Directed Research Project Report

George Lifchits

December 23, 2015

Abstract

This report summarizes the outcomes of my directed research project, supervised by Dr. Shohini Ghose and Dr. Ilias Kotsireas. We study Grover's quantum search algorithm, specifically in the context of finding D-optimal sequences.

For this research project, I developed software to simulate quantum algorithms. We discuss some specific challenges encountered during the development of this software, and how they were addressed in the implementation.

Then, we turn to how we might use Grover's quantum search algorithm to find D-optimal sequences. We look closely at the design of the 'oracle', which is a black-box computation that acts as a bottleneck for the efficiency of Grover's search in practical use cases. An analysis of our oracle's space and runtime complexity is provided.

This report is written to be accessible to readers with little to no background in quantum computing and some background in algorithms and computational complexity. We provide substantial background on quantum computing concepts discussed in the report and others which are useful for general knowledge of quantum computing. We also provide the background necessary to discuss the D-optimal problem.

Contents

List of Figures

1 Introduction

This term, I was interested in looking more closely at how quantum computing can be used for practical problems. In winter 2015, I had the privilege of learning about quantum computing in a course taught by Dr. Shohini Ghose at WLU, where we learned about quantum computing and learned of several interesting algorithms. Grover's quantum search algorithm in particular is fascinating, because it is touted [9] as a universally applicable method for searching that promises a significant speed-up with only a small amount of "cleverness" necessary (compared to the typical methods of optimizing search problems).

As a consequence of this research, I accomplished at least two significant learning objectives. First, I was successful in designing and implementing a program which enabled me to model quantum algorithms (on my classical laptop). I implemented several simple quantum algorithms and a simulation for Grover's search for the D-optimal problem which I discuss much more this report.

Secondly, I was able to solidify and extend my knowledge of Grover's search algorithm, understanding more about how it works, why it works, and its limitations. Indirectly, as a consequence of researching the field and in conversations with my advisors Dr. Ghose and Dr. Ilias Kotsireas, I learned about many other concepts in the areas of quantum computing and other methods of search used to solve computational problems.

2 Background

In the following sections, I provide background on some of the technical concepts that are discussed in this research.

2.1 D-optimal sequences

In this research, the problem of interest is finding D-optimal sequences. This introduction simplifies information from [2].

Consider sequences of length N where each element is either -1 or 1. D-optimal sequences are a pair of sequences which can be used to construct a $2N \times 2N$ matrix (with elements

-1 and 1) that has the maximum possible determinant. Using a D-optimal sequence pair A and B, this matrix is constructed as:

$$H = \begin{pmatrix} A & B \\ -B^T & A^T \end{pmatrix}$$

and specifically it is said that the matrix H attains Ehlich's bound:

$$\det(H) \leq 2^N(2N-1)(N-1)^{N-1}$$

see [2] for more details on this.

D-optimal sequences are formalized in the mathematical notion of 'supplementary difference sets', which are well-studied in the field of combinatorial designs. In this research, we are interested in how quantum search can benefit us without needing to resort to highly technical optimizations. As such, we avoid the nuanced technical details of D-optimal matrices and rely on simple properties which are introduced in the following sections.

2.1.1 Periodic Autocorrelation

Let $A = [a_1, a_2, \ldots, a_n]$ be a {-1, 1} sequence of length N. The periodic autocorrelation function (PAF) associated with this sequence is defined as:

$$P_A(i) = \sum_{k=1}^{N} a_k a_{k+i}, \quad i = 0, \ldots, N-1 \tag{1}$$

where $k+i$ is taken modulo N. The parameter i is called the 'lag'. [3]

2.1.2 PAF Constraint

Define a similar sequence B. The two sequences A and B are D-optimal if:

$$P_A(i) + P_B(i) = 2, \quad i = 1, \ldots, N-1$$

Due to symmetry properties of the periodic autocorrelation function (discussed in [3]), we can safely discard some computations. We define the PAF constraint as follows:

$$P_A(i) + P_B(i) = 2, \quad i = 1, \ldots, \frac{N-1}{2} \tag{2}$$

This constraint is sufficient to identify D-optimal {-1, 1} sequence pairs of length N.

2.2 Quantum computing

For the purpose of making this report available to any reader; to solidify concepts discussed in this report; and to demonstrate my enhanced knowledge of quantum computing as a result of this research term, I provide a review of some topics in quantum computing.

Quantum computing is a system of computation that is based on the theory of quantum mechanics. There is a clear distinction between "classical" and "quantum" computing: classical computing is the familiar scheme of working with bits (0s and 1s), and the algorithms which are fundamentally designed to manipulate these bits.

Quantum computing is a superset over classical computing. A proof of this, specifically showing that BPP[1] $\subseteq$ BQP[2] is given in [6]. But intuitively, we will see that quantum bits are a generalization of classical bits, so it would be unreasonable that the generalization fails in the specific case of classical computing. When (not if!) quantum computing becomes available to us, classical algorithms that are well-studied are certainly still available to us, but quantum mechanics offers extremely powerful phenomena that unlock computational methods impossible in the world of classical bits.

2.2.1 Bits and qubits

To describe states of bits (and qubits) simply, we use linear algebra. Consider a vector space $\mathbb{Z}^2$ – classical bits (0 or 1) can be represented as orthogonal unit vectors forming a basis:

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

The notation of $|x\rangle$ is called a 'ket' in Dirac notation, which denotes that x is a column vector. This notation is very helpful and commonly used in quantum computing.

