Grover's Algorithm for Unstructured Quantum Search

Matthew Bush

July 3, 2013

1 Introduction

1.1 The Promise of Quantum Computing

Quantum computing offers the promise of a radically different paradigm for computation: one that is based on the laws of quantum mechanics, where seemingly impossible things are commonplace. Because of this, a quantum computer is able to accomplish tasks, such as quantum teleportation, that intuitively seem impossible. A quantum computer is also able to solve several classes of difficult classical problems much faster than on a classical computer. This often turns a problem from hopelessly impractical to something that can be solved in a reasonable amount of time. In many ways, the difference between a quantum computer and a classical computer is as profound as the difference between a classical computer and what was called a "computer" in the early 1940s: a human being carrying out rote calculations with pencil and paper.

At present, there are two main classes of quantum algorithms that give significant speed advantages over their classical counterparts: those related to the quantum Fourier transform (QFT), and those related to quantum search. Many of the QFT-related algorithms provide an exponential speedup over the best known classical algorithms. While the quantum search algorithms provide a smaller but still significant quadratic speedup, they are also applicable to more problems. Here, we will focus on the quantum search algorithms, and on Grover's algorithm in particular.

1.2 The Difficulties of Quantum Computing

The principle of superposition is what allows quantum computing to have such enormous power. Superposition is the ability of a set of quantum bits, called qubits, to be in many or even all possible states simultaneously. Superposition is one of the "weird" consequences of quantum mechanics. It is not simply the case that we don't know which state the particle is in; it actually behaves as if it's in multiple states until it is observed. However, superposition is an extremely fragile state, which is the main reason for the tremendous practical difficulties that researchers trying to build quantum computers have encountered.

Once an observation of a quantum state has been made, the particle collapses into a single state, and the information associated with the superposition is irretrievably lost. It is because of this collapse that we refer to many of the attributes of quantum particles as "hidden information:"

we cannot determine the exact values of these attributes without examining an infinite number of identical particles. This causes problems because "observations" are very easy to make, and very difficult to avoid. Almost any interaction with an outside system is an observation that causes the collapse of superposition. Until very recently, this has meant that the best quantum computers actually built by experimentalists had lifetimes measured in nanoseconds. After the observation, the computer would have to be "built" again.

Additionally, there are many situations in which quantum computing does not provide any significant advantage over classical computing, and indeed may provide a disadvantage due to its enormous complexity. Quantum computers gain their power from the ability to massively parallelize a task. For tasks that must necessarily be done in series, this offers little to no advantage. There are also many tasks that could potentially benefit from being run on a quantum computer, but for which no efficient quantum algorithm is known. Because the laws of quantum mechanics are so different from our everyday experiences, the intuition necessary to develop a completely novel quantum algorithm is so rare that it has only been done a handful of times. Developing new algorithms will be one of the major hurdles in the future of quantum computing.

In spite of these difficulties, quantum computers seem to have a promising future in scientific computing and in running some of the "cloud" systems used today. They also have the potential to revolutionize the fields of cryptography and cryptanalysis, as quantum computers have the power to break almost any cryptosystem used today for secure communication, requiring only polynomial time, rather than the exponential time required by a classical computer. However, (at the risk of becoming as dated as the 1949 prediction in *Popular Mechanics* that "computers in the future may ... weigh only 1.5 tons."[2]) it does not seem like quantum computers will ever entirely replace the desktop computers and mobile devices we use today.

1.3 The D-Wave quantum computer

Recently, a Canadian company called D-Wave Systems has developed, sold, and deployed a 512-qubit adiabatic quantum computer. Although the details of this computer are beyond the scope of this paper, it is worth noting that there is some controversy over whether the adiabatic model qualifies as a "real" quantum computer, as there are some quantum systems it is unable to simulate. Regardless, the fact that the computer exists and has real-world usefulness is a tremendous achievement.

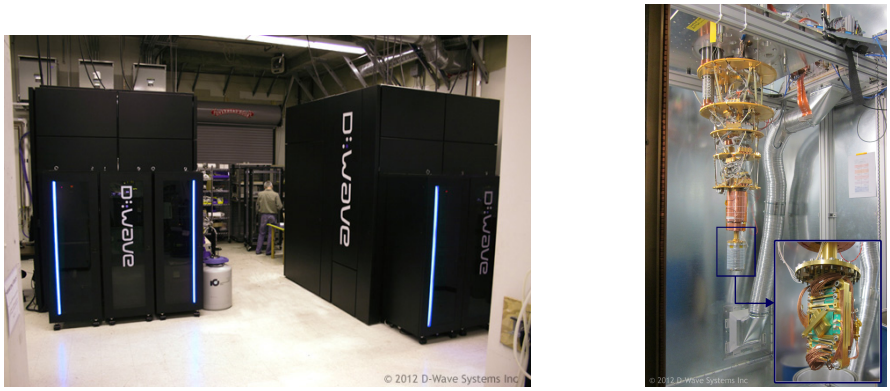


Figure 1: Two images of D-Wave Systems Inc. quantum computers. The left-hand image shows two complete computers, encased in their 10'x10'x10' refrigeration chambers. The right-hand image shows the actual processor inside the refrigeration chamber. [1].

See the Appendix for more information.

2 Quantum Computing

A quantum computer is a system built of quantum bits, called qubits, which like the classical bits of a traditional computer, may be in a state of either zero or one. Unlike classical bits, they may also be in some superposition of the two states, with amplitudes specified by complex numbers whose absolute squares sum to one. We represent a quantum computer with n qubits as a 2^n-dimensional complex vector space. The basis of this vector space, called the *computational basis*, is all the possible n-bit strings of zeros and ones.

To simulate a quantum computer using a classical computer, we represent the n-qubit quantum computer as a 2^n-dimensional complex vector space. The basis of this vector space, called the *computational basis*, is the set of all possible n-bit strings of zeros and ones. We represent an arbitrary state of the computer as a linear combination of the computational basis vectors, with coefficients equal to the probability amplitude of each particular state.

We use the physicist's "bra-ket" notation for vectors. $\langle\psi|$ is a row vector, and $|\phi\rangle$ is a column vector. For a single qubit x, $\langle x| = (x_0, x_1)$ is a vector of length 2. x_0 is the probability amplitude that $x = 0$, and x_1 is the probability amplitude that $x = 1$. Thus each qubit must have norm 1, because the probability of an event is the absolute square of the amplitude, and the probabilities must sum to 1. For two qubits, $|xy\rangle$ is the tensor product $|x\rangle \otimes |y\rangle = \langle z_{00}, z_{01}, z_{10}, z_{11}\rangle$, with z_{ij} the probability amplitude that $x = i$ and $y = k$. So an n-qubit state $|\psi\rangle$ is a vector of length 2^n, since it is the tensor product of n length-2 vectors. "Bra-ket" notation also makes it easy to define inner and outer products. $\langle\psi||\phi\rangle$ (usually written $\langle\psi|\phi\rangle$), the product of a row vector with a column vector, is the usual complex inner product, and $|\phi\rangle\langle\psi|$, the product of a column vector with a row vector, is the outer product. $|\phi\rangle|\theta\rangle$ is the tensor product $|\phi\rangle \otimes |\theta\rangle$.

After initializing the quantum computer to a particular state, the computations take the form of unitary operators. Each operator may be broken down into the composition of one or more basic gates, and the action of several gates in succession is called a circuit. Each gate may act on one or more qubits.

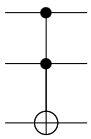
2.1 Quantum Gates

It is a theorem of quantum information that all quantum circuits must be reversible.[6] One consequence of this is that each gate must have exactly

as many outputs as it does inputs. This precludes the use of most of the gates used in classical computing, such as AND, OR, XOR, NAND, etc. It also precludes the use of the classical FANOUT operation whereby one bit is copied to another. However, in the special case that all qubits are in pure classical states, each of these gates can be simulated using some of the quantum gates discussed below.

One important quantum gate is the controlled-not gate, or CNOT. This gate operates on two qubits. The first qubit, called the control qubit, is passed through unchanged. The second qubit, called the target qubit, has the states $|0\rangle$ and $|1\rangle$ interchanged if the control qubit is $|1\rangle$, and is left unchanged if the control qubit is $|0\rangle$. In the notation of quantum circuits, we write this gate as shown on the right, with the control qubit on the upper wire and the target qubit on the lower wire.

Closely related to the CNOT gate is the Toffoli gate, also called controlled-controlled-not, or CCNOT. This behaves similarly to the CNOT gate, but with two control qubits. The target qubit is flipped only if both control qubits are $|1\rangle$. The notation for a Toffoli gate as part of a quantum circuit is on the right. As before, the target qubit is on the bottom wire.



Unlike classical systems, where the only non-trivial operation on a single gate is the NOT gate which interchanges the states 0 and 1, there are many non-trivial single-qubit gates, each corresponding to a unitary 2×2 complex matrix. While the CNOT and Toffoli gates can also be realized in classical systems, the NOT gate is the only single-qubit quantum gate with a classical analogue. Of the infinitely many single-qubit gates, several stand out as particularly important.

First is the Hadamard gate, which has matrix representation $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$. This gate is often used to create superposition, as it sends pure classical states to an equiprobable superposition of possible states. In the visualization of a qubit as a point on the Bloch sphere, the Hadamard gate first rotates the qubit by $\pi/2$ about the $\hat{y}$-axis, then rotates by π about the $\hat{x}$-axis. These two rotations are equivalent to a rotation of π about the axis $(\hat{x} + \hat{z})/\sqrt{2}$.

In addition to the Hadamard gate, the three Pauli gates X, Y and Z are very useful. Each gate derives its name from the fact that it can be visualized as a rotation by π about the respective axis on the Bloch sphere.

As matrices, they are represented as:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

The Pauli-X gate has the additional function of implementing the NOT gate, and the Pauli-Z gate effects a phase shift of -1.

Any of these single-qubit gates may also be applied to n qubits, by tensoring the matrix with itself n times.

