Final Project: Quantum Computing Exploration for the TSP Problem

Mohanna Shahrad Felicia Sun Narry Zendehrooh Kermani Riad El Mahmoudy

I. Abstract

For our ECSE 420 group project, we explored different approaches to solving the traveling salesman problem (TSP). TSP has been studied by many researchers since the early 19th century and is widely used in various fields, such as transportation and logistics, job sequencing, manufacture of microchips[1], UAV route planning[2], and DNA sequencing[3]. This project compares the performance (execution time and quality of results) of exact algorithms, metaheuristic algorithms using DWave NetworkX and OR-Tools, and quantum approaches using Azure's Quantum Inspired Optimization (QIO). In this report we discuss about a trade-off between quality results and execution time between our different approaches.

II. Introduction

A. Project Description

In this project, our team aimed to explore, implement, analyze, and contrast various ways to solve the Traveling Salesman Problem (TSP) with a focus on Quantum Computing techniques. Our main goal is to find the fastest and most efficient way of finding the optimal solution for TSP for different-sized data sets, given our current resources, by utilizing parallelization strategies and quantum computing tools and using different types of algorithms. We will conduct our experiments by first delving into exact algorithms, then using both *Google OR-Tools* and *NetworkX* tools to solve metaheuristic algorithms, and finally, we will move on to *Quantum Inspired Optimization using Azure QIO*. For the purpose of our experiments we used four different datasets that mostly come from TSPLIB, a collection of traveling salesman problem datasets maintained by Gerhard Reinelt[4].

B. Problem Definition

The Traveling Salesman Problem is a simple problem that yields various implementations and solutions. It is defined as follows: Given a directed graph with n vertices and m edges with a cost associated with each edge, we want to find the cheapest way to visit each vertex only once using the edges provided to us. The TSP solver should output the sequence of vertices in the optimal path and the associated cost. As straightforward as this problem may seem, it is known to be NP-hard. It is important to note the trade-off between correctness and scalability, i.e., receiving an optimal or approximately optimal solution in a reasonable amount of time given large input graphs. The importance of solving this problem lies in its applications, as the inputs can model multiple aspects of a practical problem. For instance, TSP is heavily used in transportation, planning, logistics, scheduling, DNA sequencing, and microchip manufacturing.

III. Exact Algorithms

To solve the Traveling Salesman Problem, we first test classical algorithms and observe their execution times. An exact algorithm is a procedure with a finite number of steps that always produces a correct solution for the given problem. This means that we are certain the algorithm will terminate and output the solution we're looking for. Regarding the Traveling Salesman Problem, we implemented and tested the brute force algorithm and the dynamic programming approach, also known as the Held-Karp algorithm.

A. Brute Force Algorithm

1) Implementation: The brute force algorithm for TSP consists of simply trying all possible permutations of paths that the graph allows us and finding the cheapest way using the given costs associated with the edges. Our algorithm uses a library function to get a list of permutations of all possible routes. In a nested for loop, we evaluate each route's cost and keep track of the cheapest. Finally, we output the cheapest route and its associated cost. This algorithm is clearly inefficient as it runs in $O(n!)$. This shows that this algorithm could not be used for larger data sets and therefore does not scale up.

2) Results: Table I summarized the average execution times found by Brute Force for our datasets after 20 runs. Note that only the small dataset of 5 cities was able to be solved due to performance bottleneck.

	5 Cities	15 Cities	26 Cities	42 Cities
Brute Force	0.00890s	Can't Solve	Can't Solve	Can't Solve

TABLE I: Summary of Brute Force Performance

B. Dynamic Programming Algorithm

1) Implementation: The dynamic programming algorithm involves getting a better runtime while still using an exact algorithm to ensure an optimal solution is always computed. The algorithm relies on recursion with a base case that outputs the cost between a dataset of 2 vertices. The recursive case finds the minimal cost by checking the edges as it shrinks its dataset until the base case is reached. This algorithm has

a runtime of $O(n^2 2^n)$, which is clearly better than the brute force algorithm but remains exponential. For this reason, we cannot scale up the datasets beyond a medium dataset.

2) Results: Table II summarized the average execution times found by Dynamic Programming (DP) for our datasets after 20 runs. Note that only the small dataset of 5 cities and the medium dataset of 15 cities were able to be solved due to performance.

