2009

Implementing Dijkstra's pathfind using quantum algorithms

Shing Yung Lo

IMPLEMENTING DIJKSTRA'S PATHFIND USING QUANTUM ALGORITHMS

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Shing Yung Lo

June 2009

IMPLEMENTING DIJKSTRA'S PATHFIND USING QUANTUM ALGORITHMS

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Shing Yung Lo

June 2009

Approved by:

[redacted]

Ernesto Gomez, Chair, Department of Computer Science and Engineering

6/11/09
Date

[redacted]

Keith Evan Schubert

[redacted]

Haiyan Qiao

ABSTRACT

With the recent rise in interest over quantum computing, several quantum algorithms have gained prominence for their radical approaches for solving traditional problems.

This thesis explores the viability of implementing Grover's search (a quantum algorithm) to implement Dijkstra's algorithm. The quantum algorithm will be used to replace the find-minimum function. However, in order for this to be practical, the quantum algorithm would have to return a position. In addition, it would also have to be able to search for multiple answers within the search space. This problem is solved by extending Grover's search, by performing the oracle onto the values within the register that exists in the search space, then performing the inversion about mean on the entire register which contains both the value and position.

To use Grover's search, Dijkstra's algorithm had to be modified slightly. The minimum value is pre-calculated based on the size of the search array. This special case is available due to the fact that the simulation is using a 15x15 fixed grid.

The obtained results (99.9784%) are very close to the theoretical expectations (99.65857%). From these results, this thesis proves that it is possible to implement Dijkstra's algorithm using Quantum algorithms.

Possible future works may be pursued by implementing Dijkstra's algorithm using other quantum algorithms, or using this modified version of Grover's search for other classical algorithms.

ACKNOWLEDGEMENTS

It goes without saying that without my parents to support me, I wouldn't have been able to write this thesis. Lots of thanks to them firstly. Next, I would like to thank Dr Gomez and Dr Schubert for spending an insane amount of time with me to solve the many problems I had with this thesis. I would also like to thank Dr Qiao for agreeing to be part of the committee of my thesis.

Also, a TON of thanks and appreciation goes out to Dr Kolehmainen of the physics department. Without her help and support during quantum mechanics, I would never have made it this far. (I took QM while missing 8 prerequisites). I would also like to take this chance to thank Dr Karant for first encouraging me to do a thesis. Without him, this thesis would never have actually started.

I would like to take this chance to thank the teachers who hired me as their TA's: Dr Voigt, Dr Zemoudeh, Dr Murphy. Working with you helped fund this thesis, and taught me a lot about dealing and meeting with new people.

DEDICATION

For My Loved Ones.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

This thesis aims to explore the viability of implementing Dijkstra's pathfind using quantum algorithms(Grover's search) through a simulation. The simulation package used is libquantum(www.libquantum.de) written in C language.

First, a quick introduction to Dijkstra's pathfind algorithm. It outputs a shortest path tree from a graph search, solving the single-source shortest path problem for a graph with non negative edge path costs. In layman's terms, it finds the shortest path from point A to point B.

Next, with regards to the quantum algorithm: Grover's search is a probabilistic algorithm that performs a search in time $O(\sqrt[2]{n})$. This is a significant speedup over most unsorted searches out there that usually perform around O(n/2).

It is important to note that Dijkstra's algorithm could produce several routes that are equally short. As I later found out, this caused a huge amount of problems with regards to how the simulation is executed, and I have produced 3 working methods to implementing Grover's search into Dijkstra's algorithm.

1.1 Significance

A lot of funding and interest has been focused into quantum computing as of late, and with good reason. Should these quantum algorithms be exploited correctly, the performance could be in the order of magnitudes.

This thesis aims to prove the viability of incorporating a quantum algorithm into a more commonly used classical algorithm. Following this, gauge the performance of this new entity. In this case, the focus has been mainly on simulating a Grover's search that would actually be practical for use with Dijkstra's algorithm. Performance wise, it does exceed the generic unsorted array. However, it is still lacking when compared to a sorted array.

The final importance of this thesis is to prove the viability of using Grover's algorithm in a fully operational simulation. It will be designed to function as a fully working algorithm with no black box limitations whatsoever. Every single experiment or simulation to this date that has been published assumes a black box scenario, which severely limits a practical application of Grover's search.

1.2 Findings

Two significant findings were made:

- First, being able to return a position from Grover's search. This in itself is a huge step as there has not been one paper that has proven how to actually come about with the position of the answer.
- Secondly, being able to implement Grover's search in a non-black box simulation

that introduces multiple results into the search space.

Other findings:

- Odd but interesting behaviors of using only values and separating the multiple results from one another within the Grover's search.
- How to modify Dijkstra's algorithm to produce a predicted minimum value from the queue based on the queue size. Works only under several specific conditions.

2. LITERATURE REVIEW

2.1 Background: A Brief Review of Dijkstra's Algorithm

Dijkstra's algorithm was formulated by Dutch computer scientist Edsger Dijkstra in 1959. There are many different variations or approaches to implementing Dijkstra's algorithm. Hence, only the one used in this simulation will be elaborated.

This simulation uses a 15 by 15 grid, with 4 edges per vertex with the exception on the end of the grid.

Dijkstra's algorithm:

1. First, set a starting and end point.
2. Then perform a neighboring vertex search based on the edges, returning all valid vertexes into an unsorted array. Sum the connecting edges from the starting point.
3. After this, search for the shortest sum of edges in the array, and return the associated vertex.
4. If that vertex is the end point, terminate. If not, go back to step 2.

Note: The history of traversal will need to be stored for every single explored vertex until the end point is reached.

2.2 Background: A Brief Review of Quantum Computing

The reader is assumed to be a graduate from the computer science field, with knowledge of basic math and physics. The following review is intended to ONLY update the reader regarding the terminologies and concepts used for quantum computing. The reader would have to refer elsewhere for all other material related to computer science.

2.2.1 Qubits

Traditionally in computer science, a 0 or 1 is stored in a register. The definition of a register with say 8 bits is called an 8 bit register.

The same concept applies to quantum computing with the difference that both 0 and 1 are stored at the same time. This is referred to in physics as a superposition. The register with say 8 qubits, is referred to as an 8 qubit register.

2.2.2 Superposition

The idea that the qubit is able to assume both the value 0 and 1 simultaneously until it is measured, is the very foundation of quantum computing. With this superposition paradigm, many new types of algorithms can be conceived, which are fundamentally faster than their classical counterparts, performing in some cases several orders of magnitude faster.[14]

Traditionally in physics, a half-silvered mirror or a stern-gerlach experiment is used to demonstrate this superposition effect.[8] In quantum computing, a gate called the Hadamard gate is generally employed. Here is its matrix notation:

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

Fig. 2.1: The Hadamard Matrix.

In order to create a superposition of 0 and 1 in a register, the Hadamard gate is used. This gate is the fundamental building block in every quantum algorithm. As this concept is extremely important, further illustration is warranted:

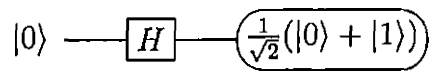

Fig. 2.2: A Simple Hadamard Example.

In the example above, the register is started off with a 0 value. When the Hadamard gate is performed, the register is put in a superposition of 0 and 1. During measurement, the result is normalized, and the quantum state collapses with a 50% probability chance that the answer will be 0 or 1.[16]

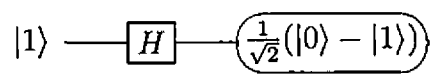

Fig. 2.3: Another Example of Using the Hadamard Gate.

In the example above, the register is started off with a 1 value. When the Hadamard gate is performed, the register is put in a superposition of 0 and 1. During measurement, there is a 50% chance that the value will be 0 or 1. *Note: The negative sign is cancelled out when the matrix is normalized.*

2.2.3 *Dirac Bra-ket Notation*

The Dirac Bra-ket notation is a commonly used notation in physics to describe the sum of an inner product from negative infinity, to infinity.[5] From the previous discussion, the terminology $|0>$ was used several times. This particular symbol referred to state 0, as the ket. $<0|$ is referred to as state 0, the bra.