2.2 No-cloning Theorem

Theorem 1 (No-Cloning)**.** *Given an arbitrary unknown quantum state $|\psi\rangle$, it is impossible to create the state $|\psi\rangle|\psi\rangle$.*

Proof. Suppose, by way of contradiction, that it is possible to create such a state. Let $|\psi\rangle$ be the state we wish to copy. We know nothing about $|\psi\rangle$ beyond the number of qubits it contains. Let $|x\rangle$ be some known state with the same number of qubits as $|\psi\rangle$. By our assumption, there exists some unitary operator U such that $U(|\psi\rangle|x\rangle) = |\psi\rangle|\psi\rangle$ for all states $|\psi\rangle$. Let $|\psi\rangle$ and $|\phi\rangle$ be two particular quantum states able to be cloned by U. Then we have that:

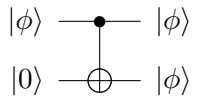
$$U(|\psi\rangle \otimes |x\rangle) = |\psi\rangle \otimes |\psi\rangle \tag{1}$$
$$U(|\phi\rangle \otimes |x\rangle) = |\phi\rangle \otimes |\phi\rangle \tag{2}$$

We then take the inner product of these two equations. Since U preserves inner products, we have that:

$$\langle x|\langle\psi|U^\dagger U|\phi\rangle|x\rangle = \langle\psi|\langle\psi|\phi\rangle|\phi\rangle \tag{3}$$
$$\langle x|\langle\psi|\phi\rangle|x\rangle = \langle\psi|\phi\rangle^2 \tag{4}$$
$$\langle x|x\rangle\langle\psi|\phi\rangle = \langle\psi|\phi\rangle^2 \tag{5}$$
$$\langle\psi|\phi\rangle = \langle\psi|\phi\rangle^2 \tag{6}$$

Which implies that either $\langle\phi|\psi\rangle = 0$ or $\langle\phi|\psi\rangle = 1$. This implies that the two states $|\phi\rangle$ and $|\psi\rangle$ are either equal or orthogonal, which means that U can only clone two particular states, not an arbitrary state as required in the statement of the theorem. □

Like many aspects of quantum mechanics, the no-cloning theorem seems counterintuitive. Intuitively, the following circuit would seem to produce two copies of $|x\rangle$:



However, this is only true if $|\phi\rangle$ is a pure classical state ($|0\rangle$ or $|1\rangle$). In the general case, it produces two qubits that are maximally entangled, rather than two independent copies of $|\phi\rangle$. If $|\phi\rangle$ is some mixed state, such as $\alpha|0\rangle + \beta|1\rangle$, this circuit will transform an initial state of $\alpha|0\rangle + \beta|1\rangle$ to a final state of $\alpha|00\rangle + \beta|11\rangle$. In this state, the two qubits always take the same value when measured. True cloning would produce two qubits that behaved independently when measured, represented by the state $|\phi\rangle|\phi\rangle$, which is $\alpha|0\rangle + \beta|1\rangle \otimes \alpha|0\rangle + \beta|1\rangle = \alpha^2|00\rangle + \alpha\beta|01\rangle + \alpha\beta|10\rangle + \beta^2|11\rangle$.

Indeed, the no-cloning theorem is a special case of the general requirement that all quantum computations must be reversible. If it were possible to clone a qubit, and then evolve the two qubits separately, running the system in reverse might result in the two clones being non-identical immediately after the cloning operator. There would then be no well-defined way to determine what qubit was input to the cloning operator, and thus no way to reverse it.

3 Unstructured Search: Classical

One area where quantum computing offers a significant speedup over classical computing is the search of an unstructured database. An unstructured database search can be modeled using a database with N entries, and a function $f : \{0, 1, \ldots, N-1\} \to \{0, 1\}$ where $f(x) = 1$ iff the database entry with index x has the sought-after property.

As an example, let us consider the database with 12 entries, corresponding to the twelve months of the year. We assign each month an index from 0 to 11. One possible search that could be done on this database would be to ask the question: "Find a month that started on a Sunday in 2012." Looking at Table 1, we can see that January, April, and July are the only months that started on a Sunday in 2012. So we may define our function

$$f(x) = \begin{cases} 1 & \text{if } x \in \{0, 3, 6\}, \\ 0 & \text{otherwise.} \end{cases}$$

When I describe an "unstructured" database, the order of the entries in the database has no useful relationship to the sought-after qualities. Table 1 satisfies that requirement to a degree. While the months are listed in order, and the null values are all at the end, the order of the months has little bearing on which day of the week they start, and the null values are unnecessary for a classical search. Perhaps a better example would be a phone book, sorted alphabetically by name, but with the task of searching by phone number. If we restrict ourselves to classical (non-quantum) methods, the task of searching the phone book for a particular number is $O(N)$. If we were to first sort the database, we could improve the performance of the search to $O(\log(N))$, but the sorting itself would require $O(N \log(N))$ time[5].

In the classical unstructured search problem, the best one can do is to check each entry individually. If there is only one entry for which $f(x) = 1$, one must examine $\lceil N/2 \rceil$ entries to guarantee a probability of success greater than $1/2$. In the case where there are M entries satisfying the search criterion, with $M > 1$, the situation becomes significantly more complicated. The probability of finding r desirable records after searching k entries is given by a hypergeometric distribution. The number of records to search to reach a probability of finding at least one desirable record greater than one half is bounded from below by $N/(2M)$, as seen in Figure 2. This may not be the best possible bound, but since the quantum search has run time $O\left(\sqrt{\frac{N}{M}}\right)$,

Month	Index	Binary Index	First day in 2012	$f(x)$
January	0	0000	Sunday	1
February	1	0001	Wednesday	0
March	2	0010	Thursday	0
April	3	0011	Sunday	1
May	4	0100	Tuesday	0
June	5	0101	Friday	0
July	6	0110	Sunday	1
August	7	0111	Wednesday	0
September	8	1000	Saturday	0
October	9	1001	Monday	0
November	10	1010	Thursday	0
December	11	1011	Saturday	0
null	12	1100	null	0
null	13	1101	null	0
null	14	1110	null	0
null	15	1111	null	0

Table 1: The twelve months of the year are each assigned an index, written in both decimal and binary. Also listed is the day on which the month started in 2012. If the month started on a Sunday, $f(x) = 1$. Otherwise, $f(x) = 0$. I have added four null rows to bring the size of the database up to a power of two. The null rows will return 0 for any query asked of them. This padding is done to make the size of the database a power of 2, which is a requirement for conducting a quantum search.

Figure 2: This figure will have two graphs showing $k - N/2M$ bounded from below by zero for varying N and M. k is the number of entries that must be checked to ensure a probability of success greater than $1/2$.

it is clear that the quantum algorithm represents at least a quadratic speedup over the classical algorithm.

4 Grover's Algorithm

Grover's algorithm exploits quantum superposition to examine every entry in the database at the same time. By applying $O\left(\sqrt{\frac{N}{M}}\right)$ iterations of the algorithm, one is able to prepare a quantum state which, when measured, has a very high probability of collapsing to a state corresponding to the index of one of the sought-after elements. This gives a quadratic speedup over the classical search, which can be quite significant for large problems.

Grover's algorithm consists of the successive application of four quantum circuits, each of which I will describe in detail. The first circuit is the oracle, which may be viewed as flipping the sign of the qubit corresponding to each "correct" answer in the computational basis. The second is a Hadamard gate, applied to n qubits. The third circuit is a "phase shift" gate, which flips the sign of all states but the all-zero state. The fourth circuit is another Hadamard gate, identical to the first.

4.1 The Quantum Oracle

The heart of Grover's algorithm is the oracle circuit. For now, it is sufficient to consider the oracle to be a "black box", the construction of which will be discussed in Section 5. The oracle takes as input $n+1$ qubits: an n-qubit register, $|x\rangle$, that at the start of the computation is initialized to the equiprobable state $|\psi\rangle = \sum_x |x\rangle/\sqrt{N}$ and one additional qubit, called the oracle qubit, $|q\rangle$. The oracle can recognize whether the input $|x\rangle$ is a solution to the search query, and if it is, executes a bit flip, interchanging the states $|0\rangle$ and $|1\rangle$, on the oracle qubit.

Since the state of the computer is always a linear combination of computational basis states, we may define the action of the oracle by defining its action on each computational basis state. For $|x\rangle \in \{|00\ldots0\rangle, \ldots, |11\ldots1\rangle\}$, if $|x\rangle$ is a solution to the query, the oracle will execute a bit flip (exchanging $|0\rangle$ and $|1\rangle$) on the work qubit. Otherwise, the oracle will not change anything. By choosing to initialize the oracle qubit to the state $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$, we are able to simplify the action of the oracle. Interchanging $|0\rangle$ and $|1\rangle$ effects a global phase shift of -1 on the state $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$. Because $|x\rangle|q\rangle$ is a tensor, we may choose to write this phase shift with the $|x\rangle$. This has the advantage of leaving the work qubit unchanged regardless of the value of $f(x)$. So we may write the action of the oracle on an element $|x\rangle$ of the computational

basis as:

$$|x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \xrightarrow{O} (-1)^{f(x)} |x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \tag{7}$$

Because the oracle qubit is unchanged by the oracle, and not acted on at all by the other three circuits, I will proceed to ignore it in the remainder of the analysis.

Because the oracle effects a phase shift of -1 on each element of the computational basis for which $f(x) = 1$, the effect of the oracle gate is to reflect the state vector (in N-dimensional space) across the vector consisting of an equally weighted superposition of all computational basis states where $f(x) \neq 1$.

4.2 The Rest of the Grover Operator: *HPH*

The remaining three steps in the Grover operator are significantly simpler than the oracle. In particular, they do not depend at all on the specific query being searched, only on the number of qubits, and even that dependence is minimal. They consist of a conditional phase-shift gate conjugated by a Hadamard gate.

The Hadamard gate on n qubits maps an element of the computational basis (i.e. a pure classical state) to a superposition of all possible states, with equal probability. Because the Hadamard gate is an involution, it maps those equiprobable states which are the image of a pure state back to the corresponding pure state. It maps all other real equiprobable vectors to another, or possibly to the same, real equiprobable vector. The situation for complex equiprobable vectors is somewhat more complicated, but none of the gates used in Grover's algorithm ever produce non-real amplitudes. Note that because we only require the probabilities to be equal, the sign of the amplitudes will vary considerably, depending on which computational basis state we use as input.

The conditional phase shift gate applies a phase shift of -1 to all states besides the all-zero state, and acts as the identity on the all-zero state. Its matrix as a unitary operator on the computational basis is the diagonal matrix with 1 in the upper-left position, and -1s in each other position along the diagonal. This matrix may also be written as $2|0\rangle\langle 0| - I$. Geometrically, this effects a reflection across the $|0\rangle$-axis. The reason for the seemingly arbitrary choice of the all-zero state is that the Hadamard gate sends the all-zero state to an equal superposition of states *all with positive amplitude*.