	5 Cities	15 Cities	26 Cities	42 Cities
DP	2.066s	1.838s	Can't Solve	Can't Solve

TABLE II: Summary of Dynamic Programming Performance

IV. Meta-heuristic Algorithms

Since heuristic methods are insufficient to solve large datasets, we have implemented meta-heuristic algorithms that do not guarantee that a globally optimal solution can be found but may provide a sufficiently good solution to an optimization problem. Metaheuristics can search for feasible solutions with less computational effort than heuristic algorithms. For this section, we have used two different solvers from two different libraries, D-Wave NetworkX and OR-Tools, to compare their run-time and the solution they found for different datasets.

A. Solvers

1) D-Wave NetworkX: D-Wave NetworkX is an extension of NetworkX, which is a python package used to create, manipulate, and analyze networks and network algorithms. D-Wave NetworkX provides tools to implement graph-theory algorithms on the D-Wave system and various samplers.[5]

2) OR-Tools: OR-Tools is an open-source software suite for optimization by Google AI, tuned for tackling the world's toughest problems in vehicle routing, flows, integer and linear programming, and constraint programming. After we model our TSP problem, we can use different solvers to solve our desired problem. Now we will go through how we modeled the TSP problem. First, we needed to convert our Pandas data frame to a list. This list consists of a distance matrix, an array whose i, j entry is the distance from location i to location j, the *number of vehicles* which is set to 1 since we are solving TSP, and lastly, the *depot*, which is the start and end location for the route(in our case is set to 0 which corresponds to the first city). Then, we create the routing *index manager* using the list elements. The routing index manager takes the number of rows of the distance matrix(the number of cities), the number of vehicles, and the node corresponding to the depot as inputs. Using the manager, we can create the *routing model*. In the next step, to use the routing solver, we create a *distance callback* function, which takes any pair of cities and returns the distance between them. Then we set the *arc cost evaluator* to tell the solver how to calculate the cost of travel between any two cities, which, in our case, it is just the distance between them. Finally, we can set the default search parameter, choose a strategy for finding the optimal solution, call the solver, and print the solution.

B. Strategies

1) Greedy Descent: Greedy Descent is a metaheuristic search method available as a sampler for both D-Wave NetworkX and OR-Tools. It is an implementation of steepest descent, which is an optimization algorithm that attempts to minimize a differentiable function by taking steps in the opposite direction of the gradient, which is the direction of the steepest descent. However, both the D-Wave NetworkX and OR-Tools Greedy Descent algorithms use local minimization instead of computing the gradient. At each step, the algorithm chooses to visit the nearest unvisited city (local minimum). Table III summarized the average execution times and distance costs found by D-Wave NetworkX and OR-Tools for our datasets:

	5 Cities	15 Cities	26 Cities	42 Cities
NetworkX	0.00665s	0.04066s	0.38308	0.91041s
OR-Tools	0.00186s	0.00565s	0.01301s	0.06275s
NetworkX Distance	19	365	1473	1258
OR-Tools Distance	19	291	953	738
Minimum Distance	19	291	937	699

TABLE III: Summary of Greedy Descent Performance

2) Simulated Annealing: Simulated Annealing (SA) is another metaheuristic search method available to both D-Wave NetworkX and OR-Tools. It is a probabilistic technique that approximates the global optimum of a given function. The further the solution space is explored, the better probability of finding a better solution. At each step, the algorithm picks a random neighbor, such as a path close to the current one in the TSP. Then, it measures its quality, and moves according to the increase or decrease in the result quality. Table IV shows the average execution times and distance costs found by D-Wave NetworkX and OR-Tools for all our different datasets:

	5 Cities	15 Cities	26 Cities	42 Cities
NetworkX	3.82153s	110.07704s	Can't Solve	Can't Solve
OR-Tools	1s	1s	1s	1s
NetworkX Distance	19	291	-	-
OR-Tools Distance	19	291	937	737
Minimum Distance	19	291	937	699

TABLE IV: Summary of Simulated Annealing Performance

The OR-Tools Simulated Annealing also requires a time limit. Given a time limit of 1s, we found that it was able to obtain the optimal result for the first three datasets (5 cities, 15 cities, and 26 cities). Increasing the time limit to 100s for the extra-large dataset (42 cities) did not affect the results, and OR-Tools SA was only able to find a suboptimal result.

