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

This thesis presents the development and analysis of a quantum algorithm tailored for solving the decision problem variant of the Bottleneck Travelling Salesman Problem (BTSP). The decision problem asks whether there exists a Hamiltonian cycle such that no edge weight within the cycle is less than a specified value (α). We construct two unitary matrices, both of whose eigenvalues are phases corresponding to the total cost associated with the Hamiltonian cycle. One matrix applies after all edge weights $\geq \alpha$ are set to zero. Utilizing quantum phase estimation, we construct an algorithm to evaluate a Hamiltonian cycle of a graph before and after removing edge weights that do not satisfy the condition. If the total cost remains unaffected, we can conclude it is a solution. The proposed quantum algorithm maintains the inherent computational complexity of the classical BTSP, $O((N-1)!)$, yet introduces the potential for parallelization. We thoroughly examine the algorithm's performance by applying it to both a 4-city and 5-city graph, simulating the quantum operations using Qiskit, an open-source quantum computing framework. Results from the simulations affirm that the algorithm effectively identifies minimized bottleneck paths. Notably, the thesis also delves into the potential scalability of the algorithm, assessing its applicability to larger graphs and more complex datasets. By analyzing various outcomes, we establish insights into the practical implementation of this quantum algorithm.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I want to express my gratitude to Prof. Matthew Choptuik for not only his supervision but also for the encouragement and support to pursue a topic that truly captivates my interest. I would like to thank Erin Bennett and Jason Li, my fellow physics undergraduates, for helping me get through school. I would also like to thank Stefan Grahovac for keeping me motivated and cheering me on throughout my degree. Finally, I would like to thank my family for their support and my sister, Mehak Tejani, for encouraging me to pursue physics.

Chapter 1

Introduction

Continuous advancements in quantum computing have enabled contemporary approaches to solve complex computational problems that were previously considered intractable using classical methods. Well known examples include Shor's [1] and Grover's [2] algorithms 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, with practical applications ranging from optimizing vehicle routes, to circuit design. Thus, finding 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: it has no known solution in polynomial time, yet its 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.

In this work, we explore the intersection of quantum computing and optimization by introducing a quantum algorithm tailored for the BTSP. It requires finding a closed tour through a set of cities while minimizing the largest cost, the "bottleneck", along any route. Its computational complexity grows exponentially with the number of cities, rendering classical solutions impractical for large-scale instances. Our approach is to capitalize on the inherent quantum parallelism, as well as the unique feature of phase encoding in quantum computing. By exploiting these phenomena, we aim to encode and manipulate the costs associated with various routes efficiently.

In the upcoming chapter, we delve into the complexities of the BTSP. We will explore quantum computing fundamentals, discuss quantum phase estimation in depth and provide a detailed explanation of our proposed algorithm, including simulations. Finally, we will evaluate the benefits and drawbacks of our algorithm based on the simulation results.

Chapter 2

Theory

2.1 The Bottleneck Travelling Salesman Problem

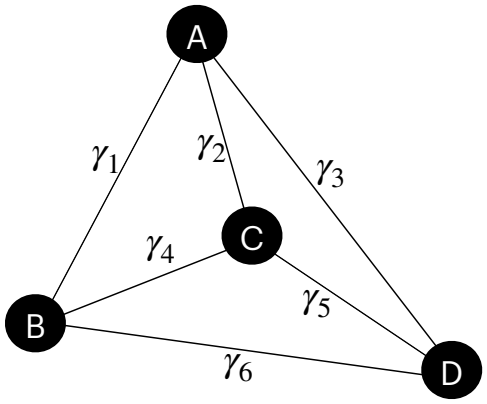


Figure 2.1: An undirected weighted graph represention of a symmetric 4-city system. The vertices represent cities and the edge weights represent the cost of travel.

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. The graph also includes edge weights, which we define as γ_i. The goal of the BTSP is to find a Hamiltonian cycle in a graph where the largest edge weight (bottleneck) is minimized. This is distinct from the Travelling Salesman Problem (TSP) where the total cost of a given cycle is minimized. The total number of Hamiltonian cycles is given by $(N-1)!$, where N is the number of nodes. We present a symmetric case in Fig. 2.1, which implies:

$$N_k \to N_{k+1} = N_{k+1} \to N_k = \gamma_i$$

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. Moreover, BTSP solutions 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 evaluate the total costs of these cycles as follows. The first cycle is the solution to the 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 changing the weight of γ_6 to 5, we illustrate that all cycles can become 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, which implies that no polynomial-time algorithm exists for a solution. A brute-force approach would require that we can run an algorithm in $O((N-1)!)$ time.

2.2 An Introduction to Quantum Computing

Quantum computing represents a fundamentally different approach to computation compared to its classical counterpart. At its core, quantum computing leverages principles of quantum physics to process information in ways that classical computers can't. Let us look at some key differences:

1. Information Representation: Classical bits are binary and can only be in one state at a time (0 or 1). In contrast, qubits can be in a superposition of the basis states.

2. Determinism: Classical computing is deterministic, meaning the output of a computation is predictable if you know the input and the algorithm. Quantum computing, however, is probabilistic; only after measurement do we achieve a deterministic reading. This means that quantum algorithms can produce different outcomes with each run with respect to their probabilities.

3. Operations: Classical computers perform operations using logical gates (`AND`, `OR`, `NOT`), whereas quantum computers use unitary operators to define quantum logic gates. Unitary operators satisfy the condition: $U^\dagger U = \mathbb{I}$, implying that all quantum operations are reversible, which is not generally the case in classical computation.

2.2.1 Frequently Used Notation

Let us begin by examining single qubit states, which are conventionally represented in the computational basis.

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

Qubits can exist in a superposition of these basis states:

$$|s\rangle = \alpha|0\rangle + \beta|1\rangle$$

Here, $|\alpha|^2$ represents the probability of measuring the state $|0\rangle$, and $|\beta|^2$ represents the probability of measuring the state $|1\rangle$. Consequently, the amplitudes must satisfy the condition:

$$|\alpha|^2 + |\beta|^2 = 1$$

Next, we examine the Pauli-X gate, whose effect on single qubit states is analogous to the `NOT` gate in classical computation.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$X|0\rangle = |1\rangle$$
$$X|1\rangle = |0\rangle$$

Below, we examine the Hadamard gate and its effects when applied to single qubit states, which is frequently used to establish a uniform superposition.

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

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

Multiple qubit states are created through the tensor product of single states. An n-qubit system spans a Hilbert space of dimension $N = 2^n$. For example, a 2-qubit system comprises four states.

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

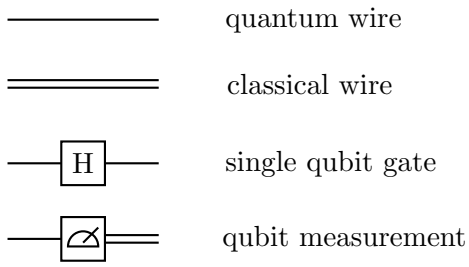
$$|00\rangle = |1\rangle \otimes |0\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, |01\rangle = |1\rangle \otimes |1\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

The Hadamard gate can also be extended to multiple qubit states. Let us see it applied to $|00\rangle$:

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

2.2.2 Quantum Circuit Components

We can represent unitary operations on qubits using quantum circuit diagrams, similar to classical circuits. Below are some frequently used circuit components:



quantum wire

classical wire

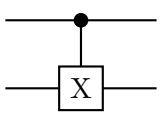
single qubit gate

qubit measurement

2.2.3 Controlled Gates

A controlled gate operates on two or more qubits, where one or more qubits act as the "control" qubits, and the others as the "target" qubits. For the context of this thesis, we will discuss controlled operations involving a single control qubit.

Below, we present the quantum circuit and matrix representation of the controlled-X gate, along with its effect on all 2-qubit states:



control qubit

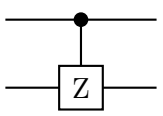
target qubit

$$CX = \begin{bmatrix} \mathbb{I}_2 & 0 \\ 0 & X \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\begin{aligned} X|00\rangle &= |00\rangle \\ X|01\rangle &= |01\rangle \\ X|10\rangle &= |11\rangle \\ X|11\rangle &= |10\rangle \end{aligned}$$

In this operation, only the states $|10\rangle$ and $|11\rangle$ undergo any change. The first qubit, positioned on the left, acts as the control qubit, which determines whether the operation is applied. The second qubit, positioned on the right, is the target qubit and is affected only if the control qubit is in the state $|1\rangle$.

Let us now explore the controlled-Z gate. We will examine the quantum circuit and matrix representation, as well as its effect on all 2-qubit states.



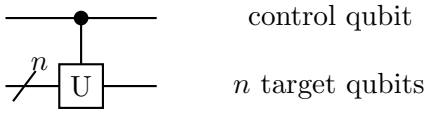
control qubit

target qubit

$$CZ = \begin{bmatrix} \mathbb{I}_2 & 0 \\ 0 & Z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

$$\begin{aligned} Z|00\rangle &= |00\rangle \\ Z|01\rangle &= |01\rangle \\ Z|10\rangle &= |10\rangle \\ Z|11\rangle &= -|11\rangle \end{aligned}$$

In this case, only the final state, $|11\rangle$ was affected, specifically altering the state's phase. We can construct customized controlled gates using any unitary matrix U. A controlled-U gate, designed to act on n target qubits, can be represented using block matrix notation. In this context, $\mathbb{I}_n$ denotes the identity matrix which contains 2^n diagonal elements:



$$CU = \begin{bmatrix} \mathbb{I}_n & 0 \\ 0 & U \end{bmatrix} \tag{2.1}$$

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 us 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:

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

$$F_N|x\rangle = \frac{1}{\sqrt{2^n}} \bigotimes_{j=1}^{n} (|0\rangle + e^{2\pi i x 2^{-j}} |1\rangle) \tag{2.2}$$

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

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

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 us 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:

$$U|\lambda\rangle = e^{2\pi i\phi}|\lambda\rangle \tag{2.3}$$

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 $|\lambda\rangle$.

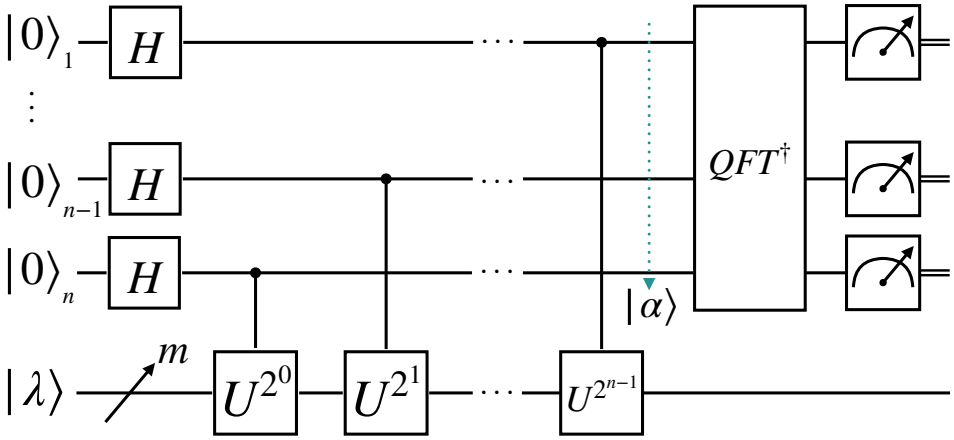


Figure 2.2: A quantum circuit representation of the phase estimation algorithm. Given $U|\lambda\rangle = e^{2\pi i\phi}|\lambda\rangle$, this algorithm allows us to generate an approximation for $\phi \in [0, 1)$. The circuit consists of two registers of qubits. The first n-qubits are initialized to $|0\rangle$ 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.

Let us 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 (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:

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

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 that the eigenstate after the CU operation is left unchanged. The phase has been encoded into the control qubit instead, a result that is due to the phase kickback. Thus, we can reuse our eigenstate for the next qubit. Consecutive qubits have double the number of CU operators as the previous, thus squaring the eigenvalue each time:

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

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:

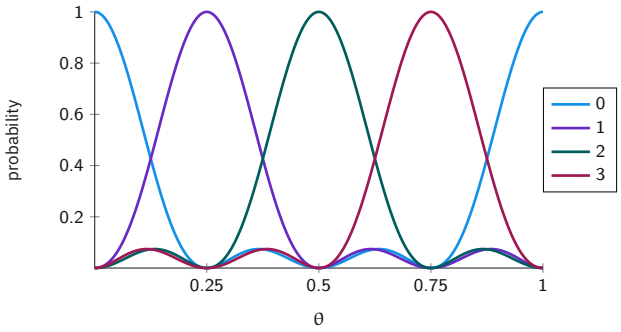
$$\begin{aligned} \mathrm{qubit}_{\mathrm{n}} &: |0\rangle + e^{2\pi i(0.\phi_1\phi_2...\phi_n)}|1\rangle \\ \mathrm{qubit}_{\mathrm{n\text{-}1}} &: |0\rangle + e^{2\pi i(0.\phi_2...\phi_n)}|1\rangle \\ &\ldots \\ \mathrm{qubit}_1 &: |0\rangle + e^{2\pi i(0.\phi_n)}|1\rangle \end{aligned}$$

After all the CU operations, we look at the form of the first register in the state $|\alpha\rangle$ and it will resemble the result of performing the QFT we saw in Equation (2.2). Here, our ω_j are:

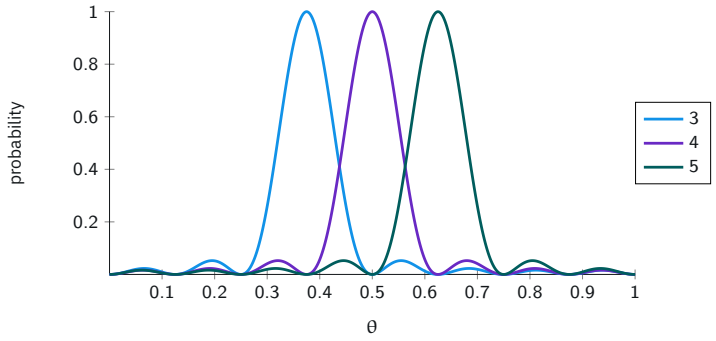
$$\begin{aligned} \omega_1 &= e^{2\pi i x 2^{-1}} = e^{2\pi i(0.\phi_n)} \\ \omega_2 &= e^{2\pi i x 2^{-2}} = e^{2\pi i(0.\phi_{n-1}\phi_n)} \\ &\ldots \\ \omega_n &= e^{2\pi i x 2^{-n}} = e^{2\pi i(0.\phi_1...\phi_n)} \end{aligned}$$

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 the computational cost as we double our use of CU gates for each additional qubit.