Because of this property of the Hadamard gate, the net effect of the Hadamard-Phase Shift-Hadamard operators is $2|\psi\rangle\langle\psi| - I$, where $|\psi\rangle =$

$1/\sqrt{N}\sum_x |x\rangle = |0\rangle H$, the equal superposition of all states. This effects a reflection across the vector ψ, as we will see in the next section.

4.3 Geometric Visualization of the Grover Operator

The action of the Grover operator may be visualized most effectively in a two-dimensional vector space. Let $N = 2^n$ be the total number of computational basis vectors, and let M be the number of computational basis vectors for which $f(x) = 1$. Define the normalized states

$$|\alpha\rangle = \frac{1}{\sqrt{N-M}} \sum_{\{x:f(x)=0\}} |x\rangle \tag{8}$$

$$|\beta\rangle = \frac{1}{\sqrt{M}} \sum_{\{x:f(x)=1\}} |x\rangle \tag{9}$$

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_x |x\rangle \tag{10}$$

Conceptually, $|\psi\rangle$ is the equal superposition of all computational basis states, and $|\alpha\rangle$ and $|\beta\rangle$ are the equal superposition of all computational basis states that are non-solutions and solutions to the search query, respectively.

It is easily verified that

$$|\psi\rangle = \sqrt{\frac{N-M}{N}}|\alpha\rangle + \sqrt{\frac{M}{N}}|\beta\rangle \tag{11}$$

As mentioned above, the unitary matrix representing HPH is $2|\psi\rangle\langle\psi| - I$. This transformation results in a reflection across the vector $|\psi\rangle$. Since the action of the oracle causes a reflection across $|\alpha\rangle$, and the composition of two reflections is a rotation, the net effect of the entire Grover operator is to rotate the state of the computer by an angle of θ towards $|\beta\rangle$. This rather beautiful and surprisingly simple description of the effect of the Grover operator is visualized in Figure 3.

As seen in Figure 3, θ is twice the angle between $|\alpha\rangle$ and $|\psi\rangle$. From Equation 11, $\theta/2$ is easily determined, which gives us

$$\theta = 2\arcsin\left(\sqrt{\frac{M}{N}}\right) \tag{12}$$

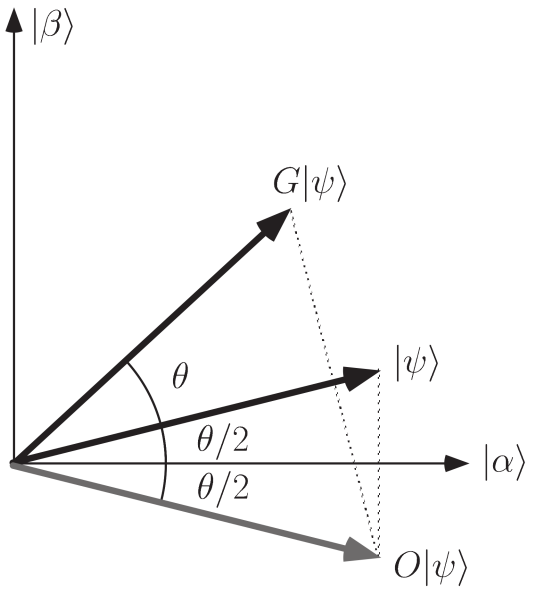


Figure 3: This shows the action of the first iteration of the Grover operator in the 2-dimensional vector space spanned by $|\alpha\rangle$ and $|\beta\rangle$. See accompanying discussion in subsection 4.3 for a full explanation. Figure from [6], p. 253.

4.4 Runtime and the Number of Iterations

Since each application of the Grover operator rotates the state vector by θ, in order to attain the maximum probability of success when we measure the final computer state, we want the state vector to be as close to $|\beta\rangle$ as possible. However, if we run the algorithm too many times, the state vector rotates *past* $|\beta\rangle$, and the probability of success goes down, so it is important to know when to stop. This requires knowledge of θ, which in turn requires knowledge of N, which of course we have, and also knowledge of M, the number of computational basis vectors for which $f(x) = 1$. If M is not known ahead of time, it can be very time-consuming to calculate. There are several ways to speed up this counting process using a quantum algorithm, but they are beyond the scope of this paper. For our purposes, we will assume that M is known ahead of time.

Since we wish to rotate the state vector from $|\psi\rangle$ to $|\beta\rangle$, an angle of $\arccos\left(\sqrt{M/N}\right)$, the number of iterations r should be be as close as possible to

$$r = \frac{\arccos\left(\sqrt{\frac{M}{N}}\right)}{\theta} \tag{13}$$

$$= \frac{\arccos\left(\sqrt{\frac{M}{N}}\right)}{2\arcsin\left(\sqrt{\frac{M}{N}}\right)} \tag{14}$$

$$= \frac{\frac{\pi}{2} - \arcsin\left(\sqrt{\frac{M}{N}}\right)}{2\arcsin\left(\sqrt{\frac{M}{N}}\right)} \tag{15}$$

$$= \frac{\pi}{4\arcsin\left(\sqrt{\frac{M}{N}}\right)} - \frac{1}{2} \tag{16}$$

Since the number of iterations generated by this formula generally will not be an integer, we apply the Grover operator $\lfloor r \rceil$ times, where $\lfloor \cdot \rceil$ denotes the closest-integer, or rounding, function.

Since in most applications of Grover's algorithm, $M \ll N$, we may apply the small-angle approximation $\sin(x) \approx x$, from which it follows that r is $O(\sqrt{N/M})$.

It is worth noting that the precision of the search also depends on M/N.

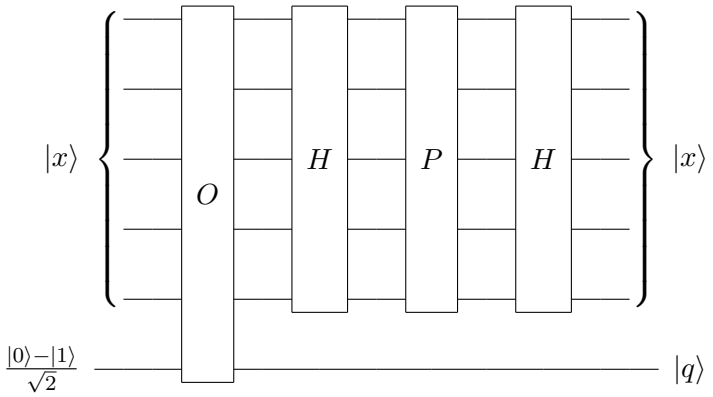


Figure 4: A circuit for the Grover operator. To perform a quantum search, this operator is applied $\lfloor r \rceil$ times. O is the Oracle circuit in Figure 5, H is the Hadamard gate on n qubits, and P is the gate that applies a phase shift of -1 to the all-zero state, and leaves the other states alone. $|q\rangle$ is the oracle qubit, which is initialized to the state $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$, and then is unchanged by the rest of the algorithm. (See Equation 7 and the accompanying discussion).

After running the Grover operator $\lfloor r \rceil$ times, the initial state $|\psi\rangle$ has been rotated by an angle of $\theta \lfloor r \rceil$ towards $|\beta\rangle$. This gives an amplitude in the $|\beta\rangle$-direction of $\sin(\theta/2 + \lfloor r \rceil \theta)$. Since $\sin(\theta/2 + r\theta) = 1$, the final amplitude is $\sin(\pi/2 - (r - \lfloor r \rceil)\theta)$. Because by definition $|r - \lfloor r \rceil| \leq 1/2$, smaller values of θ will result in a greater probability that the final state, when measured, actually collapses to a value for which $f(x) = 1$.

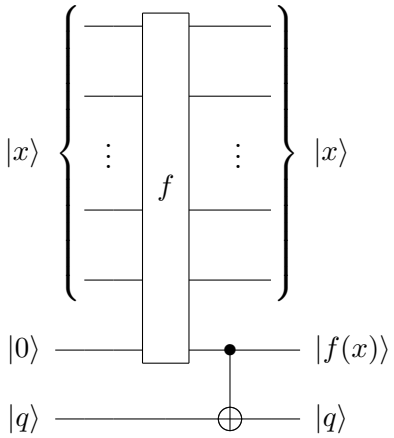


Figure 5: A circuit for building a quantum oracle, using a circuit that evaluates $f(|x\rangle)$ and a CNOT gate. The circuit for evaluating f can (and usually will) have additional work qubits, which are not shown because they are only used internally by f.

5 The Oracle in Detail

5.1 Constructing the Oracle

The name "oracle" naturally arouses some suspicion. If we have an oracle, why do we need to search at all? Why not just ask the oracle for the answer, as was done in Delphi in ancient Greece? The answer to these important questions lies in a misunderstanding of the nature of the oracle. The oracle is not required to *identify* a solution, only to *recognize* whether a particular entry in the database satisfies the search query. All that is necessary for this task is a quantum circuit that evaluates $f(x)$ and flips a work qubit iff $f(x) = 1$, as is shown in Figure 5.

Grover's algorithm can search an unstructured database, but it needs to be told what to look for. That information is contained in the function f, which is defined such that $f(x) = 1$ if $|x\rangle$ is the index of some element from the database that we are looking for, and $f(x) = 0$ otherwise. Because the algorithm runs on a quantum computer, the function f needs to be encoded in a quantum circuit. Because the Toffoli gate is universal for classical computation,[6] any classical circuit may be represented using only Toffoli gates, in a way that will run on either a classical or quantum computer. I will show that any boolean function may be represented as a boolean polynomial, and then show a circuit that evaluates every boolean polynomial.

5.2 Boolean Functions

An n-variable (or n-bit) boolean function is a function from the set of all n-bit binary words to the set $\{0, 1\}$. In counting the number of such functions, we must decide for each word whether it is mapped to 0 or to 1. Since there are 2^n words, we must make 2^n decisions, with two choices for each decision. Therefore, there are 2^{2^n} distinct n-bit boolean functions. Let $\mathcal{F}_n$ be the set of all such functions.

