Master's Thesis
Theoretical and Computational Methods
Quantum Computing

Using Quantum Computing to Improve on the Traveling Salesman Problem

Kseniya Rychkova

June 4, 2022

Supervisor(s): Boris Sokolov, Sabrina Maniscalco

Examiner(s): Boris Sokolov
Sabrina Maniscalco

University of Helsinki
Faculty of Science

PL 64 (Gustaf Hällströmin katu 2a)

00014 Helsingin yliopisto

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Theoretical and Computational Methods Quantum Computing

Tekijä — Författare — Author

Kseniya Rychkova

Työn nimi — Arbetets titel — Title

Using Quantum Computing to Improve on the Traveling Salesman Problem

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Master's Thesis	June 4, 2022	48

Tiivistelmä — Referat — Abstract

The Traveling Salesman Problem (TSP) is a well-known optimization problem. The time needed to solve TSP classically grows exponentially with the size of the input, placing it into the NP-hard computational complexity class–the class of problems that are at least as hard as any other problem solvable in nondeterministic polynomial time. Quantum computing gives us a new approach to searching through such a huge search space, using methods such as quantum annealing and phase estimation. Although the current state of quantum computers does not give us enough resources to solve TSP with a large input, we can use quantum computing methods to improve on existing classical algorithms. The thesis reviews existing methods to efficiently tackle TSP utilizing potential quantum resources, and discusses the augmentation of classical algorithms with quantum techniques to reduce the time complexity of solving this computationally challenging problem.

Avainsanat — Nyckelord — Keywords

LaTeX

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction **1**

1.1 Computational complexity 1

1.1.1 Turing machines 2

1.1.2 Computational complexity classes 4

1.1.3 Quantum complexity theory 5

1.2 Traveling salesman problem 7

2 Quantum Computing **9**

2.1 Single qubit states 9

2.2 Quantum gates 11

2.2.1 Pauli gates 12

2.2.2 Hadamard gate 12

2.2.3 I, P, S, T gates 13

2.2.4 U gate 14

2.3 Quantum measurement 14

2.4 Multiple qubit states 15

2.5 Quantum algorithms 17

2.5.1 Controlled gates 17

2.5.2 Quantum Fourier Transform 18

2.5.3 Quantum circuits 19

2.6 Quantum annealing 20

2.7 NISQ 21

3 Solving TSP classically **23**

3.1 Exact solutions 23

3.1.1 Brute-force algorithm 23

3.1.2 Held-Karp algorithm 23

3.2 Approximate solutions 24

3.2.1 Greedy algorithm 25

3.2.2 Nearest-neighbor algorithm 25

3.2.3 Christofides algorithm 26
3.2.4 Yatsenko's algorithm 27
3.2.5 Other algorithms 28

4 Solving TSP with quantum computing 29
4.1 Phase estimation 29
4.1.1 Encoding the edge weights 29
4.1.2 Constructing the unitary operators 31
4.1.3 Setting up the circuit 32
4.1.4 Interpreting the output 34
4.1.5 Analysis 34
4.2 Quantum annealing 35
4.2.1 Ising model 35
4.2.2 Encoding TSP into the Ising problem 36
4.2.3 Analysis 38
4.3 Quantum algorithm based on Held-Karp 38
4.3.1 The algorithm 38
4.3.2 Analysis 39

5 Conclusion 41
5.1 Analysis 41
5.2 Future work 42

1. Introduction

Quantum computing is a rising field in technology that offers a new and different way of performing computations. Although they might never completely replace classical computers, quantum computers can perform certain kinds of computations much more efficiently than classical computers. The Traveling Salesman Problem (TSP) is a well-known computationally difficult problem, and the best classical algorithms either take a huge amount of time (proportionally to the size of the given problem instance) to find an optimal solution, or trade accuracy for speed and do not guarantee an exact solution at all. We will analyze various classical algorithms, and then explore some quantum algorithms that could potentially improve the speed and accuracy of solving TSP.

1.1 Computational complexity

An algorithm is a finite sequence of well-defined instructions that perform a task, often used to solve a problem. There can be many different algorithms that can solve the same problem. A good way to compare algorithms is by their time and space complexities. Time and space are both limited resources when it comes to computing. Time refers to the amount of time an algorithm takes to arrive to a stopping point, usually measured in the total number of steps the algorithm takes as a function of the size of the input. Space, in the context of computation, is the number of bits in memory an algorithm needs to use for the computation, and is also measured as a function of the size of the input [1].

Big-O notation is often used when analyzing and comparing algorithms; it is used to describe the behavior of a function at its limits, and can simplify complicated functions to make it easier to compare to other functions [1]. Big-O is the asymptotic upper bound, and is written as $f(x) = \mathcal{O}(g(x))$, meaning that there exist some positive constants c and k such that $0 \leq f(x) \leq cg(x)$ for all $x \geq k$ [2]. Here the function f is the function we are estimating, and g is the comparison function. For a function f which is polynomial, it is conventional to use only the highest degree term for the Big-O notation, and all multiplicative and additive scalars are disregarded (for example,

$f(x) = 3x^3 + 2x^2 + 5x + 1$ is $\mathcal{O}(x^3)$, since $f(x) \leq 12x^3$ for all $x \geq 1$) [1].

Although the size of the input to an algorithm is technically measured in bits, it can be simplified to be the number of input values instead. For example, for an algorithm that sorts values in an array, the size of the input would be the number of values in the array, even though that may or may not equal the number of bits needed to represent the entire array [1]. For a problem whose input can be represented as a graph, the size of the input would often be the number of vertices in the graph. We will refer to the input size of the problem as n.

Here are some note-worthy time complexities with examples of algorithms with the listed runtimes, listed in order from slowest to fastest growth [3]:

- constant time, $\mathcal{O}(1)$ - accessing an array index, inserting or deleting from a list;
- logarithmic time, $\mathcal{O}(\log n)$ - binary tree functions, binary search;
- linear time, $\mathcal{O}(n)$ - summing up values in a list, for/while loops, linear search;
- quasilinear time, $\mathcal{O}(n \log^k n)$ for some positive constant k - merge sort;
- quadratic time, $\mathcal{O}(n^2)$ - traversing a 2-D array, insertion sort;
- polynomial time, $\mathcal{O}(n^k)$ for some positive constant k - maximum matchings in graphs, basic arithmetic operations, selection sort;
- exponential time, $\mathcal{O}(2^{p(n)})$ where $p(n)$ is some polynomial in n - generating all password possibilities of length n;
- and factorial time $\mathcal{O}(n!)$ - generating all permutations of a list of n elements.

Some computational problems are more difficult to solve than others. The computational problems are classified into complexity classes based on the time or space complexities of the best known (or proven) algorithms needed to solve them. Since there are different types of computers, we use Turing machines as the standard models, the deterministic version of which (processing at most one step at a time) has the same computational power as our real-world standard computer [4]. In this thesis, we will focus on the classical complexity classes polynomial time (P) and nondeterministic polynomial time (NP), as well as some probabilistic and quantum complexity classes.

1.1.1 Turing machines

Firstly, let us define a Turing machine, or TM. A Turing machine is a theoretical, abstract machine that takes as input a discrete string of symbols of a predefined, finite

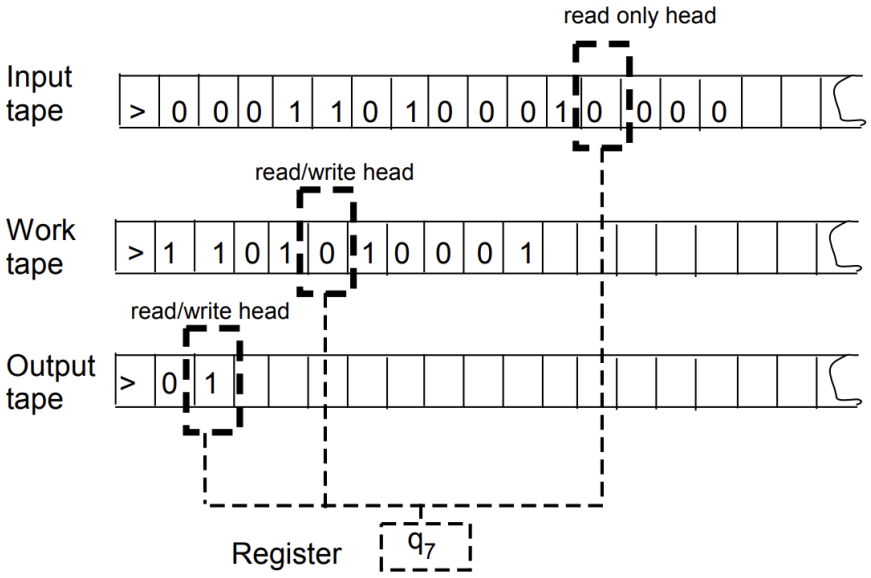

Figure 1.1: An example of a TM with three tapes. [5]

alphabet and, based on a finite set of rules, performs any of the following operations [5]:

- Read a symbol of the input.
- Read a symbol from the working space, or "scratch pad" (i.e. local memory used for performing computations).
- Write a symbol to the scratch pad, based on the values read.
- Either stop and output a 0 or a 1, or move on to the next step based on the rule set.

The scratch pad in a TM is a finite set of tapes: infinite one-directional line of cells, with each cell being able to hold a symbol from the alphabet of the machine. Each tape has a tape head: a tool that can move left or right along the tape, one cell at a time, and potentially read or write symbols on the tape. One of the tapes in the TM is the input tape, which is read-only, and another is the output tape, which is where the TM writes its results before halting [5].

The states of a TM are based on the set of rules of the TM, the positions of the tape heads, and what the readings of the tape heads are. The states include a start state and a halting state. Figure 1.1 shows the current state of the TM in the register. The TM also includes a defined transition function, which describes the rule the TM uses at each step of the computation [5].

A non-deterministic Turing machine is able to take multiple paths at once, as opposed to deciding on exactly one transition at any step. In fact, a non-deterministic TM can process a limitless number of branches at once. This way, it can perform an exponential number of tasks in polynomial time.

We use the Turing machine models to define the computational complexity classes.

1.1.2 Computational complexity classes

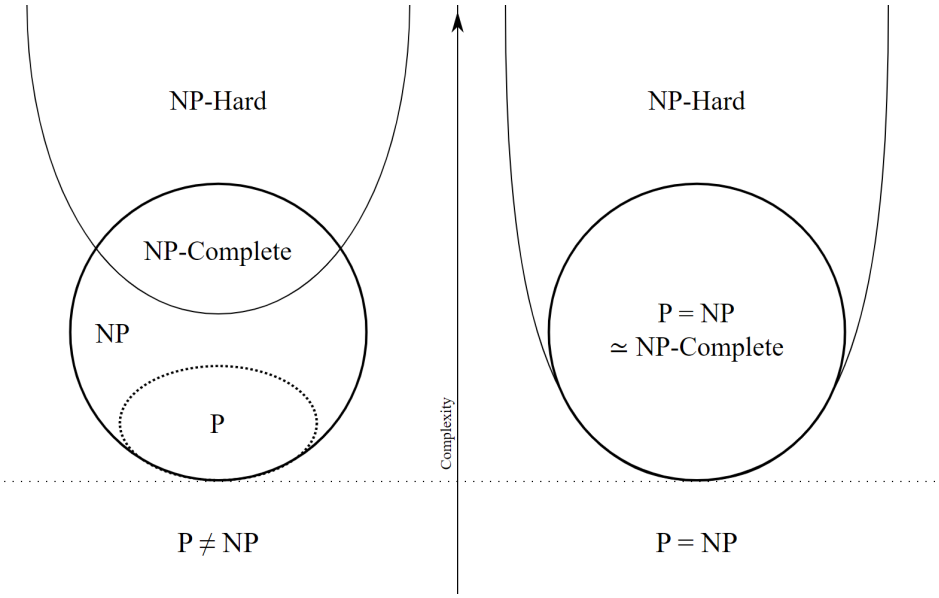

Figure 1.2: A Euler diagram for the P, NP, NP-complete, and NP-hard complexity classes. The left side is assuming that P $\neq$ NP, and the right side assumes that P = NP. [6]

The complexity class P contains all problems that can be solved by a deterministic TM in polynomial time in the worst case (the worst possible instance of a problem of the given input size) [4]. This means that there exists, or is proven that there can exist, an algorithm that solves the problem in $\mathcal{O}(n^k)$ time, where k is a positive constant. Some examples of problems in P are array-sorting problems, matrix multiplication, and finding shortest paths in networks [4].

