Zaven and Sonia Akian
College of Science and Engineering (CSE)
BS in Computer Science

SOLVING ROUTING PROBLEMS USING QUANTUM ANNEALERS

Author:
Gevorgyan Robert

Supervisor:
Avetisyan Hakob, Ph.D.

American University of Armenia
MORE THAN AN EDUCATION- A COMMITMENT

Spring 2020

Contents

Abstract

In recent years, quantum technologies are on the rise. The usage of quantum entanglement and superposition for solving problems is appealing, especially for problems closely connected to real-world systems and atomic interactions. Besides the efforts to create a universal quantum gate model computer, which is probably a decade away, researchers also use currently available tools to solve problems. One example of such effort is adiabatic quantum computing, and especially quantum annealing. In this paper, we review the capabilities of quantum annealing and demonstrate its power for solving real-world logistics problems. First, we will look at a classical NP-hard problem - traveling salesperson problem. We will try to use D-Wave tools to find an optimal solution for the given graph structure. Then we will take a graph structure of streets and try to reduce the number of congestion on the roads. In both cases, we will first transfer the problem into a QUBO (Quadratic unconstrained binary optimization) problem and then "feed" it to D-Wave quantum annealer and get the optimal solutions.

1 Introduction

Quantum computers are a relatively new and influential field of research. In the late 1970s, researchers first started to think about using quantum phenomena to perform computation. Recently, engineering advancements in the field generated much interest and attracted many big names in IT, such as IBM, Google, and Rigetti. Quantum computers allow us to perform operations on all the possible bit-strings at the same time. In the case of a classical computer, we would need to try every possible bit combination one by one. Quantum computers can offer that speedup because of the qubit superposition. Moreover, quantum interference allows us to eliminate or boost particular possibilities and get the desired answer. Several quantum algorithms, like Shor's algorithm, already showed that a fault-tolerant quantum computer could offer an exponential speedup while solving certain types of problems. However, engineering difficulties and environmental noise are still a big challenge for building a fully functioning, fault-tolerant quantum computers. For example, state-of-the-art universal quantum computers can manipulate up to 10 logical qubits (which is to say, around 50 qubits), and are still not perfect when doing so [1]. That is why, besides addressing all the research efforts to create universal quantum computers, some researchers and companies emphasize on building special kinds of quantum computers that can only solve particular types of problems. That implies looser engineering challenges and more scalability, for example, regarding the number of available qubits. For example, state-of-the-art D-Wave chips have around 5000 qubits and will be available mid-2020. [12] One such type of specialized computer is called an annealer. What is interesting about annealers is that it uses nature to solve classical optimization problems. Mostly, we couple the qubits and let them naturally take a configuration. The taken configuration is then believed to be the solution or the approximation of the exact solution. More clearly, the qubits and their couplings tend to find the state with the lowest energy - the ground state, thus finding the minimum of the given function.
As mentioned, quantum annealers solve a special kind of problem. To formulate the broad set of the problem that quantum annealers can solve, let us first take a look at the Schrodinger's equation.

$$H(t)\left|\psi(t)\right\rangle = i\hbar\frac{d}{dt}\left|\psi(t)\right\rangle \tag{1}$$

The equation (1) is the underlying motion equation for all quantum systems. What is peculiar about this equation is that it states that the time evolution of the system happens under the ruling of an energy state matrix. That matrix is called Hamiltonian, and manipulating the Hamiltonian is essentially the base of quantum computing.
In the case of D-Wave computers, we use the annealer to solve configuration problems. The

qubits in the chip can have "spins" - spin up or spin down. Besides these two classical states, the qubits can also be in any superposition of those classical states. Those qubits also have interactions between them - some stronger than others. The mathematical model that describes the Hamiltonian of the described problem is given by the formula below.

$$Obj(a_i, b_ij; q_i) = \sum_i a_i q_i + \sum_{i<j} b_i j q_i q_j \tag{2}$$

Here, a_i is the bias or the strength of the external field for each qubit. b_ij is the coupling strength between qubits i, j. Finding the ground state of this function is the solution to our optimization problem. We need to notice that in our case, q_i-s are binary variables. We can equivalently write the formula using a matrix and a binary vector:

$$Obj(x, Q) = x^T \cdot Q \cdot x \tag{3}$$

Q is a matrix with real values showing the relations, i.e., coupling strengths between the variables.

