A Quantum Algorithm for the Bottleneck Travelling Salesman Problem

by

Raveel Tejani

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE

in

The Faculty of Science

(Physics)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2024

Abstract

chapter not complete.

Table of Contents

List of Tables

List of Figures

Acknowledgements

chapter not complete.

Chapter 1

Introduction

Continuous advancements in quantum computing have opened new ways to solve complex computational problems that were previously considered intractable using classical methods. Well known examples of these advancements include Shor's [1] and Grover's [2] algorithm for factoring and unstructured search respectively. The Bottleneck Traveling Salesman Problem (BTSP) serves as a challenging optimization problem in the field of logistics and operations research. Its practical applications range from optimizing vehicle routes to circuit design. Consequently, achieving an efficient solution is highly sought after. However, the BTSP is classified as an NP-hard problem, indicating it is at least as difficult as the hardest problems in the NP class. For reference, an NP problem must satisfy two conditions: no known solution in polynomial time, and a solution can be verified in polynomial time. An NP-hard problem does not need to satisfy the verification condition. To address this problem, many heuristic approaches are explored [3][4]. However, inherent to such methods is a trade-off between precision and efficiency.

chapter not complete.

Chapter 2

Theory

2.1 The Bottleneck Travelling Salesman Problem

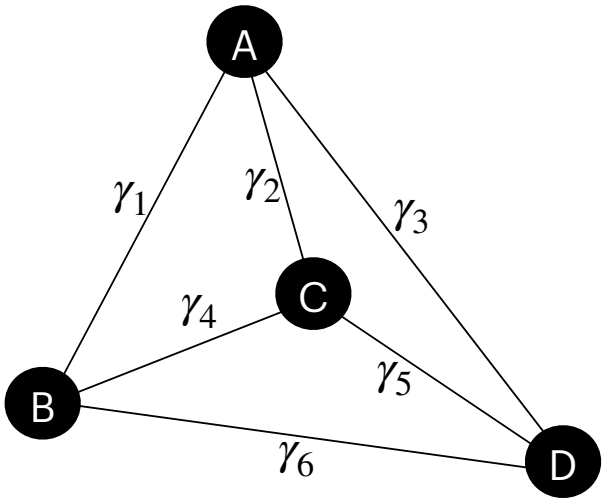


Figure 2.1: An undirected weighted graph represention of a symmetric 4-city system. The vertices represent cities and the edge weights represent the cost of travel.

The BTSP can be represented as a graph problem. We start with a graph, whose vertices are labelled A through D, representing a 4-city system. We define movement from one vertex to another as a walk, done through the edges connecting our vertices. We are interested in a particular walk known as a Hamiltonian cycle that contains every vertex exactly once before returning to the start. Our graph also includes edge weights, we define as γ_i. The BTSP is to find the Hamiltonian cycle in a graph, where the largest edge weight (bottleneck) is minimized. This is distinct from the Travelling Salesman Problem (TSP) where the combined edge weights in a given cycle is minimized. The total possible Hamiltonian cycles is given by $(N-1)!$, where N is the number of nodes. We present a symmetric case in FIG .4.4, $N_k \to N_{k+1} = N_{k+1} \to N_k = \gamma_i$. Thus the total possible cycles is $(N-1)!/2$. It is important to note that a solution to either BTSP or TSP is not unique. BTSP solutions also do not necessarily equate to the TSP solutions. We can illustrate an example below. Consider all the Hamiltonian cycles for a symmetric 4-city system:

$$A \to B \to C \to D \to A$$
$$A \to B \to D \to C \to A$$
$$A \to C \to B \to D \to A$$

Assigning some arbitrary weights, we can see the total costs of the cycles below. The first cycle is the solution to BTSP as its largest edge weight at 5 is the smallest among all three. The last cycle is a solution to the TSP as its combined edge weight is the smallest.

$$\gamma_1 + \gamma_4 + \gamma_5 + \gamma_3 = 4 + 4 + 5 + 4 = 17$$
$$\gamma_1 + \gamma_6 + \gamma_5 + \gamma_2 = 4 + 6 + 5 + 2 = 17$$
$$\gamma_2 + \gamma_4 + \gamma_6 + \gamma_3 = 2 + 4 + 6 + 4 = 16$$

By simply changing the weight of γ_6 to 5, we can illustrate all cycles are solutions to the BTSP.

$$\gamma_1 + \gamma_4 + \gamma_5 + \gamma_3 = 4 + 4 + 5 + 4 = 17$$
$$\gamma_1 + \gamma_6 + \gamma_5 + \gamma_2 = 4 + 5 + 5 + 2 = 16$$
$$\gamma_2 + \gamma_4 + \gamma_6 + \gamma_3 = 2 + 4 + 5 + 4 = 15$$

The computational complexity of the BTSP is known to be NP-hard. Implying there is no algorithm for a solution in polynomial time. A brute-force approach would imply that we can run an algorithm in $O((N-1)!)$ time.

2.2 Quantum Computing Basics

2.2.1 Frequently Used Notation

Single qubit states:

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

2-qubit states:

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, |01\rangle = |0\rangle \otimes |1\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$|00\rangle = |1\rangle \otimes |0\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, |01\rangle = |1\rangle \otimes |1\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Multiple qubit states are achieved through tensor products of single states. An n-qubit system will span a Hilbert space $N = 2^n$. We can see with the 2-qubit system we have 4 states. Quantum logic gates serve as the quantum computing counterparts to classical logic gates, performing operations on qubits instead of classical bits. Represented as unitary matrices, these operations must be reversible. Below we can see the Hadamard gate and the result of its application to single qubit states, frequently employed to establish a uniform superposition.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

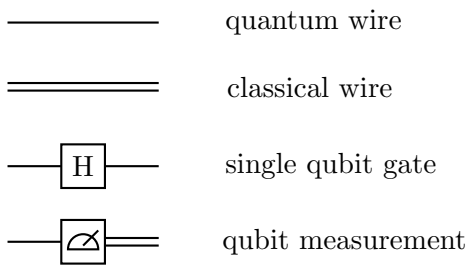
$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$$
$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$$

We can refer to equation 1 to see the result of Hadamard on an n-qubit system. Let's have a look at the Hadamard gate applied to $|00\rangle$:

$$\begin{aligned} H^{\otimes 2}|00\rangle &= H|0\rangle H|0\rangle \\ &= |+\rangle|+\rangle \\ &= (\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle))(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)) \\ &= \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle + |1\rangle) \\ &= \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \end{aligned}$$

Here we can see why Dirac notation is powerful as it simplifies tensor products and removes need for matrix representation.

2.2.2 Quantum Circuit Components



2.2.3 Controlled Gates and Phase Kickback

section not complete

1. We will describe the use of control gates.
2. We will describe the emergence of phase kickback with the use of hadamard gates discussed in the previous section. This will be important to know going into the next section.

2.3 Quantum Phase Estimation

The phase estimation algorithm initially proposed by Alexey Kitaev [5] plays an important role as a subroutine for the more widely known factoring algorithm by Peter Shor [1]. We first must briefly discuss the Quantum Fourier Transform (QFT) as it is key to understanding phase estimation [6]. Given a computational basis state $|x\rangle$, applying the QFT (F_N) results in:

$$F_N|x\rangle = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i x k N^{-1}} |k\rangle$$

Let's represent this in binary notation and decompose it into a tensor product. We can represent $|x\rangle$ as a string of bits, and the QFT as a tensor product of single qubit basis states:

$$|x\rangle = |x_1 x_2 ... x_n\rangle = |x_1\rangle \otimes |x_2\rangle \otimes ... \otimes |x_n\rangle$$

$$F_N|x\rangle = \frac{1}{\sqrt{2^n}} \bigotimes_{j=1}^{n} (|0\rangle + e^{2\pi i x 2^{-j}}|1\rangle) \tag{2.1}$$

$$= \frac{1}{\sqrt{2^n}} ((|0\rangle + \omega_1|1\rangle) \otimes (|0\rangle + \omega_2|1\rangle) \otimes ... \otimes (|0\rangle + \omega_n|1\rangle))$$

$$\begin{aligned} \omega_1 &= e^{2\pi i x 2^{-1}} = e^{2\pi i (0.x_n)} \\ \omega_2 &= e^{2\pi i x 2^{-2}} = e^{2\pi i (0.x_{n-1} x_n)} \\ &... \\ \omega_n &= e^{2\pi i x 2^{-n}} = e^{2\pi i (0.x_1 ... x_n)} \end{aligned}$$

An important characteristic of the w_j is the bit shift occurring in the exponent. If we look at w_1, the exponent has a factor $x2^{-1}$, which is equivalent to one right bit shift: $x_1 ... x_{n-1}.x_n$. Integer multiples of the exponent would imply full rotations returning to the same point thus we can ignore all the values on the left of the decimal and what remains is $0.x_n$.

Let's discuss the phase estimation problem. Given an eigenstate $|\lambda\rangle$ of a unitary operator U, we want to calculate a good approximation for $\phi \in [0, 1)$ satisfying:

$$U|\lambda\rangle = e^{2\pi i \phi}|\lambda\rangle \tag{2.2}$$

The phase estimation algorithm uses two registers of qubits. The first one will be a set of n control qubits that determine the precision of our approximation. The second register will be a set of m qubits initialized to an eigenstate $|\lambda\rangle$.

Let's walk through the quantum circuit in FIG. 2.2, to understand the inner workings of this algorithm. Our initialized state is $|0^{\otimes n}\lambda\rangle$. From here we perform the same operation we find in equation 1, where all the qubits in the first register are set to a uniform superposition on all states 2^n. The next portion of the algorithm involves applying controlled gates based on the unitary operator U. The function of these CU gates is to apply the operator U on $|\lambda\rangle$ if the control qubit is in the state $|1\rangle$. We can have a look at the effect on the n^{th} qubit, after it has been prepared in a superposition by the Hadamard gate:

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |\lambda\rangle = |0\lambda\rangle + |1\lambda\rangle$$

Applying CU and factoring out the eigenstate:

$$\begin{aligned} CU\frac{1}{\sqrt{2}}(|0\lambda\rangle + |1\lambda\rangle) &= \frac{1}{\sqrt{2}}(CU|0\lambda\rangle + CU|1\lambda\rangle) \\ &= \frac{1}{\sqrt{2}}(|0\lambda\rangle + e^{2\pi i\phi}|1\lambda\rangle) \\ &= \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i\phi}|1\rangle) \otimes |\lambda\rangle \end{aligned}$$

We can see the eigenstate after the CU operation is left unchanged. the phase has been encoded into the control qubit instead, a result is due to the phase kickback. Thus, we can