3) Guided Local Search: Guided Local Search is another metaheuristic search method. Only the OR-Tools solver has this strategy. This method sits on top of a local search algorithm to change its behavior. The local search algorithm takes a potential solution to a problem and checks its immediate neighbors to find an improved solution. The local search algorithm has a tendency to become stuck in suboptimal regions

or on plateaus where many solutions are equally fit. The GLS algorithm builds up penalties during a search to help the classical local search algorithms escape from the local minima and plateaus. Whenever the local search algorithm traps into a local optimum, then GLS modifies the objective function. then the local search will operate using the augmented objective function to bring the search out of the local optimum. This algorithm needs to have a time limit to be passed to it so that it will stop looking for a better solution after the time limit is reached. We have tested this algorithm on our datasets, and we saw that this algorithm could reach the global optimum for datasets with 5, 15, and 26 cities in 1s, which is the smallest time limit we can pass to the algorithm. However, GLS needed 6s to reach the global optimum in some cases of our largest dataset with 42 cities, but most of the time it will not reach the global optimum.

4) Tabu Search: Tabu search is another metaheuristic approach implemented by OR-Tools. Tabu Search also sits on top of a local search algorithm. Tabu Search improves the performance of local search by relaxing its basic rules. First, it allows the worsening moves to be accepted at each step if no improving move is available. This method helps when the search is stuck at a strict local minimum. In addition, prohibitions are defined to discourage the search from going back to previously-visited solutions. Tabu Search uses memory structures to track the previously visited solutions or to keep the user-provided set of rules. Then if a potential solution has been previously visited within a certain short-term period or violated a rule, it will be flagged as Tabu so that the algorithm will not consider that solution again. This algorithm also needs a time limit same as GLS. Again we tested the algorithm on our datasets, and we saw that this algorithm could reach the global optimum for all our datasets in 1s, which is the best result for our largest dataset with 42 cities.

V. Quantum Computing

Moving on to implementing our TSP solver using quantum computing methods, we chose Microsoft Azure Quantum as the development environment to run our experiments. Azure quantum service is a relatively new service that was announced publicly in February 2021. It has two main pillars:

- Quantum Computing: This service enables users to run programs on actual quantum computers that are provided by Microsoft partners such as Honeywell and IonQ.
- Quantum-Inspired Optimization: Quantum-inspired algorithms are classical algorithms where users classically emulate the essential quantum phenomena that provide the speedup. Azure Quantum optimization solvers simulate certain quantum effects like tunneling on classical hardware (normal CPUs or FPGAs). By exploiting some of the advantages of quantum computing on existing, classical hardware, they can potentially provide a speedup over traditional approaches, and therefore it is a perfect way to leverage learnings from quantum computers, without having access to large-scale quantum computers.

For this part of our project, we used the Azure QIO service to implement our TSP solver. The following sections will walk you through the process of developing our QIO-based TSP solver.

A. Setup and Prerequisites

After setting up our Azure account, we created an Azure quantum workspace which essentially is a collection of assets associated with running quantum or optimization applications. After deploying our quantum workspace, we used the workspace's hosted Jupyter notebooks as our development environment with Python as the programming language.

B. Formulating the Problem

The first step to creating a TSP solver is to first formulate our problem into a format that can be handed to the QIO solvers. In this work, we formulated our problem as a Quadratic Unconstrained Binary Optimization (QUBO) problem with the main idea to create a cost landscape for our optimization problem and let the solvers find the global minimum which is the optimal route. We first defined our cost function in terms of a matrix C in which $C[i][j]$ represents the cost of the route between node i and j of our network. These cost matrices were created based on the four datasets mentioned earlier. Having matrix C, we then defined the location vectors to represent the origin and destination nodes of the salesperson's trips between a pair of nodes. As an example, the cost of traveling from node i to j is represented as follows with all x_ks set to zero except for $x_i = 1$ in the left matrix and $X_{N+j} = 1$ in the right matrix.

