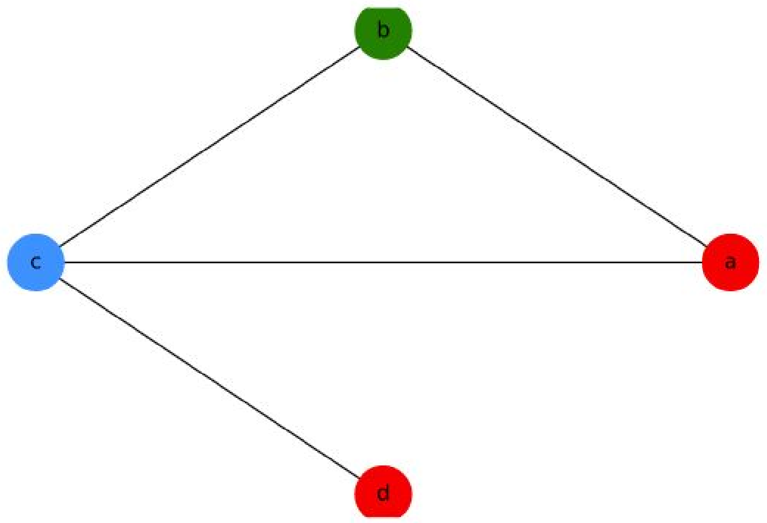
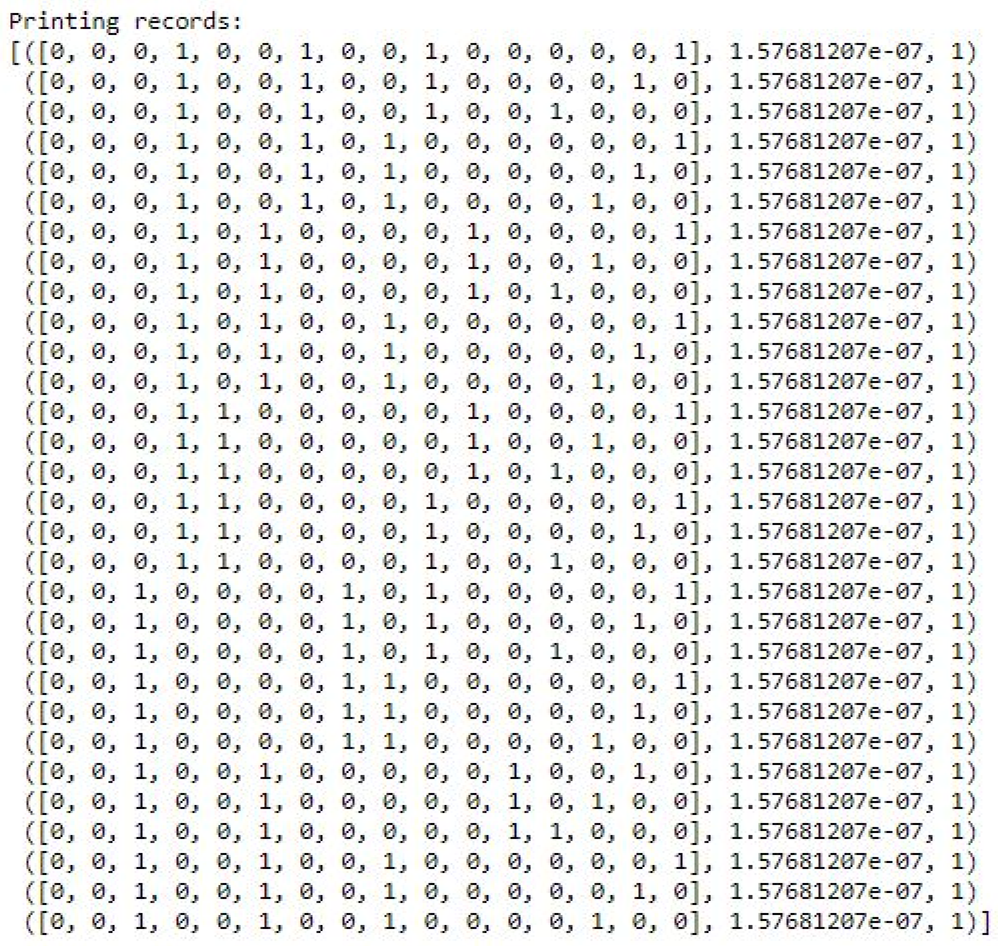


Figure 7: Valid 3-coloring from DWaveSampler

Dimod’s ExactSolver’s Results for 4-coloring

Our code was also able to find valid 4-colorings for our graph. See the first 30 samples of the simulator as well as the range of energy values for all the samples below:



```
Printing records:
[([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0], 1.57681207e-07, 1)
 ([0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0], 1.57681207e-07, 1)
 ([0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0], 1.57681207e-07, 1)]
```

Figure 8: First 30 outputs of simulator for 4-coloring

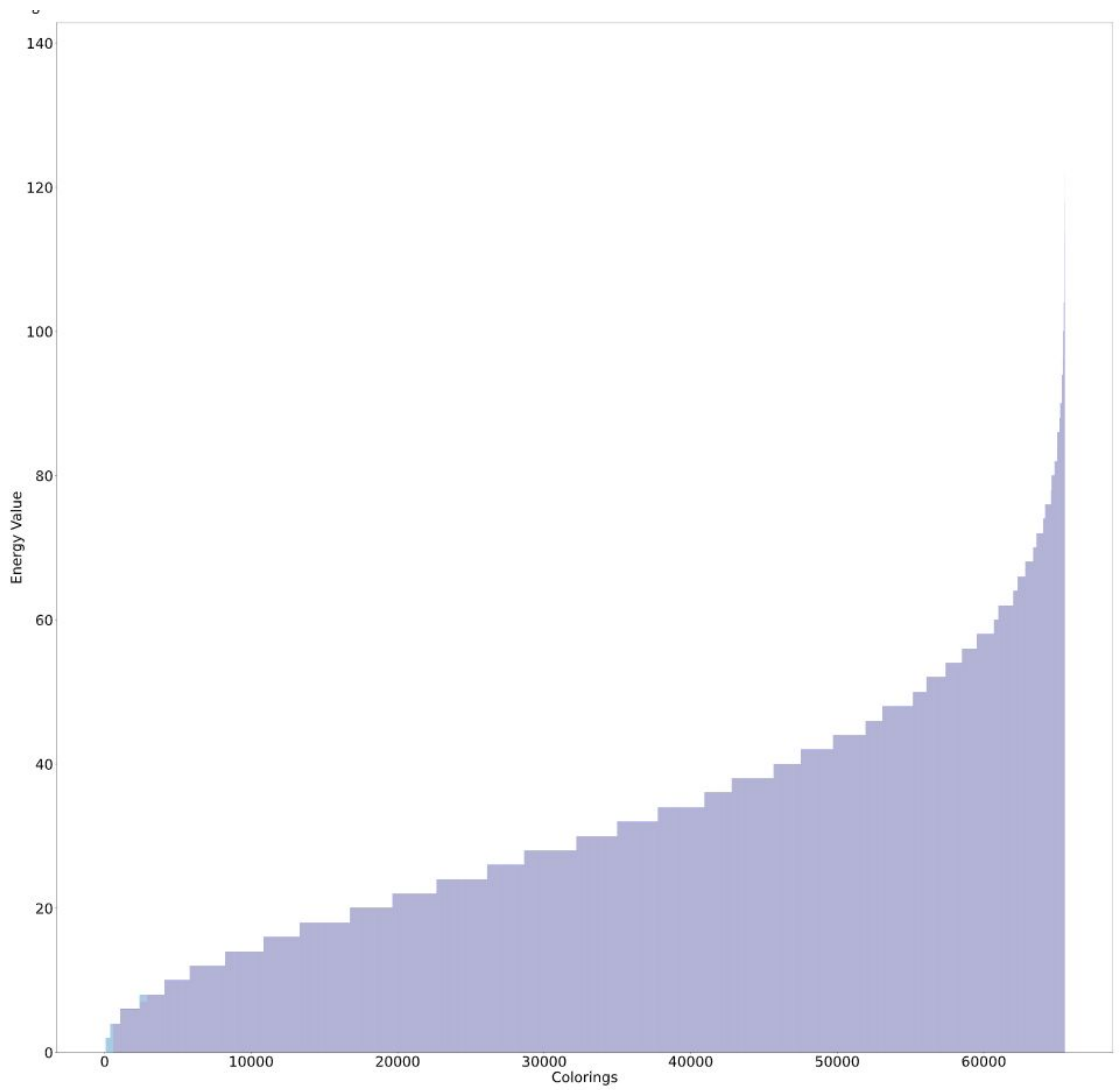
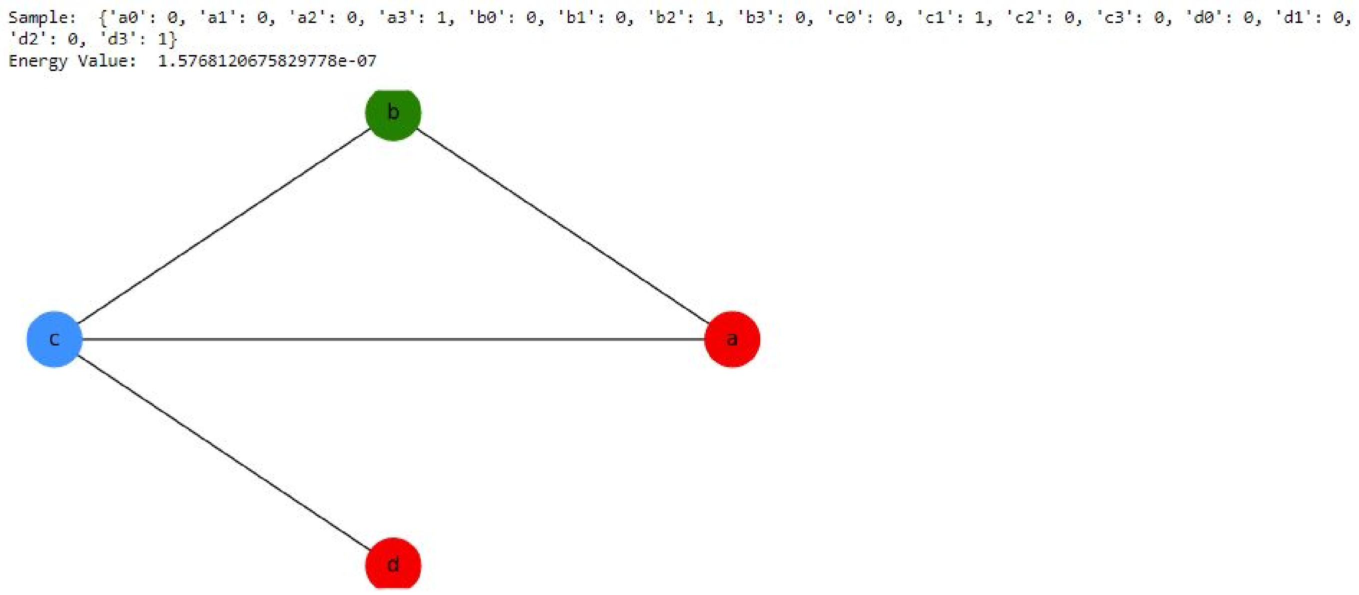