An example of how these states would be used in a quantum circuit: what would be the probability of the initial state $<0|$ appearing to be in the end state $|0>$ with no outside interference? The answer is simply, 100%. Incidentally, the chance of the state $<0|$ appearing to be in the end state $|1>$ with no interference, is 0%.

This of itself is a simple idea, but becomes a lot more useful and complicated when gates are added as shown in the previous Figure 2.2 and 2.3.

2.2.4 *Grover's Algorithm*

Grover's algorithm is a search algorithm that has a time performance of $O(\sqrt{n})$. The requirement being you must know what you are searching for.

There are several methods of visualizing Grover's search. The most common is a geometric visualization. First, break the problem into two basis parts: the probability of the right answer(on the x axis) and then the probability of a wrong answer(on the y axis). [11]

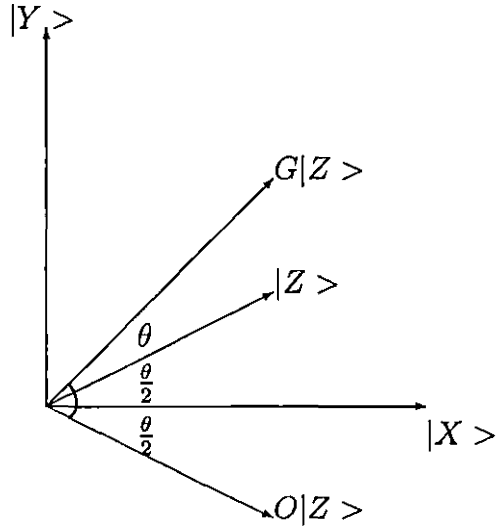

Fig. 2.4: A Geometric Visualization of Grover's Search.

The $G|Z>$ is the initial start state. It has a 50% chance of being the right or wrong answer. By performing a reflection on it with the G operator, it becomes the new state $O|Z>$. It is then reflected once more using the O operator, and it goes into the final state $|Z>$. Notice that the final answer is now closer to the X axis, which was stated above as the basis state for the correct answer.

The geometric visualization above can be rewritten into a formula as:

$$2|\psi><\psi| - I \tag{2.1}$$

It is worth noting that Grover himself described the algorithm as a wave phenomena. He talks about it as manipulating waves to cancel out the wrong answers, and to strengthen the probabilities of measuring the right one.[10].

Delving deeper, Grover's algorithm consists of two main parts: the oracle and the inversion about mean.

2.2.5 Quantum Oracles

There are several types of oracles commonly used in quantum computing. It is a common misconception that the circuit knows the answer. It doesn't, the oracle only recognizes the answer.

For example, an oracle circuit would receive several types of input. A question would be posed to that circuit, and the answer would simply be a yes or a no. Note that the design of the oracle would be completely based on the type of input, the amount of input, and the type of question that would be directed at the input.

For this simulation, the oracle being used creates a phase shift on the correct answers.

Here is a more formal definition, example and proof of the most basic oracle by Deutsch, also known as Deutsch's oracle.[20]

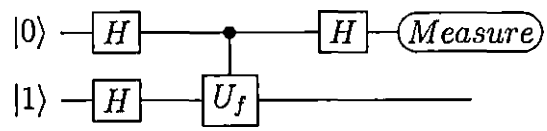

Fig. 2.5: **Deutsch's Oracle**

Where U_f is a controlled gate with the definition:

$$U_f|x > |y >= |x > |y +_2 f(x) > \tag{2.2}$$

Here is the detailed walkthrough of the oracle's circuit. The Hadamard transform initially converts $|0 > |1 >$ into:

$$\frac{1}{2}(|0 > +|1 >) \otimes (|0 > -|1 >) \tag{2.3}$$

The $\otimes$ means a tensor product, also commonly referred to as the outer product of a matrix. Now, apply the controlled-U_f gate. This results in:

$$U_f|x > \otimes(|0 > -|1 >) = |x > \otimes ((|0 > -|1 >) +_2 f(x)) \tag{2.4}$$

If the result of f(x)=0,

$$(|0 > -|1 >) +_2 f(x) = |0 > -|1 >= (-1)^0(|0 > -|1 >) = (-1)^{f(x)}(|0 > -|1 >) \tag{2.5}$$

Else if f(x)=1,

$$(|0 > -|1 >) +_2 f(x) = |1' > -|0 >= (-1)^1(|0 > -|1 >) = (-1)^{f(x)}(|0 > -|1 >) \tag{2.6}$$

From this, the summary can be made that the formula holds for all values of f(x). Hence:

$$U_f|x > \otimes(|0 > -|1 >) = (-1)^{f(x)}|x > \otimes ((|0 > -|1 >) +_2 f(x)) \tag{2.7}$$

Applying this result to the initial state yields:

$U_f \frac{1}{2}(|0> + |1>) \otimes (|0> - |1>)$

$$= \frac{1}{2}\left((-1)^{f(0)}|0> + (-1)^{f(x)}|1>\right) \otimes (|0> - |1>) \tag{2.8}$$

The final step requires applying the Hadamard gate once more to the first vector:

$\left((-1)^{f(0)}|0> + (-1)^{f(x)}|1>\right)$

$\frac{1}{2}\left((-1)^{f(0)}H|0> + (-1)^{f(1)}H|1>\right) \otimes (|0> - |1>)$

$= \frac{1}{2}\left((-1)^{f(0)}\frac{1}{(\sqrt{2})}(|0> + |1>) + (-1)^{f(1)}\frac{1}{(\sqrt{2})}(|0> - |1>)\right) \otimes (|0> - |1>)$

$$= \frac{1}{2}\left(|0>((-1)^{f(0)} + (-1)^{f(1)}) + |1>((-1)^{f(0)} - (-1)^{f(1)})\right) \otimes \frac{1}{\sqrt{2}}(|0> - |1>) \tag{2.9}$$

For this example, the purpose of the oracle is to make a comparison between a function that is constant or balanced. A function is constant if f(0)=f(1), and balanced if f(0)$\neq$ f(1), given a function f:{0,1} $\rightarrow$ {0,1}. Classically, it would require two evaluations. Using this quantum oracle, it is possible to reduce it down to only one evaluation, which is shown below.

The final step, observe:

If f(x) is constant: $(-1)^{f(0)} - (-1)^{f(1)} = 0$

The upper line vector would evaluate as:

$$\frac{1}{2}\left(|0>((-1)^{f(0)} + (-1)^{f(1)})\right) = \pm|0> \tag{2.10}$$

If f(x) is balanced: $(-1)^{f(0)} + (-1)^{f(1)} = 0$

The upper line vector would evaluate as:

$$\frac{1}{2}\left(|1>((-1)^{f(0)} - (-1)^{f(1)})\right) = \pm|1> \tag{2.11}$$

2.2.6 *Inversion About Mean*

This final important circuit of the simulation creates exactly as its name states, an inversion about the mean. The phase shift induces a negative sign into the complex probability, and when the inversion about mean is performed, it flips the negative sign and increases its probability significantly, while reducing the others to almost zero.[1]

As a result, all other answers except for the correct ones get reduced to almost zero.

