CHARLIE WELLS

ECM3401 – Individual Literature Project

Comparing the Efficiency of QUBOs with Standard Methods for Solving Classical Routing Problems

Classical Routing Problems

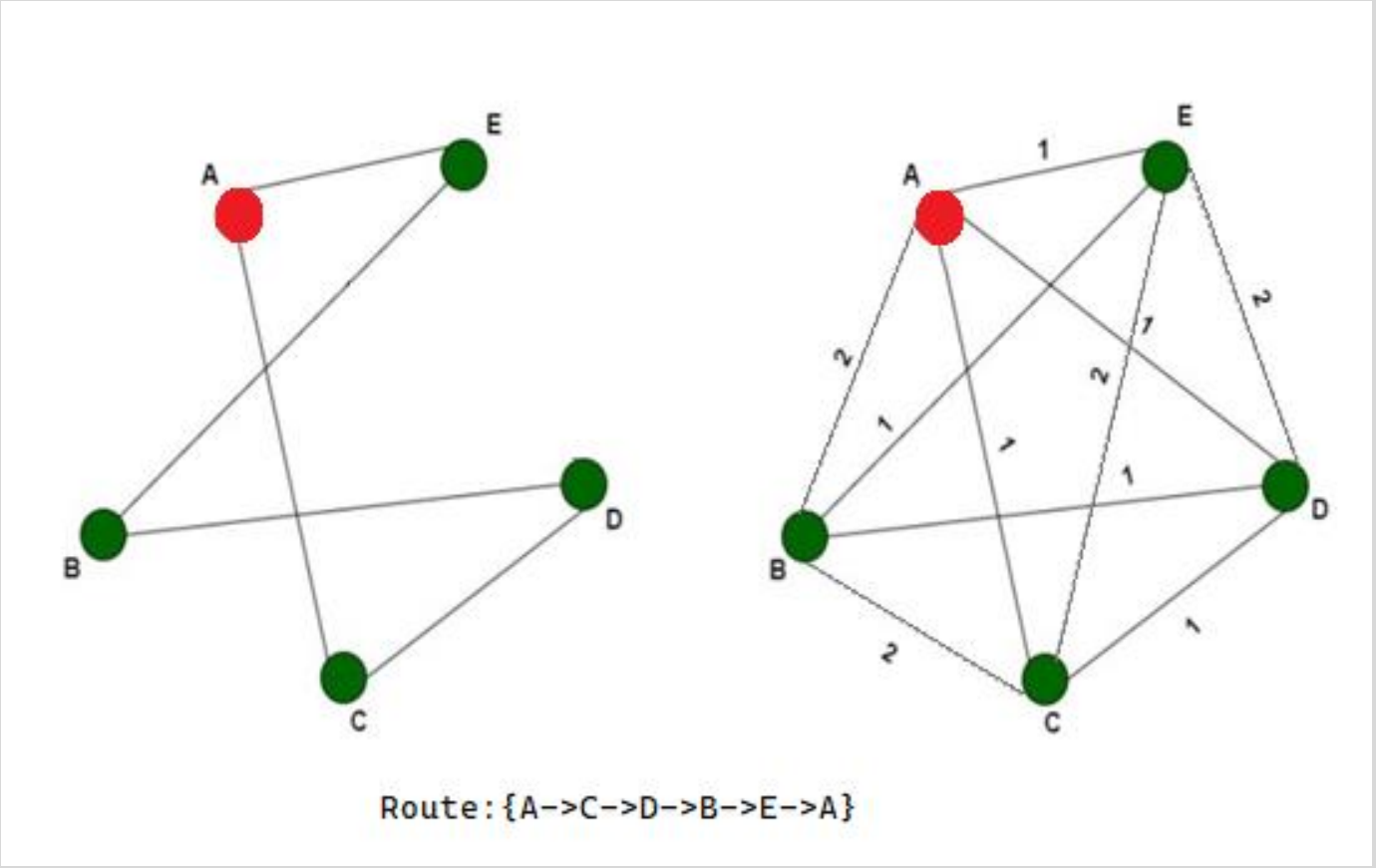
- In this project, two routing problems are considered, the travelling salesman and vehicle routing problem
- The Vehicle Routing Problem asks for the “optimal route of a fleet… between a bulk terminal and a large number of service stations”.
- The Travelling Salesman Problem is a generalisation of this Vehicle Routing Problem (where we consider only one vehicle rather than a fleet of vehicles).