Figure 9: Range of energy values from output of simulator for 4-coloring

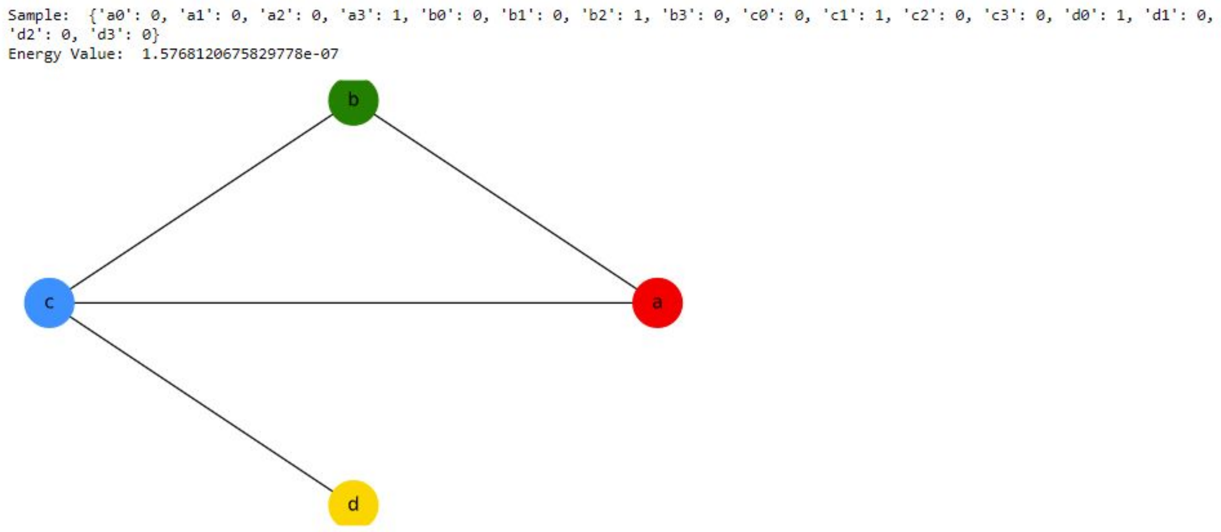
The sample corresponding to the lowest energy value, when graphed, is a valid 4-coloring of the graph, albeit by only using 3 colors.



```
Sample:  {'a0': 0, 'a1': 0, 'a2': 0, 'a3': 1, 'b0': 0, 'b1': 0, 'b2': 1, 'b3': 0, 'c0': 0, 'c1': 1, 'c2': 0, 'c3': 0, 'd0': 0, 'd1': 0,
'd2': 0, 'd3': 1}
Energy Value:  1.5768120675829778e-07
```

Figure 10: Valid 4-coloring from simulator

We can also see the graph corresponding to the 3rd lowest energy value sample is also a valid 4-coloring that actually uses 4 colors.

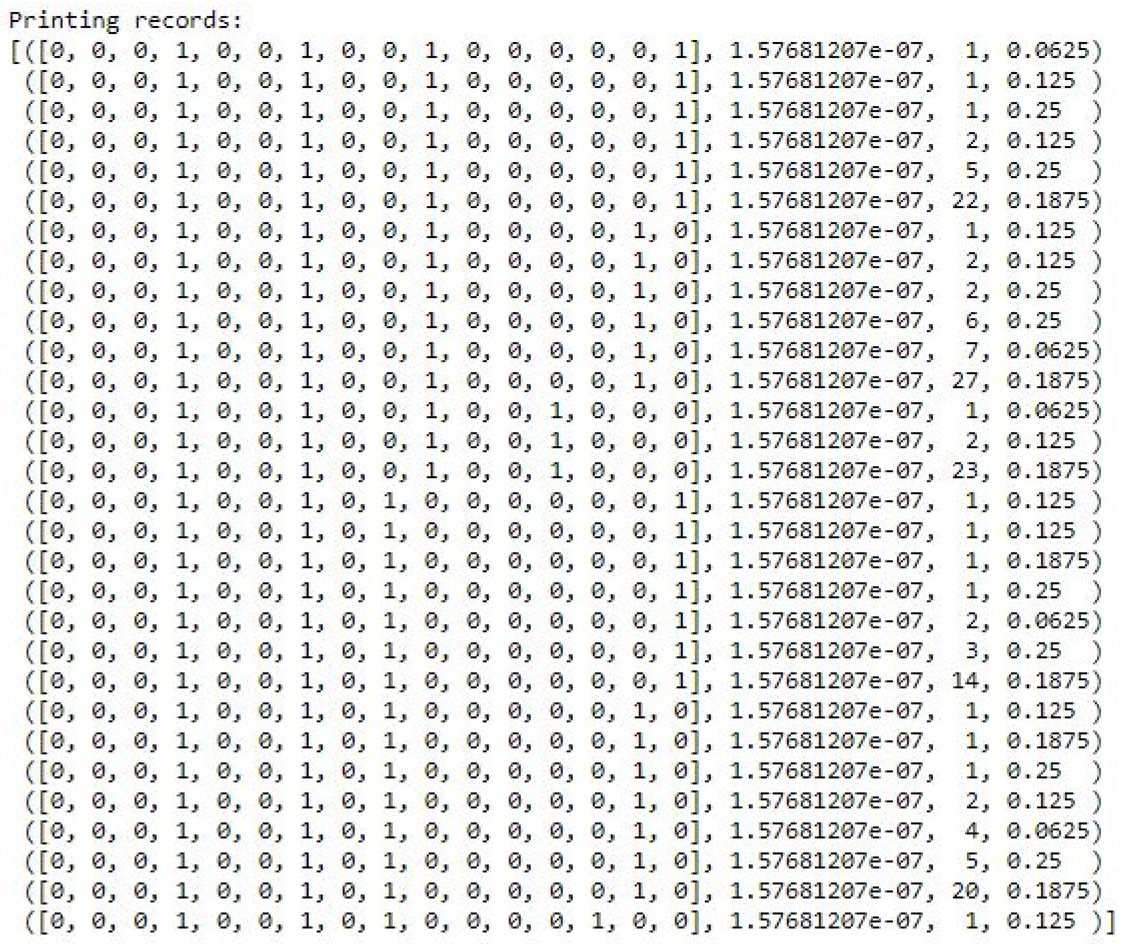


```
Sample:  {'a0': 0, 'a1': 0, 'a2': 0, 'a3': 1, 'b0': 0, 'b1': 0, 'b2': 1, 'b3': 0, 'c0': 0, 'c1': 1, 'c2': 0, 'c3': 0, 'd0': 1, 'd1': 0,
'd2': 0, 'd3': 0}
Energy Value:  1.5768120675829778e-07
```

Figure 11: Valid 4-coloring from simulator using 4 colors

DWaveSampler’s Results for 4-coloring

When running our code on real quantum hardware, we were again successful in finding 4-colorings for the graph. We can examine the lowest energy values by looking at the first 30 samples returned from the sampler.



```
Printing records:
[([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1], 1.57681207e-07,  1, 0.0625)
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1], 1.57681207e-07,  1, 0.125 )
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1], 1.57681207e-07,  1, 0.25  )
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1], 1.57681207e-07,  2, 0.125 )
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1], 1.57681207e-07,  5, 0.25  )
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1], 1.57681207e-07, 22, 0.1875)
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0], 1.57681207e-07,  1, 0.125 )
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0], 1.57681207e-07,  2, 0.125 )
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0], 1.57681207e-07,  2, 0.25  )
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0], 1.57681207e-07,  6, 0.25  )
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0], 1.57681207e-07,  7, 0.0625)
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0], 1.57681207e-07, 27, 0.1875)
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0], 1.57681207e-07,  1, 0.0625)
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0], 1.57681207e-07,  2, 0.125 )
 ([0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0], 1.57681207e-07, 23, 0.1875)
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1], 1.57681207e-07,  1, 0.125 )
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1], 1.57681207e-07,  1, 0.125 )
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1], 1.57681207e-07,  1, 0.1875)
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1], 1.57681207e-07,  1, 0.25  )
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1], 1.57681207e-07,  2, 0.0625)
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1], 1.57681207e-07,  3, 0.25  )
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1], 1.57681207e-07, 14, 0.1875)
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0], 1.57681207e-07,  1, 0.125 )
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0], 1.57681207e-07,  1, 0.1875)
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0], 1.57681207e-07,  1, 0.25  )
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0], 1.57681207e-07,  2, 0.125 )
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0], 1.57681207e-07,  4, 0.0625)
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0], 1.57681207e-07,  5, 0.25  )
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0], 1.57681207e-07, 20, 0.1875)
 ([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0], 1.57681207e-07,  1, 0.125 )]
```

Figure 12: first 30 lowest energy value samples returned from DwaveSampler

We can look at the graph of the energy values of all the samples and see the transition from near zero energy values to energy values around 2.