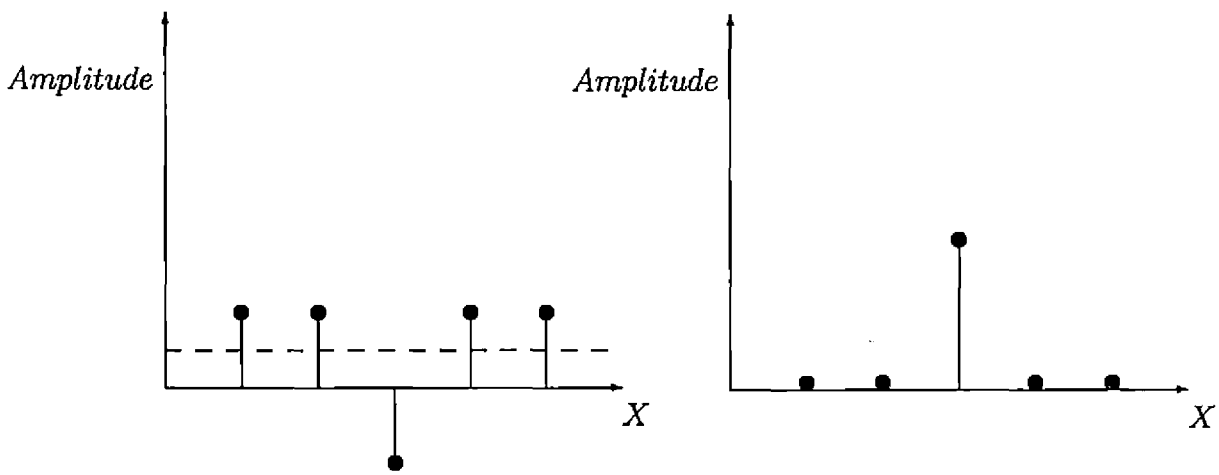

Fig. 2.6: Transition of The State After an Inversion About Mean.

As previously mentioned, the inversion can be described as:

$$2|\psi><\psi| - I \tag{2.12}$$

This is also referred to as the diffusion operator.

2.3 Miscellaneous Important Gates and Concepts

There are several more gates and concepts that are used within this simulation. This section intends to go into a brief overview of those ideas, but not delving deep into them. Weeks and months could be spent solely on explaining this section alone, hence the short summary of those ideas.

2.3.1 C-Not Gates

Controlled-not gates, flips the target bit if the control bit is set to 1.

Tab. 2.1: C-Not Table.

Before		After	
Control	Target	Control	Target
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

2.3.2 *Toffoli Gates*

An extension of the C-Not gate, it flips the target bit if both two control bits are set to 1.

Tab. 2.2: Toffoli Table.

Before			After		
Control 1	Control 2	Target	Control 1	Control 2	Target
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

2.3.3 *Quantum Fourier Transform Approach*

There are several ways to approach Grover's iteration. It is worth mentioning that an alternative way is to approach it using a quantum Fourier transform.

2.3.4 *Unitary*

Unitary simply means a transformation that doesn't change the unit of the vector after a transformation. Any type of unitary transformation is reversible. Furthermore,

this also means that there is no information loss, meaning that quantum computing produces theoretically, no energy dissipation. This is a huge change from classical computing whereby the computing is not unitary, and the hardware is constantly under the threat of energy dissipation(heat).

2.3.5 Ancillary Qubits

This term refers to the extra qubits that are required during the execution of a process. From the oracle example with 2.6, it requires one extra ancillary qubit. Also, from the oracle example with 4.2, it requires several ancillary qubits.

3. APPROACH AND METHODS OVERVIEW

As there are several ways to approach this problem. In this thesis, we shall discuss 3 methods.

Methods of implementing Grover's search:

1. Using only the position in the qubit, and assuming only knowledge of the correct position.(The black box approach)
2. Using only values in the qubit, and assuming only knowledge of the correct value.
3. Using both the values and position in the qubit, and assuming only knowledge of the correct value.

3.1 Using only the Position(The Black Box Approach)

This very simple method is explored due to the fact that almost every single example that has been shown in texts, takes on a form of the black box approach. In this case, the assumption is made that the knowledge of the correct position is known; it is not revealed how it is known; hence the name black box.

By taking the array of values from the Dijkstra's section, assume a black box that provides the correct position, then perform Grover's search on it. First, use the oracle to perform the phase shift on the correct answer.

Next, perform the inversion about mean. Based on the number of qubits, reiterate as needed.

The setup for the register is as the following:

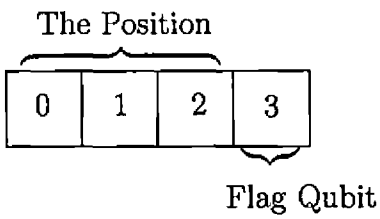

Fig. 3.1: Setup For a 3 Qubit Register Using Only the Position.

3.2 Using only the Values

This method stores only the values within the qubits. This is very similar to the first method used, with a huge problem: There are multiple possible results. Hence, there will be clashes when performing the Hadamard transform.

Similar to the second method above, use an algorithm that returns the minimum distance value based on the size of the array for a 15x15 grid. Then, by taking the array of values from Dijkstra's section, pass the array and the minimum distance into Grover's search. Firstly, the oracle will perform a phase shift on only the correct values.

Next, perform the inversion about mean. Reiterate as needed.

This method required modifications to many parts of the source code of the simulator. Even then, it does not provide a stable and complete answer. It did however, provide some interesting results. *Note: Most of the problems occurred when perform-*

ing the Hadamard transform on the values due to clashes.

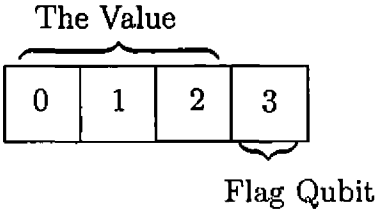

Fig. 3.2: Setup For a 3 Qubit Register Using Only the Position.

3.3 Using Both the Values and Position

This method stores both the values and position within the qubits. There are 3 sections to this structure: the first section stores the value; the second section stores the position; the final section holds the flag bit for phase shifting purposes used in the simulator.

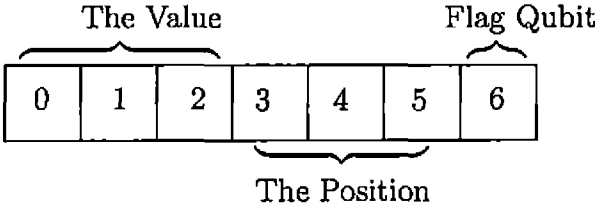

Fig. 3.3: Setup For a 3 Qubit Register Using Both Values And Position Assuming a 3 Qubit Position.

To implement: use an algorithm that returns the minimum distance value based on the size of the array for a 15x15 grid. Then, by taking the array of values from Dijkstra's section, pass the array and the minimum distance into Grover's search.

Firstly, the oracle will perform a phase shift on only the correct values(The first section) that exists within the array of values we are searching for.

Next, perform the inversion about mean on all 3 sections: the value, position, and the flag bit. Again, reiterate as needed.

4. SIMULATION OVERVIEW

The simulation program used is written by Björn Butscher and Hendrik Weimer.[2]. It is written in C, and simulates several quantum gates and circuits. Much of the Grover's iteration was referenced from their source example, but several significant alterations were made onto the Grover's example and the source code of the simulation itself.

Since Grover's algorithm is made up of two main sections: the oracle and the inversion about mean, it is appropriate to delve into the details of the implementation.

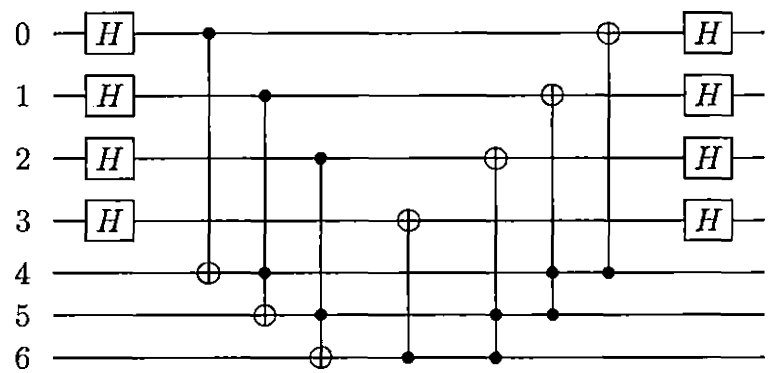

Fig. 4.1: A Quantum Oracle Circuit For a 3 Qubit Register.

Remember that the overall goal was to create a phase shift on the correct answer. To achieve this, only the answer that is being searched for will propogate the flags all the way down to the 6th qubit. When the 6th qubit is flipped, it activates the flip on

the 3rd qubit. This creates the phase shift as it switches the value with a negative sign.

