Theory and Applications of Quantum Computing

William Bernoudy

Mentor: Richard Hoshino

April 5, 2017

Contents

Chapter 1

Motivation for Quantum Computing

Moore's law states that the number of transistors that can be manufactured at a given cost will double approximately every two years [9]. As the transistor count on a microchip is proportional to computing power, the law implies that the computational power available for a given amount of money doubles every two years. The entire progression of modern technology is essentially reliant upon Moore's law and the seemingly untethered growth of computing power. Companies are able to release significantly more powerful devices each year that are smaller and more power efficient. Every year new software is released that takes advantage of the increase, allowing users to use more powerful programs with increased capability, with less waiting time for the computing processes to be completed. Moore's law has held true since the dawn of the microchip until at least very recently. However, there are physical limits to the law that prevent it from continuing indefinitely. One of the most obvious is the size of transistors. Currently, 10 nm transistors are being put into production by Intel [15]. However, transistors any smaller than 7 nm will start to experience the effects of quantum tunneling, where electrons flowing through one part of the transistor will "tunnel" through the silicon barrier to other parts, making the production of these transistors significantly more difficult [8]. Physical restrictions such as this will make the continuation of the law increasingly infeasible.

However, the end of Moore's law does not mean the end of the progression of computation. Quantum computing offers an alternative to computing on silicon chips. While quantum computers are superior to classical (e.g. transistor-based) computers only for a limited class of problems, these problems have wide-reaching applications in many fields such as machine learning, protein folding, search algorithms, and optimization problems. Many of these problems are able to be completed in exponentially shorter periods of time on quantum computers [1]. Though quantum computers will never replace classical computers, their possible applications include exciting and important future developments.

I chose to study quantum computing mainly because I believe it may play a crucial role in the development of certain technologies that will benefit many people in the same way conventional computing has. For example, understanding protein folding allows researchers to develop better drugs to prevent and cure diseases such as HIV/AIDS and cancer, and quantum computing may allow us to achieve more on this front than conventional computers ever will [20]. I also appreciate the way that quantum computing attempts to repurpose some of the strangest observed phenomena central to the fundamental nature of the universe to perform certain types of computations faster.

To accomplish this goal, I took classes such as Linear Algebra and Complex Analysis which helped me form an understanding and familiarity of the underlying concepts behind quantum computing. I then used self-study of quantum computing textbooks, research articles and blog

posts by experts in the field to learn the basics of quantum computing itself. I also used several final projects in various classes as opportunities to expand my understanding by implementing common algorithms. Finally, I interned at D-Wave Systems, currently the only company in the world producing quantum computers. During the internship, I worked with an algorithms researcher[1] to find the best way to run a certain type of problem on their computers.

1.1 Outline of Keystone

In chapter 2, I outline the developments in physics that led to the modern understanding of the quantum world, explain how conventional computing can be modeled using linear algebra, explain the basics of quantum computing, and then touch on the different types of quantum computation.

In chapter 3, I explain how it is possible to simulate a gate-based quantum computer on a conventional computer as well as describe more precisely what a quantum computer does using a mathematical construct, and then introduce the simulator I wrote.

In chapter 4, I discuss Grover's algorithm and its applications, describe how it works, and explain my implementation of it using my quantum computer simulator.

In chapter 5, I describe an associative memory algorithm and show my implementation of it on the same simulator.

In chapter 6, I explain quantum annealing and its application to solving Ising problems, compare it to simulated annealing, and discuss the difficulties with programming and using D-Wave's actual quantum annealing machine.

Finally, in chapter 7, I conclude my Keystone, discuss some of the problems quantum computing faces and what I aim to do in the future.

[1] Andrew King, aking@dwavesys.com.

Chapter 2

Introduction to Quantum Computing

2.1 Quantum Mechanics

At the dawn of the twentieth century, Max Planck made the amazing discovery that the energy level of an electromagnetic wave, i.e. light, is quantized [6]. Instead of being continuous, as previously thought, Planck discovered that there is a lower limit on level of energy that an electromagnetic wave can have, and that the energy must always be a multiple of this lower bound, called Planck's constant. In other words, the energy of an electromagnetic wave is divided up into discrete "quanta" (as he called them), or packets of energy. The word "quantum" is derived from this discovery.

Einstein expanded on Planck's findings and found that light can be modelled not only as a wave (as was the case in physics previously) but also as a particle [2]. The corresponding particle for light is called a photon. This led to the discovery of the wave-particle duality of nature, one of the fundamental principles of quantum mechanics. The wave-particle duality states that every particle or packet of energy has the properties of both waves and particles. Quantum mechanics was developed as a result of this discovery. Scientists found that quantum mechanics did an extraordinary job of explaining certain phenomena on tiny scales that classical physics could not explain (e.g. the "orbit" of an electron around the nucleus of an atom; according to classical physics, the orbit would decay and the "orbit" could not be stable) [16].

In 1926, Erwin Schrödinger published a paper outlining a new equation (called "Schrödinger's wave equation") that modelled quantum physics [14]. In the quantum understanding, all properties of a physical system are modeled by a wave function which describes the different probabilities of the system being in any of its possible configurations. This means that before the system is measured by an observer, the system exists simultaneously in some amount of all of its possible states. When they are measured (when the wave function collapses), a system state is chosen seemingly at random according to the probability distribution. Schrödinger's wave equation models the evolution of the wave function of a physical system over time. It does not predict what the actual values will be, but instead gives a spread of the probabilities for the different values one might get when the system is measured.

Schrödinger's wave equation results in some interesting properties of matter. One of these, which allows for a basis for quantum computation, is called quantum superposition. Because of a certain attribute of the solutions to the wave equation (their linearity), matter can exist in an overlap of many different states. For example, let's say we have some electrons traveling in a loop of conductive material (a circuit). In the classical understanding, the electrons must be traveling in one of the two possible directions. In the quantum mechanical understanding, as long as we do not disturb the circuit in any way (which is equivalent to observation or the wave collapsing), the

electrons can actually flow both ways at once. We could then use Schrödinger's wave equation to determine the likelihood of the electrons flowing in one direction or another once we measure the system.

2.2 Classical Computation

Classical computation usually consists of two components: memory and a processor. In the memory, binary information is stored in the form of bits. A bit is a single piece of binary data, meaning it is either a zero or a one; the information that we desire to process must thus be encoded in a sequence of zeros and ones. The processor is simply a device that performs different operations on the bits in the memory.

At any given point in time, we can talk about the system state of the processor. This essentially consists of the value of the bits in the memory. For a processor that runs n bits, the memory must be in one of 2^n possible states. For example, if we have a processor that uses a 2-bit architecture, the memory has 2 bits. This means that at any point during our processing, we could examine the memory and see that it will be in one of $2^2 = 4$ possible states:

$$00$$

$$01$$

$$10$$

$$11$$

Another way of representing the states of a classical computer is with vectors. Though unnecessary to understand classical computation, vectors will help tremendously when trying to understand quantum computation. Instead of using a zero or a one to represent the state of one bit, we can use a two dimensional vector :

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

These are the two binary states of a single bit. The $|0\rangle$ notation is simply used to indicate that in this case, we are talking about the vector that corresponds to the value of 0 in our system.

For computers with more than one bit, we are actually dealing with vectors in higher dimensional space. If we have a processor with n bits, then the vectors that correspond to the different states of the machine will have 2^n dimensions. We can see this holds true for the single bit processor, because it has $2^1 = 2$ dimensions. For a classical computer, the number of dimensions created by its bits is equal to the total number of possible states of the computer. Each state of the computer corresponds to a vector pointing directly in one of the 2^n dimensions with a magnitude of 1.

To calculate a system's state vector from the states of its bits, we can use the *tensor* product of all the states of the individual bits. The tensor product for matrices, also called the Kronecker product and denoted as $\otimes$, is a matrix operation defined for any two matrices that produces a third matrix in a way that generalizes the outer product of two vectors. For an m by n matrix $\mathbf{A}$ and a p by q matrix $\mathbf{B}$, the tensor product of $\mathbf{A}$ and $\mathbf{B}$ is an mp by nq matrix is

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}$$

where $a_{ij}\mathbf{B}$ is expanded wherever it has been placed in the resulting matrix. The following example illustrates the tensor product of two 2×2 matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1 \cdot 0 & 1 \cdot 5 & 2 \cdot 0 & 2 \cdot 5 \\ 1 \cdot 6 & 1 \cdot 7 & 2 \cdot 6 & 2 \cdot 7 \\ 3 \cdot 0 & 3 \cdot 5 & 4 \cdot 0 & 4 \cdot 5 \\ 3 \cdot 6 & 3 \cdot 7 & 4 \cdot 6 & 4 \cdot 7 \end{bmatrix} = \begin{bmatrix} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{bmatrix}.$$

Armed with the tensor product, we can compute the system state vector from the states of all its bits. For a system state S and its bits $b_1, b_2, \ldots, b_n$, we have

$$|S\rangle = |b_1\rangle \otimes |b_2\rangle \otimes \cdots \otimes |b_n\rangle$$

This works out to a simple pattern. First, let k equal the bit string $b_1 b_2 \ldots b_n$ interpreted as a base 2 number. Then S will be equal to a vector consisting of all zeros with a single 1 in the $(k+1)$th position. For example, take a 3-bit computer in the state 011. We can calculate S as follows:

$$|S\rangle = |0\rangle \otimes |1\rangle \otimes |1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Note that $011_2 = 3$, and that the 1 in the resulting vector is in position 3+1=4.

For any classical computer with a selection of bits, each bit can only be in two possible states, and so the computer must always be in one of 2^n possible states. This means that its state vector will have 2^n dimensions, and will always be pointing directly in just one of those dimensions. Any computation that we carry out on that state will always result in the vector switching to pointing directly in a new dimension with a magnitude of 1. However, this is just one way of representing information; quantum computing offers a radically different approach where the state vector is not limited to just one state at a time.

2.3 Quantum Computing

A quantum computer is a computer that takes advantage of quantum phenomena. Specifically, it is a computer based on *qubits* instead of bits. Qubits are pieces of information that can have states and interact in ways not possible with a classical understanding. For example, unlike a classical bit, a qubit can be in a superposition of multiple states at once.

Let's say we have one qubit called ψ. Its vector is given by:

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$$

where α_0 and α_1 are complex numbers satisfying the constraint

$$|\alpha_0|^2 + |\alpha_1|^2 = 1$$

This means that the values have an infinite number of possibilities. Unlike a classical bit, for which the vector could only consist of all zeroes and one "1," there are an infinite number of directions ψ can be pointing. This is a mathematical way of saying that ψ is not really in either the $|0\rangle$ state or the $|1\rangle$ state, but in a superposition of both. It exists as some amount of both at the same time.

The α values, also called the amplitudes of their corresponding states, are often written as

$$|\psi\rangle = ae^{i\theta}\,|0\rangle + be^{i\phi}\,|1\rangle$$

where a, b, θ and ϕ are all real numbers. Because we have the same constraint as before, we also have that

$$1 = |ae^{i\theta}|^2 + |be^{i\phi}|^2 = |a(\cos\theta + i\sin\theta)|^2 + |b(\cos\phi + i\sin\phi)|^2$$
$$1 = a^2(\cos^2\theta + \sin^2\theta) + b^2(\cos^2\phi + \sin^2\phi)$$
$$1 = a^2 + b^2$$

Because a and b also must satisfy this normalization constraint, we can represent them both with just a single value on the interval [0,1]. Thus we can represent a single qubit with just three real values. The Bloch sphere visualizes this representation. When the qubit is pointed directly upward (the positive z-axis), it has a value of $|0\rangle$. When it is pointed directly downwards (the negative z-axis), it has a value of $|1\rangle$. Thus the z-axis corresponds to the normalized a and b value (note that while $|0\rangle$ and $|1\rangle$ are shown on the same axis, they are still linearly independent basis vectors). Unlike the bit, we can see that there are now two axes of rotation afforded to the qubit that were not present before. However, the current state of the qubit in the Bloch sphere remains on the surface of the sphere. We can see in this example that it is pointed more closely upwards, towards the $|0\rangle$ vector, than it is downwards. This means that if we were to observe this qubit, it is more likely that it would collapse to the $|0\rangle$ state. If the qubit were pointed more towards the bottom of the sphere at the $|1\rangle$ vector, it would be more likely that we would collapse to the $|1\rangle$ state.

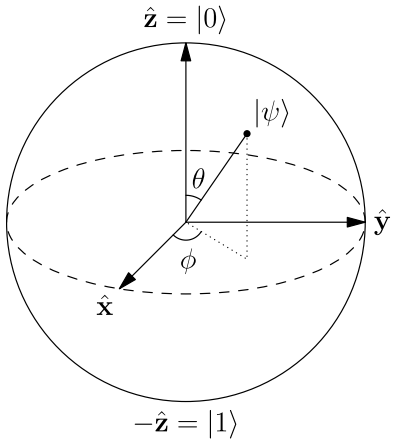


Figure 2.1: The Bloch sphere

The variability of qubits is furthered amplified when adding more qubits. Like the classical computer, the possible states for an n-qubit quantum computer exist in a 2^n dimensional space. However, this time, the vector which represents the state of the qubits does not have to be pointing in only one of the dimensions — it can exist in all of them (i.e. it can be in a superposition of all of them). For example, if we have a 3-qubit system, then the general state of the qubits, is given by:

$$|\Psi\rangle = \alpha_{000}|000\rangle + \alpha_{100}|100\rangle + \alpha_{010}|010\rangle + \alpha_{110}|110\rangle + \alpha_{001}|001\rangle + \alpha_{101}|101\rangle + \alpha_{011}|011\rangle + \alpha_{111}|111\rangle$$

$$|\Psi\rangle = \begin{pmatrix} \alpha_{000} \\ \alpha_{100} \\ \alpha_{010} \\ \alpha_{110} \\ \alpha_{001} \\ \alpha_{101} \\ \alpha_{011} \\ \alpha_{111} \end{pmatrix}$$

Thus, at any point in the computation, the state of the qubits, Ψ, can be described with a vector of length 2^n. When we measure the system, it disturbs the "quantum-ness" of the qubits, causing the wave function to collapse and to pick one value. The probability of the system collapsing into a certain state is given by the square of the magnitude of the α value corresponding to that state.

2.4 The Gate Model of Quantum Computing

To qualify as a quantum computer, a device or model must simply be able to do computation that takes advantage of quantum effects. This means that there are many approaches in the field to understanding quantum computing as well as distinct approaches to physical realizations. In this paper, I will discuss the two most common models: the gate model and quantum annealing. They have many similarities, and perhaps most importantly, the concept of representing the system state with a complex vector of length 2^n is common to both.