The NP complexity class contains problems whose solutions can be verified by a deterministic TM in polynomial time, but the solution to the problem may be more difficult to obtain [4]. Clearly, P is a subset of NP, since a solution to a P problem can be verified in polynomial time or faster. An example of a problem in NP but not in P is the Boolean satisfiability problem, also known as SAT. The problem is presented as a propositional logic formula built using variables that can take on the values true or false, conjunctions, disjunctions, negations, and parentheses, and asks the question if

there exists an assignment of values to the variables to make the formula true. Given an assignment, it is simple to check whether the assignment satisfies the formula, and can easily be done within polynomial time bounds. However, finding an assignment to satisfy the formula is much trickier.

SAT is an example of an NP-complete problem. NP-complete is a complexity class of decision problems for which answers can be checked for correctness in polynomial time, and no other NP problem is no more than a polynomial factor harder [7].

Another subset of NP is the NP-hard complexity class. NP-hard problems are "at least as hard as the hardest problems in NP". The more formal definition is that a problem H is NP-hard if any problem L in NP can be reduced in polynomial time to H, meaning that if it takes 1 unit of time to solve H, the solution of H can be used to solve L in polynomial time [8].

While the SAT problem is NP-complete, its generalized version MAX-SAT (maximum satisfiability) is NP-hard. This problem asks for the maximum number of clauses that can be made true by assigning truth values to the variables in the Boolean formula.

Figure 1.2 illustrates the relationship between the P, NP, NP-complete, and NP-hard complexity classes. Note that it is not yet proven whether or not P = NP (i.e. whether or not the problems we classify as NP can actually be solved in polynomial time), though it is widely suspected that P $\neq$ NP.

The probabilistic Turing machine (PTM) is a variation of a non-deterministic Turing machine that chooses between available transitions at each step based on some probability distribution [5].

Bounded-error probabilistic polynomial time (BPP) is the class of decision problems solvable by a PTM in polynomial time with an error probability of less than 1/3 in all instances [9].

Since quantum mechanics are probabilistic in nature, we can define quantum complexity classes in a similar way as the probabilistic complexity classes.

1.1.3 Quantum complexity theory

Similar to classical complexity classes, quantum complexity classes group together problems that can be solved within certain bounds with a quantum computer. While classical computing uses a Turing machine model to assess the difficulty of computational problems, the quantum counterpart is based on the Quantum Turing machine model (QTM). A common QTM model, in essence, uses a classical deterministic TM with an extra tape for the qubits, as shown in Figure 1.4. The qubit tape is an infinite series of qubits, with one qubit per square, each initialized to the zero state. We can define a number of tape heads scanning the qubit tape to perform operations on the

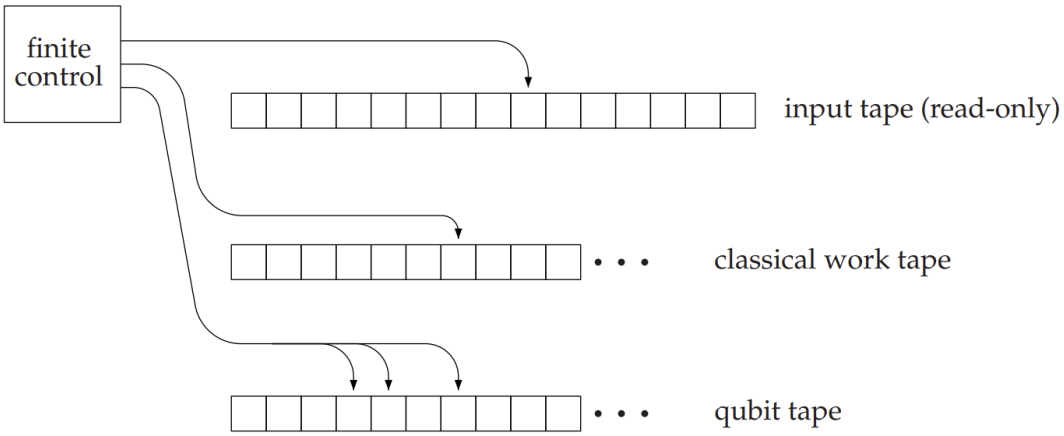

Figure 1.3: An example of a quantum Turing machine model that uses an extra tape for the quantum part. [9]

corresponding qubits. Having three tape heads allows for the QTM to model the universal gate set with up to three-qubit operations, and any more is usually unnecessary [9].

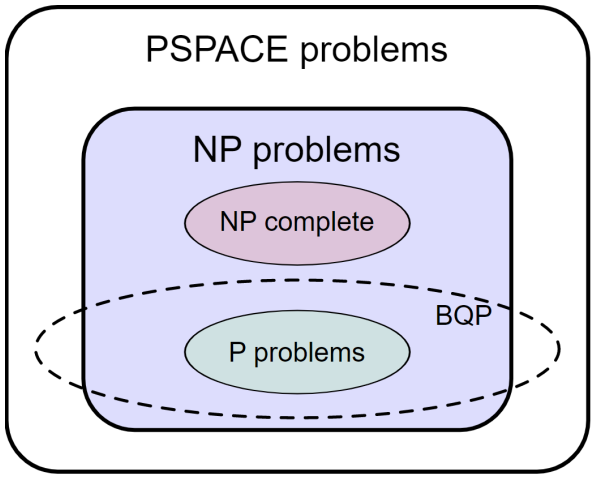

Figure 1.4: BQP is suspected (but not proven) to be within PSPACE, the class of problems that require a polynomial amount of space to be solved. Notice that it also intersects NP, and encompasses the entirety of P [10].

We can then define the quantum complexity class bounded-error quantum polynomial time (BQP) as the class of decision problems solvable by a QTM in polynomial time with an error probability of at most 1/3. This is the quantum counterpart of BPP, and BPP $\subseteq$ BQP [9].

The quantum computational analogue of NP is known as QMA, short for quantum Merlin-Arthur, and is based on the idea of a quantum proof [9]. A quantum proof is

a quantum state that plays the role of a certificate or witness to a quantum computer that functions as a verification procedure [9]–essentially a solution to the computational problem that needs to be verified. QMA is then defined as the class of problems whose solution (being in the form of a polynomial-sized quantum state) can be verified by a quantum computer in polynomial time, with accuracy of at least 2/3 [9]. We notice that NP $\subseteq$ QMA; however, there are some problems in QMA not known to be in NP, such as the local Hamiltonian problem, the quantum analogue to the classical MAX-SAT problem [9].

Since quantum computing results are probabilistic in nature, it is important to ensure that the solutions we obtain are within specific error bounds. The error bound of 1/3 is simply a convention, since it is strictly greater than 1/2 [9]. Higher accuracy is desirable, if possible.

1.2 Traveling salesman problem

The traveling salesman problem (TSP) is a well-known optimization problem. In essence, the problem is as follows:

> Given a list of cities and the distances between them, what is the shortest path through all cities (and returning to the starting city) that minimizes the total distance travelled?

TSP is an NP-hard problem in combinatorial optimization, so it is as at least as hard as the hardest problems in NP. TSP also has a decision version, which is answered by "yes" or "no":

> Given a list of cities and the distances between them, is there a path through all the cities (and returning to the starting city) with a total length $\leq k$, for some positive value k?

The decision version of TSP is an NP-complete problem. This means that the solution to the problem can be verified in polynomial time, the solution can be obtained in polynomial time with a non-deterministic TM model, and also that this problem can be used to simulate any other problem that is verifiable in polynomial time.

TSP can be presented as a graph problem, with the vertices representing the cities and weighted edges representing the distances between the cities (example shown in Figure 1.5). The graph does not necessarily need to be symmetrical. While a symmetrical graph can represent distances (the distance between city A and city B is the same as the distance between city B and city A), a non symmetrical graph can represent cost for travel (travelling from city A to city B could have a different cost

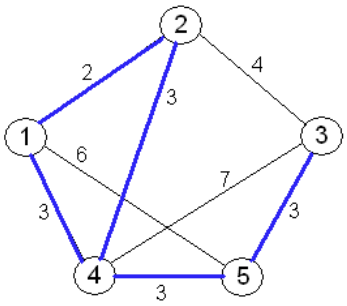

Figure 1.5: An example of a TSP instance with 5 cities, with the optimal route shown in blue.

than vice versa). The graph may or may not be a complete graph. A complete graph is one such that every vertex has an edge to every other vertex. Many algorithms to solve TSP create a representation of the problem as a complete graph, and give a sufficiently high weight to any edges not included in the original problem to discourage a path through that edge [11].

A typical way to represent a TSP graph is with $G = (V, E)$, where G is the name of the graph, V is a set of vertices, and E is the set of edges. The weights of the edges can be described with $w(u, v)$, where (u, v) denotes the edge connecting vertex u to vertex v [1].

2. Quantum Computing

In a nutshell, a quantum computer utilizes the laws of quantum mechanics to perform computations. Rather than a direct upgrade to classical computers, quantum computers solve problems differently from classical computers, and are so far only known to be useful for certain types of problems. Quantum computing, in theory, is useful for solving problems that search through huge search spaces of combinations. A classical computer would need to test each possible solution one by one, while a quantum computer is able to work with many possible solutions simultaneously to potentially arrive at a solution much quicker [12].

There are several different kinds of quantum computers, including the universal gate-based quantum computer and the quantum annealer. While both kinds are useful for solving problems with a huge search space, a quantum annealer is only useful for solving an even more specific kind of problem: optimization problems [13]. Since the TSP involves finding the optimal route out of an enormous number of possible routes, quantum annealing can potentially provide us with a speedup.

Quantum gates are the operations a quantum computer performs on quantum data (e.g. quantum bits), and a set of quantum gates is considered universal if any unitary operation (defined in Section 2.2) on the quantum data can be approximated arbitrarily well with a quantum circuit involving only the gates in the set [10]. A quantum computer equipped with a universal set of quantum gates is called a universal quantum computer [14].

The postulates of quantum mechanics are the foundation for quantum computing.

2.1 Single qubit states

A quantum bit, or qubit, is a single unit of memory in a quantum computer. The representation of a qubit is different from a classical bit: rather than discrete 0's and 1's, a qubit can be represented by a state within a Hilbert space.

Postulate 1. *A complex vector space with inner product (i.e. Hilbert space) is associated to any isolated physical system. This is called the state space of the system, and*

the system is completely described by its state vector, which is a unit vector in the state space. [10]

Postulate 1 describes the theoretical representation of a qubit as a 2-dimensional vector in a complex Hilbert space $\mathbb{C}^2$. The most commonly used basis within this Hilbert space is the computational basis, with the vectors $|0\rangle$ and $|1\rangle$:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.1}$$

The pure state of a single qubit can be written as a normalized linear combination of the basis states as such:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.2}$$

for any $\alpha, \beta \in \mathbb{C}^2$ such that $|\alpha|^2 + |\beta|^2 = 1$.

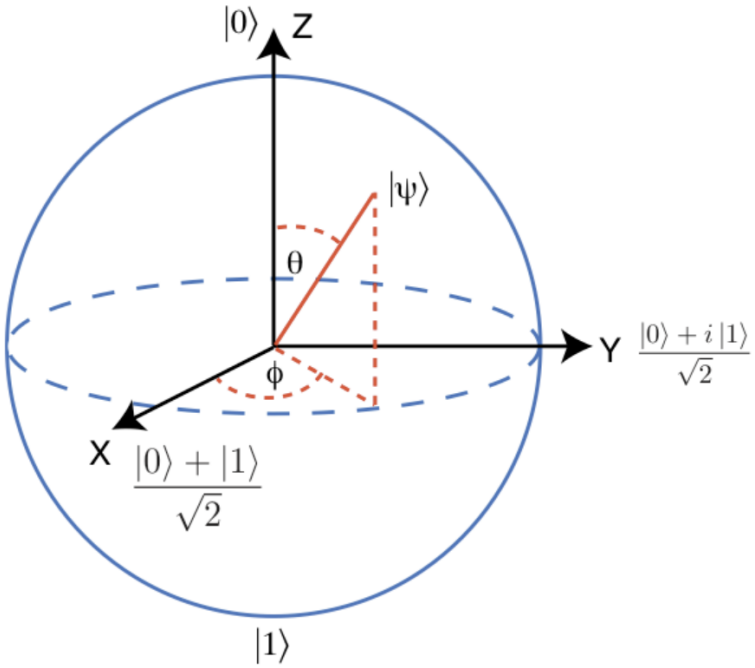

Figure 2.1: A single qubit state can be represented by a point on the Bloch sphere. [14]

A good way to visualize a qubit is with the help of the Bloch sphere, a unit 2-sphere with antipodal points corresponding to mutually orthogonal state vectors [10]. The computational basis states $|0\rangle$ and $|1\rangle$ are mapped to the opposite ends of the Z axis, as shown in Figure 2.1. The radius of the sphere is 1. Since qubit states are normalized, the state of a single qubit can be mapped to a point on or within the surface of the Bloch sphere. The general pure state formula for the Bloch sphere representation is

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle\,. \tag{2.3}$$

On the physical side, a qubit usually consists of an isolated and controlled particle. Some properties of the particle can be measured and are used as the states of the qubit. For example, if the particle is an electron, its spin property can be used: spin up would be represented as the $|0\rangle$ state and spin down would be represented as the $|1\rangle$ state. For a photon, the vertical and horizontal polarization can be used for the basis states [10].

