2022

A Quantum Solution to the Travelling Salesman Problem

CSC 2332 Final Report

Michael Luciuk | Dec 17, 2022

Table of Contents

1. Introduction 2
2. Literature Review 2
2.1. Goswami *et al.* (2004) 2
2.2. Bang *et al.* (2012) 2
2.3. Moylett *et al.* (2017) 3
2.4. Srinivasan *et al.* (2018) 3
3. Algorithm 3
3.1. Complexity Analysis 5
4. Implementation 7
4.1. PennyLane 7
4.2. Phase Estimation 8
4.3. Gover's Search 9
4.3.1. The Minimization Oracle 10
4.3.2. Improving the Success Probability of Grover 13
5. Future Work 14
6. Conclusion 14
7. References 16

1. Introduction

The optimization version of the travelling salesman problem (TSP) is a well-known graph problem wherein a salesman needs to compute the most efficient way to visit a set of n cities. Each city must be visited exactly once, and the salesman must end in the same city in which he started. This version of TSP is an NP-hard problem, and the classical brute-force approach requires time (and space) $O(n!)$ for n cities. There are other versions of TSP, and it is often phrased as a decision problem. However, we limit our discussion to the optimization version of TSP as described above.

Lots of work has been done in the classical algorithms space to improve upon the naïve, brute force approach to TSP. However, we limit our discussions to those algorithms that contain quantum components.

Rather than analyse and contemplate the diverse array of literature addressing quantum solutions to TSP, this project consisted of only a brief literature review, followed by a critical analysis of the algorithm presented by Srinivasan and colleagues in *Efficient quantum algorithm for solving travelling salesman problem: An IBM quantum experience* [1] and the implementation of a modified version of this algorithm. The algorithm presented by Srinivasan and colleagues was chosen simply out of interest as it featured two famous quantum algorithms (phase estimation and the Dürr and Høyer algorithm to find the minimum), both of which I was eager to implement.

As the bulk of this project is an implementation, the fruits of my labour lie here: https://github.com/mrl280/tsp, rather than in this report. This report focuses on explaining the quantum pieces of the chosen implementation, namely phase estimation and Gover's search.

2. Literature Review

Prior to selecting a specific quantum TSP algorithm to study and implement, I conducted a survey of the existing literature. Rather than addressing all quantum algorithm papers on TPS, this section contains but a brief summary of the works I found most interesting.

2.1. Goswami *et al.* (2004)

Goswami and colleagues [2] provide a general framework for efficiently encoding, and subsequently solving, approximate (Euclidean) TSP on a quantum computer. They prove that we can solve approximate TSP in bounded-error quantum polynomial time (BQP) resource bounds if we assume a Gaussian distribution of tour lengths. (Recall BQP is the quantum analogue of the more familiar complexity class bounded-error probabilistic polynomial time (BPP)).

2.2. Bang *et al.* (2012)

Bang and colleagues [3] provide a heuristic-based quantum algorithm to solve TSP, wherein a generalized version of Grover's search is used to amplify the probabilities of approximate solutions. It is important to remember that because, in this article, Bang and colleagues discuss heuristic-based algorithms, results are not necessarily globally optimal (heuristic algorithms usually sacrifice optimality for efficiency).

Bang and colleagues prove a quadratic speedup over the quantum algorithm's classical heuristic-based analogue, which basically just comes from the quadratic speedup that Grover provides for comparison-based search. As with [2], this work assumes Gaussian tour costs.

2.3. Moylett *et al.* (2017)

Moylett and colleagues [4] prove a quadratic speed up for TSP for bounded-degree (finite) graphs. This is done by applying quantum backtracking to the Xiao-Nagamochi algorithm discussed in [5].

2.4. Srinivasan *et al.* (2018)

Srinivasan and colleagues [1] solve TSP by encoding the costs as phases, using phase estimation to obtain the total costs, and using the Dürr and Høyer algorithm [6] to find the minimum cost cycle.

This algorithm is integrated into *The Qiskit Textbook* [7] where they note, and improve upon, some circuit errors in the original publication.

The rest of this report is dedicated to analysing, and explaining my implementation of, the algorithm presented by Srinivasan and colleagues in [1].

3. Algorithm

We start by following [1], noting that because an input cost matrix, A, is not necessarily unitary, we need to preform some preprocessing. Consider the $n = 4$ (four-city) problem, where the cost matrix, A, takes the form of a 4 by 4 matrix,

$$A = \begin{bmatrix} 0 & \emptyset_{AB} & \emptyset_{AC} & \emptyset_{AD} \\ \emptyset_{BA} & 0 & \emptyset_{BC} & \emptyset_{BD} \\ \emptyset_{CA} & \emptyset_{CB} & 0 & \emptyset_{CD} \\ \emptyset_{DA} & \emptyset_{DB} & \emptyset_{DC} & 0 \end{bmatrix},$$

where the $n^2 = 16$ matrix elements encode the costs to travel between cities. For example, $\emptyset_{AB}$ represents the cost of travelling from city A to city B. The diagonal elements are zero, because there is no cost associated with travelling directly from a city to itself. If the cost of travelling between cities in directionally independent, then we have $\emptyset_{ij} = \emptyset_{ji}$ and the matrix is symmetric about the diagonal. We make no such assumption, but we do assume all non-diagonal elements $\emptyset \in \mathbb{Z}$ and $\emptyset > 0$.