The gate model of quantum computing (also called a universal quantum computer or a Quantum Turing machine) describes a model where discrete operations applied to the system that predictably produces a new system state. In this sense, it is similar to way that we think about classical computing (the system state exists in memory and then is passed into the processor, which then predictably modifies the system state and the outcome is loaded back into the memory). The famous quantum algorithms, such as Grover's algorithm and Shor's algorithm, are all described using the gate based model. It is also called a universal quantum computer because it is able to run any quantum algorithm with a polynomial overhead. However, researchers have struggled to make a device that is able to perform gate based algorithms in any meaningful way.

In the next chapter, I will explain how the gate-based model provides easy ways of simulating a universal quantum computer.

Chapter 3

Simulating a Gate-Based Quantum Computer

Because the state of the computer can be represented with a single vector, an easy way to simulate quantum computation is to simply keep track of this vector. At the end of our computation, we can examine the vector to see which state the system is most likely to collapse into, and thus check to see if our computation worked.

To create a system of n qubits, we simply make a vector length 2^n of all zeros. Then, just as with the classical computer, depending on what we want the initial values of qubits to be we make one of the zeros in the vector a 1. The position of the 1 corresponds to the decimal form of the binary representation of the bits, e.g. if we want to start with $|100\rangle$ as the initial state, then we would change the zero in position 5 of the vector to a 1.

Just like classical computation, quantum computation is done by causing the qubits to react in certain ways through "quantum gates." In the same vein as the state of the computer, gates can conveniently represented as matrices. We can multiply the state of the system by a matrix corresponding to a gate to "simulate" the gate, as the resulting vector will be representative of the system after it has gone through the gate [17].

For classical compuatation, where the state of the system must be vector consisting of all zeros and one "1" (because it can only be in one state at once), the only valid matrices (meaning those that correspond to physically possible gates) are ones that always produce a valid system after the multiplication. If a matrix in classical computation produces anything but a vector of all zeros and one "1", then we know it's invalid. However, quantum gates do not have this restriction. This means that matrices that produce vectors with lots of different values are perfectly fine; this actually just represents a superposition of different system states. In fact, the only restriction on quantum gates is that they are reversible and that their matrix form is a *Hermitian* matrix. This means that if a quantum gate has the corresponding matrix operator $\mathbf{O}$, it is only a valid gate if $\mathbf{O}\mathbf{O}^* = \mathbf{O}^*\mathbf{O} = I$, where $\mathbf{O}^*$ denotes the adjoint of $\mathbf{O}$ (i.e. $(\mathbf{O}^*)_{ij} = \overline{\mathbf{O}_{ji}}$, with $\overline{\mathbf{O}_{ji}}$ being the complex conjugate of $\mathbf{O}_{ji}$).

The superposition principle of quantum mechanics states that any two valid quantum states can be combined linearly to create another valid state. This is a result of the fact that the Schrödinger equation is a linear equation, so any of its solutions (which are wave functions, or in our case, system states of the qubits in our quantum computer) can also be combined linearly. This principle is the reason for why we can model any gate as a matrix, as a gate is simply a linear combination of the previous amplitudes of each possible state of the system. Similarly, matrix multiplication on a vector defines a linear transformation.

Most quantum gates only act on 1, 2 or 3 qubits. Some common examples include the three Pauli matrices

$$\underset{\text{Pauli-x}}{\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}} \quad \underset{\text{Pauli-y}}{\begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix}} \quad \underset{\text{Pauli-z}}{\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}}$$

The Pauli-x matrix, also called a NOT gate, "flips" the value of a single qubit, taking $|0\rangle$ to $|1\rangle$ and vice versa. It is equivalent to rotating the qubit state π radians around the x-axis on the Bloch sphere. The Pauli-y and z matrices also rotate the qubit state π radians on the Bloch sphere around the y and z axes respectively.

Other common examples include the Hadamard, CNOT, SWAP and the Toffoli gates

$$\underset{\text{Hadamard}}{\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}} \quad \underset{\text{CNOT}}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}} \quad \underset{\text{SWAP}}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}} \quad \underset{\text{Toffoli}}{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}}$$

The Hadamard gate, when applied to a qubit with a pure state of $|0\rangle$ or $|1\rangle$, will put the qubit in a equal superposition of both states resulting in a measurement of 0 or 1 being equally likely. For example, applied to $|0\rangle$, we have

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

The CNOT gate, or the controlled-NOT gate, applies a NOT gate to the second qubit only on the states where the first qubit is $|1\rangle$. For example, it will map $|10\rangle \to |11\rangle$ and $|11\rangle \to |10\rangle$, but $|01\rangle \to |01\rangle$. The SWAP gate swaps the states of the two qubits, mapping $|10\rangle \to |01\rangle$, etc. The Toffoli gate is a double controlled NOT gate, meaning it will only apply a NOT to the third qubit if both the first and second are in the $|1\rangle$ state.

However, we cannot use only matrix multiplication to apply a gate that works on fewer qubits than our n-qubit system. Thus, we need a way generate a larger matrix that performs the gate on only the qubits we want. This matrix is generated by combining each matrix with the tensor product, similar to how we can combine the single bit vectors to get the system state of a classical computer. Thus if we want to find the operator $\mathbf{O}$ that works on an n-qubit quantum computer for which we are applying single qubit operator O_i to qubit i, we can do

$$\mathbf{O} = O_1 \otimes O_2 \otimes \ldots \otimes O_n$$

If we do not want to apply an operator to some of the qubits, we can simply use the 2 by 2 identity matrix (I_2) instead of another matrix. For example, if we have a 3-qubit system and we want to generate the operator that would be equivalent to applying a Pauli-x operator to the first qubit and a Pauli-z operator to the third qubit, then we would have the following overall operator:

$$\mathbf{O} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

If we want to apply a multiple qubit operator, we can just use that matrix in the chain of tensor products and leave out the next few I_2 matrices (as long as it is being applied to consecutive qubits).

One problem with simulation for this technique is that it is slow (because tensor products are slow) and that there is no way to handle multiple qubit gates being applied to non-consecutive qubits. Luckily, Lee Spector has provided an algorithm for that very purpose in his book *Automatic Quantum Computer Programming: A Genetic Programming Approach* [17]:

To expand gate matrix G (explicitly) for application to an n-qubit system:

1. Create a $2^n \times 2^n$ matrix M.
2. Let Q be the set of qubit indices to which the operator is being applied, and Q' be the set of the remaining qubit indices.
3. $M_{ij} = 0$ if i and j differ from one another, in their binary representations, in any of the positions referenced by indices in Q'.
4. Otherwise concatenate bits from the binary representation of i in the positions referenced by the indices in Q (in numerical order), to produce $i*$. Similarly, concatenate bits from the binary representation of j in the positions referenced by the indices in Q (in numerical order), to produce $j*$. Then set $M_{ij} = G_{i*j*}$.
5. Return M.

Figure 3.1: Lee Spector's algorithm, quoted from p.21

For any matrix $\mathbf{O}$ that represents an operator, the value at $\mathbf{O}_{ij}$ represents how much the amplitude of state $|j\rangle$ will get mixed into the amplitude of the state $|i\rangle$ in the resulting system state. Thus, on the third step, it follows that if i and j differ in their bit representations at any position of qubits that will not be affected by the resulting operator, then M_{ij} should be 0. This is because the result of the application of the gate will not depend on and will not change the state of any qubit which the operator is not being applied to (i.e. the states of all the qubits whose indices are in Q'). Thus the outcome amplitude of state $|i\rangle$ should only be a function of the previous amplitudes of the states where the unaffected qubits have the same 0 or 1 state.

Following the same logic for the fourth step of the algorithm, to fill in M_{ij} where $|i\rangle$ and $|j\rangle$ have the same states for unaffected qubits, we need to look at the states of the affected qubits in $|i\rangle$ and $|j\rangle$ and use the corresponding element of G. That is, it will use the element of G which

represents how much the amplitude of the state $|e\rangle$ will get mixed into the resulting amplitude of $|f\rangle$, where $|e\rangle$ is the state of the qubits G acts on in $|i\rangle$ and $|f\rangle$ is the state of the qubits G acts on in $|j\rangle$.

Now that we have a way to represent the state of the system and a way to apply gates, we have everything necessary for simulating a quantum computer. My implementation of a simulator in Racket can be seen in appendix A.

In the next chapter, I introduce Grover's algorithm, one of the most important algorithms discovered for a universal quantum computer, and explain how I implemented it using my simulator.

Chapter 4

Implementing Grover's Algorithm on a Quantum Computer Simulator

4.1 Introduction

Grover's algorithm is one of the two most important quantum algorithms. Given a search function, which returns true if the input is the data we are searching for and false otherwise, Grover's algorithm is able to find the correct data entry out of N possible entries with a complexity of $O(\sqrt{N})$. This is a significant speedup over the classical algorithm of simply checking each data entry one by one which has a complexity of $O(N)$. Because of how common this problem is, Grover's algorithm has a huge amount of applications, from artificial intelligence to protein sequence comparison. Another good example is the 3-SAT problem: while the best known classical algorithm for solving the 3-SAT problem is of time complexity $O(2^n)$, Grover's algorithm can be used to reduce this to $O(2^{n/2})$.

However, on its own, Grover's algorithm needs a quantum search function, or oracle, meaning that the search function must be able to run on the same quantum computer the main algorithm is running on. Thus we also need a way to convert a classical search function to a quantum one.

4.2 Implementing Grover's Algorithm

Most simply, Grover's algorithm finds the one value that satisfies a given search function. It consists of applying a short series of gates over and over again until we arrive at the value we want. I will refer to one series of these gates as a Grover step. Each step consists of applying the oracle operator, U_ω, and then the Grover diffusion operator, U_S.

The oracle is a series of quantum gates which, if the solution to our search function is present in the superposition of states, flips the phase of that state. Thus, its matrix takes the form of the identity matrix with a single 1 changed to -1. For example, if we have 4 possible solutions to our search function and the correct value is 2, then the oracle could be represented by the matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Because the oracle actually would consist of a series of quantum gates, we will not simply be able to see where the -1 is and declare that our solution. However, to avoid needing to come up

with an oracle for testing, we can simply use a matrix of the above form.

The Grover diffusion operator, U_s, consists of three gates: a Hadamard gate applied to all qubits, a phase flip of the $|0^n\rangle$ state, and then a Hadamard gate again applied to all qubits. A Hadamard gate, when given a pure state (i.e. no superposition), simply gives us an even distribution across all qubits. The single qubit form is given by the matrix

$$H_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

If we are going to apply the gate to all qubits (as what happens during Grover's algorithm), then the general form is given by

$$H_n = \frac{1}{\sqrt{2}} \begin{pmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{pmatrix}$$

In my construction of a Hadamard matrix, I used the following formula which results in the same matrix:

$$(H_n)_{ij} = \frac{(-1)^{i \cdot j}}{2^{n/2}}$$

Now we need a matrix that does the phase flip of the $|0^n\rangle$ state. This consists of the identity matrix except with the first 1 flipped to a -1. Thus, for 2 qubits, it would be

$$I_{-1,2} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Thus, altogether, the Grover diffusion operator is given by

$$U_s = H_n I_{-1,n} H_n$$

And the entire Grover step is given by

$$U_\omega U_s = U_\omega H_n I_{-1,n} H_n$$

Now that we have defined a Grover step, it easy to define the rest of the algorithm:

Grover's algorithm (searching over $N = 2^n$ possibilites):

1. Initialize all n qubits to $|1\rangle$.
2. Apply the Hadamard gate to all qubits.
3. Apply the Grover step, $U_\omega U_s$, approximately $\frac{\pi}{4}\sqrt{N}$ times.
4. Measure qubits.

Notice that we only have to apply the Grover step $\frac{\pi}{4}\sqrt{2^n}$ times. This means that the algorithm has a time complexity of $O(\sqrt{N})$, a massive speedup from the $O(N)$ time that the classical algorithm of checking each entry requires.

Here is my implementation of Grover's algorithm in Racket:

```
(define Grover (lambda (input-U_omega) ; An implementation of Grover's algorithm,
    input-U_omega is a matrix representation the oracle operator
   (let ([steps 0] [qubits (exact-round (/ (log (matrix-num-cols input-U_omega)) (log
       2)))]) ; Requires log(N) qubits where N is the width of the matrix representing
       U_omega
      (cond
         [(= qubits 1) (set! steps 0)]
         [(= qubits 2) (set! steps 1)]
         [else (set! steps (exact-round (* (/ pi 4) (sqrt (expt 2 qubits)))))]) ; # of
             steps ~pi*sqrt(N)/4

      (set! U_omega input-U_omega)
      (display "The number of required qubits is ") (displayln qubits)
      (display "Number of operations required is ") (displayln (+ 1 steps))

      (initialize-register (build-list qubits (lambda (x) 0))) ; Initialize all qubits to
          |0>

      (Hadamard register) ; Apply a Hadamard gate to all qubits

      (for ([i steps])
         (Hadamard (phase-flip-0-state (Hadamard (Oracle register))))) ; Apply the Grover
             Diffusion operator
      )))
```

Note how I apply a Hadamard gate on all qubits with *(Hadamard register)*, and how this will result in a new register. I can thus apply two series of Hadamard gates with *(Hadamard (Hadamard register))*. The rest of the code for Grover's algorithm can be found in appendix B.

4.3 Generating the Oracle

To actually make Grover's algorithm useful, we need a way to generate the oracle from our search function $f(x)$. This is done by first representing our search function as a series of AND, OR and NOT gates in a classical circuit, and the simulating that circuit with a quantum circuit. Because OR is not reversible (the inputs of an OR gate cannot be recovered from the result), we will need more qubits than there were bits in the classical circuit, but this increase is by a constant factor for a given N.

The quantum NOT gate is the same as the classical one. It is given by the matrix

$$NOT = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

We will need to simulate both the AND gate and the OR gate with different quantum circuits. Luckily we have the Toffoli gate, which can do both, and is given by the matrix:

$$CCNOT = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

The Toffoli gate is also called the CCNOT gate, or controlled-controlled-not gate, because it only performs NOT on the third qubit if the other two are both $|1\rangle$. Thus the Toffoli gate on its own functions as an AND gate for the first two qubits, while the third becomes the result of the AND operation.

Now we only need OR. We can define OR as the boolean expression $\neg(\neg x_1 \wedge \neg x_2)$. This means that we can simulate the OR gate with 1 Toffoli gate and 5 quantum NOT gates.

