Implementing Grover's Algorithm on a Quantum Computer Simulator

William Bernoudy

Abstract

I explain the basics of quantum computing, the quantum computation simulator I wrote in Scheme, and how Grover's algorithm could be implemented on a quantum computer. I then present a method of designing a oracle down to the gate level from a given classical circuit representing the search function.

1 Introduction

1.1 Grover's algorithm

Grover's algorithm is one of the two important quantum algorithms, or algorithms that run on a quantum computer. Given a search function, which returns true if the data we are searching for is given and false if otherwise, Grover's algorithm is able to find the correct data entry out of N possible entries with a complexity of $O(\sqrt{N})$. This is a significant speedup over the classical algorithm (on a non-quantum computer) of simply checking each data entry one by one which has a complexity of $O(N)$. Because of how common this problem is, Grover's algorithm has a huge amount of applications, from artificial intelligence to protein sequence comparison. Another good example is the 3-SAT problem: while the best know classical algorithm for solving the 3-SAT problem is of time complexity $O(2^n)$, Grover's algorithm can be used to reduce this to $O(2^{n/2})$. Because of the speedups it provides as well as its wide applicability, an implementation of Grover's algorithm will be very useful for the world.

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

1.2 How a quantum computer works

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

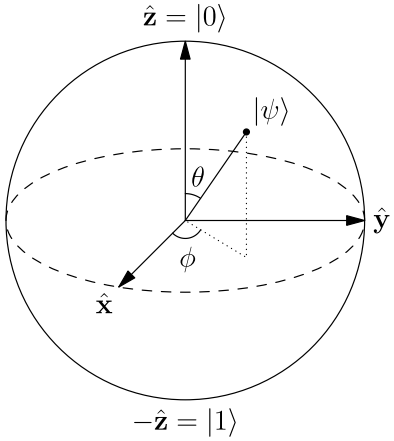
Let's say we have one qubit called ψ. Its vector is given by:

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

where α_0 and α_1 are complex numbers under the single constraint

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

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



A single qubit can be visualized with a Bloch sphere, depicted to the left. As we saw with the single bit before, when the qubit is pointed directly upward (the positive z axis), it has a value of 0. When it is pointed directly downwards (the negative z axis), it has a value of 1. However, unlike the bit, we can see that there are now two axes of rotation afforded to the qubit that were not present before (represented by θ and ϕ). The current state of the qubit in the Bloch sphere is shown by the line with a dot on the end labeled $|\psi\rangle$. We can see that it is pointed more closely upwards, towards the vector, than it is downwards. This means that if we were to observe this qubit, it is more likely that it would collapse to the $|0\rangle$ state. If the qubit were pointed more towards the bottom of the sphere at the vector, it would be more likely that we would collapse to the $|1\rangle$ state. Notice that these likelihoods correspond to the $|\alpha_0|^2 + |\alpha_1|^2 = 1$ equation, where each term is the likelihood that the qubit will collapse to the corresponding state.

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

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

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

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

2 Simulating a quantum computer

Because the state of the computer can be represented with a single vector, an easy way to simulate quantum computation is to simply keep track of this vector. At the end of our computation, we

can examine the vector to see which state the system is most likely to collapse into, and thus check to see if our computation worked.

To create a system of n-qubits, we simply make a vector length 2^n of all zeros. Then, depending on what we want the initial values of qubits to be, we make one of the zeros in the vector a 1. The position of the 1 corresponds to the decimal form of the binary representation of the bits, e.g. if we want to start with $|100\rangle$ as the initial state, then we would change the zero in position 4 of the vector to a 1.

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

For classical compuatation, where the state of the system must be vector consisting of all zeros and one "1" (because it can only be in one state at once), the only valid matrices (meaning those that correspond to physically possible gates) are ones that always produce a valid system after the multiplication. If a matrix in classical computation produces anything by a vector of all zeros and one "1", then we know it's invalid. However, the only requirement for a quantum gate is that it is reversible. This means that matrices that produce vectors with lots of different values are perfectly fine; this actually just represents superposition of the qubits.

Most quantum gates only act on 1, 2 or 3 qubits. However, we cannot use a multiply the smaller matrix that represents these gates with the much larger n-length vector representing the system. Thus, we need a way generate a larger matrix that performs the gate on only the qubits we want. Luckily, Lee Spector has provided an algorithm for that very purpose in his book *Automatic Quantum Computer Programming: A Genetic Programming Approach* [4]:

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

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

Figure 1: Lee Spector's algorithm, quoted from p.21

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

3 Implementing Grover's algorithm

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

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

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

Because the oracle actually would consist of a series of quantum gates, we wouldn't simply be able to see where the -1 is and declare that our solution. However, in order that we don't have to come up with an oracle for testing, we can simply use a matrix of the above form.