$$\text{Travel cost for single trip} = \begin{bmatrix} x_0 & x_1 & \dots & x_{N-1} \end{bmatrix} \begin{bmatrix} c_{0,0} & c_{0,1} & \dots & c_{0,N-1} \\ c_{1,0} & c_{1,1} & \dots & c_{1,N-1} \\ \vdots & \ddots & \ddots & \vdots \\ c_{N-1,0} & c_{N-1,1} & \dots & c_{N-1,N-1} \end{bmatrix} \begin{bmatrix} x_N \\ x_{N+1} \\ \vdots \\ x_{2N-1} \end{bmatrix}$$

Finally, we defined the travel costs for a complete route to be the sum of the single trip costs formulated above which can be mathematically written as follows:

$$\text{Travel cost of route} = \sum_{k=0}^{N-1} \left(\begin{bmatrix} x_{Nk} & x_{Nk+1} & \dots & x_{Nk+N-1} \end{bmatrix} \begin{bmatrix} c_{0,0} & c_{0,1} & \dots & c_{0,N-1} \\ c_{1,0} & c_{1,1} & \dots & c_{1,N-1} \\ \vdots & \ddots & \ddots & \vdots \\ c_{N-1,0} & c_{N-1,1} & \dots & c_{N-1,N-1} \end{bmatrix} \begin{bmatrix} x_{N(k+1)} \\ x_{N(k+1)+1} \\ \vdots \\ x_{N(k+1)+N-1} \end{bmatrix} \right)$$

After having the cost notations set and fixed in our code, we added the following TSP constraints to our formulation. It is easy to see that only minimizing the cost functions as they are doesn't respect many of the problem's constraints. Thus, we modified the cost functions by adding some penalty weights to enforce the validity of the output routes. In other words, we reshaped the rugged optimization landscape such that for invalid solutions the solver cannot find the minima.

1) The salesperson should not be in more than one node at a time.
2) The salesperson must be at some node at each given point in time.

3) The salesperson must not travel to the same node more than once except for the headquarters.
4) The salesperson must start and finish the route in a specific node. In our code, we hard-coded that the salesperson starts the route from node N0 and ends the route at the same node.

We consulted with several similar examples from the Azure documentation to define these constraints mathematically and turn them into code. (Detailed explanation of their definition is provided in our submitted code.)

Finally, note that this part of the project is the bulk of the work as how well the problem is defined and the accuracy of constraints' formulations impacts the solvers' performance significantly. Therefore, extensive weight tuning and rewriting of the constraints is needed in order to maximize the probability of the solvers reaching minima in the cost landscape. We will further discuss this note in the Results section of the report.

C. Submitting the Optimization Problem

In the previous section, we discussed the problem's formulation and how we turned it into a PUBO problem. Next, we hand the problem to the QIO solvers to analyze their performance. In this project, we used four different QIO solvers as follows.

1) Simulated Annealing: Simulated annealing is a Monte Carlo search method named from the heating-cooling methodology of metal annealing. It rephrases the optimization problem as a thermodynamic system and considers the energy of a single system by simulating a state of varying temperatures where the temperature of a state influences the decision-making probability at each step. It is best applicable for problems with convex landscapes. Azure supports the simulated annealing solver on the CPU only.

2) Parallel Tempering: Parallel tempering is a variant of simulated annealing. It rephrases the optimization problem as a thermodynamic system just like the simulated annealing with the difference that it runs multiple copies of a system (replicas), randomly initialized, at different temperatures and then exchanges configurations at different temperatures to find the optimal configuration. With this modification, it enables walkers that were previously stuck in local optima to be bumped out of them and hence encourages wider exploration of the problem space. This method generally outperforms the simulated annealing solving on hard problems with rugged landscapes. Azure supports the parallel tempering solver on the CPU only.

3) Quantum Monte Carlo: Quantum Monte Carlo is a Metropolis annealing algorithm, similar to simulated annealing. It starts at a low temperature and improves the solution by searching across barriers with some probability as an external perturbation applied to the system. As this external field is varied over every Monte Carlo step, the configuration may be able to tunnel through energy barriers and evolve towards a desired ground state without possessing the thermal energy that it needs to climb the barriers, as would be required in simulated annealing. Due to its large overhead, the Quantum Monte Carlo solvers are best applicable for small hard problems. Azure supports the Quantum Monte Carlo solver on the CPU only.