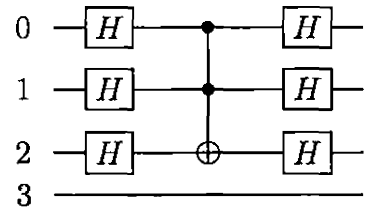

Fig. 4.2: An Inversion About Mean Circuit For a 3 Qubit Register.

The inversion about mean is activated. The flag bit 3 is ignored; this circuit inverts the amplitude of the most significant bit.

4.1 Implementation of the Quantum Register

The simulation libquantum provides the following structure[2]:

```
quantum_reg {
int width; /* number of qubits in the register */
int size; /* number of non-zero vectors */
int hashw; /* width of the hash array */
quantum_reg_node *node;
int *hash;
};
quantum_reg_node
{
COMPLEX_FLOAT amplitude;
MAX_UNSIGNED state;
};
```

4.2 Implementation of Important Quantum Gates

The quantum C-not gate is implemented as:

```
extern void quantum_cnot(int control, int target, quantum_reg *reg);
```

The Toffoli gate is implemented as:

```
quantum_toffoli(int control1, int control2, int target, quantum_reg *reg);
```

The sigma x, sigma y, sigma z gates are implemented as:

```
extern void quantum_sigma_x(int target, quantum_reg *reg);
```

```
extern void quantum_sigma_y(int target, quantum_reg *reg);
```

```
extern void quantum_sigma_z(int target, quantum_reg *reg);
```

These gates perform the pauli spin operations.

The quantum Hadamard gate is implemented as:

```
extern void quantum_Hadamard(int target, quantum_reg *reg);
```

4.3 Implementation of Measure

The following implementation performs a measurement on the entire quantum register. It returns the result of the measurement.

```
extern MAX_UNSIGNED quantum_measure(quantum_reg reg);
```

The next measurement used in the simulation performs a measurement on a single qubit. The width of the quantum register is reduced by one. It returns the result of the measurement, i.e. 0 or 1.

```
extern MAX_UNSIGNED quantum_measure(quantum_reg reg);
```

4.4 Initial Setup of the Simulation

The libquantum simulation receives an array of values from the Dijkstra's program. The array contains the values of the distance, and also the position associated with the value.

Tab. 4.1: Illustration of the Array Table.

Distance	Position
4	0
2	1
4	2
m	n

Following this, within the libquantum simulation itself, the size of the quantum register is determined based on the largest value in the array, and also based on the total positions in the array (for the aforementioned 3rd approach). The formula used is as follows: 1 to 4 - minimum value of 1, 5-20 - min of 2, 21-52 - min of 3, 53-100 - min of 4 and so forth until the 15th.

Then, the Hadamard gate is performed to initialize the states to equal probability. Following this, the quantum register states are then set to the values in the existing array. As a result, the register state will contain all the values from the array, and each value will have an equal probability to be measured.

Tab. 4.2: Example of the Initial Register State for 3 Qubits.

State	Initial Probability
0	0.125
1	0.125
2	0.125
3	0.125
4	0.125
5	0.125
6	0.125
7	0.125

The table above illustrates an example of the initial states after the Hadamard gates. The first column contains the states, and they all have the initial probability of 12.5% to be measured.

5. RESULTS

As mentioned before, this simulation will be running three types of implementation.

5.1 Results - Using only the Position

This is the starting ground for the thesis. By assuming a black box setup, the purpose of this method was to see if it was viable to modify Dijkstra's pathfind to use Grover. First, pass the array of numbers from Dijkstra's pathfind. Then, create the states with equal probabilities using Hadamard gates, and replace the values with those from the array. The output is shown as below for a 3 qubit register.

State: 0	Amplitude: 0.353553	Probability: 0.125000
State: 1	Amplitude: 0.353553	Probability: 0.125000
State: 2	Amplitude: 0.353553	Probability: 0.125000
State: 3	Amplitude: 0.353553	Probability: 0.125000
State: 4	Amplitude: 0.353553	Probability: 0.125000
State: 5	Amplitude: 0.353553	Probability: 0.125000
State: 6	Amplitude: 0.353553	Probability: 0.125000
State: 7	Amplitude: 0.353553	Probability: 0.125000

Following this, since a blackbox setup was assumed, the answer will be assumed

to be on position 5, which is state 5 in the above case. After running the oracle once:

State: 0	Amplitude: 0.353553	Probability: 0.125000
State: 1	Amplitude: 0.353553	Probability: 0.125000
State: 2	Amplitude: 0.353553	Probability: 0.125000
State: 3	Amplitude: 0.353553	Probability: 0.125000
State: 4	Amplitude: 0.353553	Probability: 0.125000
State: 6	Amplitude: 0.353553	Probability: 0.125000
State: 7	Amplitude: 0.353553	Probability: 0.125000
State: 5	Amplitude: -0.353553	Probability: 0.125000

State 5's amplitude has now been modified with a phase shift, a negative value in the complex plane. The probability when squared is still the same as it was before. Next, after implementing the inversion about mean.

State: 0	Amplitude: -0.176777	Probability: 0.031250
State: 1	Amplitude: -0.176777	Probability: 0.031250
State: 3	Amplitude: -0.176777	Probability: 0.031250
State: 5	Amplitude: -0.883883	Probability: 0.781250
State: 2	Amplitude: -0.176777	Probability: 0.031250
State: 4	Amplitude: -0.176777	Probability: 0.031250
State: 7	Amplitude: -0.176777	Probability: 0.031250
State: 6	Amplitude: -0.176777	Probability: 0.031250

After the first iteration, the probability of measuring the correct answer is 78.125%.

Based on the calculation:

$$\frac{\pi}{4} \times \sqrt{2^n} \tag{5.1}$$

Where n is the number of qubits used. In this case, the total iteration to perform is 2. The iteration is performed once more, and the result are as follows:

State: 0	Amplitude: -0.088388	Probability: 0.007813
State: 1	Amplitude: -0.088388	Probability: 0.007813
State: 3	Amplitude: -0.088388	Probability: 0.007813
State: 5	Amplitude: -0.972272	Probability: 0.945313
State: 2	Amplitude: -0.088388	Probability: 0.007813
State: 4	Amplitude: -0.088388	Probability: 0.007813
State: 7	Amplitude: -0.088388	Probability: 0.007813
State: 6	Amplitude: -0.088388	Probability: 0.007813

The final probability after the second iteration results in 94.5313% of measuring the correct answer. The accuracy of the measurement is dependent on the total qubits used, and will approach 100% as the number of qubits increases. The formula for measuring the accuracy is as follows:

$$\sin^2\left(\frac{2r+1}{2}\theta\right) \tag{5.2}$$

r being the total of Grover iterations, and θ is defined as:

$$\theta = 2arcsin \times \frac{1}{\sqrt{2^n}} \tag{5.3}$$

5.2 *Results - Using only the Values*

Initially, this section was investigated on the basis that there would be multiple result clashes within the simulation, hence it would be impossible to proceed with implementing both positions and values without first overcoming this clash hurdle regarding the values. Several changes to the source code had to be made to handle the value clashes.