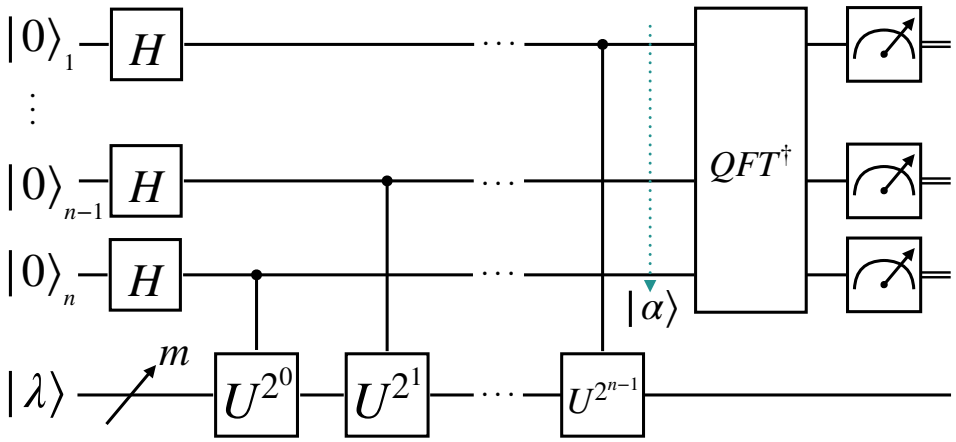


Figure 2.2: A quantum circuit representation of the phase estimation algorithm. Given, $U|\lambda\rangle = e^{2\pi i\phi}|\lambda\rangle$, this algorithm allows us to generate an approximation for $\phi \in [0, 1)$. The circuit consists of two registers of qubits, the first n-qubits are initialized to $|0\rangle$ and contribute to the precision of the ϕ value obtained. The second register of m-qubits is initialized to the eigenstate of U. The Hadamard gates, H, are used to create a uniform superposition in the first register. The control gates based on U are responsible for encoding phase to the qubits in the first register. Finally, a $QFT^\dagger$ is performed on the first register to extract the encoded phase. Each subsequent qubit in the first register would require double the control gates. Thus, with a large n we obtain a more precise value for ϕ, but also exponentially increase our computation time.

reuse our eigenstate for the next qubit. Consecutive qubits have double the amount of CU operators as the previous, thus squaring the eigenvalue each time:

$$\begin{aligned} \text{qubit}_{\text{n-1}} &: |0\rangle + e^{2\pi i\phi 2}|1\rangle \\ &... \\ \text{qubit}_1 &: |0\rangle + e^{2\pi i\phi 2^n}|1\rangle \end{aligned}$$

We know that the value of $\phi < 1$. We can represent this in binary notation in the form $0.\phi_1\phi_2...\phi_n$:

$$\phi = \sum_{j=1}^{n} \phi_j 2^{-j}$$

If we have another look at the control qubits using binary notation for ϕ instead, we can see the result of the multiple CU operations simply results in right bit shifts:

$$\begin{aligned} \text{qubit}_{\text{n}} &: |0\rangle + e^{2\pi i(0.\phi_1\phi_2...\phi_n)}|1\rangle \\ \text{qubit}_{\text{n-1}} &: |0\rangle + e^{2\pi i(0.\phi_2...\phi_n)}|1\rangle \\ &... \\ \text{qubit}_1 &: |0\rangle + e^{2\pi i(0.\phi_n)}|1\rangle \end{aligned}$$

If we look at the form of first register after all the CU operations in the state $|\alpha\rangle$, it will resemble the result of performing the QFT we saw in equation 5. Where our ω_j are:

$$\begin{aligned}
\omega_1 &= e^{2\pi i x 2^{-1}} = e^{2\pi i(0.\phi_n)} \\
\omega_2 &= e^{2\pi i x 2^{-2}} = e^{2\pi i(0.\phi_{n-1}\phi_n)} \\
&\ldots \\
\omega_n &= e^{2\pi i x 2^{-n}} = e^{2\pi i(0.\phi_1 \ldots \phi_n)}
\end{aligned}$$

simply performing the inverse QFT will give us $|\phi\rangle = |\phi_1\phi_2...\phi_n\rangle$. We can immediately see the approximation is limited by the number of qubits in the first register. A simple strategy would be to increase the number of qubits; however this would also increase computational cost as we double our use of CU gates for each additional qubit.

Chapter 3

Algorithm for the Constraint Problem

The constrained version of the BTSP asks whether there exists a Hamiltonian cycle in which the weight of every edge is less than a specified threshold α. In such a cycle, if we denote the weight of any edge as γ_i, then it must satisfy the condition:

$$\gamma_i < \alpha$$

We will construct the algorithm in the following steps:

1. Normalize edge weights so no single hamiltonian cycle is greater than or equal to 1. This is to ensure we can appropriately use the phase estimation algorithm.

2. Construct a unitary operator that holds information regarding the hamiltonian cyles as phases in the diagonal. The approach will follow inline with what was found the paper: [7]

3. Set all edgeweights $> \alpha$ to zero and construct a secondary unitary operator similiar to step 2.

4. Create controlled gates using the unitary operators constructed in Step 2 and 3.

5. Identify all the eigenstates of the unitary operators that map to the phases associated with the hamiltonian cycles.

6. Perform phase estimation twice with an eigenstate using the control gates to evaluate the hamiltonian cycle before and after the edgeweights $> \alpha$ are set to zero.

7. Compare the two phases achieved. If they are equal, the corresponding hamiltonian cycle is a solution that satisfies the constraint.

3.1 Normalize Edge Weights

A hamiltonian cycle of a complete graph with N nodes requires N edge-weights to complete the cycle. A single graph consists a total of $N(N-1)$ edgeweights in a directed graph. we can choose the largest N and use these to normalize the edge weights. Let w describe our edge-weights and will be a list of $m = N(N-1)$ elements:

$$w = \{w_1, w_2, \ldots, w_m\}$$

Sort w in descending order to obtain:

$$w' = \{w'_1, w'_2, \ldots, w'_m\} \tag{3.1}$$

Where: $w'_1 \geq w'_2 \geq \ldots \geq w'_m$

The sum S of the largest N items in w can be described as:

$$S = \sum_{i=1}^{N} w'_i \tag{3.2}$$

We can now perform the normalization. Let $\tilde{w}$ describe our normalized edge-weights:

$$\tilde{w} = (S + \epsilon)^{-1} w \tag{3.3}$$

Where: $\epsilon > 0$. The purpose of ϵ is to make sure if any normalized hamiltonian cycle is exactly equal to S then we do not have a zero phase.

3.2 Unitary Operator and Eigenstates associated with the Hamiltonian Cycles

We start by constructing diagonal matrices U_j, one for each node in a complete graph and describe the matrix elements:

$$[U_j]_{kk} = e^{2\pi i \gamma_{jk}(1-\delta_{jk})} \tag{3.4}$$

Where: $1 \leq j, k \leq N$ N denotes the total number of nodes. γ_{jk} represents the edgeweight connecting node $j \to k$

Then we construct U, a tensor product of all the diagonal matrices:

$$U = \bigotimes_{j}^{N} U_j \tag{3.5}$$

U will be a $N^N \times N^N$ matrix with only the diagonal elements populated. Because the diagonal elements will entirely consist of phases $[U]_{kk} = e^{i\alpha_{kk}}$. We can confirm the unitary operator condition is satisfied: $U^\dagger U = \mathbb{1}$.

Given U's diagonal nature, its eigenstates align with the basis vectors. Our focus is on specific eigenstates corresponding to the Hamiltonian cycles, determined by the phases. To comprehend how the diagonal elements of U derive from the individual U_j matrices, we visualize the tensor product construction. Notably, the product populates the diagonal elements of U allowing a simplification where we consider these elements directly:

$$
\begin{aligned}
[U]_0 &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_0 \cdot [U_{N-1}]_0 \cdot [U_N]_0 \\
[U]_1 &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_0 \cdot [U_{N-1}]_0 \cdot [U_N]_1 \\
&\vdots \\
[U]_{N-1} &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_0 \cdot [U_{N-1}]_0 \cdot [U_N]_{N-1} \\
[U]_N &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_0 \cdot [U_{N-1}]_1 \cdot [U_N]_0 \\
&\vdots \\
[U]_{2N-1} &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_0 \cdot [U_{N-1}]_1 \cdot [U_N]_{N-1} \\
[U]_{2N} &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_0 \cdot [U_{N-1}]_2 \cdot [U_N]_0 \\
&\vdots \\
[U]_{N^2-1} &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_0 \cdot [U_0]_{N-1} \cdot [U_N]_{N-1} \\
[U]_{N^2} &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_1 \cdot [U_{N-1}]_0 \cdot [U_N]_0 \\
&\vdots \\
[U]_{N^2+N-1} &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_1 \cdot [U_{N-1}]_0 \cdot [U_N]_{N-1} \\
[U]_{N^2+N} &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_1 \cdot [U_{N-1}]_1 \cdot [U_N]_0 \\
&\vdots \\
[U]_{N^3-1} &= [U_1]_0 \cdot [U_2]_0 \cdot \ldots \cdot [U_{N-2}]_{N-1} \cdot [U_{N-1}]_{N-1} \cdot [U_N]_{N-1}
\end{aligned}
$$

This pattern indicates:

$$[U]_k = [U_1]_{\alpha_{N-1}} \cdot [U_2]_{\alpha_{N-2}} \cdot \ldots \cdot [U_{N-2}]_{\alpha_2} \cdot [U_2]_{\alpha_1} \cdot [U_N]_{\alpha_0} \tag{3.6}$$

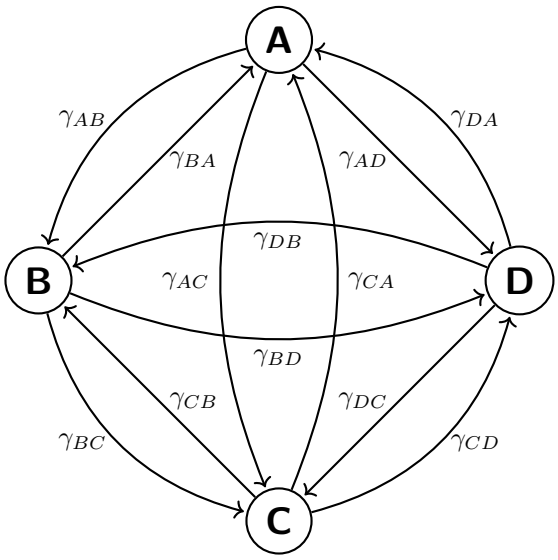
With:

$$\alpha_i = \left(k // \left(N^i\right)\right) \% N$$

Where:
// denotes integer division
% is the modulus operation

We can see the pattern follows base-N counting where N is the total number of nodes. Simply converting k into base-N, will give us the indices of our original diagonal matrices. To locate the relevent eigenstates, we start by identifying the elements in U_j associated with a hamiltonian cycle, retrieve their respective indices, convert this string of indices from Base-N to k. and we would have identified our eigenstate $|k\rangle$.