In quantum computing, our bits are replaced with 'qubits'. Like a bit, a qubit is a 0 or 1, but its value is *probabilistic*. In general, a qubit is described:

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

[1]complexity class: bounded-error probabilistic polynomial time.

[2]complexity class: bounded-error quantum polynomial time.

The values α and β are complex values, called *amplitudes*. For this qubit $|\psi\rangle$, the probability of its value being 0 is $|\alpha|^2$ and the probability of it being 1 is $|\beta|^2$. Consequently, we have $|\alpha|^2 + |\beta|^2 = 1$. The same qubit corresponds to a vector in $\mathbb{C}^2$:

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Let's consider a new quantum state, with two bits. We use the tensor product (Kronecker product) of linear algebra to describe the resultant state:

$$|\phi\rangle = \begin{bmatrix} \alpha_1 \\ \beta_1 \end{bmatrix} \otimes \begin{bmatrix} \alpha_2 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} \alpha_1\alpha_2 \\ \alpha_1\beta_2 \\ \beta_1\alpha_2 \\ \beta_1\beta_2 \end{bmatrix}$$

For this two-qubit example $|\phi\rangle$, the state is a column vector in $\mathbb{C}^4$. Much literature in quantum computing makes generous use of the term *Hilbert space*: this is simply alternative terminology for an inner product space [1]. For us, the a Hilbert space is a vector space in which all the necessary operations of quantum computing are available.

We can eschew the linear algebra approach and use kets instead. The same two-qubit state as above can be written as:

$$\begin{aligned} |\phi\rangle &= (\alpha_1|0\rangle + \beta_1|1\rangle) \otimes (\alpha_2|0\rangle + \beta_2|1\rangle) \\ &= \alpha_1\alpha_2|00\rangle + \alpha_1\beta_2|01\rangle + \beta_1\alpha_2|10\rangle + \beta_1\beta_2|11\rangle \end{aligned}$$

This example shows how we can exploit Dirac notation to make the computation of tensor product by hand as procedurally simple as multiplying polynomials.

This example also demonstrates an important fact about quantum computing. If these were classical bits, the state $|\phi\rangle$ could conceivably be described to demonstrate that two bits gives us four potential encodings of information. In practice of course, we are plainly aware that these two bits will only ever be one of those four combinations. Quantum is in stark opposition to this obvious concept: in quantum computing, the system $|\phi\rangle$ is said to be in a state of *superposition*, and that this system of two qubits is actually in all four possible states all at once.

Superposition is hard to describe and hard to believe, but it has been empirically shown to appear in quantum phenomena everywhere. It is an important contributor to the power of quantum computing. It means that with two qubits, we can work with all four of their configurations of the at the same time.

2.2.2 Uncertainty principle

There is a fundamental caveat to the farfetched idea of superposition – and even though places limits on the usefulness of superposition, the caveat itself is perhaps even stranger.

When a quantum system is 'observed', its state is said to collapse into a specific classical state with some probability. When we say 'observed', we mean just that. It is empirically understood that the observation of a quantum state is what causes it to collapse, and we can even define (and manipulate) the probabilities of how a quantum state will collapse. But the 'observation' is a tricky concept, and not well understood [13]. There are competing interpretations of quantum mechanics that attempt to provide an explanation for these ideas, but they are impossible to test, and can be philosophically heavy. For the purposes of this report, we will be better off taking quantum physics for granted, and moving on.

Observation will destroy the amplitudes of a qubit, which collapses it into a classical state. For example, a qubit $|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ has equal probability of collapsing into either the 0 or 1 state. However, if we have a qubit $|\psi\rangle = 0|0\rangle + 1|1\rangle$, we are certain that any measurement of the qubit will always collapse it into the 1 state. We can construct quantum systems of several qubits, and as long as we don't do a measurement, we can play with the amplitudes in interesting ways.

Classical algorithms fundamentally manipulate bits, with the intent of constructing a sequence of bits that is useful to us. Similarly, the goal of quantum algorithms is to manipulate the amplitudes of qubits until we are reasonably sure that, once we measure our qubits, they will collapse into a state that yields a solution to our problem. This influences much of the way that quantum computing differs from classical computing. Classical algorithms tend to make heavy use of decision structures (the famous if-statement). In quantum computing, an if-statement would require measuring the qubit in order to make a comparison, and a measurement destroys quantum information. This serves to demonstrate the real challenge that the designers of quantum algorithms face: how exactly can we exploit bizarre quantum phenomena to our advantage?

2.2.3 Quantum gates

Like classical computing requires logic gates for manipulating the values of bits, this analogy extends to quantum computing. Quantum gates are described as *unitary operators.* A linear operator A is unitary if it maintains the following property:

$$AA^\dagger = A^\dagger A = I$$

where $A^\dagger = (A^*)^T$ and A^* denotes complex conjugation. More detail is provided in [1], for example. However, the important properties of unitary operators is that they preserve the norm of a vector and they are invertible. Norm preservation is important to ensure that the sum of squared amplitudes in a quantum state never exceeds 1, and invertibility is a postulate of quantum mechanics. In figure 1 below, we show some important unitary gates for a single qubit system.

$$\mathrm{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \mathrm{Y} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad \mathrm{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \qquad \begin{aligned} \mathrm{H} &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\ &= \frac{1}{\sqrt{2}}(\mathrm{X} + \mathrm{Z}) \end{aligned}$$

Figure 1: Some important gates in quantum computing. The X, Y, and Z gates are known as the Pauli operators, and are sometimes noted σ_x, σ_y, and σ_z (respectively). The X gate in particular is a simple bit flip. The last gate, H, is called the Hadamard gate.