This method follows the same steps as used previously, with the big difference being that there are now two equal values in the registers. One way to differentiate these two states would be to use a flag bit named "multiple" inside the array. From the initial setup state, there are two solutions(value 5) to the problem.

```
State: 0 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 1 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 2 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 3 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 4 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 5 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 5 Amplitude: 0.353553 Probability: 0.125000 multiples: 1
State: 6 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
```

Following this, the oracle is performed. The values/state 5 should have their phases flipped.

```
State: 0 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 1 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 2 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 3 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 4 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 6 Amplitude: 0.353553 Probability: 0.125000 multiples: 0
State: 5 Amplitude: -0.353553 Probability: 0.125000 multiples: 0
State: 5 Amplitude: -0.353553 Probability: 0.125000 multiples: 1
```

As we can observe from the results, the states 5 does have the amplitude inverted. So far, the modifications to the source code have produced the desired results. Now, proceed to the inversion about mean.

```
State: 0 Amplitude: -0.088388 Probability: 0.007812 multiples: 0
State: 1 Amplitude: -0.088388 Probability: 0.007812 multiples: 0
State: 3 Amplitude: -0.088388 Probability: 0.007812 multiples: 0
State: 3 Amplitude: 0.088388 Probability: 0.007813 multiples: 1
State: 5 Amplitude: -0.795495 Probability: 0.632812 multiples: 0
State: 5 Amplitude: -0.265165 Probability: 0.070313 multiples: 1
State: 7 Amplitude: -0.441942 Probability: 0.195312 multiples: 0
State: 0 Amplitude: 0.088388 Probability: 0.007813 multiples: 1
State: 2 Amplitude: -0.088388 Probability: 0.007812 multiples: 0
State: 1 Amplitude: 0.088388 Probability: 0.007813 multiples: 1
State: 2 Amplitude: 0.088388 Probability: 0.007813 multiples: 1
```

```
State: 4 Amplitude: -0.088388 Probability: 0.007812 multiples: 0
State: 7 Amplitude: 0.088388 Probability: 0.007813 multiples: 1
State: 4 Amplitude: 0.088388 Probability: 0.007813 multiples: 1
State: 6 Amplitude: -0.088388 Probability: 0.007812 multiples: 0
State: 6 Amplitude: 0.088388 Probability: 0.007813 multiples: 1
```

From the printout above, it shows that the value 5 has a chance of about 70% to be measured. It is worth noting that from several other tests performed, as the number of qubits is increased, the accuracy does indeed approach 100%. Also, only one inversion needs to be performed in this case.

Do keep in mind, practically, it does not seem possible(at this time) to be able to separate out one entangled state from another using the multiple flag bit that was shown above.

5.3 Results - Using both the Positions and Values

After several months spent on trying to separate the values and prevent the clashes, this final method was finally simulated. It turned out to be extraordinarily easy, and did not require any implementations from the previous section. There were no clashes, as every value was set unique with the addition of its unique position. Even though the oracle performs the phase shift on the duplicate values, the addition of the positions created a unique key for every single slot for the inversion about mean. With perfect hindsight looking back, it was foolish to have tried to prevent the clashes of the values. However, back then I was not familiar with the workings of the simulation,

and especially with how libquantum had implemented Grover's search. Most of this knowledge only came when i went into the source code and tried to prevent the clashes using only the values. C'est la vie.

As used previously, the setup of 3 qubits will be used again. However, this time the total qubits for the position will also equal 3 qubits, bringing the total qubits to 6. As a result, the list of results will be quite long. Keep in mind, $2^6 = 64$ results.

Again, firstly, initialize the states to equal probabilities, then replace the values and positions from the array.

```
State: 0 000 Amplitude: -0.125000 Probability: 0.015625
State: 1 001 Amplitude: -0.125000 Probability: 0.015625
State: 2 010 Amplitude: -0.125000 Probability: 0.015625
State: 3 011 Amplitude: -0.125000 Probability: 0.015625
State: 4 100 Amplitude: -0.125000 Probability: 0.015625
State: 5 101 Amplitude: -0.125000 Probability: 0.015625
State: 6 110 Amplitude: -0.125000 Probability: 0.015625
State: 7 111 Amplitude: -0.125000 Probability: 0.015625
State: 8 000 Amplitude: -0.125000 Probability: 0.015625
State: 9 001 Amplitude: -0.125000 Probability: 0.015625
State: 10 010 Amplitude: -0.125000 Probability: 0.015625
State: 11 011 Amplitude: -0.125000 Probability: 0.015625
State: 12 100 Amplitude: -0.125000 Probability: 0.015625
State: 13 101 Amplitude: -0.125000 Probability: 0.015625
```

```
State: 14 110 Amplitude: -0.125000 Probability: 0.015625
State: 15 111 Amplitude: -0.125000 Probability: 0.015625
State: 16 000 Amplitude: -0.125000 Probability: 0.015625
State: 17 001 Amplitude: -0.125000 Probability: 0.015625
State: 18 010 Amplitude: -0.125000 Probability: 0.015625
State: 19 011 Amplitude: -0.125000 Probability: 0.015625
State: 20 100 Amplitude: -0.125000 Probability: 0.015625
State: 21 101 Amplitude: -0.125000 Probability: 0.015625
State: 22 110 Amplitude: -0.125000 Probability: 0.015625
State: 23 111 Amplitude: -0.125000 Probability: 0.015625
State: 24 000 Amplitude: -0.125000 Probability: 0.015625
State: 25 001 Amplitude: -0.125000 Probability: 0.015625
State: 26 010 Amplitude: -0.125000 Probability: 0.015625
State: 27 011 Amplitude: -0.125000 Probability: 0.015625
State: 28 100 Amplitude: -0.125000 Probability: 0.015625
State: 29 101 Amplitude: -0.125000 Probability: 0.015625
State: 30 110 Amplitude: -0.125000 Probability: 0.015625
State: 31 111 Amplitude: -0.125000 Probability: 0.015625
State: 32 000 Amplitude: -0.125000 Probability: 0.015625
State: 33 001 Amplitude: -0.125000 Probability: 0.015625
State: 34 010 Amplitude: -0.125000 Probability: 0.015625
State: 35 011 Amplitude: -0.125000 Probability: 0.015625
State: 36 100 Amplitude: -0.125000 Probability: 0.015625
```

```
State: 37 101 Amplitude: -0.125000 Probability: 0.015625
State: 38 110 Amplitude: -0.125000 Probability: 0.015625
State: 39 111 Amplitude: -0.125000 Probability: 0.015625
State: 40 000 Amplitude: -0.125000 Probability: 0.015625
State: 41 001 Amplitude: -0.125000 Probability: 0.015625
State: 42 010 Amplitude: -0.125000 Probability: 0.015625
State: 43 011 Amplitude: -0.125000 Probability: 0.015625
State: 44 100 Amplitude: -0.125000 Probability: 0.015625
State: 45 101 Amplitude: -0.125000 Probability: 0.015625
State: 46 110 Amplitude: -0.125000 Probability: 0.015625
State: 47 111 Amplitude: -0.125000 Probability: 0.015625
State: 48 000 Amplitude: -0.125000 Probability: 0.015625
State: 49 001 Amplitude: -0.125000 Probability: 0.015625
State: 50 010 Amplitude: -0.125000 Probability: 0.015625
State: 51 011 Amplitude: -0.125000 Probability: 0.015625
State: 52 100 Amplitude: -0.125000 Probability: 0.015625
State: 53 101 Amplitude: -0.125000 Probability: 0.015625
State: 54 110 Amplitude: -0.125000 Probability: 0.015625
State: 55 111 Amplitude: -0.125000 Probability: 0.015625
State: 56 000 Amplitude: -0.125000 Probability: 0.015625
State: 57 001 Amplitude: -0.125000 Probability: 0.015625
State: 58 010 Amplitude: -0.125000 Probability: 0.015625
State: 59 011 Amplitude: -0.125000 Probability: 0.015625
```

```
State: 60 100 Amplitude: -0.125000 Probability: 0.015625
State: 61 101 Amplitude: -0.125000 Probability: 0.015625
State: 62 110 Amplitude: -0.125000 Probability: 0.015625
State: 63 111 Amplitude: -0.125000 Probability: 0.015625
```

In addition to the previous results, there is now a new column containing binary bits as the printout. This holds the value, while the state holds the position. In this particular case, there are 8 possible answers. The next step is the oracle, and the result is as follows:

```
State: 0 000 Amplitude: -0.125000 Probability: 0.015625
State: 1 001 Amplitude: -0.125000 Probability: 0.015625
State: 2 010 Amplitude: -0.125000 Probability: 0.015625
State: 3 011 Amplitude: -0.125000 Probability: 0.015625
State: 4 100 Amplitude: -0.125000 Probability: 0.015625
State: 6 110 Amplitude: -0.125000 Probability: 0.015625
State: 7 111 Amplitude: -0.125000 Probability: 0.015625
State: 8 000 Amplitude: -0.125000 Probability: 0.015625
State: 9 001 Amplitude: -0.125000 Probability: 0.015625
State: 10 010 Amplitude: -0.125000 Probability: 0.015625
State: 11 011 Amplitude: -0.125000 Probability: 0.015625
State: 12 100 Amplitude: -0.125000 Probability: 0.015625
State: 14 110 Amplitude: -0.125000 Probability: 0.015625
```