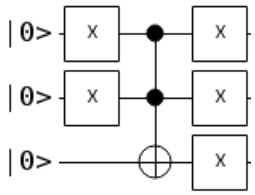


Figure 4.1: Representation of OR gate using 1 Toffoli gate and 5 NOT gates

For every AND and every OR in the classical circuit we are given, we will need one extra qubit. The extra qubit is always the third bit for the Toffoli gate. By using the quantum equivalents of the classical logic gates, and introducing extra qubits when necessary, we now have every part necessary to simulate the classical circuit on the quantum circuit such that the result of the classical circuit is produced by the quantum one. However, this is not yet our oracle.

At this point, we have a ciruit takes the input $|x\rangle \left|0^k\right\rangle$ and gives $|junk(x)\rangle |f(x)\rangle$ where $f(x)$ is our classical search function. However, we need it to be of the form [4]

$$|x\rangle |q\rangle \rightarrow |x\rangle |f(x) \oplus q\rangle$$

What this means is that we need to preserve the state of $|x\rangle$ through the circuit. To accomplish this, we can use a technique called "uncomputation" for the quantum ciruit. Because every quantum gate is reversible, to get $|x\rangle$ back, we just have to perform all the gates we applied in backward order. However, before we do that, we copy the result of the search function onto a new qubit using a CNOT gate. Then we can perform the uncomputation on the rest of the qubits, resulting in the desired form for the output.

At this point, the general steps for our algorithm are:

1. Given an input function in the form of a classical circuit, we simulate each classical gate with a quantum gate, taking note of the order and which qubits we applied the gates to.

2. Copy the output of the function, which should be the state of nth qubit, to an extra qubit.

3. Perform the uncomputation by applying the same quantum gates we applied before but in the opposite order.

After we have achieved this circuit, we then need to complete the final step necessary for the oracle. By definition, the oracle flips the phase of the system state for a single input. This means that if the output qubit is in the $|1\rangle$ state, then we want to flip the phase of the state that produced. This can be done with one more extra qubit and a Pauli-z gate with a control on the ouput qubit.

However, we still have one final problem: the output qubit needs to be returned to its original state from before the oracle. The simplest way to return it to its original state is performing the simulated classical circuit once again. After this, we have will a quantum circuit that fully implements the oracle for Grover's algorithm.

To construct a circuit that implements the oracle for Grover's algorithm:

1. Obtain a classical circuit, called C_c that implements the desired search function using AND, OR and NOT gates.
2. Using quantum NOT and Toffoli gates, simulate the classical circuit with a quantum one, called C_q. (Note: for $N = 2^n$ possible solutions to the search function, and g classical gates, this quantum circuit will require $n + g$ qubits.)
3. Copy the output of this circuit to another qubit (called the output qubit) by using a quantum controlled NOT gate.
4. Using one more qubit, perform a Pauli-z gate on this extra qubit with a control on the output qubit. Note that the oracle function will require a total of $n + g + 2 = n_q$ qubits.
5. Apply C_q again to return the output qubit back to its original state.

My implementation in Scheme of this algorithm for generating the oracle can be seen in appendix C.

4.4 Conclusion

By representing a quantum computer as a series of matrix operations, I was able to successfully simulate quantum computation as well as perform Grover's algorithm. Furthermore, I was able to come up with a simple algorithm for generating a quantum circuit of only common quantum gates that simulates a given classical circuit. As this is a necessary step in order to use Grover's algorithm in a real world situation, this demonstrates how quantum computers might be used in the future. In the next chapter, I show how to implement an adaption of Grover's algorithm to the problem of associative memory.

Chapter 5

Implementing Associative Memory on a Quantum Computer Simulator

5.1 Introduction

Associative memory deals with the problem of "learning" a set of patterns, and then being able to "recall" a certain pattern when presented with just a part of the pattern. There are many simple classical algorithms that can achieve this task, such as storing the patterns in a database and then searching over the database (and each pattern) to find the one that matches. However, these can be slow and require lots of storage space. Another classical solution to this problem is using a Hopfield network, a type of neural network which can be used to implement associative memory. However, if the patterns are of length n, the neural network requires n neurons, and is then limited in how many patterns it can store (usually less than half of n).

Quantum computing allows a significant improvement over neural network techniques. Dan Ventura and Tony Martinez, in a paper titled *Quantum Associative Memory*, present two algorithms that achieve associative memory that allow up to $m = 2^n$ patterns to be stored on n qubits, with a training complexity of $O(mn)$ and a recall complexity of $O(\sqrt{n})$ [19].

In order to show how this algorithm might actually be implemented on a universal quantum computer, each stage of Ventura's and Martinez's (V&M) paper was achieved and simulated using only common quantum gates. Unless noted, all parts of the algorithms are taken directly from their paper.

5.2 Explanation of Algorithms

5.2.1 Overview

Given a set P of patterns to be learned, each pattern is loaded one at a time onto the n qubits until the system state has an equal distribution of all the patterns present. A slightly modified Grover's algorithm can then be used to retrieve a pattern based on a known portion of it.

5.2.2 Learning Stage

For patterns of length n, n qubits are needed to store the pattern (called the storage qubits). However, during the learning stage, three extra qubits are necessary: one as an intermediate qubit which is always returned to $|0\rangle$ (called g), and two as control qubits (called c_1 and c_2). Note that this is a slight modification of V&M's algorithm, which required $2n + 2$ qubits. This is because I

used a Z-gate with controls on every one of the storage bits for the sake of much shorter simulation times.[1] For the algorithm, we require two special gates:

$$\text{C0NOT} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{CS}_\text{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{\frac{p-1}{p}} & \frac{-1}{\sqrt{p}} \\ 0 & 0 & \frac{1}{\sqrt{p}} & \sqrt{\frac{p-1}{p}} \end{pmatrix}$$

C0NOT is simply a controlled-not gate that flips the second bit if the first is in the $|0\rangle$ state, instead of $|1\rangle$. It can be constructed by doing a NOT on the first qubit, a CNOT, and then another NOT on the first.

CS_p allows us to slice the state with the control bit as a one into two states, one being $\frac{1}{n}$ of the superposition, and the other being $\frac{p}{n}$ of the superposition.[2]

Here are the steps of the algorithm for each pattern in P:

Learning a single pattern on n qubits:

1. Apply C0NOT gates with a control on c_2 to storage qubit if the corresponding bit in the pattern is a 1.
2. Apply CNOT gate with control on c_2 to c_1.
3. Apply CS_p with control on c_1 onto c_2, where p is the number of patterns remaining including the pattern being learned.
4. Apply NOT gates to every qubit where the corresponding bit in the pattern is a 0.
5. Apply CNOT gate with controls on every storage qubit to g.
6. Apply CNOT gate with control on g to c_1.
7. Repeat step 5.
8. Repeat step 4.

We can repeat this algorithm for every pattern until P has been "learned" by the quantum computer. By the end of the algorithm, all three extra qubits, g, c_1 and c_2, will be unentangled from the rest of the system so we can safely discard them.

Note that is single pattern learning algorithm is $O(n)$, and as we must apply it for m patterns, the entire learning algorithm is $O(mn)$.

[1] Any single qubit gate U with an arbitrary number of controls can be achieved by using a series of CNOT gates and twice the qubits.

[2] A better and more in depth explanation of how this gate works can be found in V&M's paper.

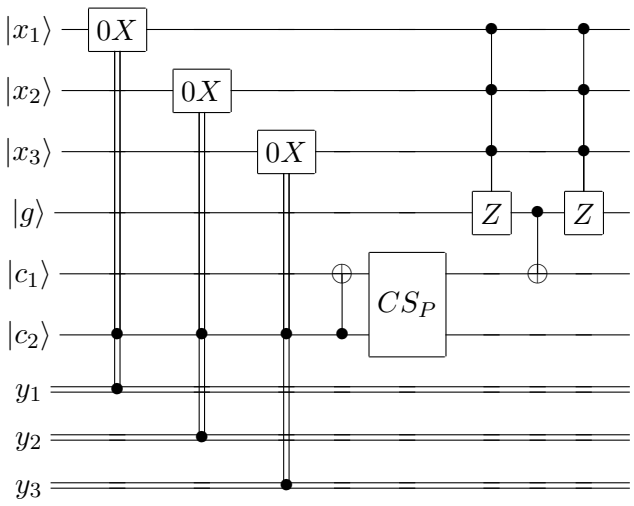


Figure 5.1: This example is for loading patterns with a length of 3. The pattern would come in loaded on the classical bits, y_1, y_2 and y_3. The 0X gate represents a controlled NOT except that the control looks for a zero on $c2$.

5.2.3 Recall algorithm

As previously stated, the recall algorithm uses a modified Grover search to find the correct learned pattern. The first difference is that we do not prepare the system in any way by applying Hadamard gates at the beginning, as we already have the superposition we want. The second is that at the beginning, since the Grover diffusion operator will also flip the phases of all the states representing the patterns (instead of just the desired pattern), we need another operator that flips all the phases of the patterns back. We call this operator I_P. In terms of its matrix, it can be represented as an identity matrix where the ith 1 is changed to a -1 if i is the decimal representation of one of the patterns. We can construct it with gates by going through every pattern, applying a NOT gate on the qubits that correspond to zeros in the pattern, applying a controlled Pauli-z gate with controls on every qubit (except the one with the Pauli-z gate, which can be any of them), and then re-applying the same NOT gates. This method can be seen in the function *constr-patterns-phaser* in my code. We will notate the normal Grover diffusion operator with G.

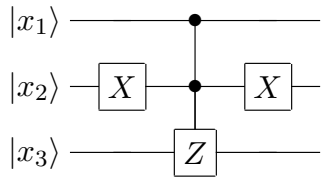


Figure 5.2: This example represents one pattern for I_P. The pattern would be (1 0 1), so we need to apply the NOT gate to the second qubit because it is a zero.

We want the oracle for Grover's algorithm to be a quantum circuit which flips the phase of the system if it is in the correct pattern. Thus, we can construct it simply by applying NOT gates to all qubits that correspond to zeros in the part of the pattern we know, applying a controlled Pauli-z gate with controls on all the qubits corresponding to bits we know onto one of the qubits corresponding to the qubits we do not know, and then applying the same NOT gates again. This method can be seen in the function *constr-search-phaser* in my code. We will notate this oracle function with I_s.

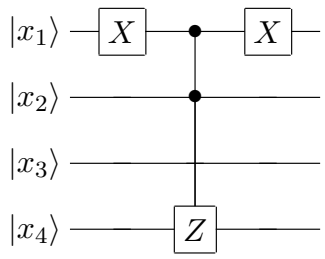


Figure 5.3: This example would correspond to the partial sequence (0 1 ? ?). We apply a NOT gate to the first qubit because we know it is a zero, then a Z with controls on the known qubits with a target on one of the unkowns, and then re-apply the NOT.

Recalling pattern given partial sequence

1. Apply $I_s G I_P G$ to the qubits.
2. Repeat T times:
   (a) Apply $I_s G$ to the qubits.
3. Measure register.

Where

$$T = \frac{\frac{\pi}{2} - \arctan\left(\frac{k}{l}\sqrt{\frac{r_0+r_1}{N-r_0-r_1}}\right)}{\arccos\left(1 - 2\frac{r_0+r_1}{N}\right)}$$

and

$$k = 4a - ab + \frac{r_1}{r_0 + r_1}$$

$$l = -ab + \frac{2a(N + p - r_0 - 2r_1)}{N - r_0 - r_1} - \frac{p - r_1}{N - r_0 - r_1}$$

$$a = \frac{2(p - 2r_1)}{N}$$

$$b = \frac{4(p + r_0)}{N}$$

and where p is the number of patterns, r_0 is the number of marked states (states involving the unknown qubits that the controlled Pauli-z gates in I_P did not act on, or, in the matrix representation, all the states corresponding to the -1s in the matrix) that do not correspond to learned patterns, and r_1 all the marked states that do correspond to learned states. In my code, I assumed that there was only one learned state that the recall should return, so I assumed r_1 is 1 and r_0 is equal to $2^u - 1$ where u is the number of unknowns in the partial sequence.[3]

Though I have no understanding of T, V&M assert that the second algorithm is still $O(\sqrt{n})$ and the same as Grover's algorithm. This makes sense in that though it was modified slightly, the algorithm remains essentially the same.

[3]If you play around with a few more examples, you might notice that the code will not always produce the correctly matched string. This is covered in the conclusion.

5.3 Implementation and Usage

The implementation of quantum associative memory written for my quantum computer simulator has been included in appendix D.

5.4 Discussion and Conclusion

Though in general I was able to successfully implement most parts of the algorithms outlined by V&M, there was one notable issue that I had: The calculation of T often gave me the wrong value. As a result, the code would often apply the oracle and Grover diffusion operator too many times and thus the final superposition was not likely to return the correct value. For example, when 6 patterns of length 4 are learned, and a partial sequence with two unknowns is used to recall, V&M's definition of T gives $T = 3$. However, repeating the Grover step results in a final superposition with $\approx 26\%$ chance of returning the wrong pattern, and only a $\approx 9\%$ chance of retrieving the correct pattern. If we set $T = 0$, however, the final system state has a $\approx 84\%$ chance of returning the correct pattern. While this may be due to a mistake in my code or in my implementation of it, I have always been able to achieve a high rate of getting back the correct pattern using different values of T, which suggests there is an issue with my (or V&M's) calculation of T.

In the next chapter, I discuss the quantum Fourier transform and how to achieve it with simple quantum gates.

Chapter 6

The Quantum Fourier Transform

6.1 Overview

The discrete Fourier transform (DFT) is a function which takes a "signal" in time space and breaks it down into its constituent frequencies, such that if you were to add up all of those produced frequencies, you would return to the original function. For example, we could record some music by recording regular samples of the sound signal, and then use the DFT to examine which sound frequencies were present and their amplitudes. The DFT has wide ranging applications from music to radio astronomy.

6.2 QFT with a Matrix

The quantum Fourier transform is the unitary discrete Fourier transform on the state of the quantum computer. The classical unitary discrete Fourier transform is defined as a map $F : \mathbb{C}^N \to \mathbb{C}^N$ where $N = 2^n$ and

$$x_k \to \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} e^{2\pi i k \frac{m}{N}} x_m$$

where x is a vector of the samples. Note that this is slightly different than the classical DFT as defined in the previous section because it is unitary, and so it preserves the inner product of two valid vectors as input. In this case, it just means normalizing the output with a factor of $\frac{1}{\sqrt{2}}$.