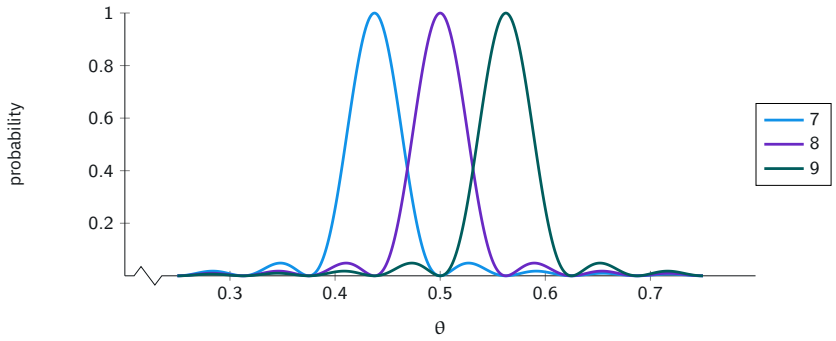
It is possible that ϕ is not a discrete value that can be exactly represented by n qubits, i.e., $|\phi\rangle \neq |\phi_1\phi_2...\phi_n\rangle$. Fortunately, this algorithm still provides a good approximation, with the best outcome occurring with a probability of at least $4/\pi^2 \approx 40\%$. Should the approximation deviate by more than 2^{-n}, the probability of measuring this less favorable outcome is at most 25% [7]. This concept is further illustrated in Fig. 2.3.



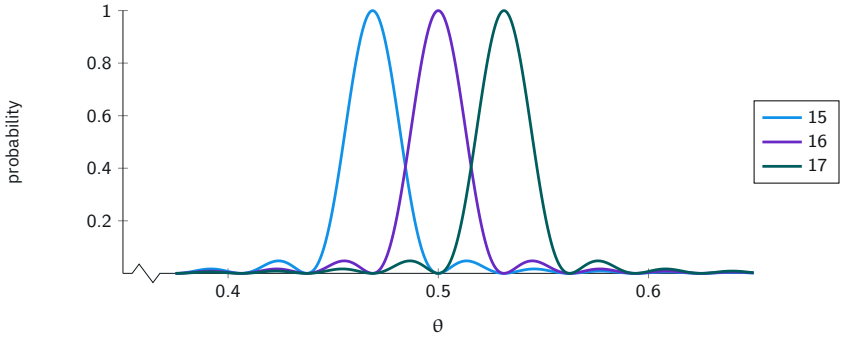
(a) 2-qubit estimation



(b) 3-qubit estimation



(c) 4-qubit estimation



(d) 5-qubit estimation

Figure 2.3: Varying degree of precision by the phase estimation algorithm as a function of the number of qubits (2, 3, 4, and 5). [7].

Chapter 3

Algorithm

The decision problem of the BTSP seeks to determine if there exists a Hamiltonian cycle where the weight of each 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:

$$\gamma_i < \alpha \tag{3.1}$$

We will construct the algorithm in the following steps:

1. Normalize edge weights to ensure that no single Hamiltonian cycle's total weight is greater than or equal to 1, allowing for effective use of the phase estimation algorithm.

2. Construct a unitary operator that encodes information about the Hamiltonian cycles, representing these as phases along its diagonal. Our approach aligns with the findings presented by Ramakrishnan, Sharma, and Punnen [8].

3. Set all edge weights $\geq \alpha$ to zero and construct a secondary unitary operator similar to the one described in 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, using the same eigenstate and controlled gates, to evaluate the Hamiltonian cycle both before and after setting edge weights $\geq \alpha$ to zero.

7. Compare the two resulting phases; if they match, the Hamiltonian cycle under examination satisfies the condition and is a viable solution.

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 of a total of $N(N-1)$ edge weights in a directed graph. We can choose the largest N and use these to normalize the edge weights. Let w represent our edge-weights, which 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:

$$w' = \{w'_1, w'_2, \ldots, w'_m\} \tag{3.2}$$

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:

$$S = \sum_{i=1}^{N} w'_i \tag{3.3}$$

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

$$\tilde{w} = (S + \epsilon)^{-1} w \tag{3.4}$$

Where:

$\epsilon > 0$

The purpose of ϵ is to ensure that if any normalized Hamiltonian cycle's weight is exactly equal to S, it does not result in a zero phase.

3.2 Unitary Operator and Eigenstates associated with the Hamiltonian Cycles

We begin by constructing diagonal matrices U_j, one for each node in a complete graph, and describing the elements of these matrices:

$$[U_j]_{kk} = e^{2\pi i \gamma_{jk}(1-\delta_{jk})} \tag{3.5}$$

Where:

$1 \leq j, k \leq N$

N denotes the total number of nodes.

γ_{jk} represents the edgeweight connecting node $j \to k$.

Next, we construct U, which is the tensor product of all the diagonal matrices:

$$U = \bigotimes_{j}^{N} U_j \tag{3.6}$$

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

Given U's diagonal nature, its eigenstates align with the basis computational states. We specifically focus on eigenstates that correspond to Hamiltonian cycles, which are determined by

their phases. To understand how the diagonal elements of U are derived from the individual U_j matrices, we examine the tensor product construction. This process fills the diagonal elements of U, thus simplifying our approach by allowing us to directly consider these elements.

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

This pattern indicates:

$$[U]_k = [U_1]_{\alpha_{N-1}} \cdot [U_2]_{\alpha_{N-2}} \cdot \ldots \cdot [U_{N-2}]_{\alpha_2} \cdot [U_2]_{\alpha_1} \cdot [U_N]_{\alpha_0} \tag{3.7}$$

Where:

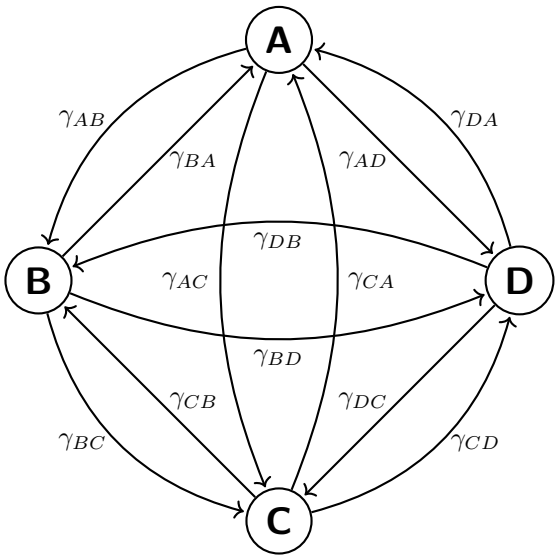
$\alpha_i = \left(k // \left(N^i\right)\right) \% N$

// denotes integer division

% is the modulus operation

We observe that this pattern follows base-N counting, where N is the total number of nodes. Converting k into base-N yields the indices of our original diagonal matrices. To identify the relevant eigenstates, we first locate the elements in U_j associated with a Hamiltonian cycle. Then, we retrieve their respective indices and convert this sequence from base-N to k. This process allows us to identify our eigenstate $|k\rangle$.

3.2.1 The Four City Graph



For the 4-city graph, we will begin with four matrices to represent the 12 edge weights. Utilizing Equation (3.5), 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 let us construct the other diagonals:

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

Next, we construct the tensor product with Equation (3.6):

$$U = A \otimes B \otimes C \otimes D \tag{3.8}$$

For the sake of convenience in dealing only with the diagonals, we can similarly state $u = \text{diag}(U)$, thus:

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

Referring to Equation (3.7) concerning the matrix elements for U, the diagonal elements for the 4-city graph will simplify to:

$$u_k = a_{\alpha_3} \cdot b_{\alpha_2} \cdot c_{\alpha_1} \cdot d_{\alpha_0} \tag{3.9}$$

Where:

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

Let us proceed with identifying the eigenstate corresponding to a specific Hamiltonian cycle. Let us consider the cycle represented by the following path:

$$A \to D \to B \to C \to A \tag{3.10}$$

From this point, we can identify the edge weights that are relevant to us:

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

Therefore, the phase we aim to estimate would be determined 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 (3.9):

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

From here, we simply read out the indices. Based on our discussion under Equation (3.7), we can infer that 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 Hamiltonian cycle (3.10) 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 Table 3.2.

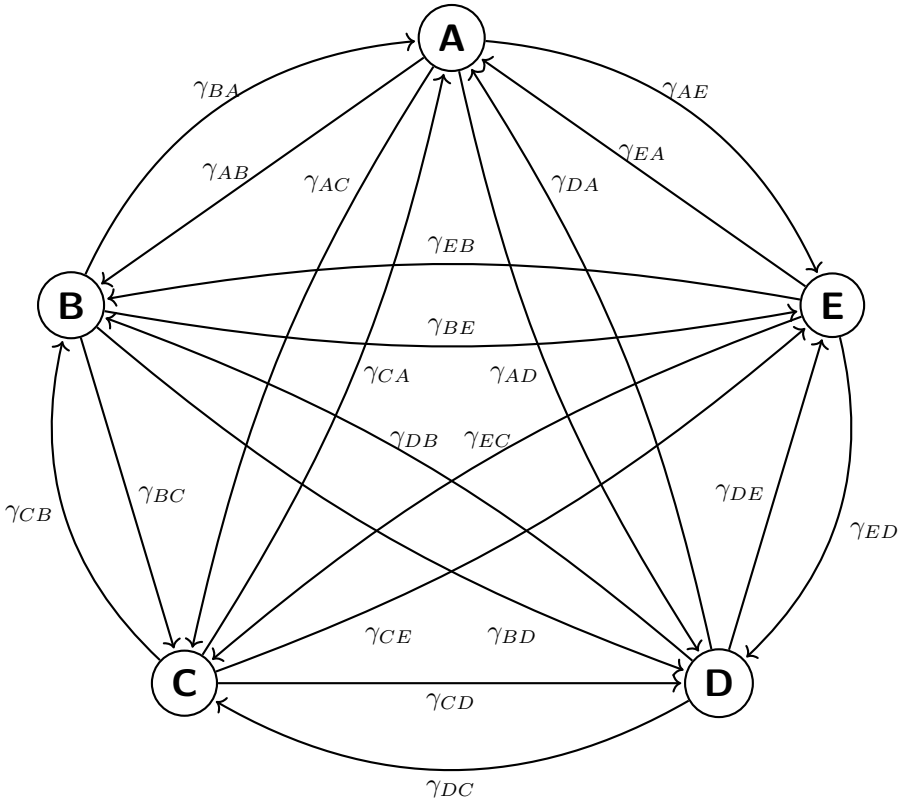
Hamiltonian Cycle	**Total Cost**	**Diagonal Elements Product**
$A \to B \to C \to D \to A$	$\gamma_{AB} + \gamma_{BC} + \gamma_{CD} + \gamma_{DA}$	$a_1 \cdot b_2 \cdot c_3 \cdot d_0$
$A \to B \to D \to C \to A$	$\gamma_{AB} + \gamma_{BD} + \gamma_{DC} + \gamma_{CA}$	$a_1 \cdot b_3 \cdot d_2 \cdot c_0$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

Table 3.1: Hamiltonian cycles of the directed 4-city graph with their total cost and diagonal element products (3.9).

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 (3.8), containing the total cost of a Hamiltonian cycle after normalization of the directed 4-city graph.

3.2.2 The Five City Graph



For the 5-city graph, we will begin with five matrices to represent the 20 edge weights. Utilizing Equation (3.5), where $j, k \in \{A, B, C, D, E\}$, we can construct matrix A:

$$U = A \otimes B \otimes C \otimes D \otimes E \tag{3.11}$$

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

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

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

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

Referring to Equation (3.7) concerning the matrix elements for U, the diagonal elements for the 5-city graph will simplify to:

$$u_k = a_{\alpha_4} \cdot b_{\alpha_3} \cdot c_{\alpha_2} \cdot d_{\alpha_1} \cdot e_{\alpha_0} \tag{3.12}$$

Where:

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

Let us proceed with identifying the eigenstate corresponding to a specific Hamiltonian cycle. Let us consider the cycle represented by the following path:

$$A \to E \to D \to C \to B \to A \tag{3.13}$$

From this point, we can identify the edge weights that are relevant to us:

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

Therefore, the phase we aim to estimate would be determined 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 (3.12):

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

From here, we simply read out the indices. Based on our discussion under Equation (3.7), we can infer that we are working in base 5. 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 Hamiltonian cycle (3.13) 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 Table 3.4.

Hamiltonian Cycle	**Total Cost**	**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 5-city graph with their total cost and diagonal element products (3.12).

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

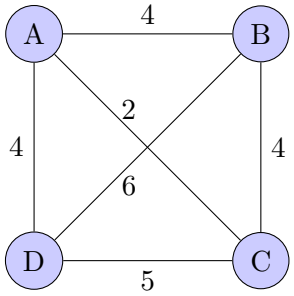
Table 3.4: Eigenstates of matrix U (3.11), containing the total cost of a Hamiltonian cycle after normalization of the directed 5-city graph.