State: 15 111 Amplitude: -0.125000 Probability: 0.015625

State: 16 000 Amplitude: -0.125000 Probability: 0.015625

State: 17 001 Amplitude: -0.125000 Probability: 0.015625

State: 18 010 Amplitude: -0.125000 Probability: 0.015625

State: 19 011 Amplitude: -0.125000 Probability: 0.015625

State: 20 100 Amplitude: -0.125000 Probability: 0.015625

State: 22 110 Amplitude: -0.125000 Probability: 0.015625

State: 23 111 Amplitude: -0.125000 Probability: 0.015625

State: 24 000 Amplitude: -0.125000 Probability: 0.015625

State: 25 001 Amplitude: -0.125000 Probability: 0.015625

State: 26 010 Amplitude: -0.125000 Probability: 0.015625

State: 27 011 Amplitude: -0.125000 Probability: 0.015625

State: 28 100 Amplitude: -0.125000 Probability: 0.015625

State: 30 110 Amplitude: -0.125000 Probability: 0.015625

State: 31 111 Amplitude: -0.125000 Probability: 0.015625

State: 32 000 Amplitude: -0.125000 Probability: 0.015625

State: 33 001 Amplitude: -0.125000 Probability: 0.015625

State: 34 010 Amplitude: -0.125000 Probability: 0.015625

State: 35 011 Amplitude: -0.125000 Probability: 0.015625

State: 36 100 Amplitude: -0.125000 Probability: 0.015625

State: 38 110 Amplitude: -0.125000 Probability: 0.015625

State: 39 111 Amplitude: -0.125000 Probability: 0.015625

```
State: 40 000 Amplitude: -0.125000 Probability: 0.015625
State: 41 001 Amplitude: -0.125000 Probability: 0.015625
State: 42 010 Amplitude: -0.125000 Probability: 0.015625
State: 43 011 Amplitude: -0.125000 Probability: 0.015625
State: 44 100 Amplitude: -0.125000 Probability: 0.015625
State: 46 110 Amplitude: -0.125000 Probability: 0.015625
State: 47 111 Amplitude: -0.125000 Probability: 0.015625
State: 48 000 Amplitude: -0.125000 Probability: 0.015625
State: 49 001 Amplitude: -0.125000 Probability: 0.015625
State: 50 010 Amplitude: -0.125000 Probability: 0.015625
State: 51 011 Amplitude: -0.125000 Probability: 0.015625
State: 52 100 Amplitude: -0.125000 Probability: 0.015625
State: 54 110 Amplitude: -0.125000 Probability: 0.015625
State: 55 111 Amplitude: -0.125000 Probability: 0.015625
State: 56 000 Amplitude: -0.125000 Probability: 0.015625
State: 57 001 Amplitude: -0.125000 Probability: 0.015625
State: 58 010 Amplitude: -0.125000 Probability: 0.015625
State: 59 011 Amplitude: -0.125000 Probability: 0.015625
State: 60 100 Amplitude: -0.125000 Probability: 0.015625
State: 62 110 Amplitude: -0.125000 Probability: 0.015625
State: 63 111 Amplitude: -0.125000 Probability: 0.015625
State: 5 101 Amplitude: 0.125000 Probability: 0.015625
```

```
State: 13 101 Amplitude: 0.125000 Probability: 0.015625
State: 21 101 Amplitude: 0.125000 Probability: 0.015625
State: 29 101 Amplitude: 0.125000 Probability: 0.015625
State: 37 101 Amplitude: 0.125000 Probability: 0.015625
State: 45 101 Amplitude: 0.125000 Probability: 0.015625
State: 53 101 Amplitude: 0.125000 Probability: 0.015625
State: 61 101 Amplitude: 0.125000 Probability: 0.015625
```

The oracle is only performed on the values (not the position), and the results do show that only the ones with the correct values(101) have their phases flipped. Next, the inversion about mean is performed.

```
State: 5 101 Amplitude: 0.312500 Probability: 0.097656
State: 1 001 Amplitude: 0.062500 Probability: 0.003906
State: 3 011 Amplitude: 0.062500 Probability: 0.003906
State: 0 000 Amplitude: 0.062500 Probability: 0.003906
State: 2 010 Amplitude: 0.062500 Probability: 0.003906
State: 7 111 Amplitude: 0.062500 Probability: 0.003906
State: 4 100 Amplitude: 0.062500 Probability: 0.003906
State: 6 110 Amplitude: 0.062500 Probability: 0.003906
State: 13 101 Amplitude: 0.312500 Probability: 0.097656
State: 9 001 Amplitude: 0.062500 Probability: 0.003906
State: 11 011 Amplitude: 0.062500 Probability: 0.003906
State: 8 000 Amplitude: 0.062500 Probability: 0.003906
```

State: 10 010 Amplitude: 0.062500 Probability: 0.003906

State: 15 111 Amplitude: 0.062500 Probability: 0.003906

State: 12 100 Amplitude: 0.062500 Probability: 0.003906

State: 14 110 Amplitude: 0.062500 Probability: 0.003906

State: 21 101 Amplitude: 0.312500 Probability: 0.097656

State: 17 001 Amplitude: 0.062500 Probability: 0.003906

State: 19 011 Amplitude: 0.062500 Probability: 0.003906

State: 16 000 Amplitude: 0.062500 Probability: 0.003906

State: 18 010 Amplitude: 0.062500 Probability: 0.003906

State: 23 111 Amplitude: 0.062500 Probability: 0.003906

State: 20 100 Amplitude: 0.062500 Probability: 0.003906

State: 22 110 Amplitude: 0.062500 Probability: 0.003906

State: 29 101 Amplitude: 0.312500 Probability: 0.097656

State: 25 001 Amplitude: 0.062500 Probability: 0.003906

State: 27 011 Amplitude: 0.062500 Probability: 0.003906

State: 24 000 Amplitude: 0.062500 Probability: 0.003906

State: 26 010 Amplitude: 0.062500 Probability: 0.003906

State: 31 111 Amplitude: 0.062500 Probability: 0.003906

State: 28 100 Amplitude: 0.062500 Probability: 0.003906

State: 30 110 Amplitude: 0.062500 Probability: 0.003906

State: 37 101 Amplitude: 0.312500 Probability: 0.097656

State: 33 001 Amplitude: 0.062500 Probability: 0.003906

State: 35 011 Amplitude: 0.062500 Probability: 0.003906

```
State: 32 000 Amplitude: 0.062500 Probability: 0.003906
State: 34 010 Amplitude: 0.062500 Probability: 0.003906
State: 39 111 Amplitude: 0.062500 Probability: 0.003906
State: 36 100 Amplitude: 0.062500 Probability: 0.003906
State: 38 110 Amplitude: 0.062500 Probability: 0.003906
State: 45 101 Amplitude: 0.312500 Probability: 0.097656
State: 41 001 Amplitude: 0.062500 Probability: 0.003906
State: 43 011 Amplitude: 0.062500 Probability: 0.003906
State: 40 000 Amplitude: 0.062500 Probability: 0.003906
State: 42 010 Amplitude: 0.062500 Probability: 0.003906
State: 47 111 Amplitude: 0.062500 Probability: 0.003906
State: 44 100 Amplitude: 0.062500 Probability: 0.003906
State: 46 110 Amplitude: 0.062500 Probability: 0.003906
State: 53 101 Amplitude: 0.312500 Probability: 0.097656
State: 49 001 Amplitude: 0.062500 Probability: 0.003906
State: 51 011 Amplitude: 0.062500 Probability: 0.003906
State: 48 000 Amplitude: 0.062500 Probability: 0.003906
State: 50 010 Amplitude: 0.062500 Probability: 0.003906
State: 55 111 Amplitude: 0.062500 Probability: 0.003906
State: 52 100 Amplitude: 0.062500 Probability: 0.003906
State: 54 110 Amplitude: 0.062500 Probability: 0.003906
State: 61 101 Amplitude: 0.312500 Probability: 0.097656
```

```
State: 57 001 Amplitude: 0.062500 Probability: 0.003906
State: 59 011 Amplitude: 0.062500 Probability: 0.003906
State: 56 000 Amplitude: 0.062500 Probability: 0.003906
State: 58 010 Amplitude: 0.062500 Probability: 0.003906
State: 63 111 Amplitude: 0.062500 Probability: 0.003906
State: 60 100 Amplitude: 0.062500 Probability: 0.003906
State: 62 110 Amplitude: 0.062500 Probability: 0.003906
```