2 Tools and Technologies

The project uses **Python 3** and **Anaconda** distribution for an optimal setup and package management. It also uses several packages like **NumPy** for array processing and **Matplotlib** for visualization purposes. The main components, i.e., annealing related tasks, are implemented using the **D-Wave Ocean** framework. The project uses both simulations to run locally and cloud SAPI to sample the problem on a real-world quantum annealer. The annealer the paper accesses is **DW_2000Q**. The **DW_2000Q** is a D-Wave provided solver and has 2000 qubits. The simulation part uses a package called **neal**. This package is a part of the **D-Wave Ocean** framework and allows the user to simulate annealing tasks locally. Finally, the project uses **git** for version control [1].

3 Traveling Salesman Problem

3.1 Problem Statement

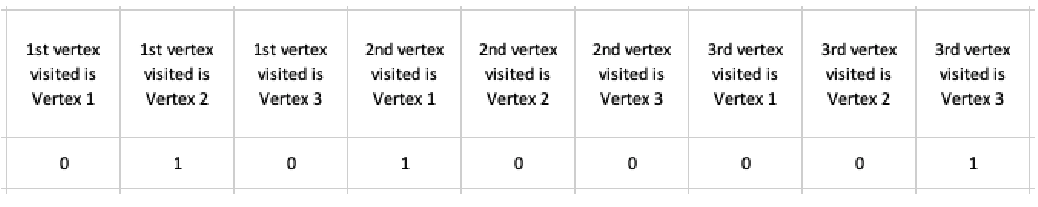
Traveling Salesman Problem or TSP is a combinatorial optimization problem. It is an NP-complete problem. [11] Below is the formulation of the problem:

Given a list of cities and the distances between each pair of cities, find the shortest possible route that visits each city and returns to the origin city

In ideal scenarios, there is no polynomial-time solution known for this problem. The general way of doing this is to consider all the possible routes. Of course, some optimization routines are still available, but the important thing is that the solution will still have non-polynomial speed on a classical computers. The best-known exact algorithms on classical computers have exponential speed. [6] In the case of quantum optimization, we jump over the exponential barrier to do it much faster. That is because of the usage of superposition. Superposition allows us to consider all possible cases at the same time. We can formulate this problem as a quantum unconstrained binary optimization problem (known as QUBO) and solve it on a quantum annealer.

3.2 Formulation as a QUBO

To transform the problem into a QUBO, we need to "flatten" it. First, number the vertices from 1...N. We take binary variables equal to the square of the number of cities - N^2. Each set of N variables will correspond to an order. Denote each binary variable as q_{ij}, where j is the number of the vertex, and i is its order in the path. Generally, the q_{ij} will be the answer to the binary question **"Was the city j (j $<$ N) visited i-th in the route**. For example, take a look at the encoding [0, 1, 0, 1, 0, 0, 0, 0, 1]. We have 9 bits, which means we have three cities. The first triple shows us (0, 1, 0). The encoding shows that the first city visited was the city labeled as 2. The second triple shows (1, 0, 0), so the second city on the route is the city number one. Finally, the third triple shows (0, 0, 1), which means the last city visited is the city with label 3.



1st vertex visited is Vertex 1	1st vertex visited is Vertex 2	1st vertex visited is Vertex 3	2nd vertex visited is Vertex 1	2nd vertex visited is Vertex 2	2nd vertex visited is Vertex 3	3rd vertex visited is Vertex 1	3rd vertex visited is Vertex 2	3rd vertex visited is Vertex 3
0	1	0	1	0	0	0	0	1

Figure 1: The questions and the answers

[1]The project can be found online: `https://github.com/robgev/quantum-routing`

The distance between these two cities defines the couplings between the pair of qubits. In the formula as mentioned above (3), now we have x binary vector of size equal to N^2 where N is the number of cities and an $N^2 x N^2$ matrix Q. Notice that we have a constraint in the problem. We cannot visit two cities at the same time. Again returning to the example as mentioned earlier, we cannot have bit triples like (1, 1, 0) or (1, 1, 1). (1, 1, 0) would mean that the first visited city is city N1 and city N2 and (1, 1, 1) the first visited city is all three at the same time. Both of those are logically impossible. We need to ensure that there is exactly one bit with value 1 in each group of N variables. We split the statement into two different sub-constraints.