An n-variable boolean polynomial is a polynomial of the form

$$f(x_0, x_1, \ldots, x_{n-1}) = \sum_{\substack{i_0, i_1, \ldots, \\ i_{n-1} \in \{0,1\}}} a_{i_{n-1} i_{n-2} \ldots i_0} x_{n-1}^{i_{n-1}} x_{n-2}^{i_{n-2}} \cdots x_0^{i_0} \qquad (17)$$

with the variables x_j and the coefficients a_k coming from the field $\mathbb{F}_2 = \{0, 1\}$. Because of the fact that for $x \in \mathbb{F}_2$, $x^2 = x$, we need only consider exponents of 0 or 1. In counting such polynomials, we need only choose the value of the coefficient for each particular combination of exponents 0 and 1. Thus, once again, we must make 2^n choices, with two options for each choice, so there are 2^{2^n} polynomials ways to write a polynomial in the above form. Let $\mathcal{P}_n$ be the set of all such polynomials.

Since every polynomial is a function, it is clear that $\mathcal{P}_n \subseteq \mathcal{F}_n$. Note also that the coefficients $a_{i_{n-1}, i_{n-2}, \ldots, i_0}$ are themselves a function defined by the formula $i_{n-1} i_{n-2} \ldots i_0 \mapsto a_{i_{n-1} i_{n-2} \ldots i_0}$. We can write an explicit morphism $\tau : \mathcal{P}_n \to \mathcal{F}_n$. Furthermore, τ is a linear mapping, since it is the evaluation map of a polynomial, and also an involution, as I will prove below. First we will adopt the following ~~conventions~~ notational morass.

- Each coefficient $a_{i_{n-1} i_{n-2} \ldots i_0}$ is represented as an element A of the power set $\mathscr{A}_n := \mathfrak{P}(\{0, \ldots, n-1\})$ where an integer $j \in A$ if and only if $i_j = 1$.
- A polynomial $p \in \mathcal{P}_n$ is represented as an element P of the power set $\mathscr{P}_n := \mathfrak{P}(\mathscr{A}_n)$, where $A \in P$ if and only if $A = 1$ as a coefficient of the polynomial.
- An input $i_{n-1} i_{n-2} \ldots i_0$ to an n-variable boolean function is represented as an element I of the power set $\mathscr{I}_n := \mathfrak{P}(\{0, \ldots, n-1\})$, where an integer $j \in I$ if and only if $i_j = 1$.
- A boolean function $f \in \mathcal{F}_n$ is represented as an element F of the power set $\mathscr{F}_n := \mathfrak{P}(\mathscr{I}_n)$ where $I \in F$ if and only if $F(I) = 1$.

- Observe that since A and I are both elements of the power set $\mathfrak{P}(\{0, \ldots, n-1\})$, it makes sense to refer to one as a subset or superset of the other.

With these conventions, we may define τ as follows (all sums are taken modulo 2):

$$\tau(P) = F, \text{ where } F(I) = \sum_{A \subseteq I} \sum_{A \in P} 1 \tag{18}$$

So as an element of $\mathscr{F}_n$,

$$F = \left\{ I \middle| \sum_{A \subseteq I} \sum_{A \in P} 1 = 1 \right\} \tag{19}$$

I claim that τ is an involution.

Proof. Let $F = \tau(P)$. Since $P \in \mathscr{P}_n = \mathfrak{P}(\mathfrak{P}(\{0, \ldots, n-1\}))$, and $F \in \mathscr{F}_n = \mathfrak{P}(\mathfrak{P}(\{0, \ldots, n-1\}))$, we may then apply τ to F.

$$\tau(F) = G \in \mathscr{F}_n, \text{ where } G = \left\{ J \middle| \sum_{J \subseteq I} \sum_{J \in F} 1 = 1 \right\} \tag{20}$$

Since $J \in F$ iff $F(J) = 1$, and $F = \tau(P)$,

$$G = \left\{ I \middle| \sum_{J \subseteq I} F(J) = 1 \right\} \tag{21}$$

$$G = \left\{ I \middle| \sum_{J \subseteq I} \sum_{A \subseteq J} \sum_{A \in P} 1 = 1 \right\} \tag{22}$$

$$G = \left\{ I \middle| \sum_{J : A \subseteq J \subseteq I} \sum_{A \in P} 1 = 1 \right\} \tag{23}$$

since the second sumation and summand are independent from J

$$G = \left\{ I \middle| \operatorname{card}(\{J : A \subseteq J \subseteq I\}) \sum_{A \in P} 1 = 1 \right\} \tag{24}$$

where $\operatorname{card}(X)$ is the cardinality of the set X. We may write J as the disjoint union $J = A \sqcup K$ for some $K \subseteq I \setminus A$. Then $\operatorname{card}(\{J : A \subseteq J \subseteq I\})$

is equal to the number of subsets of $I \setminus A$. This is necessarily a power of two. Because we are working modulo 2, the only 2-power which is nonzero is $2^0 = 1$. Therefore $I \setminus A = \varnothing$, so $I = A$. So we have

$$G = \left\{ A \middle| \sum_{A \in P} 1 = 1 \right\} \tag{25}$$

$$G = \left\{ A \middle| A \in P \right\} \tag{26}$$

$$\tau(F) = P \tag{27}$$

Since we assumed $\tau(P) = F$, and the choice of P was arbitrary, τ is an involution. □

In addition to the above formula involving lists of coefficients and values, we may also use an algebraic method to convert a boolean function into a polynomial. Given a logical formula for a boolean function, such as

$$(\neg x_0 \wedge \neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_0 \wedge x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_0 \wedge x_1 \wedge x_2 \wedge \neg x_3) \tag{28}$$

and adopting the convention that 1 = true and 0 = false, the logical formula may be translated directly into a polynomial over $\mathbb{F}_2$ by observing the following rules:

$$\begin{aligned} \neg x_i &\mapsto (1 - x_i) \\ A \wedge B &\mapsto AB \\ A \vee B &\mapsto (1 - (1 - A)(1 - B)) = AB + A + B \end{aligned}$$

Applying these rules to the the case of the boolean formula in Equation 28 (which happens to express the function f described in table 1, with x_0 as the least significant bit), gives the polynomial

$$\begin{aligned} &\big((1 - x_0)(1 - x_1)(1 - x_2)(1 - x_3)\big) \vee \\ &\qquad \big(x_0 x_1 (1 - x_2)(1 - x_3)\big) \vee \big((1 - x_0) x_1 x_2 (1 - x_3)\big) \end{aligned} \tag{29}$$

$$\begin{aligned} = 1 - &\big(1 - (1 - x_0)(1 - x_1)(1 - x_2)(1 - x_3)\big) \\ &\big(1 - x_0 x_1 (1 - x_2)(1 - x_3)\big)\big(1 - (1 - x_0) x_1 x_2 (1 - x_3)\big) \end{aligned} \tag{30}$$

$\vdots \quad \vdots$ after much simplification $\vdots \quad \vdots$

$$= x_0x_1x_2x_3 + x_0x_1x_2 + x_0x_2x_3 + x_0x_2 + x_0x_3 + x_0 + x_1x_3 + x_1 + x_2x_3 + x_2 + x_3 + 1 \quad (31)$$

$$= \sum_{\substack{i_3,i_2,i_1,\\ i_0\in\{0,1\}}} a_{i_3i_2i_1i_0} x_3^{i_3} x_2^{i_2} x_1^{i_1} x_0^{i_0} \quad (32)$$

Where $a_{i_3i_2i_1i_0} = 0$ for $i_3i_2i_1i_0 \in \{1110, 1011, 0110, 0011\}$ and $a_{i_3i_2i_1i_0} = 1$ otherwise.

5.3 A Universal Circuit

Any boolean polynomial, and therefore any boolean function, can be expressed using a quantum circuit, made up entirely of Toffoli gates. This is possible because the Toffoli gate performs both of the arithmetic operations essential to constructing a polynomial: addition and multiplication. The Toffoli gate with control bits x and y, and target bit z, gives an output on the target wire of $xy + z$. By using work qubits that are initialized to known states (either $|0\rangle$ or $|1\rangle$), we may express any polynomial using Toffoli gates.

It is also useful to recognize that the CNOT gate performs mod-2 addition, and is in fact the same as the Toffoli gate with one of the control qubits initialized to 1.

The quantum circuit in Figure 6 will evaluate any 2-variable polynomial of the form

$$\sum_{i_1,i_0\in\{0,1\}} a_{i_1i_0} x_1^{i_1} x_0^{i_0} \quad (33)$$

This may easily be extended for functions involving more variables, using the following procedure to build an $n+1$-variable circuit from an n-variable circuit.

1. Start with the circuit for evaluating n-variable polynomials.
2. Duplicate the circuit for evaluating n-variable polynomials, using new qubits for the $|a_k\rangle$ and the existing qubits for the $|x_i\rangle$.
3. Rewrite the labels for the $|a_k\rangle$ qubits by adding a leading one to the binary representations of the old qubits, and a leading zero to the binary representations of the new qubits.

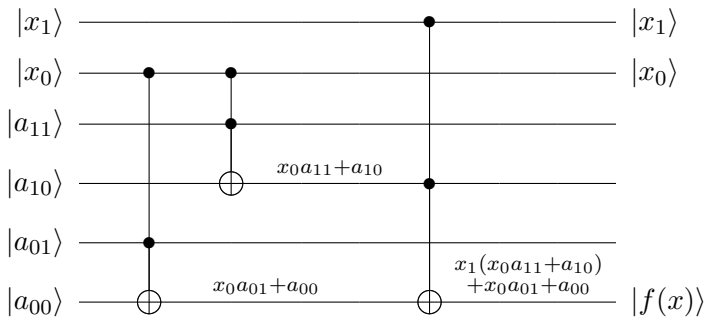


Figure 6: This circuit evaluates the 2-variable polynomial $a_{11}x_1x_0 + a_{10}x_1 + a_{01}x_0 + a_{00}$. Labels have been added to indicate the values carried by each wire after each time it is modified.

4. Add one additional qubit to the top of the circuit, with input $|x_n\rangle$.

5. Add a Toffoli gate with control qubits $|x_n\rangle$ and $|a_{10\ldots0}\rangle$, and target qubit $|a_{00\ldots0}\rangle$

The output remains on the bottom qubit, $|a_{00\ldots0}\rangle$.

The circuit built using this procedure for evaluating m-variable polynomials uses 2^m work qubits (to specify the coefficients of the polynomial) and $2^m - 1$ Toffoli gates. In most cases, this circuit is far from optimal; most functions can be implemented using significantly fewer gates and work qubits. The circuits built using this procedure for $n = 3$ and $n = 4$ are shown in Figures 7 and 8.