As we can observe, the states/positions with the value 5 have a probability of 0.097656. With 8 duplicates, the total probability of measuring the correct answer is 78.1248% after the first iteration. Using the Grover's iteration formula that was shown previously, the total iteration required is 6 times. Here is the final result after 6 iterations.

```
State: 5 101 Amplitude: 0.353516 Probability: 0.124973
State: 1 001 Amplitude: 0.001953 Probability: 0.000004
State: 3 011 Amplitude: 0.001953 Probability: 0.000004
State: 0 000 Amplitude: 0.001953 Probability: 0.000004
State: 2 010 Amplitude: 0.001953 Probability: 0.000004
State: 7 111 Amplitude: 0.001953 Probability: 0.000004
State: 4 100 Amplitude: 0.001953 Probability: 0.000004
State: 6 110 Amplitude: 0.001953 Probability: 0.000004
State: 13 101 Amplitude: 0.353516 Probability: 0.124973
State: 9 001 Amplitude: 0.001953 Probability: 0.000004
```

State: 11 011 Amplitude: 0.001953 Probability: 0.000004

State: 8 000 Amplitude: 0.001953 Probability: 0.000004

State: 10 010 Amplitude: 0.001953 Probability: 0.000004

State: 15 111 Amplitude: 0.001953 Probability: 0.000004

State: 12 100 Amplitude: 0.001953 Probability: 0.000004

State: 14 110 Amplitude: 0.001953 Probability: 0.000004

State: 21 101 Amplitude: 0.353516 Probability: 0.124973

State: 17 001 Amplitude: 0.001953 Probability: 0.000004

State: 19 011 Amplitude: 0.001953 Probability: 0.000004

State: 16 000 Amplitude: 0.001953 Probability: 0.000004

State: 18 010 Amplitude: 0.001953 Probability: 0.000004

State: 23 111 Amplitude: 0.001953 Probability: 0.000004

State: 20 100 Amplitude: 0.001953 Probability: 0.000004

State: 22 110 Amplitude: 0.001953 Probability: 0.000004

State: 29 101 Amplitude: 0.353516 Probability: 0.124973

State: 25 001 Amplitude: 0.001953 Probability: 0.000004

State: 27 011 Amplitude: 0.001953 Probability: 0.000004

State: 24 000 Amplitude: 0.001953 Probability: 0.000004

State: 26 010 Amplitude: 0.001953 Probability: 0.000004

State: 31 111 Amplitude: 0.001953 Probability: 0.000004

State: 28 100 Amplitude: 0.001953 Probability: 0.000004

State: 30 110 Amplitude: 0.001953 Probability: 0.000004

```
State: 37 101 Amplitude: 0.353516 Probability: 0.124973
State: 33 001 Amplitude: 0.001953 Probability: 0.000004
State: 35 011 Amplitude: 0.001953 Probability: 0.000004
State: 32 000 Amplitude: 0.001953 Probability: 0.000004
State: 34 010 Amplitude: 0.001953 Probability: 0.000004
State: 39 111 Amplitude: 0.001953 Probability: 0.000004
State: 36 100 Amplitude: 0.001953 Probability: 0.000004
State: 38 110 Amplitude: 0.001953 Probability: 0.000004
State: 45 101 Amplitude: 0.353516 Probability: 0.124973
State: 41 001 Amplitude: 0.001953 Probability: 0.000004
State: 43 011 Amplitude: 0.001953 Probability: 0.000004
State: 40 000 Amplitude: 0.001953 Probability: 0.000004
State: 42 010 Amplitude: 0.001953 Probability: 0.000004
State: 47 111 Amplitude: 0.001953 Probability: 0.000004
State: 44 100 Amplitude: 0.001953 Probability: 0.000004
State: 46 110 Amplitude: 0.001953 Probability: 0.000004
State: 53 101 Amplitude: 0.353516 Probability: 0.124973
State: 49 001 Amplitude: 0.001953 Probability: 0.000004
State: 51 011 Amplitude: 0.001953 Probability: 0.000004
State: 48 000 Amplitude: 0.001953 Probability: 0.000004
State: 50 010 Amplitude: 0.001953 Probability: 0.000004
State: 55 111 Amplitude: 0.001953 Probability: 0.000004
```

```
State: 52 100 Amplitude: 0.001953 Probability: 0.000004
State: 54 110 Amplitude: 0.001953 Probability: 0.000004
State: 61 101 Amplitude: 0.353516 Probability: 0.124973
State: 57 001 Amplitude: 0.001953 Probability: 0.000004
State: 59 011 Amplitude: 0.001953 Probability: 0.000004
State: 56 000 Amplitude: 0.001953 Probability: 0.000004
State: 58 010 Amplitude: 0.001953 Probability: 0.000004
State: 63 111 Amplitude: 0.001953 Probability: 0.000004
State: 60 100 Amplitude: 0.001953 Probability: 0.000004
State: 62 110 Amplitude: 0.001953 Probability: 0.000004
```

From the final state of the system, the states/positions with the value 5 have a separate probability of 0.124973. With 8 duplicates, the total probability of measuring the correct answer is 99.9784%.

To verify this result, from the previously given equation, the total iterations used is 6, and the θ is:

$$2 \times arcsin(0.125) = 0.2506557 \tag{5.4}$$

Therefore, the total probability is:

$$sin^2\left(\frac{13}{2} \times 0.2506557\right) = 0.9965857 \tag{5.5}$$

Which is very close to the experiment's result.

5.4 Time Efficiency

After proving that it is possible to implement Dijkstra's pathfind using a quantum algorithm, the next step is to compare the time efficiency. Dijkstra's pathfind is originally:

$$O\left(V^2 + E\right) \tag{5.6}$$

V being the vertices, E being the connecting edges. Going into more details:

$$O(V \times (\textit{Time to extract minimum value from the array}) + E \times (\textit{Time to decrease key in array}))$$

Using a basic unsorted array, the time to search through the entire array is V, hence, V^2. However, this simulation had several special boundaries: a fixed space of 15x15 grid; each vertex has 4 edges. It is possible to use a special algorithm that precalculates the minimum distance needed based on the array size. A search for one value through an unsorted array always averages out to $\frac{V}{2}$. With that, the array search is reduced to:

$$O\left(\frac{V^2}{2} + E\right) \tag{5.7}$$

Now, from Grover's algorithm:

$$O\left(\sqrt{V}\right) \tag{5.8}$$

Where V is the number of vertex within the array. Hence, the new resultant algorithm should theoretically be:

$$O\left(V \times \sqrt{V} + E\right) \tag{5.9}$$

This is significantly faster than the classical method of searching. It is however, slower than a Fibonacci heap implementation, which has an overall speed of:

$$O\left(V \times \log(V) + E\right) \tag{5.10}$$

5.5 Additional Notes

With the setup shown from the previous section, using an oracle that performs an inversion on only the values would result in a possible equal measurement with the value in every possible location. (Due to the superposition effect) Hence, to actually implement this in the simulation, the oracle would also have to check to ensure that the position bit exists in the array. To emphasize again, the oracle will need to check first, the search value. Next, check if the following bits also exist through a index reference to the position in the array.

5.6 The Running Simulation

Below is a screenshot of the running program and how it looks graphically.