- For each step i, we should have a visited city (each i-th group of N variables should have a 1)
- For each step i, there should be only one variable which is 1

If both of those are satisfied, we will have only one bit of value 1 in each group of N variables. Both of those constraints have the same general mathematical expression. As a constraint, we get a sum of shape:

$$(\sum q_{ij} - 1)^2 \tag{4}$$

The final objective function now looks like:

$$Obj = \sum_{i,j} cost(i,j) + \lambda \sum_{j} (\sum_{i} q_{ij} - 1)^2 + \lambda \sum_{i} (\sum_{j} q_{ij} - 1)^2 \tag{5}$$

Where $cost(s)$ represents the cost of the current element. In the context of the problem, cost(s) for any two vertices u, v is the distance from vertex u to vertex v (or vice versa)
In the case of the first constraint, we run over all the orders (the inner sum is over i's) and in the case of the second one we run over all the vertices (the inner sum is over j's):
We can further simplify the constraints to get a more convenient form for working with matrices. To get a sense of how we can implement these constraints on Python, let us open the brackets for the square for the case of the first constraint. As $x^2 = x$ in binary, we get the following after opening the square.

$$Constraint_1(q) = \sum_{i=1}^{3} (q_{ij} - 1)^2 = -q_{1j} - q_{2j} - q_{3j} + 2q_{1j}q_{2j} + 2q_{2j}q_{3j} + 2q_{1j}q_{3j} + 1 \tag{6}$$

This form allows us to work with indices and coefficients in an item-wise manner. That is why the form mentioned in the equation (10) is more convenient from the programming perspective. Finally, the lambda hyper-parameter can be any arbitrary significant number to make the penalty for logical errors twice bigger than the cost of taking another, valid, route. It can, for example, be a number greater than the maximum of the distances between cities. In the context of this problem, we will take a number greater than the maximum possible cost of any route as lambda, i.e., n times the maximum of the distances. Both of our constraints are equally important and should be valid at the same time; hence the same lambda value is applied to both.

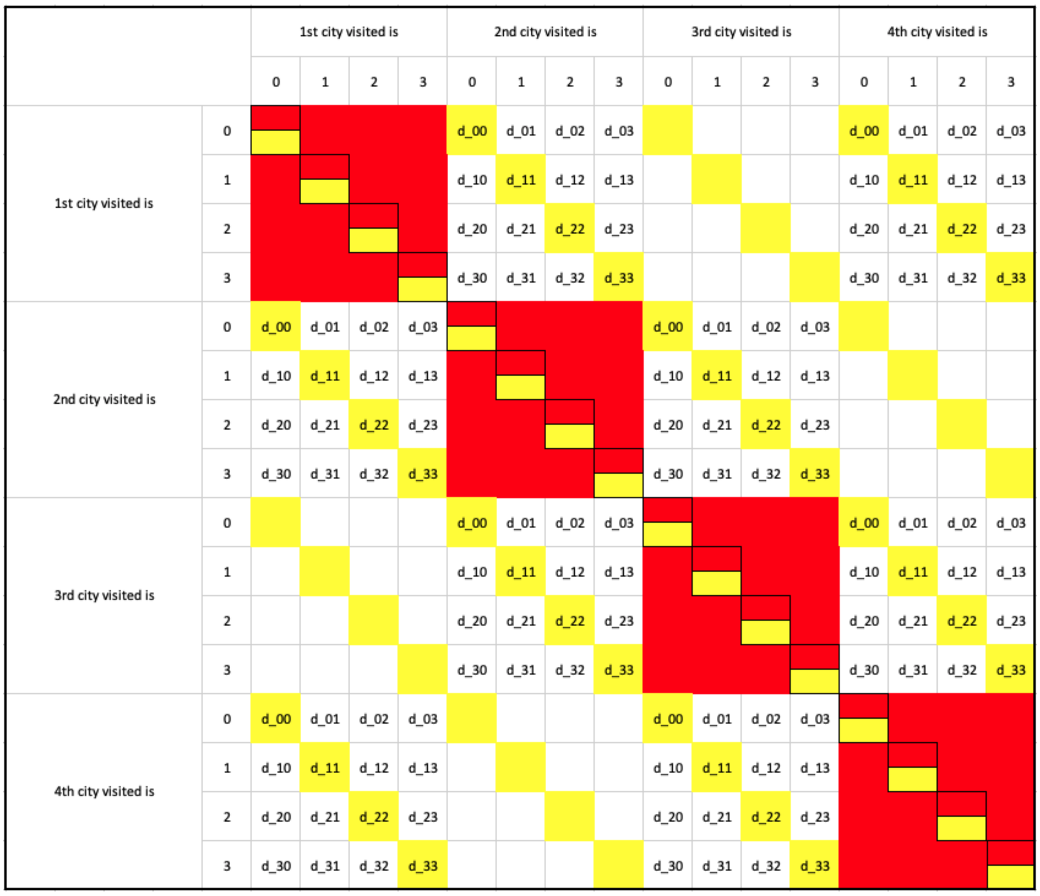
Note that we assume that the given input is a complete bidirectional graph, i.e., all the possible edges are available. In case the graph is not complete, we add another penalty constraint which applies a significant penalty if the edge does not exist.