4) Tabu Search: Tabu search is a neighborhood search algorithm that employs a tabu list. A tabu list represents a set of potential solutions that the search is forbidden to visit for a number of steps. The decision-making process per step is similar to a greedy algorithm, but with a list of forbidden moves. This solver is best applicable for high-density problems with convex landscapes. Azure supports the tabu search solver on CPU only.

Based on the Azure QIO documentation, each of the solvers mentioned above has both parameter-free and parameterized modes except for the Quantum Monte Carlo that only comes with the parameterized mode. Each solver comes with different sets of parameters such as the number of sweeps, number of restarts, timeout, etc. However, using the parameterized version is only recommended if the developer already has a good familiarity with each solver's terminology and/or has an idea of the effect of each parameter value on the landscape. We used the parameterized version just to be able to set the timeout during the experiments.

Finally, note that all of the QIO solvers discussed above are heuristics. This means that they are not guaranteed to find the optimal solution or a valid one. For this reason, we also coded a function to parse and validate the outputted solutions to measure their quality. The results and discussion on the performance of the QIO solvers are included in the next sections.

D. Discussion on QIO Solvers Performance

As mentioned in the previous section, the QIO solvers used in this project are all heuristics. Therefore, the quality of the returned solutions is heavily impacted by how well the cost functions are encoded and the constraints' weights are tuned. Even though we attempted to test the solvers with a range of different parameters, for getting the optimal solver much more extensive tuning and testing is needed which we, unfortunately, did not have the computation power and time to perform for this project. However, we believe that further tuning wll improve the solvers' performance.

Table V reports the mean and median response time of the solvers for the small dataset with 5 cities taken over 20 experiments. As illustrated in the table, the solvers have a roughly similar performance with Quantum Monte Carlo outperforming slightly better. As discussed in the previous section, Quantum Monte Carlo outperforms the other methods for smaller problems and this is aligned with our observation. As shown in the table, all the QIO solvers outputted the optimal solution (the route with a length of 19).

	Simulated Annealing Solver	Parallel Tempering Solver	Quantum Monte Carlo Solver	Tabu Search Solver
Mean Response Time (s)	2.98613	2.91878	2.58652	2.77618
Median Response Time (s)	2.92596	2.88719	2.34010	2.36031
Percentage of Optimal Solutions	100%	100%	100%	100%

TABLE V: Performance of the QIO solvers on the dataset with 5 cities

As mentioned above, the QIO solvers (except for the Quantum Monte Carlo) use a timeout parameter that sets the time that the solver searches through the optimization landscape for the local minima. As we increase the number of cities in the datasets, the solvers return lower-quality solutions in a fixed amount of time. To study this trade-off (between the response time and the quality of solution), tables VI,VII, VIII are created that summarize the average solution route length for each solver for different values of timeout from 2 to 128 seconds taken over 5 consecutive experiments. We will further discuss the observations from these experiments in the *Results and Discussion* section.

We excluded the Quantum Monte Carlo results in this part for two main reasons. First, this type of solver does not take a timeout as a parameter and instead has other values to be tuned (such as sweeps, trotter_number,transverse_field_start/stop, etc) and also as mentioned earlier Quantum Monte Carlo works the best for small hard problems due to the added overhead of quantum tunneling. Therefore, we only tested this method on the dataset with 5 cities.

	Route Length		
Timeout	Simulated Annealing	Parallel Tempering	Tabu Search
2	348.2	360.8	634.0
4	339.4	353.0	634.0
8	326.2	335.4	634.0
16	310.6	316.6	634.0
32	303.8	321.8	469.5
64	302.2	303.0	520.3
128	291.8	305.0	634.0

TABLE VI: Average Solution length per timeout for the medium dataset with optimal route length of 291 taken over 5 consecutive runs

Timeout	Simulated Annealing	Parallel Tempering	Tabu Search
2	0	0	0
4	0	0	0
8	0	0	0
16	0	0	0
32	0	0	0
64	1	1	0
128	4	1	0

TABLE VII: Number of optimal paths returned by the solvers for the medium dataset for 5 consecutive runs

	Route Length		
Timeout	Simulated Annealing	Parallel Tempering	Tabu Search
2	1031.0	1053.4	1046.0
4	1028.8	1031.0	1096.4
8	1031.0	1031.0	1052.8
16	1022.0	1052.8	1096.4
32	1027.0	1031.0	1031.0
64	1031.0	1052.8	1052.8
128	1026.6	1031.0	1074.6