A fundamental unitary operator is the controlled-NOT (CNOT). It uses two qubits: a *control* and a *target.* The effect of the CNOT is to flip the target qubit only if the control qubit is on. It is a fascinating operation, because intuitively it would require a measurement of the control qubit, but it does not.

$$\mathbf{C}_{10} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad \mathbf{C}_{01} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 2: Both CNOT unitary operators for a 2-qubit state. The notation CNOT_{xy} indicates the target qubit index and control qubit index, respectively.

2.2.4 Entanglement

The last concept I wanted to discuss which is central to quantum mechanics is *entanglement*. In quantum computing, entanglement refers to a special configuration of qubits that is called a 'non-tensor product', because an entangled state cannot be decomposed into distinct states via the tensor product as described in 2.2.1.

With a entangled pair of qubits, a measurement of one of the qubits immediately yields the state of the other qubit without needing another measurement. An entangled pair state looks like this:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

By now we understand this means that the system $|\psi\rangle$ will be measured to be either 00 or 11 with equal probability. But how did this state come to be? It isn't clear how such a state came out of a tensor product relationship. It turns out that this state is very easy to construct (in theory):

$$\begin{aligned}
|\psi\rangle &= |0_1 0_2\rangle && \text{We start with two zeroed qubits.} \\
&= \mathrm{H}_1|0_1 0_2\rangle && \text{Apply the Hadamard gate to the first qubit.} \\
&= \frac{1}{\sqrt{2}}(|0_1\rangle + |1_1\rangle)|0_2\rangle && \text{(Result of applying Hadamard)} \\
&= \mathrm{CNOT}_{12}\frac{1}{\sqrt{2}}(|0_1 0_2\rangle + |1_1 0_2\rangle)) && \text{Apply a controlled-NOT (target=1, control=2)} \\
&= \frac{1}{\sqrt{2}}(|0_1 0_2\rangle + |1_1 1_2\rangle) && \text{(Result of CNOT) We have the entangled pair.}
\end{aligned}$$

2.2.5 Grover's quantum search algorithm

Grover introduced his quantum mechanical algorithm for search in [5]. Basically, Grover's search is an algorithm which employs a quantum computer to search a "database" in sub-linear time. Specifically, for a database containing N items, we can find matching items with $O(\sqrt{N})$ iterations. It is an ingenious algorithm which exploits quantum superposition and qubit amplitudes, which can be negative in quantum computing. Grover's main technique is called "inversion by the mean".

We can describe Grover's search with the tools discussed so far. Grover's works on a quantum system of n qubits – such a system has $N = 2^n$ possible states, and therefore has N amplitudes to work with.

1. Initialize the system. We start out with n qubits all in the $|0\rangle$ state:

$$|\gamma\rangle = |000\ldots0\rangle$$

   Then we apply the Hadamard (H) gate to each qubit, which has the effect of putting the N possible states in equal superposition:

$$\begin{aligned}|\gamma\rangle &= \frac{1}{\sqrt{N}}|00\ldots00\rangle + \frac{1}{\sqrt{N}}|00\ldots01\rangle + \cdots + \frac{1}{\sqrt{N}}|11\ldots10\rangle + \frac{1}{\sqrt{N}}|11\ldots11\rangle \\ &= \frac{1}{\sqrt{N}}\sum_{i=0}^{N-1}|i\rangle\end{aligned}$$

   This step requires $O(\log_2 N) = O(n)$ applications of the Hadamard gate.

2. Perform the following 'Grover iteration' $(\pi/4)\sqrt{N}$ times:[3]

   (a) Apply the oracle as a function f with a unitary operator defined as:

$$\mathbf{U}_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$$

   (b) Apply the diffusion operator D, which is defined as:

$$D = I + \frac{2}{N}\mathbf{1}$$

[3]This many iterations is needed for a search problem containing one solution. Problems with several solutions are considered in [10] and [12], but it is unclear what a precise formula is for the exact number of iterations which maximizes probability of success. I found that $(\pi/4)\sqrt{N/M}$ is a crude approximation.

Here, I is an identity matrix order $N \times N$, and $\mathbf{1}$ is an $N \times N$ matrix where each entry is 1.

3. Measure the system $|\gamma\rangle$, thereby collapsing the superposition. All states that satisfy the oracle function f have a high probability of being measured. Specifically, the probability of a satisfying state being measured is $O(\frac{1}{2})$ and $\Omega(1)$.

Incredibly, the number of iterations necessary to reach a high probability of success is just $O(\sqrt{N})$. Classical algorithms can never do better than $O(N)$, so Grover's search provides a quadratic speedup here. An interesting observation which was initially counter-intuitive to me is that increasing the number of iterations does *not* improve the probability of finding a solution. As the number of iterations grows, the probability of measuring a correct solution oscillates periodically. This is shown with an example simulation of Grover's search in figure 3.