3.3 Matrix Construction

Now we are ready to construct the matrix Q. The matrix is upper (lower) triangular, so populating only half of it is enough. The general algorithm will have several steps. We have N vertices to visit. Remember that we have N^2 variables denoted as q_{ij}, where j is the number of the vertex, and i is its order in the path.

1. For each q_{ij} and $q_{(i+1)k}$ populate the corresponding cells with d_{jk}, where d_{jk} is the distance between the vertices j and k.
2. Add the first constraint values: Subtract λ from each item on the main diagonal and add 2λ for each item that corresponds to same-city couples. These are on the main diagonal of every 4x4 grid that does not include elements from the main diagonal.
3. Add the second constraint values: Subtract λ from each item on the main diagonal and add 2λ for each item that corresponds to same-order couples. These are items that are above the main diagonal but stay in the same 4x4 grid.

The construction of Q is given by the example of $N = 4$ in the figure below.

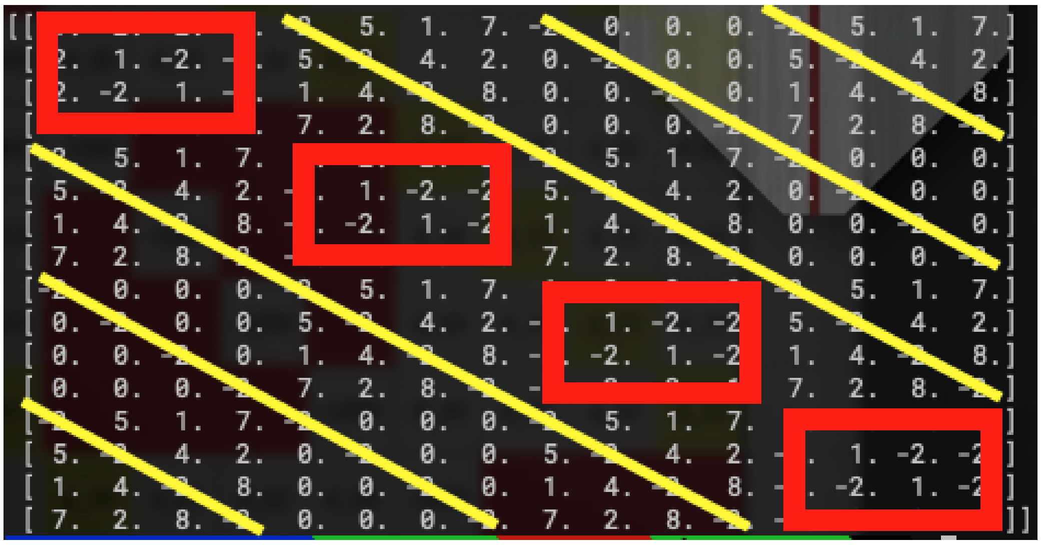


		1st city visited is				2nd city visited is				3rd city visited is				4th city visited is			
		0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
1st city visited is	0	🟥🟨	🟥	🟥	🟥	d_00	d_01	d_02	d_03	🟨				d_00	d_01	d_02	d_03
	1	🟥	🟥🟨	🟥	🟥	d_10	d_11	d_12	d_13		🟨			d_10	d_11	d_12	d_13
	2	🟥	🟥	🟥🟨	🟥	d_20	d_21	d_22	d_23			🟨		d_20	d_21	d_22	d_23
	3	🟥	🟥	🟥	🟥🟨	d_30	d_31	d_32	d_33				🟨	d_30	d_31	d_32	d_33
2nd city visited is	0	d_00	d_01	d_02	d_03	🟥🟨	🟥	🟥	🟥	d_00	d_01	d_02	d_03	🟨			
	1	d_10	d_11	d_12	d_13	🟥	🟥🟨	🟥	🟥	d_10	d_11	d_12	d_13		🟨		
	2	d_20	d_21	d_22	d_23	🟥	🟥	🟥🟨	🟥	d_20	d_21	d_22	d_23			🟨	
	3	d_30	d_31	d_32	d_33	🟥	🟥	🟥	🟥🟨	d_30	d_31	d_32	d_33				🟨
3rd city visited is	0	🟨				d_00	d_01	d_02	d_03	🟥🟨	🟥	🟥	🟥	d_00	d_01	d_02	d_03
	1		🟨			d_10	d_11	d_12	d_13	🟥	🟥🟨	🟥	🟥	d_10	d_11	d_12	d_13
	2			🟨		d_20	d_21	d_22	d_23	🟥	🟥	🟥🟨	🟥	d_20	d_21	d_22	d_23
	3				🟨	d_30	d_31	d_32	d_33	🟥	🟥	🟥	🟥🟨	d_30	d_31	d_32	d_33
4th city visited is	0	d_00	d_01	d_02	d_03	🟨				d_00	d_01	d_02	d_03	🟥🟨	🟥	🟥	🟥
	1	d_10	d_11	d_12	d_13		🟨			d_10	d_11	d_12	d_13	🟥	🟥🟨	🟥	🟥
	2	d_20	d_21	d_22	d_23			🟨		d_20	d_21	d_22	d_23	🟥	🟥	🟥🟨	🟥
	3	d_30	d_31	d_32	d_33				🟨	d_30	d_31	d_32	d_33	🟥	🟥	🟥	🟥🟨