Still following [1], we elementwise exponentiate A, promoting the encoded costs to phases,

$$B = \begin{bmatrix} 1 & e^{i\emptyset_{AB}} & e^{i\emptyset_{AC}} & e^{i\emptyset_{AD}} \\ e^{i\emptyset_{BA}} & 1 & e^{i\emptyset_{BC}} & e^{i\emptyset_{BD}} \\ e^{i\emptyset_{CA}} & e^{i\emptyset_{CB}} & 1 & e^{i\emptyset_{CD}} \\ e^{i\emptyset_{DA}} & e^{i\emptyset_{DB}} & e^{i\emptyset_{DC}} & 1 \end{bmatrix}.$$

Now, we strip out the rows from B, and use them to build n diagonal matrices: U_A, U_B, U_C, and U_D. We need n matrices because we need one for each city, each summarizing the costs to travel from that city to each of the others. Srinivasan includes the diagonal elements in their decomposition. However, we note that they are unnecessary (because they are always unity) and omit them in our construction. For example, here is our U_A:

$$U_A = \begin{bmatrix} e^{i\emptyset_{AB}} & & \\ & e^{i\emptyset_{AC}} & \\ & & e^{i\emptyset_{AD}} \end{bmatrix}$$

Notice these matrices are unitary. To see that U_A is unitary, we multiply by the complex transpose $U_A^\dagger$:

$$U_A U_A^\dagger = \begin{bmatrix} e^{i\emptyset_{AB}} & & \\ & e^{i\emptyset_{AC}} & \\ & & e^{i\emptyset_{AD}} \end{bmatrix} \begin{bmatrix} e^{-i\emptyset_{AB}} & & \\ & e^{-i\emptyset_{AC}} & \\ & & e^{-i\emptyset_{AD}} \end{bmatrix}$$

$$= \begin{bmatrix} e^{i\emptyset_{AB}}e^{-i\emptyset_{AB}} & & \\ & e^{i\emptyset_{AC}}e^{-i\emptyset_{AC}} & \\ & & e^{i\emptyset_{AD}}e^{-i\emptyset_{AD}} \end{bmatrix} = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} = I$$

Now, we can tensor these unitaries together to build one big unitary, $U = U_A \otimes U_B \otimes U_C \otimes U_D$, that contains all the cost information provided in the problem.

$$U = \begin{bmatrix} e^{i(\emptyset_{AB}+\emptyset_{BA}+\emptyset_{CA}+\emptyset_{DA})} & & & \\ & e^{i(\emptyset_{AB}+\emptyset_{BA}+\emptyset_{CA}+\emptyset_{DB})} & & \\ & & \ddots & \\ & & & e^{i(\emptyset_{AD}+\emptyset_{BD}+\emptyset_{CD}+\emptyset_{DC})} \end{bmatrix}$$

Because the exponents add together, each element in U now contains the total cost of a 4 city route. However, out of these $(4-1)^4 = 81$ routes, only $(4-1)! = 6$ correspond to Hamiltonian circuits. Recall a Hamiltonian circuit is path through a graph that visits each vertex exactly once, ending back at the starting vertex, and the solution to TSP is just the minimum cost Hamiltonian circuit. Srinivasan and colleagues claim that for a given number of cities, we know the location of the diagonal that correspond to Hamiltonian circuits. However, identifying Hamiltonian circuits is itself an NP-complete problem.

Since U is diagonal, the eigenvectors are just the standard basis vectors, and Srinivasan continues by using phase estimation to compute the costs associated with the $(n-1)!$ basis vectors corresponding to valid Hamiltonian circuits.

We note that pre-identified Hamiltonian circuits allows us to skip phase estimation. If we have access to precomputed Hamiltonian circuits, we can compute the total costs directly. However, phase estimation is an important quantum algorithm, and one well worth implementing, even if its integration into our TSP solver is merely an academic exercise.

To this end, we directly compute the total costs from pre-identified Hamiltonian circuit data and embed them as phases within the diagonal elements of an $(n-1)!$ by $(n-1)!$ unitary matrix, U'. We then perform phase estimation on U' to recover the costs we just embedded. This approach has one major benefit over performing phase estimation on U directly — U' is much smaller than U. Notice that because this is an artificial problem and we know the answer, we can check to make sure we got it right. The details of phase estimation, including a detailed example for the three-city case, can be found in Section 4.2.

At this point, whether found directly or through phase estimation, we have an unsorted list of integers corresponding to total cycle costs. We need to find the minimum of this list, and this is achieved using the Dürr and Høyer algorithm to find the minimum [6] (Grover enhanced binary search). The details of this algorithm's implementation and a few simple examples can be found in Section 4.3.

Once we have all the pieces, they can be integrated into our complete algorithm:

1. Given an input cost matrix A, use precomputed Hamiltonian circuits to compute a list of total costs.

2. Encode these total costs as phases into a diagonal, unitary matrix U', and use phase estimation to recover them (optional, this step is just for practice with phase estimation).
3. Use the Dürr and Høyer algorithm to find the minimum circuit cost.
4. From the index of minimum circuit cost, return the minimum cost Hamiltonian circuit.

To be explicit, our only deviations from [1] are step 1, which they omit, and step 2, where they perform phase estimation using the much larger matrix U.

3.1. Complexity Analysis

Now that we have our algorithm, lets investigate the complexity. We limit our discussion to the quantum pieces of our algorithm, namely phase estimation and the Dürr and Høyer algorithm to find the minimum.

The quantum circuit for phase estimation is presented in Fig. 1.