Travelling Salesman Problem

Starting from a depot (In this case, city A), travel to all cities once before returning home.

The distance between each city is given, and the graph is assumed to be bidirectional

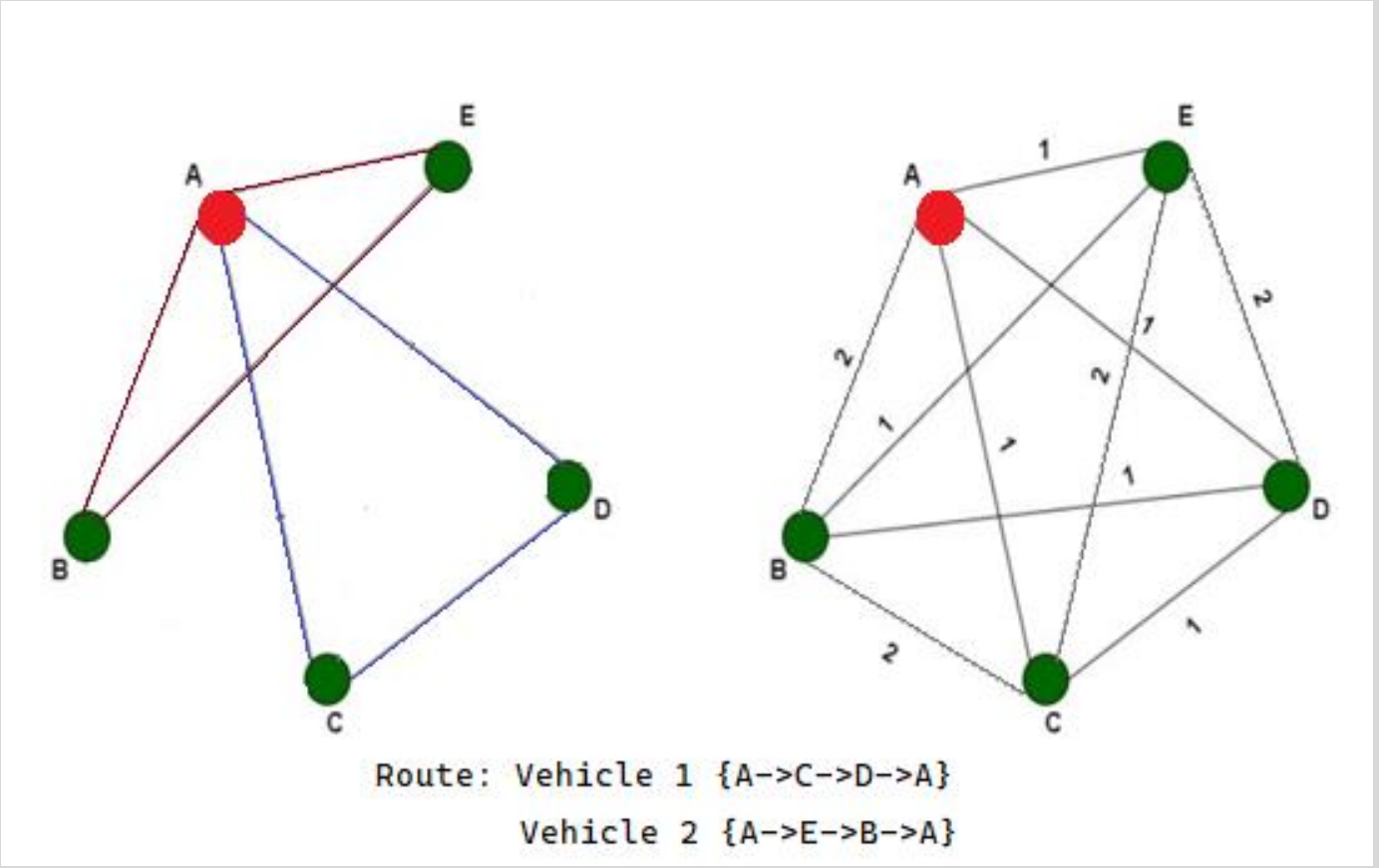
The objective is to minimise the total distance travelled.