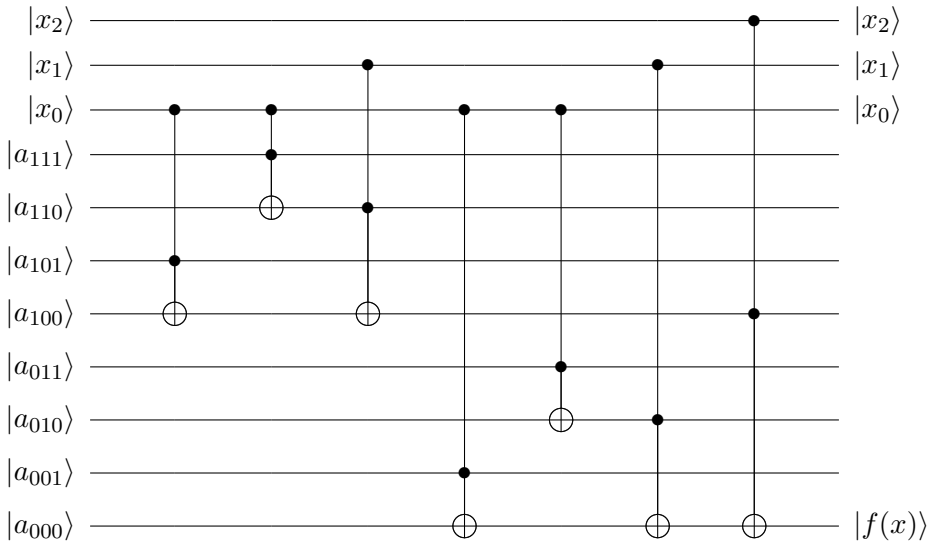


Figure 7: This circuit evaluates any 3-variable polynomial over $\mathbb{F}_2$.

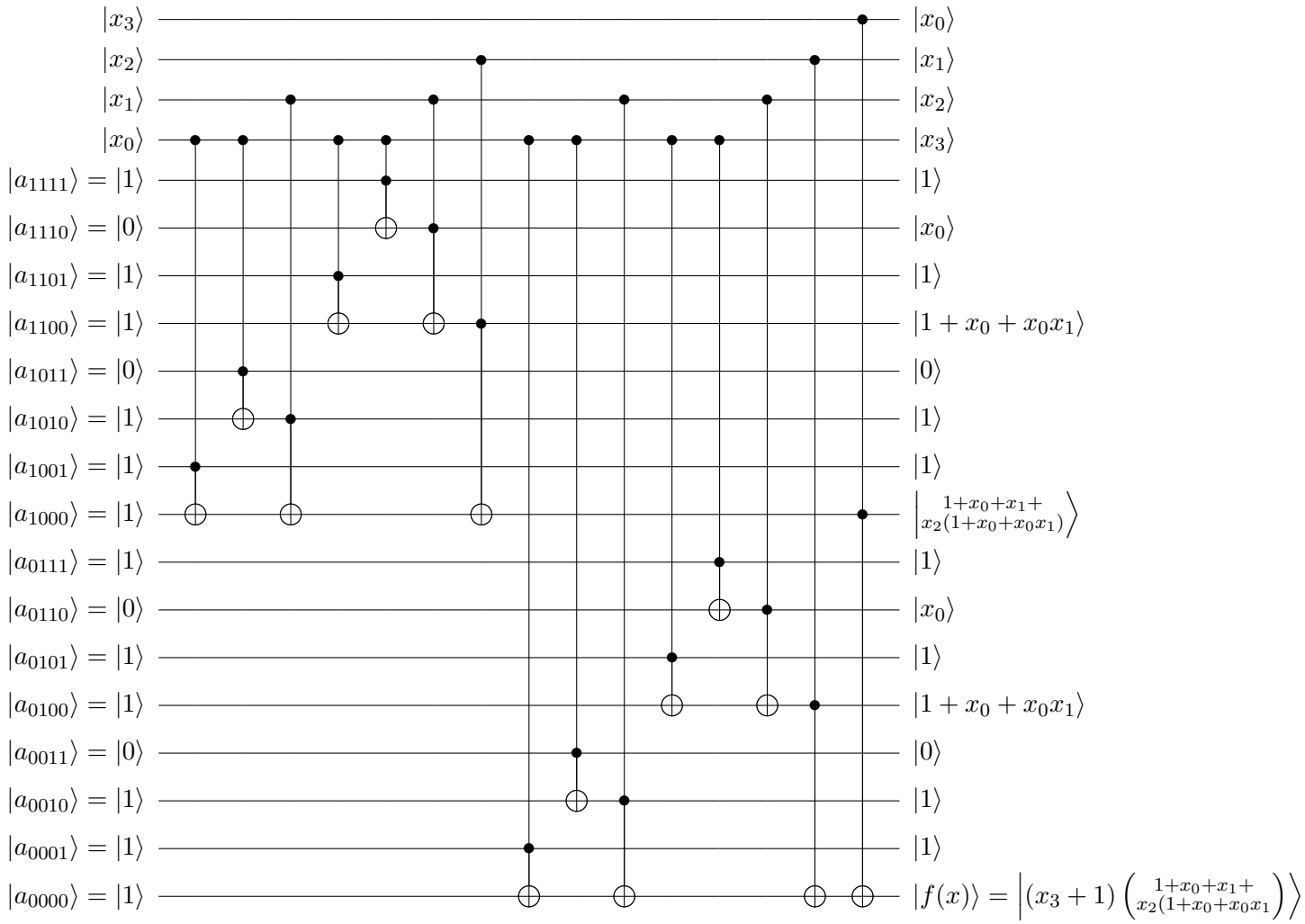


Figure 8: This circuit evaluates any 4-variable polynomial over $\mathbb{F}_2$, with the work qubits initialized to the coefficients of the polynomial. The values given as inputs and outputs correspond to the function originally described in 1, and then in equations 28 through 32. It uses 16 work qubits and 15 gates to represent this function, which is very inefficient, but has the advantage that the same circuit, with different inputs to the work qubits, can evaluate any 4-variable polynomial.

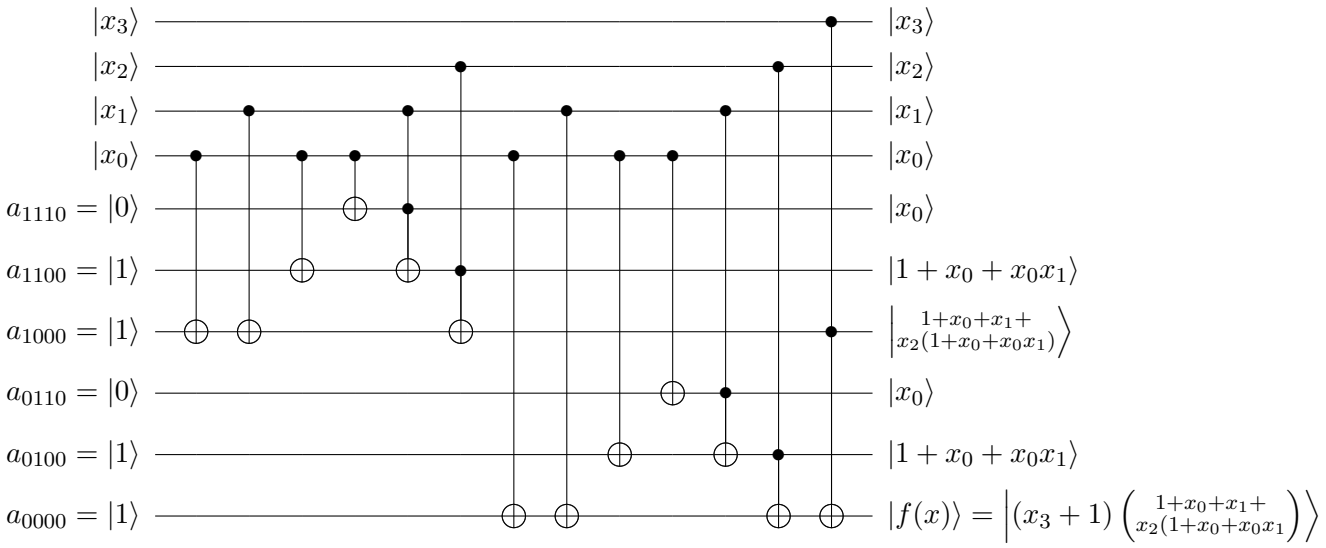


Figure 9: This is a simplified version of the circuit in Figure 8. It is no longer able to evaluate any arbitrary polynomial; I am showing the process by which redundant gates and work qubits can be eliminated. Toffoli gates with one control always at zero have been deleted, since they will act as the identity. After eliminating these gates, all work qubits that are never changed were eliminated, with the Toffoli gates they were controlling replaced by CNOT gates. It uses 13 gates and 6 work qubits, down from the 15 gates and 16 work qubits required in the general circuit. It is still far from optimal.

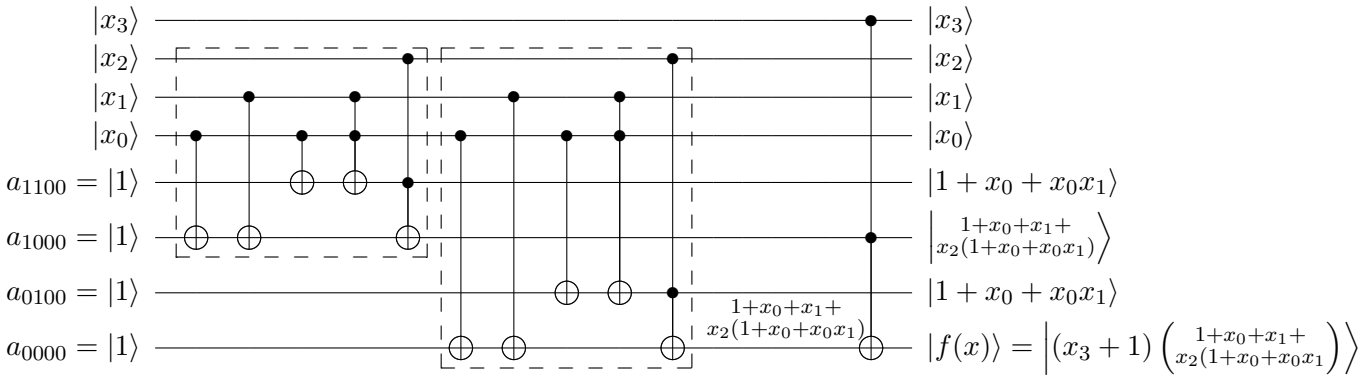


Figure 10: This is a further simplification of the circuit in Figure 9. The qubits initialized to zero, with a final value of $|x_0\rangle$, have been eliminated, since we can just use the $|x_0\rangle$ wire for a control instead. It uses 11 gates and 4 work qubits. The fact that the two boxed sections compute the same value ($|1 + x_0 + x_1 + x_2(1 + x_0 + x_0x_1)\rangle$) will be exploited in the next simplification.