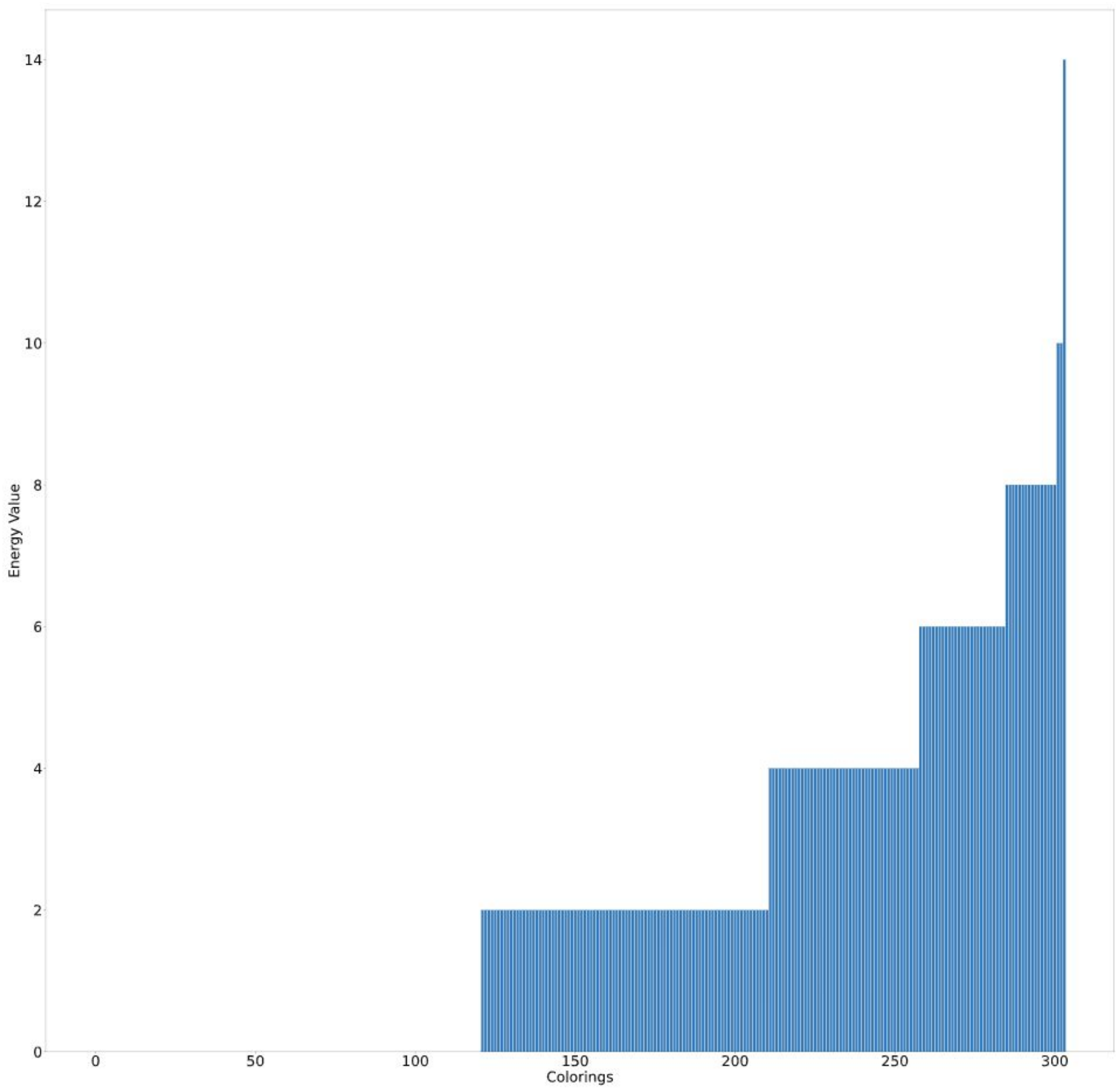
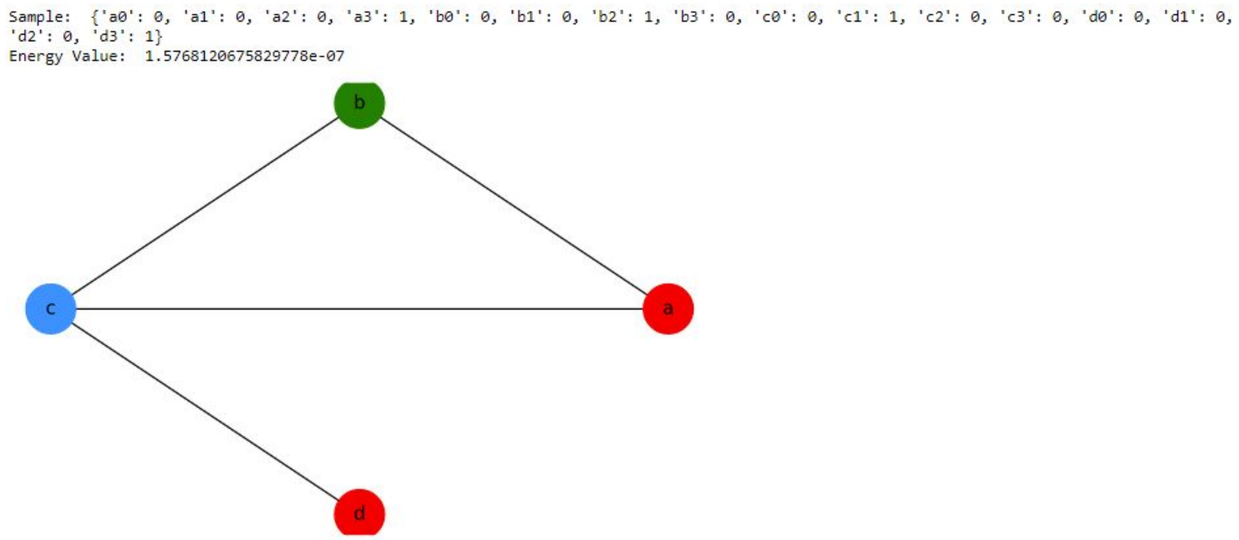


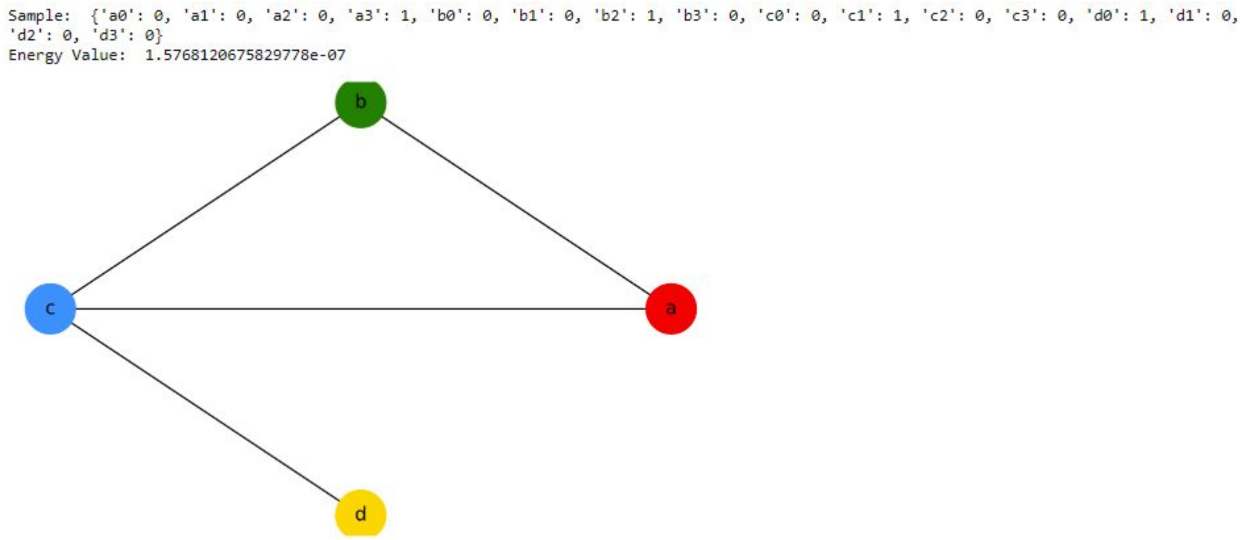
Figure 13: Range of energy values from output of DWaveSampler for 4-coloring

We also see that the sample corresponding to the lowest energy value results in a valid 4-coloring, albeit while only using 3 colors. And we can see that the 13th lowest energy value sample corresponds to a valid 4-coloring that uses all 4 colors.



```
Sample:  {'a0': 0, 'a1': 0, 'a2': 0, 'a3': 1, 'b0': 0, 'b1': 0, 'b2': 1, 'b3': 0, 'c0': 0, 'c1': 1, 'c2': 0, 'c3': 0, 'd0': 0, 'd1': 0,
'd2': 0, 'd3': 1}
Energy Value:  1.5768120675829778e-07
```

Figure 14: Valid 4-coloring from DWaveSampler



```
Sample:  {'a0': 0, 'a1': 0, 'a2': 0, 'a3': 1, 'b0': 0, 'b1': 0, 'b2': 1, 'b3': 0, 'c0': 0, 'c1': 1, 'c2': 0, 'c3': 0, 'd0': 1, 'd1': 0,
'd2': 0, 'd3': 0}
Energy Value:  1.5768120675829778e-07
```

Figure 15: Valid 4-coloring from DwaveSampler using 4 colors

Gate-Based Quantum Computing for Graph Coloring

There are several techniques which can be used to perform graph coloring on the IBMQ gate based machine. One of these techniques is quantum approximation optimization algorithm (QAOA). The QAOA method first derives the objective function for the graph and determines the constraints to impose on the graph. The hamiltonian matrix can be constructed, and used it to derive the appropriate ising model [9]. An alternative technique is VQE (Variational Quantum Eigensolver) which is a hybrid algorithm which attempts to find eigenvalues of the hamiltonian matrix for the problem. Of the various techniques discussed above the generalized quantum method will be used to solve the combinatorial k-coloring problem.