Vehicle Routing Problem

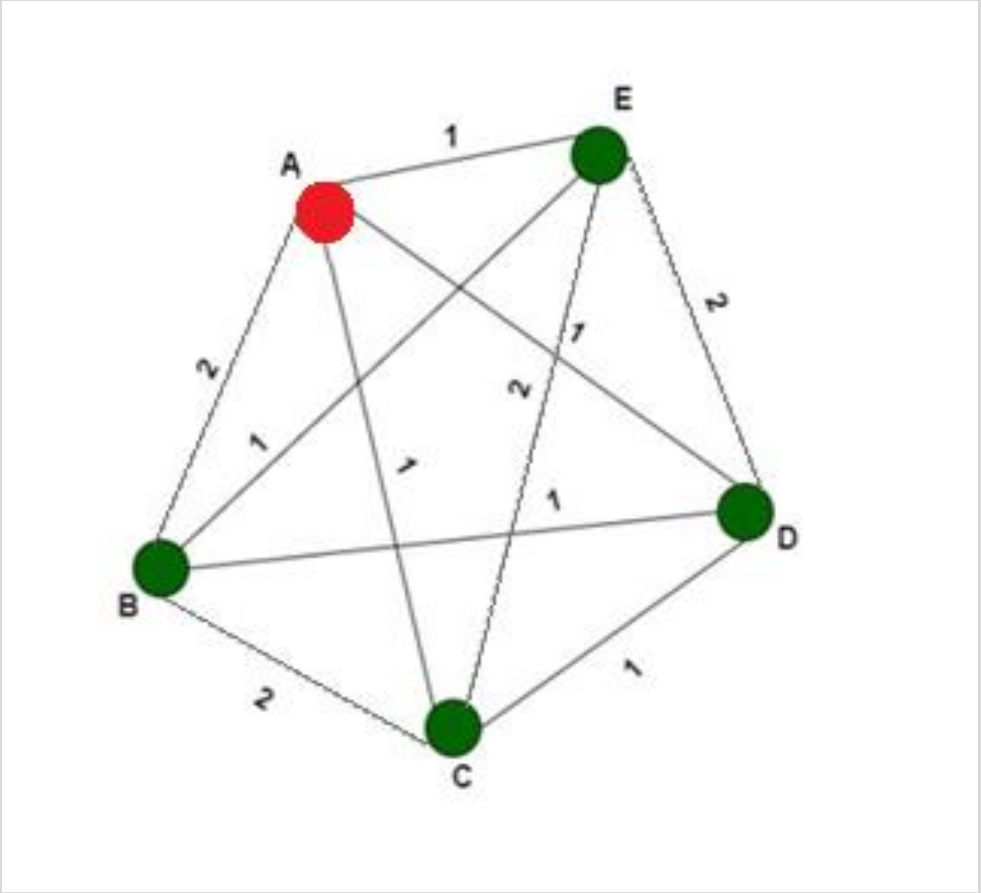
Starting from a depot (In this case, city A), all cities must be visited by exactly one vehicle. Each vehicle must then return to the depot.

The objective differs depending on the use case, but normally consists of minimise the total distance travelled, or the maximum distance travelled by a single vehicle.



Common algorithms to solve routing problems

- **Brute Force** - Iterating through every possible permutation. The minimum result is guaranteed to be optimal)
- **Nearest Neighbour** – A simple greedy algorithm, each vehicle visits the closest destination which has not been visited.
- **Ant Colony Optimisation** – Simulates ants moving through the graph leaving pheromone trails. The lower the distance, the more pheromones will be deposited. Further ants are more likely to travel paths with more pheromones.



QUBOs and their formulation

- An instance of a Binary Quadratic Model (BQM), for which each value can be either 0 or 1
- A mathematical equation which has the correct form to be processed by a quantum processing unit (found inside a quantum computer)

$$f_Q(x) = x^\top Q x = \sum_{i=1}^{n} \sum_{j=i}^{n} Q_{ij} x_i x_j$$

A matrix whose values define a weight for each index

Variables that can take the values 0 or 1

$$\sum_{i=1}^{n} \sum_{j=i}^{n} Q_{ij} x_i x_j$$

For each pair of indices, if both x_i and $x_j = 1$, the weight Q_{ij} is added, however, when either variable has value 0, their product is 0 so the weight is ignored.