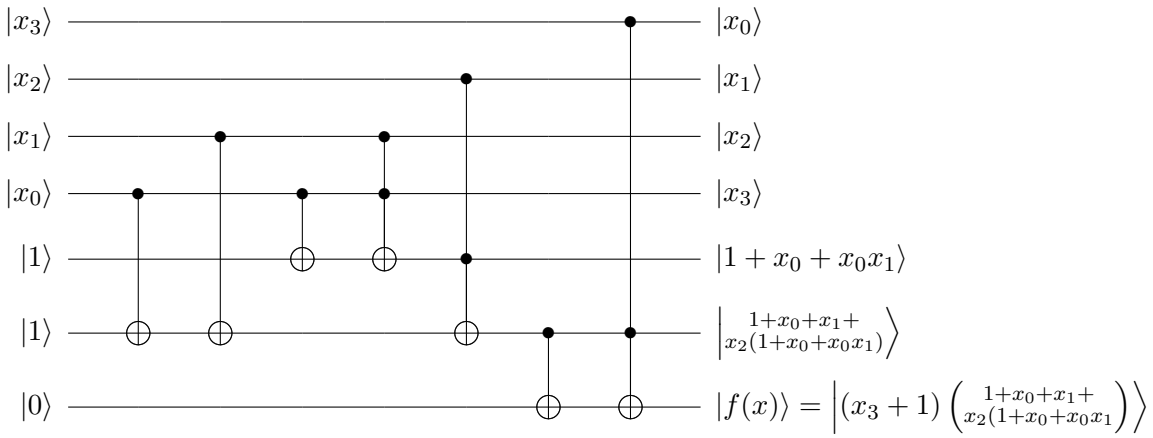


Figure 11: This is a further simplification of the circuit in figure 10. The simplification is based on the recognition that in the previous circuit, two boxed regions both computed the value $|1 + x_0 + x_1 + x_2(1 + x_0 + x_0x_1)\rangle$. Instead of duplicating that effort, this circuit computes it once, then "copies" it to the bottom-most wire using a CNOT gate. It uses 8 gates and 3 work qubits. I was unable to simplify the circuit any more than this, but I was able to create a completely new circuit which uses fewer gates.

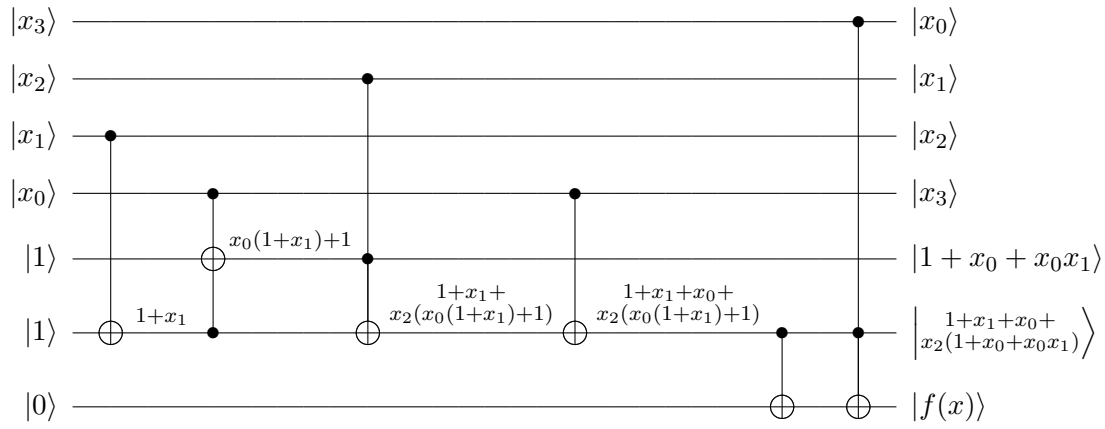


Figure 12: I created this circuit from scratch, to evaluate the polynomial $(x_3 + 1)(x_2(x_0(x_1 + 1) + 1) + x_0 + x_1 + 1)$, which is a factored form of the polynomial used in Figures 9 through 11, and in equation 32. It uses 3 work qubits and 7 gates. While I cannot prove that this is the optimal circuit, it is the best I have been able to find.

6 My Implementation of Grover's Algorithm

By simulating a quantum computer on a classical computer, the user is able to see the probability amplitudes of each computational basis state at any given point: information that would be unknowable on an actual quantum computer. This can provide valuable insight for designing new quantum algorithms. However, this insight comes at significant cost. Not surprisingly, quantum computer simulators are much slower than an actual quantum computer, and require significantly more resources, albeit classical resources rather than quantum ones.

I implemented Grover's algorithm using MATLAB. Detailed usage instructions for the program `qsearch.m` are provided at the beginning of Section 7.

In order to accurately simulate a quantum computer using a classical computer, it is necessary to keep track of the probability amplitudes of every computational basis state. Since there are $N = 2^n$ computational basis states in an n-qubit system, the data processing requirements become unwieldy quite quickly. Depending on the particulars of the system on which it is running, and the options for optimization chosen, the computations start to become infeasible somewhere around 13-17 qubits.

The simplest way to implement the unitary transformations effected by each quantum gate is as a unitary matrix. However, Because the unitary matrices required to implement a transformation on n qubits have dimensions N by N, for even modest values of n, they require a very large amount of memory to store. (A matrix for operating on 14 qubits requires 2 gigabytes of RAM, with each additional qubit quadrupling the memory required.) Even if you have the RAM necessary to store them, implementing unitary operators this way means that each time a unitary operation is computed requires N^2 multiplications and $N^2 - N$ additions. Fortunately, there are some optimization strategies that enable the simulation of more qubits using fewer resources. The command-line options for the program allow the user to specify which of these optimizations are used; the default options are generally the most efficient, but there may be a few exceptions to this.

One very straightforward way to reduce both memory usage and the number of computations necessary to run the simulation is to use sparse matrices whenever practical. A sparse matrix is a more efficient means of storing a matrix on a computer, provided the matrix contains mostly zeros. The unitary matrices for the phase-shift and oracle gates both satisfy this requirement, as they each have N nonzero entries, and $N^2 - N$ zero entries. By storing only the nonzero entries, together with their locations, storage

space is greatly reduced. Additionally, operations on sparse matrices are significantly faster, because MATLAB only performs the calculations that will give nontrivial results. For example, in computing the product

$$\begin{bmatrix} 1 & 2 & -3 \end{bmatrix} \begin{bmatrix} -1 & 0 & 2 \\ 2 & 0 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

with the square matrix stored as a standard (full) matrix, MATLAB would perform many unnecessary calculations, most of which would consist of multiplication by zero or addition of zero. If the square matrix is stored as a sparse matrix, MATLAB "knows" that the zero entries won't end up contributing to the final result, and skips all those steps. So in this example, performing the multiplication using a full matrix would involve 9 multiplications and 6 additions, while computing it using a sparse matrix would involve only 4 multiplications and 2 additions.

Unfortunately, the Hadamard matrix has no zero entries, and nearly as many -1's as 1's, making it a poor choice for representation using a sparse matrix. This would seem to create a hard limit on most mainstream hardware at $n = 15$, since the Hadamard matrix for $n = 16$ requires 32GB of RAM. Fortunately, there is an alternative: the Fast Walsh-Hadamard Transform (FWHT). Similar to the Fast Fourier Transform, the FWHT is an algorithm for computing the Hadamard transform of a vector of length N using $N \log_2(N)$ operations. This is an improvement over the N^2 operations required for multiplication by the Hadamard matrix, and more importantly, it doesn't require storing the entire matrix in memory. However, in the tests I've run, using MATLAB's built-in FWHT function is actually slower than the matrix multiplication. I can think of two possible explanations for this. The first is that MATLAB automatically parallelizes matrix multiplication, allowing it to operate on multiple processor cores simultaneously. The FWHT function is executed on only one core at a time.

The second reason is that matrix multiplication consists of a fixed set of multiplications, additions and subtractions, which are only calculated once (when the matrix is created), and then implemented for each multiplication. The built-in FWHT function has to determine which additions and subtractions to perform every time it is called. Since the list of $N \log N$ additions and subtractions to perform in the FWHT algorithm depends only on the length of the vectors, it can be calculated once at the beginning of the program, stored in memory, and then recalled each time it is needed, rather than recalculating it. This provides a tremendous (in some cases 100-fold) speedup over using the built-in FWHT function.

The comparison between the pre-calculated function and matrix multiplication is not quite as clear. For reasons I have not been able to determine, while the matrix multiplication has a consistent runtime for a given number of qubits, the pre-calculated function method is sometimes much faster than matrix multiplication, and sometimes much slower, even when run with the same options on the same computer. I believe that the pre-calculated method is superior, as it seems to be so theoretically, and in practice its fastest runtimes are a significant improvement over the fastest runtimes of the matrix multiplication method. The pre-calculated method also uses much less RAM than the matrix multiplication method, allowing it to run with a greater number of qubits.

Another aspect of my implementation which deserves mention is the oracle. In a sense, the program is "cheating": it knows what index values it is searching for, whether they were input by the user, or randomly generated by the computer, and uses that knowledge to create the oracle matrix. However, this behavior is a direct consequence of the fact that this is a simulation. As discussed in Section 5, building a quantum oracle only requires a circuit that recognizes a solution to a search problem by flipping a given qubit. By using quantum superposition, the quantum computer is able to evaluate that circuit for all possible inputs and find one that is a solution to the search problem. Absent quantum superposition, the classical simulator is forced to manually evaluate the circuit at all possible inputs in order to determine the appropriate unitary operator for the oracle. Since this would also result in the computer knowing the answer ahead of time, it did not seem worthwhile to implement a routine that could take as input a function in some form other than a list of values that return true.