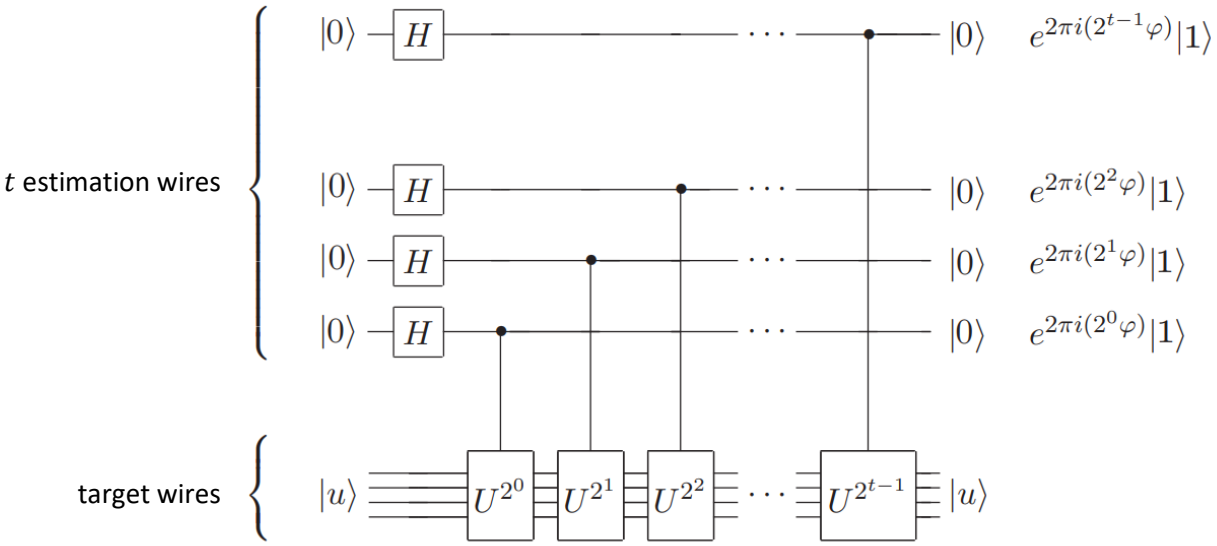

Fig. 1. Circuit for Quantum Phase Estimation. [8, pg. 222]

As seen in Fig. 1, phase estimation requires $O(t^2)$ operations, where t is the number of estimation wires used, and one call to an oracle that performs a controlled-U^j operation for $j \in \mathbb{Z}$ [8].

The Dürr and Høyer algorithm to find the minimum is just binary search on top of Grover's search. The general idea behind Dürr and Høyer is illustrated in Fig. 2.

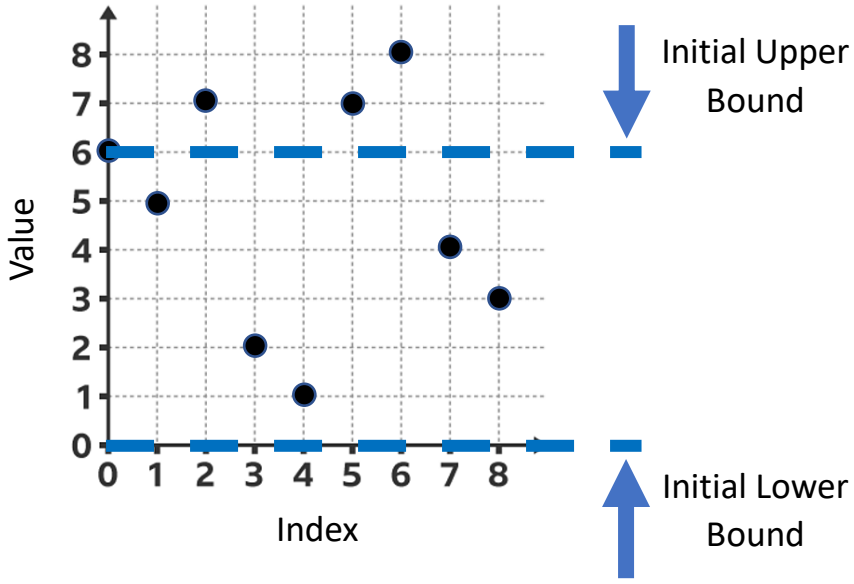

Fig. 2. Visualizing Dürr and Høyer. An example plot of list element value as a function of list index.

Because we are searching a finite list, the function plotted in Fig. 2 has a finitie domain. We are performing binary seach along the y-axis. Because we assumed all city costs $\emptyset > 0$, we can use an initial lower bound of 0. Since we are looking for the minimum, we can just use the first element in our search space as our initial upper bound; worst case scenario, this first element is the largest. At each iteration, we find the midpoint between the bounds, and use Grover to search for an element smaller than said midpoint. If such an element exists, we lower our upper bound to the midpoint, and if no such element exists, we raise our lower bound to the midpoint. We continue in this fasion till the bounds coverge on the minimum element [6].

For a sorted search space of size M, binary search has the well-known time complexity $O(log(M))$. Here, $M = initial\_upper\_bound - initial\_lower\_bound =$ `arr[0]` where `arr` is the list we are searching.

Grover's search is a quantum algorithm for unstructured search. The quantum circuit for Grover's search is shown in Fig. 3.