In this simple case, finding the minimum of this sum (assuming all values of Q are positive), would be setting every x = 0.

In the formulation of the TSP and VRP, all values of Q (the distances between each location) are positive, however, we have to consider that setting all x = 0 will not yield a valid solution.

To resolve this, we typically introduce penalty terms which are added onto this cost function. This increases the total weight if a constraint is violated (say, we do not travel to every location).

Quantum computing

- QUBOs are in the form to be processed and solved by a quantum computer
- Shor demonstrated that prime factoring on a quantum computer will always be faster than on a classical computer if the prime number is sufficiently large.
- How efficient is generating a QUBO and solving it compared to classical methods? How much faster is the runtime and how accurate are the solutions produced?

Comparing the Efficiency of QUBOs with Standard Methods for Solving Classical Routing Problems

Aims of the Project

- Design and formulate QUBOs for both the travelling salesman problem and the vehicle routing problem.
- Implement this QUBO formulation in python, alongside the implementation of three algorithms discussed previously (Brute force, nearest neighbour, ant colony optimisation).
- Compare the runtime and accuracy of each algorithm, for varying distances and number of destinations
- Conclude whether solving a QUBO to obtain the solution is more optimal than the current classical methods for either the travelling salesman or vehicle routing problem.

Optimising the brute force algorithm

Had to ensure that the brute force algorithm was optimised to avoid memory issues for larger quantities of cities.

In the TSP case, this algorithm requires the calculation of every permutation (a problem with complexity n!).

Through these optimisations, this was reduced to finding (n-1)!/2 permutations.

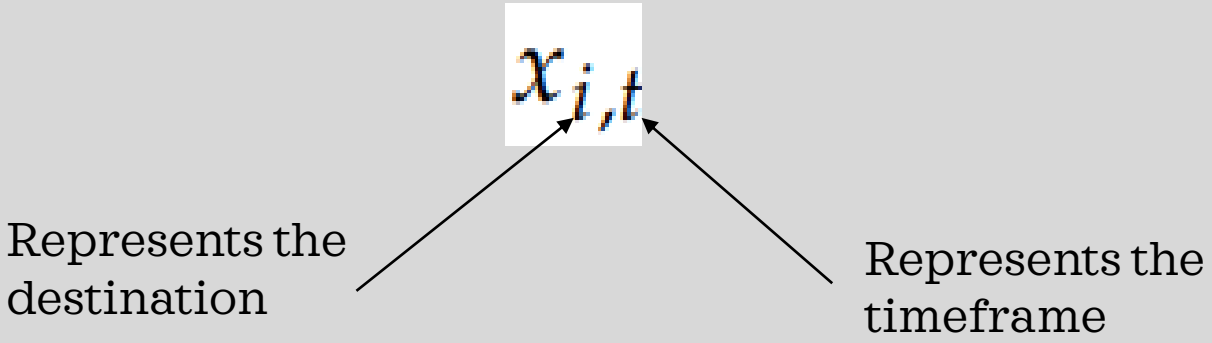
Similarly for the VRP case, the algorithm requires the calculation of every set partition (and every permutation of those set partitions).

Using set theory, the number of set partitions was greatly reduced, and hence, the runtime of the algorithm was reduced.

Designing the QUBO

- QUBO had to have a natural formulation, which could be extended from the TSP case into the VRP case.

Decided on using two variable formulation



So, the variable has value 1, if at time **t**, the vehicle visited destination **i**

Designing the QUBO (Cost Function)

$$\sum_{i=0}^{N+1} \sum_{j=0}^{N+1} \sum_{t=0}^{N} d_{i,j} x_{i,t} x_{j,t+1}$$

One can calculate that for the chosen formulation, the cost function is given by the following summation, with $d_{i,j}$ representing our distance matrix (the distance between each node).

However, we need to also include constraints and penalty terms to ensure that the minimum solution is not x = 0 everywhere.

Designing the QUBO Constraints

The first requirement for the travelling salesman problem is that each node should be visited exactly once.

This means that for every destination **i** we must have that **x** = 1 at exactly one time index **t**.

$$\sum_{t=0}^{N} x_{i,t} = 1 \qquad \text{For each } i \in \{0, \ldots, N\}$$

Designing the QUBO Constraints

Similarly, at each time **t**, a node should be visited, and this should be the only node visited at that time.