The quantum version is almost exactly the same, except that we transform quantum states to other quantum states according to

$$|x_k\rangle \to \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} e^{2\pi i k \frac{m}{N}} |x_m\rangle$$

where $|x_k\rangle$ is the current complex magnitude of state $|k\rangle$. If we let $\omega_n = e^{2\pi i/N}$ (leaving it as just ω where the n value is obvious), we can simplify this to

$$|x_k\rangle \to \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} \omega^{km} |x_m\rangle$$

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_k \\ \vdots \\ x_{N-1} \end{pmatrix} \rightarrow \frac{1}{\sqrt{N}} \begin{pmatrix} \sum_{m=0}^{N-1} \omega^{0 \cdot n} x_m \\ \sum_{m=0}^{N-1} \omega^{m} x_m \\ \sum_{m=0}^{N-1} \omega^{2m} x_m \\ \vdots \\ \sum_{m=0}^{N-1} \omega^{km} x_m \\ \vdots \\ \sum_{m=0}^{N-1} \omega^{(N-1)m} x_m \end{pmatrix}$$

As we can see, each value in the resulting vector is a linear combination of the values of the original quantum state. This suggests that it is both a valid quantum transformation (as all quantum transformations should be linear and reversible) and that we can represent it as a matrix multiplication. In fact, it is easy to see how the matrix will work: each row should pertain to a different sequence of rotations. The first row should be simply be all 1s, as the first value of the resulting vector is just a sum of all the original states. In other words, it should not rotate the original values at all, meaning it should look like $\omega^0, \omega^0, ..., \omega^0 = 1, 1, ..., 1$. The next row should be the sequence of increasing powers of ω, as $k = 1$. This should look like $\omega^0, \omega^1, \omega^2, ...\omega^{N-1}$. The next row should be a sequence of ω where the power goes up by two every time (because $k = 2$), so $\omega^0, \omega^2, \omega^4, ...\omega^{2(N-1)}$. Following this pattern gives us the matrix which performs the DFT on a quantum state:

$$F_N = \begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{pmatrix}$$

6.3 Gate-Based QFT

We have shown that we can construct a valid matrix operator that performs the QFT on any quantum state. However, this does not tell us very much about how one would go about actually constructing the machine that would perform this. To show this, as well as to demonstrate how well the algorithm will run on a quantum computer, we want to break down the transformation into a series of simple quantum gates.

Luckily, the QFT can be easily broken down into simple quantum gates. In fact, we can define the set up of the gates recursively: as long as we have the series of gates the works on $n-1$ qubits, it is easy to produce the next gates which compute it on n qubits.

6.3.1 Single Qubit Case

We will start with the case of performing the QFT on a single qubit. We want to find a series of gates which performs

$$|q_0\rangle \rightarrow F_2 |q_0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} \omega^0 & \omega^0 \\ \omega^0 & \omega^1 \end{pmatrix} |q_0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} |q_0\rangle .$$

Immediately, we recognize that F_2 has the exact same matrix representation as the Hadamard gate. In fact, the Hadamard transform is exactly F_2, the QFT on a single qubit.

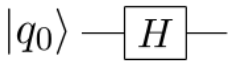


Figure 6.1: The QFT on a single qubit consists of a single Hadamard gate.

6.3.2 Recursive Case

Now let us consider the case where we already have a circuit which performs the QFT on $n-1$ qubits and we want to use this to build one which performs the QFT on n qubits. To do this, we will first simplify the problem by assuming that our original quantum state is actually a classical state, meaning that every qubit is equal to either $|0\rangle$ or $|1\rangle$. In this case, if the original state is equal to $|j\rangle$, the QFT is reduced to

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_k \\ \vdots \\ x_{N-1} \end{pmatrix} \to \frac{1}{\sqrt{N}} \begin{pmatrix} \sum_{m=0}^{N-1} \omega^{0\cdot n} x_m \\ \sum_{m=0}^{N-1} \omega^{m} x_m \\ \sum_{m=0}^{N-1} \omega^{2m} x_m \\ \vdots \\ \sum_{m=0}^{N-1} \omega^{km} x_m \\ \vdots \\ \sum_{m=0}^{N-1} \omega^{(N-1)m} x_m \end{pmatrix} = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 \\ \omega^{j} \\ \omega^{2j} \\ \vdots \\ \omega^{kj} \\ \vdots \\ \omega^{(N-1)j} \end{pmatrix}$$

Note that this is because every value of x vector is 0 except at the single index j where $x_j = 1$, meaning that all the sums in the transform will simplify to just ω^{kj}. We can then break this final state down into the tensor products of the individual qubit states after the transform. First notice that

$$\frac{1}{\sqrt{N}} \begin{pmatrix} 1 \\ \omega^{j} \\ \omega^{2j} \\ \vdots \\ \omega^{kj} \\ \vdots \\ \omega^{(N-1)j} \end{pmatrix} = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 \\ \omega^{2^{n-1}j} \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \omega^{2^{n-2}j} \end{pmatrix} \otimes \cdots \otimes \begin{pmatrix} 1 \\ \omega^{2j} \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \omega^{j} \end{pmatrix}$$

We now assume that we already have a circuit which performs this transform on $n-1$ qubits, except that the output is completely reversed. In other words, the output becomes

$$\frac{1}{\sqrt{2^{n-1}}} \begin{pmatrix} 1 \\ \omega_{n-1}^{j_*} \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \omega_{n-1}^{2j_*} \end{pmatrix} \otimes \cdots \otimes \begin{pmatrix} 1 \\ \omega_{n-1}^{2^{n-3}j_*} \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \omega_{n-1}^{2^{n-2}j_*} \end{pmatrix}$$

instead, where j_* is the j but with the bits in reverse order as well. Because $\omega_{n-1}^{k} = e^{2\pi i k/2^{n-1}} = e^{2\pi i 2k/2^{n}} = \omega^{2k}$, we can rewrite this as

$$\frac{1}{\sqrt{2^{n-1}}} \begin{pmatrix} 1 \\ \omega^{2j_*} \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \omega^{2\cdot 2j_*} \end{pmatrix} \otimes \cdots \otimes \begin{pmatrix} 1 \\ \omega^{2\cdot 2^{n-3}j_*} \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \omega^{2\cdot 2^{n-2}j_*} \end{pmatrix}.$$

We now introduce another qubit q_0 in a classical state, and apply our assumed QFT on the $n-1$ following qubits. This leaves us with the following state:

$$\frac{1}{\sqrt{2^{n-1}}} \begin{pmatrix} 1 \\ \omega^{2j_*} \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \omega^{2\cdot 2j_*} \end{pmatrix} \otimes \cdots \otimes \begin{pmatrix} 1 \\ \omega^{2\cdot 2^{n-3}j_*} \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \omega^{2\cdot 2^{n-2}j_*} \end{pmatrix} \otimes q_{n-1}.$$

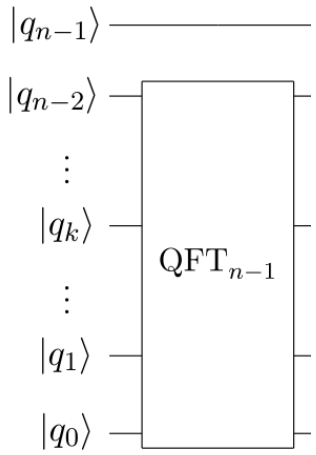


Figure 6.2: The reverse order QFT on qubits 1 through $n-1$.

Let the state of q_n be $|b\rangle$ where b is either 0 or 1. If the state of the first $n-1$ qubits is $|j\rangle$, then the new state is $|j+2^n b\rangle$, because we inserted q_{n-1} at the end of the sequence.

We now make the crucial step: we use a series of controlled phase gates to rotate the first $n-1$ qubits by the correct amount if $b=1$. A controlled phase gate CR_θ on two qubits, with a control on the first, rotates the second qubit by θ radians. It is represented by the matrix

$$CR_\theta = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{pmatrix}$$

If we use a series of these gates, always placing the control on the nth qubit and then rotating the first qubit by $2\pi/N$ radians, the second qubit by $2\cdot 2\pi/N$ radians, the third by $3\cdot 2\pi/N$ radians and so on, we will be left with following quantum state:

$$\frac{1}{\sqrt{2^{n-1}}}\begin{pmatrix}1\\ \omega^{2j_*}\omega^{b}\end{pmatrix}\otimes\begin{pmatrix}1\\ \omega^{2\cdot 2j_*}\omega^{2b}\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}1\\ \omega^{2\cdot 2^{n-3}j_*}\omega^{(n-1)b}\end{pmatrix}\otimes\begin{pmatrix}1\\ \omega^{2\cdot 2^{n-2}j_*}\omega^{nb}\end{pmatrix}\otimes q_{n-1}$$

$$=\frac{1}{\sqrt{2^{n-1}}}\begin{pmatrix}1\\ \omega^{2j_*+b}\end{pmatrix}\otimes\begin{pmatrix}1\\ \omega^{2(2j_*+b)}\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}1\\ \omega^{2^{n-3}(2j_*+b)}\end{pmatrix}\otimes\begin{pmatrix}1\\ \omega^{2^{n-2}(2j_*+b)}\end{pmatrix}\otimes q_{n-1}$$

Because the bit reversal of $2j_*+b$ is $j+2^n b$, if we let $p=j+2^n b$, we can write the state as

$$=\frac{1}{\sqrt{2^{n-1}}}\begin{pmatrix}1\\ \omega^{p_*}\end{pmatrix}\otimes\begin{pmatrix}1\\ \omega^{2p_*}\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}1\\ \omega^{2^{n-3}p_*}\end{pmatrix}\otimes\begin{pmatrix}1\\ \omega^{2^{n-2}p_*}\end{pmatrix}\otimes q_{n-1}$$

The final step we need to take is then put the final qubit, q_{n-1}, into the correct state. We know it should be in the state

$$\begin{pmatrix}1\\ \omega^{2^{n-1}p_*}\end{pmatrix}=\begin{pmatrix}1\\ e^{2\pi i 2^{n-1}p_*/2^n}\end{pmatrix}=\begin{pmatrix}1\\ e^{\pi i p_*}\end{pmatrix}=\begin{pmatrix}1\\ e^{\pi i(b+2j_*)}\end{pmatrix}=\begin{pmatrix}1\\ e^{\pi i b}\end{pmatrix}$$

We can easily accomplish this in the exact same way as in the single qubit case: a single Hadamard gate. We know that

$$H|0\rangle=\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}1\\ 0\end{pmatrix}=\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ 1\end{pmatrix}$$

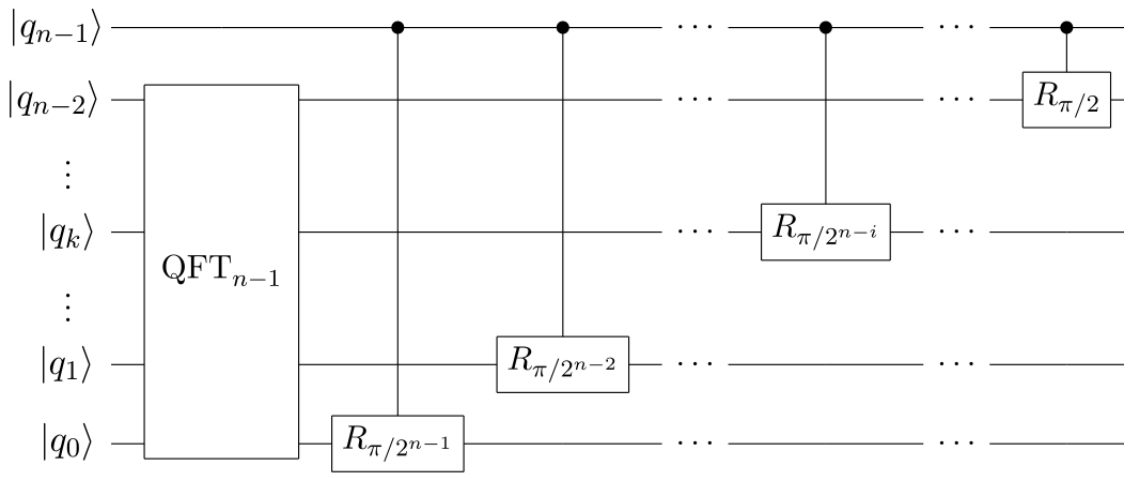


Figure 6.3: The next stage of the circuit. The solid dots represent where the control of the phase gates is placed.

and that

$$H|1\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}$$

This means if we apply the Hadamard gate to qubit q_0, whose state is $|b\rangle$, we get

$$H|b\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1^b\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\e^{\pi i b}\end{pmatrix}$$

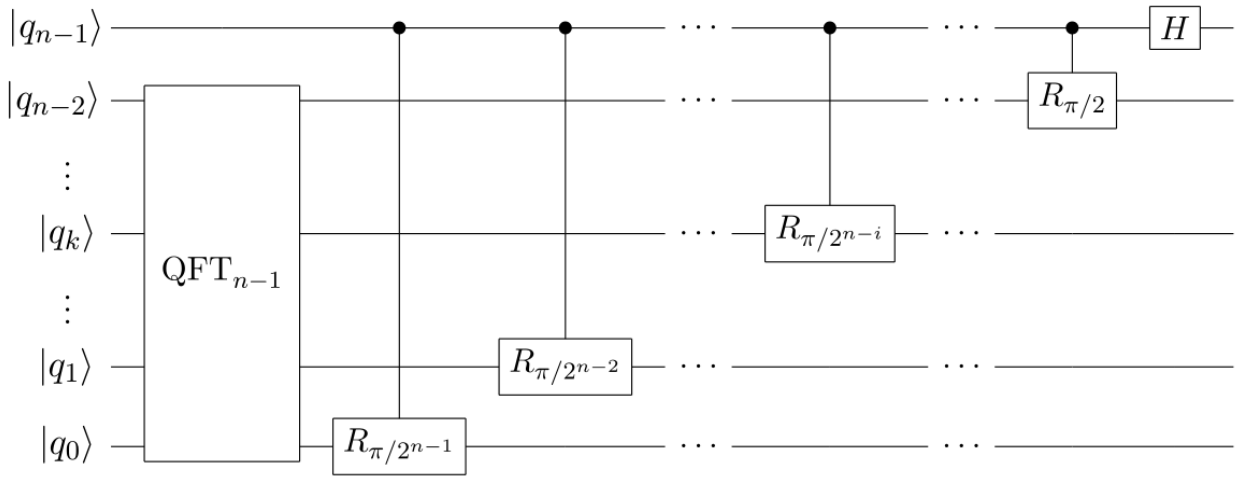


Figure 6.4: The final stage of the circuit, with a Hadamard on the final qubit.