2.2 Quantum gates

The operations that can be performed on a qubit are explained by Postulate 2:

Postulate 2. *A unitary transformation describes the evolution of a closed quantum system, so the state of the system $|\psi\rangle$ at time t_1 is related to state $|\psi'\rangle$ at time t_2 by a unitary operator U which depends only on the times t_1 and t_2,*

$$|\psi'\rangle = U|\psi\rangle \tag{2.4}$$

The Schrödinger equation is another way to describe the evolution of a closed quantum system:

$$i\hbar\frac{d|\psi\rangle}{dt} = H|\psi\rangle \tag{2.5}$$

where H is the Hamiltonian, a fixed Hermitian operator that represents the energy of the system. [10]

Another definition for a unitary operator is that it is a transformation on a vector within the Hilbert space that preserves the inner product, or the "length" of the vector within the Hilbert space [10].

In the universal gate-based quantum computing model, the unitary transformations are called gates, and can be written simply as unitary matrices. On the Bloch sphere, the unitary operators change the position of the state vector. To apply the gate to a qubit, we can perform matrix multiplication of the gate's matrix to the state vector, with the matrix on the left side of the vector.

One of the properties of a unitary operator is that its inverse is equal to its Hermitian adjoint (also called conjugate transpose), denoted by the dagger symbol †. For a unitary operator U and its adjoint $U^\dagger$, we have

$$UU^\dagger = U^\dagger U = I \tag{2.6}$$

where I is the identity operator [10]. Since a unitary operator preserves inner product, applying a unitary operator to a single qubit state will not make the qubit vector leave the surface of the Bloch sphere [10].

These unitary operators are made into quantum gates in the universal quantum computer. Some important gates are the Pauli gates, the Hadamard gate, the I gate, and the P, S, T, and U gates.

2.2.1 Pauli gates

The Pauli gates are the X, Y, and Z gates, and rotate the qubit state by π radians about the respective axis on the Bloch sphere [14].

The matrix representation for the X gate is

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0|. \tag{2.7}$$

In terms of the computational basis, the X gate transforms the state $|0\rangle$ into $|1\rangle$ and the state $|1\rangle$ into $|0\rangle$. Because of this logical inversion in the computational basis, the X gate is often also referred to as the NOT gate.

The Y gate can be written as

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = -i\,|0\rangle\langle 1| + i\,|1\rangle\langle 0|, \tag{2.8}$$

and the Z gate as

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = |0\rangle\langle 0| - |1\rangle\langle 1|. \tag{2.9}$$

Notice that the Z gate does not change the state of the qubit if it is in one of the computational basis states. This is because the states $|0\rangle$ and $|1\rangle$ are the eigenstates of the Z gate. The computational basis is sometimes called the Z basis for this reason [14].

In addition to the computational basis, another popular basis is based on the eigenstates of the X gate: the X basis (also known as the Hadamard basis, and we will see why shortly). This basis consists of the states

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \text{ and } |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}. \tag{2.10}$$

2.2.2 Hadamard gate

The Hadamard gate, or H gate, creates an equal superposition of the computational basis states $|0\rangle$ and $|1\rangle$, and places the state vector away from the poles of the Bloch sphere. The matrix form of the Hadamard gate is

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{2.11}$$

This is a rotation of the state vector around the vector $\begin{bmatrix}1, & 0, & 1\end{bmatrix}$ (the line between the X and Z axis) on the Bloch sphere [14].

The Hadamard gate transforms the computational basis into the X basis. This transformation can also be reversed by applying the Hadamard gate again [10]. The equal superposition means that measuring a state in the Hadamard basis gives a 50% chance of measuring 0 and a 50% chance of measuring 1. We will discuss quantum measurement in a future section.

Superposition is incredibly useful for quantum computing. Being able to be in a combination of basis states allows us to encode much more information in one qubit than in one classical bit. Superposition is the main mechanic that gives quantum computing an advantage over classical computing.

2.2.3 I, P, S, T gates

The I gate is the identity gate, and makes no change to the qubit state. The I gate applied to a single qubit can be written in matrix form as the identity matrix,

$$I = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}. \tag{2.12}$$

It can be used in calculations, for example when showing that two gates are inverses of each other [14].

The P gate, also called the phase gate, requires a real-valued parameter ϕ for its definition. The matrix form of the P gate is:

$$P(\phi) = \begin{bmatrix}1 & 0\\ 0 & e^{i\phi}\end{bmatrix}. \tag{2.13}$$

This gate rotates the qubit state by ϕ radians around the Z axis [14].

A commonly used ϕ value for the P gate is $\phi = \pi/2$, giving us the S gate. The matrix representation of the S gate is:

$$S = \begin{bmatrix}1 & 0\\ 0 & e^{i\pi/2}\end{bmatrix}. \tag{2.14}$$

Another name for the S gate is the $\sqrt{Z}$ gate, since applying it twice will give us the same transformation as applying the Z gate once [14]. Notice that the S gate is not its own inverse; its inverse, denoted as $S^\dagger$, is the P gate with $\phi = -\pi/2$, as such [14]:

$$S^\dagger = \begin{bmatrix}1 & 0\\ 0 & e^{-i\pi/2}\end{bmatrix}. \tag{2.15}$$

Since the definition of a unitary operator states that its adjoint is equal to its inverse, we can show that for a qubit state $|\psi\rangle$, we have

$$SS^\dagger |\psi\rangle = S^\dagger S |\psi\rangle = I |\psi\rangle = |\psi\rangle . \tag{2.16}$$

The T gate is another gate that is a variation of the P gate with a specific parameter value, this time with $\phi = \pi/4$ [14]. The T gate and its adjoint are written as:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}, T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix} \tag{2.17}$$

Similar to the S gate, the T gate is sometimes called the $\sqrt[4]{Z}$ gate, since applying it four times is the same as applying a Z gate once [14].

2.2.4 U gate

The U gate is the most general of all single-qubit quantum gates, and is parameterized by the angles of rotation around each of the three axes [14]. It takes the form

$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda} \sin\left(\frac{\theta}{2}\right) \\ e^{i\phi} \sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)} \cos\left(\frac{\theta}{2}\right) \end{bmatrix} . \tag{2.18}$$

We can show that, for example, the H and P gates are equivalent to specific cases of the U gate [14]:

$$U(\pi/2, 0, \pi) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = H, \text{ and} \tag{2.19}$$

$$U(0, 0, \lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix} = P. \tag{2.20}$$

We can use the U gate to construct any other single-qubit gate. This is extremely useful for many quantum algorithms, such as phase estimation.

2.3 Quantum measurement

In order to get usable results from qubits, which we can see have an infinite number of possible states, we need to measure them. Postulate 3 explains how quantum measurement works.

Postulate 3. *Quantum measurements are described by a collection of measurement operators $\{M_m\}$ acting on the state space of the system, such that a measurement gives outcome m with probability*

$$p(m) = (M_m |\psi\rangle , M_m |\psi\rangle) = \langle\psi| M_m^\dagger M_m |\psi\rangle . \tag{2.21}$$

The state after measuring the observed value m is

$$\frac{M_m |\psi\rangle}{\sqrt{\langle\psi| M_m^\dagger M_m |\psi\rangle}}. \tag{2.22}$$

The measurement operators satisfy the completeness equation,

$$\sum_m M_m^\dagger M_m = I, \tag{2.23}$$

and the sum of the probabilities is also 1. [10]

The measurement operators are Hermitian, or self-adjoint, so $M_m^\dagger = M_m$. As mentioned before, the most common measurement basis is the computational basis, the basis consisting of the vectors $|0\rangle$ and $|1\rangle$. The computational basis measurement operators are then $M_0 = |0\rangle\langle 0|$ and $M_1 = |1\rangle\langle 1|$, and the probability of measuring $m \in \{0, 1\}$ is

$$p(m) = \langle\psi| M_m^\dagger M_m |\psi\rangle = \langle\psi| M_m M_m |\psi\rangle = \langle\psi| M_m |\psi\rangle \tag{2.24}$$

since M_m is a projector (an operator that satisfies $M^2 = M$) in the computational basis [10].

A state in equal superposition has a uniform probability distribution for the measurements in the computational basis [10]. A simple way to achieve this is to apply the Hadamard gate to a qubit in one of the computational basis states. After that, if the qubit is measured, there is a 50% chance of measuring a 0 and a 50% chance of measuring a 1. In simple terms, when a state is measured, its state is flattened or projected onto the measurement basis, and we obtain an outcome with a probability that depends on the state of the qubit prior to measuring.

We can see that we lose information when we measure a qubit, since we can only observe one of two possible results. If the qubit was in superposition prior to measurement, the state of the qubit is unavoidably changed. Hence, we want to make sure we have performed all the operations we wanted to perform on a qubit before we measure it.

2.4 Multiple qubit states

Postulate 4 explains the mathematical representation of multiple-qubit states.

Postulate 4. *The state space of a composite system is the tensor product of the state spaces of the component systems. Thus, if the component systems are numbered 1 through n, and system number i is prepared in the state $|\psi_i\rangle$, then the joint state of the total system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle$. [10]*

Hence, a state consisting of n qubits can be represented by a vector in a 2^n-dimensional Hilbert space, with the help of the tensor product. The computational basis for a two-qubit state then would be:

$$\begin{aligned}
|00\rangle = |0\rangle \otimes |0\rangle &= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \\
|01\rangle = |0\rangle \otimes |1\rangle &= \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \\
|10\rangle = |1\rangle \otimes |0\rangle &= \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \\
|11\rangle = |1\rangle \otimes |1\rangle &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.
\end{aligned} \tag{2.25}$$

Then, the general two-qubit state can be written as:

$$|a\rangle = a_{00}\,|00\rangle + a_{01}\,|01\rangle + a_{10}\,|10\rangle + a_{11}\,|11\rangle = \begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix}, \tag{2.26}$$

with $|a_{00}| + |a_{01}| + |a_{10}| + |a_{11}| = 1$ [10].

We can see that the information that can be stored by multiple qubits is exponential, since a superposition of n qubits allows for the quantum computer to operate on 2^n combinations at once [10]. The amount of information that 500 qubits can represent would not be possible to represent with even more than 2^{500} classical bits [15].

A state with multiple qubits cannot always be represented with a Bloch sphere for each qubit, however. An entangled state (also called an inseparable state) is one such that cannot be expressed as a combination of two separate qubit states. The Bell states are examples of entangled two-qubit states [10]:

$$|\beta_{00}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}; \tag{2.27}$$

$$|\beta_{01}\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}}; \tag{2.28}$$

$$|\beta_{10}\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}}; \tag{2.29}$$

$$|\beta_{11}\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}. \tag{2.30}$$

An interesting property of entanglement is that the outcome of the measurement of one qubit affects the result of the other. For example, for the Bell state $|\beta_{00}\rangle$, there is a 50% chance of measuring 0 in both qubits and a 50% chance of measuring 1 in both qubits–and 0% chance of measuring any other combination of results [10]. This is a curious mechanic in quantum physics: for entangled particles, the outcome of one measurement is directly linked to the outcome of the other measurement, even with a physical distance between them. There is no way to know prior which outcome both measurements will take.

2.5 Quantum algorithms

2.5.1 Controlled gates

In addition to the various single-qubit gates, there are quantum gates that handle multiple qubits at once. Probably the most important class of multi-qubit gates are the controlled gates. The Controlled-NOT (CNOT) gate is a two-qubit gate that uses the first qubit as the control and applies the NOT gate (also known as the X gate) on the second qubit if the first qubit is in the $|1\rangle$ state, and makes no change if it is in the $|0\rangle$ state [10]. This gate is less simple if the controlled qubit is in a state that is not one of the computational basis states, though. The matrix representation of CNOT is [10]:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{2.31}$$

The CNOT gate transforms the state $|10\rangle$ into $|11\rangle$ and the state $|11\rangle$ into $|10\rangle$, and leaves the states $|00\rangle$ and $|01\rangle$ unchanged.