This means that for every time **t** we must have that **x** = 1 at exactly one destination index **i**.

$$\sum_{i=0}^{N} x_{i,t} = 1 \qquad \text{For each } t \in \{0, \ldots, N\}$$

Designing the QUBO Constraints

The final constraint is not strictly necessary, however it agrees with our definition of the travelling salesman so has been included anyway.

We state the a vehicle has to finish at a specific node (as the vehicle has to return to the starting node). So given a starting node **s,** and considering we have **N** destinations

$$x_{s,N} = 1 \qquad \text{For given starting node } s$$

Designing the QUBO Constraints

Each of these constraints must be combined with a penalty term, chosen as to increase the weight of the solution if a constraint has been violated.

If the penalty term is too high, we may end up overconstraining and struggle to find the global minima, however, if the penalty term is too low, the solution may not be penalised enough for violating a constraint, leading to erroneous optimal solutions.

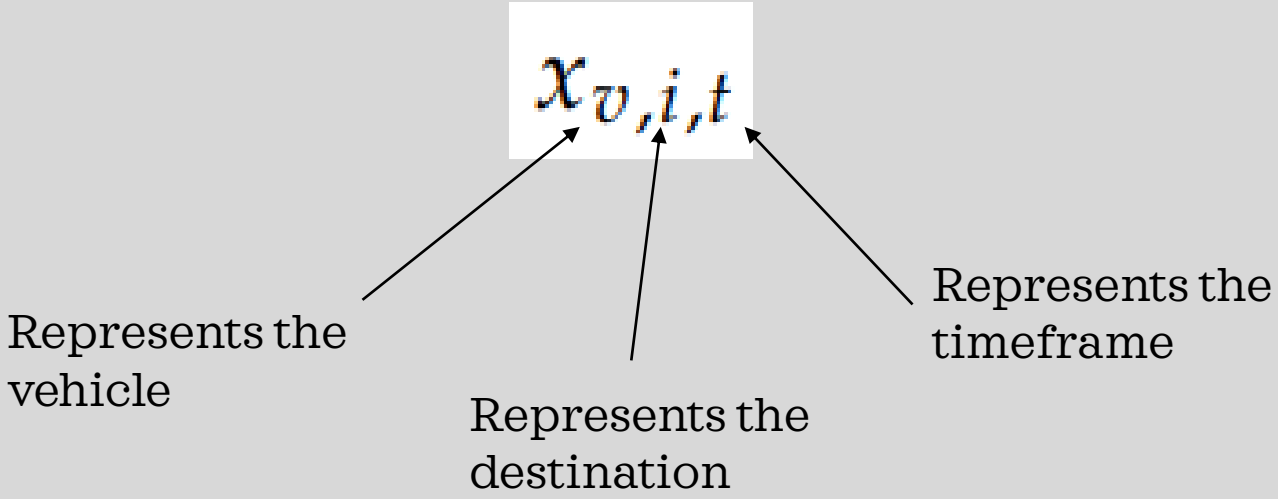
To determine the penalty weight, I made use of the MQC method. This meant setting the penalty weight equal to the maximum distance between any two cities:

$$MQC = \max\{d_{i,j}\} \quad \text{For } i, j \in \{0, \ldots, N\}$$

Designing the QUBO (VRP)

For the vehicle routing problem, the formulated QUBO is an extension of the prior QUBO.

In this case, we must also consider a variable number of vehicles, so our variable now becomes



Designing the QUBO (VRP)

$$\sum_{i=0}^{N}\sum_{j=0}^{N}\sum_{t=0}^{N} d_{i,j}x_{v,i,t}x_{v,j,t+1} \qquad \text{For } v \in \{0, \ldots, M\}$$

For M vehicles, our cost function looks as such, which is identical to our previous cost function (except now we consider a set of vehicles rather than a single vehicle).

We again need to consider the constraints on each vehicle in the vehicle routing problem.

Designing the QUBO Constraints (VRP)

The constraints for the VRP can be categorized into static and dynamic.

Static constraints remain the same regardless of the vehicle's route. For example, each node must be visited by exactly one vehicle, each vehicle must start at the starting node, etc.

Dynamic constraints are those which change depending on the number of cities each vehicle visits. For instance, each vehicle must return to the starting node and cannot leave once it has returned.