3.2.1 The Four City Graph



For the 4 city problem we will start with four matrices to represent the 12 edgeweights from each node. Using 3.4, Where: $j, k \in \{A, B, C, D\}$. we can construct matrix A:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i2\pi\gamma_{AB}} & 0 & 0 \\ 0 & 0 & e^{i2\pi\gamma_{AC}} & 0 \\ 0 & 0 & 0 & e^{i2\pi\gamma_{AD}} \end{bmatrix}$$

Since we only populate the diagonal elements, we can ignore the other elements of the matrix. Let $a = \text{diag}(A)$, and lets construct the other diagonals:

$$a = \begin{bmatrix} 1 \\ e^{i2\pi\gamma_{AB}} \\ e^{i2\pi\gamma_{AC}} \\ e^{i2\pi\gamma_{AD}} \end{bmatrix} \; b = \begin{bmatrix} e^{i2\pi\gamma_{BA}} \\ 1 \\ e^{i2\pi\gamma_{BC}} \\ e^{i2\pi\gamma_{BD}} \end{bmatrix} \; c = \begin{bmatrix} e^{i2\pi\gamma_{CA}} \\ e^{i2\pi\gamma_{CB}} \\ 1 \\ e^{i2\pi\gamma_{CD}} \end{bmatrix} \; d = \begin{bmatrix} e^{i2\pi\gamma_{DA}} \\ e^{i2\pi\gamma_{DB}} \\ e^{i2\pi\gamma_{DC}} \\ 1 \end{bmatrix}$$

We then construct the tensor product with equation 3.5:

$$U = A \otimes B \otimes C \otimes D \tag{3.7}$$

The convience of dealing with only the diagonals we can similiarly state $u = \text{diag}(U)$, thus:

$$u = a \otimes b \otimes c \otimes d$$

Refering to equation 3.6 regarding the matrix elements for U, the diagonal elements for the 4-city problem will reduce to:

$$u_k = a_{\alpha_3} \cdot b_{\alpha_2} \cdot c_{\alpha_1} \cdot d_{\alpha_0} \tag{3.8}$$

With:

$$\alpha_i = \left(k // \left(4^i\right)\right) \% 4$$

lets walk through identifying the eigenstate of one hamiltonian cycle. Lets say the cycle we choose is the following:

$$A \to D \to B \to C \to A \tag{3.9}$$

from here we can identify the edgeweights we care about are:

$$\gamma_{AD} + \gamma_{DB} + \gamma_{BC} + \gamma_{CA}$$

Thus the phase we would like to estimate would be given by the following element product:

$$a_3 \cdot d_1 \cdot b_2 \cdot c_0 = e^{i2\pi(\gamma_{AD}+\gamma_{DB}+\gamma_{BC}+\gamma_{CA})}$$

To correctly identify the eigenstate, we need to rearrange the product to match the form of equation 3.8:

$$a_3 \cdot d_1 \cdot b_2 \cdot c_0 = a_3 \cdot b_2 \cdot c_0 \cdot d_1$$

From here we simply read out the indices and based on our discussion under equation 3.6 , we can infer we are working in base 4. Thus we simply need to convert to base 10 to understand the exact column number and to base 2 to be used as the initialized eigenstate.

$$\text{base } 4 = 3201 \leftrightarrow \text{base } 10 = 225 \leftrightarrow \text{base } 2 = 11100001$$

Thus our eigenstate for 3.12 will be $|225\rangle$ or $|11100001\rangle$. We can perform an identical process for all the hamiltonian cycles to find their corresponding eigenstates. These are all listed in table 3.1 and and 3.2

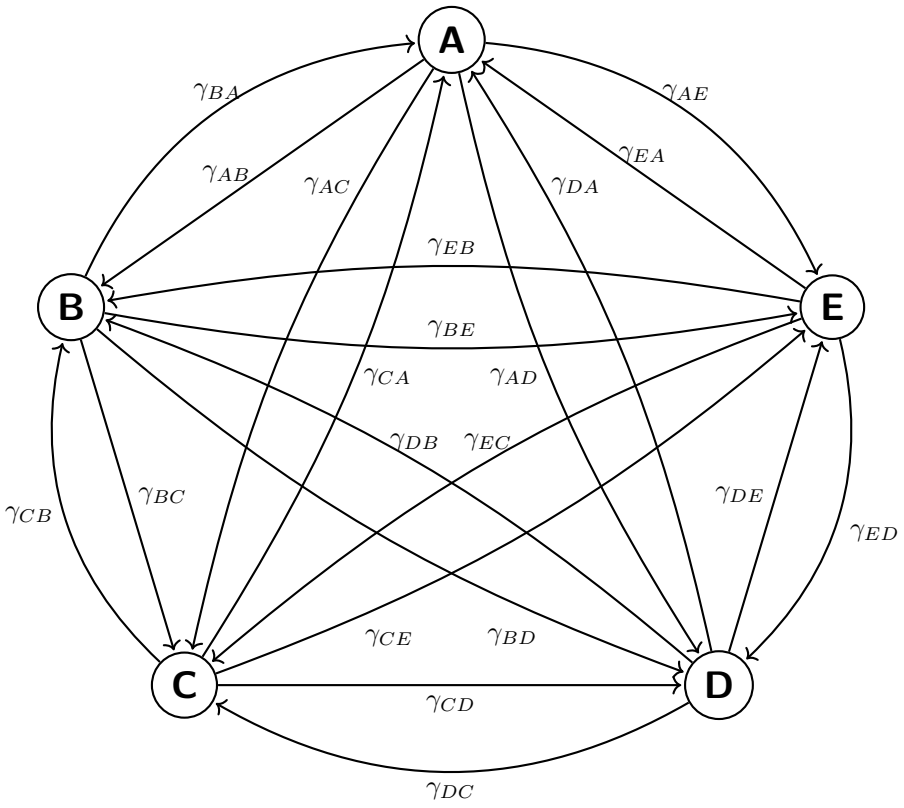
Hamiltonian Cycle	**Edge Weights Sum**	**Diagonal Elements Product**
$A \to B \to C \to D \to A$	$\gamma_{AB} + \gamma_{BC} + \gamma_{CD} + \gamma_{DA}$	$a_1 \cdot b_2 \cdot c_3 \cdot d_0$
$A \to B \to D \to C \to A$	$\gamma_{AB} + \gamma_{BD} + \gamma_{DC} + \gamma_{CA}$	$a_1 \cdot b_3 \cdot d_2 \cdot c_0$
$A \to C \to B \to D \to A$	$\gamma_{AC} + \gamma_{CB} + \gamma_{BD} + \gamma_{DA}$	$a_2 \cdot c_1 \cdot b_3 \cdot d_0$
$A \to C \to D \to B \to A$	$\gamma_{AC} + \gamma_{CD} + \gamma_{DB} + \gamma_{BA}$	$a_2 \cdot c_3 \cdot d_1 \cdot b_0$
$A \to D \to B \to C \to A$	$\gamma_{AD} + \gamma_{DB} + \gamma_{BC} + \gamma_{CA}$	$a_3 \cdot d_1 \cdot b_2 \cdot c_0$
$A \to D \to C \to B \to A$	$\gamma_{AD} + \gamma_{DC} + \gamma_{CB} + \gamma_{BA}$	$a_3 \cdot d_2 \cdot c_1 \cdot b_0$

Table 3.1: Hamiltonian cycles of the directed 4-city graph with their edgeweight summation and expected diagonal element products (3.8).

Rearranged Indices (Base 4)	**Base 10**	**Base 2**
1230	108	01101100
1302	114	01110010
2310	180	10110100
2031	141	10001101
3201	225	11100001
3012	198	11000110

Table 3.2: Eigenstates of matrix U (3.7), containing the normalized hamiltonian cycle edge weight sum of the directed 4-city graph.

3.2.2 The Five City Graph



We construct our matrix U with the tensor product as in equation 3.5.

$$U = A \otimes B \otimes C \otimes D \otimes E \tag{3.10}$$

Similiar to the four-city graph we only populate the diagonal elements, Let $a = \text{diag}(A)$, and let us construct the other diagonals:

$$a = \begin{bmatrix} 1 \\ e^{i2\pi\gamma_{AB}} \\ e^{i2\pi\gamma_{AC}} \\ e^{i2\pi\gamma_{AD}} \\ e^{i2\pi\gamma_{AE}} \end{bmatrix} \quad b = \begin{bmatrix} e^{i2\pi\gamma_{BA}} \\ 1 \\ e^{i2\pi\gamma_{BC}} \\ e^{i2\pi\gamma_{BD}} \\ e^{i2\pi\gamma_{BE}} \end{bmatrix} \quad c = \begin{bmatrix} e^{i2\pi\gamma_{CA}} \\ e^{i2\pi\gamma_{CB}} \\ 1 \\ e^{i2\pi\gamma_{CD}} \\ e^{i2\pi\gamma_{CE}} \end{bmatrix} \quad d = \begin{bmatrix} e^{i2\pi\gamma_{DA}} \\ e^{i2\pi\gamma_{DB}} \\ e^{i2\pi\gamma_{DC}} \\ 1 \\ e^{i2\pi\gamma_{DE}} \end{bmatrix} \quad e = \begin{bmatrix} e^{i2\pi\gamma_{EA}} \\ e^{i2\pi\gamma_{EB}} \\ e^{i2\pi\gamma_{EC}} \\ e^{i2\pi\gamma_{ED}} \\ 1 \end{bmatrix}$$

We can state $u = \text{diag}(U)$, thus:

$$u = a \otimes b \otimes c \otimes d \otimes e$$

Refering to equation 3.6 regarding the matrix elements for U, the diagonal elements for the 5-city problem will reduce to:

$$u_k = a_{\alpha_4} \cdot b_{\alpha_3} \cdot c_{\alpha_2} \cdot d_{\alpha_1} \cdot e_{\alpha_0} \tag{3.11}$$

With:

$$\alpha_i = \left(k // \left(5^i\right)\right) \%5$$

Let us walk through identifying the eigenstate of one hamiltonian cycle:

$$A \to E \to D \to C \to B \to A \tag{3.12}$$

from here we can identify the edgeweights we care about are:

$$\gamma_{AE} + \gamma_{ED} + \gamma_{DC} + \gamma_{CB} + \gamma_{BA}$$

Thus the phase we would like to estimate would be given by the following element product:

$$a_4 \cdot e_3 \cdot d_2 \cdot c_1 \cdot b_0 = e^{i2\pi(\gamma_{AE}+\gamma_{ED}+\gamma_{DC}+\gamma_{CB}+\gamma_{BA})}$$

To correctly identify the eigenstate, we need to rearrange the product to match the form of equation 3.11:

$$a_4 \cdot e_3 \cdot d_2 \cdot c_1 \cdot b_0 = a_4 \cdot b_0 \cdot c_1 \cdot d_2 \cdot e_3$$

From here we simply read out the indices and based on our discussion under equation 3.6 , we can infer we are working in base 4. Thus we simply need to convert to base 10 to understand the exact column number and to base 2 to be used as the initialized eigenstate.

$$\text{base } 5 = 40123 \leftrightarrow \text{base } 10 = 2538 \leftrightarrow \text{base } 2 = 100111101010$$

Thus our eigenstate for 3.12 will be $|2538\rangle$ or $|100111101010\rangle$. We can perform an identical process for all the hamiltonian cycles to find their corresponding eigenstates. These are all listed in table 3.3 and 3.4