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

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

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

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

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

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

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

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

Thus, altogether, the Grover diffusion operator is given by

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

And the entire Grover step is given by

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

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

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

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

The reason that we only have to apply the Grover step $\frac{\pi}{4}\sqrt{2^n}$ has to do with how the algorithm works, which is not something I examined in my project. However, notice that the algorithm has a time complexity of $O(\sqrt{N})$, a massive speedup from the $O(N)$ time that the classical algorithm of checking each entry requires.

To implement my simulation and this algorithm, I used Scheme because Lisp (the family of languages Scheme belongs to) will likely be used to actually run algorithms on future quantum computers. I thus designed my code in such a way that the simulation with matrices can be ignored, and the code would require minimal changes to actually run on a quantum computer.

Here is my implementation of Grover's algorithm:

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

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

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

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

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

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

4 Generating the oracle

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

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

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

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

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

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

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

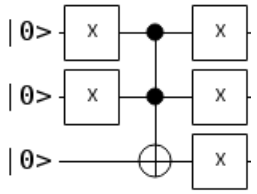


Figure 2: Representation of OR gate using 1 Toffoli gate and 5 NOT gates

For every AND and every OR in the classical circuit we are given, we will need one extra qubit. The extra qubit is always the third bit for the Toffoli gate. We now have every part to simulate the classical circuit on the quantum circuit. However, this is not yet our oracle.

At this point, we have a ciruit takes the input $|x\rangle \left|0^k\right\rangle$ and gives $|junk(x)\rangle |f(x)\rangle$. However, we need it to be of the form [5]

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

What this means is that we need to preserve the state of $|x\rangle$ through the circuit. Though this wouldn't be possible on a classical computer, we can use a handy little trick called "uncomputation" for the quantum ciruit. Because every quantum gate is reversible, to get $|x\rangle$ back, we just have to perform all our gates in backward order. However, before we do that, we copy the result of the search function onto a new qubit using a CNOT gate. Then we can perform the uncomputation on the rest of the qubits, resulting in the desired form for the output.

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

1. Given an input function in the form of a classical circuit, we simulate each classical gate with a quantum gate, taking note of the order and which qubits we applied the gates to.
2. Copy the output of the function, which should be the state of *nth* qubit, to an extra qubit.
3. Perform the uncomputation by applying the same quantum gates we applied before but in the opposite order.

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

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

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

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

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

5 Conclusion

By representing a quantum computer as a series of matrix operations, I was able to successfully simulate quantum computation as well as perform Grover's algorithm. Furthermore, I was able

to come up with a simple algorithm for generating a quantum circuit of only common quantum gates that simulates a given classical circuit. As this is a necessary step in order to use Grover's algorithm in a real world situation, this demonstrates how quantum computers might be used in the future.

Appendix A quetzal.rkt

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

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

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

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

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

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

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

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

(define register null)

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

(define initialize-register (lambda (lq)
```

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

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

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

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

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

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

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

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

```
   [1 0]
]))

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

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

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

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

(define Toffoli-gate (matrix [ ; also known as the CCNOT gate
   [1 0 0 0 0 0 0 0]
   [0 1 0 0 0 0 0 0]
   [0 0 1 0 0 0 0 0]
   [0 0 0 1 0 0 0 0]
   [0 0 0 0 1 0 0 0]
   [0 0 0 0 0 1 0 0]
   [0 0 0 0 0 0 0 1]
   [0 0 0 0 0 0 1 0]
]))
```

Appendix B Grover.rkt

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

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

(require "quetzal.rkt")
```

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

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

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

(define H-matrix null)

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

(define pf-matrix null)

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

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

(define U_omega null)

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

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

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

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

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

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

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

Appendix C oracle-constructor.rkt

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

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

(require "quetzal.rkt")
(require "Grover.rkt")

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

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

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

(define total-qubits 0)

(define next-safe-qubit 0)
```

```
(define uncomputer null)

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

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

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

(define NOT (lambda (qubit)
   (set! computer-strings-for-latex (string-append computer-strings-for-latex
       (string-append "\tX\tq" (number->string qubit) "\n"))) ; just for LaTeX output
   (set! U_omega (matrix* U_omega (Pauli-X total-qubits qubit)))
   (set! uncomputer (matrix* (Pauli-X total-qubits qubit) uncomputer))
   (set! uncomputer-strings-for-latex (string-append (string-append "\tX\tq"
       (number->string qubit) "\n") uncomputer-strings-for-latex)) ; just for LaTeX
       output
   qubit))