The formulation of these constraints depends on how many destinations that vehicle has visited.

Static Constraints (VRP)

In my report I further detail how each of these constraints were obtained and their meaning in context:

$$\sum_{t=0}^{N}\sum_{v=0}^{M} x_{v,i,t} = 1 \qquad \text{For } i \in \{1, \ldots, N\}$$

Each node is visited only once

$$\sum_{t=0}^{N+1} x_{v,S,t} = 2 \qquad \text{For } v \in \{0, \ldots, M\}$$

The starting node is visited twice

$$x_{v,S,0} = 1 \qquad \text{For } v \in \{0, \ldots, M\}$$

$$x_{v,S,1} = 0 \qquad \text{For } v \in \{0, \ldots, M\}$$

The vehicle starts at the starting node, then immediately travels to a new node

Dynamic Constraints (VRP)

For the dynamic constraints, consider **L** to represent the number of destinations each vehicle has visited (this is unique to each vehicle)

$$x_{v,S,L} = 1 \qquad \text{For } v \in \{0, \ldots, M\}$$

$$\sum_{i=0}^{N} x_{v,i,t} = 1 \qquad \text{For } v \in \{0, \ldots, M\}, t \in \{0, \ldots, L+2\}$$

Each vehicle returns to the starting node after visiting **L** locations

A vehicle must visit a new node during each time period, however, once it returns to the starting node, it no longer visits any further nodes.

Python Implementation

For the python implementation, I made use of the pyqubo library, allowing the creation of QUBO models through Pythonic syntax

Originally chosen because it had direct integration with D-Wave (allowing me to solve the QUBO using a suite of cloud computers).
Documentation was detailed in comparison to other QUBO libraries in Python

Solved locally using simulated annealing, which is from the d-wave neal library.

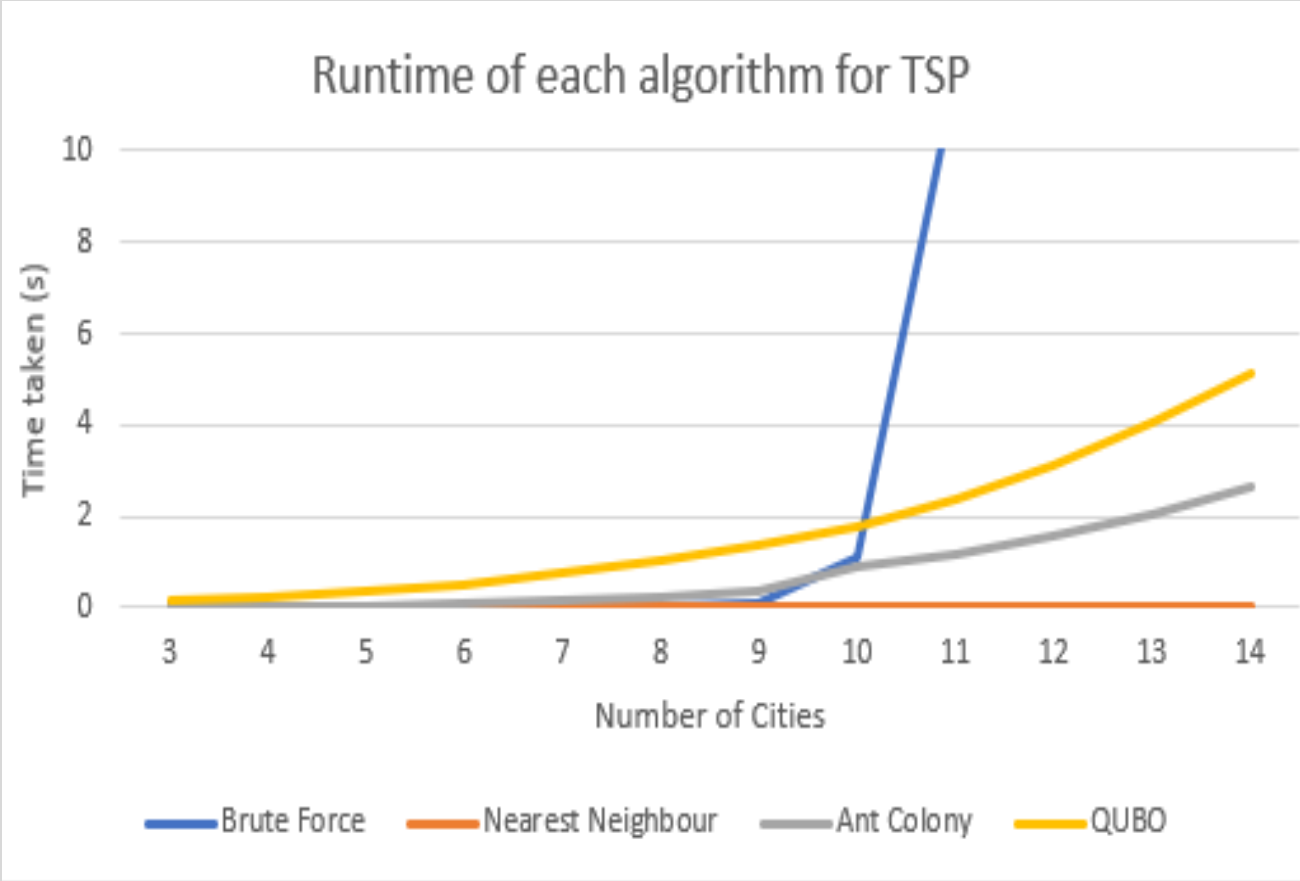
Requires various parameters which were tuned using experimentation, documentation and by investigating papers which have attempted similar QUBO formulations