Figure 2: Construction of Q for $N = 4$. Yellow cells correspond to the first constraint and the red ones correspond to the second constraint

3.4 Coding

Let us now implement this in python and see the results. We will have an input adjacency matrix specifying the distances between the cities. We will have three main functions that will populate the matrix, doing the steps specified above. Each one of these three functions consists of nested for loops and populates a Python dictionary.



```
[[[illegible]            [illegible] 5.  1.  7. -[illegible] 0.  0.  0. -[illegible] 5.  1.  7.]
 [[illegible]2.  1. -2. -[illegible].  5. -[illegible] 4.  2.  0. -[illegible] 0.  0.  5. -[illegible] 4.  2.]
 [[illegible]2. -2.  1. -[illegible].  1.  4. -[illegible] 8.  0.  0. -[illegible] 0.  1.  4. -[illegible] 8.]
 [[illegible].  7.  2.  8. -[illegible] 0.  0.  0. -[illegible] 7.  2.  8. -[illegible]]
 [-[illegible] 5.  1.  7. [illegible] -[illegible] 5.  1.  7. -[illegible] 0.  0.  0.]
 [ 5. -[illegible] 4.  2. -[illegible].  1. -2. -2[illegible] 5. -[illegible] 4.  2.  0. -[illegible] 0.  0.]
 [ 1.  4. -[illegible] 8. -[illegible]. -2.  1. -2[illegible] 1.  4. -[illegible] 8.  0.  0. -[illegible] 0.]
 [ 7.  2.  8. -[illegible] -[illegible] 7.  2.  8. -[illegible] 0.  0.  0. -[illegible]]
 [-[illegible] 0.  0.  0. -[illegible] 5.  1.  7. [illegible] -[illegible] 5.  1.  7.]
 [ 0. -[illegible] 0.  0.  5. -[illegible] 4.  2. -[illegible].  1. -2. -2[illegible] 5. -[illegible] 4.  2.]
 [ 0.  0. -[illegible] 0.  1.  4. -[illegible] 8. -[illegible]. -2.  1. -2[illegible] 1.  4. -[illegible] 8.]
 [ 0.  0.  0. -[illegible] 7.  2.  8. -[illegible] -[illegible] 7.  2.  8. -[illegible]]
 [-[illegible] 5.  1.  7. -[illegible] 0.  0.  0. -[illegible] 5.  1.  7. [illegible]]
 [ 5. -[illegible] 4.  2.  0. -[illegible] 0.  0.  5. -[illegible] 4.  2. -[illegible].  1. -2. -2[illegible]]
 [ 1.  4. -[illegible] 8.  0.  0. -[illegible] 0.  1.  4. -[illegible] 8. -[illegible]. -2.  1. -2[illegible]]
 [ 7.  2.  8. -[illegible] 0.  0.  0. -[illegible].  7.  2.  8. -[illegible] -[illegible]]]
```

Figure 3: The shape we saw in Figure 2 on the computer-generated matrix.

We use a dictionary instead of an array of arrays or NumPy matrices because of the input format of the D-Wave sampler. The sampler provides a function called **sample_qubo**, which takes in a dictionary, where the keys are 2-value pairs consisting of qubit numbers, and the value corresponding to the key is the strength of the coupling between these two qubits. The function call will take the constructed QUBO problem and send it to either a simulator or a D-Wave computer through the cloud. Before sending the problem to the solver, the **sample_qubo** function will convert the QUBO matrix to a binary quadratic model. After the conversion, the program does the actual sampling. The solution that we will get back from the sampling will then be processed to a human-readable format through some classical post-processing.

4 Traffic Flow Optimization

4.1 Problem Statement

Traffic and traffic optimization is one of the most practical and essential problems of nowadays. With the rise of autonomous vehicles and increasing numbers of cars, traffic flow optimization could be a necessity in the future, especially in big cities. Fewer traffic jams will mean less air pollution, more open roads for emergency vehicles. The general problem we will try to solve is:

Given the routes r, consisting of several sections of n cars, find a routing that minimizes the number of traffic jams, i.e., each section s appears the least possible amount of times in all combined routes.