Hamiltonian Cycle	**Edge Weights Sum**	**Matrix Elements Product**
$A \to B \to C \to D \to E \to A$	$\gamma_{AB} + \gamma_{BC} + \gamma_{CD} + \gamma_{DE} + \gamma_{EA}$	$a_1 \cdot b_2 \cdot c_3 \cdot d_4 \cdot e_0$
$A \to B \to C \to E \to D \to A$	$\gamma_{AB} + \gamma_{BC} + \gamma_{CE} + \gamma_{ED} + \gamma_{DA}$	$a_1 \cdot b_2 \cdot c_4 \cdot e_3 \cdot d_0$
$A \to B \to D \to C \to E \to A$	$\gamma_{AB} + \gamma_{BD} + \gamma_{DC} + \gamma_{CE} + \gamma_{EA}$	$a_1 \cdot b_3 \cdot d_2 \cdot c_4 \cdot e_0$
$A \to B \to D \to E \to C \to A$	$\gamma_{AB} + \gamma_{BD} + \gamma_{DE} + \gamma_{EC} + \gamma_{CA}$	$a_1 \cdot b_3 \cdot d_4 \cdot e_2 \cdot c_0$
$A \to B \to E \to C \to D \to A$	$\gamma_{AB} + \gamma_{BE} + \gamma_{EC} + \gamma_{CD} + \gamma_{DA}$	$a_1 \cdot b_4 \cdot e_2 \cdot c_3 \cdot d_0$
$A \to B \to E \to D \to C \to A$	$\gamma_{AB} + \gamma_{BE} + \gamma_{ED} + \gamma_{DC} + \gamma_{CA}$	$a_1 \cdot b_4 \cdot e_3 \cdot d_2 \cdot c_0$
$A \to C \to B \to D \to E \to A$	$\gamma_{AC} + \gamma_{CB} + \gamma_{BD} + \gamma_{DE} + \gamma_{EA}$	$a_2 \cdot c_1 \cdot b_3 \cdot d_4 \cdot e_0$
$A \to C \to B \to E \to D \to A$	$\gamma_{AC} + \gamma_{CB} + \gamma_{BE} + \gamma_{ED} + \gamma_{DA}$	$a_2 \cdot c_1 \cdot b_4 \cdot e_3 \cdot d_0$
$A \to C \to D \to B \to E \to A$	$\gamma_{AC} + \gamma_{CD} + \gamma_{DB} + \gamma_{BE} + \gamma_{EA}$	$a_2 \cdot c_3 \cdot d_1 \cdot b_4 \cdot e_0$
$A \to C \to D \to E \to B \to A$	$\gamma_{AC} + \gamma_{CD} + \gamma_{DE} + \gamma_{EB} + \gamma_{BA}$	$a_2 \cdot c_3 \cdot d_4 \cdot e_1 \cdot b_0$
$A \to C \to E \to B \to D \to A$	$\gamma_{AC} + \gamma_{CE} + \gamma_{EB} + \gamma_{BD} + \gamma_{DA}$	$a_2 \cdot c_4 \cdot e_1 \cdot b_3 \cdot d_0$
$A \to C \to E \to D \to B \to A$	$\gamma_{AC} + \gamma_{CE} + \gamma_{ED} + \gamma_{DB} + \gamma_{BA}$	$a_2 \cdot c_4 \cdot e_3 \cdot d_1 \cdot b_0$
$A \to D \to B \to C \to E \to A$	$\gamma_{AD} + \gamma_{DB} + \gamma_{BC} + \gamma_{CE} + \gamma_{EA}$	$a_3 \cdot d_1 \cdot b_2 \cdot c_4 \cdot e_0$
$A \to D \to B \to E \to C \to A$	$\gamma_{AD} + \gamma_{DB} + \gamma_{BE} + \gamma_{EC} + \gamma_{CA}$	$a_3 \cdot d_1 \cdot b_4 \cdot e_2 \cdot c_0$
$A \to D \to C \to B \to E \to A$	$\gamma_{AD} + \gamma_{DC} + \gamma_{CB} + \gamma_{BE} + \gamma_{EA}$	$a_3 \cdot d_2 \cdot c_1 \cdot b_4 \cdot e_0$
$A \to D \to C \to E \to B \to A$	$\gamma_{AD} + \gamma_{DC} + \gamma_{CE} + \gamma_{EB} + \gamma_{BA}$	$a_3 \cdot d_2 \cdot c_4 \cdot e_1 \cdot b_0$
$A \to D \to E \to B \to C \to A$	$\gamma_{AD} + \gamma_{DE} + \gamma_{EB} + \gamma_{BC} + \gamma_{CA}$	$a_3 \cdot d_4 \cdot e_1 \cdot b_2 \cdot c_0$
$A \to D \to E \to C \to B \to A$	$\gamma_{AD} + \gamma_{DE} + \gamma_{EC} + \gamma_{CB} + \gamma_{BA}$	$a_3 \cdot d_4 \cdot e_2 \cdot c_1 \cdot b_0$
$A \to E \to B \to C \to D \to A$	$\gamma_{AE} + \gamma_{EB} + \gamma_{BC} + \gamma_{CD} + \gamma_{DA}$	$a_4 \cdot e_1 \cdot b_2 \cdot c_3 \cdot d_0$
$A \to E \to B \to D \to C \to A$	$\gamma_{AE} + \gamma_{EB} + \gamma_{BD} + \gamma_{DC} + \gamma_{CA}$	$a_4 \cdot e_1 \cdot b_3 \cdot d_2 \cdot c_0$
$A \to E \to C \to B \to D \to A$	$\gamma_{AE} + \gamma_{EC} + \gamma_{CB} + \gamma_{BD} + \gamma_{DA}$	$a_4 \cdot e_2 \cdot c_1 \cdot b_3 \cdot d_0$
$A \to E \to C \to D \to B \to A$	$\gamma_{AE} + \gamma_{EC} + \gamma_{CD} + \gamma_{DB} + \gamma_{BA}$	$a_4 \cdot e_2 \cdot c_3 \cdot d_1 \cdot b_0$
$A \to E \to D \to B \to C \to A$	$\gamma_{AE} + \gamma_{ED} + \gamma_{DB} + \gamma_{BC} + \gamma_{CA}$	$a_4 \cdot e_3 \cdot d_1 \cdot b_2 \cdot c_0$
$A \to E \to D \to C \to B \to A$	$\gamma_{AE} + \gamma_{ED} + \gamma_{DC} + \gamma_{CB} + \gamma_{BA}$	$a_4 \cdot e_3 \cdot d_2 \cdot c_1 \cdot b_0$

Table 3.3: Hamiltonian cycles of the directed 4-city graph with their edgeweight summation and expected diagonal element products (3.11).

Rearranged Indices (Base 5)	**Base 10**	**Base 2**
12340	970	001111001010
12403	978	001111010010
13420	1110	010001010110
13042	1022	001111111110
14302	1202	010010110010
14023	1138	010001110010
23140	1670	011010000110
24103	1778	011011110010
24310	1830	011100100110
20341	1346	010101000010
23401	1726	011010111110
20413	1358	010101001110
32410	2230	100010110110
34012	2382	100101001110
34120	2410	100101101010
30421	1986	011111000010
32041	2146	100001100010
30142	1922	011110000010
42301	2826	101100001010
43021	2886	101101000110
43102	2902	101101010110
40312	2582	101000010110
42013	2758	101011000110
40123	2538	100111101010

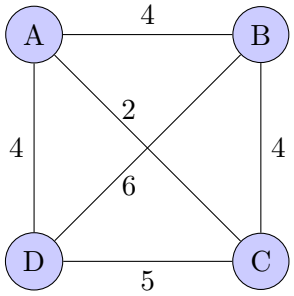
Table 3.4: Eigenstates of matrix U (3.10), containing the normalized hamiltonian cycle edge weight sum of the directed 4-city graph

Chapter 4

Constraint Solutions

Lets consider the following example of a symmetric 4-city system we are considering the symmetric case thus we only need to look at three hamiltonian cycles. The constraint for our BTSP in this case will involve $\gamma < 6$:

4.1 An Undirected 4-city Graph



$$w = \{w_{AB}, w_{AC}, w_{AD}, w_{BC}, w_{CD}, w_{BD}\} = \{4, 2, 4, 4, 5, 6\}$$

4.1.1 Algorithm Construction

We will follow the instructions highlighted at the begining of chapter 3. We start by normalizing our edge weights. We need to sort our all edge weights in descending order as in equation 3.1:

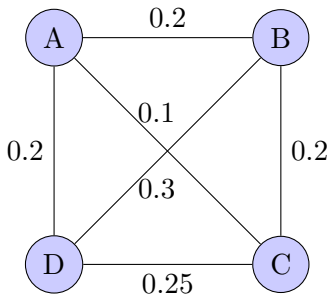
$$w' = \{6, 5, 4, 4, 4, 2\}$$

Then we need to retrieve the sum S as in equation 3.2

$$S = \sum_{i=1}^{4} w'_i = 6 + 5 + 4 + 4 = 19$$

From here we can normalize our edgeweights as in 3.3, we can set $\epsilon = 1$

$$\tilde{w} = \frac{\{4, 2, 4, 4, 5, 6\}}{20} = \{0.2, 0.1, 0.2, 0.2, 0.25, 0.3\}$$



Now we need to construct the unitary operator and eigenstates. Our matrix U and U' diagonals will look like the following:

$$u = \begin{bmatrix} 1 \\ e^{i2\pi(0.2)} \\ e^{i2\pi(0.1)} \\ e^{i2\pi(0.2)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.2)} \\ 1 \\ e^{i2\pi(0.2)} \\ e^{i2\pi(0.3)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.1)} \\ e^{i2\pi(0.2)} \\ 1 \\ e^{i2\pi(0.25)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.2)} \\ e^{i2\pi(0.3)} \\ e^{i2\pi(0.25)} \\ 1 \end{bmatrix}$$

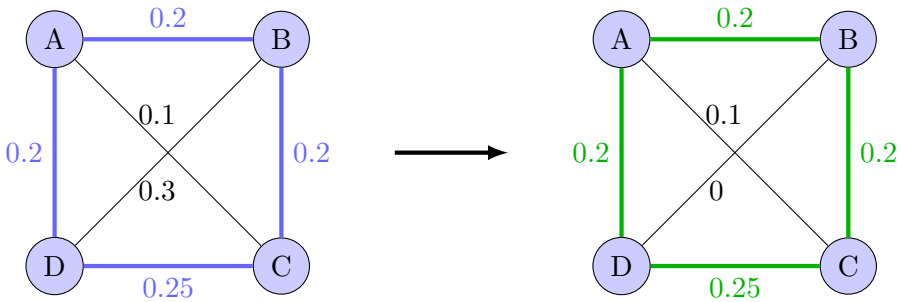
$$u' = \begin{bmatrix} 1 \\ e^{i2\pi(0.2)} \\ e^{i2\pi(0.1)} \\ e^{i2\pi(0.2)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.2)} \\ 1 \\ e^{i2\pi(0.2)} \\ e^{i2\pi(0)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.1)} \\ e^{i2\pi(0.2)} \\ 1 \\ e^{i2\pi(0.25)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.2)} \\ e^{i2\pi(0)} \\ e^{i2\pi(0.25)} \\ 1 \end{bmatrix}$$

Because we are dealing with the symmetric case. Based on Table 3.1 we can simply use the results and conversions from the first three cycles. Thus we will be estimating phases using the following eigenstates:

$$|108\rangle = |01101100\rangle$$
$$|114\rangle = |01110010\rangle$$
$$|180\rangle = |10110100\rangle$$