The CNOT gate acting on one qubit with two other qubits as control is called

the Toffoli gate (or CCNOT), and is represented by the matrix [10]:

$$CCNOT = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \tag{2.32}$$

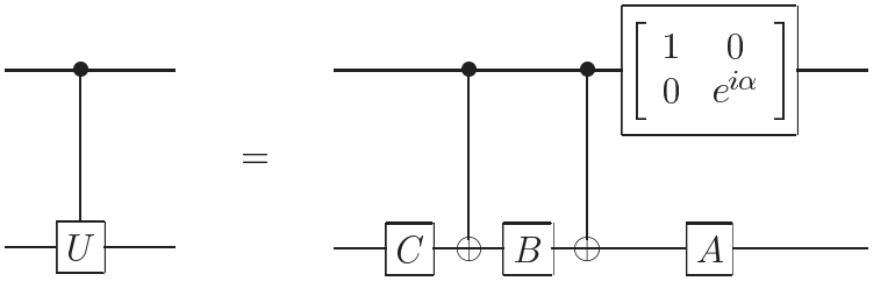

Figure 2.2: The circuit for the CU gate can be decomposed into single-qubit gates and CNOT gates. [10]

We can make any single-qubit gate U into a controlled-U gate by decomposing U into the form $U = e^{i\alpha}AXBXC$, where A, B, and C are single qubit operations such that $ABC = I$, and $e^{i\alpha}$ is some overall phase shift [10]. We then write the controlled-U gate as $CU = (P(\alpha) \otimes A)(CNOT)(I \otimes B)(CNOT)(I \otimes C)$. Notice that we turn the single-qubit X gate into the controlled-X (CNOT) gate and construct a P gate using the phase shift. Figure 2.2 shows the circuit implementation of the controlled-U gate. We will discuss quantum circuits and their representations shortly.

2.5.2 Quantum Fourier Transform

The quantum Fourier transform (QFT) is the key to many important quantum algorithms, including phase estimation. QFT is an efficient algorithm for performing a discrete Fourier transform of quantum mechanical amplitudes [10].

The discrete (classical) Fourier transform transforms a function into one that is easier to manage. It acts on a vector $(x_0, \ldots, x_{N-1})$ and maps it to the vector $(y_0, \ldots, y_{N-1})$ by the following formula [14]:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega_N^{jk}, \tag{2.33}$$

where $\omega_N^{jk} = e^{2\pi i \frac{jk}{N}}$ [14].

The QFT acts on a quantum state $|X\rangle = \sum_{j=0}^{N-1} x_j |j\rangle$ and maps it to the quantum state $|Y\rangle = \sum_{k=0}^{N-1} y_k |k\rangle$ according to the same formula [14]:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega_N^{jk}, \tag{2.34}$$

with the same definition for ω_N^{jk} [14]. This transformation affects only the amplitudes of the state [14].

The unitary matrix representation of the QFT is [14]:

$$U_{QFT} = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} \omega_N^{jk} |k\rangle \langle j| . \tag{2.35}$$

Applying the QFT transforms a quantum state from the computational (Z) basis to the Fourier basis (also known as the Hadamard basis, or X basis). The Hadamard gate is a single-qubit QFT, and transforms the computational basis states $|0\rangle$ and $|1\rangle$ to the Hadamard basis states $|+\rangle$ and $|-\rangle$, respectively [14].

Since quantum gates are unitary operators, all gates have an inverse that can reverse the operations performed on the qubits. The inverse quantum Fourier transform reverses these changes to the quantum state, and puts qubits in the Hadamard basis back into the computational basis [10].

2.5.3 Quantum circuits

Qubits, quantum gates, measurements, and resets are all components that can be used to build a quantum circuit [14]. A quantum circuit is a computational routine consisting of coherent quantum operations on quantum data such as qubits, and concurrent real-time classical computation, and any quantum program can be represented by a sequence of quantum circuits and non-concurrent classical computation [14].

Figures 2.2 and 2.3 show two examples of quantum circuits. The circuit in Figure 2.3 is the circuit for quantum teleportation. The circuit is interpreted from left to right, and each vertical segment is applied in order of appearance. This circuit handles three qubits, q_0, q_1, and q_2, and two classical bits, crz and crx. Qubit q_0 is initialized to some state $|\psi\rangle$, and q_1 and q_2 are initialized to $|0\rangle$, which is the conventionally default starting state. Then the Hadamard gate is applied to q_1, and then CNOT is applied to q_2 with q_1 as the control. The circuit then performs another CNOT, this time on q_1 with q_0 as the control, and then the Hadamard gate on q_0. We then measure q_0 and q_1 and store the measurement information in the classical bits crz and crx, respectively. Finally, we perform a controlled-X operation on q_2 using crx as the control (i.e. if crx is 0, we make no change, else we apply the X gate to q_2), and then controlled-Z on q_2 with crz

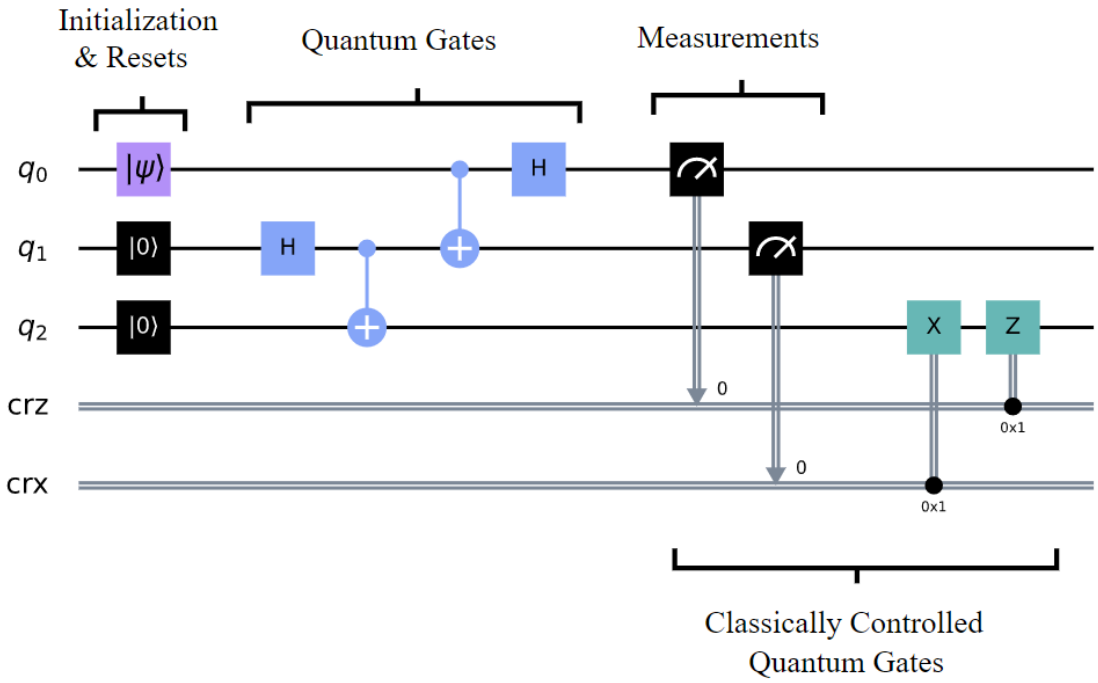

Figure 2.3: An example quantum circuit with three qubits (q_0, q_1, and q_2) and two classical bits (crz and crx). This circuit performs an algorithm called quantum teleportation. [14]

as control. This is the circuit for the quantum teleportation algorithm–the initial state of q_0, $|\psi\rangle$, is "teleported" to qubit q_2 [14]. By the end of the algorithm, we now have q_2 in the state $|\psi\rangle$, and the states are q_0 and q_1 are unimportant. Quantum mechanics do not allow us to create exact copies of states (shown by the no-cloning theorem [10]), so the quantum teleportation protocol is a way to move quantum information from one qubit to another [10].

Typically, a quantum circuit begins with initialization, followed by a series of quantum gates, and ends with measurement. Algorithms that utilize quantum computing usually begin with a classical procedure to translate the problem into something a quantum computer can work with, then run the appropriate quantum circuit(s), and then classically process the results of the quantum part [14].

2.6 Quantum annealing

Quantum annealing is a different quantum computing method that utilizes the adiabatic theorem and other areas of quantum mechanics to find the optimal solution within a large search space. The adiabatic theorem states that the Hamiltonian H_0 of a quantum system in the ground state can be evolved into a different Hamiltonian H_P

without leaving the ground state via the formula

$$H(t) = \left(1 - \frac{t}{T}\right) H_0 + \frac{t}{T} H_P \tag{2.36}$$

assuming a large enough time T, and assuming that H_0 and H_P do not commute [13]. This is useful because we can set up the Hamiltonian H_0 to have a ground state which is easy to prepare and find. H_P is then the Hamiltonian whose ground state we are trying to find to solve the computational problem [16].

The method of using quantum annealing to solve optimization problems is called adiabatic quantum optimization (AQO) [16]. This typically involves mapping the optimization problem to a problem easily solvable with a quantum annealer (such as the Ising problem, to be discussed later), utilizing the machine to find the optimal solution, and then decoding the results to find the optimal solution to the original problem.

Quantum annealers do not utilize quantum gates in the way universal gate-based quantum computers to, and have differently implemented qubits. This limits the types of problems solvable with a quantum annealer.

2.7 NISQ

The current state of existing quantum computers is described as the Noisy Intermediate-Scale Quantum (NISQ) era. As of today, the largest set of connected qubits equipped with universal quantum gates is IBM's 127-qubit Eagle processor, unveiled in November 2021 [17]. Although this is a huge milestone in regards to quantum technology, a quantum computer of this size is not enough to change the world in a practical way. It is, however, a useful and vital step towards improving our technology's computational capabilities [18].

In addition to such a low number of connected qubits, universal gate-based quantum computers currently struggle with noise: the optimal conditions required for noise-free quantum computing are extremely difficult to achieve, if not completely impossible [18]. The potential error within the quantum system scales with the number of connections between the qubits, as well as the number of operations performed [10, 18].

Quantum annealers have been around for longer than universal gate-based quantum computers. The current state-of-the-art quantum annealer is a D-Wave machine named "Pegasus", with a quantum processor chip equipped with 5,000 low-noise qubits, released in February 2019 [13]. This type of quantum computer, however, is only useful for certain types of problems. We often need to restructure computational problems in order to make the most of quantum annealers.

In spite of these practical obstacles, we can explore the theoretical benefits of quantum technology. Quantum computing can potentially give us significant improve-

ments for solving computationally difficult problems such as the TSP.

3. Solving TSP classically

For the following sections, let n be the number of vertices, or cities, in the arbitrary TSP instance. Let m be the number of edges. Assume the TSP graph is directed and complete, which gives us the general worst-case instance since it gives the greatest amount of information to search for a solution in.

3.1 Exact solutions

The following algorithms find the exact optimal tour for the given TSP instance. These algorithms perform slower than ones that find "good enough" solutions, but guarantee to output the best solution possible.

3.1.1 Brute-force algorithm

The brute-force algorithm tests out all possible solutions and compares them to find the shortest path. This takes $\mathcal{O}(n!)$ time, since there are $n!$ permutations of all vertices and it takes linear time to calculate the total path distance of each permutation (simply summing the weights of the edges between each vertex in the permutation) [19]. This algorithm takes $\mathcal{O}(n^2)$ space, which is the space needed to store the graph itself [20]. It doesn't require any significant amount of extra workspace.

3.1.2 Held-Karp algorithm

The Held-Karp Algorithm (also known as the Bellman-Held-Karp Algorithm) is a dynamic programming approach to solving TSP, so it takes advantage of previous calculations in a recursive manner in order to reach a solution quicker [1]. This algorithm works as follows:

The vertices are numbered $1, 2, ..., n$, with some vertex numbered as 1 being the starting point. The starting point doesn't matter, since the solution returns to the starting point to make a cycle [19]. The next set of steps calculate for each set of vertices $S \subseteq \{2, ..., n\}$ and every vertex $e \neq 1$ not contained in S, the shortest one-way

path from vertex 1 to vertex e that passes through all cities in S but not through any city outside of S. Call this distance $g(S, e)$, with $d(u, v)$ denoting the length of the direct edge from vertex u to vertex v. For the set $S = \emptyset$, $g(S, e) = g(\emptyset, e) = d(1, e)$, so simply the length of the edge from vertex 1 to vertex e. For sets S that contain only one element, for example $S = \{a\}$, we have $g(S, e) = g(\{a\}, e) =$ the length of the path $1 \to a \to e$. Once we get to sets S that contain two or more elements, we need to consider multiple path options and select the shortest. For example, for $S = \{a, b\}$, we have $g(S, e) = g(\{a, b\}, e) =$ the shorter of the two possible paths $1 \to a \to b \to e$ or $1 \to b \to a \to e$. We can use the results of the calculations for sets containing k elements to quickly find the results for sets containing $k + 1$ elements. For example, if we know that path $1 \to a \to b \to c$ is shorter than $1 \to b \to a \to c$, then the path $1 \to a \to b \to c \to e$ is certainly shorter than $1 \to b \to a \to c \to e$ [19] [21].