Again, that is a graph problem, where we need to find a set of a subset of edges, in which case each vertex appears a minimum amount of times. We can say that the weight of an edge is equal to the number of times it is included in the cars' routes. The more cars on the road - the more time it will take to go through the road.

4.2 Formulation as a QUBO

The formulation of the given problem as a QUBO is similar to the one given above. In this case, however, our variables correspond to routes. Each variable will answer the question, "Does the car i take the route j?". As we can see, each car will have r variables associated, where r is the number of routes per car. In total, we have r * n variables. How many possible routes are there? For a small network of roads, we can consider all the possible routes available. However, the scale of routes grows very fast, and even in case of small S (number of sections), we will need hundreds of qubits to represent the problem. The r * n scales with both S (r is the number of routes and is, in fact, permutations over S) and n. To avoid this problem, we will do pre-processing of data and choose alternative routes for cars by hand. After running the optimization and getting the results, we can iterate over it again to optimize starting from that configuration. As [10] suggests, we can have a predefined number of alternative routes chosen for each car. This way, we will need k * n (in the paper [10], k = 3) variables, which is viable as it scales only with the number of cars. The quantity we want to minimize is the number of intersections between the routes. The intersection happens when the given section s appears in both of the routes of two different cars. This way, we make sure that no traffic jams will occur and the most optimal time of travel will be reached for all cars. So, the cost function for a single segment s will look like:

$$cost(s) = \sum_{C} (q_{ij})^2 \tag{7}$$

Here, C is the set of all the routes that contain the segment s. For coding purposes again, let us consider an example. Suppose the route N2 of the car N1, the route N3 of the car N2, and the route N2 of car N3 share a segment. Then:

$$cost(s) = \sum_{C} (q_{ij})^2 = q_{12} + q_{23} + q_{32} + 2q_{12}q_{23} + 2q_{23}q_{32} + 2q_{12}q_{32} + 1 \tag{8}$$

The only constraint we have is to make sure that each car takes only one of the k routes dedicated to that specific car. That is the same constraint as in the case of the traveling salesman problem. As we have already seen, the constraint is:

$$Constraint(q) = \sum_{j=1}^{k} (q_{ij} - 1)^2 \tag{9}$$

In case of $k = 3$ we get:

$$Constraint(q) = \sum_{j=1}^{3}(q_{ij} - 1)^2 = -q_{i1} - q_{i2} - q_{i3} + 2q_{i1}q_{i2} + 2q_{i2}q_{i3} + 2q_{i2}q_{i3} + 1 \tag{10}$$

The final objective function looks like:

$$Obj = \sum_s cost(s) + \lambda \sum_i (\sum_j q_{ij} - 1)^2 \tag{11}$$

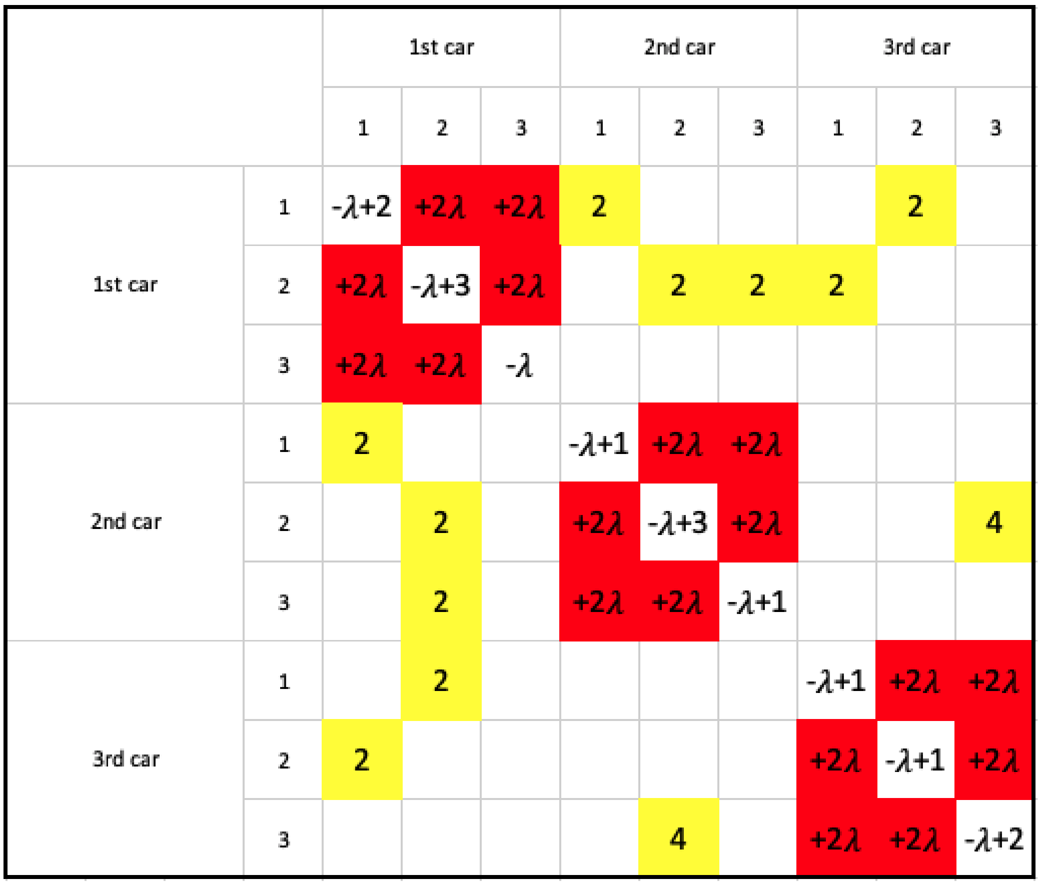
In this case, for the lambda hyperparameter, we take a number that is bigger than the maximum number of times a single car has a sharing segment s. It is better to have a car share a segment of a route than it is to technically make a car to take two routes or neglect one of the other cars.