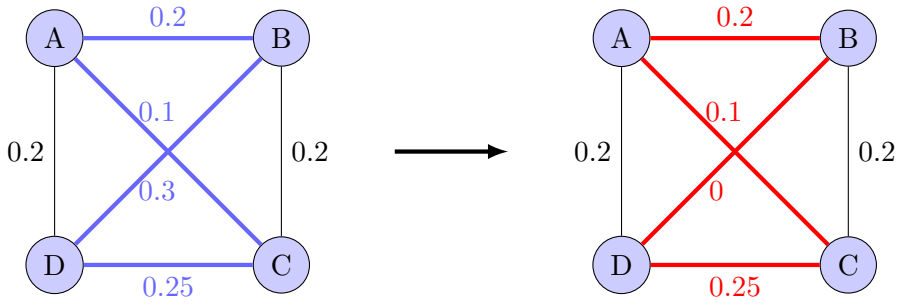
And expect the following phases:

Cycle 1: $A \to B \to C \to D \to A$



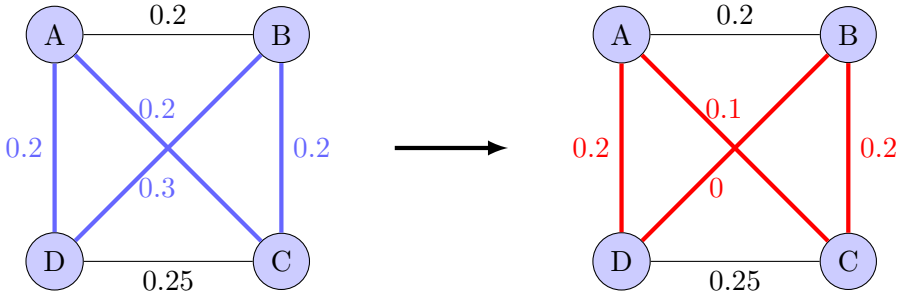
$$u_{108} = e^{i2\pi(\gamma_{AB}+\gamma_{BC}+\gamma_{CD}+\gamma_{DA})} = e^{i2\pi(0.85)}$$
$$u'_{108} = e^{i2\pi(\gamma_{AB}+\gamma_{BC}+\gamma_{CD}+\gamma_{DA})} = e^{i2\pi(0.85)}$$

Cycle 2: $A \to B \to D \to C \to A$



$$u_{114} = e^{i2\pi(\gamma_{AB}+\gamma_{BD}+\gamma_{DC}+\gamma_{CA})} = e^{i2\pi(0.85)}$$
$$u'_{114} = e^{i2\pi(\gamma_{AB}+\gamma_{BD}+\gamma_{DC}+\gamma_{CA})} = e^{i2\pi(0.55)}$$

Cycle 3: $A \to C \to B \to D \to A$



$$u_{180} = e^{i2\pi(\gamma_{AC}+\gamma_{CB}+\gamma_{BD}+\gamma_{DA})} = e^{i2\pi(0.80)}$$
$$u'_{180} = e^{i2\pi(\gamma_{AC}+\gamma_{CB}+\gamma_{BD}+\gamma_{DA})} = e^{i2\pi(0.50)}$$

4.1.2 Results: Simulations with Qiskit

The basis of our algorithm is to perform phase estimation before and after the edgeweights that do not satify the criteria above are removed. Below we highlight each hamiltonian cycles before and after the criteria. If the hamiltonian cycle is not affected we highlight in green vs red.

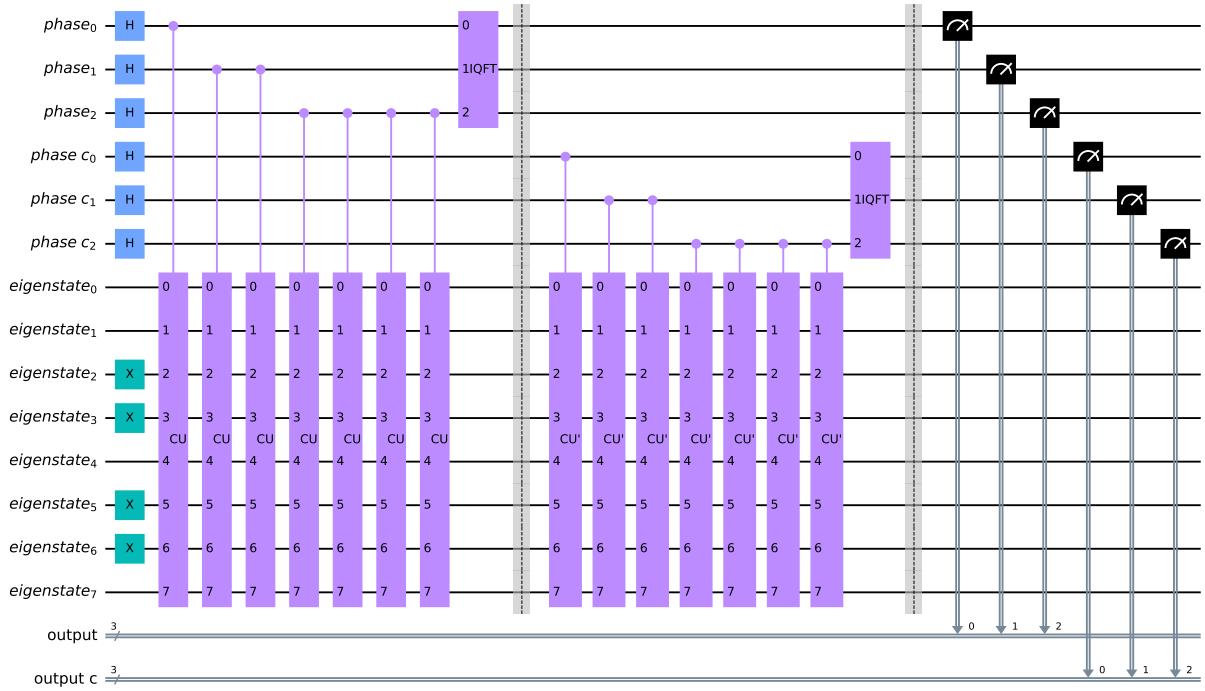
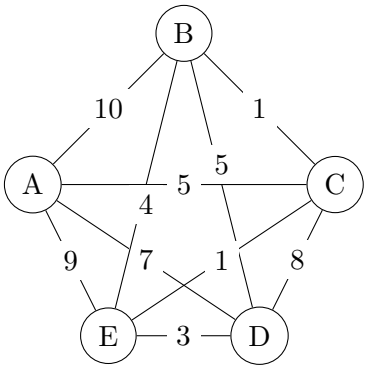


Figure 4.1: the quantum circuit for the BTSP: 3 qubit phase estimation is performed measuring the hamiltonian cycle $A \to B \to C \to D \to A$, using the corresponding eigenstate is $|01101100\rangle$. Due to Qiskit convention on qubit ordering, the eigentate is initialized in reverse. The CU gate denotes the control unitary matrix containing all the hamiltonian cycles. The CU' gate inhabits the same cycles but before it was constructed, all edgeweights not satisfying the constraint, $\geq \alpha$, were set to zero. We have two sets of 3 qubits to be measured and stored in to classical registers labelled 'output' and 'output c'.