In general: Let $S \subset V \backslash \{x_1\} \equiv \{x_2, x_3, \ldots, x_n\}$ be a subset of size s, with $1 \leq s \leq n - 1$. For each vertex $x_i \in S$, we define $\text{cost}(x_i, S)$ as the length of the shortest path from x_1 to x_i which travels to each of the remaining vertices in S once. Specifically, we do this by implementing the following recursion formula [19]:

$$\text{cost}(x_i, S) = \min_{x_j} \{\text{cost}(x_j, S \backslash \{x_i\}) + D_{ji}\}, \tag{3.1}$$

where $x_j \in S \backslash \{x_i\}$. For the case where S only contains one element, define [19]:

$$\text{cost}(x_i, S) = D_{1i}. \tag{3.2}$$

The Held-Karp algorithm takes $\mathcal{O}(2^n n^2)$ time and $\mathcal{O}(n 2^n)$ space [22]. This is much faster than the brute force algorithm, but it uses more space by a factor of 2^n.

This is a significant speedup over the brute-force algorithm, since it does not check absolutely every permutation of cities possible. Once it has found the shortest path through a subset of the cities, it does not need to check another permutation of that subset of cities when another city is added to the route [23]. The implementation of this algorithm into code is also very simple, since the recursion gives the overall problem the same structure as the smaller subproblems [21].

3.2 Approximate solutions

The following algorithms are approximation algorithms: they do not guarantee to find the exact optimal route through all cities, but instead find a solution that is "good enough" [20]. There is a trade-off between the resources required to arrive at a solution and the accuracy of the solution.

A way to assess the quality of the solution acquired via an approximation algorithm is with the use of the Held-Karp (HK) lower bound. The HK lower bound is

obtained by using a technique called Lagrangian relaxation (also known as subgradient optimization), and gives us a good approximation of the length of the optimal tour in a TSP instance [24]. The HK lower bound is, on average, within 0.8% of the length of the optimal route for random TSP instances with many thousands of cities [24], so it is a generally excellent estimator of the true optimal solution of a TSP instance. The HK lower bound is obtained by modifying the graph to obtain different minimum-1 trees to approximate the value of the optimal TSP tour.

3.2.1 Greedy algorithm

The first and perhaps most obvious approximation algorithm for solving TSP is the general greedy algorithm. A greedy algorithm is one that makes the choice that looks best at the moment, i.e. makes a locally optimal choice in hope that it will lead to a globally optimal solution [1]. Although the implementation of a greedy algorithm is typically straight forward, it does not always lead to the best results [1].

The general greedy algorithm for TSP involves adding the shortest edges to the tour one by one, until all cities are connected [25]. A way to do this is to sort all edges from shortest to longest. We then add the shortest edge that will neither create a situation where any single vertex has more than 2 edges, nor a cycle with less than the total number of cities. This process is repeated until we have a cycle containing all cities in the TSP [25].

We discover that for every number of cities $n \geq 2$, there is an instance of TSP for which this greedy algorithm results in the worst tour [26]. Clearly, this method is very far from optimal in all cases. However, it runs in a $\mathcal{O}(n^2 \log_2(n))$ time complexity, and is able to reach a solution typically within 15-20% of the HK lower bound [20].

3.2.2 Nearest-neighbor algorithm

The nearest-neighbor (NN) algorithm is another greedy or naive algorithm. It is also quite straight forward, and is incredibly simple to implement. The general idea is to keep moving to the nearest unvisited city until all cities are visited.

In detail, algorithm is as follows:

1. Begin with all cities set to "unvisited". We will need to keep track of which cities we have already visited.

2. Select an origin city.

3. Select the closest neighboring unvisited city by comparing the edges originating at current city to any unvisited neighboring cities. If all cities have been visited,

connect the current city back to the starting city and end the program. Otherwise, connect the newly found city to the current city.

4. Repeat Step 3 for every new city added to the path until all cities have been visited and are connected [27].

The NN algorithm runs in worst-case $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ workspace (in addition to the $\mathcal{O}(n^2)$ space required to store the input graph), and obtains a solution within 25% of the HK lower bound [27].

A variation of this algorithm is the repetitive nearest-neighbor algorithm (RNN), which performs the NN algorithm for every starting vertex, and selects the best tour obtained [26]. The time complexity for RNN is hence $\mathcal{O}(n^3 \log_2(n))$, since NN is repeated $\mathcal{O}(n)$ times (once for every vertex). The best route found by RNN will be better than at least $n/2 - 1$ other routes [27].

3.2.3 Christofides algorithm

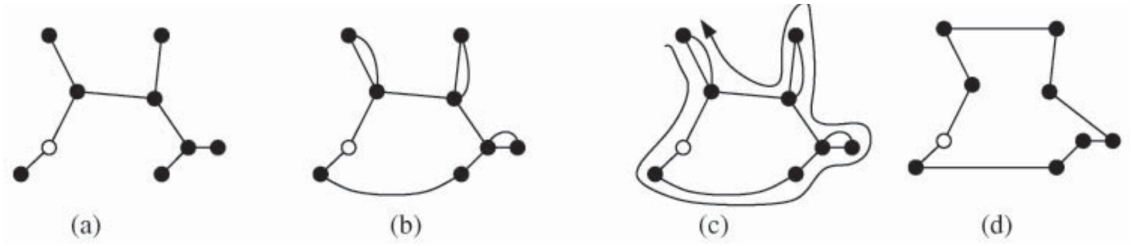

Figure 3.1: The Christofides algorithm in action. (a) shows a minimum spanning tree M, (b) shows the minimum-weight perfect matching P on the vertices in W (with the curved edges being the ones in P), (c) shows the Eulerian circuit C, and (d) shows the final tour for the TSP [28].

The Christofides algorithm works for cases of TSP that are symmetric (edge from vertex a to vertex b is the same weight as the edge from vertex b to a) and follow the triangle inequality [25]. The triangle inequality states that for every three vertices a, b, and c, the weights w of the edges between them must satisfy $w(a, b) + w(b, c) \geq w(a, c)$ [27]. This is a typical feature in an x-y metric space coordinate plane [25]. The algorithm is as follows [28]:

1. Construct a minimum spanning tree M from the graph G representing the TSP instance, as shown in part (a) of Figure 3.1. A minimum spanning tree is a subset of edges of a connected, undirected, edge-weighted graph that connects all vertices together without any cycles, such that the total weight of the edges is minimized [1].

2. Let W be the set of vertices of G that have odd degree in M, and let H be the subgraph of G induced by the vertices in W. So then H is the graph that has W as its vertices, and all edges from G that join these vertices.

3. Find a minimum-weight perfect matching P in the subgraph consisting of the vertices from H. A perfect matching is a set of all pairwise non-adjacent edges such that all vertices of the graph are incident to an edge in the matching, and a minimum-weight perfect matching is one such that the total weight of the edges in the matching are minimized [1].

4. Combine the graphs M and P to create a graph G' without combining parallel edges into single edges–allow G' to have two copies of the same edge, if that is the result (illustrated in part (b) of Figure 3.1).

5. Create a Eulerian circuit C in G', which is a circuit that visits each edge exactly once, as shown in part (c) in Figure 3.1.

6. Convert C into a tour by skipping over previously visited vertices (part (d) in Figure 3.1). [28]

A minimum spanning tree (Step 1) can be found in $\mathcal{O}(m \log n)$ time or faster [29], which along with the rest of the steps are relatively insignificant to Step 2, which is the most time-consuming step of the algorithm. Step 2 takes $\mathcal{O}(n^3)$ time [28]. However, despite being slower than some other approximation algorithms, the Christofides algorithm guarantees a TSP solution within 3/2 of the total weight of the optimal tour [28].

3.2.4 Yatsenko's algorithm

Yatsenko gives an interesting approach to solving TSP. The steps are as follows [30]:

1. Construct a route with three points. Two of the points are the farthest apart on the graph in terms of the total edge weight between them, and the third point is added to connect all three to maximize the sum of the distance between the three.

2. Add points one at a time as such: for each edge on the route, select a third point whose addition to the route would change the length of the route by the smallest amount (this change of length is called disturbance, and is equal to the sum of the two added edges minus the removed edge). Then, out of the thus selected third points (one per edge), add to the route the one whose addition creates the greatest disturbance.

3. Repeat Step 2 until all points are added to the route. [30]

Although Yatsenko claims that this algorithm gives an exact solution in polynomial time [31], numerous counterexamples have been given to show that this algorithm, in fact, does not guarantee useful results at all [30]. In order to solve the subproblems in order to optimize the tour obtained with this algorithm, the total running time of the algorithm would not be polynomial in the worst case [30].

3.2.5 Other algorithms

There are many other approximation algorithms for solving TSP, such as the Genetic algorithm [20], Ant Colony Optimization algorithm [27], Farthest Insertion algorithm [25], and others. The various approaches each balance the trade-off between resources needed (especially running time) and the accuracy of the results. However, we still notice that in order to guarantee the most optimal TSP tour, we require an algorithm with a much slower running time.

4. Solving TSP with quantum computing

There are many ways to approach TSP classically, ranging from iterating through permutations to breaking down the problem in certain ways. Quantum computing, which we know can be useful for optimizing and for searching for solutions through large search spaces, gives us a different way of tackling TSP.

4.1 Phase estimation

This is a quantum computing approach to solving TSP, utilizing a quantum computing technique called phase estimation.

One approach using phase estimation is described by the Qiskit Textbook [14]. In a nutshell, the method follows this framework:

1. Encode the edge weights of the TSP instance as phases.
2. Establish unitary operators in such a way that their eigenvectors are the computational basis states and eigenvalues are different combinations of the aforementioned phases.
3. Build a quantum circuit to apply phase estimation to specific eigenstates to compute the total distances for all routes.
4. Use a quantum search algorithm to find the minimum total distance and the route associated with it. [14]

We will use an example TSP instance of $n = 4$ cities with arbitrary distances between them to explain the algorithm.

4.1.1 Encoding the edge weights

Figure 4.1 shows a directed complete graph, with the vertices representing cities and the edges representing the cost or distance between cities. Since the graph is directed,

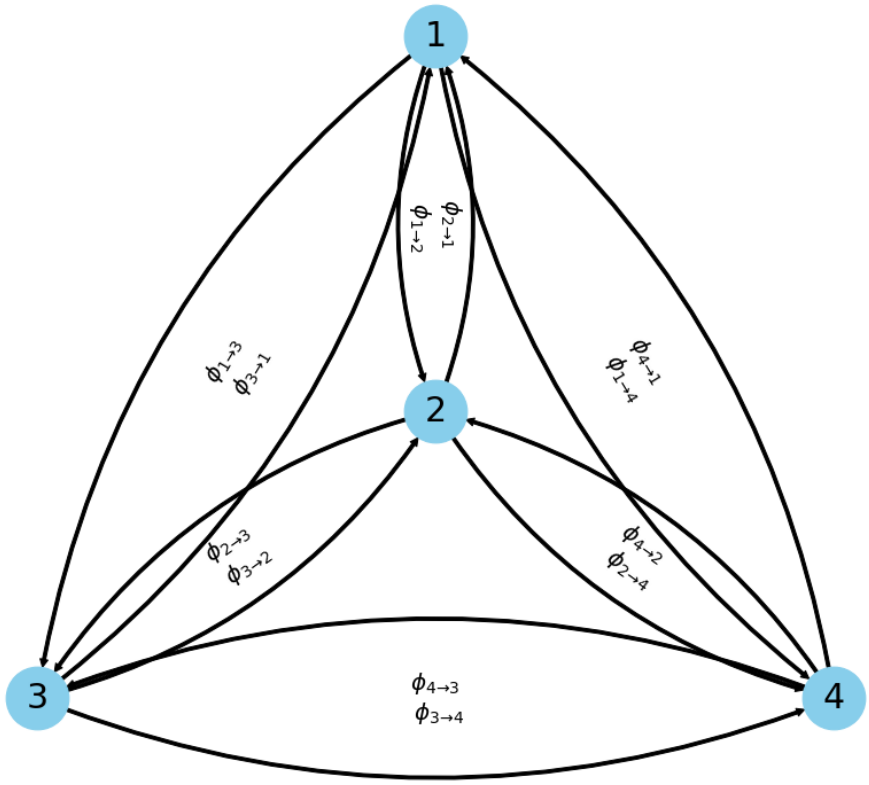

Figure 4.1: An example graph with 4 vertices and edges labeled with their phases. [14]

the weight from city a to city b can be different from the weight from city b to city a. This can be made into an undirected graph simply by making the weight from a to b equal to the weight from b to a.

The given distances are encoded as phases, so $\phi_{i\to j}$ is the cost from city i to city j, with $i, j \in [1, n]$.