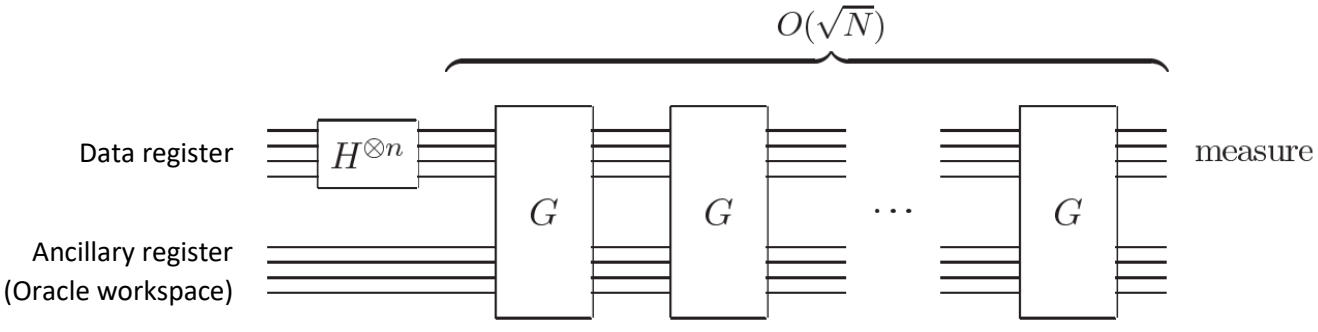

Fig. 3. Circuit for Grover's Search. [8, pg. 251]

Grover's search requires $O(\sqrt{2^n} = \sqrt{N})$ calls to a marking oracle, G, for a search space of n elements [8]. Here, n is the length of our input list `arr`. Combining the complexity of binary search with that of Grover's search, we find the Dürr and Høyer algorithm requires $O(\sqrt{2^n} \cdot log(M))$ queries to G.

Both the algorithm presented by Srinivasan and colleagues in [1] and our adaptation represent a quadratic speedup over the classical brute-force approach. We are not improving efficiency by performing phase estimation (as discussed above, phase estimation is unnecessary here), but we get this quadratic speedup by utilizing Grover. This quadratic speedup aligns with the quadratic speedups featured in our literature review, namely [3] and [4].

4. Implementation

Our TSP solver was implemented in Python. The source code lives here: https://github.com/mrl280/tsp. The source files are summarized in Table I.

TABLE I

SUMMARY OF SOURCE FILES

File	Description of Contents	Quantum or Classical
`pre_computed_hamiltonian_cycles.py`	We assume access to pre-computed Hamiltonian cycles; they are defined here.	Completely classical
`find_cycle_costs.py`	Code to compute cycle costs from an input cost matrix using pre-computed Hamiltonian cycles.	Completely classical
`phase_estimation.py`	Phase estimation code, including functions to build U' from input cost matrix A, the phase estimation circuit, and the overhead to build and execute the quantum circuit.	Contains both classical and quantum components
`grover_enhanced_minimization.py`	The classical binary search overhead for the Dürr and Høyer quantum algorithm to find the minimum.	Completely classical
`grover_for_minimization.py`	The quantum circuit (including the quantum oracle) and the overhead to build and execute the quantum circuit.	Contains both classical and quantum components
`tsp.py`	All the pieces fit together into a complete solver.	Completely classical

Herein, we limit our discussion to the quantum components of the implementation, namely phase estimation and Grover's search. However, the reader is encouraged to explore the classical components of the source code, they are well documented and quite straight forward.

All quantum circuits were simulated on classical hardware.

4.1. PennyLane

Xanadu is a full-stack quantum computing company based in Toronto, Canada. One of their software products, PennyLane, is a cross-platform Python library for quantum computing [9].

PennyLane was used to implement the quantum components of our solution, such as the quantum circuits found in `phase_estimation.py` and `grover_for_minimization.py`. It is customary to

```
import pennylane as qml
```

Therefore, whenever we refer to some function `foo()` using `qml.foo()` syntax, we are referring to PennyLane's `foo()` function.

4.2. Phase Estimation

Phase estimation source code, including the quantum circuit `phase_estimation_circuit()`, lives in `phase_estimation.py`.

Recall that phase estimation is a quantum algorithm to estimate the phase, φ, of the eigenvalue, $\lambda = e^{2\pi i \cdot \varphi}$, of a unitary matrix U, given some eigenvector $|u\rangle$. That is, we are solving

$$U|u\rangle = e^{2\pi i \cdot \varphi}|u\rangle$$

for the phase, $\varphi \in [0, 1)$.

We start with a list of cycle costs. As an example, consider the following three-city input matrix:

$$A = \begin{bmatrix} 0 & 7 & 2 \\ 5 & 0 & 9 \\ 17 & 1 & 0 \end{bmatrix}.$$

The dimension of this example problem is $n = 3$, and so we expect $(3-1)! = 2$ Hamiltonian circuits. Using `find_cycle_costs()`, the costs of these two circuits are returned as a 2-element list: `[33, 8]`. Now, to encode these costs as exponents, we must normalize them to within the range $[0, 2\pi)$. To this end, we note that the maximum possible cycle cost is $n \cdot \max(A) = 51$, and the minimum possible cycle cost is $n \cdot \min(A) = 0$. Notice, we could use Dürr and Høyer to find these minimum and maximum values if we bounded the problem, which could be done by assuming some global lower and upper bounds (perhaps 0 and half the circumference of the Earth, respectively). Normalizing, we take

$$[33, \quad 8] \rightarrow [4.066, \quad 0.986],$$

and encode these values in our unitary matrix U':

$$U' = \begin{bmatrix} e^{4.066\, i} & \\ & e^{0.986\, i} \end{bmatrix}.$$

Because U' is diagonal, its eigenvectors are just the standard basis vectors,

$$|u_0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |u_1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The number of target wires, `n_target_wires`, depends on the dimension of the problem. The `target_wires` register represents the lower register in Fig. 1. In our case, since all eigenstates can be represented by a single qubit, we only require a single target wire. The target wire was prepared in the basis states with the help of `qml.BasisState()`.