Putting this altogether, we get a circuit which will result in the state

$$\frac{1}{\sqrt{2^{n-1}}}\begin{pmatrix}1\\\omega^{p_*}\end{pmatrix}\otimes\begin{pmatrix}1\\\omega^{2p_*}\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}1\\\omega^{2^{n-3}p_*}\end{pmatrix}\otimes\begin{pmatrix}1\\\omega^{2^{n-2}p_*}\end{pmatrix}\otimes\frac{1}{\sqrt{2}}\begin{pmatrix}1\\e^{\pi i b}\end{pmatrix}$$

$$=\frac{1}{\sqrt{N}}\begin{pmatrix}1\\\omega^{p_*}\end{pmatrix}\otimes\begin{pmatrix}1\\\omega^{2p_*}\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}1\\\omega^{2^{n-3}p_*}\end{pmatrix}\otimes\begin{pmatrix}1\\\omega^{2^{n-2}p_*}\end{pmatrix}\otimes\begin{pmatrix}1\\\omega^{2^{n-1}p_*}\end{pmatrix}$$

This is thus the result of the QFT with the order of the qubits reversed. Because we have shown that we can do the QFT on a single qubit using a Hadamard gate (noting the reverse ordering is the same in the single qubit case), and that if we have a circuit which implements the reverse qubit order QFT on $n-1$ qubits we can get the reverse order QFT on n qubits, we have shown that we can make a circuit implementing the reverse order QFT on any n qubits as long as the qubits are in an initially classical state.

We also know that we can swap the state of any two qubits using a $SWAP$ gate, defined as

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Thus, if it is needed to have a circuit for which the result is in the correct order, we can simply use $SWAP$ gates to obtain the result of the true QFT. This can be done by first reversing the order of the qubits using $SWAP$ gates, and then running the reverse order QFT circuit as defined above.

Finally, we conclude that though we started with the assumption that the qubits were in a classical state, the circuit we came up with will work for any starting quantum state. This is first because any quantum state is a linear combination of all the possible classical states, that is any quantum state $|\Psi\rangle$ with n qubits can be written as

$$|\Psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \alpha_2 |2\rangle + ... + \alpha_{n-1} |n-1\rangle$$

where all $\alpha_k \in \mathbb{C}$. Secondly, because we made sure to only use valid quantum gates in the construction of our QFT circuit, we know that the circuit is also simply a linear transformation of the original state. This means we can think of our circuit simply doing the QFT on each of classical states present in the original state in proportion to each state's magnitude (where α_k is the magnitude of state $|k\rangle$).

To motivate this, we can simply use matrix multiplication on each of the matrix forms of the different gates we used in our circuit to see what the overall function is on two qubits (noticing that we must do this in reverse order of how we apply the gates):

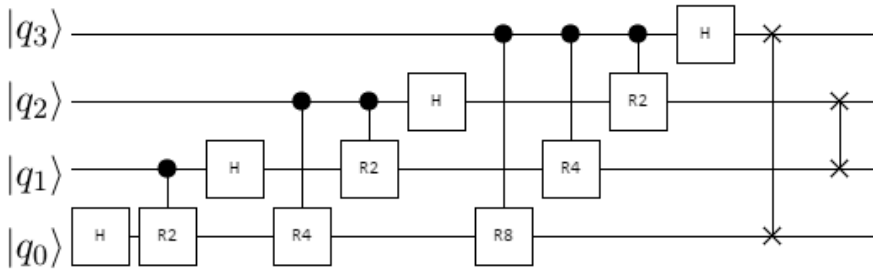
$$(H \otimes I_2) CR_{\pi/2} (I_2 \otimes H) SWAP$$

$$= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} = F_4.$$

Thus we can confirm our circuit works in this case.

Here is an example of the full circuit for 4 qubits: Note that here the X gates are swap gates and R2, R4 and R8 gates correspond to $CR_{\pi/2}$, $CR_{\pi/4}$ and $CR_{\pi/8}$ respectively.



6.4 Conclusion

The QFT is surprisingly efficient. In the kth recursive stage of the QFT, we use exactly k gates ($k-1$ controlled phase gates and one Hadamard). This means that overall, we use $1+2+3+...+n = \frac{n(n+1)}{2} = O(n^2)$ operations (generally, the $SWAP$ gates are not counted as they are not necessary). However, there exist even more efficient implementations of the QFT using only $O(n \log n)$ gates.

One important difference between the QFT and the classical DFT is that the QFT runs on quantum states and produces the result of the transform as quantum state, meaning that we would not be able to examine exactly what the result of the transform was. This we run into the fundamental problem that we can only "measure" a quantum state, meaning that it collapses to a single classical value and the rest of the quantum information is lost. However, the QFT still plays a very important role as a subroutine of other quantum algorithms. One notable one is that the QFT is a part of Shor's algorithm which is the best known factoring algorithm.

In the next chapter, I explain a different model of quantum computing called quantum annealing, and explain how it can be used to solve a certain type of a physics problem.

Chapter 7

Embedding 3D Lattice Ising Problems on the D-Wave Processor

7.1 Introduction to the Ising Model

The Ising Model is a mathematical model inspired by ferromagnets in physics. Ising problems consist of a selection of n discrete spin variables (s, a vector of length n), weights placed on each variable (h, a vector of length n), and weights on each pair of variables (J, an $n * n$ matrix). Note that the diagonal entries of J have no effect, and that $J_{ij} = J_{ji}$. For every spin variable $s_k \in s$, $s_k \in \{-1, 1\}$. The energy for a problem h and J and a given spin state s is given by

$$E(s) = \sum h_i s_i + \sum_{i,j>i} J_{ij} s_i s_j.$$

Our aim is to find good ways of finding low energy states for a given problem h and J. If we are able to find a state that has the lowest possible energy, or the *ground energy*, it is called a *ground state*.

Generally, the *dimension* of an lattice Ising problem is understood to be the dimension of the lattice, not of the connectivity graph itself. This means a 1-dimensional lattice problem is a string with a connection between consecutive qubits, a 2-dimensional problem is a grid where each variable is connected to its four neighbors, and so on. It is worth noting that lattice Ising problems of greater that 2 dimensions have shown to be NP-hard, making the problem interesting for both physicists looking for good ways of making models as well as computer scientists.

7.2 Simulated annealing

Simulated annealing is a simple but effective heuristic algorithm for solving Ising problems. It is inspired by the physical annealing process. Physicists noted that if you take a hot ferromagnet, apply a magnetic field, and then cool it very quickly, the magnetic particles end up in a state of disorder similar to what would be found in its hot state. However, if you cool it slowly, then the particles will align themselves during the cooling process resulting in a well ordered magnet by the end. This process is called annealing.

Simulated annealing takes this idea and attempts to replicate it. It is a Metropolis algorithm, meaning small changes are made to the system state and accepted if they improve the energy. However, even if the change results in a higher energy state, simulated annealing will accept the change with a certain probability which is a function of temperature. Over the course of the

algorithm, we slowly lower the temperature, meaning worse states will have a higher likelihood of being accepted towards the beginning of the anneal than at the end. This concept allows simulated annealing to work itself out of local minima at the beginning, and then slowly settle in to the minimum it finds itself in at the end.

Given a state s with n spins, a new state s', and the current temperature T, simulated annealing will accept the new state with probability

$$P(s, s', T) = e^{-(E(s')-E(s))/T}$$

Usually, this Metropolis update will be applied a fixed number of times. The entire anneal is divided up into *sweeps*, where in each sweep the Metropolis update is applied to every variable in a random ordering.

An overview of simulated as applied to the Ising model is as follows:

Input: $\beta_i, \beta_f, num\_sweeps, h, J$. (the β values represent inverse temperature, so $\beta = 1/T$)

1. Start with a random state s.
2. Let $\beta = \beta_i, \beta\_step = (\beta_f - \beta_i)/num\_sweeps$.
3. Repeat num_sweeps times:
   (a) For i in a random permutation of $[0, 1, ..., n-1]$:
      i. Let $s' = s$.
      ii. Set $s'[i] = -s'[i]$.
      iii. If $e^{-(E(s')-E(s))\beta} > random(0, 1)$, let $s = s'$.
   (b) Set $\beta = \beta + \beta\_step$.
4. Return s.

We can also adjust how β_step is calculated, as well as make it a function of the current step to allow for varying cooling schedules.

7.3 Quantum Annealing

Quantum annealing is another technique used to solve Ising problems. In the same vein as gate based quantum computing, quantum annealing uses a complex vector Ψ to represent the system state during the anneal instead of a discrete binary one. At the end of the anneal, the state vector is projected onto a spin vector which becomes the result of the process.

For Ising problems, we define a Hamiltonian $H(s)$ as follows:

$$H(s) = \frac{\Delta(s)}{2} \sum_i \sigma_i^x + \frac{\epsilon(s)}{2} \left(\sum_i \sigma_i^z + \sum_{i,j>i} J_{ij} \sigma_i^z \sigma_j^z \right)$$

A Hamiltonian is a quantum operator which describes how the physical system will evolve over time. This can be seen in the Schrödinger equation $H\Psi = E\Psi$, where H is the Hamiltonian operator, Ψ is the system state, and E is a eigenvalue (the energy of the state). As H changes over time, Ψ will also evolve and the energy of the system will change as well.

In the Hamiltonian described above, $s \in [0, 1]$ is the annealing parameter, and σ^x and σ^z are the Pauli-x and Pauli-z matrices respectively. The first expression represents the *transverse field* and the second represents the *longitudinal* field. $\Delta(s)/2$ is the energy scale for the transverse field and $\epsilon(s)/2$ is the energy scale for the longitudinal field. At the beginning of the anneal, when $s = 0$, we have $\Delta(s)/2 \gg \epsilon(s)/2$ such that the transverse field dominates. By the end, at $s = 1$, we have $\epsilon(s)/2 \gg \Delta(s)/2$ so that the longitudinal field, where our desired Ising problem h and J are actually represented, dominates.

The notation σ_i^x means the Pauli-x matrix as applied to the ith qubit. This means it is actually a 2^n by 2^n matrix where n is the number of qubits (or the length of Ψ). It can be found by $\sigma_i^x = \underbrace{I_2 \otimes I_2 ...}_{i \text{ times}} \otimes \sigma^x \otimes \underbrace{I_2 \otimes I_2 ...}_{n-i-1 \text{ times}}$ where I_2 is the 2 by 2 identity matrix. The same logic applies for σ_i^z except with the Pauli-z matrix.

At the beginning of the anneal, when $s \approx 0$, the first expression (the transverse field) will dominate, and the system state should tend towards a superposition of all possible states such that if we "measured" the processor we would see all states with equal likelihood. As the annealing parameter increases, the second expression (the longitudinal field) will begin to dominate.

The *Adiabatic Theorem* states that if this transition from transverse field to longitudinal field is made slowly enough, a quantum annealer will stay in one of the lowest energy states at any given point during the anneal. This is important in that it allows us to be sure of quantum annealing's viability as an approach to solving Ising problems. Otherwise, the technique might be able to solve all Ising problems. However, because of the Adiabatic Theorem, we can be sure that a true quantum annealer will be able to find ground states to any Ising problem (though note it does not place restrictions on the time needed to progress through an anneal in relation to the problem size).

7.4 The D-Wave Quantum Annealer

Though to perform the quantum annealing algorithm on an Ising problem of a reasonable size would be impossible, luckily D-Wave has built several real quantum annealers using superconducting loops of niobium wire. Due to physical limitations, they chose to build their chips using a Chimera structured graph, pictured below. A Chimera graph consists of *cells* of $k_{4,4}$ subgraphs which then have four qubits connected vertically to the corresponding qubits in the cells above and below, and four connected horizontally to corresponding qubits. The edges connecting qubits are also called couplers.

Though there are many parameters that the D-Wave chip accepts, the main process of using it to solve problems was to send a problem h and J, and then receive a number of "reads" back that were the solutions the chip found.

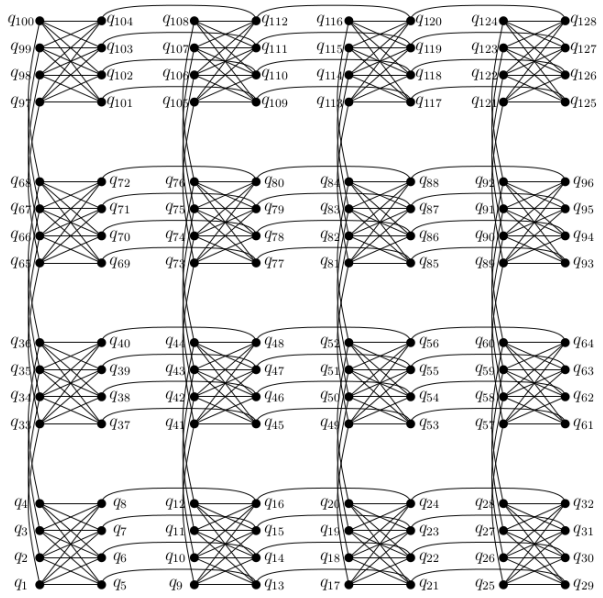


Figure 7.1: 16 Chimera cells, called a $C4$

7.5 Graph Embedding

The fixed shape of the Chimera graph of the D-Wave annealer introduces the problem of running graphs with non-Chimera topologies. Obviously not all interesting Ising problems can be simply placed onto a Chimera graph using a single qubit for every problem variable and single coupler for every problem edge. For instance, I wanted to find the best ways of running 3D lattice Ising problems on the D-Wave processor.

To get around this, we use several qubits and their couplers to form a *chain* which then represents a single variable from the original problem. If we assign a *chain weight* to the couplers of the chain of low enough negative value relative the rest of the weights in the problem J, the chain should always align itself such that all of its qubits have the same spin. If we can find a set of chains on the Chimera graph corresponding to problem variables such that if there is an edge between two problem variables there is also a coupler in the Chimera graph between at least one pair of qubits in each chain, then we have found an embedding of the problem graph for the Chimera.

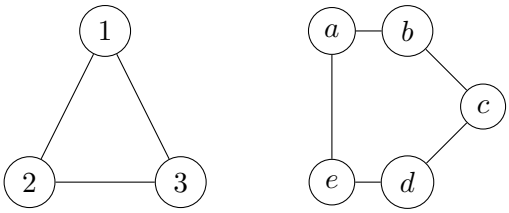
To make this more formal, for an embedding of a graph G with n vertices $V_G = \{v_1, v_2, \ldots, v_n\}$ on graph H with m vertices $V_H = \{u_1, u_2, \ldots, u_m\}$, we define an embedding $M : v_i \in V_G \to \mathbf{s}_i \subset V_G$. For M to be a valid embedding, there are three requirements:

1. All $\mathbf{s}_i$ must be disjoint with each other, i.e. for all $1 \leq i < j \leq n$, $s_i \cap s_j = \emptyset$.

2. All $\mathbf{s}_i$ must be connected in H. This means that for all $u_j \in \mathbf{s}_i$, there is at least one other vertex $u_k \in \mathbf{s}_i$ such that $u_j u_k \in E(H)$.