The generalized quantum method for the K-coloring problem which has N nodes (regions on the map) will require somewhere in the order of O(N^2) and O(N^4) gates to formulate the problem[8]. The method used is a modified version of the one discussed in the HAL paper by Maurice Clerc. For the 3-coloring, 2 are required for each node, and 4 ancilla qubits are required, these ancilla qubits are reused. Representing each constraint/edge on the graph requires 10 CCX-gates and 8 X-gates and 4 reset-gates. This means the number of gates increases linearly with respect to the number of edges in the graph. The 4-coloring also required the same number of gates, but additional constraints can be pre-computed to add additional requirements to the graph. This will increase the number of gates required. Like the D-Wave method, the size of the problem is a major consideration when attempting to formulate a graph coloring problem with a generalized quantum algorithm due to the physical limitation of quantum computers[8]. The algorithm works by defining a constraint everywhere there exists an edge between two nodes. Additionally, the algorithm calculates constraints between two vertices with a common neighbour. This allows the coloring to be unique across multiple nodes.

The 2-color graph coloring problem was attempted on specific predefined graphs. From this, it was established that the |1> and the |0> states can be used to represent the two colors. With this finding, it was established that to represent k colors will require $\log_2(k)$ number of qubits for each node. To promote unique coloring of the graph, specific transitions are defined when two nodes connected by an edge are encountered. An example of a constraint is shown in the figure below.

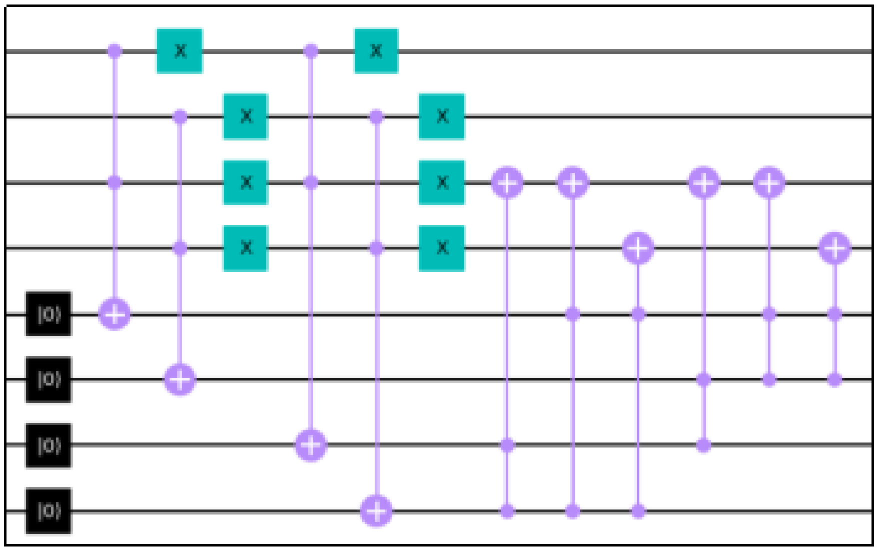


Figure 16 : Show the constraint between two nodes (the first 4 qubits - 2 each) as represented as a qiskit circuit for 3/4 coloring.

3-Coloring

The input qubits are set into a superposition state which represent the nodes in the graph. The input qubits also comprise a unitary which attempts to remove the |11> state from the circuit [8]. Even on a graph which cannot be colored, the algorithm returns an output, which will have to be verified using a classical algorithm.

A major problem for a large number of colors is to encode the problem/solution in the superposition states which can then be used to color the graph. For 3 and 4 colorings, the encoding required representing unique colors using the |00>, |01>, |10>, and |11> states. All the results obtained for the 3-coloring gate based model relate to the following 4 node graph.

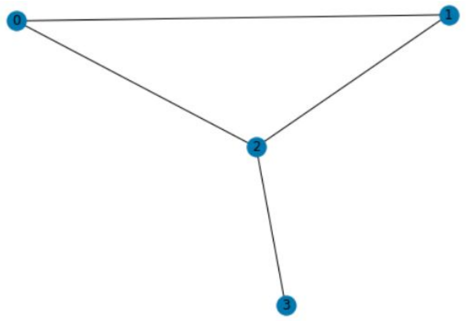


Figure 17: The graph on which all the tests and experiments for 3-coloring were conducted.

The algorithm was run both on the QASM simulator and IBMQ_paris machine. The simulation was run with a noise model acquired for the IBMQ_paris machine to keep the simulator consistent with the real hardware. The figures below show the results obtained for the specific graph shown.

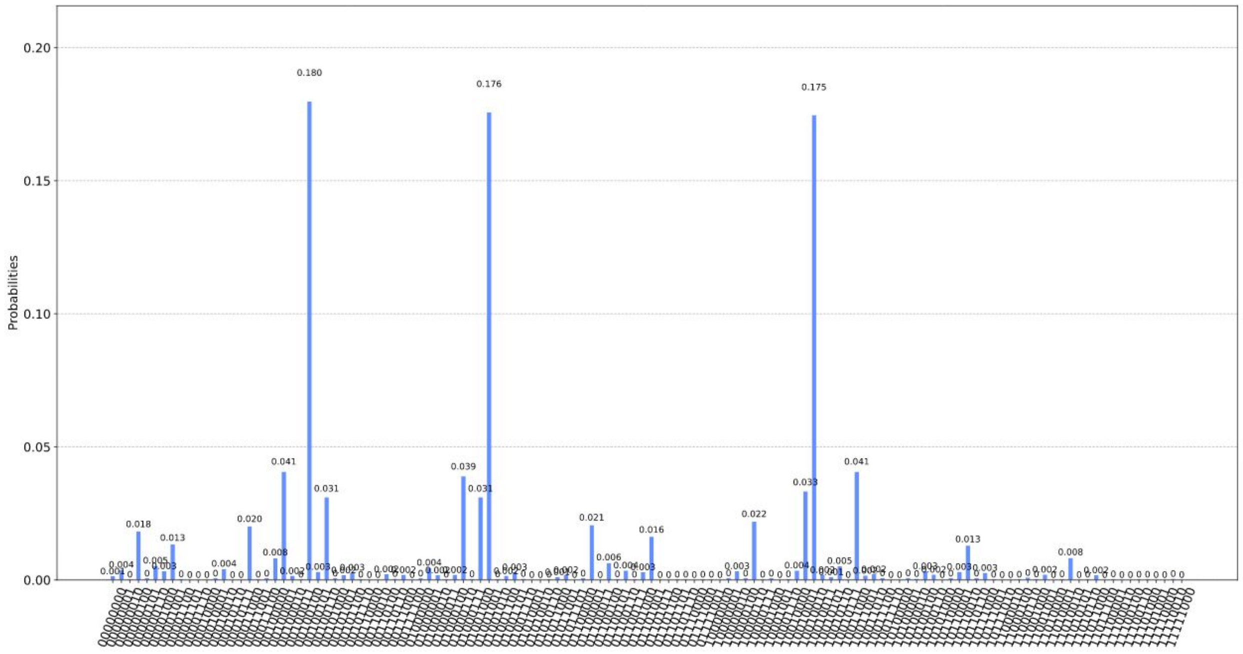


Figure 18: Results from the simulation with IBMQ_paris noise model

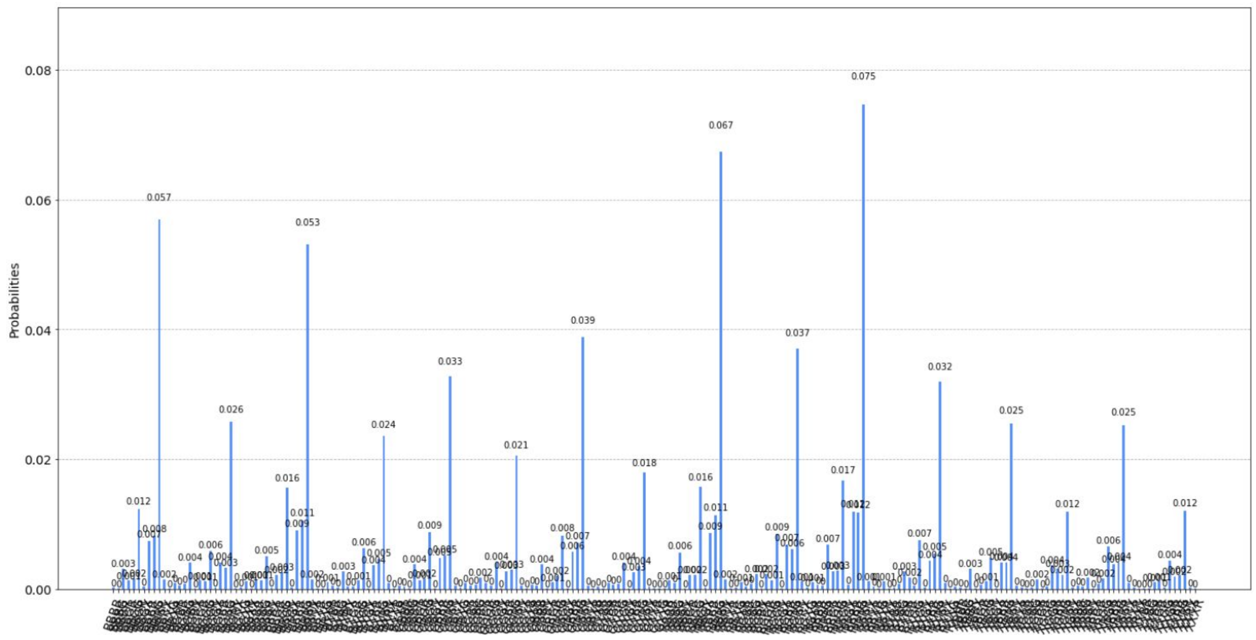


Figure 19: Results from the real IBMQ_paris machine before filtering invalid results