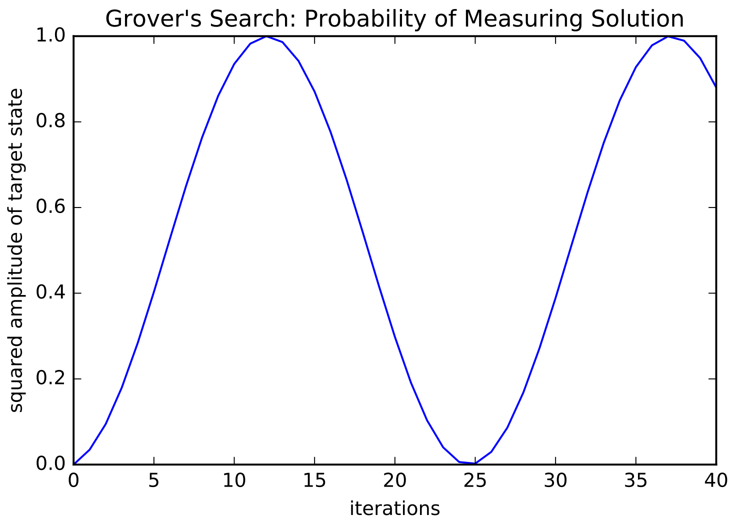

Figure 3: Grover's search probability changing over multiple iterations. This plot is generated from a simulation of Grover's search for a system of 8 qubits, looking for the value 220. The correct number of iterations is $\lfloor \pi/4 \times \sqrt{256} \rfloor = 12$, and it is shown here that the probability of measuring the solution is maximized at 12 iterations ($P = 99.995\%$). We see how the success probability oscillates like a sine wave. Another maximum exists at the 37th iteration.

However, there is an important caveat to the Grover speedup. In Grover's algorithm we use a function $f(x)$ which outputs 1 if the input state x is a solution to our search problem, and 0 otherwise. The unitary operator $\mathbf{U}_f$ corresponding to this function is referred to as an 'oracle'. In [5] Grover assumes that the oracle can be evaluated in *unit time* for any state.

If we want to do a search for whether an element exists in the space, an $O(1)$ oracle makes sense. But if we want to use Grover's algorithm for any search problems, then the oracle, whose function was previously abstracted away from our concern, suddenly becomes very important. In fact, the number of times we perform the Grover iteration $O(\sqrt{N})$ is a runtime complexity that is measured not in seconds, but in the number of times we query the oracle. Therefore, it can be said that for a state x, and oracle running time $r(x)$, the runtime of Grover's search will be $r(x)O(\sqrt{N})$. In section 3.2.2 I describe how we developed an oracle that is used in Grover's search to identify D-optimal sequences. Of course, we pay special attention to the $r(x)$ of this oracle.

3 Methods

In this section I will explain specifically what I did in this research term. Broadly, it was two main things: writing a program that would simulate quantum algorithms, and then designing an oracle to show how Grover's search works for the D-optimal search problem.

3.1 Designing a program to simulate quantum algorithms

The first challenge in this research term was to find a suitable way to simulate quantum algorithms on my computer. The most straight-forward implementation I could think of was one that revolved around the idea of the quantum state as being a vector with complex values. The resulting program then provides methods that allow the user to easily perform operations on that state vector.

Since the vector and matrix operations are of central concern in such an implementation, I chose to use Numpy [11], which is a very useful and comprehensive library for scientific computing in Python. In particular, Numpy offers a fully-featured implementation of linear algebra and its various operations.

The state vector is the heart of my program. For a quantum state with n qubits, the state vector will have 2^n entries. A convenient note that helps in understanding the implementation is that for each integer $x \in \{0, \ldots, 2^n - 1\}$, x is both a possible state of the qubits, and the index to the amplitude of that state. In the sections that follow, I discuss some of the finer points of implementing a quantum computing simulation program.

3.1.1 Applying unitary gates

This is the most fundamental and basic operation. My implementation simply takes the provided unitary operator provided as the argument `unitary` and multiplies it with `self.state`, which is the state vector (in this case, `*` performs matrix multiplication). For simplicity's sake, instead of checking that the provided matrix is a norm-preserving unitary matrix, we simply renormalize the state vector in order to ensure it remains of unit length.

```python
def apply_unitary(self, unitary):
    # multiply the unitary operator to this state
    new_state = unitary * self.state
    # renormalize the state
    norm_factor = sum(p**2 for p in to_row(new_state))
    norm_new_state = 1/norm_factor * new_state # scalar multiplication
    return Q(norm_new_state)
```

Figure 4: Applying a unitary gate with Python

3.1.2 Applying a function

For performing more general computations, we have the idea of applying a function to the system. In this context, a function takes a state and returns a binary output (0 or 1). The function is applied to the quantum state by taking the contents of a designated 'input' register, using its state as the single argument to the function, and then applying the output of the function to an 'output' register.

Mermin [9] discusses the process of applying functions in a quantum algorithm as "the general computational process". Specifically he defines function application as a unitary operator $\mathbf{U}_f$ defined as follows:

$$\mathbf{U}_f(|x\rangle_n |y\rangle_m) = |x\rangle_n |y \oplus f(x)\rangle_m$$

$|x\rangle_n$ is an input register of n qubits and $|y\rangle$ is an output register of m qubits. The argument x that is provided to the function f is simply the binary encoding of the input register.