Code Demonstration

Results (TSP)

For each algorithm, the runtime was compared for an increasing number of cities.

- Nearest Neighbour algorithm, as expected, has the fastest runtime.
- QUBO is faster than the brute force method when considering more than 10 cities
- The increase in runtime for ACO is steeper than QUBO, thus, for a large number of cities (40+), solving a QUBO is faster than the ACO algorithm



Results (TSP)

For each algorithm, the "correctness" of the answers provided were also compared

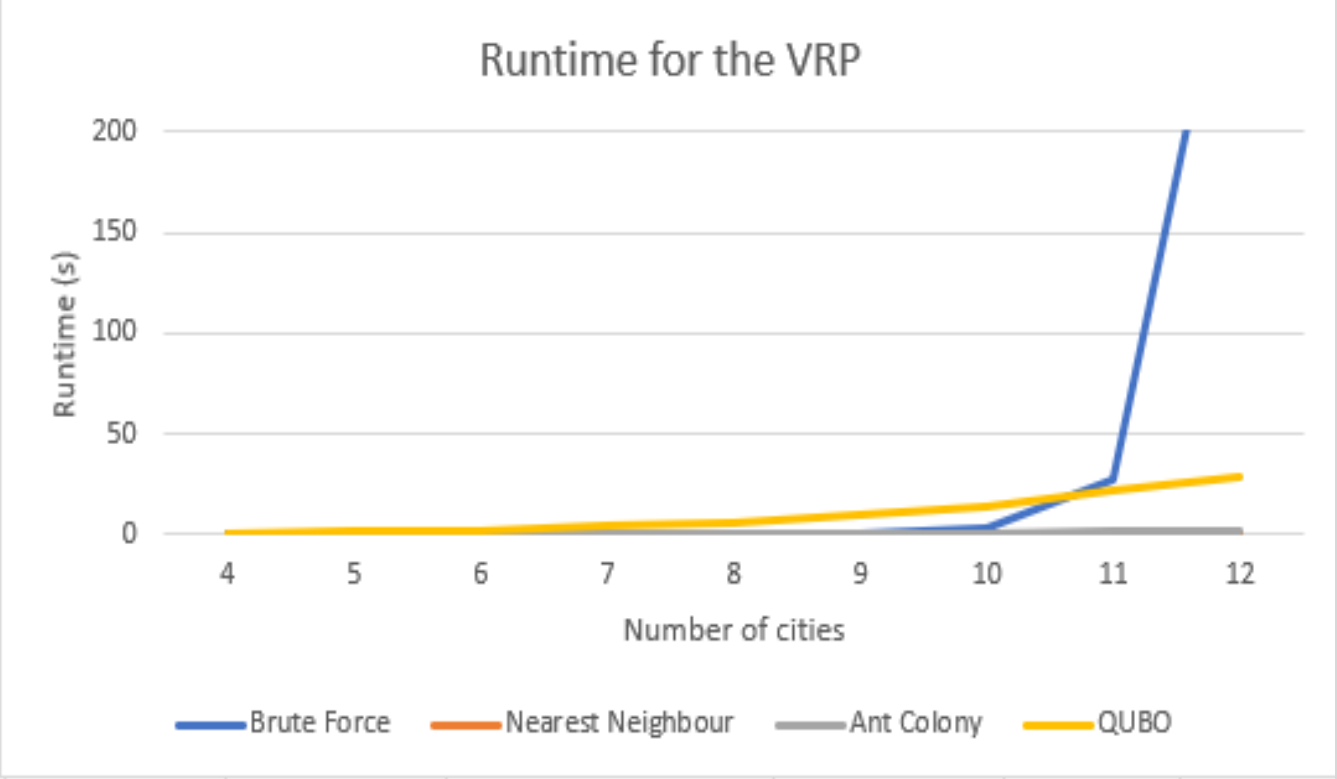
- In almost every case, the lower bound of the QUBO is better than the nearest neighbour algorithm
- There is no discernible pattern for when the ACO average is better than the QUBO average, however, the lower bound for the ACO is typically more optimal