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

(define AND (lambda (qubit1 qubit2)
   (set! U_omega (matrix* U_omega (Toffoli total-qubits (list qubit1 qubit2
       next-safe-qubit))))
   (set! computer-strings-for-latex (string-append computer-strings-for-latex
       (string-append "\ttoffoli\tq" (number->string qubit1) ",q" (number->string
       qubit2) ",q" (number->string next-safe-qubit) "\n"))) ; just for LaTeX output
   (set! uncomputer (matrix* (Toffoli total-qubits (list qubit1 qubit2 next-safe-qubit))
       uncomputer))
   (set! uncomputer-strings-for-latex (string-append (string-append "\ttoffoli\tq"
       (number->string qubit1) ",q" (number->string qubit2) ",q" (number->string
       next-safe-qubit) "\n") uncomputer-strings-for-latex)) ; just for LaTeX output
   (set! next-safe-qubit (+ next-safe-qubit 1))
```

```
   (- next-safe-qubit 1))) ; This, by defintion, is the output of the toffoli gate (the
       third qubit), so that's what we want to return

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

(define get-base-qubits (lambda (boolean-expression)
   (cond
      [(null? boolean-expression) 0]
      [(number? (car boolean-expression)) (max (car boolean-expression) (get-base-qubits
          (cdr boolean-expression)))]
      [(list? (car boolean-expression)) (max (get-base-qubits (car boolean-expression))
          (get-base-qubits (cdr boolean-expression)))]
      [else (get-base-qubits (cdr boolean-expression))])))

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

(define U_omega null)

(define input-qubits null)

(define generate-U_omega (lambda (boolean-expression)
   (let ([base-qubits (+ 1 (get-base-qubits boolean-expression))] [temp-matrix '()]) ;
       base-qubits is equal to the # of qubits needed to implement the circuit before
       uncomputation
      (set! input-qubits (- base-qubits 1))
      (set! next-safe-qubit base-qubits)
      (set! total-qubits (+ base-qubits (get-extra-qubits boolean-expression) 2)) ; Two
          extra: one to save the answer before we uncompute, and another for the phase
          flipper
      (set! U_omega (identity-matrix (expt 2 total-qubits)))
      (set! uncomputer (identity-matrix (expt 2 total-qubits)))
      (eval boolean-expression ns)
      (set! U_omega (matrix* U_omega (G-nqubit-constructor (expt 2 total-qubits) (list (-
          total-qubits 3) (- total-qubits 2)) CNOT-gate))) ; Copy output of simulated
          classical circuit to the nth qubit
      (set! U_omega (matrix* U_omega uncomputer))
      (set! U_omega (matrix* U_omega
         (G-nqubit-constructor (expt 2 total-qubits) (list (- total-qubits 2) (-
             total-qubits 1)) CNOT-gate)
         (G-nqubit-constructor (expt 2 total-qubits) (list (- total-qubits 2) (-
             total-qubits 1)) controlled-Z-gate)
```

```
        (G-nqubit-constructor (expt 2 total-qubits) (list (- total-qubits 2) (-
            total-qubits 1)) CNOT-gate)
        U_omega))
    (void (write-file "./circuit-files/qcircuit.qasm" (string-append
        "\tqubit\tq" (string-join (map (lambda (num) (number->string num)) (range
            total-qubits)) ",0\n\tqubit\tq") ",0\n" ; Set up necessary qubits
        computer-strings-for-latex ; Add the compute string
        "\tcnot\tq" (number->string (- total-qubits 3)) ",q" (number->string (-
            total-qubits 2)) "\n" ; CNOT for the copy on to (n-1)th qubit
        uncomputer-strings-for-latex ; Add the uncompute string
        "\tcnot\tq" (number->string (- total-qubits 2)) ",q" (number->string (-
            total-qubits 1)) "\n" ; CNOT for the copy on to nth qubit
        "\tc-z\tq" (number->string (- total-qubits 2)) ",q" (number->string (-
            total-qubits 1)) "\n" ; controlled Z on (n-1)th and nth qubits
        "\tcnot\tq" (number->string (- total-qubits 2)) ",q" (number->string (-
            total-qubits 1)) "\n" ; CNOT for the uncopy on to nth qubit
        computer-strings-for-latex ; Now we do the whole first half again, starting with
            the compute string
        "\tcnot\tq" (number->string (- total-qubits 3)) ",q" (number->string (-
            total-qubits 2)) "\n" ; CNOT for the uncopy on to (n-1)th qubit
        uncomputer-strings-for-latex ; Add the uncompute string
    ))))))

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

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

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

    (set! U_omega input-U_omega)
    (display "The number of required qubits is ") (displayln qubits)
```

```
    (display "Number of operations required is ") (displayln (+ 1 steps))

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

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

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

References

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

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

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

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

[5] http://mathoverflow.net/questions/102779/how-much-does-a-quantum-oracle-to-find-a-needle-in-a-haystack-really-cost