4.2 An Undirected 5-City Graph



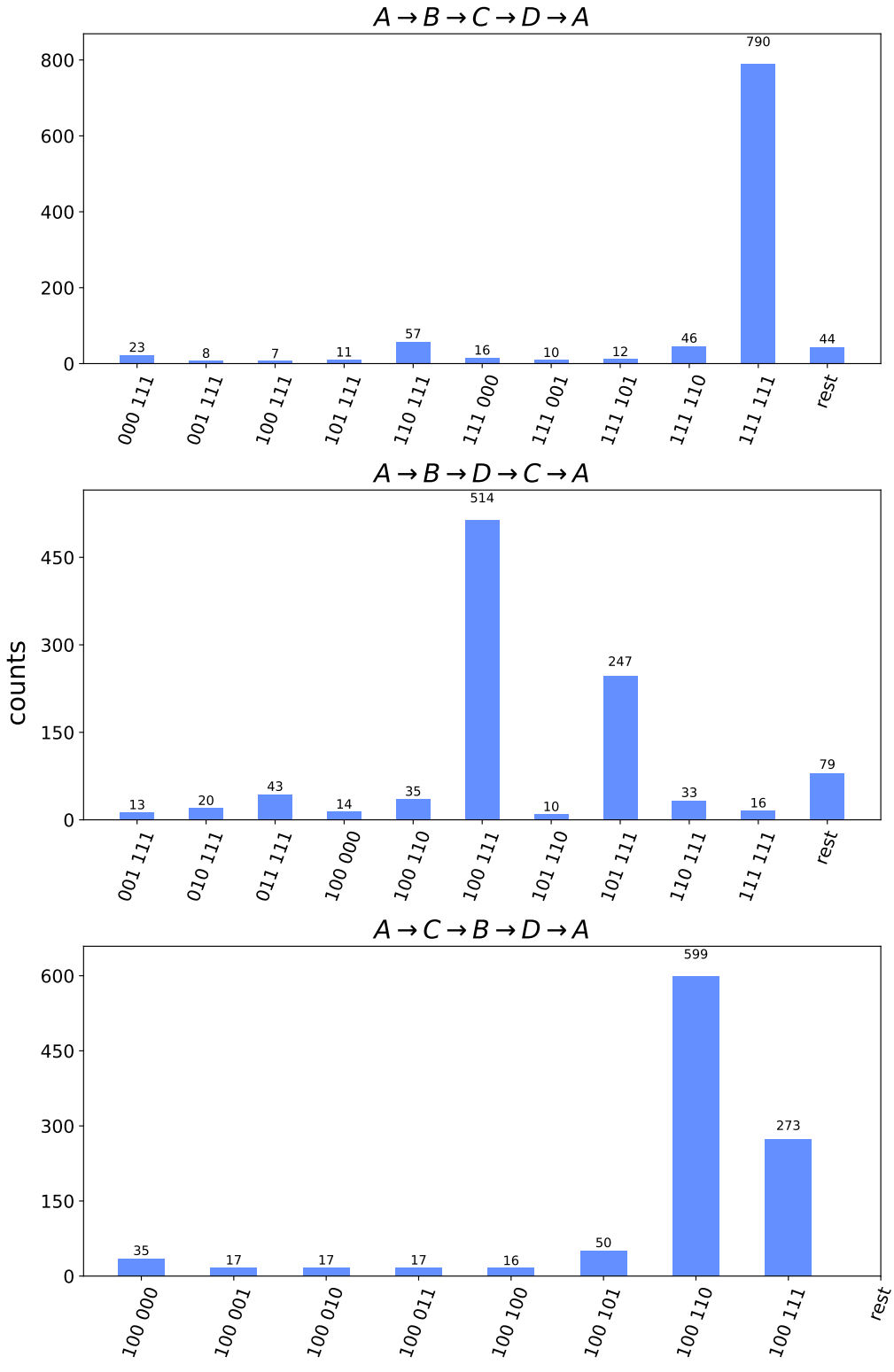


Figure 4.2: 3-qubit phase estimation for the 4-city problem

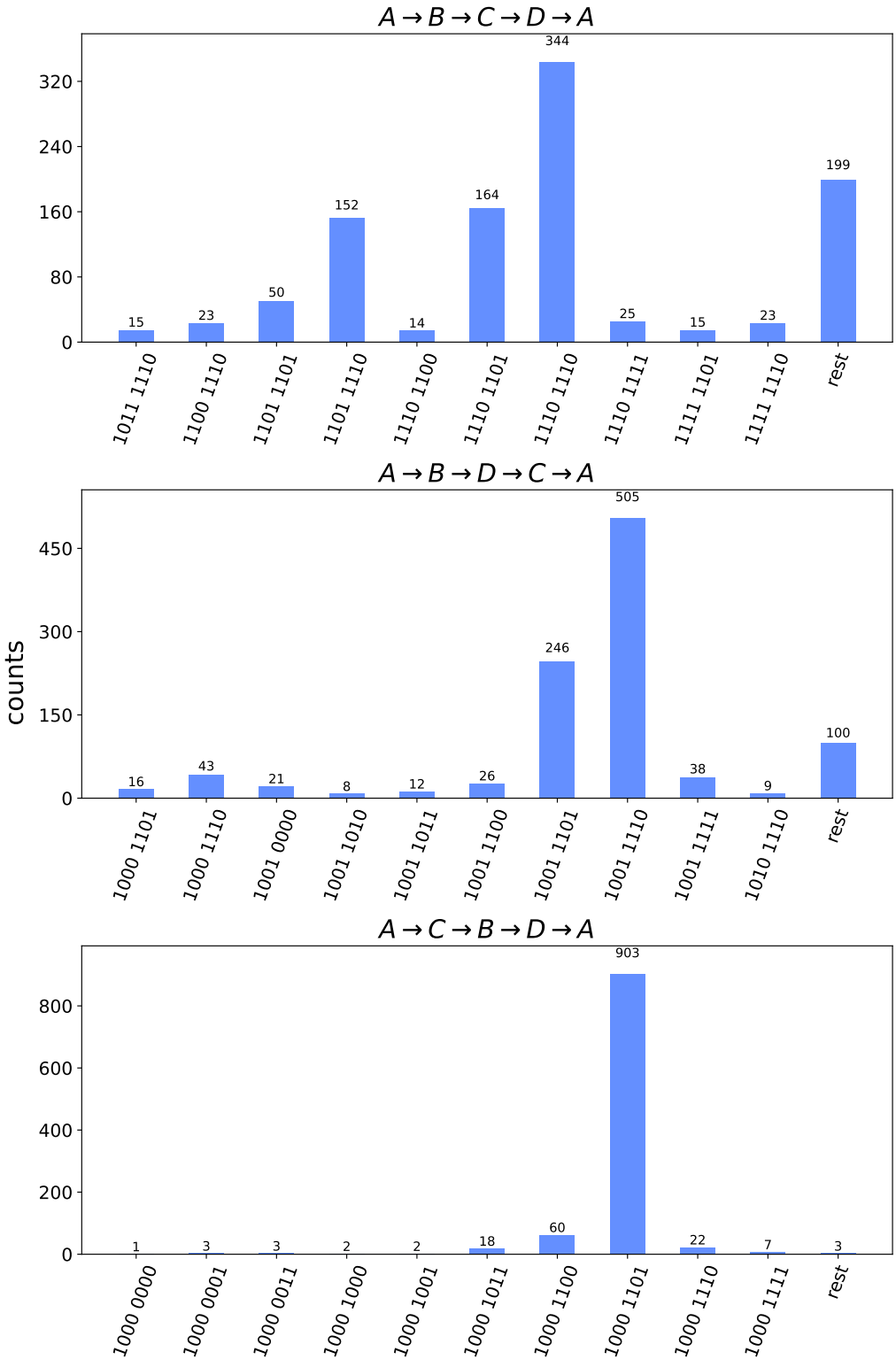


Figure 4.3: 4-qubit phase estimation for the 4-city problem

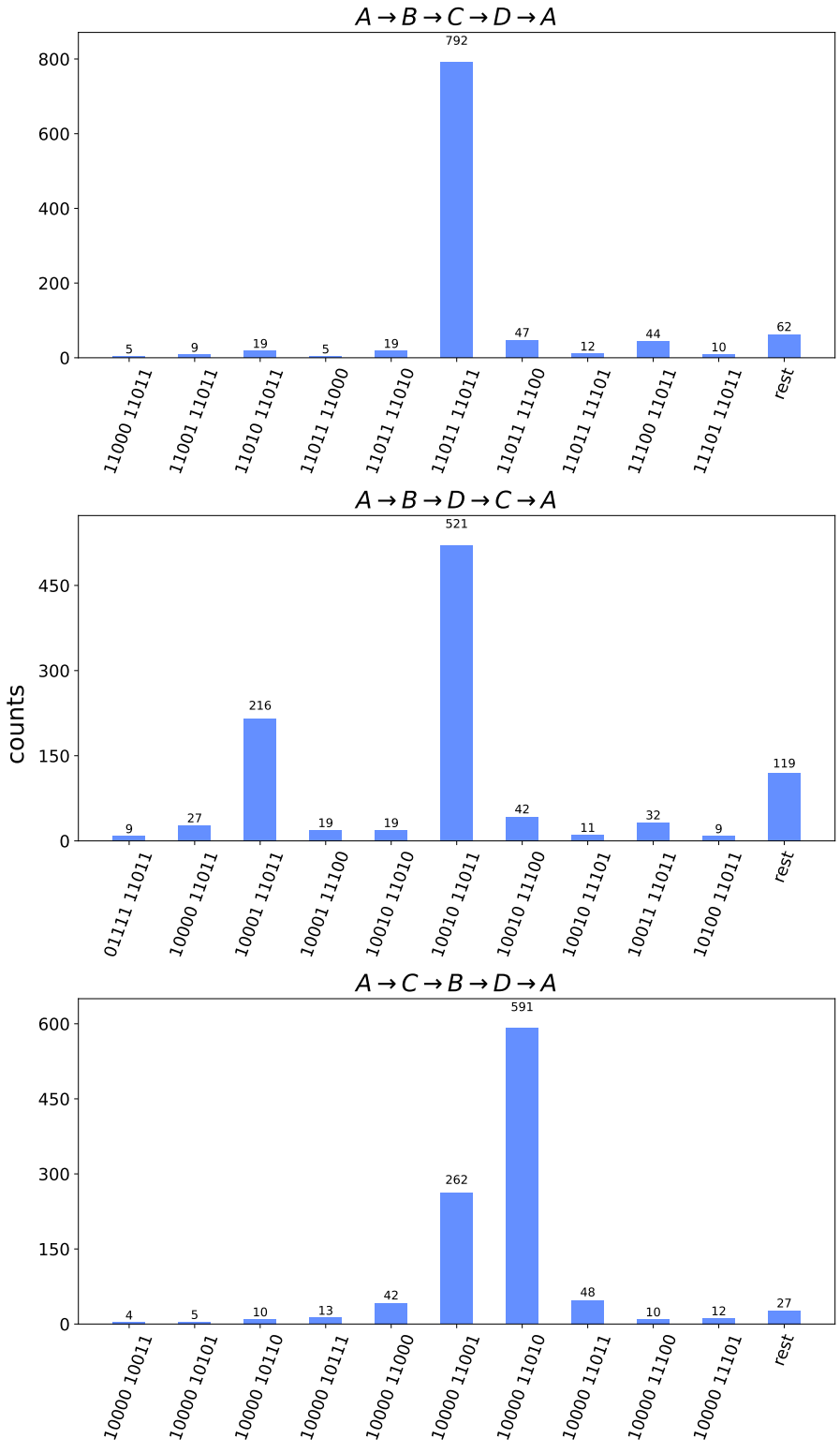


Figure 4.4: 5-qubit phase estimation for the 4-city problem

Chapter 5

Discussion

1. this section will dicuss the advantages and drawbacks of the algorithm proposed.
2. how the algorithm scales with the number of cities ($2^x = x^x$)

Bibliography

[1] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[2] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996.

[3] John Larusic. A heuristic for solving the bottleneck traveling salesman problem. 2005.

[4] Ravi Ramakrishnan, Prabha Sharma, and Abraham P. Punnen. An efficient heuristic algorithm for the bottleneck traveling salesman problem. *Opsearch*, 46(3):275–288, 09 2009. Copyright - Operational Research Society of India 2009; Last updated - 2023-08-22.

[5] A. Yu. Kitaev. Quantum measurements and the abelian stabilizer problem, 1995.

[6] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[7] Karthik Srinivasan, Saipriya Satyajit, Bikash K. Behera, and Prasanta K. Panigrahi. Efficient quantum algorithm for solving travelling salesman problem: An ibm quantum experience, 2018.

Appendix A

Code: Calculating the Hamiltonian Cycles and Locating Eigenstates

[1]:
```
from itertools import permutations
import numpy as np

def hamiltonian_cycles(cities, symmetric = False):
    'returns a list of all possible hamiltonian cycles for a given list of cities'
    start = cities[0]
    cycles = []
    for permutation in permutations(cities[1:]):
        cycle = start + ''.join(permutation) + start
        if symmetric:
            if cycle[::-1] not in cycles:
                cycles.append(cycle)
        else:
            cycles.append(cycle)
    return cycles

def map_indices(cities, symmetric = False):
    'returns all the hamiltonian cycles and the indices'
    cycles = hamiltonian_cycles(cities, symmetric = symmetric)
    index_map = {cities[i]: str(range(len(cities))[i]) for i in range(len(cities))}
    indices_cycles = []
    for cycle in cycles:
        indices_city = ''
        for city in cycle[1:]:
            indices_city += index_map[city]
        indices_cycles.append(indices_city)
    return cycles, indices_cycles

def sort_indices(cycles, indices_cycles):
    'sorts the indices'
    results = []
    for cycle, index in zip(cycles, indices_cycles):
        pairs = list(zip(cycle, index))
        sorted_pairs = sorted(pairs, key = lambda pair: pair[0]) # sorts by city name
        sorted_index = ''.join([pair[1] for pair in sorted_pairs])
        results.append([cycle, index, sorted_index])
    return results
```

[2]:
```
cities = ['A', 'B', 'C', 'D', 'E']
cycles, indices_cycles = map_indices(cities, symmetric = True)
cycles, indices_cycles
```

[2]:
```
(['ABCDEA',
  'ABCEDA',
  'ABDCEA',
  'ABDECA',
  'ABECDA',
  'ABEDCA',
  'ACBDEA',
  'ACBEDA',
  'ACDBEA',
  'ACEBDA',
  'ADBCEA',
```

```
 'ADCBEA'],
['12340',
 '12430',
 '13240',
 '13420',
 '14230',
 '14320',
 '21340',
 '21430',
 '23140',
 '24130',
 '31240',
 '32140'])
```

[3]:
```
table = sort_indices(cycles, indices_cycles)
print("cycle, index, sorted_index")
table
```

```
cycle, index, sorted_index
```

[3]:
```
[['ABCDEA', '12340', '12340'],
 ['ABCEDA', '12430', '12403'],
 ['ABDCEA', '13240', '13420'],
 ['ABDECA', '13420', '13042'],
 ['ABECDA', '14230', '14302'],
 ['ABEDCA', '14320', '14023'],
 ['ACBDEA', '21340', '23140'],
 ['ACBEDA', '21430', '24103'],
 ['ACDBEA', '23140', '24310'],
 ['ACEBDA', '24130', '23401'],
 ['ADBCEA', '31240', '32410'],
 ['ADCBEA', '32140', '34120']]
```