We get to choose the number of estimation wires, `n_estimation_wires`, based on our desired accuracy. The `estimation_wires` register represents the upper register in Fig. 1. The error in our

estimate will decrease exponentially with the number of estimation qubits. For example, when we run phase estimation using only 3 estimation wires, we find,

$$[\varphi_1, \quad \varphi_2] = [31.875, \quad 6.375].$$

This is a 3.53% and 25.5% error, respectively. However, when we increase to 5 estimation wires, we find,

$$[\varphi_1, \quad \varphi_2] = [33.46875, \quad 7.96875],$$

and the errors reduce to 1.40% and 0.392%, respectively.

We conclude our discussion of phase estimation by noting that the eigenvalues of a diagonal matrix are just the diagonal elements. Therefore, if we have access to the matrix form of a diagonal operator, there is no need to perform phase estimation to obtain its eigenvalues.

4.3. Gover's Search

Gover's search was implemented as part of the Dürr and Høyer quantum algorithm for minimization. While the classical binary-search overhead is located in `grover_enhanced_minimization.py`, the quantum subroutine, `grover_for_minimization()`, is located in `grover_for_minimization.py`.

`grover_for_minimization(arr, x)` takes two input parameters, an array of integers `arr` and an integer `x`, and returns **`True`** if a value smaller than `x` exists in `arr`, and **`False`** otherwise. To better understand the problem that `grover_for_minimization()` is trying to solve, and to ensure our binary search overhead was working properly, we first implemented a completely classical function that performs the same task: `grover_for_minimization_classical()`.

Grover's search requires two registers. Our data register contains `len(arr)` qubits. And, for reasons that will become apparent when we discuss our Grover oracle implementation, our ancillary register (the oracle workspace) contains enough ancillary qubits to represent a binary encoding of `sum(arr)` or `x`, whichever is greater. Recall that with n bits you can represent the integers 0 through $2^n - 1$.

At its simplest, Grover's search consists of three steps (implemented in `minimization_circuit()`):

1. Create an equal superposition by applying Hadamard gates to each wire in the data register.
2. Use a marking oracle to mark solution states. That is, if the state $|i\rangle$ represents a solution, take $|i\rangle \rightarrow -|i\rangle$. Notice, this phase inversion is only applied to solution states, the oracle leaves all other states untouched.
3. Apply the Gover operator to amplify the probability of the marked elements (amplify the probability of obtaining correct solutions). The Grover operator is easily applied with `qml.GroverOperator()`.

Optimally, steps 2 and 3 are applied $\left\lceil \frac{\pi}{4}\sqrt{\frac{\mathrm{N}}{M}} \right\rceil$ times for a search space of size N containing M marked elements.

It is important to keep in mind that we need to check and see if the element returned from Grover is actually less than `x`. If there are no elements in `arr` less than `x`, no elements will be marked, and Grover will return a random element. If the returned element from Grover is less than `x`, we return **`True`**. Otherwise, we return **`False`**.

4.3.1. The Minimization Oracle

The most challenging part of this project was designing and implementing the oracle used in step 2 of Grover's search. To understand the chosen approach, consider the example search space $[7, 10, 6, 18]$. Notice, the length of this array is 4, so we will require 4 data qubits. Once we create a superposition, these four data bits together represent 16 binary states:

$$\frac{1}{4}[|0000\rangle + |0001\rangle + \cdots + |1110\rangle + |1111\rangle].$$

As shown in Fig. 4, we perform controlled additions to the ancillary register, controlled off the data register.