How this works can be illustrated with a simple example. Say we have a 3-qubit state $|\psi\rangle = |0_3 0_2 0_1\rangle$ and a function $f(x) = 1$. For simplicity, my implementation assumes that the first bit is the output register, and all bits following comprise the input register. Therefore the result of applying the function f to the state $|\psi\rangle$ is as follows:

step	input	output	$\|\psi\rangle$
before applying f	00	0	$\|000\rangle$
after applying f	00	1	$\|001\rangle$

Figure 5: A simple example of the general computational process

This example suffices to demonstrate the realized behaviour of the function applicator. We start with the state $\alpha|000\rangle$ and get the state $\alpha|001\rangle$ (where $|\alpha|^2 = 1$). What happened here is just a swap – we take the amplitude that was assigned to the initial state (000) and transfer that amplitude to the resulting state (001). To reinforce the example for 3 qubits, we can rewrite $|\psi\rangle$ with generalized amplitudes:

$$|\psi\rangle = \alpha_1|000\rangle + \alpha_2|001\rangle + \alpha_3|010\rangle + \alpha_4|011\rangle + \alpha_5|100\rangle + \alpha_6|101\rangle + \alpha_7|110\rangle + \alpha_8|111\rangle$$

and now applying the function f as above, we are left with the following:

$$\begin{aligned}|\psi\rangle &= \alpha_1|001\rangle + \alpha_2|000\rangle + \alpha_3|011\rangle + \alpha_4|010\rangle + \alpha_5|101\rangle + \alpha_6|100\rangle + \alpha_7|111\rangle + \alpha_8|110\rangle \\ &= \alpha_2|000\rangle + \alpha_1|001\rangle + \alpha_4|010\rangle + \alpha_3|011\rangle + \alpha_6|100\rangle + \alpha_5|101\rangle + \alpha_8|110\rangle + \alpha_7|111\rangle\end{aligned}$$

From this we can see that the real effect of $\mathbf{U}_f$ is to swap the amplitudes of the states where the input registers x are the same and where $f(x) \neq y$.

In figure A.2, I show Python code that performs function application using this logic. For each index in the state vector, it uses the function to compute the swap index, and then swaps the amplitudes at these indices. In my implementation, the output register is always a single qubit at the first index of the system (figure 3.1.2 demonstrates this).

3.1.3 Controlled-NOT operation

Defining behaviour for the CNOT operation was a initial point of confusion for me. Mermin [9] introduces the CNOT as a unitary operation on a 2-qubit state, by providing two unitary CNOT operators (see figure 2). However, if we were interested in performing a CNOT between control and target qubits at arbitrary indices of a larger quantum system, we run into a challenge.

Similar to the function applicator logic, instead of defining the unitary gate outright, I was able to use the amplitude swapping method as a proxy for what the unitary operator would do. Example of code which performs CNOT swapping is shown in figure A.1.

3.1.4 Testing the implementation

Before I even revisited Grover's search, my goal was to make my implementation capable of performing simpler quantum algorithms. My first test was to simulate the algorithm of Superdense coding. [7] (see figure A.3 for example code of this). After that, I was quickly able to implement the Deutsch-Josza algorithm [8] (see figure A.4 for a code example of this). In both cases, I found that my implementation yields a very expressive simulation in very few lines of code.

3.1.5 Efficiency of classical simulation of a quantum computer

Although my implementation achieves the expressive efficiency of quantum algorithms, and even appears to take just as many steps, the source of inefficiency arises from maintaining the state vector, which has 2^n entries for an n qubit system. In a quantum computer, each individual qubit (or entangled pair) would maintain their own amplitudes.

Indeed, in a quantum computer, the state vector is impossible to compute. The amplitudes of qubits can be predicted but they cannot be observed, or else we will destroy them. So, the 'state vector' is an artefact of classical simulation that fully describes how a quantum system will behave.

3.2 Grover's search for the D-optimal problem

3.2.1 Encoding the problem

Firstly, we describe a scheme which allows us to encode the D-optimal problem in a quantum computer. Recall that in the D-optimal problem, we are concerned with finding pairs of sequences of a fixed length N. These sequences are binary with values either -1 or 1.

For sequences of length N, we use a quantum system with $2N$ qubits. This gives us a state space of size 2^{2N}, which encodes all possible combinations of sequence pairs length N.

Each state in the system corresponds to a binary string of length $2N$. To decode the string into sequences, we simply split the string at the midpoint, yielding two binary strings of length N. To convert the binary string to the desired binary sequences, apply the mapping:

$$0 \mapsto -1$$
$$1 \mapsto 1$$

3.2.2 Design of an oracle for the D-optimal problem

As discussed in section 2.2.5, Grover's search queries an oracle that behaves like a function $f(x)$ for a state x: the oracle yields 1 if x is a solution and 0 otherwise. Here we present a suitable oracle for the D-optimal problem.

Consider sequences length N and define $M = \frac{N-1}{2}$. Recall the PAF constraint in 2.1.2:

$$P_A(i) + P_B(i) = 2, \quad i = 1, \ldots, M$$

We design the oracle to produce a result similar to this computation. Since we want the oracle to be compatible with parallelism in Grover's search, we need a closed-form function that does not perform intermediate comparisons (which require measuring the qubits).