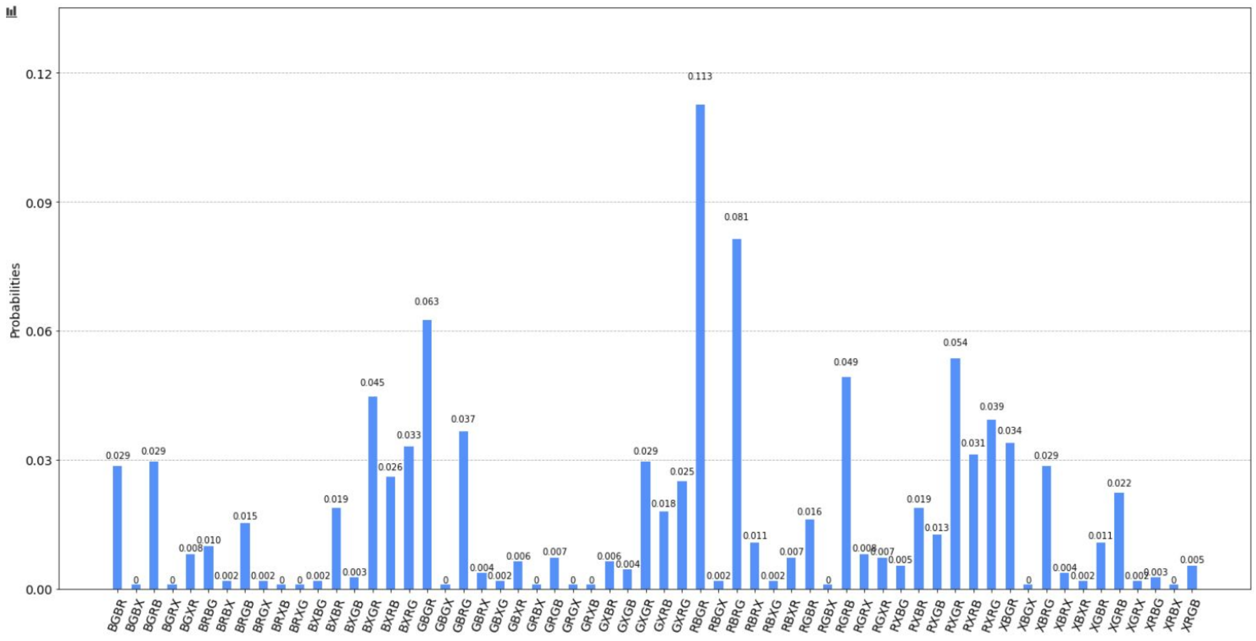


Figure 20: Results from the real IBMQ_paris machine after filtering invalid results

The results obtained for the simulator are much better compared to the real system. On the simulator, the results do not need to be filtered to obtain the valid coloring. The filtering performed is simply a linear pass through the edges of the graph to see if the two vertices at either end of the edge are colored the same. Doing this successfully yields a valid coloring on real hardware as the highest probability. The highest probability is always a 3 coloring. Specific results can be obtained in the jupyter notebook 3-coloring-qiskit-real.ipynb on the github page.

4-Coloring

The circuit for the 4 coloring is the same as the circuit for the 3 coloring except the unitary to eliminate the |11> state is no longer used. The graph used to perform the 4-coloring tests is shown below.

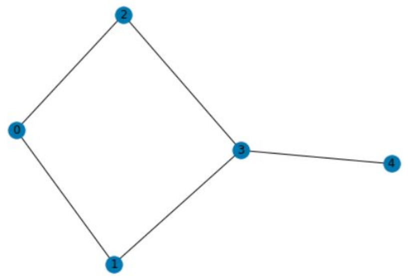


Figure 21: The graph used to perform the 4-coloring on.

The results for the 4-coloring show a similar discrepancy between the simulated with noise modelling and the real hardware. The simulation was run with the noise model obtained from the IBMQ_paris machine and the real hardware tests were conducted on the IBMQ_paris machine. The results can be seen in the figures below, where once again the results from the real hardware need to be filtered to obtain the valid colorings. Due to the nature of this graph, we see that because a 2 coloring can satisfy the graph, the highest result after filtering is actually a 2 coloring. But looking further down the results we observe the 4 coloring, the specific details and plots viewed in the jupyter notebook named 4-coloring-qiskit-real.ipynb on the github page.

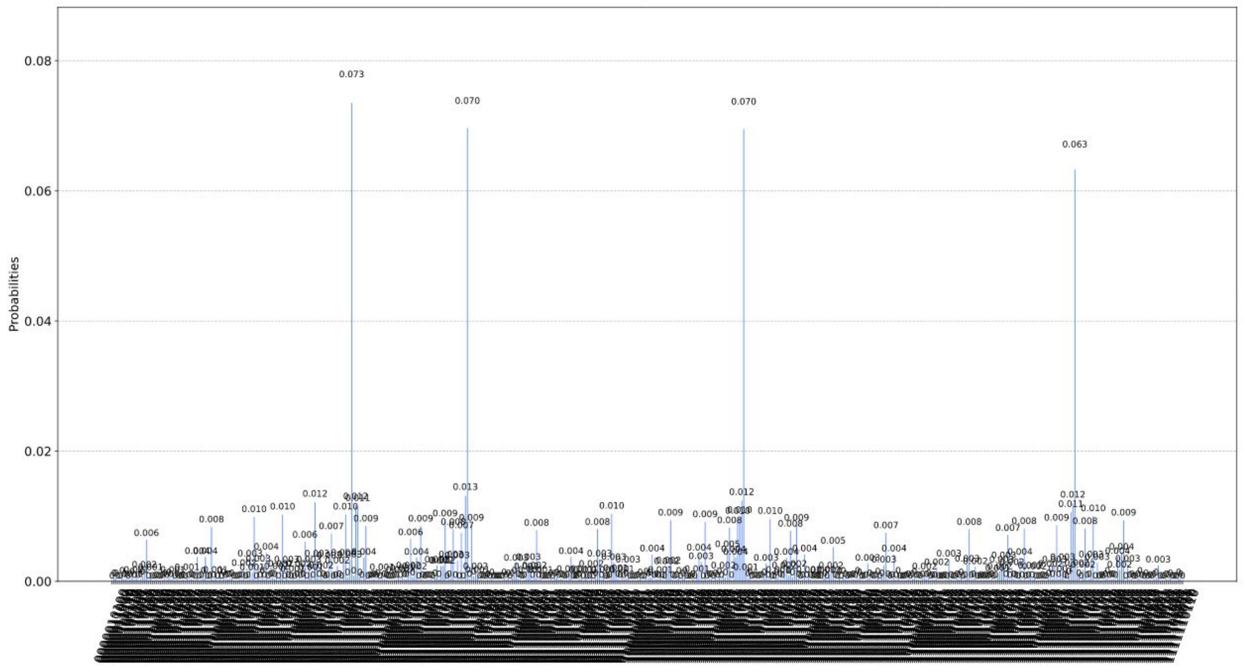


Figure 22: The results from the simulator for the 4-coloring graph shown on Figure 21.

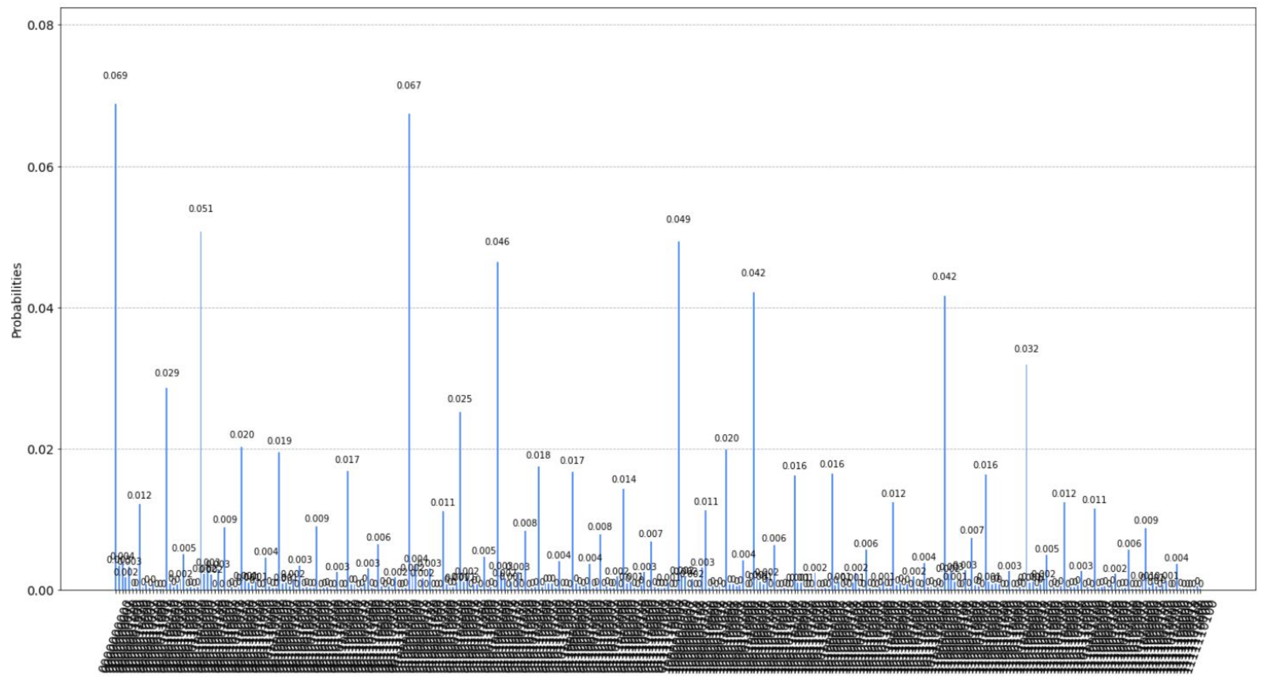


Figure 23: The results from the real hardware for the 4-coloring graph shown in Figure 21.

Comparing and Contrasting Results between Gate-Based and Annealing

Accuracy of results

Overall, the D-Wave annealing results provided much more accurate results. The lowest energy values output by the sampler consistently corresponded to valid colorings for both 3-colorings and 4-colorings. In contrast, with the IBM gate-based system, the highest probable colorings corresponded to invalid colorings for both 3-colorings and 4-colorings. However, once the results were filtered to remove any colorings with two adjacent colors being the same, we were then able to get a valid 3 and 4-coloring.

Runtime

To compare runtime, we created 5 graphs as seen below and tried to find a 4-coloring for each graph. The time required to get a 4-coloring output for each system was then plotted.