3. If there is an edge $v_i v_j \in E(G)$, then there are some u_{i*} and u_{j*} such that $u_{i*} u_{j*} \in E(H)$. This ensures that for any edge in G, there is at least one corresponding edge in H that we can map to.

For an example, let's take k_3 graph (called G with vertices $1, 2, 3$) and embed it in a cyclic graph with 5 vertices (called H with vertices a, b, c, d, e).



Now we can create an embedding by noticing that we can group b and c together and e and d in H to get the same graph as G. Let M be our embedding and let

$$M(1) \to \{a\}$$
$$M(2) \to \{e, d\}$$
$$M(3) \to \{b, c\}$$

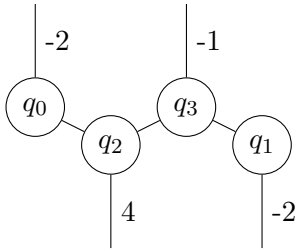
M is a valid embedding because the sets of vertices in H it produces are disjoint, the sets are connected, and because each edge in G can be mapped to at least one edge in H ($12 \to ae$, $23 \to dc$, and $13 \to ab$).

Most of the time, there is not a single way to embed a given graph onto another (in the example, we could have mapped $1 \to \{a, b, c\}$ and still gotten a valid embedding). When embedding an Ising problem on the D-Wave hardware, how the embedding is done affects the performance of the hardware on the original problem. Several factors such as average chain length, how different the chain lengths are in the embedding, how symmetric the chains are, chain strength, and the number of available couplers between chains, all affect how easily the hardware is able to find good solutions to the original problem.

7.5.1 Algebraic Chain Strength

The algebraic chain strengths of the chains in an embedding for a given problem h and J is defined as the minimum absolute value that we can assign to the weight of the edges of the chains such that every ground state of the embedded problem corresponds to a ground state of the original (or *logical*) problem. This is equivalent to stating that every ground state of the embedded problem will have no *broken chains*, or chains where some of the qubits are spin up and the others are spin down. There is a simple way to calculate the algebraic chain strength. For each edge in the chain, calculate the sum of the absolute values of the edge weights coming off the entire chain on side of the edge, and then on the other side. The algebraic chain strength is the minimum of two values.

Consider an example: a chain of 4 qubits q_0, q_1, q_2, q_3 such that there is chain edge from q_0 to q_1, q_1 to q_2, and q_2 to q_3. Each of these qubits has single edge coming off of it representing an original problem edge. Now examine the edge in the middle, q_1q_2. On one side, there are two qubits q_0, q_1 for which the sum of the absolute values of the logical edges is 6. On the other side, qubits q_2, q_3 have a sum of the absolute edge weights of 3. Thus the algebraic chain strength for this chain is $min(6, 3) = 3$.



7.6 Embedding 3D Lattice Ising Problems

Physicists are interested in 3D lattice Ising problems, where the problem graph is a 3D grid. This means every variable in paired with 6 others, or 2 each in the x, y and z directions. I looked at two subsets of these problems: $4 * k * k$ lattices and $8 * k * k$ lattices.

For $4 * k * k$ lattices, there is a simple embedding for the Chimera graph where every chain has two qubits. Thus each Chimera tile (a $k_{4,4}$) has a $4 * 1 * 1$ column of the original problem lattice embedded in it. All of the chains are symmetric with an algebraic chain strength of 3. The vertical z-axis edges of the column were all contained within a single tile, and then the x and y-axis edges became the couplers going between tiles. Since there is a single obvious way to do this embedding, I did not spend much time on it.

For $8 * k * k$ lattices, the best embeddings use chains of 4 qubits. A single $8 * 1 * 1$ column of the lattice is embedding into a square of 4 Chimera tiles, called a $C2$ (note a $C2$ has $4 \cdot 8 = 32$ total qubits, allowing 8 chains of 4 qubits each). In each of the embeddings, the chains take the same shape, and the x and y-axis edges become the couplers connecting $C2$'s together. The only differences were in the placement of the z-axis edges, which could either be connected to the two qubits on either end of the chain, or to the two qubits in the middle of the chain. For each of these possible embeddings, it is also possible to "split" the z-axis edge and put it on two couplers, as each chain is connected two every other with two couplers. Instead of putting the full weight of the original lattice edge, we use $1/2$ the original weight on each coupler. In total, this allowed for 4 embeddings. I worked primarily on testing which of these embeddings produced the best results.

Note: In the following diagrams, the left side has a C4 ($4 * 4$ Chimera cells) with a $8 * 4 * 4$ lattice embedded on it, where blue couplers belong to a chain, and red edges go between chains corresponding to the original lattice edges. On the right, a single chain is shown with a similar color scheme (z-axis couplers are highlighted with green as they are the only edges that change among the embeddings).

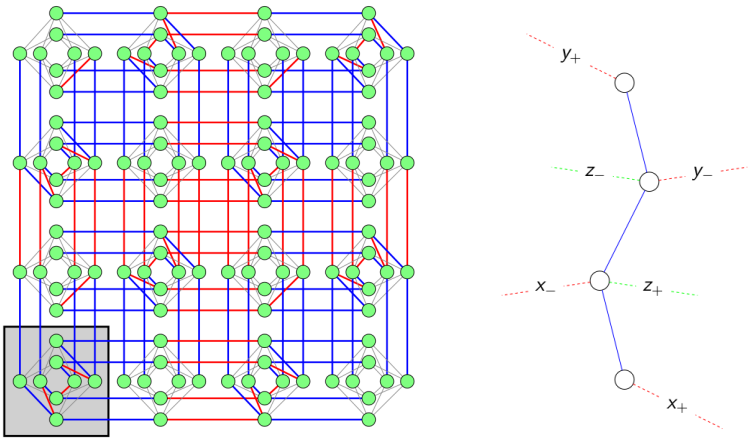


Figure 7.2: Left: Embedding 1. A C4 with the first embedding I did. The C4 embeds a $8*2*2$ lattice. **Right:** A single chain diagram for this embedding.

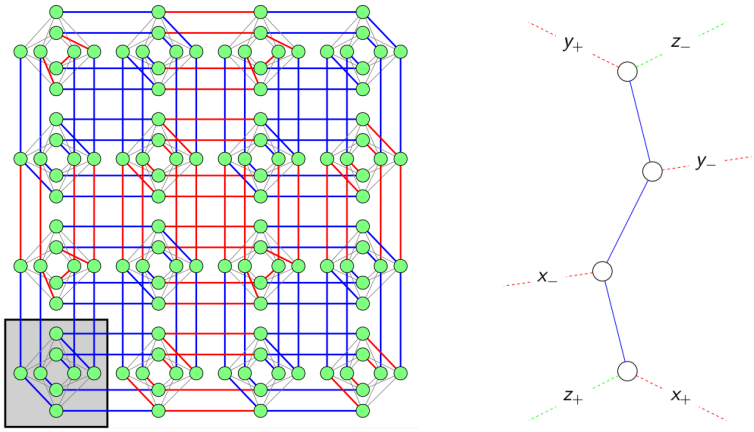


Figure 7.3: Left: Embedding 2. Instead of placing z-axis edges on the middle of the chain, they were placed on the end. **Right:** A single chain diagram for this embedding.

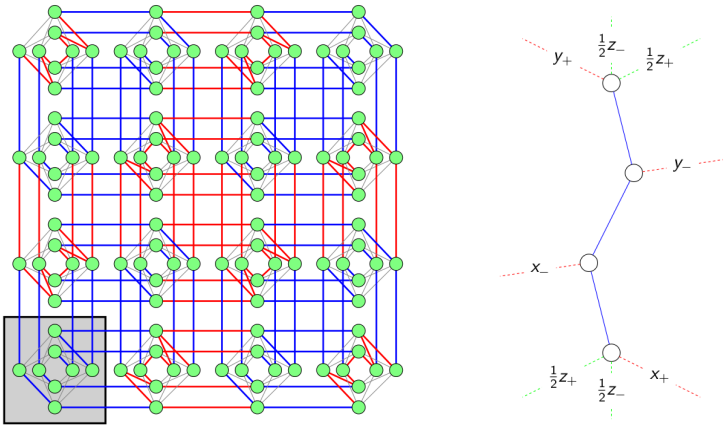


Figure 7.4: Left: Embedding 3. Similar to embedding 2, but the z-axis edges are spread across the two available couplers. **Right:** A single chain diagram for this embedding. Note that this will reduce the algebraic chain strength to 2 instead of 3 as with embedding 2.

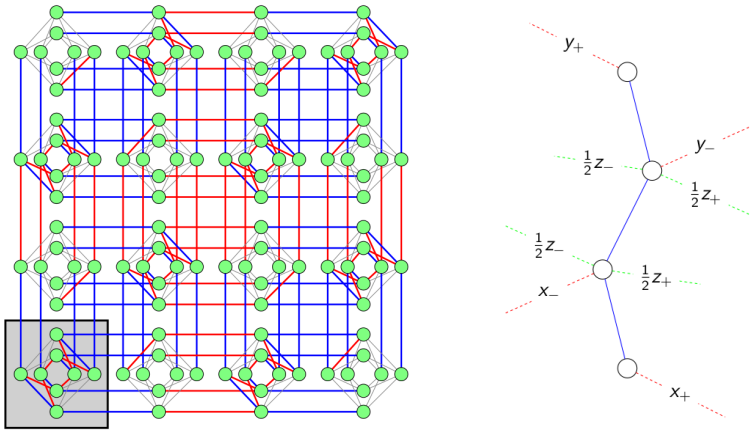


Figure 7.5: Left: Embedding 4. Similar to embedding 1, but with split z-axis edges. **Right:** A single chain diagram for this embedding. Again, the algebraic chain strength has been reduced to 2 from 3.

7.6.1 Chain Strength

Apart from testing the four embeddings, another important factor was determining the best chain strength to use for a given embedding. Though it would be ideal to set the chain strength to be the algebraic chain strength or greater of each embedding, this does not always give the best results. This is because the D-Wave processor has a limited energy scale available, and the energy scale is quantized and not perfectly accurate. This means as you increase the chain strength, the weights of the problem edges must go down and will eventually be distorted.

7.7 Success Statistics

Several different important statistics were used to measure the success an embedding. These were useful in comparing the embeddings and deciding which was best.

1. **Success probability**

   This is simply a measure of how likely the D-Wave chip would find a ground state of the original problem. However, as the problems got bigger and bigger, it became infeasible to get enough reads from the chip to accurately determine the success probability. Thus other statistics were also used.

2. **Mean residual energy**

   A certain state's residual energy is the difference between the energy of the state and the ground energy of the problem. Thus the mean residual energy for a collection of solutions returned is the mean of the residual energies of the states. Though this will give a good indication of how well a technique is at solving some problems, it is still possible to have a low mean residual energy while still having a low success probability (as there may be many easy low energy states but hard to reach ground states).

3. **Broken chain probability**

   The probability that a chain would be broken for a given read. This is also a good indicator of how well an embedding is performing, but if using a post processing technique as mentioned before, it does not directly correlated to success probability or mean residual energy. It is quite possible to have an embedding and chain strength that results in more broken chains but higher success probability, etc.

4. **Time to solution**

   Time to solution is a success statistic based on the amount of "time", or in the case of the D-Wave machine the number of reads, needed such that there is a 99% chance of achieving a ground state. The time to solution t for a success probability s is

   $$(1-s)^t < 0.01$$

   $$t \cdot \log(1-s) < \log(0.01)$$

   $$t > \frac{\log(0.01)}{\log(1-s)}$$

   If we want to compare techniques which have different computation times (as opposed to just using a "read" as a single unit of time), such as with simulated annealing where the computation time is based on the number of sweeps, we can multiply this final value by that time.

By examining these statistics across the four embeddings as well as a variety of chain strengths on random problems, I was able to determine the best embedding and chain strength for use on the D-Wave processor. I also compared the performance of the D-Wave processor against simulated annealing both on the original lattice problems as well as the embedded problems. Since simulated annealing gives a good approximation for how hard a problem is, these served as benchmarks for the D-Wave processor.

Chapter 8

Conclusion

Despite quantum computing only offering a speedup for a limited class of problems, it still offers endless possible applications. In this Keystone I have discussed algorithms that can search a problem space efficiently, perform an associative memory recall, run the discrete Fourier transform, and solve graph problems with applications in physics. However, quantum computing as a field still faces many challenges that it must overcome to be truly useful outside of hypotheticals and academia.

One of the most prominent of these is the physical construction of the computers. D-Wave's accomplishments on the front of quantum annealing, such as their 2048-qubit annealer, are impressive, but it remains difficult to classify their performance. However, recent iterations appear to be beating out the classical solvers on hard Ising problems by large factors [11]. For universal or gate based quantum computers, which are necessary for the more famous algorithms such as Grover's and Shor's, progress is slow. 21 remains the largest number factored on a quantum computer using Shor's algorithm [5]. Scalable gate-based designs remain out of reach.

Another problem faced by quantum computing is that it remains difficult to access for those who are not physicists. If it is to have a large impact, quantum computing will need to be accessible to programmers and people without a physics background. This means a large amount of work will need to be put into developing tooling and better educational resources such that people can start writing applications for these new computers. In the future, I hope to work on creating these resources, and ideally, contribute to the expanding area of quantum computing research.

Bibliography

[1] "Applications: Helping solve some of the most difficult challenges." (2014). Retrieved from http://www.dwavesys.com/quantum-computing/applications

[2] Einstein, A. (1905). Über einen die Erzeugung und Verwandlung des Lichtes betreffenden heuristischen Gesichtspunkt. Annalen der Physik, 322(6), 132-148.

[3] Grover, L. K. (1996, July). A fast quantum mechanical algorithm for database search. *In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (pp. 212-219). ACM.

[4] Kothari, R. (2012, August 9). How much does a quantum oracle to find a needle in a haystack really cost? Retrieved January 20, 2017, from http://mathoverflow.net/questions/102779/how-much-does-a-quantum-oracle-to-find-a-needle-in-a-haystack-really-cost

[5] Martin-Lopez, E., Laing, A., Lawson, T., Alvarez, R., Zhou, X., & O'brien, J. L. (2013). Experimental realisation of Shor's quantum factoring algorithm using qubit recycling. *2013 Conference on Lasers & Electro-Optics Europe & International Quantum Electronics Conference CLEO EUROPE/IQEC*. doi:10.1109/cleoe-iqec.2013.6801701

[6] Mehra, J., & Rechenberg, H. (2001). The historical development of quantum theory (Vol. 1). Springer Science & Business Media.

[7] Mermin, N. D. (2007). Quantum Computer Science: An Introduction. New York, NY: Cambridge University Press.