To check that the sum of the PAFs is two for each lag i, we subtract 2 each time and expect 0. Even before subtracting 2, we could expect these terms to be negative, so we take the absolute value in order to ensure our computation is 0 only in the case of D-optimal sequences. All told, we are left with the following computation:

$$S = \sum_{i=1}^{M} |P_A(i) + P_B(i) - 2|$$

S is guaranteed to be 0 if sequences A and B are D-optimal. Otherwise, S is a positive number. Suppose that the number S is stored in a system of qubits (space complexity analysis follows in section 3.2.4). We apply the X (NOT) gate to these qubits and then perform AND on all qubits, computing the resulting AND in a final output qubit. This output qubit will be 1 if the sequences A and B are D-optimal, and 0 if they are not.

3.2.3 Simulation results

I implemented the oracle as described and used it in my simulated implementation of Grover's search. Figure 6 demonstrates how the simulation works.

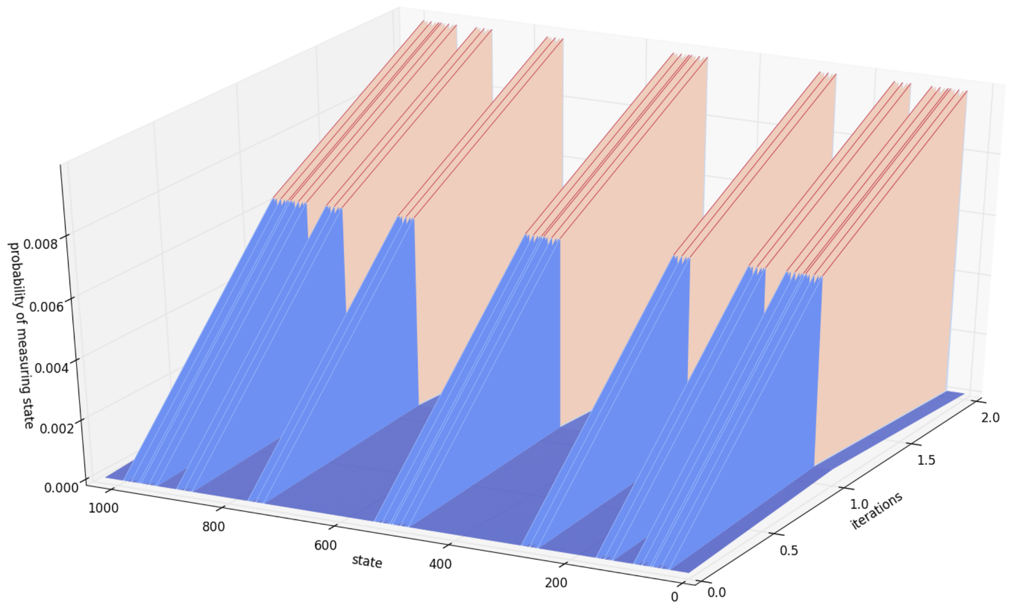

Figure 6: Surface plot of Grover's search simulation for D-optimal sequences. The plot shows the squared amplitude of each possible state along the axis labelled 'state'. The 'iterations' axis shows how the squared amplitudes change after each Grover iteration. This simulation uses the oracle as described in section 3.2.2 to find D-optimal sequences of length 5. This system contains $2*5 = 10$ qubits, yielding $2^{10} = 1024$ possible sequence pairs. 100 of the 1024 states have squared amplitude equal to 1.0%, each of them corresponding to a D-optimal sequence pair. Therefore, any measurement of the system will yield a D-optimal sequence pair. It takes 2 Grover iterations for the system to reach this state.

3.2.4 Space complexity of the D-optimal oracle

Recall that our oracle computes a value S which equals 0 for a state which encodes a D-optimal sequence pair, and 1 otherwise.

From the definition of the periodic autocorrelation function (see equation (1)), we observe that a sequence of length N yields a PAF which computes N pairwise terms. Since elements in our sequences are (-1, 1), any individual term will also be either -1 or 1. Thus we conclude that for any sequence of length N, the PAF computation will be at most N and at least $-N$. With this in mind, we can deduce worst-case space requirements to store the computed value S:

$$\begin{aligned}
S &= \sum_{s=1}^{M} |P_A(s) + P_B(s) - 2| \\
&\leq \sum_{s=1}^{M} (N + N - 2) \\
&\leq M(2N - 2) \\
&\leq \frac{N-1}{2}(2N - 2) \\
&\leq N(N - 1)
\end{aligned}$$

Since we know S is bounded by N^2, we deduce that we need at most $\lceil \log_2(N^2) \rceil = \lceil 2\log_2 N \rceil$ bits to store S. For the final result of 0 or 1, we perform AND between each bit and store the result with one additional bit. Ultimately, from this analysis we conclude that the space complexity of the oracle is $O(\log_2 N)$.