TABLE VIII: Average Solution length per timeout for the large dataset with optimal route length of 937 taken over 5 consecutive runs

	Route Length		
Timeout	Simulated Annealing	Parallel Tempering	Tabu Search
2	999.6	699.0	699.0
4	950.2	699.0	699.0
8	937.0	699.0	699.0
16	910.2	699.0	699.0
32	904.0	699.0	699.0
64	896.2	699.0	699.0
128	835.7	699.0	699.0

TABLE IX: Average Solution length per timeout for the xlarge dataset with optimal route length of 699 taken over 5 consecutive runs

An important observation we made in this part was that while setting the timeout of the QIO solvers to different values in $\{2, 4, 8, 16, 32, 64, 128\}$, the execution time of the solvers was greater than the value of the timeout. (For instance, the call to the solver with a timeout of 2 seconds took approximately 6 seconds) This could be because of the added latency of calling the QIO solvers from the quantum workspace in Azure or any other factor that increases the solver calls latency while running the program in our hosted Jupyter notebook. This was a worth-mentioning point as we expect that the portion of the execution time that is solely related to the QIO solvers be smaller than the values measured and reported in the plots of the next section.

VI. Results and Discussion

Figure 1 summarizes the response time in seconds per run for all the algorithms used in our project. From the two plots in figure 1, the main observations we made for the small dataset with 5 cities are as follows:

- The dynamic programming approach had the best performance with an average of 0.00029 seconds over 20 consecutive experiments followed by the OR-Tools greedy descent algorithm with an average response time of 0.00186 seconds.
- The OR-Tools simulated annealing approach's response time is a constant line at 1 seconds. The reason for this is that this method takes a timeout (of at least 1 seconds) as its parameter and in all of the 20 experiments, it outputted the solution after using all the timeout intervals.

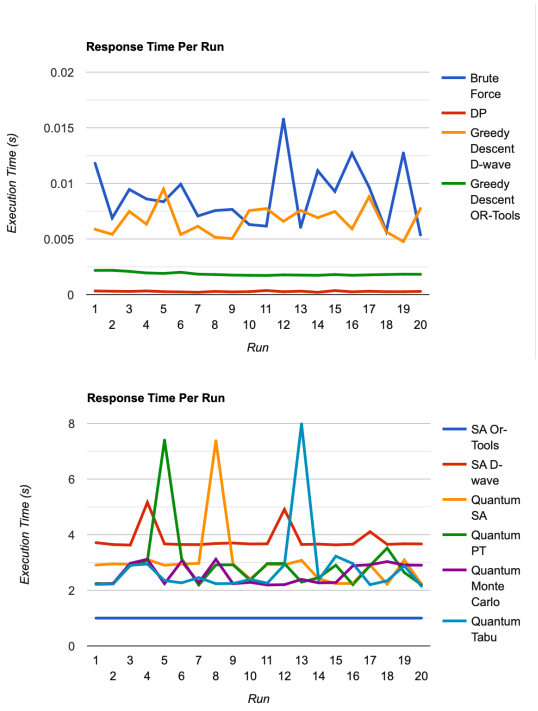

Fig. 1: Performance of all the algorithms on the small dataset

- Overall, the QIO approach did not outperform the other methods except for the D-Wave simulated annealing. Among the QIO solvers themselves, however, Quantum Monte Carlo and Tabu search solvers had a better performance compared to simulated annealing and parallel tempering solvers.
- We sometimes observed sharp jumps in the response time of the QIO solvers. (Look at experiments 5, 8, and 13 for parallel tempering, simulated annealing, and Tabu search respectively) We think one potential reason that could lead to these high response times (nearly 7 seconds) could be the high load on the Azure kernel we ran our hosted Jupyter notebook on and consequently an added latency to our experiments' results.
- All of the methods returned optimal solutions in all 20 experiments (100% success rate) except for the D-Wave greedy descent that never found the optimal solution and D-Wave simulated annealing that found the optimal solution in 19 experiments out of 20 (95% success rate).