Chapter 4

Simulations

4.1 An Undirected 4-City Graph

Let us consider the following example of a symmetric 4-city system. In this undirected graph, we need to look at $(4-1)!/2 = 3$ Hamiltonian cycles. The condition for our BTSP in this case will involve finding a Hamiltonian cycle where all edge weights are < 6:



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

4.1.1 Algorithm Construction

We will follow the instructions highlighted at the beginning of Chapter 3. Step 1 requires normalizing the edge weights. This involves sorting the edge weights in descending order, as specified in Equation (3.2):

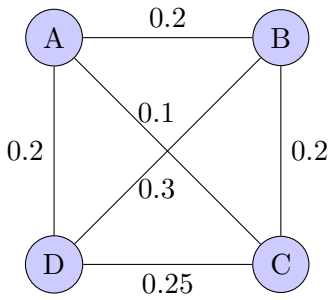
$$w' = \{6, 5, 4, 4, 4, 2\}$$

Next, we retrieve the sum S, as specified in Equation (3.3):

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

We proceed to normalize our edge weights as described in Equation (3.4), where we 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 move on to constructing the unitary operator and finding its corresponding eigenstates. The diagonals of matrices U and U' are as follows:

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

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

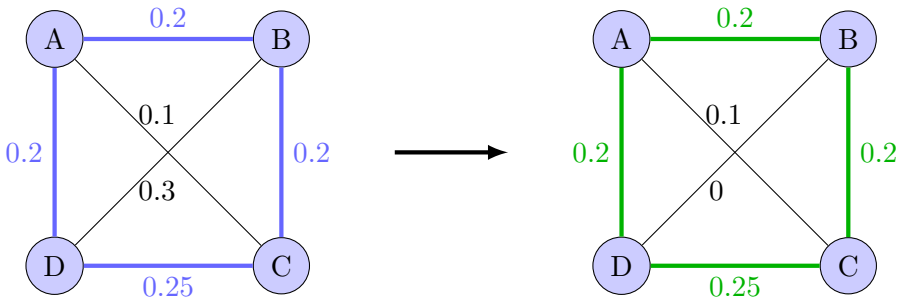
To create our controlled matrices, we employ the block matrix structure illustrated in 2.1. Since both U and U' have 4^4 diagonal elements, we require 8 eigenstate qubits to represent all 256 states.

$$CU = \begin{bmatrix} \mathbb{I}_8 & 0 \\ 0 & U \end{bmatrix}, \quad CU' = \begin{bmatrix} \mathbb{I}_8 & 0 \\ 0 & U' \end{bmatrix}$$

Given our problem is symmetric, we can utilize the results and conversions provided in Table 3.1 for the first three cycles. Hence, we will estimate phases using the following eigenstates:

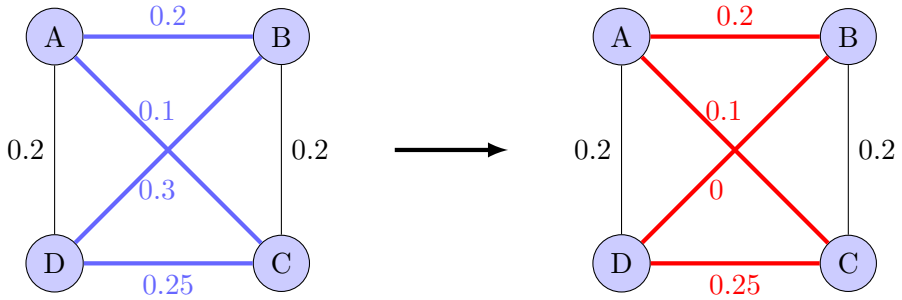
$$|108\rangle = |01101100\rangle$$
$$|114\rangle = |01110010\rangle$$
$$|180\rangle = |10110100\rangle$$

And we can anticipate the following phases:
Cycle 1: $A \to B \to C \to D \to A$



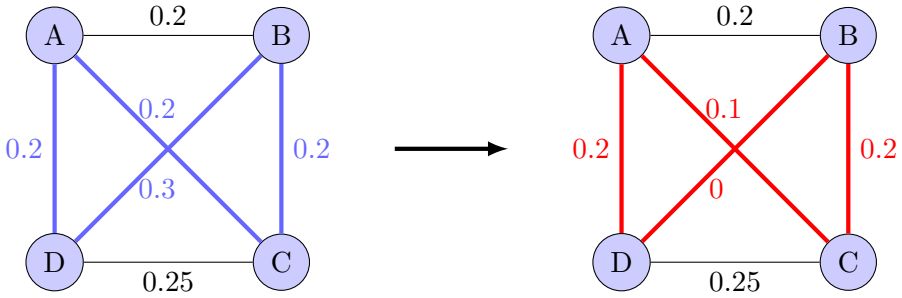
$$u_{108} = e^{i2\pi(\gamma_{AB}+\gamma_{BC}+\gamma_{CD}+\gamma_{DA})} = e^{i2\pi(0.85)}$$
$$u'_{108} = e^{i2\pi(\gamma_{AB}+\gamma_{BC}+\gamma_{CD}+\gamma_{DA})} = e^{i2\pi(0.85)}$$

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

We use Qiskit, an open-source quantum computing framework, for running simulations. Fig. 4.1 illustrates the circuit for our first Hamiltonian cycle initialized in the eigenstate $|01101100\rangle$. Similar circuits will be executed for the other two Hamiltonian cycles, with the only variation being the eigenstate initialization. Our simulations are ideal, implying that our results are not affected by noise. Each circuit is executed 1024 times by default, providing us with a quasi-probability distribution.

We observe in Fig. 4.2 that our 3-qubit phase estimation for the first cycle with the highest counts yields 111 111. The binary digits on the left correspond to the phase estimated after our condition is satisfied, while those on the right represent the estimation before. These binary digits can be converted to their decimal representation:

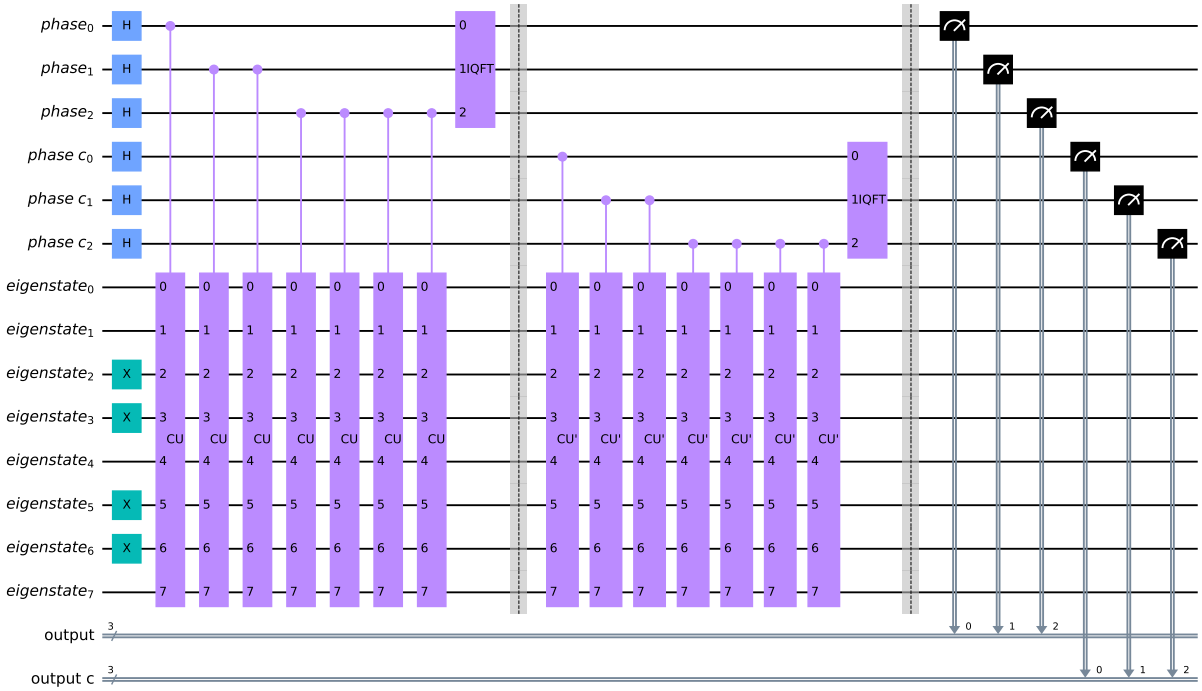


Figure 4.1: The quantum circuit for the 4-city graph involves a 3-qubit phase estimation, measuring the Hamiltonian cycle $A \to B \to C \to D \to A$. The corresponding eigenstate is $|108\rangle = |01101100\rangle$, but due to Qiskit's convention on qubit ordering, the eigenstate is initialized in reverse. The CU gate represents the control unitary matrix containing all the Hamiltonian cycles. The CU' gate includes the same cycles, but before its construction, all edge weights not satisfying the condition (< 6) were set to zero. We have two sets of 3 qubits to be measured and stored in classical registers labelled 'output' and 'output c'.

$$\text{Cycle 1}: \quad 0.111,\ 0.111 = 0.875,\ 0.875$$

We can perform a similar analysis for the other two cycles:

$$\text{Cycle 2}: \quad 0.100,\ 0.111 = 0.500,\ 0.875$$

$$\text{Cycle 3}: \quad 0.110,\ 0.111 = 0.875,\ 0.750$$

For each cycle, we can examine the two states with the highest counts and summarize them in Table 4.1. This table also presents higher-order qubit estimations, namely 4 and 5.