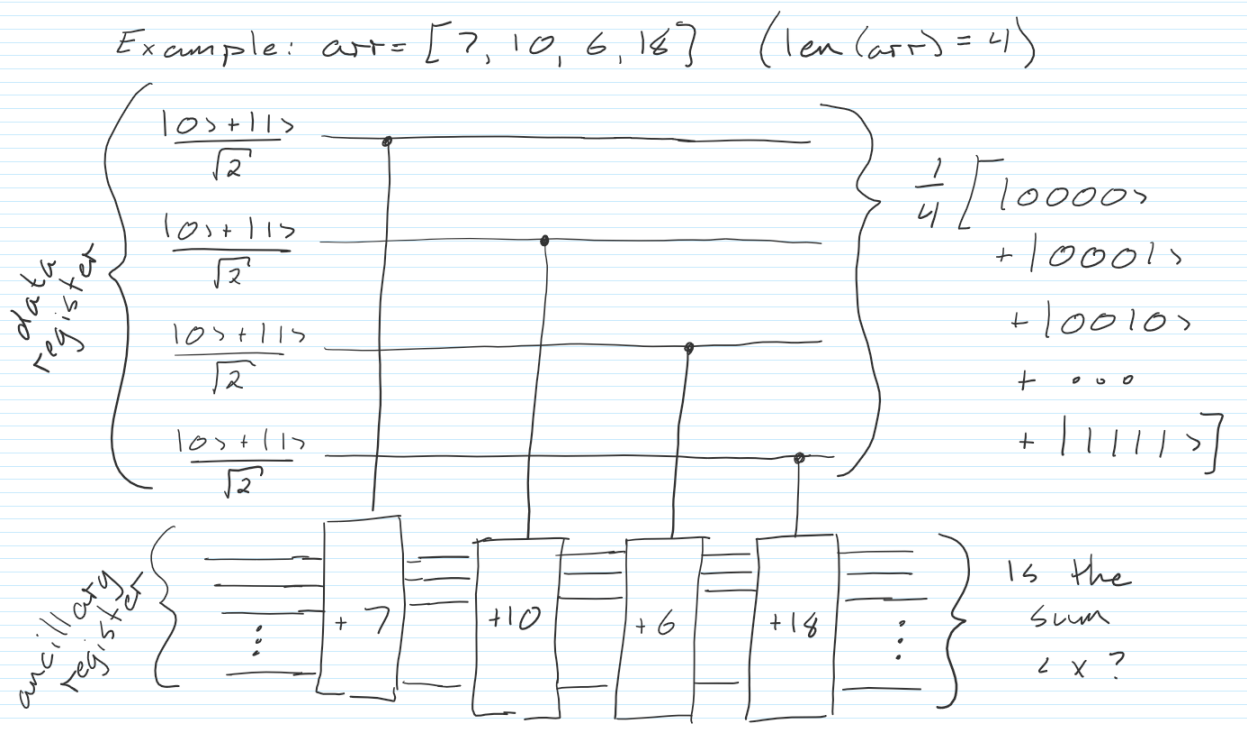

Fig. 4. The Minimization Oracle.

Additions are performed in the Fourier basis using `add_k_fourier()`. We use `qml.QFT()` and its adjoint, to convert back and forth between the computational basis and the Fourier basis.

Then, we just check to see if the sum on the ancillary wires is less than `x`. If so, we use `qml.FlipSign()` to mark the state. Notice we need to exclude the $|0000\rangle$ state because 0 will always be less than `x`. This is fine because we assumed all city costs $\emptyset > 0$.

To gain an improved understanding of how this works, let us consider a few examples. We start by testing with `x=4`, an element less than the minimum value in our search space. Fig. 5. shows the probablity distribution of the 16 states in our data register after searching for an element less than the minimum value in our search space.

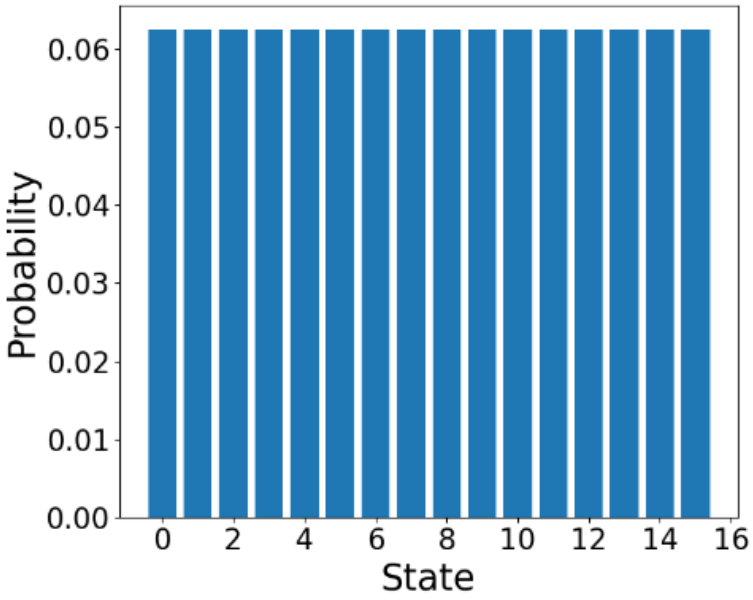

Fig. 5. Probability distribution when checking for elements less than `x=4` via `grover_for_minimization(arr=[7, 10, 6, 18], x=4)`

As seen in Fig. 5, because no solutions exist, no states are marked, and no states are amplified. In this case, we will get a random element, find it is greater than `x`, and end up returning **`False`**, as we should.

Next, we investigate the case where only one element in our search space is less than `x`. The results for `x=7` are shown in Fig. 6.

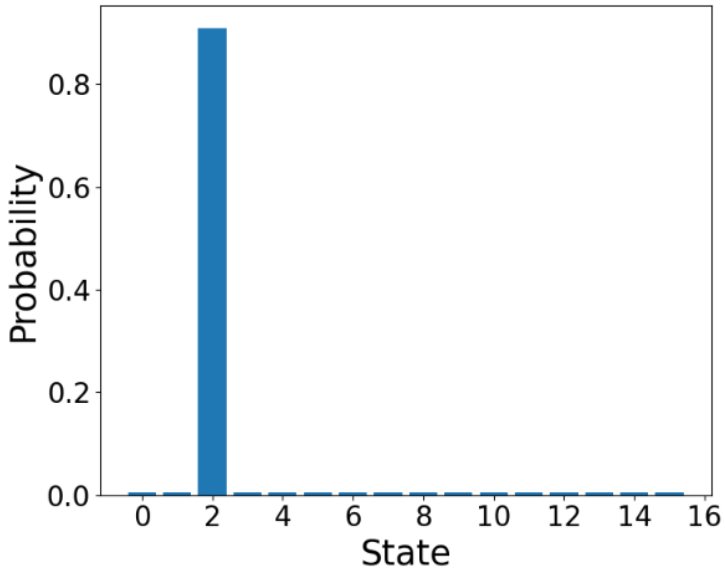

Fig. 6. Probability distribution when checking for elements less than `x=7` via `grover_for_minimization(arr=[7, 10, 6, 18], x=7)`

As seen in Fig. 6, when a single solutions exists, this single state is marked and amplified. A solution state $2 = |0010\rangle$ tells us that only the third element in the list, 6, is a solution. With greater than 50% probability, we measure this marked state, find $6 < 7$, and return `True`, as we should.

Finally, we investigate the case where several elements in our search space are less than `x`. The results for `x=14` are shown in Fig. 7.