[8] Moammer, K. (2015, February 25). Intel Abandoning Silicon With 7nm and Beyond - Silicon Alternatives Coming By 2020. *Wccf tech*. Retrieved January 20, 2017, from http://wccftech.com/intel-abandoning-silicon-7nm/

[9] Moore, G.E. (1965). Cramming More Components Onto Integrated Circuits. Electronics, 38(8), 82-85.

[10] Neven, H. (2013, May 16). Launching the Quantum Artificial Intelligence Lab. Retrieved March 10, 2015, from http://googleresearch.blogspot.ca/2013/05/launching-quantum-artificial.html

[11] Neven, H. (2015, December 08). When can Quantum Annealing win? Retrieved January 20, 2017, from https://research.googleblog.com/2015/12/when-can-quantum-annealing-win.html

[12] "Quantum Computing: How D-Wave Systems Work." (2014). Retrieved from http://www.dwavesys.com/quantum-computing

[13] Rojas, R. (2004). A tutorial introduction to the lambda calculus. DOI= http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf.

[14] Schrödinger, E. (1926). An undulatory theory of the mechanics of atoms and molecules. Physical Review, 28(6), 1049.

[15] Simonite, T. (2016, August 15). Intel Puts the Brakes on Moore's Law. *MIT Technology Review*. Retrieved January 20, 2017, from https://www.technologyreview.com/s/601102/intel-puts-the-brakes-on-moores-law/

[16] Smith, Brian. "Quantum Ideas: Week 2" Lecture Notes. *University of Oxford*. Retrieved from https://users.physics.ox.ac.uk/ smithb/website/coursenotes/qi/QILectureNotes2.pdf

[17] Spector, L. (2004). Automatic Quantum Computer Programming: a genetic programming approach (Vol. 7). *Springer Science and Business Media*.

[18] Van Tonder, A. (2004). A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5), 1109-1135.

[19] Ventura, D., & Martinez, T. (2000). Quantum associative memory. *Information Sciences*, 124(1), 273-296.

[20] What is protein folding? (n.d.). Retrieved January 20, 2017, from http://fold.it/portal/info/about

Appendix A

quantum-computer-simulator.rkt

```
  #lang racket
(require math)
(require racket/vector)

(require srfi/1)
(require 2htdp/batch-io)
(define-namespace-anchor a)
(define ns (namespace-anchor->namespace a))

(provide matrix-print initialize-register measure-register apply-gate)
(provide register Hadamard-gate Pauli-X-gate Pauli-Y-gate Pauli-Z-gate CNOT-gate
    QSwap-gate Toffoli-gate)
(provide bits bits->row-matrix set-register) ; for Grover.rkt
(provide G-nqubit-constructor) ; for oracle-constructor.rkt

;--------------------Quantum simulator functions---------------------;

(define matrix-print (lambda (matrix)
  (let ([size (square-matrix-size matrix)])
    (for ([m size]) (for ([n size])
         (display (array-ref matrix (list->vector (list m n)))) (display " "))
       (displayln "")))))

(define bits (lambda (n l) ; Returns a list length l of the digits of n in binary with
    leading zeroes
  (let ([n-bits null])
    (set! n-bits (let loop ((n n) (binary '()))
      (if (= 0 n)
        binary
        (loop (arithmetic-shift n -1) (cons (bitwise-and n 1) binary)))))
    (append (build-list (- l (length n-bits)) (lambda (x) 0)) n-bits))))

(define bits->dec (lambda (n)
  (string->number (string-append "#b" (foldr string-append "" (map number->string
     n))))))

(define bits->row-matrix (lambda (bits)
  (build-matrix 1 (length bits) (lambda (i j) (list-ref bits j)))))
```

```
(define register null)

(define set-register (lambda (psi)
   (set! register psi)))

(define initialize-register (lambda (lq)
   (set! register (array->mutable-array (make-array (list->vector (list 1 (expt 2
       (length lq)))) 0)))
   (array-set! register (list->vector (list 0 (bits->dec lq))) 1)
   (set! register (mutable-array-copy register))))

(define measure-register (lambda ()
   (let ([psi (matrix->list register)] [q-index 0] [max 0] [probabilities null])
      (set! probabilities (map (lambda (qubit) (magnitude qubit)) psi))
      (for ([qubit (length psi)])
         (when (< max (list-ref probabilities qubit))
            (set! max (list-ref probabilities qubit))
            (set! q-index qubit)))
      (display "The most likely result is |")
      (display (~r q-index #:base 2 #:min-width (exact-round (/ (log (length psi)) (log
          2))) #:pad-string "0"))
      (display "> with a probability of ") (displayln (* max max)))))

(define G-nqubit-constructor (lambda (N Q G)
   (let ([Q (reverse Q)] [n (exact-round (/ (log N) (log 2)))] [i-binary '()] [j-binary
       '()] [i-j-differ #f] [Qprime '()] [i-star '()] [j-star '()])
      (set! Qprime (for/list ([index n] #:when (not (member index Q))) index))
      (build-matrix N N (lambda (i j)
         (letrec
            ([i-binary (bits i n)]
            [j-binary (bits j n)]
            [i-j-checker (lambda (Qprime)
               (cond [(null? Qprime)
                     (array-ref G (list->vector (map bits->dec (map reverse
                         (call-with-values
                        (lambda () (for/lists (l1 l2) ([x Q]) (values
                           (list-ref i-binary x)
                           (list-ref j-binary x)))) list)))))]
                  [(not (= (list-ref i-binary (car Qprime)) (list-ref j-binary (car
                      Qprime)))) 0]
                  [else (i-j-checker (cdr Qprime))]))])
            (i-j-checker Qprime)))))))

(define apply-gate (lambda (psi qubits G)
   (let ([new-psi (matrix* psi (G-nqubit-constructor (matrix-num-cols register) qubits
       G))])
      (set! register new-psi)
      new-psi)))

;----------------Quantum gates--------------------;

(define 1oversqrt2 (/ 1 (sqrt 2)))
```

```
(define Hadamard-gate (matrix [
   [1oversqrt2 1oversqrt2]
   [1oversqrt2 (* 1oversqrt2 -1)]
]))

(define Pauli-X-gate (matrix [ ; also known as the NOT gate
   [0 1]
   [1 0]
]))

(define Pauli-Y-gate (matrix [
   [0 0-i]
   [0+i 0]
]))

(define Pauli-Z-gate (matrix [
   [1 0]
   [0 -1]
]))

(define CNOT-gate (matrix [
   [1 0 0 0]
   [0 1 0 0]
   [0 0 0 1]
   [0 0 1 0]
]))

(define QSwap-gate (matrix [
   [1 0 0 0]
   [0 0 1 0]
   [0 1 0 0]
   [0 0 0 1]
]))

(define Toffoli-gate (matrix [ ; also known as the CCNOT gate
   [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]
]))
```

Appendix B

grover.rkt

```
  #lang racket
(require math)
(require racket/vector)

(require srfi/1)
(require 2htdp/batch-io)
(define-namespace-anchor a)
(define ns (namespace-anchor->namespace a))

(require "quantum-computer-simulator.rkt")

(provide generate-fake-Oracle Grover)
(provide phase-flip-0-state) ; for oracle-constructor.rkt

;-----------Constructors for the gates for Grover's algorithm--------;

(define make-Hadamard (lambda (N)
  (let ([constant (exact->inexact (/ 1 (expt 2 (/ (/ (log N) (log 2)) 2))))])
    (build-matrix N N (lambda (i j)
      (* constant (expt -1 (matrix-dot (bits->row-matrix (bits i N)) (bits->row-matrix
          (bits j N)))))))))

(define H-matrix null)

(define Hadamard (lambda (psi)
  (cond
    [(or (null? H-matrix) (not (eq? (matrix-num-cols H-matrix) (matrix-num-cols
        register))))
      (set! H-matrix (make-Hadamard (matrix-num-cols register)))
      (let ([new-psi (matrix* psi H-matrix)])
        (set-register new-psi)
        new-psi)]
    [else (let ([new-psi (matrix* psi H-matrix)])
      (set-register new-psi)
      new-psi)])))

(define pf-matrix null)
```

```
(define make-phase-flipper (lambda (N)
   (diagonal-matrix (cons -1 (build-list (sub1 (matrix-num-cols register)) (lambda (x)
       1))))))

(define phase-flip-0-state (lambda (psi)
   (cond
      [(or (null? pf-matrix) (not (eq? (matrix-num-cols pf-matrix) (matrix-num-cols
          register))))
        (set! pf-matrix (make-phase-flipper (matrix-num-cols register)))
        (let ([new-psi (matrix* psi pf-matrix)])
          (set-register new-psi)
          new-psi)]
      [else (let ([new-psi (matrix* psi pf-matrix)])
        (set-register new-psi)
        new-psi)])))

(define U_omega null)

(define Oracle (lambda (psi)
   (let ([new-psi (matrix* psi U_omega)])
     (set-register new-psi)
     new-psi)))

(define generate-fake-Oracle (lambda (N solution)
   (diagonal-matrix (append (build-list solution (lambda (x) 1)) '(-1) (build-list (sub1
       (- N solution)) (lambda (x) 1))))))

(define Grover (lambda (input-U_omega) ; An implementation of Grover's algorithm,
    input-U_omega is a matrix representation the oracle operator
   (let ([steps 0] [qubits (exact-round (/ (log (matrix-num-cols input-U_omega)) (log
       2)))]) ; Requires log(N) qubits where N is the width of the matrix representing
       U_omega
      (cond
        [(= qubits 1) (set! steps 0)]
        [(= qubits 2) (set! steps 1)]
        [else (set! steps (exact-round (* (/ pi 4) (sqrt (expt 2 qubits)))))]) ; # of
            steps ~pi*sqrt(N)/4

      (set! U_omega input-U_omega)
      (display "The number of required qubits is ") (displayln qubits)
      (display "Number of operations required is ") (displayln (+ 1 steps))

      (initialize-register (build-list qubits (lambda (x) 0))) ; Initialize all qubits to
          |0>

      (Hadamard register) ; Apply a Hadamard gate to all qubits

      (for ([i steps])
        (Hadamard (phase-flip-0-state (Hadamard (Oracle register))))) ; Apply the Grover
            Diffusion operator
      )))
```