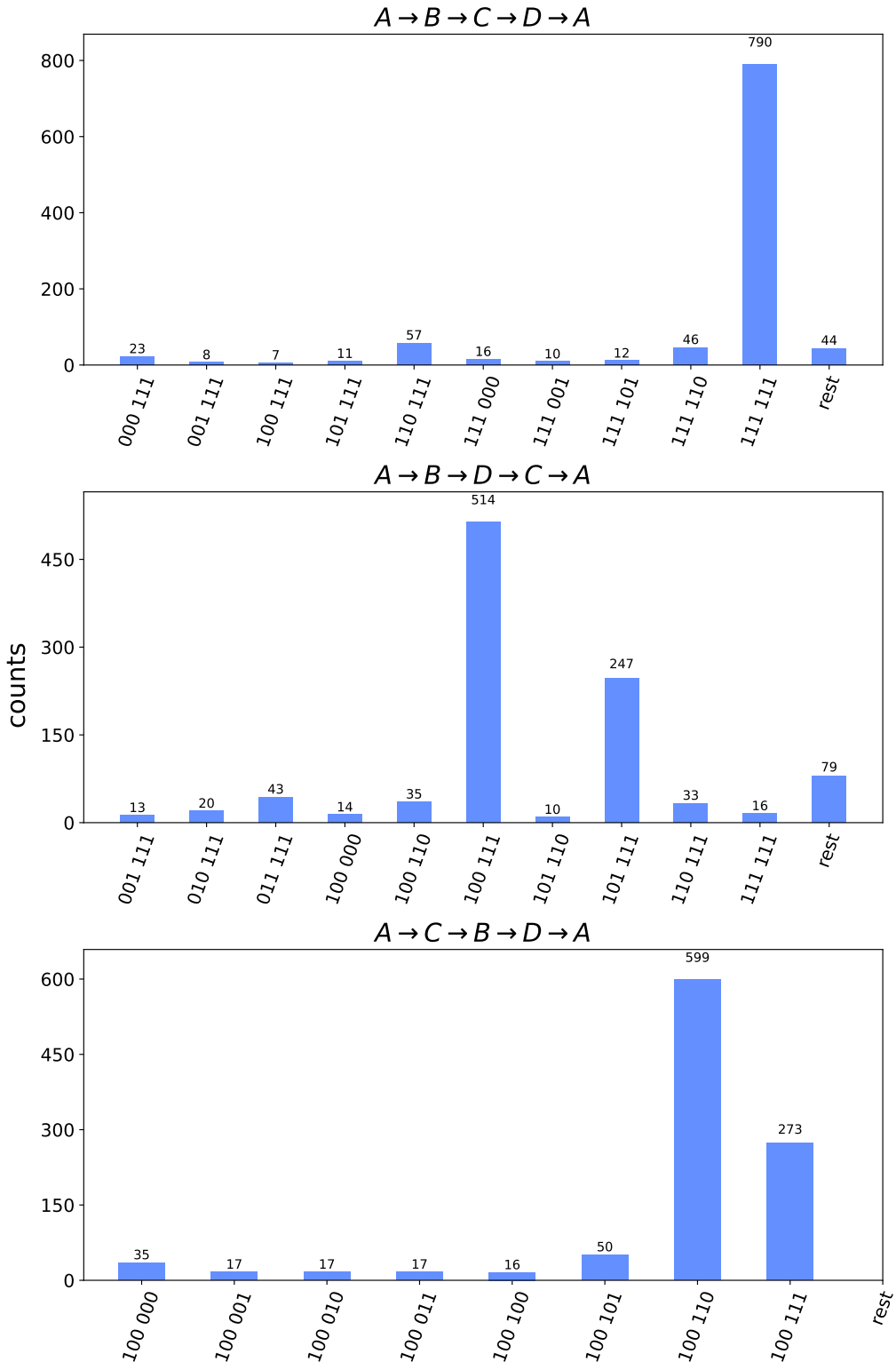


Figure 4.2: 3-qubit phase estimation for the 4-city graph

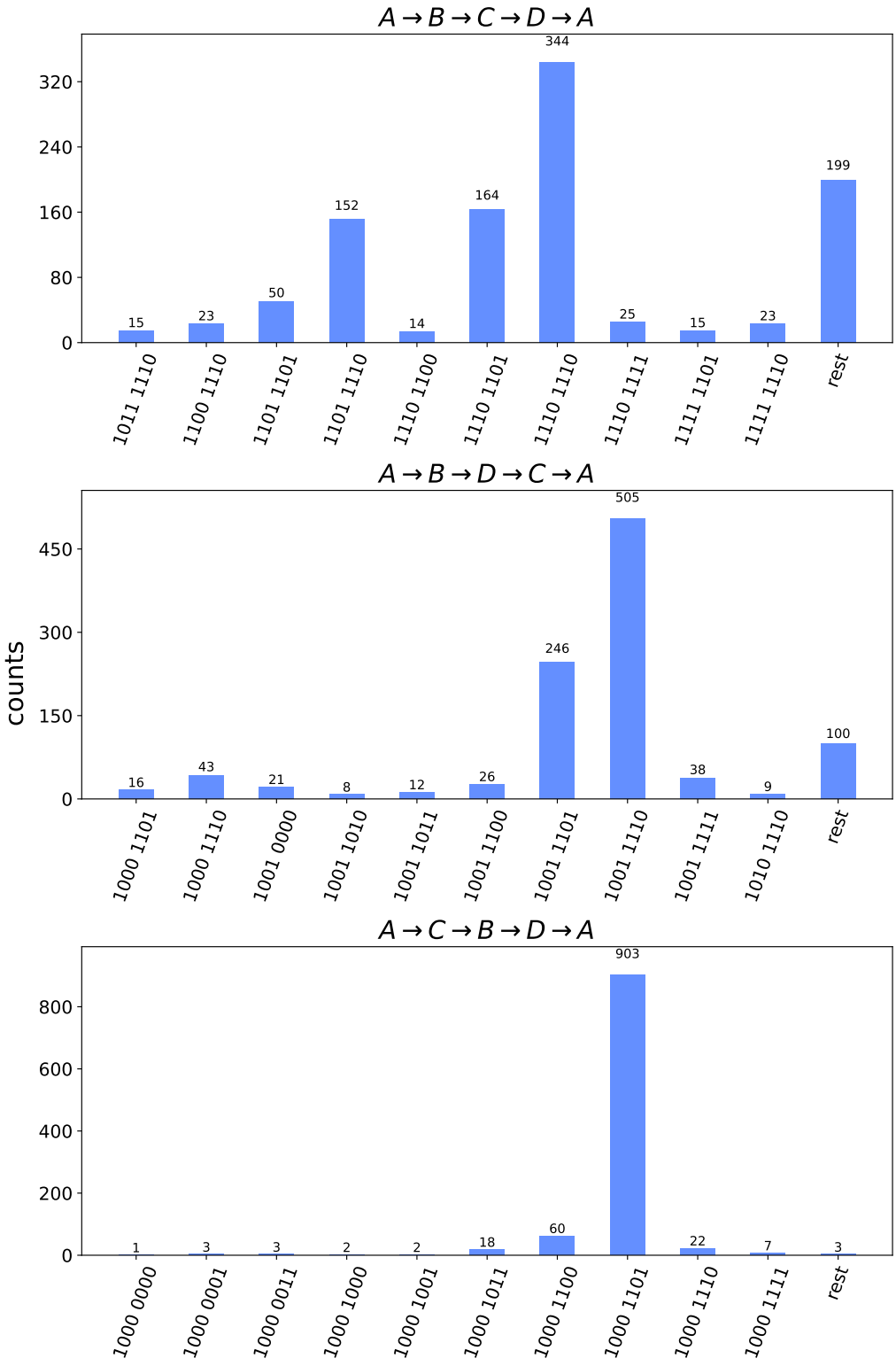


Figure 4.3: 4-qubit phase estimation for the 4-city graph

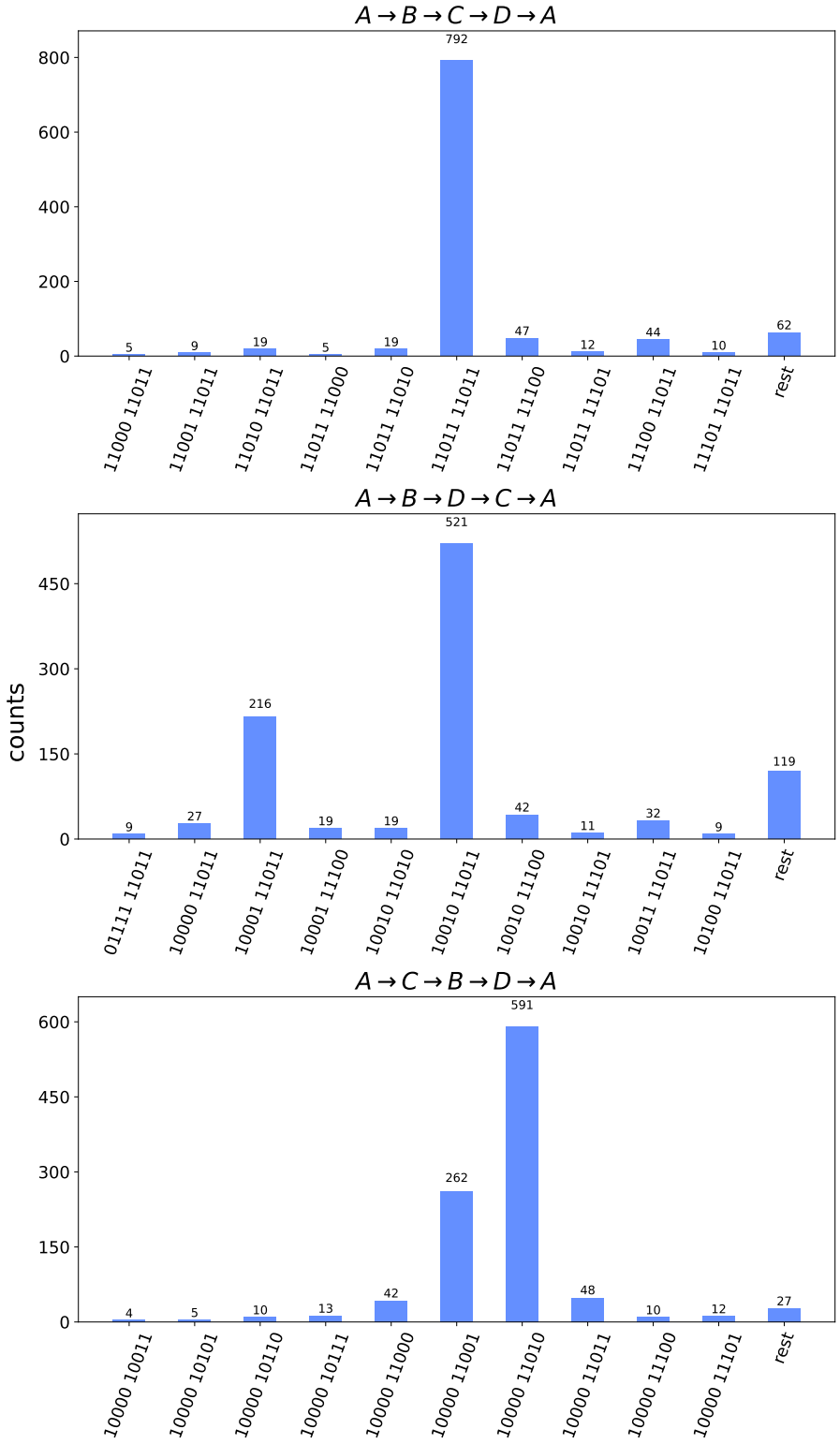


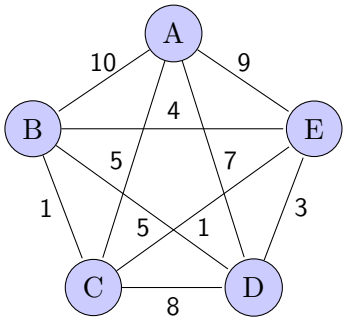
Figure 4.4: 5-qubit phase estimation for the 4-city graph

Cycle	Expected	Phase Qubits	Highest Counts	Prob.	2nd Highest Counts	Prob.
1	0.85, 0.85	3	0.875, 0.875	77%	0.75, 0.875	6%
		4	0.875, 0.875	34%	0.875, 0.8125	16%
		5	0.84375, 0.84375	77%	0.84375, 0.875	5%
2	0.55, 0.85	3	0.5, 0.875	50%	0.625, 0.875	24%
		4	0.5625, 0.875	49%	0.5625, 0.8125	24%
		5	0.5625, 0.84375	51%	0.53125, 0.84375	21%
3	0.50, 0.80	3	0.5, 0.75	58%	0.5, 0.875	27%
		4	0.5, 0.8125	88%	0.5, 0.75	6%
		5	0.5, 0.8125	58%	0.5, 0.78125	26%

Table 4.1: Simulation results for the 4-city graph.

4.2 An Undirected 5-City Graph

Let us consider the following example of a symmetric 5-city system. In this undirected graph, we need to look at $(5-1)!/2 = 12$ Hamiltonian cycles. The condition for our BTSP in this case will involve finding a Hamiltonian cycle where all edge weights are < 9:



4.2.1 Algorithm Construction

We will follow the instructions highlighted at the beginning of Chapter 3. Step 1 requires normalizing the edge weights. This involves sorting the edge weights in descending order, as specified in Equation (3.2):

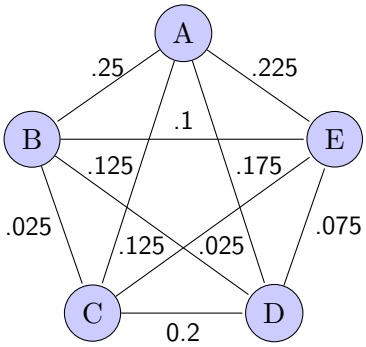
$$w' = \{10, 9, 8, 7, 5, 5, 4, 3, 1, 1\}$$

Next, we retrieve the sum S, as specified in Equation (3.3):

$$S = \sum_{i=1}^{5} w'_i = 10 + 9 + 8 + 7 + 5 = 39$$

We proceed to normalize our edge weights as described in Equation (3.4), where we set $\epsilon = 1$.

$$\tilde{w} = \frac{\{10, 9, 8, 7, 5, 5, 4, 3, 1, 1\}}{40} = \{0.25, 0.225, 0.2, 0.175, 0.125, 0.125, 0.1, 0.075, 0.025, 0.025\}$$



Now, we move on to constructing the unitary operator and finding its corresponding eigenstates. The diagonals of matrices U and U' are as follows:

$$u = \begin{bmatrix} 1 \\ e^{i2\pi(0.25)} \\ e^{i2\pi(0.125)} \\ e^{i2\pi(0.175)} \\ e^{i2\pi(0.225)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.25)} \\ 1 \\ e^{i2\pi(0.025)} \\ e^{i2\pi(0.125)} \\ e^{i2\pi(0.1)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.125)} \\ e^{i2\pi(0.025)} \\ 1 \\ e^{i2\pi(0.2)} \\ e^{i2\pi(0.025)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.175)} \\ e^{i2\pi(0.125)} \\ e^{i2\pi(0.2)} \\ 1 \\ e^{i2\pi(0.075)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.225)} \\ e^{i2\pi(0.1)} \\ e^{i2\pi(0.025)} \\ e^{i2\pi(0.075)} \\ 1 \end{bmatrix}$$

$$u' = \begin{bmatrix} 1 \\ e^{i2\pi(0)} \\ e^{i2\pi(0.125)} \\ e^{i2\pi(0.175)} \\ e^{i2\pi(0)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0)} \\ 1 \\ e^{i2\pi(0.025)} \\ e^{i2\pi(0.125)} \\ e^{i2\pi(0.1)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.125)} \\ e^{i2\pi(0.025)} \\ 1 \\ e^{i2\pi(0.2)} \\ e^{i2\pi(0.025)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0.175)} \\ e^{i2\pi(0.125)} \\ e^{i2\pi(0.2)} \\ 1 \\ e^{i2\pi(0.075)} \end{bmatrix} \otimes \begin{bmatrix} e^{i2\pi(0)} \\ e^{i2\pi(0.1)} \\ e^{i2\pi(0.025)} \\ e^{i2\pi(0.075)} \\ 1 \end{bmatrix}$$

The number of diagonal elements in U and U' is $5^5 = 3125$, thus we need at least 12 eigenstate qubits to represent all 3125 states. We will have left over states ($2^{12} - 5^5 = 971$). To accomodate this issue we need to add 1's to the end of the diagonal of matrices U and U'. With this method, we do not affect the eigenstates.

$$\text{diag}(U) = [u, \text{ones}(971)], \quad \text{diag}(U') = [u', \text{ones}(971)] \tag{4.1}$$

Now we can create our controlled matrices. We employ the block matrix structure illustrated in 2.1. We require 12 eigenstate qubits to represent all 256 states.