The complexity of the problem grows exponentially with the number of cities. Therefore, moving on to using our bigger datasets, exact algorithms such as brute force and dynamic programming are not suitable anymore. We observed that they led to memory overflows and therefore are not practical solutions with the larger datasets. Figure 2 shows the performance of the OR-Tool greedy descent method with the growing size of the dataset (i.e. growing complexity of the problem) As we expected, increasing the number of nodes resulted in an increased response time. In addition to the increased time needed to find the optional solution, we will

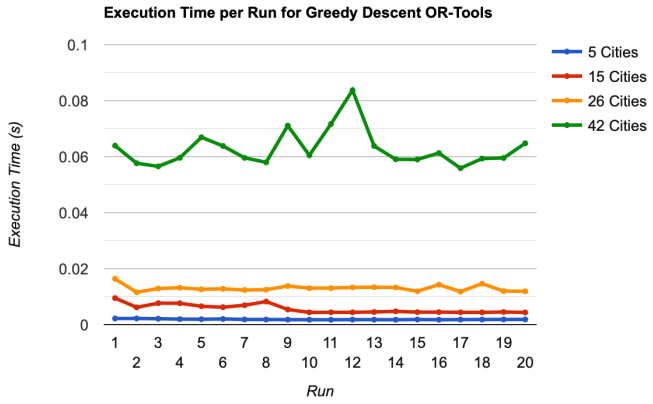

Fig. 2: Performance of the OR-Tools greedy descent with growing complexity of TSP

also discuss the decreased solution quality with the added complexity of the problem below. As the QIO solvers use a timeout parameter, the response time per run plots were not the best option to demonstrate the solver's performance. For this reason, we created the plots in figures 3, 4, and 5 that summarize the quality of the solutions found by the QIO solvers. (The exact data points in these plots can be found in *Discussion on QIO Solvers Performance* subsection)

As shown in figure 3, the optimal route length for the medium dataset with 15 cities is 291 (the green line in the plot). Simulated annealing and parallel tempering solvers get closer and closer to the optimal solution and with a timeout of $64 - 128$ seconds, both of these methods can find the optimal path. However, the average solution length outputted by the Tabu search solver did not reach the optimal solution even with a timeout of 128 seconds.

Moving on to the large dataset with 26 cities, the optimal route length of 937 is again shown by the green color in figure 4. For this dataset, none of the solvers get close to the optimal solution with any of the timeout values $\{2, 4, 8, 16, 32, 64, 128\}$. An interesting observation was that all of the solvers for all of the timeout values we tested mostly outputted a path with length 1031. A potential reason for this could be that the path with a length of 1031 is a local minimum in the optimization landscape and the QIO solvers get trapped at this point most of the time. This suggests the impact of knowing the type of optimization landscape before formulating the optimization problem and handing it to the solvers. Therefore, we believe that the structure of the TSP that we defined and tuned is probably not consistent with the structure of the optimization(cost) landscape of our large dataset and thus the QIO solvers require more time to be able to escape from this local minimum.

Finally, figure 5, shows the average solution length for the xlarge dataset with 42 cities and an optimal route length of 699. The results of this experiment were interesting as both the parallel tempering and tabu search solvers found the

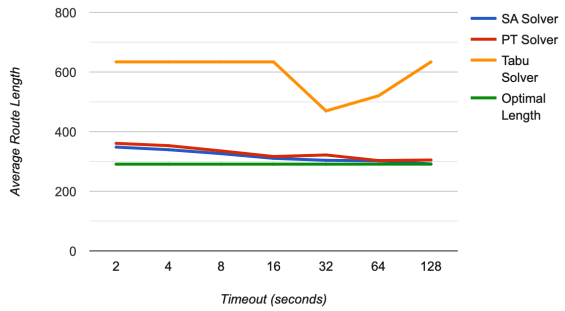

Fig. 3: Average route length per timeout for different QIO solvers for the medium dataset with the optimal length of 291

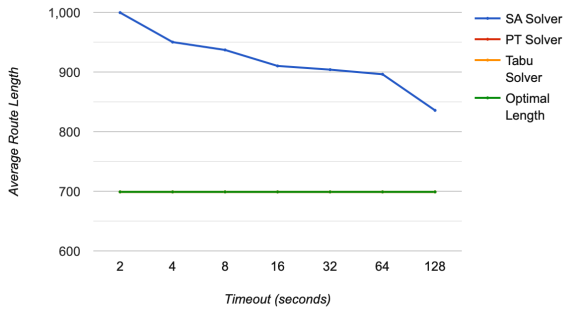

Fig. 5: Average route length per timeout for different QIO solvers for the xlarge dataset with the optimal length of 699