Appendix C

oracle-constructor.rkt

```
  #lang racket
(require math)
(require racket/vector)

(require srfi/1)
(require 2htdp/batch-io)
(define-namespace-anchor a)
(define ns (namespace-anchor->namespace a))

(require "quantum-computer-simulator.rkt")
(require "Grover.rkt")

(provide NOT NOT OR OR AND AND Grover-from-classical-circuit U_omega input-qubits
    generate-U_omega)

;--------------

(define Oracle (lambda (psi)
  (matrix* psi U_omega)))

(define total-qubits 0)

(define next-safe-qubit 0)

(define uncomputer null)

(define computer-strings-for-latex "")
(define uncomputer-strings-for-latex "")

(define Pauli-X (lambda (n qubit)
  (G-nqubit-constructor (expt 2 n) (list qubit) Pauli-X-gate)))

(define Toffoli (lambda (n qubits)
  (G-nqubit-constructor (expt 2 n) qubits Toffoli-gate)))

(define NOT (lambda (qubit)
  (set! computer-strings-for-latex (string-append computer-strings-for-latex
      (string-append "\tX\tq" (number->string qubit) "\n"))) ; just for LaTeX output
```

```
   (set! U_omega (matrix* U_omega (Pauli-X total-qubits qubit)))
   (set! uncomputer (matrix* (Pauli-X total-qubits qubit) uncomputer))
   (set! uncomputer-strings-for-latex (string-append (string-append "\tX\tq"
       (number->string qubit) "\n") uncomputer-strings-for-latex)) ; just for LaTeX
       output
   qubit))

(define OR (lambda (qubit1 qubit2)
   (void (NOT qubit1))
   (void (NOT qubit2))
   (set! U_omega (matrix* U_omega (Toffoli total-qubits (list qubit1 qubit2
       next-safe-qubit))))
   (set! computer-strings-for-latex (string-append computer-strings-for-latex
       (string-append "\ttoffoli\tq" (number->string qubit1) ",q" (number->string
       qubit2) ",q" (number->string next-safe-qubit) "\n"))) ; just for LaTeX output
   (set! uncomputer (matrix* (Toffoli total-qubits (list qubit1 qubit2 next-safe-qubit))
       uncomputer))
   (set! uncomputer-strings-for-latex (string-append (string-append "\ttoffoli\tq"
       (number->string qubit1) ",q" (number->string qubit2) ",q" (number->string
       next-safe-qubit) "\n") uncomputer-strings-for-latex)) ; just for LaTeX output
   (void (NOT qubit1))
   (void (NOT qubit2))
   (void (NOT next-safe-qubit))
   (set! next-safe-qubit (+ next-safe-qubit 1))
   (- next-safe-qubit 1))) ; This, by defintion, is the output of the toffoli gate (the
       third qubit), so that's what we want to return

(define AND (lambda (qubit1 qubit2)
   (set! U_omega (matrix* U_omega (Toffoli total-qubits (list qubit1 qubit2
       next-safe-qubit))))
   (set! computer-strings-for-latex (string-append computer-strings-for-latex
       (string-append "\ttoffoli\tq" (number->string qubit1) ",q" (number->string
       qubit2) ",q" (number->string next-safe-qubit) "\n"))) ; just for LaTeX output
   (set! uncomputer (matrix* (Toffoli total-qubits (list qubit1 qubit2 next-safe-qubit))
       uncomputer))
   (set! uncomputer-strings-for-latex (string-append (string-append "\ttoffoli\tq"
       (number->string qubit1) ",q" (number->string qubit2) ",q" (number->string
       next-safe-qubit) "\n") uncomputer-strings-for-latex)) ; just for LaTeX output
   (set! next-safe-qubit (+ next-safe-qubit 1))
   (- next-safe-qubit 1))) ; This, by defintion, is the output of the toffoli gate (the
       third qubit), so that's what we want to return

(define get-extra-qubits (lambda (boolean-expression)
   (cond
      [(null? boolean-expression) 0]
      [(or (eq? 'OR (car boolean-expression)) (eq? 'AND (car boolean-expression))) (+ 1
          (get-extra-qubits (cdr boolean-expression)))]
      [(list? (car boolean-expression)) (+ (get-extra-qubits (car boolean-expression))
          (get-extra-qubits (cdr boolean-expression)))]
      [else (get-extra-qubits (cdr boolean-expression))])))

(define get-base-qubits (lambda (boolean-expression)
   (cond
      [(null? boolean-expression) 0]
```

```
      [(number? (car boolean-expression)) (max (car boolean-expression) (get-base-qubits
          (cdr boolean-expression)))]
      [(list? (car boolean-expression)) (max (get-base-qubits (car boolean-expression))
          (get-base-qubits (cdr boolean-expression)))]
      [else (get-base-qubits (cdr boolean-expression))]))

(define controlled-Z-gate (matrix [
   [1 0 0 0]
   [0 1 0 0]
   [0 0 1 0]
   [0 0 0 -1]
]))

(define U_omega null)

(define input-qubits null)

(define generate-U_omega (lambda (boolean-expression)
   (let ([base-qubits (+ 1 (get-base-qubits boolean-expression))] [temp-matrix '()]) ;
       base-qubits is equal to the # of qubits needed to implement the circuit before
       uncomputation
      (set! input-qubits (- base-qubits 1))
      (set! next-safe-qubit base-qubits)
      (set! total-qubits (+ base-qubits (get-extra-qubits boolean-expression) 2)) ; Two
          extra: one to save the answer before we uncompute, and another for the phase
          flipper
      (set! U_omega (identity-matrix (expt 2 total-qubits)))
      (set! uncomputer (identity-matrix (expt 2 total-qubits)))
      (eval boolean-expression ns)
      (set! U_omega (matrix* U_omega (G-nqubit-constructor (expt 2 total-qubits) (list (-
          total-qubits 3) (- total-qubits 2)) CNOT-gate))) ; Copy output of simulated
          classical circuit to the nth qubit
      (set! U_omega (matrix* U_omega uncomputer))
      (set! U_omega (matrix* U_omega
         (G-nqubit-constructor (expt 2 total-qubits) (list (- total-qubits 2) (-
             total-qubits 1)) CNOT-gate)
         (G-nqubit-constructor (expt 2 total-qubits) (list (- total-qubits 2) (-
             total-qubits 1)) controlled-Z-gate)
         (G-nqubit-constructor (expt 2 total-qubits) (list (- total-qubits 2) (-
             total-qubits 1)) CNOT-gate)
         U_omega))
      (void (write-file "./circuit-files/qcircuit.qasm" (string-append
         "\tqubit\tq" (string-join (map (lambda (num) (number->string num)) (range
             total-qubits)) ",0\n\tqubit\tq") ",0\n" ; Set up necessary qubits
         computer-strings-for-latex ; Add the compute string
         "\tcnot\tq" (number->string (- total-qubits 3)) ",q" (number->string (-
             total-qubits 2)) "\n" ; CNOT for the copy on to (n-1)th qubit
         uncomputer-strings-for-latex ; Add the uncompute string
         "\tcnot\tq" (number->string (- total-qubits 2)) ",q" (number->string (-
             total-qubits 1)) "\n" ; CNOT for the copy on to nth qubit
         "\tc-z\tq" (number->string (- total-qubits 2)) ",q" (number->string (-
             total-qubits 1)) "\n" ; controlled Z on (n-1)th and nth qubits
         "\tcnot\tq" (number->string (- total-qubits 2)) ",q" (number->string (-
             total-qubits 1)) "\n" ; CNOT for the uncopy on to nth qubit
```

```
        computer-strings-for-latex ; Now we do the whole first half again, starting with
            the compute string
        "\tcnot\tq" (number->string (- total-qubits 3)) ",q" (number->string (-
            total-qubits 2)) "\n" ; CNOT for the uncopy on to (n-1)th qubit
        uncomputer-strings-for-latex ; Add the uncompute string
      ))))))

(define special-make-Hadamard (lambda (N up-to-qubit)
   (letrec ([apply-H (lambda (M qubit)
      (cond
         [(= qubit up-to-qubit) (matrix* M (G-nqubit-constructor N (list qubit)
             Hadamard-gate))]
         [else (matrix* M (apply-H (G-nqubit-constructor N (list qubit) Hadamard-gate) (+
             qubit 1)))])])
      (apply-H (identity-matrix N) 0))))

(define Grover-from-classical-circuit (lambda (input-U_omega input-qubits) ; An
    implementation of Grover's algorithm, input-U_omega is a matrix representation the
    oracle operator
   (let ([special-Hadamard null]
      [special-H-matrix (special-make-Hadamard (matrix-num-cols input-U_omega)
          input-qubits)]
      [steps 0]
      [qubits (exact-round (/ (log (matrix-num-cols input-U_omega)) (log 2)))]) ;
          Requires log(N) qubits where N is the width of the matrix representing U_omega
      (set! special-Hadamard (lambda (psi)
         (let ([new-psi (matrix* psi special-H-matrix)])
           (set-register new-psi)
           new-psi)))

      (cond
         [(= input-qubits 0) (set! steps 0)]
         [(= input-qubits 1) (set! steps 1)]
         [else (set! steps (exact-round (* (/ pi 4) (sqrt (expt 2 input-qubits)))))]) ; #
             of steps ~pi*sqrt(N)/4

      (set! U_omega input-U_omega)
      (display "The number of required qubits is ") (displayln qubits)
      (display "Number of operations required is ") (displayln (+ 1 steps))

      (initialize-register (build-list qubits (lambda (x) 0))) ; Initialize all qubits to
          |0>

      (special-Hadamard register) ; Apply a Hadamard gate to the input qubits

      (for ([i steps])
         (special-Hadamard (phase-flip-0-state (special-Hadamard (Oracle register))))) ;
             Apply the Grover Diffusion operator
      )))
   \begin{lstlisting}
```

Appendix D

qassoc.rkt

D.1 Source code

```
#lang racket
(require math)
(require racket/vector)

(require 2htdp/batch-io)
(define-namespace-anchor a)
(define ns (namespace-anchor->namespace a))

(require "quetzal.rkt") ; note that quetzal.rkt must be present in current directory

;-----------------Start qassoc.rkt--------------------;

(define (S-con p)
    (matrix [
       [1 0 0 0]
       [0 1 0 0]
       [0 0 (sqrt (/ (sub1 p) p)) (/ 1 (sqrt p))]
       [0 0 (/ 1 (sqrt p)) (sqrt (/ (sub1 p) p))]
    ]))

(define (learn patterns)
    (letrec ([bits (length (car patterns))]
        [qubits (+ (length (car patterns)) 3)] ; n qubits for storage, 1 intermediate
            qubit, 2 control qubits
        [CONOT (matrix* (G-nqubit-constructor 4 '(0) Pauli-X-gate) ; CONOT is the same as
            CNOT surrounded by nots on the control qubit
            CNOT-gate
            (G-nqubit-constructor 4 '(0) Pauli-X-gate))]
        [big-control-X (matrix-stack (list (submatrix (identity-matrix (expt 2 (+ 1
            bits))) (- (expt 2 (+ 1 bits)) 2) (expt 2 (+ 1 bits)))
                                    (matrix-row (identity-matrix (expt 2 (+ 1 bits)))
                                        (- (expt 2 (+ 1 bits)) 1))
                                    (matrix-row (identity-matrix (expt 2 (+ 1 bits)))
                                        (- (expt 2 (+ 1 bits)) 2))))])
```

```
        (initialize-register (build-list qubits (lambda (x) 0))) ; Initialize all qubits
            to |0>

        (for ([pattern patterns] [p-index (length patterns)])

            (for ([bit pattern] [b-index bits])
                (if (= bit 1)
                    (apply-gate register (list (sub1 qubits) b-index) CONOT) ; flip the
                        corresponding qubit if the bit in the pattern is a 1
                    (void)))

            (apply-gate register (list (- qubits 1) (- qubits 2)) CONOT) ; flip the c1
                control qubit
            (apply-gate register (list (- qubits 2) (- qubits 1)) (S-con (- (length
                patterns) p-index))) ; apply the "save" gate to control qubits

            (for ([bit pattern] [b-index bits])
                (if (= bit 0)
                    (apply-gate register (list b-index) Pauli-X-gate) ; apply not gates on
                        all qubits whose corresponding bit is 0
                    (void)))

            (apply-gate register (build-list (- qubits 2) values) big-control-X) ; apply
                not gate with controls on all storage qubits on intermediate bit
            (apply-gate register (list (- qubits 3) (- qubits 2)) CNOT-gate) ; CNOT with
                control on intermediate qubit, target on c1
            (apply-gate register (build-list (- qubits 2) values) big-control-X) ;
                reverse the controlled not

            (for ([bit pattern] [b-index bits]) ; reverse all the not gates to the
                storage qubits
                (if (= bit 0)
                    (apply-gate register (list b-index) Pauli-X-gate)
                    (apply-gate register (list (sub1 qubits) b-index) CONOT))))

        (apply-gate register (list (- qubits 1)) Pauli-X-gate) ; the algorithm will leave
            all states with the c2 bit as |1>, so not all these

        (remove-unentangled-qubit register) ; We can now ignore the last three qubits as
            they are not entangled
        (remove-unentangled-qubit register)
        (remove-unentangled-qubit register)))

(define (remove-unentangled-qubit reg) ; removes the last qubit from the system (assumes
    it is unentangled)
    (let [(new-reg (submatrix reg 1 (build-list (/ (matrix-num-cols reg) 2) (lambda (x)
        (* x 2)))))]
        (set-register new-reg)
        new-reg))


(define (constr-big-control-Z N)
    (diagonal-matrix (build-list N (lambda (x) (if (= x (- N 1)) -1 1)))))
```

```
(define (get-not-qubits qubits mtau) ; gets all the qubits which we know should be
    flipped
           (cond [(null? mtau) '()]
               [(eq? (car mtau) 0) (cons (- qubits (length mtau)) (get-not-qubits qubits
                   (cdr mtau)))]
               [else (get-not-qubits qubits (cdr mtau))]))

(define (constr-search-phaser tau)
    (letrec ([qubits (length tau)]
        [N (expt 2 qubits)]
        [find-unknown (lambda (mtau)  ; get an unknown to apply a controlled Z gate to
            (if (eq? (car mtau) '?) (- qubits (length mtau))
                (find-unknown (cdr mtau))))]
        [known-bits (filter (lambda (x) (not (eq? (list-ref tau x) '?))) (build-list
            qubits values))]
        [big-control-Z (constr-big-control-Z (expt 2 (+ (length known-bits) 1)))]
        [to-not-gate (lambda (q) ; applies an X gate
            (G-nqubit-constructor N (list q) Pauli-X-gate))]
        [to-Z-gate (lambda (q)  ; applies the Z gate to the two possible states of the
            unknown qubit
            (matrix* (G-nqubit-constructor N (append known-bits (list q)) big-control-Z)
                ; first a Z on the unknown
                (to-not-gate q) ; flip the unknown
                (G-nqubit-constructor N (append known-bits (list q)) big-control-Z) ;
                    another Z
                (to-not-gate q) ; flip it back to preserve the state
                ))])

        (eval (cons matrix* (append
            (map to-not-gate (get-not-qubits qubits tau)) ; first apply X to all the
                qubits that should be flipped (so that if we have the right pattern, all
                qubits are 1)
            (list (to-Z-gate (find-unknown tau))) ; controlled Z gate on on of the
                unknowns
            (map to-not-gate (reverse (get-not-qubits qubits tau))))) ns) ; apply the
                same Xs in reverse order to preserve state
        ))

(define (constr-patterns-phaser patterns)
    (letrec ([qubits (length (car patterns))]
        [N (expt 2 qubits)]
        [big-control-Z (constr-big-control-Z N)]
        [to-not-gate (lambda (q) ; applies an X gate
            (G-nqubit-constructor N (list q) Pauli-X-gate))]
        [constr-helper (lambda (pats)
            (cond [(null? pats) (list (identity-matrix N))]
                [else (append (map to-not-gate (get-not-qubits qubits (car pats)))
                    (list big-control-Z)
                    (map to-not-gate (get-not-qubits qubits (car pats)))
                    (constr-helper (cdr pats)))]))])

        (eval (cons matrix* (constr-helper patterns)) ns)
        ))
```

```
(define make-Hadamard (lambda (N)
    (let ([constant (exact->inexact (/ 1 (expt 2 (/ (/ (log N) (log 2)) 2))))])
        (build-matrix N N (lambda (i j)
            (* constant (expt -1 (matrix-dot (bits->row-matrix (bits i N))
                (bits->row-matrix (bits j N))))))))))

(define (calc-T p r0 r1 N) ; p: total patterns, r0: # of states that don't correspond to
    patterns, r1: # that do
    (letrec ([a (/ (* 2 (- p (* 2 r1))) N)]
        [b (/ (* 4 (+ p r0)) N)]
        [k (+ (- (* 4 a) (* a b)) (/ r1 (+ r0 r1)))]
        [l (- (/ (* 2 a (+ N (- p r0 (* 2 r1)))) (- N r0 r1)) (* a b) (/ (- p r1) (- N r0
            r1)))])

        (exact-round (/ (- (/ pi 2) (atan (/ (* k (sqrt (/ (+ r0 r1) (- N r0 r1)))) l)))
            (acos (- 1 (/ (* 2 (+ r1 r0)) N)))))))

(define (Grover-part tau patterns)
    (letrec ([qubits (length tau)]
        [N (expt 2 qubits)]
        [all-qubits (build-list qubits values)]
        [full-Hadamard (make-Hadamard N)]
        [Hadamard (lambda (reg)
            (apply-gate register all-qubits full-Hadamard))]
        [search-phase-flip-gates (constr-search-phaser tau)]
        [patterns-phase-flip-gates (constr-patterns-phaser patterns)]
        [T (calc-T (length patterns) (- (expt 2 (count (lambda (x) (eq? x '?)) tau)) 1) 1
            N)] ; Assume that there is only one match in the learned patterns
        [big-control-Z (constr-big-control-Z N)])

        (apply-gate register all-qubits search-phase-flip-gates) ; Apply the phase flip
            for the one we're searching for

        (Hadamard (apply-gate (Hadamard register) (reverse all-qubits) big-control-Z)) ;
            One Grover diffusion

        (apply-gate register all-qubits patterns-phase-flip-gates) ; Apply gate that
            flips the phase of all patterns

        (Hadamard (apply-gate (Hadamard register) (reverse all-qubits) big-control-Z)) ;
            A second Grover diffusion

        (for ([i T]) ; We apply the search phase flipper and the Gover diffusion T times
            (apply-gate register all-qubits search-phase-flip-gates) ; Apply the phase
                flip for the one we're searching for
            (Hadamard (apply-gate (Hadamard register) (reverse all-qubits)
                big-control-Z)) ; Apply a Grover diffusion
        )))
```

D.2 Usage of Code

The learning algorithm is called *learn*, and the recall algorithm is called *Grover-part*. You can use them like so:

```
> (define P '((0 0 0 0) (0 0 1 1) (0 1 1 0) (1 0 0 1) (1 1 0 0) (1 1 1 1)))
> (learn P)
> (Grover-part '(0 1 1 ?) P)
> (measure-register)
The most likely result is |0110> with a probability of 0.8437500000000002
```