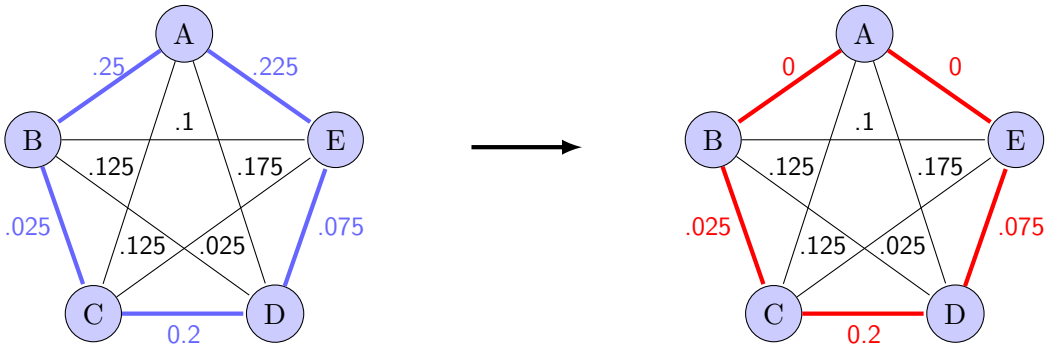
$$CU = \begin{bmatrix} \mathbb{I}_{12} & 0 \\ 0 & U \end{bmatrix}, \quad CU' = \begin{bmatrix} \mathbb{I}_{12} & 0 \\ 0 & U' \end{bmatrix}$$

Given our problem is symmetric, we can utilize half of the results and conversions provided in Table 3.3. Hence, we will estimate phases using the following eigenstates:

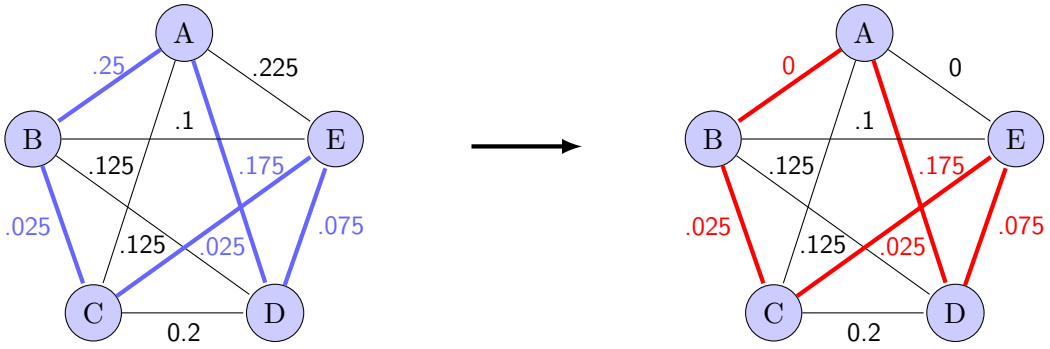
$$|970\rangle = |001111001010\rangle$$
$$|978\rangle = |001111010010\rangle$$
$$|1110\rangle = |010001010110\rangle$$
$$|1022\rangle = |001111111110\rangle$$
$$|1202\rangle = |010010110010\rangle$$
$$|1138\rangle = |010001110010\rangle$$
$$|1670\rangle = |011010000110\rangle$$
$$|1778\rangle = |011011110010\rangle$$
$$|1830\rangle = |011100100110\rangle$$
$$|1726\rangle = |011010111110\rangle$$
$$|2230\rangle = |100010110110\rangle$$
$$|2410\rangle = |100101101010\rangle$$

And we can anticipate the following phases:

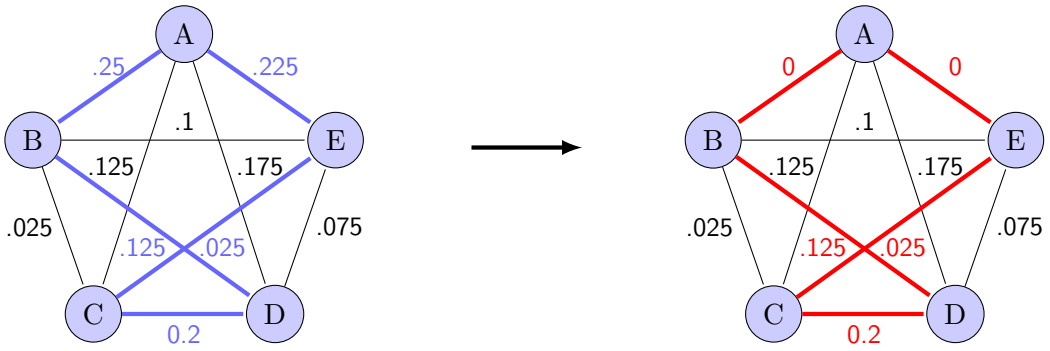
Cycle 1: $A \to B \to C \to D \to E \to A$. $u_{970} = e^{i2\pi(0.775)}$, $u'_{970} = e^{i2\pi(0.3)}$



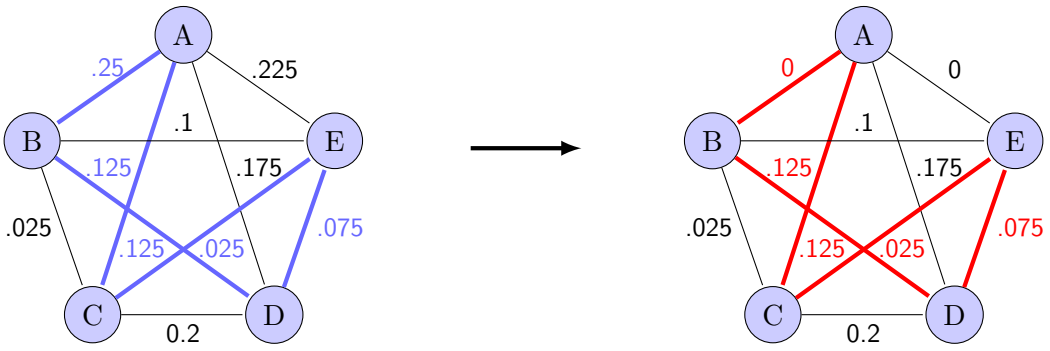
Cycle 2: $A \to B \to C \to E \to D \to A$. $u_{978} = e^{i2\pi(0.55)}$, $u'_{978} = e^{i2\pi(0.3)}$



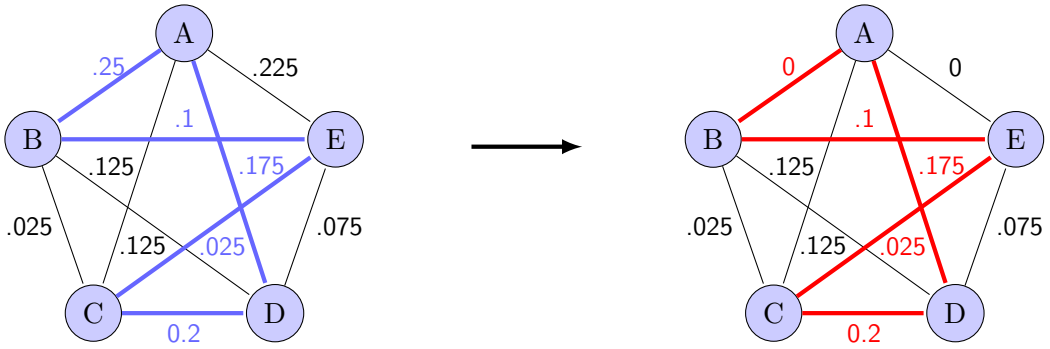
Cycle 3: $A \to B \to D \to C \to E \to A$. $u_{1110} = e^{i2\pi(0.825)}$, $u'_{1110} = e^{i2\pi(0.35)}$



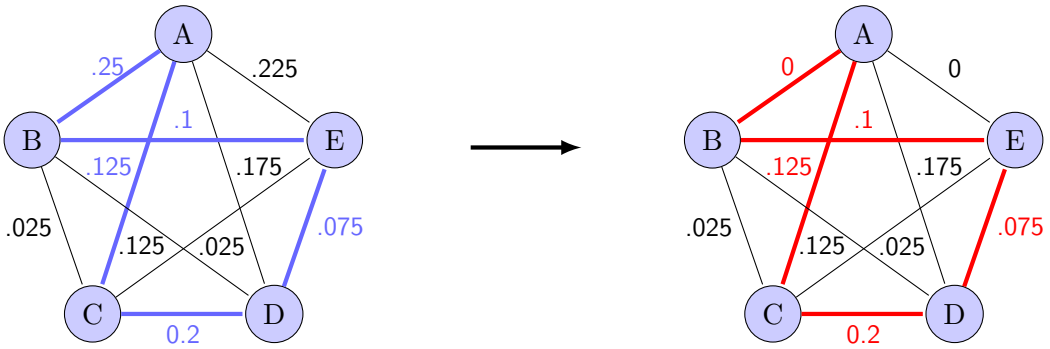
Cycle 4: $A \to B \to D \to E \to C \to A$. $u_{1022} = e^{i2\pi(0.6)}$, $u'_{1022} = e^{i2\pi(0.35)}$



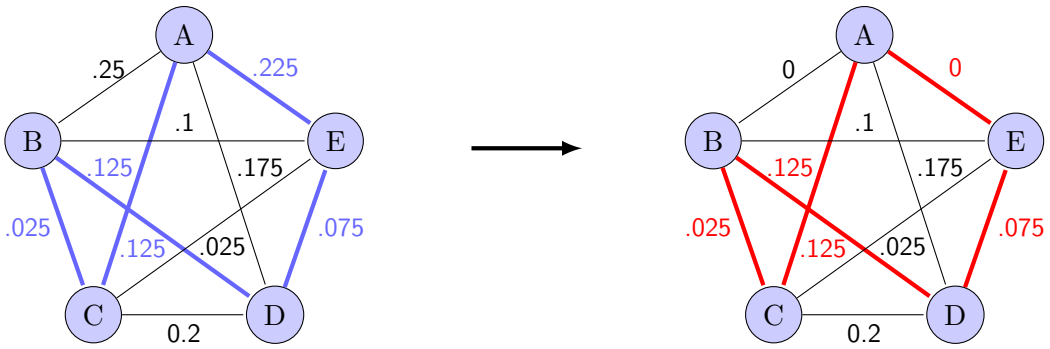
Cycle 5: $A \to B \to E \to C \to D \to A$. $u_{1202} = e^{i2\pi(0.75)}$, $u'_{1202} = e^{i2\pi(0.5)}$



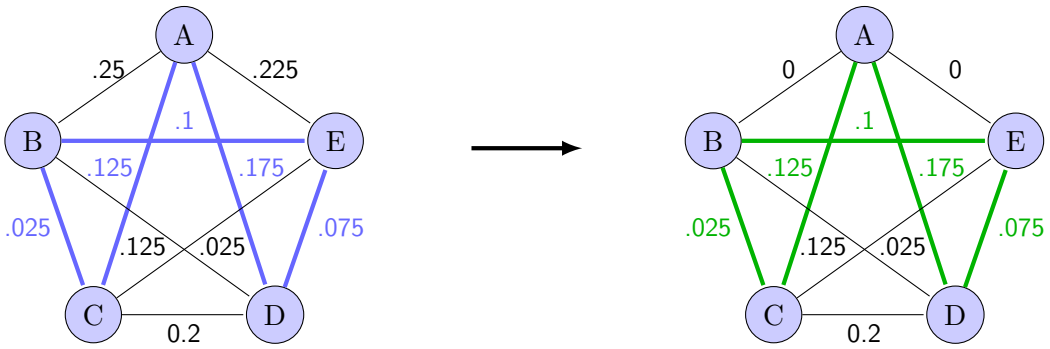
Cycle 6: $A \to B \to E \to D \to C \to A$. $u_{1138} = e^{i2\pi(0.75)}$, $u'_{1138} = e^{i2\pi(0.5)}$



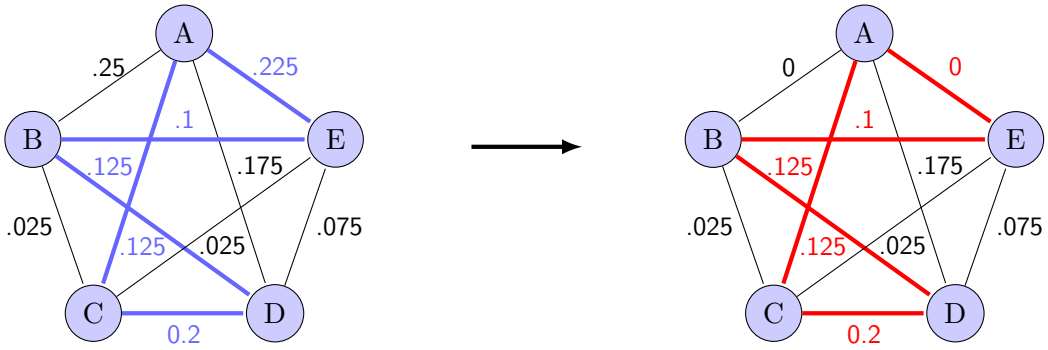
Cycle 7: $A \to C \to B \to D \to E \to A$. $u_{1670} = e^{i2\pi(0.575)}$, $u'_{1670} = e^{i2\pi(0.35)}$



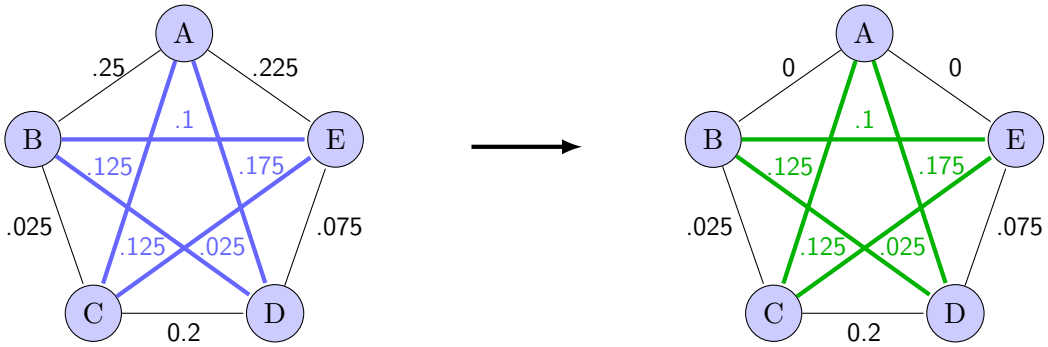
Cycle 8: $A \to C \to B \to E \to D \to A$. $u_{1778} = e^{i2\pi(0.5)}$, $u'_{1778} = e^{i2\pi(0.5)}$