We can represent the graph as an $n \times n$ matrix, with the element in index (i, j) being the phase (encoded weight) of the edge $i \to j$. For our example, this would give us matrix A [14]:

$$A = \begin{bmatrix} \phi_{1\to1} & \phi_{1\to2} & \phi_{1\to3} & \phi_{1\to4} \\ \phi_{2\to1} & \phi_{2\to2} & \phi_{2\to3} & \phi_{2\to4} \\ \phi_{3\to1} & \phi_{3\to2} & \phi_{3\to3} & \phi_{3\to4} \\ \phi_{4\to1} & \phi_{4\to2} & \phi_{4\to3} & \phi_{4\to4} \end{bmatrix} \tag{4.1}$$

Notice that all diagonal elements of A are 0 [14].

Since A is not guaranteed to be unitary, in order to construct unitary matrices, we first construct a matrix B using A by taking $e^{i\phi_{i\to j}}$ for every element (i, j) in A:

[14]

$$B = \begin{bmatrix} e^{i\phi_{1\to1}} & e^{i\phi_{1\to2}} & e^{i\phi_{1\to3}} & e^{i\phi_{1\to4}} \\ e^{i\phi_{2\to1}} & e^{i\phi_{2\to2}} & e^{i\phi_{2\to3}} & e^{i\phi_{2\to4}} \\ e^{i\phi_{3\to1}} & e^{i\phi_{3\to2}} & e^{i\phi_{3\to3}} & e^{i\phi_{3\to4}} \\ e^{i\phi_{4\to1}} & e^{i\phi_{4\to2}} & e^{i\phi_{4\to3}} & e^{i\phi_{4\to4}} \end{bmatrix} \tag{4.2}$$

The diagonal elements of B are 1 [14].

4.1.2 Constructing the unitary operators

Now we can construct a unitary matrix U_j from B for each city j as such [14]:

$$U_j = (\sum_{i=1}^{n} B[j][i] \times \text{outer product of all possible basis vectors}), \tag{4.3}$$

where $j, i \in [1, n]$. The rest of the elements in U_j are set to 0. As a result, U_j is a diagonal unitary matrix created from column j of B [14].

Using our example graph of 4 cities, we only need a basis for two qubits to construct U_j. Here is an example construction for U_1 [14]:

$$\begin{aligned} U_1 =& e^{i\phi_{1\to1}} |00\rangle\langle 00| + e^{i\phi_{2\to1}} |01\rangle\langle 01| + e^{i\phi_{3\to1}} |10\rangle\langle 10| + e^{i\phi_{4\to1}} |11\rangle\langle 11| \\ =& \begin{bmatrix} e^{i\phi_{1\to1}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & e^{i\phi_{2\to1}} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\ &+ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & e^{i\phi_{3\to1}} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & e^{i\phi_{4\to1}} \end{bmatrix} \\ =& \begin{bmatrix} e^{i\phi_{1\to1}} & 0 & 0 & 0 \\ 0 & e^{i\phi_{2\to1}} & 0 & 0 \\ 0 & 0 & e^{i\phi_{3\to1}} & 0 \\ 0 & 0 & 0 & e^{i\phi_{4\to1}}. \end{bmatrix} \end{aligned} \tag{4.4}$$

We combine these U_j matrices into one unitary matrix using the tensor product: $U = U_1 \otimes U_2 \otimes \cdots \otimes U_n$. Since each U_j is a $n \times n$ matrix (4×4 in our example) and each U_j is a diagonal matrix, U is a $n^n \times n^n$ diagonal matrix [32]. U has $(n-1)!$ eigenstates with eigenvalues being the total cost (encoded as phases) of the corresponding TSP tour [32]. We can normalize the phases to be in the range $[0, 2\pi]$ once we know the range of distances between the cities in the TSP instance, and then the phase estimation algorithm will find the eigenvalues of matrix U [32].

Sequence path	Eigenstate
1 - 2 - 3 - 4	$\|11000110\rangle$
1 - 2 - 4 - 3	$\|10000111\rangle$
1 - 4 - 2 - 3	$\|10001101\rangle$
1 - 4 - 3 - 2	$\|01001110\rangle$
1 - 3 - 2 - 4	$\|11001001\rangle$
1 - 3 - 4 - 2	$\|01001011\rangle$

Table 4.1: The possible solutions to our example TSP instance, excluding equivalent routes, and their corresponding eigenstates. [14]

4.1.3 Setting up the circuit

The total number of possible combinations of cities is $n!$, but since the TSP tours are cycles, there are $(n-1)!$ unique possible routes (since, for example, the route $1 \to 2 \to 3 \to 4$ is equivalent to the route $2 \to 3 \to 4 \to 1$) [32]. In our example with 4 cities, this gives us $(4-1)! = 3! = 6$ unique routes. Column 1 in Table 4.1 gives all the possible unique tours, with vertex 1 as the starting city. Additionally, if the graph is symmetrical, and the weight of edge (i, j) is the same as the weight of edge (j, i), then there are only $(n-1)!/2$ unique routes [14].

The paths are encoded into the computational basis states (which are also the corresponding eigenstates of the unitary matrix U that we constructed prior) via the formula: [14]

$$|\psi\rangle = \otimes_j \, |f(j) - 1\rangle \tag{4.5}$$

for j from 1 to n, and where the function $f(j)$ gives us the city from which we travelled to city j. Since the cities are numbered in decimal form, after calculating $f(j) - 1$, we need to convert the result to binary [14]. For example, for the path $1 \to 2 \to 3 \to 4$, we get [14]:

$$\begin{aligned} |f(1) - 1\rangle &= |4 - 1\rangle = |3_{10}\rangle = |11_2\rangle \\ |f(2) - 1\rangle &= |1 - 1\rangle = |0_{10}\rangle = |00_2\rangle \\ |f(3) - 1\rangle &= |2 - 1\rangle = |1_{10}\rangle = |01_2\rangle \\ |f(4) - 1\rangle &= |3 - 1\rangle = |2_{10}\rangle = |10_2\rangle \end{aligned} \tag{4.6}$$

Taking the tensor product of these results, we have:

$$|11\rangle \otimes |00\rangle \otimes |01\rangle \otimes |10\rangle = |11000110\rangle \, . \tag{4.7}$$

Column 2 of Table 4.1 shows the eigenstates corresponding to each of the tours in our example.

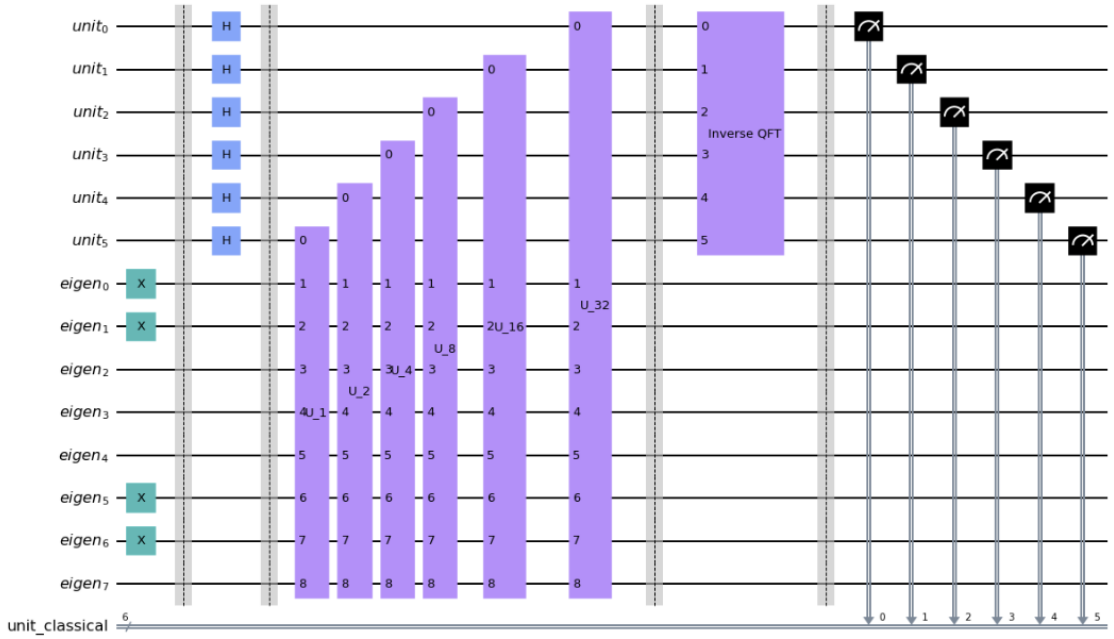

Figure 4.2: The complete circuit for phase estimation on the $1 \to 2 \to 3 \to 4$ tour in the example TSP instance with 4 cities. [14]

Now, in order to use phase estimation, we need to construct a controlled-U gate: $CU \equiv C(U_1 \otimes U_2 \otimes U_3 \otimes U_4) \equiv CU_1 \otimes CU_2 \otimes CU_3 \otimes CU_4$ [14]. We can decompose each of the U_j matrices into controlled-unitaries as such:

$$\begin{aligned} U_j &= \begin{bmatrix} e^{ia} & 0 & 0 & 0 \\ 0 & e^{ib} & 0 & 0 \\ 0 & 0 & e^{ic} & 0 \\ 0 & 0 & 0 & e^{id} \end{bmatrix} \\ &= \left(\begin{bmatrix} 1 & 0 \\ 0 & e^{i(c-a)} \end{bmatrix} \otimes \begin{bmatrix} e^{ia} & 0 \\ 0 & e^{ib} \end{bmatrix} \right) \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i(d-c+a-b)} \end{bmatrix}. \end{aligned} \tag{4.8}$$

The matrix $\begin{bmatrix} 1 & 0 \\ 0 & e^{i(c-a)} \end{bmatrix}$ is the unitary gate $U_1(c-a)$, the matrix $\begin{bmatrix} e^{ia} & 0 \\ 0 & e^{ib} \end{bmatrix}$ is the unitary gate $U_1(b-a)$ (with the global phase e^{ia} factored out), and $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i(d-c+a-b)} \end{bmatrix}$ is a controlled unitary matrix $CU_1(d-c+a-b)$ [14].

We then need to make each of the matrices above into controlled unitary matrices,

so $U_1(c-a)$ into $CU_1(c-a)$, $U_1(b-a)$ into $CU_1(b-a)$, and $CU_1(d-c+a-b)$ into $CCU_1(d-c+a-b)$ [14]. The latter is controlled controlled U_1 [14]. This step is necessary to use Qiskit, the Python-based quantum programming language, to build this circuit [14].

Figure 4.2 shows a complete circuit for the phase estimation algorithm on the $1 \to 2 \to 3 \to 4$ tour in our example. The corresponding eigenstate, $|11000110\rangle$, is encoded into the set of qubits labeled *eigen*, and the phase estimation is performed onto the *unit* qubits. After applying our U gate in increasing powers of 2 with each of the *unit* qubits as control, the *unit* qubits are put back into the computational basis with the inverse quantum Fourier transform. The result of this algorithm gives us the binary encoding of the estimated total phase of the corresponding TSP tour [14].

4.1.4 Interpreting the output

To decode the 6-bit binary result of the phase estimation algorithm, we first convert it to decimal. For example, if our result was the bit string 100100, that would be 24 in decimal. The phase θ that was estimated by the algorithm can then be found with the formula:

$$\theta = \frac{d}{2^b}, \tag{4.9}$$

where d is the decimal value we obtained from the circuit, and b is the number of bits in the bit string. In our example case, this gives us $\theta = \frac{24}{2^6} = 0.375$ [14]. However, if the phase we were trying to estimate is not in the form $d/2^b$, the phase estimation algorithm will not have a 100% probability of outputting the exact corresponding bit string. To obtain more accurate results, the algorithm needs to be run many times and the results need to be plotted to find the two most likely outputs. The true phase will be between the two most likely outputs, closer to the most likely output [14].

This circuit needs to be set up and run for each TSP tour. For other tours, we simply change which *eigen* qubits the X gates are applied to, to construct our eigenstate. After the phases are estimated for each tour, we can use a search algorithm to find the minimum value of the tour lengths [14].

4.1.5 Analysis

In order to obtain the phase accurate up to m bits with probability of success at least $1-\epsilon$, the number of qubits t needed for phase estimation is given by [32]:

$$t = m + \lceil \log\left(2 + \frac{1}{2\epsilon}\right) \rceil. \tag{4.10}$$

In addition to the phase estimation qubits, this algorithm requires $n \log_2(n-1)$ qubits to encode the TSP tours for an instance with n cities.