3.2.5 Computational complexity of the D-optimal oracle

To determine the number of computational steps required for the PAF oracle, we use a similar analysis. Again, from the definition of the PAF (see equation 1), there are N pairwise terms for a sequence of length N. This corresponds to N multiplicative operations and $N - 1$ additions. Since we are aware different operations might have different computational concerns, we will decompose the complexity of the oracle into the number of individual operations required. Let π represent a multiplication (product), let α be an addition (or subtraction) operation, and let λ represent the absolute value operation.

$$\begin{aligned}
S &= \sum_{s=1}^{M} |P_A(s) + P_B(s) - 2| \\
C_S &= \sum_{s=1}^{M} 2(N\pi + (N-1)\alpha) + \alpha + \lambda \\
&= M\big(2N\pi + (2N-1)\alpha + \lambda\big) \\
&= \frac{N-1}{2}\big(2N\pi + (2N-1)\alpha + \lambda\big) \\
&= (N^2 - N)\pi + (N^2 - 2N - 1)\alpha + \frac{N-1}{2}\lambda
\end{aligned}$$

We derive C_S, which is a function of (π, α, λ) which expresses the exact runtime necessary to compute the value S.

In addition to C_S, we consider the NOT and AND operations which we use on the bits that store the computed value of S. Following our space complexity analysis (see section 3.2.4) in which we determine that S can be stored with at most N^2 bits, we would need to perform N^2 NOT operations, and $N^2 - 1$ pairwise AND operations. Let a be the runtime of an AND computation and let n be the runtime of a NOT computation. We are left with a function C_o which we use to describe the total computational complexity of the oracle:

$$\begin{aligned}
C_o(\pi, \alpha, \lambda, a, n) &= C_S + (N^2)n + (N^2 - 1)a \\
&= (N^2 - N)\pi + (N^2 - 2N - 1)\alpha + \frac{N-1}{2}\lambda + (2\log_2 N)n + (2\log_2 N - 1)a
\end{aligned}$$

We conclude that the oracle runs in $O(N^2)$ time.

3.2.6 Implications of oracle complexity analysis

In sections 3.2.4 and 3.2.5 I provide analysis to demonstrate that the D-optimal oracle we provide in section 3.2.2 runs in time $O(N^2)$.

In [12], Viamontes et. al. describe a scheme they developed which uses a novel data structure called a QuIDD to represent systems of qubits. They claim that oracles which require polynomial resources could offer a speedup on *classical* computers that is proportional to implementing Grover's on a quantum computer. However, this is highly dependent on how the oracle is implemented in their QuIDD framework – it is unclear how resource requirements for an oracle map to the QuIDD scheme. In addition to this, they do not have a freely available implementation of the QuIDD scheme, so I was unable to test their claim.

As for my quantum simulation program, a k-qubit system requires maintaining a state vector containing 2^k entries. Furthermore, my classical simulation is unable to take advantage of Grover parallelism, so one single Grover iteration requires (an obscene) runtime that is proportional to $(k^2)(2^k)$.

In a quantum computer, assuming nothing in our oracle is lost in translation, the runtime to find D-optimal sequence pairs of length N would be on the order of $O(N^2 N^{1/2}) = O(N^{3/2})$. Considering the fact that we performed no optimization tricks that involve a deep understanding of the D-optimal problem, this is a remarkable runtime.

4 Conclusion

This term, I conducted directed research to understand how we could use Grover's search algorithm to find a solution to the D-optimal problem. We constructed an oracle that can be used in Grover's search, analyzed its resource requirements, and concluded that finding D-optimal sequences of length N on a quantum computer could be as fast as $O(N^{3/2})$. I also demonstrate the methodology I used to successfully implement a program that can be used to simulate quantum algorithms on a regular computer. In the process, I learned a great deal about quantum computing.

4.1 Further questions

This research raised some interesting questions, which could merit further study.

One question is to verify the claims of the QuIDD scheme presented by Viamontes et. al. As we explained in section 3.2.6, they claim that their invention has the potential to offer speedups on classical computers proportional to an ideal quantum circuit. It is potentially valuable to experimentally verify their claim by implementing their QuIDD scheme and attempt to implement Grover's search for a problem like the D-optimal problem we studied in this report.

In section 3.2.2, where we describe the oracle we use in Grover's search, it is important to be cognizant of the fact that this oracle is implemented classically. Viamontes et. al. point out that a Grover's search oracle could certainly be designed with quantum circuits. In such cases, the runtime of the oracle is potentially much faster than the classical oracle we described in this report. On the other hand, Viamontes et. al. rightly point out that it may be prohibitively challenging to come up with a quantum oracle that beats the classical one. Indeed, coming up with such an oracle could present a challenge even greater than coming up with clever optimizations in classical methods. It is not just possible, but likely that someone studying a specific problem (the D-optimal problem is just one example) could have an easier time fine-tuning a classical search algorithm than they would be able to devise a clever quantum algorithm to get speedup.

Appendices

```python
def apply_cnot(self, control, target):
    control_mask = 2 ** control # 0 everywhere and 1 at the control bit
    target_mask = 2 ** target   # 0 everywhere and 1 at the target bit
    new_state = self.state
    swapped = set([])
    for qubit in range(2 ** self.num_qubits):
        control_bit = qubit & control_mask
        if control_bit == 0: continue # to next qubit; don't flip target qubit
        # compute the target bit index
        target_idx = qubit ^ target_mask
        if (target_idx, qubit) in swapped: continue # avoid double-swapping
        # perform the row swap
        new_state = row_swap(new_state, qubit, target_idx)
        # add both (i, j) and (j, i) to the swapped set
        swapped.add((target_idx, qubit))
        swapped.add((qubit, target_idx))
    return Q(new_state)
```