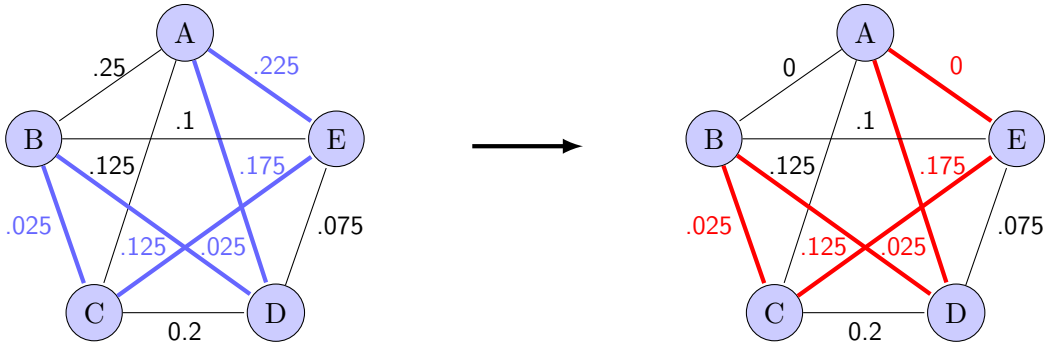
Cycle 9: $A \to C \to D \to B \to E \to A$. $u_{1830} = e^{i2\pi(0.775)}$, $u'_{1830} = e^{i2\pi(0.55)}$



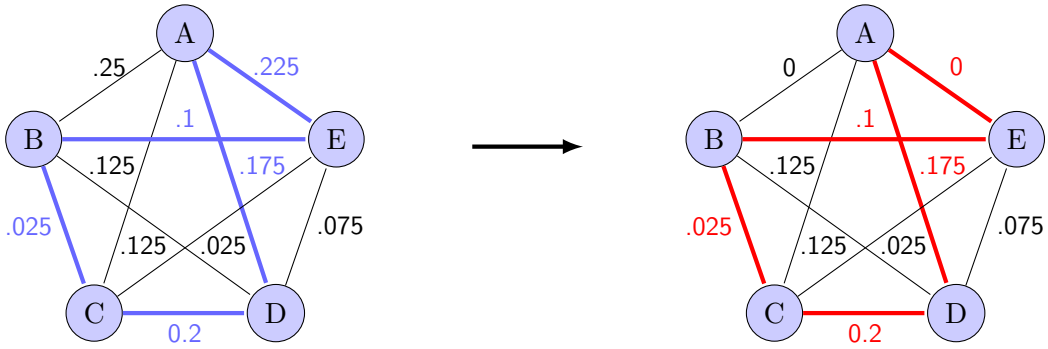
Cycle 10: $A \to C \to E \to B \to D \to A$. $u_{1726} = e^{i2\pi(0.55)}$, $u'_{1726} = e^{i2\pi(0.55)}$



Cycle 11: $A \to D \to B \to C \to E \to A$. $u_{2230} = e^{i2\pi(0.575)}$, $u'_{2230} = e^{i2\pi(0.35)}$



Cycle 12: $A \to D \to C \to B \to E \to A$. $u_{2410} = e^{i2\pi(0.725)}$, $u'_{2410} = e^{i2\pi(0.5)}$



4.2.2 Results: Simulations with Qiskit

Fig. 4.5 illustrates the circuit for our first Hamiltonian cycle initialized in the eigenstate $|970\rangle = |001111001010\rangle$. Similar circuits will be executed for the other eleven Hamiltonian cycles, with the only variation being the eigenstate initialization. Our simulations are ideal, implying that our results are not affected by noise. Each circuit is executed 1024 times by default, providing us with a quasi-probability distribution.

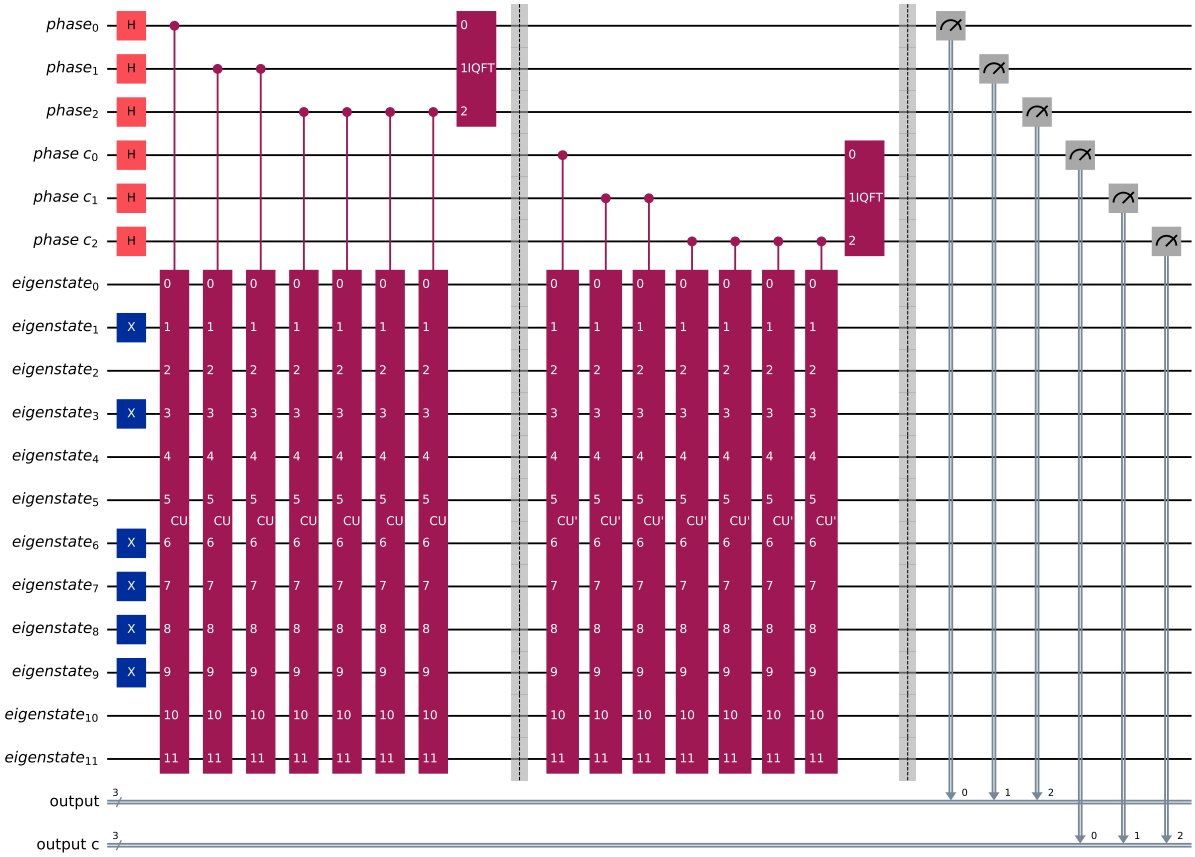


Figure 4.5: The quantum circuit for the 5-city graph involves a 3-qubit phase estimation, measuring the Hamiltonian cycle $A \to B \to C \to D \to E \to A$. The corresponding eigenstate is $|970\rangle = |001111001010\rangle$, but due to Qiskit's convention on qubit ordering, the eigenstate is initialized in reverse. The CU gate represents the control unitary matrix containing all the Hamiltonian cycles. The CU' gate includes the same cycles, but before its construction, all edge weights not satisfying the condition (< 9) were set to zero. We have two sets of 3 qubits to be measured and stored in classical registers labelled 'output' and 'output c'.

As we did for the 4-city graph, for each Hamiltonian cycle, we can examine the two states with the highest counts and summarize them in Table 4.2. This table also presents 4-qubit estimation.

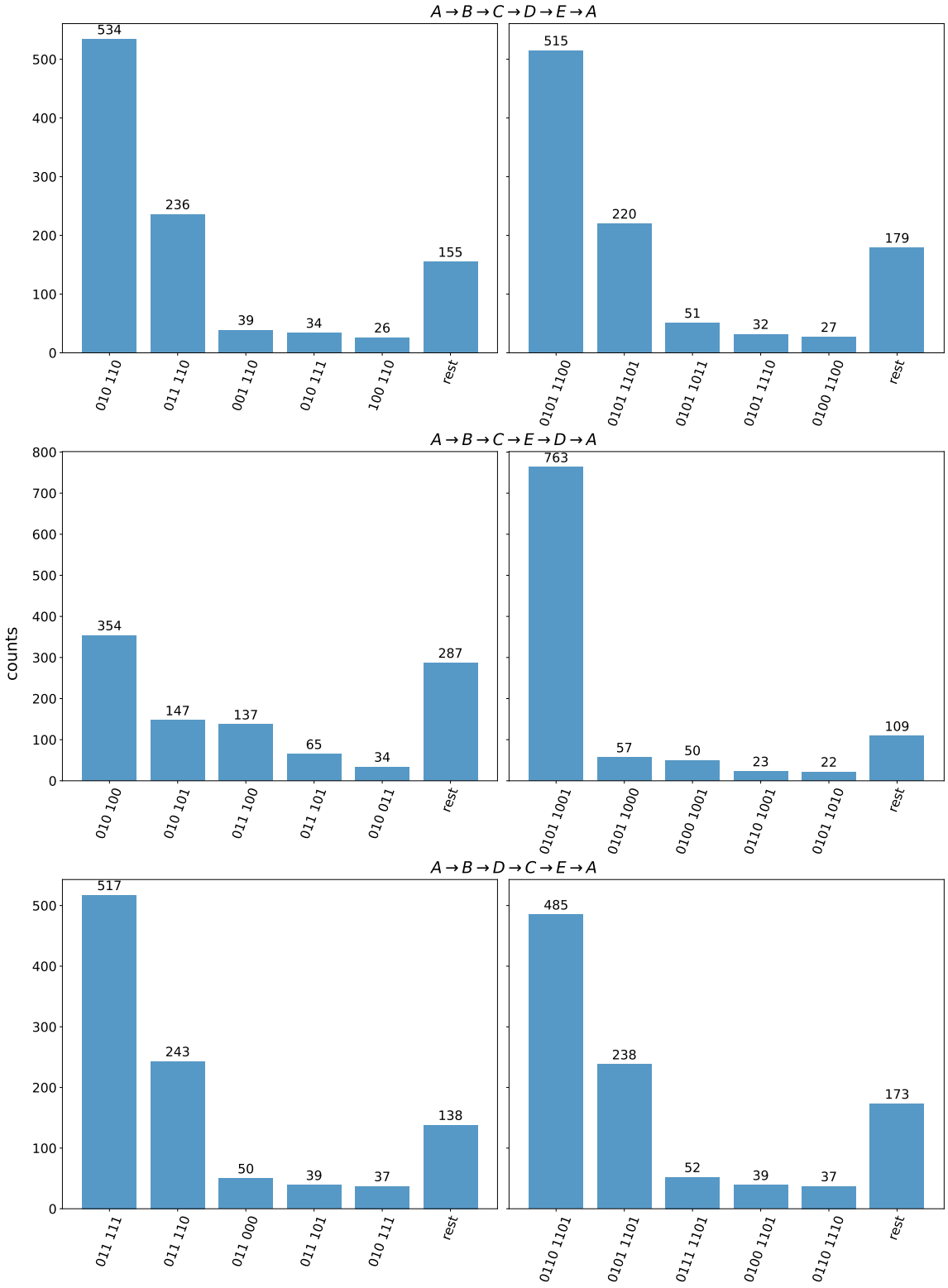


Figure 4.6: 3 qubit (left column) & 4 qubit (right column) phase estimation for the 5-city graph. Hamiltonian cycles: 1 to 3

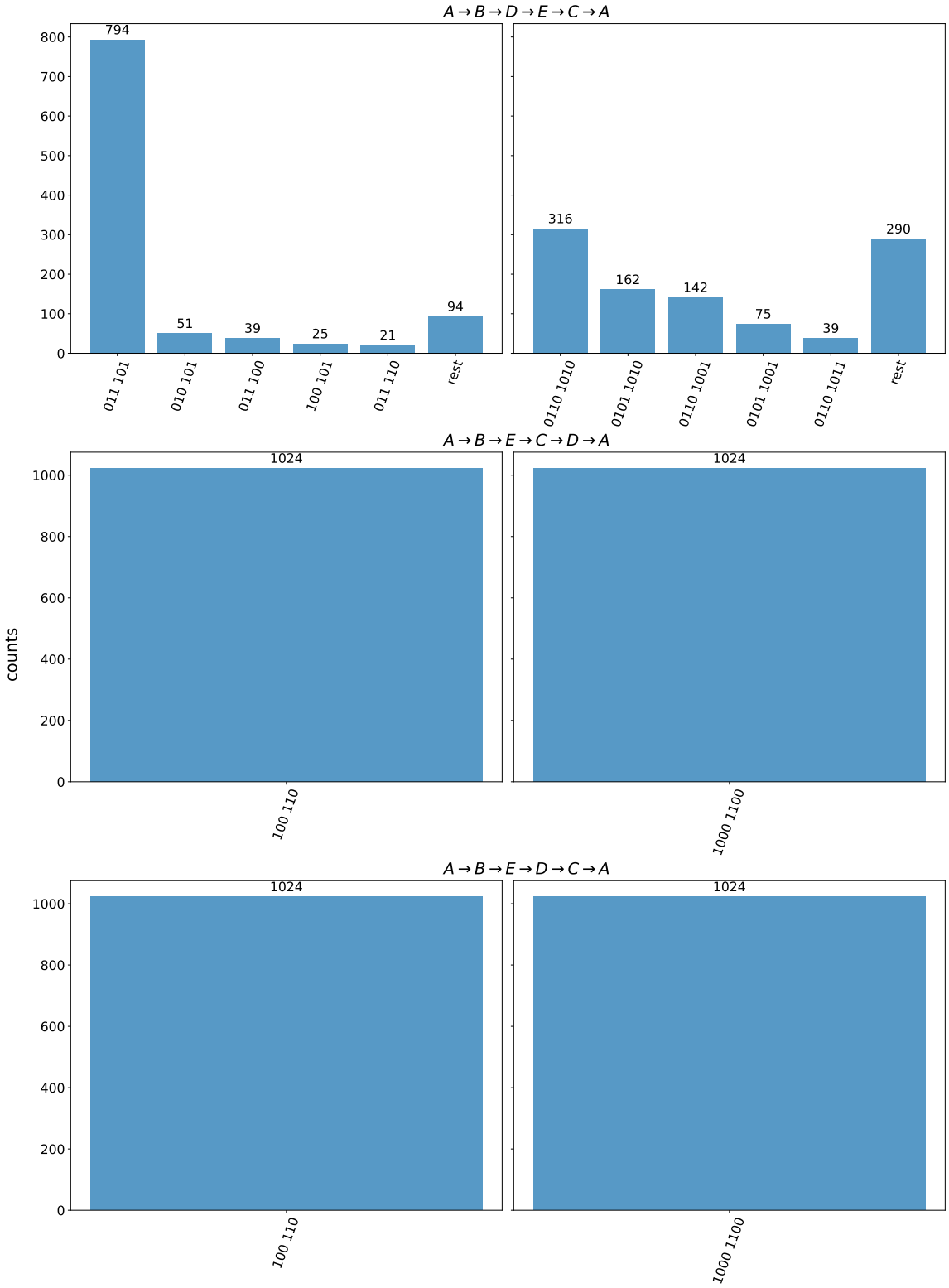


Figure 4.7: 3 qubit (left column) & 4 qubit (right column) phase estimation for the 5-city graph. Hamiltonian cycles: 4 to 6