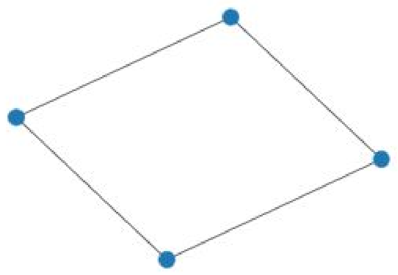
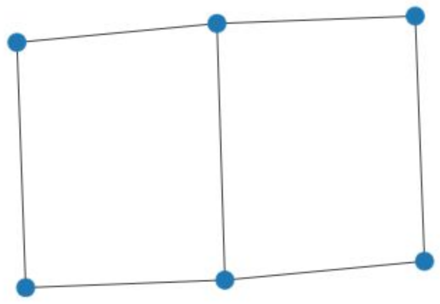



Figure 24: Graph 1 and Graph 2

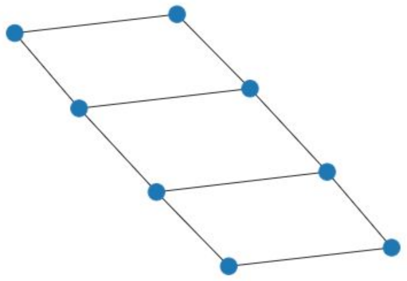
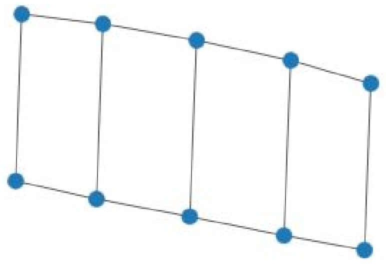



Figure 25: Graph 3 and Graph 4

Graph	Nodes + Edges	Qiskit (seconds)	D-Wave (seconds)
Graph 1	4 + 4	23.17	1.922545
Graph 2	6 + 7	29.22	1.922545
Graph 3	8 + 10	33.67	1.922509
Graph 4	10 + 13	40.84	1.922546

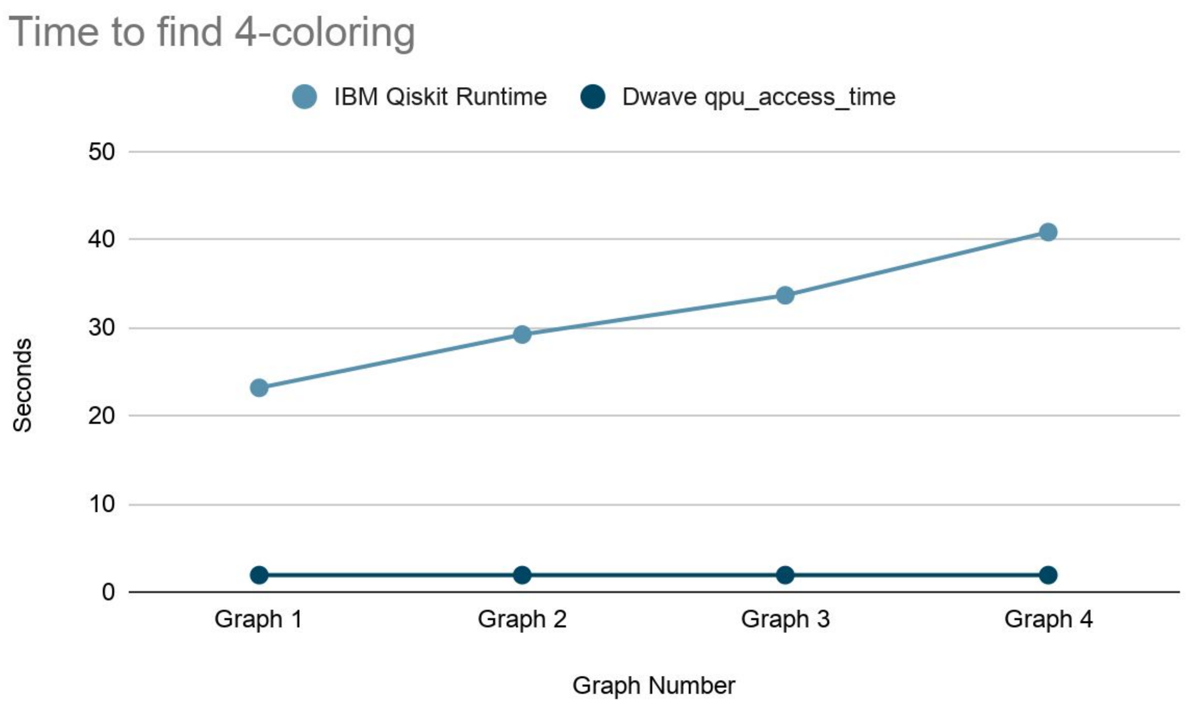


Figure 26 - Graph of increasingly bigger graphs vs. runtime to find 4-colorings

We can see that the gate based system's required time to find a 4-color for each graph was significantly higher than the annealing machine's time. We can also see that as the graph size increased, the runtime for the gate-based machine to find a 4-coloring increased significantly. This was not the case with the annealing machine. Scaling up the size of the graph didn't affect the qpu_access_time significantly.

Design

On the gate-based machine, the design for this 3-coloring and 4-coloring is almost identical except for the unitary in the 3-coloring used to remove the |11> state. Both colorings apply constraints between adjacent nodes and nodes with the same adjacent neighbour. Both perform the same transition when two nodes are found to have the same color. If two nodes have |00> state, one of the nodes will transition to the |01>, if both nodes have a |01> then one will transition to |10>, if both are |10> then one will transition to |11> and finally, if both are |11> then one will transition to the |00> state. Like the program for the D-wave machine, the circuit represents the constraints between the nodes. The larger the graph, the more constraints will be generated. There are two types of constraints, the explicit ones, which are the edges defined in the graph and the implicit ones, the ones generated by the classical algorithm which converts the graph into a constraint satisfaction problem. The specific algorithm implementation can be viewed on github.

The implementation of the graph coloring algorithm on the D-Wave machine is similar to the implementation we used for the gate-based machine in that it is based on constraints. In the implementation for the annealing machine, we create a constraint satisfaction problem that is then converted into a BQM. However, unlike the implementation done in Qiskit, we do not have direct transitions between colors in the implementation for the annealing machine. Instead, the colors are allowed to transition to any other color and the energy value will either be lowered or raised depending on if that transition met the constraints defined in the CSP.

Overall, the design and implementation for the D-Wave annealing machine was simpler and more intuitive and is thus our preferred method for solving combinatorial optimization problems.

Scalability

The qiskit machine, specifically the IBMQ_paris and IBMQ_toronto are capable of circuits with up to 27 qubits, this means for the 3 and 4 graph coloring the algorithms it can handle up to 10 nodes. The limitation is solely dependent on the number of nodes and not the number of edges in the graph because the ancilla qubits are being reused courtesy of the reset gate which was recently introduced by qiskit. If the size of the problem is too large then real machines will not be able to compute it, the same applies to the QASM simulator, it can only handle up to 32 qubits which means a problem of 14 nodes and any number of edges.

The dimod ExactSolver is able to run calculations extremely fast on problems that involve up to 16 variables. The number of variables/qubits required is equal to the number of nodes multiplied by the number of colors. So this quantum simulator would be able to handle our 4 node graph with up to 4 colors.

The real D-Wave annealing machine we ran our code on was DW_2000Q_6 which has 2041 qubits so the size of problems the annealing machine can handle is significantly larger than the gate-based machine.

Extensions to our Graph Coloring Problem

D-Wave Annealing Extensions

Using the code we wrote to handle finding a k-coloring for any k value of a graph, we were able to find a 3-coloring solution to the graph coloring problem of Canada’s provinces. The D-Wave documentation’s example shows a 4-coloring for this graph, but we were able to find a valid coloring that uses 1 less color.

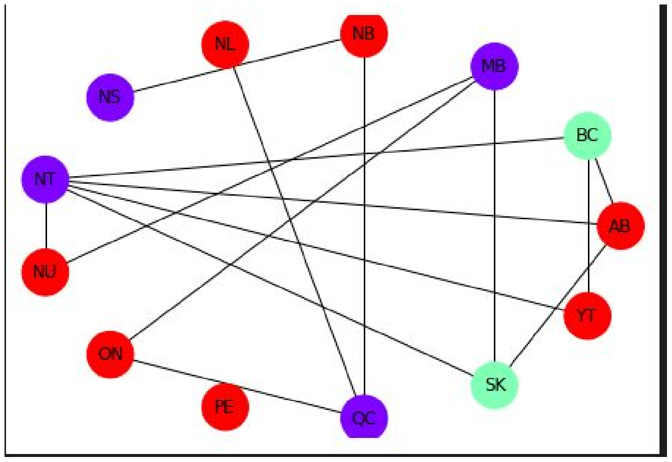


Figure 27 - 3-coloring for Canada's provinces

Another extension to the graph coloring problem we explored was finding the chromatic number. The chromatic number is the minimum k value such that a valid k-coloring exists for a graph. This chromatic number problem also has many useful applications including in the scheduling application mentioned earlier. The chromatic number corresponds to the minimum number of timeslots required to complete all the required jobs such that no conflicts between shared resources occur. The output from running our code in ChromaticNumber.py shows that 2 is not the chromatic number since there isn't a 2-coloring but that 3 is the chromatic number since 3 is the minimum k value such that a k-coloring exists.

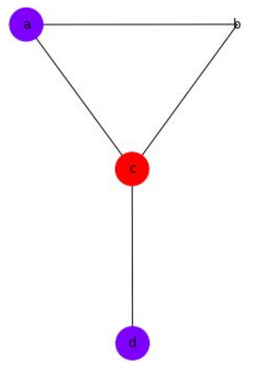


Figure 28 - 2 is not the chromatic number

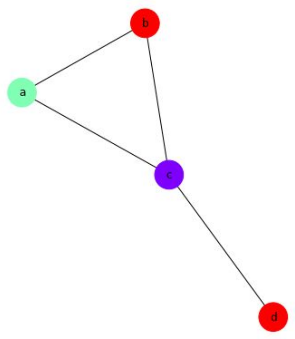


Figure 29 - 3 is the chromatic number

Qiskit Gate-Based Extensions

An aspect we further explored was the impact of removing the unitary from the 3-coloring and observing the impact it has on the results obtained.