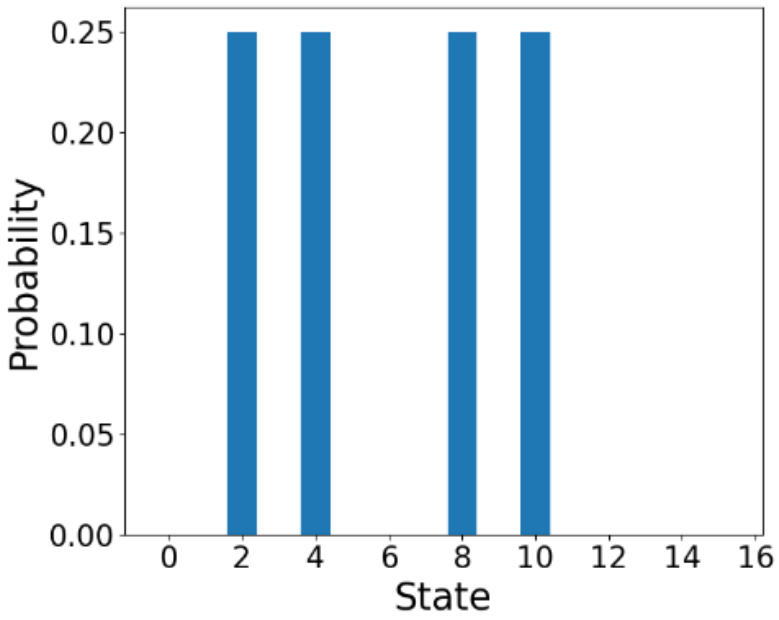

Fig. 7. Probability distribution when checking for elements less than `x=14` via `grover_for_minimization(arr=[7, 10, 6, 18], x=14)`

As seen in Fig. 7, we have 4 marked (and thus amplified) states. These solution states are summarized in Table II.

TABLE II

SOLUTION STATES WHEN SEARCHING FOR ELEMENTS LESS THAN `x=14`

State Number	Binary Encoding	Solution
2	$\|0010\rangle$	6
4	$\|0100\rangle$	10
8	$\|1000\rangle$	7
10	$\|1010\rangle$	$7 + 6$

The last row in Table II reminds us that sometimes we may get multiple-element solutions. This is fine because, if the sum of elements is less than `x`, then each element is sure to be less than x (another important consequence of our assumption that all city costs $\emptyset > 0$), so we can just use the first element.

An import takeaway from Fig. 6 is that Grover's search is, in its simplest form, not deterministic. In Fig. 7, because the success probability $\frac{4}{16} = \frac{1}{4}$ is a majic number, we were able to amplify the probability of success to 100%. However, we are not always so lucky.

4.3.2. Improving the Success Probability of Grover

Because the number of solution states are unknown, we don't know how many iterations of Grover are optimal. There are multiple strategies to address this, including the algorithm summarized in [10, Section 2], where we incrementally (exponentially) increase the number of Grover iterations from 1 to $\sqrt{2^n}$, each time checking to see if we have found a solution. If we find a solution, then great, we return early. Otherwise, we increase the number of Grover iterations to accommodate the fact that there might be more solutions than previously assumed.

However, even after an optimal number of Grover iterations, Grover's search is not deterministic (again, we see this in Fig. 6). To address this issue, we consider the approach suggested by Long in [11], where we replace the phase inversion in the oracle by a phase rotation through some angle,

$$\theta = 2 \cdot sin^{-1}\left[\frac{\sin\left(\frac{\pi}{4J_{opt}+6}\right)}{\frac{\pi}{2}}\right],$$

where,

$$J_{opt} = \left\lceil \frac{\frac{\pi}{2} - \beta}{2\beta} \right\rceil$$

for $\beta = sin^{-1}\left[\frac{1}{\sqrt{n}}\right]$ where n is the size of the search space. As further explained in [11], if we measure after $J_{opt} + 1$ Grover iterations, the probability of success,

$$P_{succ} = sin^2\Big(\big(2 \cdot J_{opt} + 1\big)\beta\Big).$$

By design, P_{succ} trends to unity for large n.

Because this approach requires an arbitrary phase rotation $R_z(\theta)$, we could no longer rely on `qml.FlipSign()`, and refactored the oracle to implement the phase flip with PauliX and a controlled-PauliZ gate. However, to achieve an arbitrary rotation, we needed to go one step further, and replace the controlled-PauliZ gate with a controlled-R_z gate. However,

$$R_z(\pi) = \begin{bmatrix} e^{-i\pi/2} & \\ & e^{i\pi/2} \end{bmatrix} = \begin{bmatrix} -i & \\ & i \end{bmatrix} = -iZ,$$

showing that $R_z(\pi)$ is equivalent to PauliZ only up to a global phase. For controlled rotations, this phase matters, meaning controlled-Z is not equal to controlled-$R_z(\pi)$. To overcome this issue, the circuit identity shown in Fig. 8 was used to move the rotation off the control.

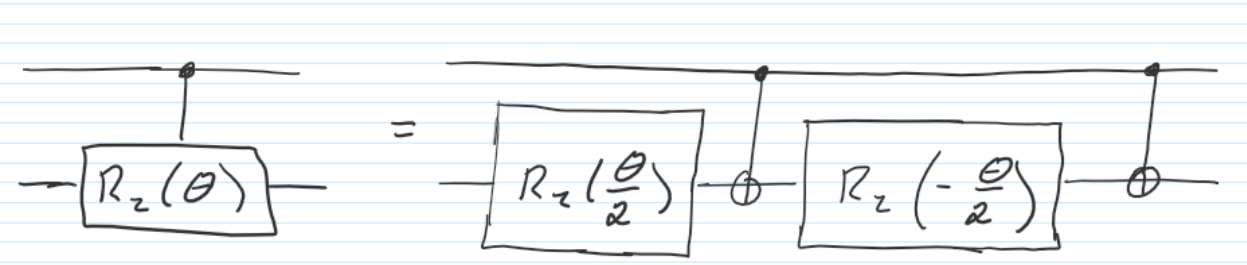

Fig. 8. Circuit identity allowing up to move rotations off the control.