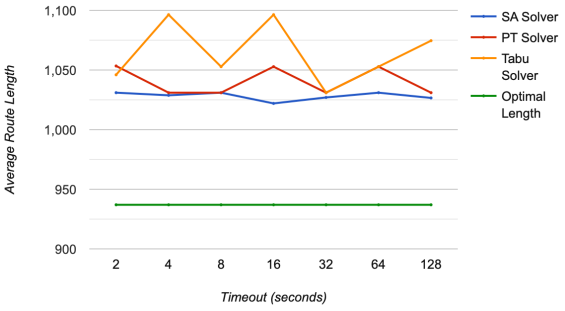

Fig. 4: Average route length per timeout for different QIO solvers for the large dataset with the optimal length of 937

optimal solution even with using the smallest timeout value (The red and orange lines are on the green line representing the optimal route length in figure 5). In contrast, the simulated annealing solver (blue line) did not get the optimal solution using different values of timeout. However, we observed that the average solution length decreased while increasing the timeout value which suggests that the solver's performance improved over time. Once again, we can conclude that the shape of the optimization landscape is a key factor in the solvers' performance. As in the case of using the xlarge dataset, even though the number of nodes is increased to 42 but the solvers showed a much better performance behavior compared to the large dataset. Therefore, knowledge of the specific optimization problem that we want to solve can significantly improve the capability and performance of the QIO solvers.

Overall, the results of our experiments did not show a significant improvement with regard to the time needed to find the optimal solution using QIO solvers over other classical techniques. However, growing the complexity of the problem and fixing the time interval to search for the optimal route, the QIO solvers can return higher-quality solutions.

There are multiple reasons behind not seeing a performance improvement with the QIO solvers. First of all, recall that we mapped the TSP problem to a Quadratic Unconstrained Binary Optimization (QUBO) problem before handing it to the QIO solvers. However, we found that TSP is not a good example of a problem that scales well in the QUBO format[6]. The study done in [6] claims that tunneling between local minima can be exponentially suppressed if the quantum dynamics are not carefully tailored to the problem.

Secondly, as mentioned earlier, a significant part of the solvers' performance depends on how well the problem is formulated. The structure of the constraints, tuning of the constants used in enforcing the constraints, and penalty weights are all key factors affecting the solver's performance. We attempted to test a range of different values and parameters in our project, however, to get the most out of the QIO potential speedup, much more extensive and in-depth tuning and testing need to be done for the specific type of the problem we want to solve. Hence, an interesting direction for future work is to study the Travelling Salesman Problem in-depth and learn how it can be rephrased as a mathematical problem and handed to the QIO solvers to maximize the quality of solutions in a fixed amount of time or minimize the response time with a fixed quality of the solution.

VII. Statement of Contribution

- Narry: Project setup and OR-Tools implementation
- Felicia: Brute Force and NetworkX implementation
- Riad: Dynamic Programming implementation
- Mohanna: Azure QIO implementation

References

[1] N. Sathya and A. Muthukumaravel, "A review of the optimization algorithms on traveling salesman problem," *Indian Journal of Science and Technology*, vol. 8, no. 29, pp. 1–4, 2015.

[2] Y. Xu and C. Che, "A brief review of the intelligent algorithm for traveling salesman problem in uav route planning," in *2019 IEEE 9th international conference on electronics information and emergency communication (ICEIEC)*. IEEE, 2019, pp. 1–7.

[3] A. M. Alanazi, G. Muhiuddin, D. A. Al-Balawi, and S. Samanta, "Different dna sequencing using dna graphs: A study," *Applied Sciences*, vol. 12, no. 11, p. 5414, 2022.

[4] G. Reinelt, "Data for the traveling salesperson problem," https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html.

[5] D.-W. S. Inc, "D-wave networkx 0.8.8 documentation," https://docs.ocean.dwavesys.com/projects/dwave-networkx/en/latest/.

[6] B. Heim, E. W. Brown, D. Wecker, and M. Troyer, "Designing adiabatic quantum optimization: A case study for the traveling salesman problem," *arXiv preprint arXiv:1702.06248*, 2017.