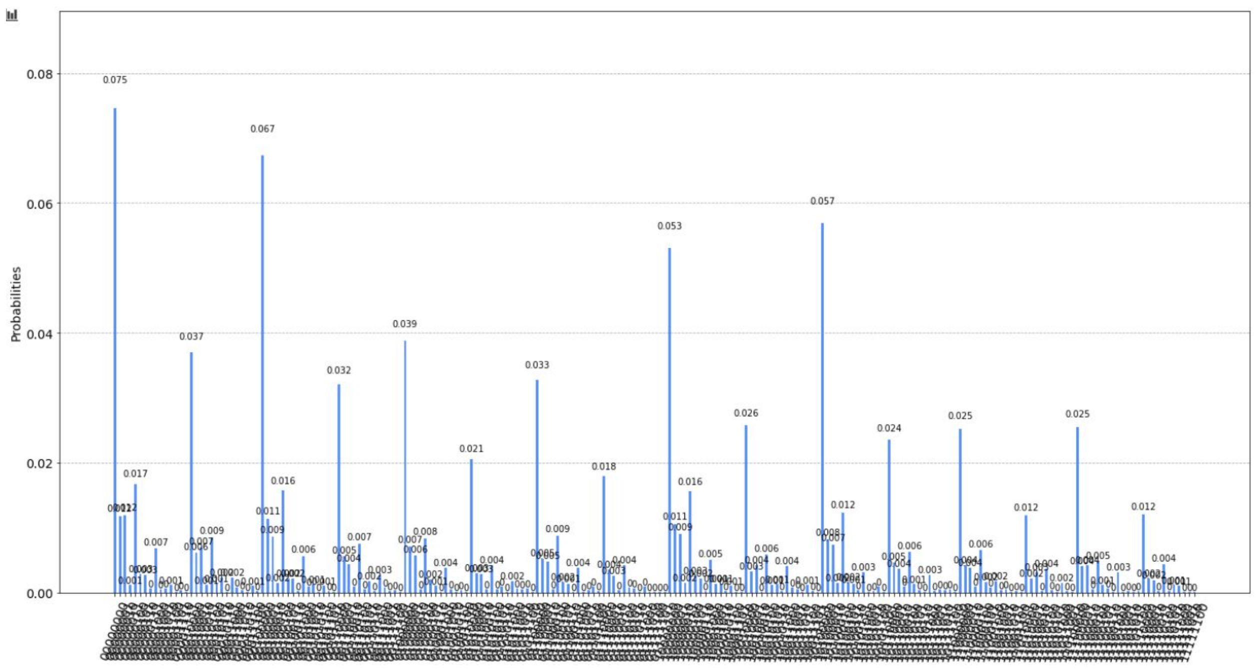


Figure 30: Results obtained on IBMQ_paris for graph on Figure 1 with unitary to remove |11> state

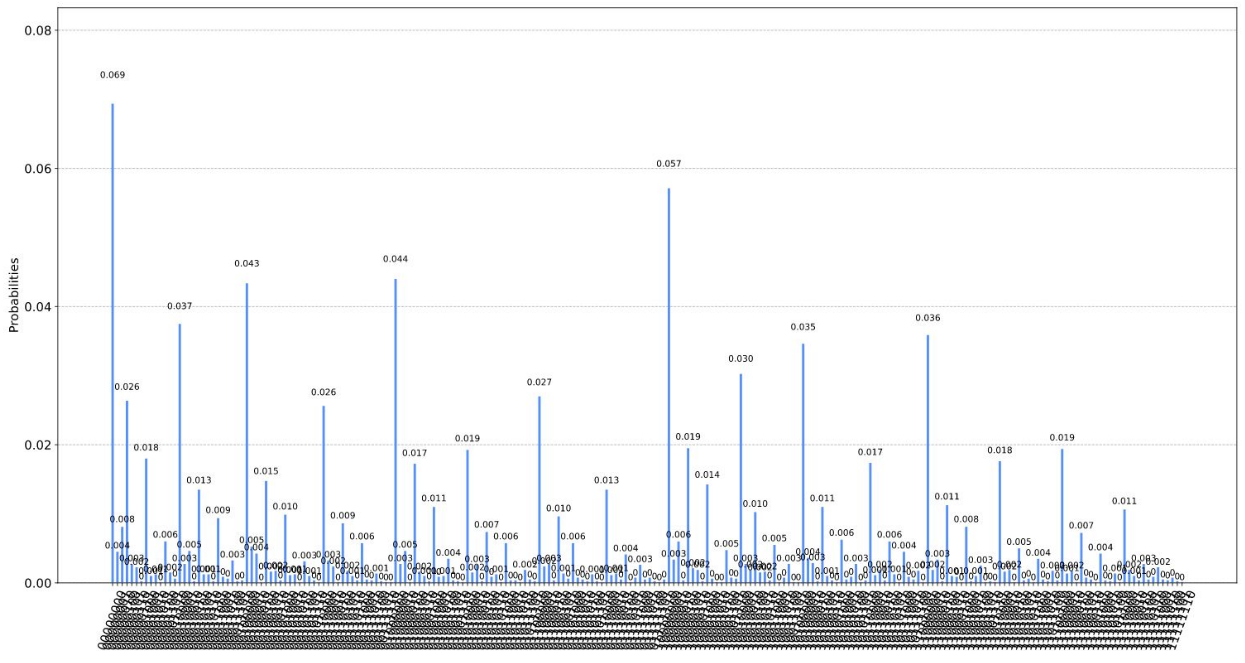


Figure 31: Results obtained on IBMQ_paris for graph on Figure 1 without unitary to remove |11> state. More detailed results can be found on github in the file 3-coloring-qiskit-real.ipynb.

Looking at the results in Figure 30 and Figure 31 it is clear to see that output does not vary as expected, there is a small difference but without further testing and statistical analysis, no significant assertions about the effect of the unitary can be made. There is a potential explanation for this, the circuit for the 3-coloring does include constraints to ensure if two nodes with |11> states are adjacent to each other then one of them will transition to the |00> state. To confirm this hypothesis further investigation would be required to prove this.

Another aspect of the gate based model we explored was to utilize the transpiler optimizations to try and reduce the amount of noise and error on real hardware. The initial findings suggested that the results were not affected by the optimization level. Three levels were compared, the level 1 which is the default when no explicit optimization is requested, level 2 and level 3. The optimizations were tested on the 3 coloring version of the code, the results can be observed below. Looking at Figure 32, 33, and 34 we observe that the results do not improve. In fact the results seem to get worse in the sense that the probability of invalid colorings is higher. This is more evident when we looked at the count of the unfiltered results and observed higher counts for invalid colorings, although when the filtered results were compared the counts for valid colorings are the same across all optimization levels. This suggests that the probability from other invalid states are being transferred between other invalid states. So overall despite the results initially looking worse, the count for valid colorings which we care about are not affected. More detailed results of this test can be obtained on the github page under the three files 3-coloring-qiskit-real-OL1.ipynb, 3-coloring-qiskit-real-OL2.ipynb, 3-coloring-qiskit-real-OL3.ipynb.