Inverse quantum Fourier transform runs in $\mathcal{O}(t)$ steps, with t being the number of qubits in the first register (the *unit* qubits in Figure 4.2) [10]. The other gates do not take any significant number of steps to run. We only need to set up the operator U once per TSP instance, since the same operator is used to test each of the eigenstates. Setting up U does not take longer than $\mathcal{O}(n^2)$ time. A classical search algorithm can find the minimum value of an unsorted array of n elements in $\mathcal{O}(n)$ time [1], which in our case would be $\mathcal{O}((n-1)!)$ since we have a total of up to $(n-1)!$ tours to find the minimum of. A quantum search algorithm improves on that by finding the minimum value of an unsorted array of n elements in $\mathcal{O}(\sqrt{n})$ time [33], which would be $\mathcal{O}\left(\sqrt{(n-1)!}\right)$, which is approximately $\mathcal{O}\left(\sqrt[4]{2\pi(n-1)}(\frac{n-1}{e})^{(n-1)/2}\right)$ by Stirling's approximation formula [1].

Qiskit Textbook claims that this algorithm grants a quadratic speedup over the classical brute force algorithm for a large number of cities [14]. Finding minimum tour length after computing them all clearly takes a significant amount of time relative to the size of the input: a TSP instance with n cities takes $\mathcal{O}\left(\sqrt{(n-1)!}\right)$ time to find the minimum tour distance. The classical brute force algorithm to solve TSP runs in $\mathcal{O}(n!)$ time, so a quadratic speedup is still rather insignificant.

4.2 Quantum annealing

As mentioned previously, quantum annealing uses the adiabatic theorem to transform one Hamiltonian into another in order to easily obtain the ground eigenstate. We use a method called adiabatic quantum optimization (AQO) to solve TSP.

The general idea for using AQO to solve TSP is as follows:

1. Map TSP to the classical Ising problem.
2. Make Ising model into a quantum computing problem.
3. Use quantum annealing to solve Ising model.
4. Decode results to find optimal TSP tour.

4.2.1 Ising model

An Ising model is a mathematical model that consists of a huge square lattice of sites, where each site can be in one of two states: -1 and +1 [34]. Neighboring sites interact in a certain way, which is given by the parameter J [34]. The total energy of the system is determined by the sum of the interactions of every neighboring pair of sites [34].

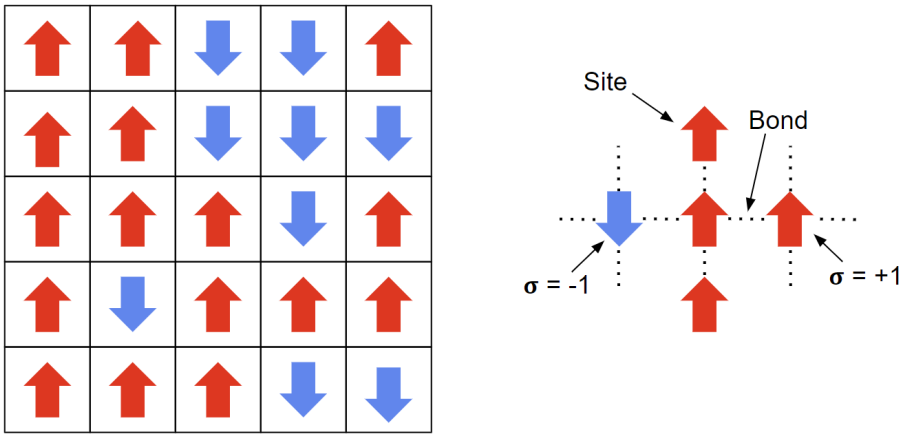

Figure 4.3: The Ising model can be pictured as a square lattice of cells, each cell being in one of two possible states. [34]

The classical Ising model can be expressed as a quadratic function of a set of N spins $s_i = \pm 1$ as such [16]:

$$H(s_1, \ldots, s_N) = -\sum_{i<j} J_{ij} s_i s_j - \sum_{i=1}^{N} h_i s_i, \tag{4.11}$$

and the quantum version of this Hamiltonian is simply [16]

$$H_P = H(\sigma_1^Z, \ldots, \sigma_N^Z), \tag{4.12}$$

where σ_i^Z is a Pauli matrix acting on the ith qubit in the Hilbert space of N qubits $|+\rangle, |-\rangle^{\otimes N}$, and J_{ij} and h_i are real numbers [16].

The Ising spin glass problem is an NP-hard problem for classical computers, and is defined as follows:

> Given an Ising model, what is the ground state energy of the Hamiltonian H? [16]

The decision version of the problem is NP-complete:

> Given an Ising model, does the ground state of H have energy ≤ 0? [16]

Note that the method of using the Ising model to solve other computational problems is also known as quadratic unconstrained binary optimization (QUBO), and that terminology is often used in mathematical literature [16].

4.2.2 Encoding TSP into the Ising problem

Consider a TSP instance over n cities. For simplification, let the starting city be fixed for each route. Let W be the $n \times n$ cost matrix with each element (i, j) representing

the weight of the edge from vertex i to vertex j. Define a binary variable b_{ti} such that $b_{ti} = 1$ if the ith city is visited at time t. Then the QUBO encoding is as such [35]:

$$H^{QUBO}(b) = A_1 \sum_{t=1}^{n}(1 - \sum_{i=1}^{n} b_{ti})^2 + A_2 \sum_{i=1}^{n}(1 - \sum_{t=1}^{n} b_{ti})^2 + B \sum_{\substack{i,j=1 \\ i \neq j}}^{n} W_{ij} \sum_{t=1}^{n} b_{ti} b_{t+1,j}. \quad (4.13)$$

Here we have parameters $A_1, A_2 > B \max_{i \neq j} W_{ij}$ to be adjusted during optimization [35].

We represent the quantum state to adiabatically evolve the Hamiltonian from H_0 to H^{QUBO} as such [35]:

$$|\theta\rangle = \prod_{i=1}^{r} \exp(-i\theta_{mix,i} H_{mix}) \exp(-i\theta_{obj,i} H) |+\rangle , \quad (4.14)$$

with r being the number of levels of the circuit [35]. Here we would have H^{QUBO} as the Hamiltonian H which we are trying to evolve the state into.

The minimum eigenvalue of the Hamiltonian H^{QUBO} is the energy of its ground state.

The maximum number of qubits needed to represent every route is $N = (n-1)^2$ [35]. However, we can adjust the encoding so that QUBO requires only $\lceil \log(n!) \rceil = n \log(n) - n \log(e) + \mathcal{O}(n)$ qubits (derived with the Stirling formula).

The optimal encoding of the TSP routes into bit strings is by using a factorial numbering system in which the ith digit starting from least significant can be any number between 0 and $i - 1$, so in general [35]:

$$(d_k \ldots d_0)_! \equiv \sum_{i=0}^{k} d_0 \cdot i!. \quad (4.15)$$

To decode this, we can compute the modulo by consecutive natural numbers, and that can be converted to the permutation of cities via Lehmer codes, which starting with the most significant factoradic digit, take the $(k+1)$th digit of the sequence $(0, 1, \ldots, k)$ [35]. The used digit is removed, and the procedure repeats for the next digits, and the taken digits in the given order directly encode a permutation [35]. The Hamiltonian can be adjusted according to the encoding [35].

After the Hamiltonians and encodings are set up, a D-Wave machine (a particular quantum annealer) can perform the required computations to obtain the eigenvalue of the ground state of the target Hamiltonian, which is the encoding of the shortest tour of the TSP [35] [16]. The quantum annealing circuit can also be modeled on a universal gate-based quantum computer, since quantum annealers have very limited uses. We try to minimize the number of gates we use for this, which relates to the number of qubits required to encode the TSP.

4.2.3 Analysis

The optimal encoding approach requires approximately $\mathcal{O}(n)\text{range}(W)$ number of measurements (where range(W) is the difference between the largest and smallest value in the cost matrix), but also requires an exponential circuit depth and volume (number of gates needed) when modeled with a universal gate-based quantum computer [35]. Other tour-encoding schemes, such as the basic QUBO encoding with Hamiltonian H^{QUBO}, while requiring more qubits to encode the instances (n^2 in the case of QUBO), take a smaller circuit depth and volume ($12n$ and $12n^3$, respectively, for QUBO) to perform the required computations [35].

The mapping of TSP to the Ising model is done in a polynomial number of steps, but the power of this polynomial can grow very rapidly [16]. The number of spins in the Ising spin glass problem scales no faster than n^3 [16].

For a problem of input size n, we typically find that AQO requires a time $T = \mathcal{O}(\exp\{\alpha n^\beta\})$ for some positive coefficients α and β, as $n \to \infty$ [16]. Although AQO may not solve NP-complete problems in polynomial time, it can still offer at least a small improvement over the classical algorithms [16].

4.3 Quantum algorithm based on Held-Karp

As we have seen, the classical Held-Karp dynamic programming algorithm breaks down the TSP into smaller subproblems to provide a huge speedup in the time complexity to obtain the exact optimal solution, with a runtime of $\mathcal{O}(n^2 2^n)$ for a TSP instance of n cities.

This quantum computing approach to TSP uses ideas from the Held-Karp algorithm by running a quantum minimum-finding algorithm on specifically-sized subproblems of the Held-Karp algorithm. This way, we can run classical Held-Karp to a point where quantum minimum-finding will help speed up the rest of the algorithm.

4.3.1 The algorithm

The algorithm is as follows [36]:

Let $G = (V, E, w)$ be the given graph representing the TSP instance, where $|V| = n$ is the number of verticies (cities), $E \subseteq V^2$ is the set of edges, and $w : E \to \mathbb{N}$ are the edge weights. We let $w(u, v) = \infty$ if the edge $\{u, v\} \notin E$. Assume we can access the appropriate edge weights in w in polynomial time, $p(n)$. Let $L = \max_{\{u,v\} \in E} w(u, v)$ be the length of the largest edge in the instance. We know that adding two integers takes $\mathcal{O}(\log L)$ time [36].

Let $D = \{(S, u, v) \mid S \subseteq V, u, v \in S\}$ and define $f : D \to \mathbb{N}$ as follows: $f(S, u, v)$ is the length of the shortest path in the graph induced by S that starts at vertex u, ends at vertex v, and visits all vertices in S exactly once. This is similar to the Held-Karp breakdown of TSP. Let $N(u)$ be the set of neighbors of vertex u in G. We can calculate $f(S, u, v)$ with the following recurrence [36]:

$$f(S, u, v) = \min_{\substack{t \in N(u) \cap S \\ t \neq v}} \{w(u, t) + f(S \setminus \{u\}, t, v)\}, \tag{4.16}$$

with $f(\{v\}, v, v) = 0$ for any v [36]. Note that $f(S, u, v)$ can also be calculated recursively by splitting S into two sets as such: let $k \in [2, |S| - 1]$ be some fixed number, then [36]:

$$f(S, u, v) = \min_{\substack{X \subset S, |X| = k \\ u \in X, v \notin X}} \min_{t \neq u}^{t \in X} \{f(X, u, t) + f((S \setminus X) \cup \{t\}, t, v)\}. \tag{4.17}$$

Let $\alpha \in (0, \frac{1}{2}]$ be a parameter to be specified later [36].

Now we perform the following steps [36]:

1. Calculate the values of $f(S, u, v)$ for all $|S| \leq (1 - \alpha)\frac{n}{4}$ classically using dynamic programming with Eq. 4.16 and store them in memory.

2. Run quantum minimum-finding algorithm over all subsets $S \subset V$ such that $|S| = n/2$ to find the answer,

$$\min_{\substack{S \subset V \\ |S| = n/2}} \min_{\substack{u, v \in S \\ u \neq v}} \{f(S, u, v) + f((V \setminus S) \cup \{u, v\}, v, u)\}. \tag{4.18}$$

   To calculate $f(S, u, v)$ for $|S| = n/2$, run quantum minimum-finding for Eq. 4.17 with $k = n/4$. To calculate $f(S, u, v)$ for $|S| = n/4$, run quantum minimum-finding for Eq. 4.17 with $k = \alpha n/4$. For any S such that $|S| = \alpha n/4$ or $|S| = (1 - \alpha)n/4$, we know the value of $f(S, u, v)$ from the classical preprocessing. [36]

4.3.2 Analysis

We use the notation $f(n) = O^*(c^n)$ for the case where $f(n) = p(n) \cdot c^n$ for $p(n)$ being some polynomial expression of n, and some constant c.

We know that the quantum minimum-finding algorithm can find a minimum value of an unordered array with a success probability of at least 2/3 in $\mathcal{O}(\sqrt{n})$ time [36].