7 MATLAB Code

The following code is color-coded to aid readability.

```
function [ComputerState,logdata] = qsearch( varargin)
% QSEARCH Perform a quantum search using Grover's algorithm.
%
%
%
% qsearch(n, M, ['opts']) performs a quantum search on an n-qubit database
% looking for M randomly chosen solutions. n must be an integer, and M
% must be a scalar.
%
% qsearch(n, ['opts']) is equivilant to qsearch(n, 1, ['opts']).
%
% qsearch(n, ANSWERS, ['opts'] ), with ANSWERS a vector of length > 1,
% performs a quantum search on an n-qubit database looking for the values
% specified in answers. This will not work if you want to search for only
% one pre-specified value.
%
% qsearch(n, 1, ANSWER, ['opts'] ), with answer a scalar will perform a
% quantum search on an n-qubit database looking for the value specified in
% answer. M will be set to 1.
%
%
% 'opts' is an optional string that may be used to control the
% methods used to conduct the search. 'opts' may contain up to five
% characters, in any order, one for each of the five parameters that may be
% adjusted: matrix method, Hadamard method, logging method, warnings, and the
    number
% of iterations. 'opts' is not case sensitive.
%
% Matrix method controls the type of matrix used to create the oracle and
% phase-shift matrixes. Characters controlling the matrix method are:
%
% 's' Use sparse double-precision matrices (the default). This uses 16*2^n
% bytes of memory to store the phase-shift and oracle matrices.
% 'd' Use double-precision, full matrices. This will require 16*2^(2n)
% bytes of memory to store the phase-shift and oracle matrices.
% 'i' Use single-precision, full matrices. This will require 8*2^(2n)
% bytes of memory to store the phase-shift and oracle matrices.
%
% NOTE: MATLAB does not support sparse, single-precision matrices.
%
%
% Hadamard method controls the method used to compute the hadamard
% transform. This can have a significant effect on runtime, since the
% Hadamard transform is used twice in each iteration. Characters
% controlling the Hadamard method are:
```

```
%
% 'p' Pre-compute the necessary additions and subtractions to apply the
% Hadamard transform on n qubits, and then apply those additions and
% subtractions using the function prehadamard.m. For large numbers of
% qubits, this provides a very large speedup over the other two methods.
% This is the default.
% 'f' Use the built-in function FWHT from the signal processing toolkit to
% compute the Hadamard transform.
% 'm' Use a 2^n by 2^n matrix to compute the Hadamard transform. This is
% generally is faster than using the built-in function, but slower than the
% precomputed function.
%
%
% Logging method controls whether and how the intermediate states of the
% computer are stored. Characters controlling the output method are:
%
% 'r' At each iteration, store the amplitudes of every computational basis
% state as a row in the matrix OUTPUTS.
% 'k' At each iteration, write the amplitudes of every computational basis
% state as a line in the tab-separated file 'iterations.txt'. This will
% significantly increase runtime if file I/O to the location of
% 'iterations.txt' is slow (i.e. the file is being transferred over a
% network).
% 'n' Do not log the amplitudes of each computational basis state. Only
% output the final state after running through itmax iterations. This is
% the default.
%
% By default, the program prints to stderr the parameters for a search
% before executing that search, and issues several warnings designed to
% avoid accidential out-of memory errors and their effect on the
% performance of your entire computer. To disable this behavior, include
% the character 'u' in the control string.
%
% The number of iterations controls how many times the program executes the
% Grover operator. By default, the program calculates the number of
% iterations according to the following equation:
% theta=asin(2*sqrt(M*(N-M))/N);
% itmax=round(acos(sqrt(M/N))/theta);
%
% If any character is specified for the number of iterations, an additional
% integer argument ITMAXADJ must be given after the 'opts' string.
% Characters controlling the number of iterations are:
%
% 't' Ignore the formula for computing itmax, and set itmax to the value
% of itmaxadj. This can be useful in combination with some of the output
% methods to see the behavior of the Grover operator over a large number of
% iterations.
% 'a' Adjust itmax by itmaxadj. This sets itmax=itmax+itmaxadj. This can
% be useful in combination with some of the output methods to see the
```

```
% behavior of the Grover operator in the vicinity of the calculated
% solution.



% start by reading the input arguments, and handling them appropriately.


if nargin<1
    error('Not enough input arguments.')
elseif ischar(varargin{1})
    error('n must be an integer.')
elseif length(varargin{1})>1
    error('n must be a scalar')
end


n=varargin{1};
N=pow2(n);

% ------------------------------------------------------------------
% Figure out whether the function we're using for the search has been
% specified on the command line, and if not, generate it.
% ------------------------------------------------------------------

% if second argument not specified, or is an option string, assume we're
% searching for only one solution
if nargin==1 || ischar(varargin{2})
    M=1;
    optstart=2;

    % if second argument is a vector, interpret it as the list of answers
elseif length(varargin{2})>1
    answers=varargin{2};
    optstart=3;

    % if second argument is 1 and third argument is a scalar,
    % third argument is the answer to look for.
elseif nargin>2 && varargin{2}==1 && length(varargin{3})==1 ...
        && ~ischar(varargin{3})
    answers=varargin{3};
    optstart=4;

    % If we've reached this point, there are at least two arguments, the
    % first two arguments are both integers, and the second argument is
    % a scalar other than one. Interpret the second argument as M.
else
    M=varargin{2};
    optstart=3;
```

```
end

    % if we don't know what the answers are yet, we need to generate them.
if ~exist('answers','var')
    rng('shuffle');
    i=1;
    while i<=M
        answers(1,i)=randi(N,1)-1;
        % make sure we've added a new solution. If the new solution was already
        % there, repeat the loop with the same value of i to pick another.
        if size(answers)==size(unique(answers))
            i=i+1;
        end
    end
else
    answers=unique(answers);
    M=length(answers);
end

% sanity checks
if M>N
    error('You''re asking me to find more than N solutions in N entries!')
end
if max(answers)>N
    error('You''re asking me to find a value that is too large!')
end

% determime the number of times to run the grover operator

theta=asin(2*sqrt(M*(N-M))/N);
itmax=round(acos(sqrt(M/N))/theta);

% ------------------------------------------------------------------
% Parse options string, and set appropriate options
% ------------------------------------------------------------------

% check if any options were set
if nargin>=optstart &&ischar(varargin{optstart})

    % check for the matrix method. Sparse is the default.
    if max(varargin{optstart}=='D') || max(varargin{optstart}=='d')
        matrixmethod='d';
    elseif max(varargin{optstart}=='I') || max(varargin{optstart}=='i')
        matrixmethod='i';
    else
        matrixmethod='s';
    end

    % check for the hadamard method. Matrix is the default.
```

```
    if max(varargin{optstart}=='F') || max(varargin{optstart}=='f')
        hadamardmethod='f';
    elseif max(varargin{optstart}=='M') || max(varargin{optstart}=='m')
        hadamardmethod='m';
%    elseif max(varargin{optstart}=='P') || max(varargin{optstart}=='p')
%        hadamardmethod='p';
%        if matlabpool('size') <=1 && warn
%            warning(['You''re trying to compute the hadamard function in\n',...
%                'parallel, using only one worker. Run ''matlabpool'' to\n',...
%                'start more workers. \n',...
%                'Press Ctrl+C to abort or enter to continue'])
%            pause
%        end
    else
        hadamardmethod='p';
    end

    % check for the loggingmethod
    if max(varargin{optstart}=='R') || max(varargin{optstart}=='r')
        loggingmethod='r';
    elseif max(varargin{optstart}=='K') || max(varargin{optstart}=='k')
        loggingmethod='k';
    else
        loggingmethod='n';
    end

    % should we allow pausing?
    if max(varargin{optstart}=='U') || max(varargin{optstart}=='u')
        pause off;
        warn=false;
    else
        pause on;
        warn=true;
    end

    % check to see if we need to adjust itmax
    if nargin>optstart && ~ischar(varargin{optstart+1}) ...
        && length(varargin{optstart+1})==1
            if max(varargin{optstart}=='t') || max(varargin{optstart}=='T')
                itmax=varargin{optstart+1};
            elseif max(varargin{optstart}=='a') || max(varargin{optstart}=='A')
                itmax=itmax+varargin{optstart+1};
            end
    end
else % if no options were set, use the defaults.
    matrixmethod='s';
    loggingmethod='n';
    hadamardmethod='m';
    warn=true;
```

```
    pause on;

end









%

%
% % switch loggingmethod
% %     case {'d','disk','disk space'}
% %         loggingmethod='disk space';
% %     case {'m','memory','mem','r','ram','RAM'}
% %         loggingmethod='memory';
% %     case {'n','none',[]}
% %         loggingmethod='none';
% %     otherwise
% %         error('I don''t know that output method. \nValid options are ''n'', ''
    none'', ''m'', ''memory'',''d'', ''disk''.')
% % end
%



% Make a lookup table for translating the single-character options to
% human-readable ones.

trans{'s'}='Sparse';
trans{'d'}='full, Double';
trans{'i'}='full, sIngle';

trans{'p'}='Pre-computed function';
trans{'f'}='built-in Function';
trans{'m'}='Matrix';

trans{'r'}='memoRy';
trans{'k'}='disK';
trans{'n'}='None';

trans{'u'}='do not paUse';

if warn
    % print to stderr
    fprintf(2,['Running quantum search using the following parameters:\n' ...
        'n=%.0f\t\tmatrixmethod=%s\nM=%.0f\t\thadamardmethod=%s\nitmax=%.0f\
            toutputmethod=%s\n ']...
```

```
        ,n,trans{matrixmethod},M,trans{hadamardmethod},itmax,trans{loggingmethod})
            ;

    % estimate storage needs, to avoid out of memory errors

    % RAM used by oracle and phase matrices
    switch matrixmethod
        case 's'
            ramused=2*24*N;
        case 'i'
            ramused=2*4*N^2;
        case 'd'
            ramused=2*8*N^2;
    end

    % RAM used by the hadamard matrix
    switch hadamardmethod
        case 'm'
            ramused=ramused+8*N^2;
    end

    % RAM used to store the computer state
    ramused=ramused+8*N;

    % RAM/disk space used to store the output data
    switch loggingmethod
        case 'r'
            ramused=ramused+8*N*itmax;
        case 'k'
            diskused=10*N*itmax;
            if diskused>pow2(500,20)
            warning(['This will require %g megabytes of disk space to store the
                output\n' ...
                'Are you sure you want to continue?\n' ...
                'Press enter to continue, or Ctrl+C to abort.'],pow2(diskused,-20))
                pause
            end
    end


    if ramused>pow2(2,30) && warn
        warning(['This will require %g gigabytes of memory\n' ...
            'Are you sure you want to continue?\n' ...
            'Press enter to continue, or Ctrl+C to abort.'],pow2(ramused,-30))
        pause
    end

% if n>14 && ~isequal(loggingmethod(1),'n') && warn
%     warning(['You''ve asked me to do a computation on %.0f qubits. \n' ...
```

```
%        'This will require more than %g gigabytes of %c just to store' ...
%        'the output and sparse matrices, and on the order of %g operations. \n \n
    ' ...
%        'This is likely to take a long time.\n' ...
%        'Are you sure you want to continue?\n' ...
%        'Press enter to continue, or Ctrl+C to abort.'],n,pow2(outsize,-30),
    loggingmethod,numcalcs)
%    pause
% end

end
% ------------------------------------------------------------------
% Beginning of the actual program
% ------------------------------------------------------------------



switch hadamardmethod
% If we're using matrices, generate a hadamard matrix of appropriate type
    case 'm'
        switch matrixmethod
            case 'i'
                H=single(hadamard(N)/sqrt(N));
            otherwise
                H=hadamard(N)/sqrt(N);
        end
% If using the custom hadamard function, compute which additions and
% subtractions to do ahead of time, so we only have to do it once.
    case 'p'
       [index1,index2]=hadsorts(n);
end


% Set up where we're going to store output, if we are going to store output.
switch loggingmethod
    case 'r'
        % pre-allocate memory for the output array. For large values of itmax or N
            ,
        % this is significantly faster than building the array one row at a time.
        logdata(1:itmax,1:N)=0;
    case 'k'
        logfile=fopen('iterations.txt','w');
        logdata=[];
    case 'n'
        logdata=[];
    otherwise
        error('unknown loggingmethod');
end
% Set up the oracle and phase-shift matrices
```

```
switch matrixmethod
    case 'i'
        OracleMatrix=generate_single_oracle(n,answers);
        PhaseMatrix=generate_single_phase(n);
    case 's'
        OracleMatrix=generate_sparse_oracle(n,answers);
        PhaseMatrix=generate_sparse_phase(n);
    case 'd'
        OracleMatrix=generate_full_oracle(n,answers);
        PhaseMatrix=generate_full_phase(n);
    otherwise
        error('unknown matrixmethod');
end


% ComputerState is the current state vector of the computer
% initialize the ComputerState to |0>
ComputerState(1:N)=0;
ComputerState(1)=1;
% intial state has computer at all zeros. Amplitude of the all-zero state
% is 1
% oracle qubut is initialized to (|0>-|1>)\sqrt(2), and remains unchanged
% throughout the computation, so we don't need to simulate it.

% Do initial Hadamard transformation to create superposition
switch hadamardmethod
    case 'm'
        ComputerState=ComputerState*H;
    case 'p'
        ComputerState=prehadamard(ComputerState,index1,index2);
    case 'f'
        ComputerState=fwht(ComputerState,[],'hadamard');
end

% set up logging, if applicable
switch loggingmethod
    case 'r'
        logdata(1,:)=ComputerState;
    case 'k'
        for w=1:N
            fprintf(logfile,'%d\t',ComputerState(w));
            fprintf(logfile,'\n');
        end
end

% do the computations
for i=1:itmax
    ComputerState=ComputerState*OracleMatrix;
    switch hadamardmethod
```

```
        case 'p' % use precomputed func for hadamard gates
            ComputerState=prehadamard(ComputerState,index1,index2);
            ComputerState=ComputerState*PhaseMatrix;
            ComputerState=prehadamard(ComputerState,index1,index2);
        case 'f' % use built-in func for hadamard gates
            ComputerState=fwht(ComputerState,[],'hadamard');
            ComputerState=ComputerState*PhaseMatrix;
            ComputerState=fwht(ComputerState,[],'hadamard');
        case 'm' % use matrices for hadamard gates
            ComputerState=ComputerState*H;
            ComputerState=ComputerState*PhaseMatrix;
            ComputerState=ComputerState*H;
    end
    switch loggingmethod
        case 'r' % log intermediate states to memory
            logdata(i+1,:)=ComputerState;
        case 'k' % log intermediate states to file
            for w=1:N
                fprintf(logfile,'%d\t',ComputerState(w));
                fprintf(logfile,'\n');
            end
    end
end
```

```
function x=fasthadamard(x)
% This function implement the Hadamard Transform.
% Code was based on fwht1d.m, written by Gylson Thomas,
% gylson_thomas@yahoo.com, in 2005-2007.
% Code was extensively modified by Matthew Bush, mbbush@gmail.com, in 2013
% to make it parallelizable.
n = floor(log2(length(x)));
N = pow2(n);
x = x(1:N);
% indexes(1:N/2*n,1:9)=0; % with troubleshooting data
%indexes(1:N/2*n,1:2)=0; % without troubleshooting data
index1(1:N/2*n)=0;
index2(1:N/2*n)=0;
% k1=N; k2=1; k3=N/2;
% logfile=fopen('hadamard5.csv','w');
% fprintf(logfile,'k1\t k2\t k3\t i1\t i2\t i3\t L1\t i\t j\t x(i)\t x(j)\n');
i1step=pow2(n-1);
% ops(1:N,1:N*n,1:2)=0;
% working in parallel, generate a list of operations to do in serial later
parfor ind=1:N/2*n


    i1=ceil(ind/i1step) % i1 runs from 1 to n, increasing by 1 every N/2
        iterations
```

```
    k1=pow2(n+1-i1); % k1 runs from N to 2, decreasing by factor 2 each time i1 is
          incremented
  %  k2=pow2(i1-1); % k2 runs from 1 to N/2, increasing by factor 2 each time i1
       is incremented
    k3=pow2(n-i1); % k3 runs from N/2 to 1, decreasing by factor 2 each time i1 is
          incremented
    i2=ceil((mod(ind-1,i1step)+1)/k3); % i2 runs from 1 to k2 increasing by 1
        every k3 iterations
                          % and resetting to 1 each time i1 is incremented
    i3=mod(ind-1,k3)+1; % i3 runs from
    L1=i2*k1-k1;
    i=i3+L1;
    j=i+k3;
%     indexes(ind,3:9)=[k1,k2,k3,i1,i2,i3,L1]; % troubleshooting data
    index1(ind)=i;index2(ind)=j;
%     ops(i,ind)=j;  % later on, add j to i.
%     ops(j,ind)=-i;
end

% working serially, apply the hadamard transform to each pair of indexes
% specified in the array indexes.
for k=1:length(index1)
          temp1= x(index1(k));
%            fprintf(logfile,'%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\n
    ',...
%               indexes(k,3),indexes(k,4),indexes(k,5),indexes(k,6),...
%               indexes(k,7),indexes(k,8),indexes(k,9),...
%               indexes(k,1),indexes(k,2),x(indexes(k,1)),x(indexes(k,2)));
          x(index1(k)) = temp1 + x(index2(k));
          x(index2(k)) = temp1 - x(index2(k));
end
x=x/sqrt(N);
```

```
function [ OracleMatrix ] = generate_sparse_oracle( NumQubits, Answer )
%generate_single_oracle
%   Answer is the vector containing all the values for which the search
%   criterion is true.
%   This implementation makes OracleMatrix a single-precision
%   matrix that can handle the case of multiple solutions.

OracleMatrix=speye(2^NumQubits);
for i=1:length(Answer);
    OracleMatrix(Answer(i)+1,Answer(i)+1)=-1;
end

end
```

```
function [ PhaseMatrix ] = generate_sparse_phase( NumQubits )
%generate_single_phase builds a single-precision matrix which flips the
```

```
%sign of the non-zero states, and leaves all-zero state alone.

PhaseMatrix=-speye(2^(NumQubits));
PhaseMatrix(1)=1;


end
```