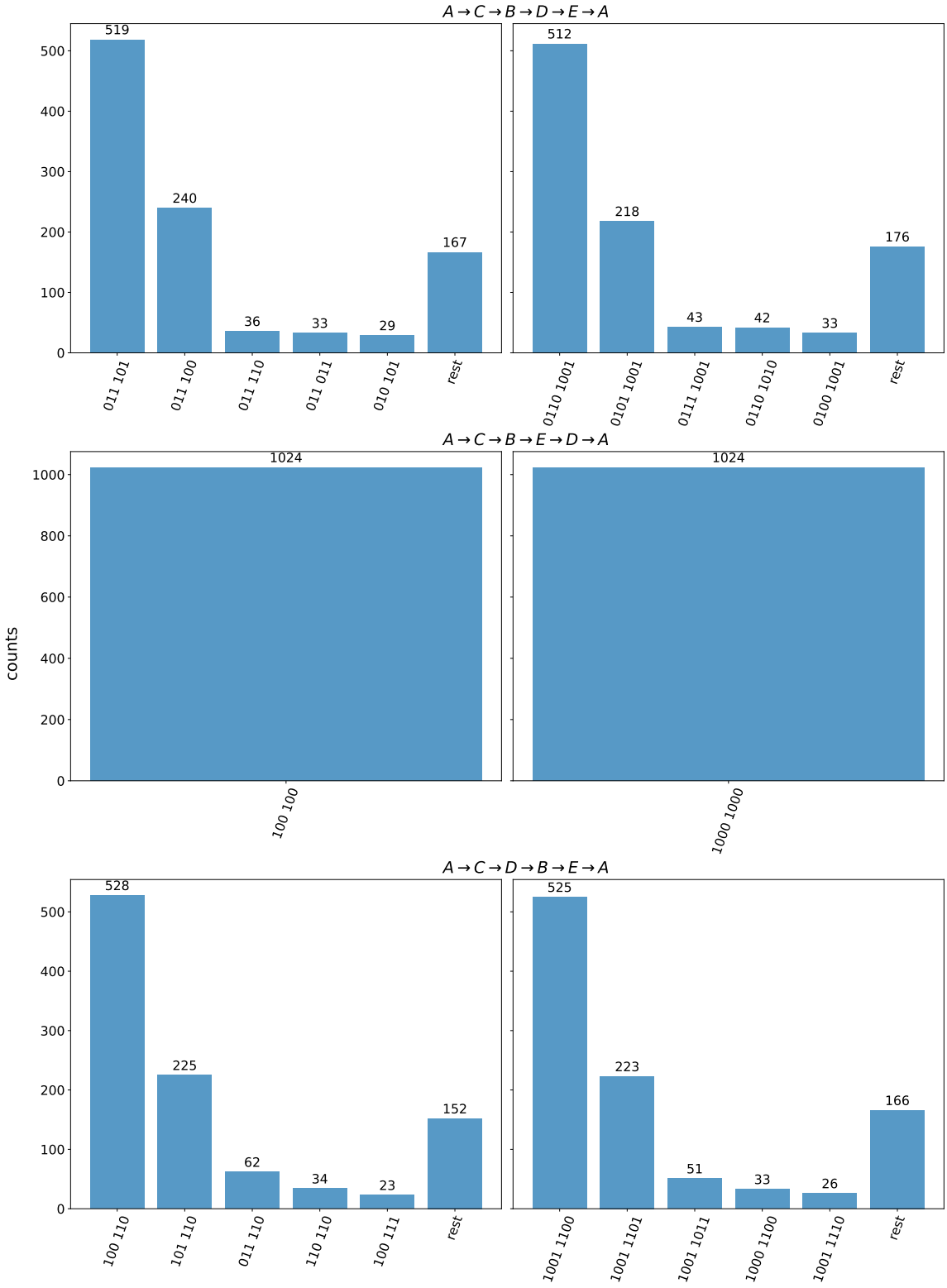


Figure 4.8: 3 qubit (left column) & 4 qubit (right column) phase estimation for the 5-city graph. Hamiltonian cycles: 7 to 9

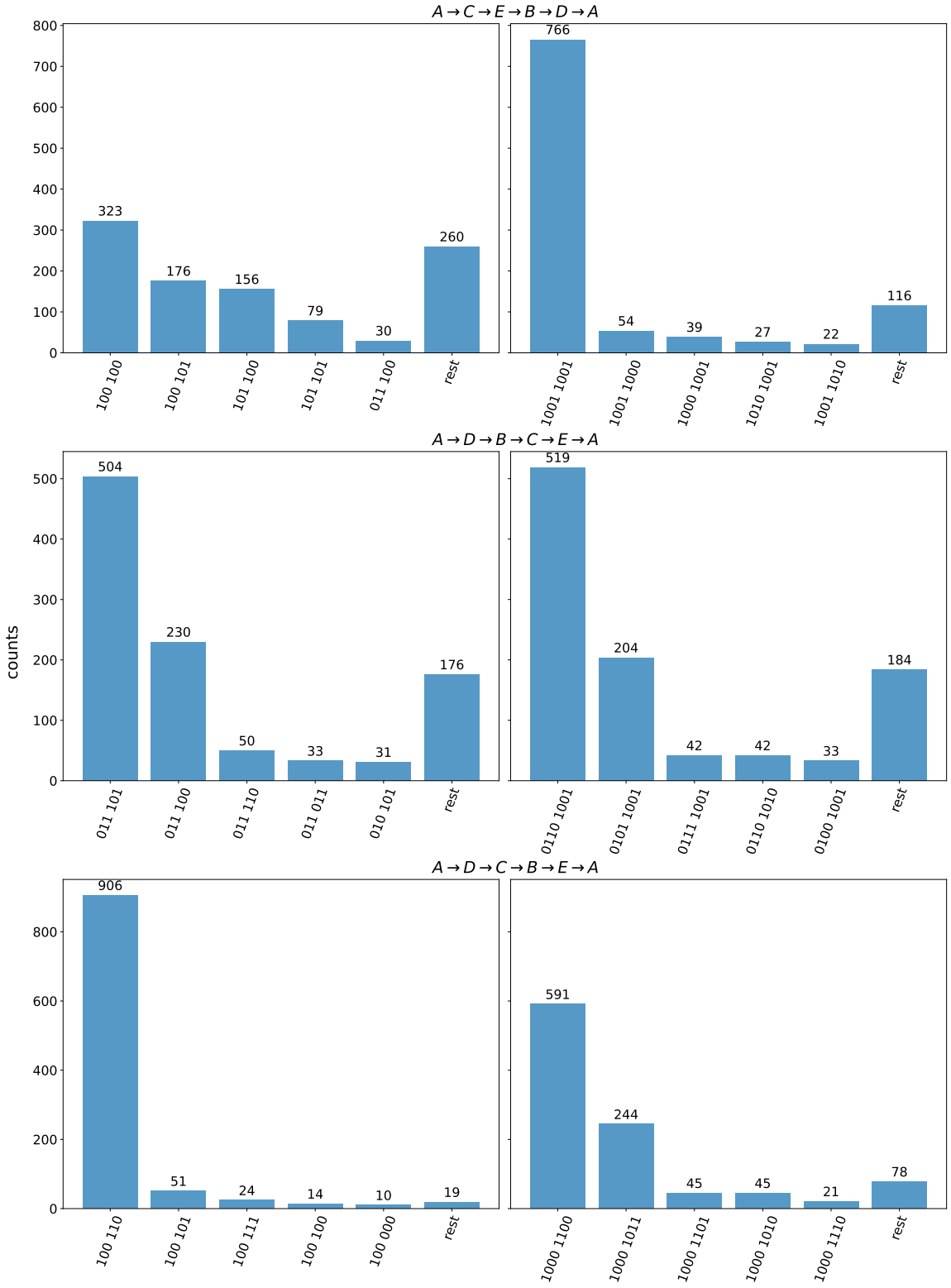


Figure 4.9: 3 qubit (left column) & 4 qubit (right column) phase estimation for the 5-city graph. Hamiltonian cycles: 10 to 12

Cycle	Expected	Phase Qubits	Highest Counts	Prob.	2nd Highest Counts	Prob.
1	0.3, 0.775	3	0.25, 0.75	52%	0.375, 0.75	23%
		4	0.3125, 0.75	50%	0.3125, 0.8125	21%
2	0.3, 0.55	3	0.25, 0.5	35%	0.25, 0.625	14%
		4	0.3125, 0.5625	75%	0.3125, 0.5	6%
3	0.35, 0.825	3	0.375, 0.875	50%	0.375, 0.75	24%
		4	0.375, 0.8125	47%	0.3125, 0.8125	23%
4	0.35, 0.6	3	0.375, 0.625	78%	0.25, 0.625	5%
		4	0.375, 0.625	31%	0.3125, 0.625	16%
5	0.5, 0.75	3	0.5, 0.75	100%		
		4	0.5, 0.75	100%		
6	0.5, 0.75	3	0.5, 0.75	100%		
		4	0.5, 0.75	100%		
7	0.35, 0.575	3	0.375, 0.625	51%	0.375, 0.5	23%
		4	0.375, 0.5625	50%	0.3125, 0.5625	21%
8	0.5, 0.5	3	0.5, 0.5	100%		
		4	0.5, 0.5	100%		
9	0.55, 0.775	3	0.5, 0.75	52%	0.625, 0.75	22%
		4	0.5625, 0.75	51%	0.5625, 0.8125	22%
10	0.55, 0.55	3	0.5, 0.5	32%	0.5, 0.625	17%
		4	0.5625, 0.5625	75%	0.5625, 0.5	5%
11	0.35, 0.575	3	0.375, 0.625	49%	0.375, 0.5	22%
		4	0.375, 0.5625	51%	0.3125, 0.5625	20%
12	0.50, 0.725	3	0.5, 0.75	88%	0.5, 0.625	5%
		4	0.5, 0.75	58%	0.5, 0.6875	24%

Table 4.2: Simulation results for the 5-city graph.

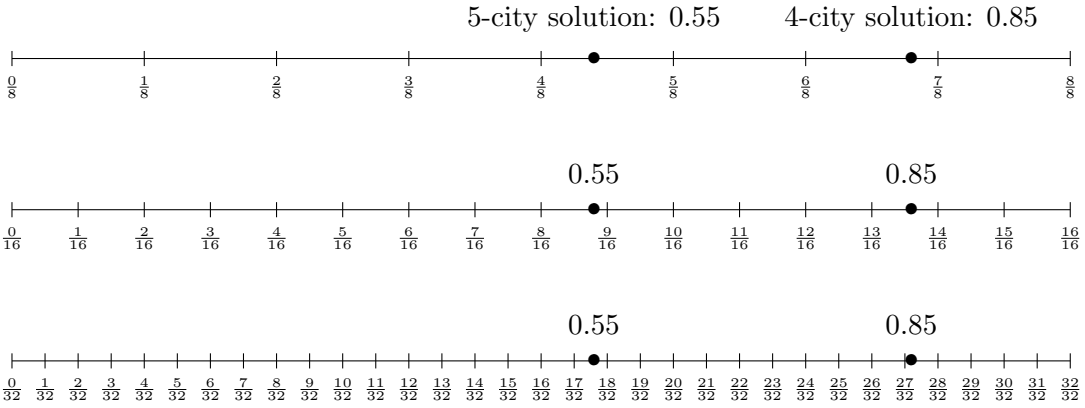
Chapter 5

Discussion

Referring to Table 4.1, we observe that cycle 1 represents the solution for our 4-city problem, as our expected phase estimation before and after satisfying the condition is both equal to 0.85. With 3-qubit estimation, we achieved a 77% quasi-probability of measuring the state $0.875, 0.875$. This suggests that the most probable result indicates the cycle as a solution. One might expect a better quasi-probability as we increase the number of phase qubits, as it should provide a finer estimate of our phases. However, with 4-qubit estimation, our quasi-probability for the most measured state, $0.875, 0.875$, drops to 34%. Despite this decrease, it remains the state with the most counts, indicating equality of our two phases and thus a solution. Further, 5-qubit estimation restores the quasi-probability for the most measured state, in this case, $0.84375, 0.84375$, to 77%. This suggests that increasing the number of phase qubits does not necessarily increase the probability of measuring a single state. To further elaborate on why this is the case, we can examine the representation of 0.85 in binary:

$$0.11011001100110011...$$

Since the binary representation of 0.85 is recurring, perfect accuracy with phase estimation is unattainable. This same challenge arises in our 5-city simulation, where cycle 10 (Fig. 4.9) represents our solution, and the phase we aim to estimate is 0.55, which also recurs in binary. In both cases, we can only establish a very good approximate value. To illustrate this further, let's examine the resolution of 3, 4, and 5 qubit estimation below and refer to Fig. 2.3 to understand the probabilities.