4.3 Matrix Construction

For the construction of a matrix, we need to identify the route sharing points. Given all the shared segments, the algorithm will look like the following:

1. If two routes share the segment s, add +1 to the two diagonal elements corresponding to those routes. For example, if the second route of the car N1 shares a segment with the third route of the car N3, then we add +1 to q_{11} and q_{88}
2. If two routes share a segment s, add +2 to the two cross-term elements. For example, if the second route of the car N1 and the third route of the car N3 share a segment, then we add +2 to q_{18} and q_{81}
3. Add $-\lambda$ to each element on the main diagonal.
4. Add $2lambda$ to each element that is not on the main diagonal of that given kxk cell.

Let's understand this in the example and a picture with the constructed matrix. Suppose we have 3 cars with 3 routes each. We have some shared segments: r_{11} shares a segment with r_{21} and r_{32}, r_{12} shares a segment with r_{23}, r_{31} and r_{22}, r_{22} shares 2 segments with r_{33} and a segment with r_{12}. Here is the final matrix of the given example:



		1st car			2nd car			3rd car		
		1	2	3	1	2	3	1	2	3
1st car	1	$-\lambda+2$	$+2\lambda$	$+2\lambda$	2				2	
	2	$+2\lambda$	$-\lambda+3$	$+2\lambda$		2	2	2		
	3	$+2\lambda$	$+2\lambda$	$-\lambda$						
2nd car	1	2			$-\lambda+1$	$+2\lambda$	$+2\lambda$			
	2		2		$+2\lambda$	$-\lambda+3$	$+2\lambda$			4
	3		2		$+2\lambda$	$+2\lambda$	$-\lambda+1$			
3rd car	1		2					$-\lambda+1$	$+2\lambda$	$+2\lambda$
	2	2						$+2\lambda$	$-\lambda+1$	$+2\lambda$
	3					4		$+2\lambda$	$+2\lambda$	$-\lambda+2$

Figure 4: Construction of Q for $k = 3$ and $n = 3$.

4.4 Coding

For the most part, we can reuse the code written for the Traveling Salesman Problem. However, we still need to change the cost population function. Besides that, we also need to remove one of the constraint functions. We will tweak the code a little bit from the previous part. The functions will get more arguments to match the requirements of this problem as well. Other than that, we also need to write the classical post-processing part.

5 Limitations

For running the QUBO on the real-world quantum annealer using the cloud, we will need a D-Wave API Key, which may not be available. That is why we will mainly use simulated annealer. As such, we will not work with larger problem sizes and stay near the limit of 4-5 cities. In the case of real-world quantum annealer D-Wave 2000Q, the embedding and qubit number limitations still put an upper bound to the maximum number of cities/cars that the quantum annealer can handle. Embedding even for ten cities for TSP or three cars with three routes each for TFO (both around nine qubits) will be a hard problem for the D-Wave 2000Q annealer and may fail. To solve the problem for more than nine cities, we need to think of a hybrid solution using a qbsolv. [15]