Cities	Output path length for TSP							
	Brute Force	Nearest Neighbour	Ant Colony Optimisation			QUBO		
			Low	Average	High	Low	Average	High
3	163	163	163	163	163	163	163	163
4	178	178	178	180	186	178	178	178
5	244	259	244	254	259	244	244	244
6	233	233	233	244	271	233	233	233
7	321	322	322	335	370	321	323	326
8	249	293	249	284	340	249	263	279
9	323	323	323	342	374	339	359	363
10	248	374	248	301	417	280	308	343
11	253	349	273	324	392	289	294	318
12	358	445	382	427	465	423	444	458
13	374	468	374	438	498	444	473	508
14	379	437	414	454	520	426	489	528

Results (VRP)

The same trend can be seen when comparing the runtime of each algorithm for VRP.

- Brute force is slower than QUBO when considering more than 10 cities
- QUBO is slower than the other two heuristic algorithms



Results (VRP)

Similar to the TSP case, we see that the lower-bound of the QUBO is typically better than the nearest neighbour algorithm.

In this case, we also see that in the majority of tests, the QUBO solution outperforms the solution provided by the ACO, with the lower-bound and mean being better on average.

Cities	Output path length							
	Brute Force	Nearest Neighbour	Ant Colony Optimisation			QUBO		
			Low	Average	High	Low	Average	High
4	138	138	138	138	138	138	138	138
5	94	111	94	109	126	94	94	94
6	137	144	137	152	196	137	137	137
7	128	164	128	133	164	128	128	128
8	159	170	174	204	241	159	160	168
9	148	150	162	189	228	149	155	166
10	167	174	188	207	250	171	184	188
11	141	172	164	184	223	145	169	183
12	168	236	183	217	269	192	199	213

Summary

The QUBO formulation produces a more optimal path than the nearest neighbour solution for almost every test performed. In the VRP case, the QUBO also outperforms the ant colony optimisation algorithm, both on average and in terms of the lowest result produced.

While the nearest neighbour algorithm has the lowest runtime, the QUBO has the second lowest when considering large enough problems (when compared to the ACO and brute force algorithms) in the TSP case, and is faster than the brute force solution when considering more than 10 cities in the VRP case.

Future Directions

Test and compare additional algorithms for solving the QUBO (simulated annealing, tabu search, genetic algorithms).

Attempt to solve the problem using quantum computers by making use of D-Waves quantum suite.

Consider different formulations of the QUBO and investigate whether changing the formulation will increase its efficiency.

Further tune parameters for ACO, the simulated annealing algorithm and the penalty term of the QUBO.

Conclusions

- Successfully formulated QUBOs for both the travelling salesman problem and the vehicle routing problem
- Implemented the QUBO formulation in python, alongside the implementation of the brute force, nearest neighbour and ant colony optimisation algorithms
- Compared the runtime and accuracy of each algorithm for varying numbers of destinations and distances.
- Concluded that solving a QUBO to obtain the solution is, in some cases, more optimal than the classical methods for both travelling salesman and vehicle routing.

Thanks for watching – Charlie W.

Link to code on OneDrive: QUBO Project Code.zip

Link to video on OneDrive: QUBO Presentation Video.mp4