Figure A.1: Implementation of arbitrarily-indexed controlled-NOT in Python

```python
def apply_func(self, func):
    state_vector = self.state
    swapped = set([])
    for current_state in range(2 ** self.num_qubits):
        x = current_state >> 1  # take bits 1-n
        y = current_state & 1   # take just bit 0
        # new state is the original x bits with y ⊕ f(x) bit at end
        new_state = (x << 1) + (y ^ func(x))
        if (current_state, new_state) not in swapped:
            # swap amplitudes at indices `current_state' and computed `new_state'
            state_vector = row_swap(state_vector, current_state, new_state)
            swapped.add((current_state, new_state)) # avoid double swapping
            swapped.add((new_state, current_state))
    # resulting state is the one with swapped rows
    return Q(state_vector)
```

Figure A.2: Implementation of the general quantum computational process in Python

```
from quantum import *
# Alice and Bob each have one qubit in an entangled pair
state = get_entangled_pair()
ALICE = 1  # index of Alice's qubit
BOB = 0    # index of Bob's qubit
# Alice sends message `ab` as defined.
a, b = 1, 0
if a == 1: # Alice applies Z to her qubit.
    state = state.apply_gate(Z, ALICE)
if b == 1: # Alice applies X to her qubit.
    state = state.apply_gate(X, ALICE)
# Bob applies CNOT gate
# using Alice's qubit as control, his qubit as target.
state = state.apply_cnot(ALICE, BOB)
# Alice applies Hadamard gate on her qubit
state = state.apply_gate(H, ALICE)
# Bob measures the qubits to obtain the message ab.
measured = state.measure()
print("Measured state =", bin(measured))
>>> Measured state = 0b10
```

Figure A.3: Implementation of Superdense coding algorithm in Python. Observe that the message `ab` is `10`, which is exactly what is measured from the quantum state at the end of the program.

```
from quantum import *  # `Q', `kron', `zero', `H', `X'
# `func' is the function we wish to test
# it is constant, but Deutsch-Josza will confirm this.
def func(x): return 0
# start with |00⟩
state = Q(kron(zero, zero))
# apply X gate to all qubits.
for i in range(state.num_qubits):
    state = state.apply_gate(X, i)
# apply H gate to all qubits.
for i in range(state.num_qubits):
    state = state.apply_gate(H, i)
# apply the function
state = state.apply_func(func)
# apply H on the last qubit
state = state.apply_gate(H, 1)
print(state)

[[ 0.00000000+0.j]
 [ 0.00000000+0.j]
 [ 0.70710678+0.j]
 [-0.70710678+0.j]]
```

Figure A.4: Implementation of the Deutsch-Josza algorithm in Python. The 4×1 vector at the end represents our quantum state $|\psi\rangle = \frac{1}{\sqrt{2}}|1_b 0_a\rangle + \frac{-1}{\sqrt{2}}|1_b 1_a\rangle$ If we measure qubit b to be a 1, then we know the function is constant, and if it is measured as a 0 then we know the function is balanced. Two states of $|\psi\rangle$ have a probability of being measured, and both states have qubit b as one. The algorithm works as expected.

References

[1] Dawar, A. 2004, *Quantum Computing - Lectures*. [Online]. Available: `http://www.cl.cam.ac.uk/Teaching/current/QuantComp/` [Accessed Oct. 6, 2015]

[2] Đoković, D.Ž. & Kotsireas, I.S. "New Results on D-Optimal Matrices." (2012). *Journal of Combinatorial Designs.* (20), pp. 278–289.

[3] Kotsireas, I.S. "Algorithms and Metaheuristics for Combinatorial Matrices." (2013). *Handbook of Combinatorial Optimization*. pp. 283–305.

[4] Ghose, S. 2015, *CP310A: Introduction to Quantum Computing - Lectures*.

[5] Grover, Lov K., "A fast quantum mechanical algorithm for database search". (1996). *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*.

[6] J. Watrous. Quantum computational complexity. *Encyclopedia of Complexity and System Science*. (2009). pp. 13. [Online]. Available: `https://cs.uwaterloo.ca/~watrous/Papers/QuantumComputationalComplexity.pdf`.

[7] Lifchits, G, Superdense Coding Python example. (2015). Github repository. [Online]. Available: `https://github.com/glifchits/d-optimal-matrices/blob/master/quantum-demonstrations/Superdense-Coding.ipynb`

[8] Lifchits, G, Deutsch-Josza algorithm example. (2015). Github repository. [Online]. Available: `https://github.com/glifchits/d-optimal-matrices/blob/master/quantum-demonstrations/Deutch-Algorithm.ipynb`

[9] Mermin, N. (2007). *Quantum computer science: An introduction.* Cambridge, UK: Cambridge University Press.

[10] M. Boyer, G. Brassard, P. Hoyer & A. Tapp, *Tight bounds on quantum searching*, Proceedings, PhysComp 1996. [Online]. Available: `http://arxiv.org/pdf/quant-ph/9605034.pdf`

[11] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13, 22–30 (2011), DOI:10.1109/MCSE.2011.37

[12] Viamontes, G.F., Markov, I.L., & Hayes J.P. “Is Quantum Search Practical?” (2005). *Computing in Science and Engineering.* (2005). Vol. 7 (3). pp. 22–30.

[13] Why does observation collapse the wave function? (n.d.). Retrieved December 20, 2015, from http://physics.stackexchange.com/questions/35328/why-does-observation-collapse-the-wave-function