The functions for generating the full, double-precision and full, single-precision oracle and phase matrices are not included, since they are almost identical to those for generating the sparse matrices.

8 Conclusions

In his original 1996 paper[4], Lov Grover proposed a novel and extremely powerful method for using a quantum computer to search an unstructured database. This provided a quadratic speedup over optimal classical algorithms[5]. Because the algorithm doesn't require any sort of structure, it is applicable to a wide variety of problems, including searching for cryptographic keys, statistical computations, and several classes of **NP** problems[6]. While its speed advantage is not as great as the other major class of quantum algorithms, those based on the quantum Fourier transform, it is considerably simpler to implement.

Using a classical computer to simulate a quantum computer has the advantage of providing access to the amplitudes of each computational basis state at any point in the program. This can be tremendously useful for debugging purposes, especially since it is impossible to observe the amplitudes on an actual quantum computer. While this particular application was not intended to develop a novel algorithm, it did provide insight into how and why Grover's algorithm works.

Using MATLAB as a language to implement the simulation had, as always, advantages and disadvantages.

A Technology description from D-Wave

The information below is published by D-Wave Systems Inc. [3].

Technology Deep Dive

The quantum processor inside the D-Wave One^{TM} is comprised of three main types of component:

Qubits: The most important elements of the quantum processor; the job of the qubits is to be able to store a piece of information either as a 0, or a 1, or—using quantum mechanical effects—as a superposition of both 0 and 1 at the same time.

Couplers: Couplers are elements that connect the qubits together. The job of the couplers is to try to force the two qubits to which they are joined to either be in the same states, (e.g. 00 or 11), or opposite states (01 or 10), depending upon how they are programmed.

Programmable Magnetic Memory: An extensive network of peripheral circuitry surrounds the qubits and couplers. This circuitry consists mainly of digital-to-analog converters and a device addressing system (bus). This circuitry allows each qubit and coupler to be programmed so that the user can specify the problem that they wish to solve.

During processor operation...

In the D-Wave processor, the qubits can slowly be tuned (annealed) from their superposition state (where they are 0 and 1 at the same time) into to a classical state (where they are either 0 or 1). When this is done in the presence of the programmed memory elements on the processor, the 0 and 1 states that the qubits end up settling into gives the answer to a user-defined problem. All circuitry on the D-Wave processors is made from a material known as a superconductor, which is cooled to 20mK, (near absolute zero) in order for the quantum effects to manifest in the material.

References

[1] An image from d-wave company website.

[2] Popular mechanics.

[3] Technology deep dive.

[4] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, STOC '96, New York, NY, USA, 1996. ACM.

[5] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, 2nd edition, May 1998.

[6] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 1st edition, October 2004.