[4]:
```
# Base 10 and 2 conversions
table = np.array(table)
indices = table[:,2]
base_10 = np.array([int(indices[i], len(cities)) for i in range(len(indices))])
base_2  = np.array(["{0:b}".format(base_10[i]) for i in range(len(base_10))])
table = np.append(table, base_10.reshape(-1,1), axis=1)
table = np.append(table, base_2.reshape(-1,1), axis=1)
print("cycle, index, sorted_index, base 10, base 2 \n",table)
```

```
cycle, index, sorted_index, base 10, base 2
 [['ABCDEA' '12340' '12340' '970' '1111001010']
 ['ABCEDA' '12430' '12403' '978' '1111010010']
 ['ABDCEA' '13240' '13420' '1110' '10001010110']
 ['ABDECA' '13420' '13042' '1022' '1111111110']
 ['ABECDA' '14230' '14302' '1202' '10010110010']
 ['ABEDCA' '14320' '14023' '1138' '10001110010']
 ['ACBDEA' '21340' '23140' '1670' '11010000110']
 ['ACBEDA' '21430' '24103' '1778' '11011110010']
 ['ACDBEA' '23140' '24310' '1830' '11100100110']
 ['ACEBDA' '24130' '23401' '1726' '11010111110']
 ['ADBCEA' '31240' '32410' '2230' '100010110110']
```

```
['ADCBEA' '32140' '34120' '2410' '100101101010']]
```

Appendix B

Code: 4-City Problem

[1]:
```
import numpy as np
from matplotlib import pyplot as plt
import math

# for circuit construction
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister

# QFT circuit needed for phase estimation
from qiskit.circuit.library import QFT

# for creating custom gates
from qiskit import quantum_info as qi

# for simulation
from qiskit_aer import Aer
from qiskit import transpile
from qiskit.visualization import plot_histogram

# for storing data later
import pandas as pd
```

[2]:
```
#4-city graph edge weights as described in chapter 4.1

w_1 = 4 # a <-> b
w_2 = 2 # a <-> c
w_3 = 4 # a <-> d
w_4 = 4 # b <-> c
w_5 = 5 # c <-> d
w_6 = 6 # b <-> d

weights = []

for i in range(1, 7):
    variable_name = "w_" + str(i)
    current_number = locals()[variable_name]
    weights.append(current_number)

#sorting edge weights
sorted_weights = np.sort(weights)[::-1]

# normalization factor
S = np.sum(sorted_weights[:4])

# epsilon
eps = 1

weights = weights / (S + eps)
weights
```

[2]:
```
array([0.2 , 0.1 , 0.2 , 0.2 , 0.25, 0.3 ])
```

[3]:
```
## solutions
# A->B->C->D->A
```

```
# A->B->D->C->A
# A->C->B->D->A

print('solution 1: {:.2f}'.format(weights[0] + weights[3] + weights[4] + weights[2]))
print('solution 2: {}'.format(weights[0] + weights[5] + weights[4] + weights[1]))
print('solution 3: {}'.format(weights[1] + weights[3] + weights[5] + weights[2]))
print(" ")
print('solutions after max weight removed')
print('solution 1: {:.2f}'.format(weights[0] + weights[3] + weights[4] + weights[2]))
print('solution 2: {}'.format(weights[0]  + weights[4] + weights[1]))
print('solution 3: {}'.format(weights[1] + weights[3] + weights[2]))
```

```
solution 1: 0.85
solution 2: 0.85
solution 3: 0.8

solutions after max weight removed
solution 1: 0.85
solution 2: 0.55
solution 3: 0.5
```

[4]:
```
## Creating CU matrix

m = 8 # eigenvalue qubits
U111 = 1
U122 = np.exp(1j * weights[0] * 2 * np.pi)
U133 = np.exp(1j * weights[1] * 2 * np.pi)
U144 = np.exp(1j * weights[2] * 2 * np.pi)
U1 = np.diag([U111, U122, U133, U144])

U211 = np.exp(1j * weights[0] * 2 * np.pi)
U222 = 1
U233 = np.exp(1j * weights[3] * 2 * np.pi)
U244 = np.exp(1j * weights[5] * 2 * np.pi)
U2 = np.diag([U211, U222, U233, U244])

U311 = np.exp(1j * weights[1] * 2 * np.pi)
U322 = np.exp(1j * weights[3] * 2 * np.pi)
U333 = 1
U344 = np.exp(1j * weights[4] * 2 * np.pi)
U3 = np.diag([U311, U322, U333, U344])

U411 = np.exp(1j * weights[2] * 2 * np.pi)
U422 = np.exp(1j * weights[5] * 2 * np.pi)
U433 = np.exp(1j * weights[4] * 2 * np.pi)
U444 = 1
U4 = np.diag([U411, U422, U433, U444])

U = np.kron(np.kron(np.kron(U1,U2),U3),U4)
print(np.all(np.diag(U) != 0)) # confirming only the diagonal is being used.

Ugate = qi.Operator(U).to_instruction()
Ugate.label = "CU"
CUgate = Ugate.control()
```

True

[5]:
```
## confirming the eigenstates are correct

eigstatelist = ['01101100', #1230 A->B->C->D->A
                '01110010', #1302 A->B->D->C->A
                '10110100'] #2310 A->C->B->D->A

U_angles = np.diag(np.angle(U))/2/np.pi
eiglistint = [int(eigstatelist[i], 2) for i in range(len(eigstatelist))]

# converting from (-pi,pi) to (0,2pi)
sol_check = U_angles[eiglistint] + 1
sol_check
```

[5]:
```
array([0.85, 0.85, 0.8 ])
```

[6]:
```
### creating CU'
## removing edge-weight value 6 (normalized val = 0.3)
max_index = np.where(weights == 0.3)[0][0]
weights[max_index] = 0

U111 = 1
U122 = np.exp(1j * weights[0] * 2 * np.pi)
U133 = np.exp(1j * weights[1] * 2 * np.pi)
U144 = np.exp(1j * weights[2] * 2 * np.pi)
U1 = np.diag([U111, U122, U133, U144])

U211 = np.exp(1j * weights[0] * 2 * np.pi)
U222 = 1
U233 = np.exp(1j * weights[3] * 2 * np.pi)
U244 = np.exp(1j * weights[5] * 2 * np.pi)
U2 = np.diag([U211, U222, U233, U244])

U311 = np.exp(1j * weights[1] * 2 * np.pi)
U322 = np.exp(1j * weights[3] * 2 * np.pi)
U333 = 1
U344 = np.exp(1j * weights[4] * 2 * np.pi)
U3 = np.diag([U311, U322, U333, U344])

U411 = np.exp(1j * weights[2] * 2 * np.pi)
U422 = np.exp(1j * weights[5] * 2 * np.pi)
U433 = np.exp(1j * weights[4] * 2 * np.pi)
U444 = 1
U4 = np.diag([U411, U422, U433, U444])

Up = np.kron(np.kron(np.kron(U1,U2),U3),U4)
print(np.all(np.diag(Up) != 0))# confirming only the diagonal is being used.

UPgate = qi.Operator(Up).to_instruction()
UPgate.label = "CU'"
CUPgate = UPgate.control()
```

True

[7]:
```
U_angles = np.diag(np.angle(Up))/2/np.pi
eiglistint = [int(eigstatelist[i], 2) for i in range(len(eigstatelist))]

# converting from (-pi,pi) to (0,2pi)
sol_check = U_angles[eiglistint] + 1
sol_check[2] -= 1
sol_check
```

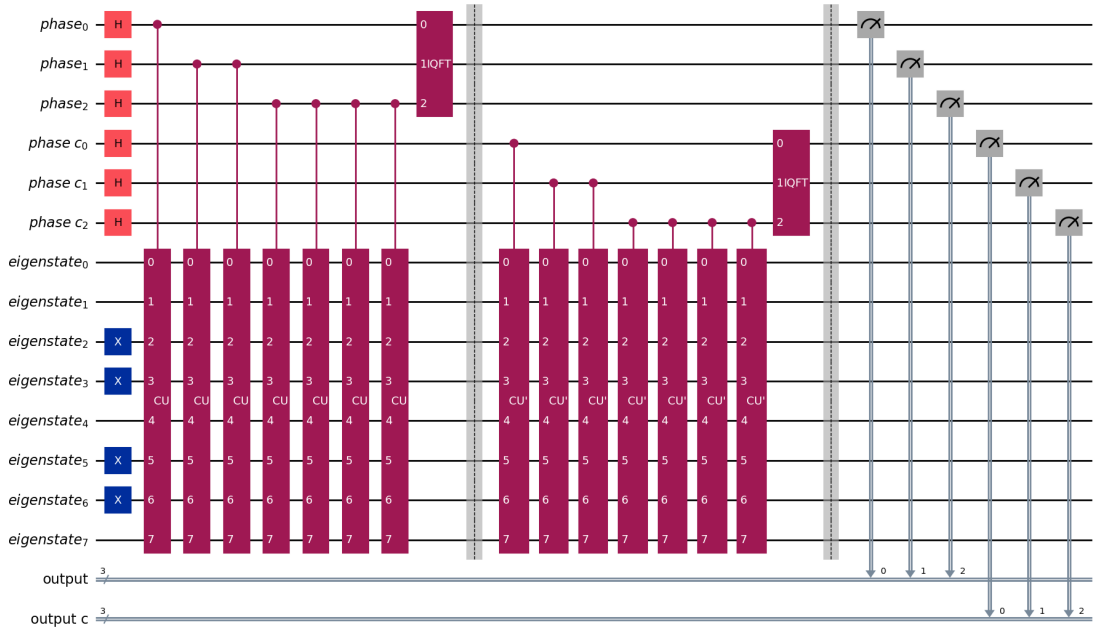
[7]:
```
array([0.85, 0.55, 0.5 ])
```

[8]:
```
def bitstring_converter(string):
    '''
    converts binary values < 1 to decimal
    specifically for the results retrieved
    from simulation
    '''
    values = []
    value = 0
    j = 0
    for i, v in enumerate(string):
        if v == '1':
            value += 1/(2**(i+1-j))
        elif v == " ":
            values.append(value)
            value = 0
            j = i+1
        if i == len(string)-1:
            values.append(value)
    return values[::-1]



def SingleHamiltonianCycle(eig, n):
    # we need a register for the eigenstate:
    eigst   = QuantumRegister(m, name = 'eigenstate')

    # we need two registers for the constrained problem:
    phase   = QuantumRegister(n, name = 'phase')
    phase_c = QuantumRegister(n, name = 'phase c')
    cr      = ClassicalRegister(n, 'output')
    cr_c    = ClassicalRegister(n, 'output c')

    # constructing the circuit (Initialization):
    qc = QuantumCircuit(phase, phase_c,eigst, cr,cr_c)

    # Apply H-Gates to phase qubits:
    for qubit in range(2*n):
        qc.h(qubit)

    for ind, val in enumerate(eig):
        if(int(val)):
            qc.x(ind + 2*n)
```

```
    ## Phase Estimation
    eig_qubits = np.arange(0,m) + 2*n

    repetitions = 1
    for counting_qubit in range(2*n):
        if counting_qubit == n:
            repetitions = 1
            qc.append(QFT(num_qubits = n, inverse = True, do_swaps=True), phase)
            qc.barrier()
        applied_qubits = np.append([counting_qubit], [eig_qubits])
        for i in range(repetitions):
            if counting_qubit < n:
                qc.append(CUgate, list(applied_qubits)); # This is CU
            else:
                qc.append(CUPgate, list(applied_qubits));
        repetitions *= 2

    qc.append(QFT(num_qubits = n, inverse = True, do_swaps=True), phase_c)
    qc.barrier()

    qc.measure(phase,cr)
    qc.measure(phase_c,cr_c)

    return qc
```