These Grover improvements were tested for search spaces up to $n = 8$, and probability distributions agreed with [11] for these small values of n. Unfortunately, the advantages of Long's improvements are not reaped until we reach much larger $n \geq\approx 100$. At $n = 8$, $P_{succ} = sin^2\left(0.69 \cdot \frac{\pi}{2}\right) \approx 0.78$, which is not any better naïve Grover's search.

5. Future Work

Despite best efforts, several interesting tasks remain outstanding:

- In this project, we only studied a few of the most interesting quantum TSP algorithms. A complete review of TSP literature, both classical and quantum alike, remains outstanding.
- Based on quick search, the academic community has developed a wide array of Grover search improvements. While we studied and implemented a couple of these improvements, the vast majority remain unconsidered.
- All quantum subroutines were run in simulation. In the future, we should try running our quantum subroutines on actual quantum hardware. There are sure to be further restrictions imposed on our TSP solver by current qubit quality limitations.
- Throughout this project, we restricted ourselves to integer costs. In the future, we should extend functionality to support all comparable data types, especially floating-point numbers.
- As in [1], we assumed access to precomputed Hamiltonian circuits. This is a major simplifying assumption that future studies should address.

6. Conclusion

In this project we implemented a variation of the algorithm for the travelling salesman problem (TSP) presented by Srinivasan and colleagues in [1]. A TSP solver was developed in Python; the quantum components developed using Xanadu's cross-platform PennyLane library. The quantum components were simulated on classical hardware.

In their algorithm, Srinivasan and colleagues encode costs as phases and use phase estimation to extract only those total costs corresponding to pre-identified Hamiltonian circuits. Then, they employ the Dürr and Høyer algorithm to find the minimum cost circuit.

We noted that pre-identified Hamiltonian circuits allowed us to compile the list of total costs directly and therefore phase estimation was unnecessary here. However, phase estimation was still implemented and

integrated into our solver to improve understanding of the phase estimation algorithm itself. We ended up preforming phase estimation on a much smaller matrix than Srinivasan and colleagues.

The Dürr and Høyer quantum algorithm for minimization is just binary search on top of Grover's search. Gover's search required a minimization oracle, which was implement by performing controlled additions in Fourier space. Multiple improvements to Grover's search were studied and implemented in an attempt to make the algorithm deterministic even when the number of marked elements was unknown.

Both the algorithm presented by Srinivasan and colleagues and our adaptation represent a quadratic speedup over the classical brute-force approach. We do not see any speedups from phase estimation (which is unnecessary here), but we get the quadratic speedup by employing Grover for unstructured search. This quadratic speedup aligns with the quadratic speedups featured in the literature.

Despite initial confusion induced by Srinivasan and colleagues' unnecessary use of phase estimation, this turned out to be an excellent project, and I learned much about the travelling salesman problem, quantum phase estimation, the Dürr and Høyer quantum algorithm for minimization (and by extension Grover's search), and the PennyLane Python library.

7. References

[1] K. Srinivasan, S. Satyajit, B. K. Behera, and P. K. Panigrahi, "Efficient quantum algorithm for solving travelling salesman problem: An IBM quantum experience," 2018. https://arxiv.org/abs/1805.10928.

[2] D. Goswami, H. Karnick, P. Jain, and H. K. Maji, "Towards Efficiently Solving Quantum Traveling Salesman Problem," 2004, https://arxiv.org/abs/quant-ph/0411013.

[3] J. Bang, J. Ryu, C. Lee, S. Yoo, J. Lim, and J. Lee, "A quantum heuristic algorithm for the traveling salesman problem," *Journal of the Korean Physical Society*, vol. 61, no. 12, pp. 1944–1949, 2012, http://doi.org/10.3938/jkps.61.1944.

[4] D. J. Moylett, N. Linden, and A. Montanaro, "Quantum speedup of the traveling-salesman problem for bounded-degree graphs," *Physical review*. vol. 95, no. 3, 2017, http://doi.org/10.1103/PhysRevA.95.032323.

[5] M. Xiao and H. Nagamochi, "An Exact Algorithm for TSP in Degree-3 Graphs Via Circuit Procedure and Amortization on Connectivity Structure," *Algorithmica*, vol. 74, no. 2, pp. 713–741, 2016, http://doi.org/10.1007/s00453-015-9970-4.

[6] C. Dürr and P. Høyer, "A Quantum Algorithm for Finding the Minimum," 1996. https://arxiv.org/abs/quant-ph/9607014.

[7] IBM, "Solving the Travelling Salesman Problem using Phase Estimation" in *The Quiskit Textbook*. 2021. [Online]. https://qiskit.org/textbook/ch-paper-implementations/tsp.html.

[8] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, 10th anniversary ed. Cambridge: Cambridge University Press, 2010.

[9] V. Bergholm et al., "PennyLane: Automatic differentiation of hybrid quantum-classical computations," 2018, https://arxiv.org/10.48550/arxiv.1811.04968.

[10] F. Song, "Early days following Grover's quantum search algorithm," 2017, https://arxiv.org/10.48550/arxiv.1709.01236.

[11] G. L. Long, "Grover algorithm with zero theoretical failure rate," *Physical review*. *A, Atomic, molecular, and optical physics*, vol. 64, no. 2, 2001, http://doi.org/10.1103/PhysRevA.64.022307.