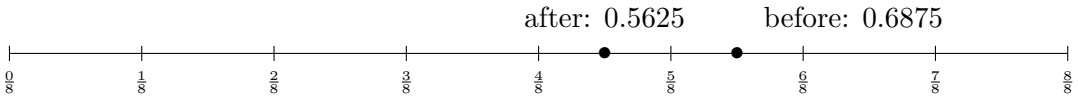
An intuitive way to interpret the number lines is that when the value is near a specific step, that step becomes more likely to be measured. If the value falls perfectly between two steps, the best possible approximation is either the lower or upper bound, both with an equal probability, approximately $P_b \approx 40\%$. It's important to note that we are measuring two independent phases for each Hamiltonian cycle, so our best approximation will be $P_b^2 \approx 16\%$.

We can see for the 4-qubit estimation, the introduction of a more precise value of 13/16 increased its probability of being measured as a good approximation for 0.85, and in turn decreased the probability of $7/8 = 14/16$ being measured. Performing 5-qubit estimation does

improve our results but does not justify the computational cost considering we achieved similar results with 3-qubit estimation. We see a similar issue with our results for the 5-city problem; we only simulated 3 and 4-qubit estimation, but had we done 5, our confidence in the result would have decreased.

Unless we obtain a fortunate solution as in cycle 8 of the 5-city problem (Fig. 4.8), where our $\phi = 0.5$, can be represented exactly, we need to run phase estimation a number of times from which we can use the mode of our results as the solution. And as we've discussed earlier, using a larger number of phase qubits does not necessarily improve your confidence in the result. If we only consider the states with the highest counts in Tables 4.1 & 4.2, then our results perfectly predict the solutions and non-solutions.

The question we need to ask now is how many phase qubits are needed to correctly identify a solution or non-solution based on the most frequent state measured? And the answer is we need to make sure the resolution of our phase qubits to be finer than the condition value, α. Let's consider a non-solution where the reduction to ϕ is exactly the resolution of 3-qubit estimation, $\alpha = 0.125$:



Based on our best approximation, we are likely to obtain any of the following four results most frequently with $P_b^2 \approx 16\%$:

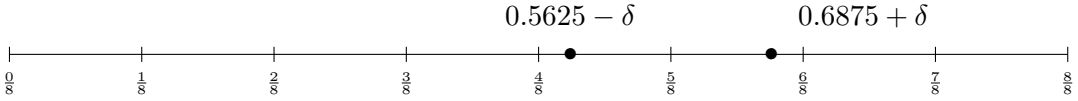
$$P_1 : 0.625, 0.625$$

$$P_2 : 0.500, 0.625$$

$$P_3 : 0.625, 0.750$$

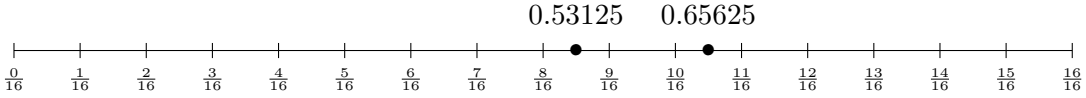
$$P_4 : 0.500, 0.750$$

Here, we run the risk of selecting P_1, which implies a solution. By extending the distance between our ϕ values, we reduce the probability of measuring 0.625. We can predict, based on qualitative analysis, how probable our four results above will be:



$$P_1 < P_2 = P_3 < P_4$$

We actually simulated a similar case in cycle 7 of our 5-city problem (Fig. 4.8), where the least probable result of the top four states indicated the cycle was a solution. If $\delta << 1$, we would have to run our circuit many times over to appropriately measure the non-solution as the most frequent result. Otherwise, we run the risk again of predicting a solution with P_1 as it would be very close to P_4. A safe bet would be to have a phase resolution at least twice the condition value. Let's illustrate this again by inconveniently placing the phases we hope to estimate perfectly in between steps:



We now run no risk of the four most probable states indicating a solution. As our measurement will likely collapse to any of these four options with $P_b^2 \approx 16\%$:

$$P_1 : 0.5000, 0.6250$$

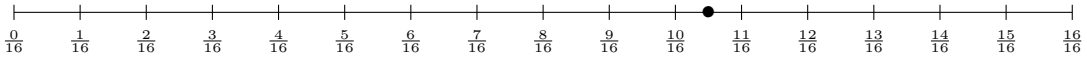
$$P_2 : 0.5000, 0.6875$$

$$P_3 : 0.5625, 0.6250$$

$$P_4 : 0.5625, 0.6875$$

We can further demonstrate that in the case of a solution, two of the four states will indicate it as a solution, while the other two will not. However, this discrepancy can be attributed to rounding errors, as our condition would result in a much greater reduction in phase. To illustrate, let's consider the phase 0.65625:

before & after: 0.65625



$$P_1 : 0.6250, 0.6250$$

$$P_2 : 0.6250, 0.6875$$

$$P_3 : 0.6875, 0.6250$$

$$P_4 : 0.6875, 0.6875$$

We can also discuss the number of eigenstate qubits needed. Our method asks us to construct a matrix of size $N^N \times N^N$, where N represents the number of cities. This does result in a unitary matrix however, it cannot be neccesarily represented by an exact number of qubits. We encountered this issue in our 5-city graph, where we introduced ones at the end of the matrix, Equation (4.1). If n represents the number of qubits we need and m represents the number of cities, we need to solve the equation:

$$2^n = m^m$$

$$n = m\log_2(m)$$

We can conclude that the number of eigenstate qubits needed will scale polynomially, where it is less than m^2. For example, in the case of 5 cities, $n \approx 11.6$, which we round up to 12 qubits.

Finally, because we iterate through all Hamiltonian cycles to solve this problem, our time complexity remains $O((N-1)!)$. Alternatively, we can solve this decision problem by running all Hamiltonian cycles in parallel, which reduces our execution time. However, the tradeoff would be introducing exponential space complexity, $O((N-1)!)$.

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] Phase-estimation and factoring: Ibm quantum learning.

[8] 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

Hamiltonian Cycles and Locating Eigenstates Code

The code detailed in this section can generate the information found in Tables 3.1, 3.2, 3.3 and 3.4, which construct the Hamiltonian cycles and locate the 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

Simulation Code with Qiskit

The code detailed in this section demonstrates simulations for the four-city graph, as discussed in Section 4.1.2. This code can be extended to model the five-city graph presented in Section 4.2.2, and can be further adapted to accommodate even larger city graphs.

[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
```

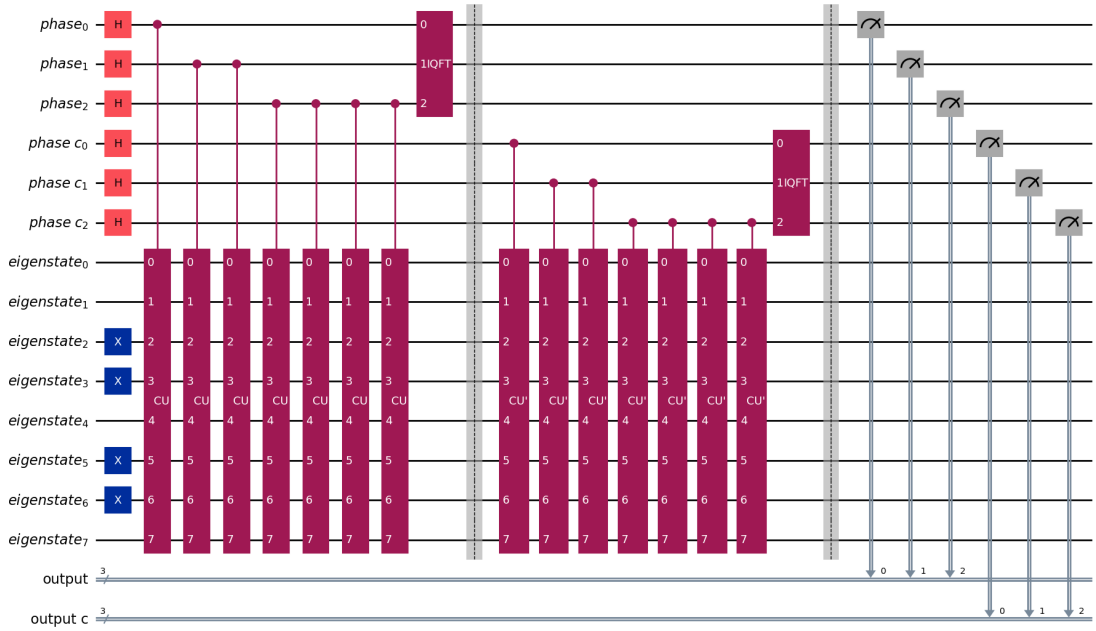
[7]:
```
array([0.85, 0.55, 0.5 ])
```

[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')
```

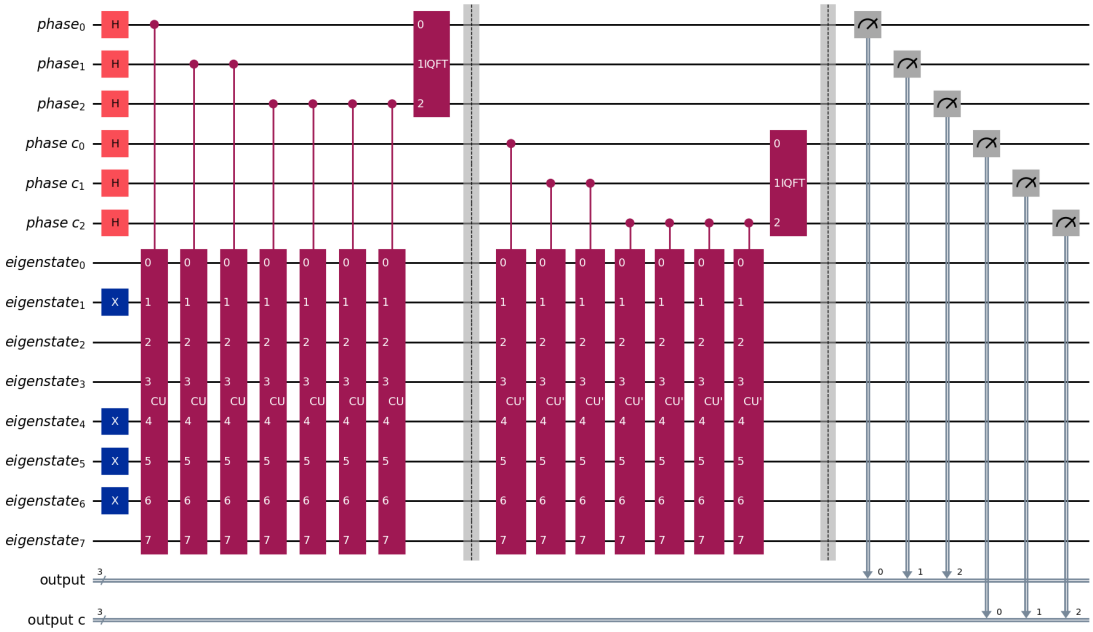
[9]:



[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')
```

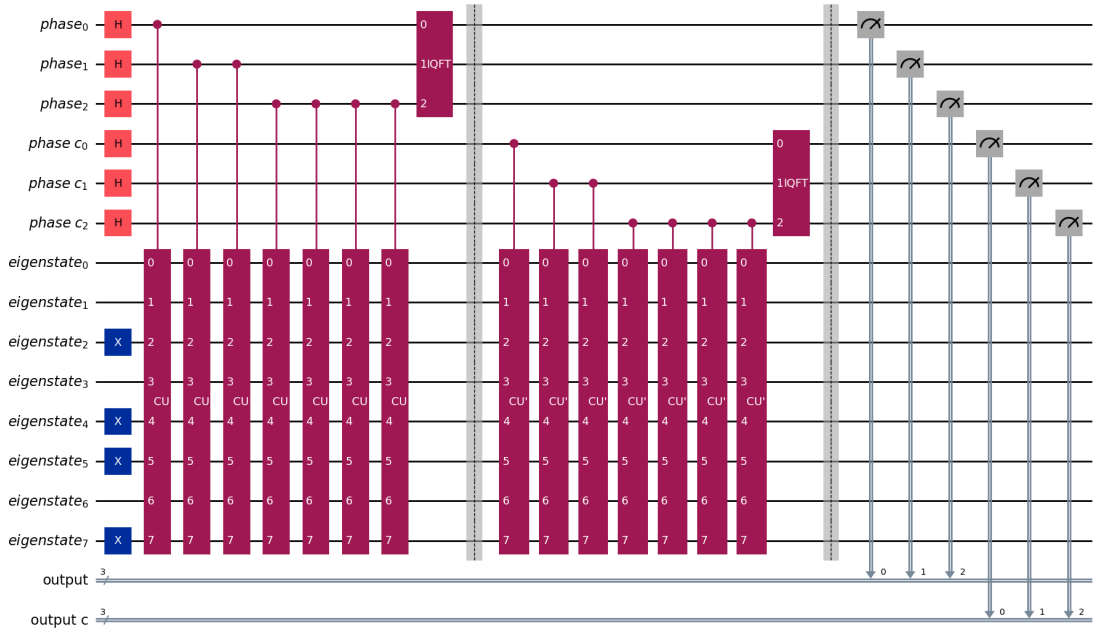
[11]:



[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()
```