[9]:
```
n = 3 ## number of estimation qubits.

## A->B->C->D->A
eig = eigstatelist[0]
eig = eig[::-1] # needs to be reversed (Qiskit convention)
qc1 = SingleHamiltonianCycle(eig, n)
qc1.draw(fold=-1, output='mpl')
```

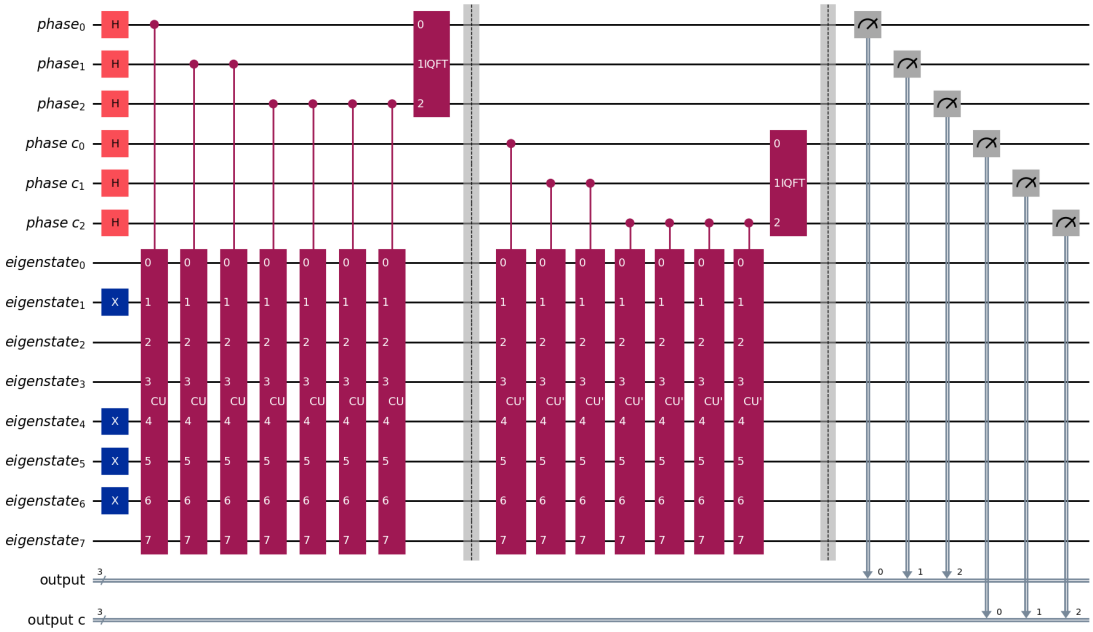
[9]:



[10]:
```
simulator = Aer.get_backend('qasm_simulator')
qc1 = transpile(qc1, simulator)
result = simulator.run(qc1).result()
counts1 = result.get_counts(qc1)
```

[11]:
```
## A->B->D->C->A
eig = eigstatelist[1]
eig = eig[::-1] # needs to be reversed (Qiskit convention)
qc2 = SingleHamiltonianCycle(eig, n)
qc2.draw(fold=-1, output='mpl')
```

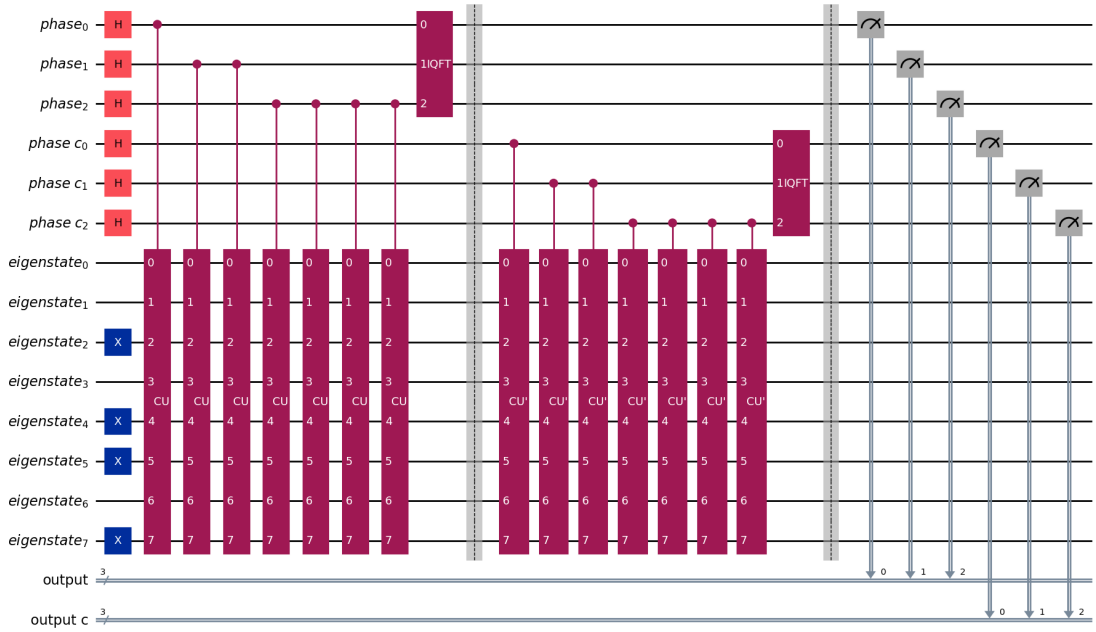
[11]:



[12]:
```
simulator = Aer.get_backend('qasm_simulator')
qc2 = transpile(qc2, simulator)
result = simulator.run(qc2).result()
counts2 = result.get_counts(qc2)
```

[13]:
```
## A->C->B->D->A
eig = eigstatelist[2]
eig = eig[::-1] # needs to be reversed (Qiskit convention)
qc3 = SingleHamiltonianCycle(eig, n)
qc3.draw(fold=-1, output='mpl')
```

[13]:



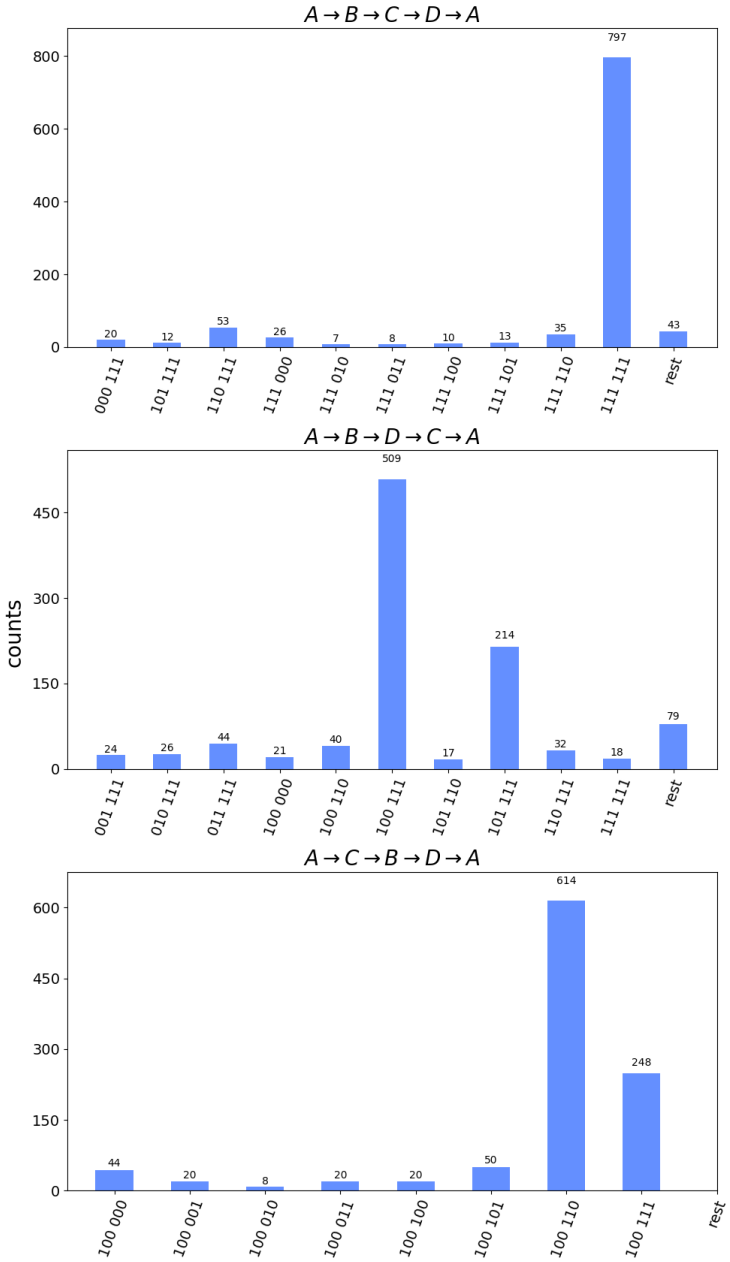
[14]:
```
simulator = Aer.get_backend('qasm_simulator')
qc3 = transpile(qc3, simulator)
result = simulator.run(qc3).result()
counts3 = result.get_counts(qc3)
```

[15]:
```
# quick check
# printing most probable values [before, after] constraint values removed
print('       before  after')
print('cycle 1', bitstring_converter(max(counts1, key=counts1.get)))
print('cycle 2', bitstring_converter(max(counts2, key=counts2.get)))
print('cycle 3', bitstring_converter(max(counts3, key=counts3.get)))
```

```
       before  after
cycle 1 [0.875, 0.875]
cycle 2 [0.875, 0.5]
cycle 3 [0.75, 0.5]
```

[16]:
```
fig, ax = plt.subplots(3, 1, figsize=(10, 17))
fig.subplots_adjust(hspace=10)
fig.supylabel('counts', fontsize=20)
plot_histogram(counts1, number_to_keep = 10, ax= ax[0])
plot_histogram(counts2, number_to_keep = 10, ax = ax[1])
plot_histogram(counts3, number_to_keep = 10, ax = ax[2])

ax[0].set_title('$A \\rightarrow B \\rightarrow C \\rightarrow D \\rightarrow A$',␣
 ↪fontsize = 20)
ax[1].set_title('$A \\rightarrow B \\rightarrow D \\rightarrow C \\rightarrow A$',␣
 ↪fontsize = 20)
```

```
ax[2].set_title('$A \\rightarrow C \\rightarrow B \\rightarrow D \\rightarrow A$',␣
↪fontsize = 20)

ax[0].set_ylabel('')
ax[1].set_ylabel('')
ax[2].set_ylabel('')
ax[2].grid(False)
plt.tight_layout()
plt.savefig('plots/'+ str(n) + 'qubit-4city_other.pdf')
plt.show()
```



Appendix C

Code: 5-city constraint problem

1. this section will include any code used to produce graphs in the simulated section.