The time complexity of the classical preprocessing part is [36]:

$$O^*\left(\binom{n}{\leq (1 - \alpha)n/4}\right) = O^*\left(2^{H(\frac{1-\alpha}{4})n}\right). \tag{4.19}$$

The complexity of the quantum part, considering the complexity of quantum minimum-finding, is [36]:

$$O^*\left(\sqrt{\binom{n}{n/2}\binom{n/2}{n/4}\binom{n/4}{\alpha n/4}}\right) = O^*\left(2^{\frac{1}{2}\left(1+\frac{1}{2}+\frac{H(\alpha)}{4}\right)n}\right). \tag{4.20}$$

The total complexity of the algorithm is optimized when the complexities of the two parts, expressions 4.19 and 4.20, are equal. We do this by selecting the optimal value for α, which we find to be approximately 0.055362 [36]. This makes the running time of the whole algorithm $O^*(2^{0.788595n}) = O^*(1.727391...^n)$. The arithmetic operations on integers increase the time complexity by a multiplicative factor of $\mathcal{O}(\log L)$ [36].

This TSP approach does not consider the amount of physical resources required for the computations. The number of qubits required to perform quantum minimum-finding is $\log_2 m$, where m is the size of the set we search through [10]. In this case, we search through $m = \mathcal{O}(2^{n/2})$ elements, so the number of qubits needed is $\log_2 \mathcal{O}(2^{n/2}) = \mathcal{O}(n/2)$ qubits, which is excellent compared to the qubits required for other quantum computing approaches to TSP.

5. Conclusion

5.1 Analysis

We compare the performances of the various algorithms for solving TSP.

The following table compares the time and space complexities of the algorithms, as well as labels whether they involve quantum computing and whether or not give an exact optimal solution (as opposed to an approximate solution).

Algorithm	Time	Space	Quantum?	Exact?
Brute Force	$\mathcal{O}(n!)$	$\mathcal{O}(n^2)$	No	Yes
Held-Karp	$\mathcal{O}(2^n n^2)$	$\mathcal{O}(n2^n)$	No	Yes
General greedy	$\mathcal{O}(n^2 \log_2(n))$	$\mathcal{O}(n^2)$	No	No
NN	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	No	No
RNN	$\mathcal{O}(n^3 \log_2(n))$	$\mathcal{O}(n^2)$	No	No
Christofides	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	No	No
Yatsenko's	$> \mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	No	No
Phase estimation	$\mathcal{O}(\sqrt{(n-1)!})$	a qubits	Yes	Yes
QUBO	$\mathcal{O}(\exp\{\alpha n^\beta\})$	$\mathcal{O}(n^2)$ qubits	Yes	Yes
AQO with optimal encoding	$\mathcal{O}(\exp\{\alpha n^\beta\})$	$\mathcal{O}(n\lceil\log(n!)\rceil)$	Yes	Yes
Quantum Held-Karp	$O^*(1.727391...^n)$	$\mathcal{O}(n/2)$ qubits	Yes	Yes

Table 5.1: A comparison between all the discussed TSP approaches. Here we have $a = m + \lceil \log_2(2 + \frac{2}{2\epsilon}) \rceil + n \log_2(n-1)$.

We notice that the classical approximation algorithms have a much quicker time complexity than the classical exact solution algorithms for TSP. The classical approximation algorithms generally run in polynomial time, whereas to get an exact solution, the best classical algorithm runs in over exponential time. The phase estimation algorithm still runs in an exponential time complexity (as we had shown before, $\mathcal{O}(\sqrt{(n-1)!}) \approx \mathcal{O}(\sqrt[4]{2\pi(n-1)}(\frac{n-1}{e})^{(n-1)/2})$ despite utilizing quantum computing. The quantum annealing approach, however, is quite promising, arriving at the exact TSP solution in exponential time. Finally, the quantum computing augmenta-

tion to the Held-Karp algorithm, despite still running in exponential time, reduces the time from the classical Held-Karp's $\mathcal{O}(2^n n^2)$ to $\mathcal{O}(p(n) \cdot 1.727391...^n)$ (for $p(n)$ being some polynomial in n). Comparatively, even though the two time complexities are very similar, this is still a speed-up in obtaining the exact TSP solution.

The space complexity with the classical algorithms is not typically a matter of concern. The amount of memory space available in classical computers makes it possible to solve TSP for very large instances; it is the time complexity that is the main issue.

We see that involving quantum computing does, in fact, improve on the best known classical algorithm for obtaining the exact optimal route in the TSP. However, the exact optimal solution TSP remains an NP-hard problem–none of the exact solution algorithms, including the ones utilizing quantum computing, run faster than in exponential time.

The quantum computing approaches are promising; however, the required numbers of qubits are generally too much for the current state of existing quantum computers. Note that the qubit requirement for the two quantum annealing algorithms is in terms of quantum annealer qubits–there are thousands of low-noise qubits effectively available for quantum annealing, so at this time, this algorithm is more practical for solving larger instances of TSP than the universal gate-based algorithms [13]. For the universal gate-based quantum computers, though, we currently do not have enough qubits (let alone fault-tolerant circuitry) to work with large TSP instances, so classical computing is still currently the best way to solve TSP.

5.2 Future work

We already see that combining quantum computing with the classical Held-Karp dynamic programming algorithm shows improvement on solving this difficult computational problem. More algorithms may be developed that take advantage of the techniques that some of the other classical approaches use. Currently we know that quantum computing is best used for searching for a single solution through huge search spaces, making it more useful for finding exact TSP solutions than for approximating. However, there may be ways to utilize quantum computing to improve on the classical approximation algorithms, either to improve the accuracy of the results, or to provide an even greater speed-up.

Quantum computing systems will keep improving as researchers experiment and discover new ways to construct quantum systems, both physically and theoretically. As we build our experience with quantum technology, we can improve on old algorithms to solve computationally difficult problems, as well as invent new approaches with our

newly realized resources.

Bibliography

[1] Thomas H Cormen et al. *Introduction to Algorithms*. Third edition. MIT Press, 2009.

[2] Paul E Black. *big-O notation*. Sept. 2019. URL: https://xlinux.nist.gov/dads/HTML/bigOnotation.html.

[3] Great Learning Team and Balabaskar. *Why is Time Complexity Essential and What is Time Complexity?* Jan. 2022. URL: https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/.

[4] Eric Allender, Michael C Loui, and Kenneth W Regan. *Complexity Classes*. Jan. 2007. URL: https://cse.buffalo.edu/~regan/papers/pdf/ALRch27.pdf.

[5] Sanjeev Arora and Boaz Barak. *Computational complexity: A modern approach*. Cambridge University Press, 2016.

[6] Behnam Esfahbod. *P np np-complete np-hard*. File: `complexity_classes.png`. 2007. URL: https://commons.wikimedia.org/wiki/File:P_np_np-complete_np-hard.svg.

[7] Paul E Black. *NP-compete*. Apr. 2021. URL: https://xlinux.nist.gov/dads/HTML/npcomplete.html.

[8] D. E. Knuth. "Postscript about NP-Hard Problems". In: *SIGACT News* 6.2 (Apr. 1974), pp. 15–16. ISSN: 0163-5700. DOI: 10.1145/1008304.1008305. URL: https://doi.org/10.1145/1008304.1008305.

[9] John Watrous. *Quantum Computational Complexity*. 2008. DOI: 10.48550/ARXIV.0804.3401. URL: https://arxiv.org/abs/0804.3401.

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

[11] Michael Hahsler and Kurt Hornik. "TSP—Infrastructure for the Traveling Salesperson Problem". In: *Journal of Statistical Software* 23.2 (2007), pp. 1–21. DOI: 10.18637/jss.v023.i02. URL: https://www.jstatsoft.org/index.php/jss/article/view/v023i02.

[12] Scott Aaronson. *What makes quantum computing so hard to explain?* Nov. 2021. URL: https://www.quantamagazine.org/why-is-quantum-computing-so-hard-to-explain-20210608/.

[13] Cem Dilmegani. *Quantum Annealing in 2022: Practical quantum computing.* Feb. 2022. URL: https://research.aimultiple.com/quantum-annealing/.

[14] MD Sajid Anis et al. *Qiskit: An Open-source Framework for Quantum Computing.* 2021. DOI: 10.5281/zenodo.2573505.

[15] *What is a qubit?* URL: https://azure.microsoft.com/en-us/overview/what-is-a-qubit/#introduction.

[16] Andrew Lucas. "Ising formulations of many NP problems". In: 2 (2014). ISSN: 2296-424X. DOI: 10.3389/fphy.2014.00005. URL: https://www.frontiersin.org/article/10.3389/fphy.2014.00005.

[17] Hugh Collins and Kortney Easterly. *IBM Unveils Breakthrough 127-Qubit Quantum Processor.* Nov. 2021. URL: https://newsroom.ibm.com/2021-11-16-IBM-Unveils-Breakthrough-127-Qubit-Quantum-Processor.

[18] John Preskill. "Quantum Computing in the NISQ era and beyond". In: *Quantum* 2 (Aug. 2018), p. 79. DOI: 10.22331/q-2018-08-06-79. URL: https://doi.org/10.22331%2Fq-2018-08-06-79.

[19] Quang Nhat Nguyen. *Travelling Salesman Problem and Bellman-Held-Karp Algorithm.* May 2020. URL: http://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf.

[20] Haider Abdulkarim and Ibrahim Fadhil Alshammari. "Comparison of Algorithms for Solving Traveling Salesman Problem". In: *International Journal of Engineering and Advanced Technology* ISSN (Aug. 2015), pp. 2249–8958.

[21] Maria José Serna Iglesias. *Dynamic programming.* Feb. 2015. URL: https://web.archive.org/web/20150208031521/http://www.cs.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf.

[22] Charles Hutchinson et al. "Traveling Salesman Problem". In: 2016.

[23] Michael Held and Richard M. Karp. "A dynamic programming approach to sequencing problems". In: *Proceedings of the 16th ACM national meeting, ACM 1961, USA.* Ed. by Thomas C. Rowan. ACM, 1961, p. 71. DOI: 10.1145/800029.808532. URL: https://doi.org/10.1145/800029.808532.

[24] Christine L. Valenzuela and Antonia J. Jones. "Estimating the Held-Karp lower bound for the geometric TSP". In: *European Journal of Operational Research* 102.1 (1997), pp. 157–175. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(96)00214-7. URL: https://www.sciencedirect.com/science/article/pii/S0377221796002147.

[25] Lawrence Weru. *11 animated algorithms for the traveling salesman problem*. Aug. 2021. URL: https://stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/.

[26] Gregory Gutin, Anders Yeo, and Alexey Zverovich. "Traveling Salesman Should not be Greedy: Domination Analysis of Greedy-Type Heuristics for the TSP". In: *Discrete Applied Mathematics* 117 (Mar. 2002), pp. 81–86. DOI: 10.1016/S0166-218X(01)00195-0.

[27] Alex Neoh et al. *An Evaluation of the Traveling Salesman Problem*. Aug. 2020. URL: https://scholarworks.calstate.edu/concern/theses/8g84mp499?locale=en.

[28] Michael T. Goodrich and Roberto Tamassia. *Algorithm design and applications*. Wiley, 2015.

[29] Seth Pettie and Vijaya Ramachandran. "An Optimal Minimum Spanning Tree Algorithm". In: *J. ACM* 49.1 (Jan. 2002), pp. 16–34. ISSN: 0004-5411. DOI: 10.1145/505241.505243. URL: https://doi.org/10.1145/505241.505243.

[30] Christopher Clingerman, Jeremiah Hemphill, and Corey Proscia. *Analysis and Counterexamples Regarding Yatsenko's Polynomial-Time Algorithm for Solving the Traveling Salesman Problem*. 2008. DOI: 10.48550/ARXIV.0801.0474. URL: https://arxiv.org/abs/0801.0474.

[31] Vadim Yatsenko. "Fast Exact Method for Solving the Travelling Salesman Problem". In: (Mar. 2007).

[32] Karthik Srinivasan et al. *Efficient quantum algorithm for solving travelling salesman problem: An IBM quantum experience*. 2018. DOI: 10.48550/ARXIV.1805.10928. URL: https://arxiv.org/abs/1805.10928.

[33] Christoph Dürr and Peter Hoyer. "A Quantum Algorithm for Finding the Minimum". In: *CoRR* quant-ph/9607014 (July 1996).

[34] Jeffrey Chang. *The Ising Model*. 2019. URL: https://stanford.edu/~jeffjar/statmech/intro4.html.

[35] Adam Glos, Aleksandra Krawiec, and Zoltán Zimborás. *Space-efficient binary optimization for variational computing*. 2020. DOI: 10.48550/ARXIV.2009.07309. URL: https://arxiv.org/abs/2009.07309.

[36] Andris Ambainis et al. *Quantum Speedups for Exponential-Time Dynamic Programming Algorithms*. 2018. DOI: 10.48550/ARXIV.1807.05209. URL: https://arxiv.org/abs/1807.05209.