6 Conclusions and Future Work

6.1 Future Work and Final Considerations

As future work, we consider upgrading the solution to a quantum-classical hybrid. To jump over the limitation of embedding, we can consider the **qbsolv** package. The package allows the users to divide the QUBO into smaller sub-problems (called subQUBOs) and solve them sequentially on the D-Wave annealer. [15][10] The division of the problem into subproblems is a classical stage using a **tabu search** and is heuristic. The **qbsolv** runs the optimization several times until no further optimization is found. We can control the number of runs using the **num_repeats** parameter. After running the small subproblems, **qbsolv** merges the results and outputs the final solution of the initial problem. We intend to use the **qbsolv** to scale the solution. For the traveling salesman problem, we can specify a starting point, thus decreasing the number of qubits needed form N^2 (which considers all points as starting points) to $(N-1)^2$ (as the starting node is already visited when we start the problem, so we do not need decision bits for that one) For the traffic flow optimization problem, we can add more variables that affect the traffic. Also, we can find more features subject to optimization. We intend to create a web interface. The interface will take real-world traffic data, pre-process it, and optimize the traffic using a quantum annealer. In the case of the traffic data availability, we will run the analysis over Yerevan's traffic and see what roads could be used more, what roads could be beneficial to build, etc.

6.2 Conclusions

Quantum technologies and especially quantum machine learning have vast room to grow and even more significant potential. As neural networks came to revolutionize the way we thought about machine learning, quantum computing will help us make predictions more efficiently and correctly. Researchers already developed various optimization techniques that pair well with both near-term universal quantum computers and adiabatic quantum computers, used to solve specific tasks. As we wait for the creation of fault-tolerant quantum computers, such techniques already can help us solve several significant problems. We can use those solutions to solve real-world problems now. Of course, the more qubits and engineering advancements in the field will reflect on the accuracy and speed of getting the results.

References

[1] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, et al., *Quantum supremacy using a programmable superconducting processor*, Nature, 574 (2019), pp. 505–510.

[2] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, *Quantum machine learning*, Nature, 549 (2017), p. 195–202.

[3] M. Booth, S. P. Reinhardt, and A. Roy, *Partitioning optimization problems for hybrid classical*, quantum execution. Technical Report, (2017), pp. 01–09.

[4] P. Date, *Efficiently embedding qubo problems on adiabatic quantum computers*. https://www.dwavesys.com/sites/default/files/25_Wed_PM_Date.pdf, 2018.

[5] S. Feld, C. Roch, T. Gabor, C. Seidel, F. Neukart, I. Galter, W. Mauerer, and C. Linnhoff-Popien, *A hybrid solution method for the capacitated vehicle routing problem using a quantum annealer*, Frontiers in ICT, 6 (2019), p. 13.

[6] M. Held and R. M. Karp, *A dynamic programming approach to sequencing problems*, Journal of the Society for Industrial and Applied mathematics, 10 (1962), pp. 196–210.

[7] R. Y. Li, R. Di Felice, R. Rohs, and D. A. Lidar, *Quantum annealing versus classical machine learning applied to a simplified computational biology problem*, NPJ quantum information, 4 (2018), pp. 1–10.

[8] A. Lucas, *Ising formulations of many np problems*, Frontiers in Physics, 2 (2014), p. 5.

[9] R. Martoňák, G. E. Santoro, and E. Tosatti, *Quantum annealing of the traveling-salesman problem*, Physical Review E, 70 (2004), p. 057701.

[10] F. Neukart, G. Compostella, C. Seidel, D. Von Dollen, S. Yarkoni, and B. Parney, *Traffic flow optimization using a quantum annealer*, Frontiers in ICT, 4 (2017), p. 29.

[11] C. H. Papadimitriou and P. CH, *The euclidean traveling salesman problem is np-complete.*, (1977).

[12] J. Sanders, *D-wave announces 5,000-qubit fifth generation quantum annealer*. https://www.techrepublic.com/article/d-wave-announces-5000-qubit-fifth-generation-quantum-annealer/, 2019.

[13] M. Schuld, *Quantum machine learning 1.0*. https://medium.com/xanaduai/quantum-machine-learning-1-0-76a525c8cf69, 2018.

[14] T. G. Sebastian Feld, *Project qasar: Results and hands-on demonstration of a joint project of volkswagen and lmu*. https://www.dwavesys.com/sites/default/files/lmu-merged-published.pdf, 2018.

[15] D.-W. Systems, *D-wave initiates open quantum software environment*. https://www.dwavesys.com/press-releases/d-wave-initiates-open-quantum-software-environment, 2017.

[16] R. H. Warren, *Solving the traveling salesman problem on a quantum annealer*, SN Applied Sciences, 2 (2020), p. 75.