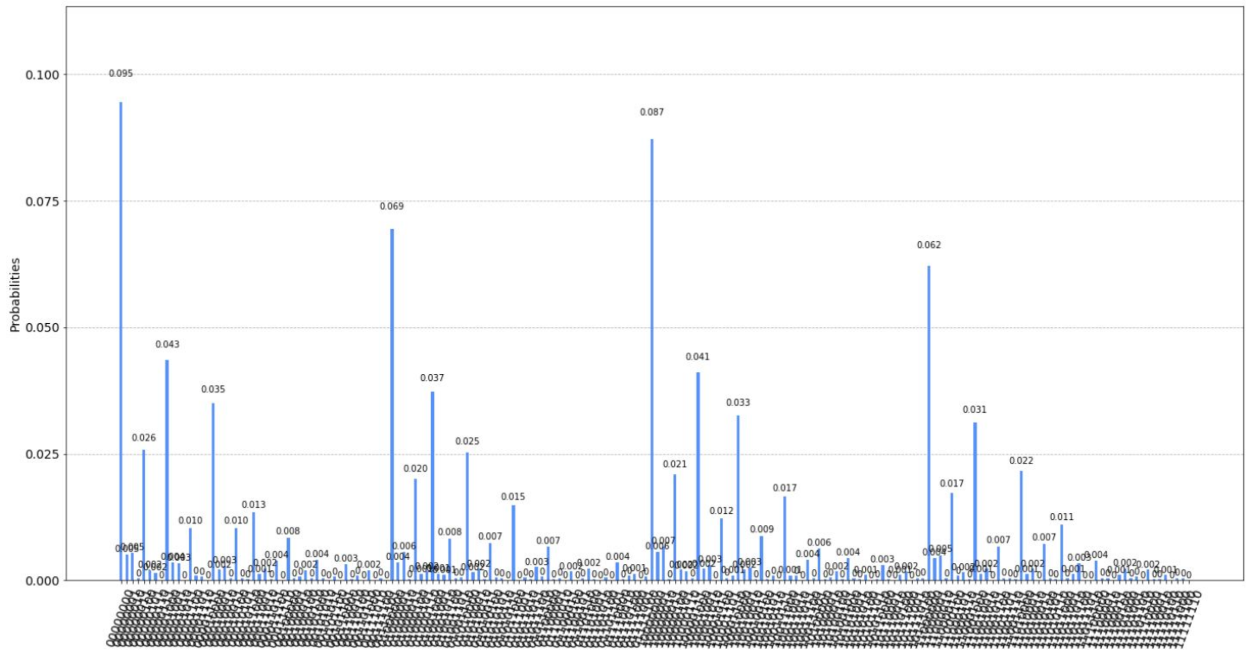


Figure 32: The results of 3-coloring of Figure 1 with optimization level 1

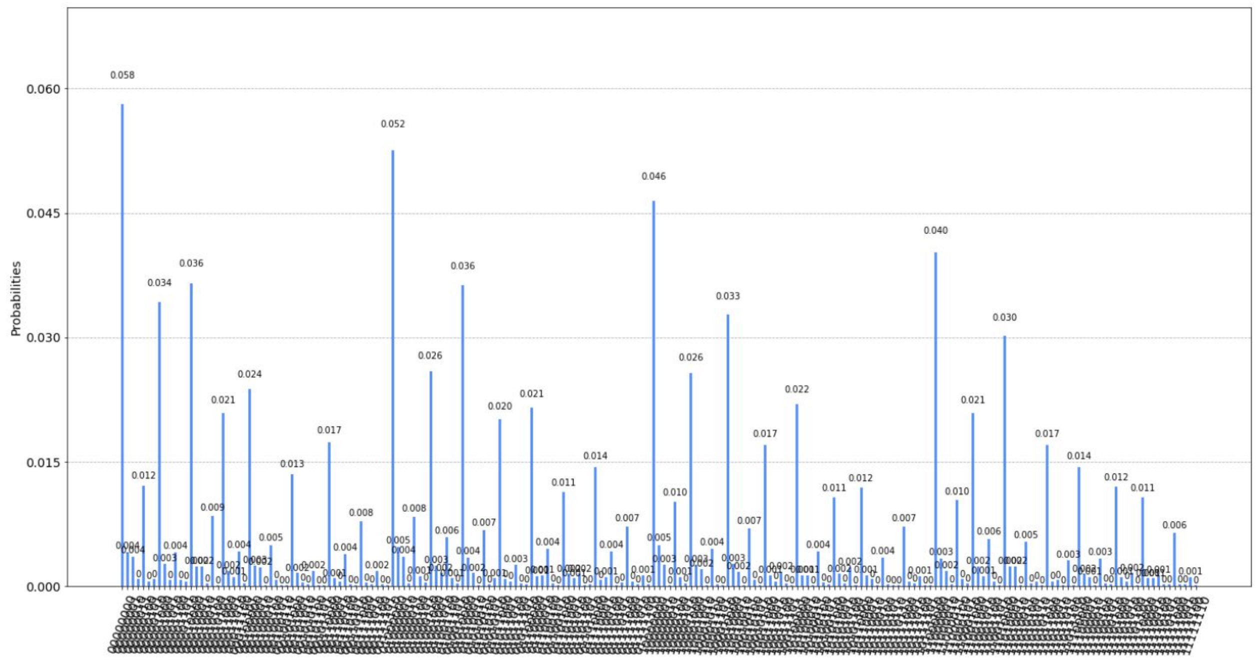


Figure 33: The results of 3-coloring of Figure 1 with optimization level 2

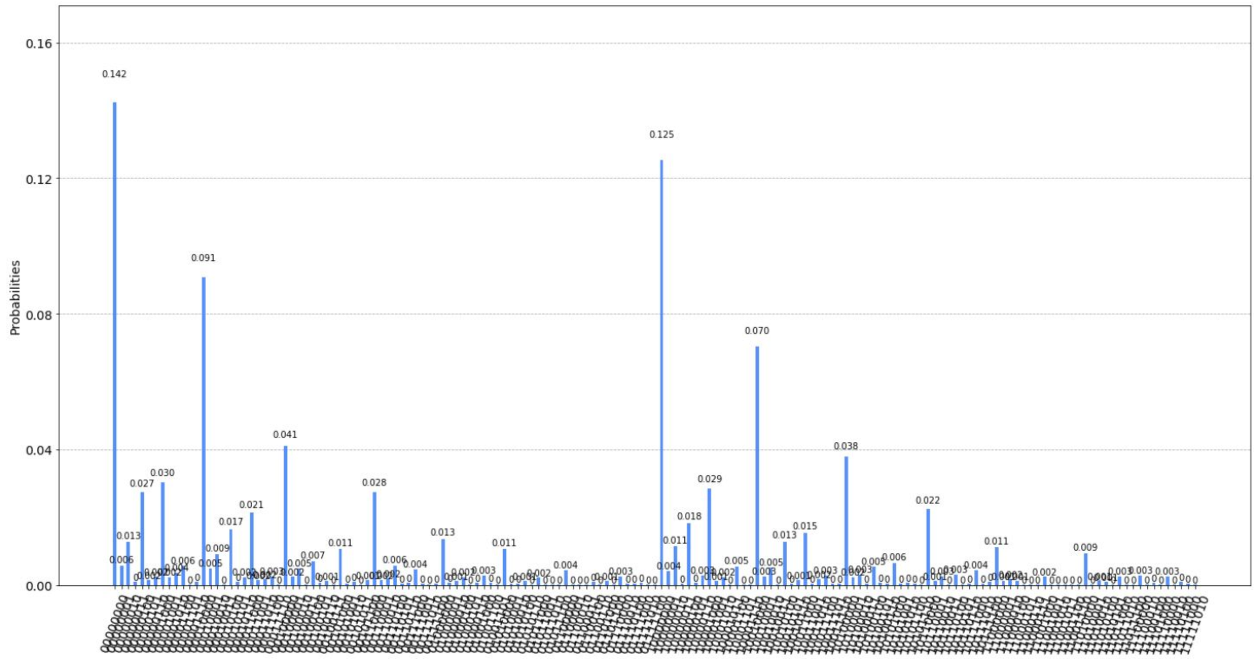


Figure 34: The results of 3-coloring of Figure 1 with optimization level 2

Further Research

One of the main things to look into next is to pinpoint the cause of the error on the IBM Q real hardware and attempt to reduce it. To aid in this endeavor it would be beneficial to perform error modeling in the QASM simulator and compare with the hardware to try and narrow down the cause of errors and potentially find more reliable alternatives. Another way to better understand the error in real hardware would be to use custom mapping of qubits to reduce the amount of error which seems to be propagating through the circuit.

Furthermore, on both the gate-based and quantum annealing machines, exploring the minimum number of measurements required to obtain a statistically reliable result would be interesting to see how much time can be reduced from the runtime due to this.

Another aspect we would like to explore further is to compare the generalized method on the qiskit machine with VQE and QAOA [11] in terms of accuracy of results, time complexity, minimum number of measurements required to obtain reliable results.

We implemented a very inefficient way of calculating the chromatic number, so further research should definitely be dedicated to finding more efficient algorithms for this. One such quantum algorithm involves splitting up the graph into maximum independent sets and using grover's algorithm to search those for the minimum of their chromatic numbers which is then used to derive the overall chromatic number for the graph.

In addition, it would be interesting to see if we could find the total number of valid k-colorings for a graph. Classical approximation approaches involve Markov Chain Monte Carlo and random walks on graphs so it would be interesting to see how a quantum approach and algorithm differs and improves on the classical approaches.

Individual Contributions

In the initial stages of the project, both Nachiket and Carl contributed to finding sources, reading research papers, working to narrow down the scope of the problem, creating a timeline for the aspect of our project, and working to complete the project description assignment for HW4 and the Report 2 for HW5.

Later on, we decided to divide the work between the two of us. It was easy to separate the project out into two parts, the D-Wave Annealing implementation and the IBM Qiskit Gate-Based implementation. Carl worked on all the code related to implementation on the D-Wave machine and Nachiket worked on all the code related to the IBM Qiskit implementation. We then worked together to come up with metrics and ways of comparing and contrasting our results.

Carl wrote the parts of this report corresponding to the introduction, goals, and quantum annealing approach, results, and extensions. Nachiket wrote the parts of this report corresponding to the quantum gate-based system's approach, results, and extensions to the graph coloring problem. We both contributed to the compare and contrast sections and the further research section.

Citations

[1] Marx, D., “Graph Colouring Problems and their Applications in Scheduling”. Periodica Polytechnica Ser. El. Eng. Vol. 48, No. 1, PP. 11-16, Dec. 2003.

[2] Oh, Y., Mohammadbagherpoor, H., Dreher, P., Singh, A., Yu, X., & Rindos, A, “Solving Multi-Coloring Combinatorial Optimization Problems Using Hybrid Quantum Algorithms,” 2019, https://arxiv.org/abs/1911.00595.

[3] Kosowski, A., & Manuszewski, K. “Classical Coloring of Graphs,” 2008. http://fileadmin.cs.lth.se/cs/Personal/Andrzej_Lingas/k-m.pdf

[4] Dwave. https://arcb.csc.ncsu.edu/~mueller//qc/qc18/readings/dwave4-map.pdf [Accessed 8 October 2020]

[5] Wang, Yang & Lü, Zhipeng & Glover, Fred & Hao, Jin-Kao. (2012). A Multilevel Algorithm for Large Unconstrained Binary Quadratic Optimization. 395-408. 10.1007/978-3-642-29828-8_26.

[6] D-Wave Systems Inc. Wave System Documentation. Retrieved November 17, 2020 from https://docs.dwavesys.com/docs/latest/index.html

[7] Dahl, E. D., D-Wave Systems, “Programming with D-Wave:Map Coloring Problem,” 2013, https://www.dwavesys.com/sites/default/files/Map%20Coloring%20WP2.pdf

[8] Maurice Clerc. A general quantum method to solve the graph K-colouring problem. 2020. hal-02891847v2

[9] Do, M., Wang, Z., O'Gorman, B., Venturelli, D., Rieffel, E., & Frank, J, “Planning for Compilation of a Quantum Algorithm for Graph Coloring” 2020, https://arxiv.org/abs/2002.10917.

[10] Banerjee, Asmita & Behera, Bikash & Das, Kunal & Panigrahi, Prasanta. (2019). Checking and Coloring of Graphs Through Quantum Circuits: An IBM Quantum Experience. 10.13140/RG.2.2.20371.22568.

[11] Stuart Hadfield, Zhihui Wang, Eleanor G. Rieffel, Bryan O'Gorman, Davide Venturelli, and Rupak Biswas. 2017. Quantum Approximate Optimization with Hard and Soft Constraints. In *Proceedings of the Second International Workshop on Post Moores Era Supercomputing (PMES'17)*. Association for Computing Machinery, New York, NY, USA, 15–21. DOI:https://doi.org/10.1145/3149526.3149530

[12] Shimizu, K., Mori, R. Exponential-time Quantum Algorithms for Graph Coloring Problems. 2019. https://arxiv.org/abs/1907.00529