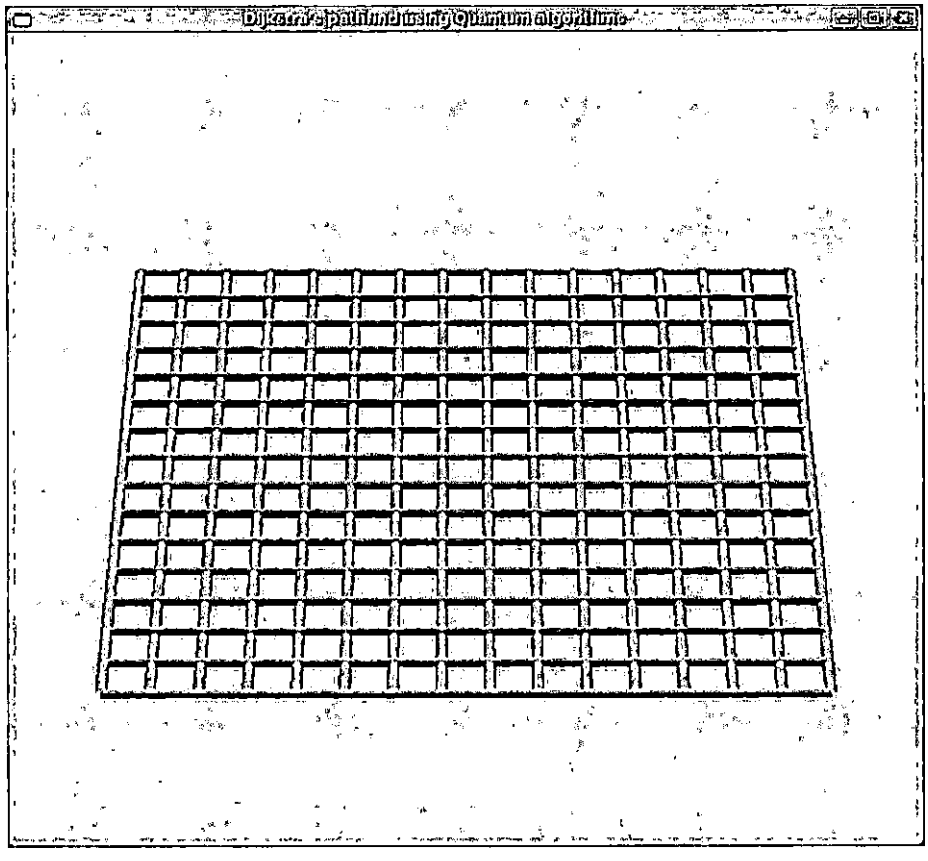

Fig. 5.1: Screen Shot of the Running Program

Following this, the program will also display the results from both the classical algorithm and the quantum algorithm.

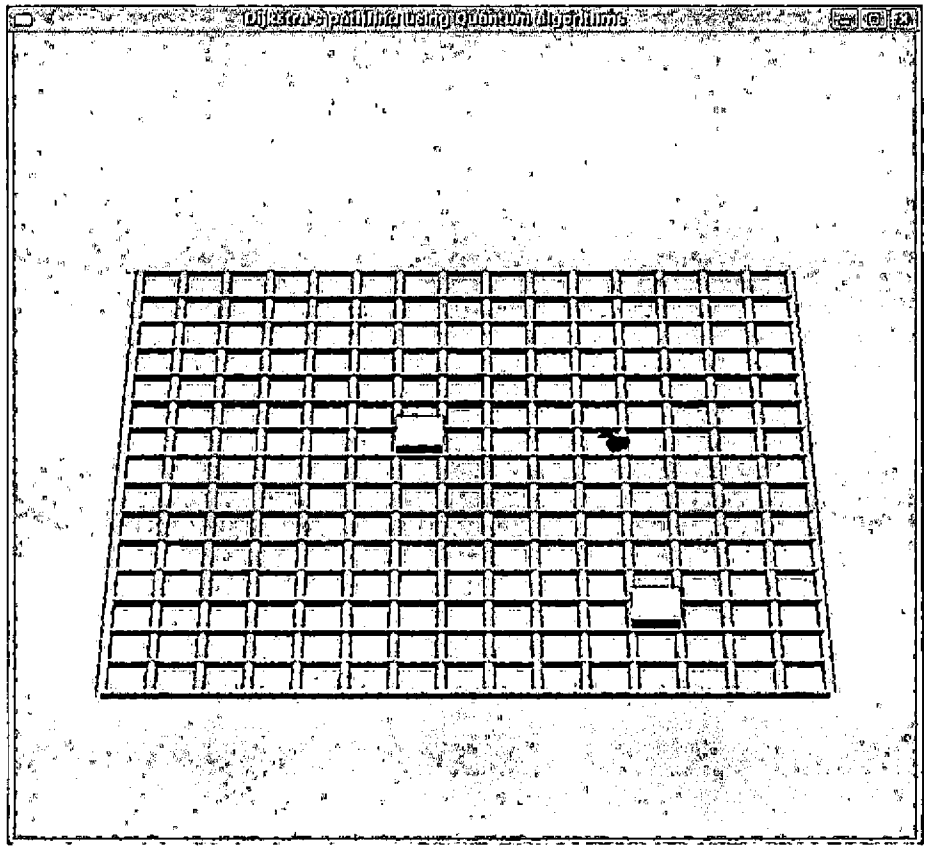

Fig. 5.2: Screen Shot Continued

6. CONCLUSION

6.1 Summary of Findings

The purpose of this thesis is to determine if it is possible to implement Dijkstra's pathfind using quantum algorithms. To achieve a fully functional Grover's search without a black box scenario, the simulation was done in 3 steps. First, by assuming a black box scenario to create a prototype of the full simulation. Secondly, by using only the values to see if a clash of values could be prevented. This then leads to the third and final step, using both the values and position inside Grover's search.

The results from section 5.3 shows that the simulation did indeed meet our theoretical expectations of 99.65857%, and that this implementation of Grover's search returned a meaningful result back to the calling program.

To summarize the process: perform the oracle on the values of the register, then perform the inversion about mean on the entire register which contains both the values and position.

6.2 Suggestions for Future Work

In this thesis, only Grover's search was thoroughly explored. It is definitely worth investigating to see if other Quantum algorithms could be used to implement Dijkstra's algorithm. Other possible methods might include solving Dijkstra's algorithm

using a permutation approach, which could in turn lead to a new approach in solving the Travelling Salesman's problem.

Other areas worth expanding upon would include transforming Dijkstra's algorithm(or finding special cases of it) into a hidden subgroup problem, and looking for an approach via: Period-finding, Order-finding, Order of a permutation, and the Hidden linear function approach.

APPENDIX A

SOURCE CODE

A.1 First Method of Implementation

The following is the code for the first implementation of Grover's algorithm assuming a black box scenario.

```
/* grover.c: Implementation of Grover's search algorithm

   Copyright 2003 Bjoern Butscher, Hendrik Weimer

   This file is part of libquantum

   libquantum is free software; you can redistribute it and/
      or modify
   it under the terms of the GNU General Public License as
      published
   by the Free Software Foundation; either version 3 of the
      License,
   or (at your option) any later version.

   libquantum is distributed in the hope that it will be
      useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
      the GNU
   General Public License for more details.
```

```
   You should have received a copy of the GNU General Public
      License
   along with libquantum; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
   MA 02110-1301, USA

*/
//Modified extensively by sean for his thesis.

#include <quantum.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <complex.h>


#ifdef M_PI
#define pi M_PI
#else
#define pi 3.141592654
#endif
```

```
void oracle(int state, quantum_reg *reg)
{
  int i;

  for(i=0;i<reg->width;i++)
    {
      if(!(state & (1 << i)))
  {
    quantum_sigma_x(i, reg);
  }
    }

  quantum_toffoli(0, 1, reg->width+1, reg);

  for(i=1;i<reg->width;i++)
    {
      quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
    }

  quantum_cnot(reg->width+i, reg->width, reg);

  for(i=reg->width-1;i>0;i--)
```

```
    {
      quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
    }

  quantum_toffoli(0, 1, reg->width+1, reg);

  for(i=0;i<reg->width;i++)
    {
      if(!(state & (1 << i)))
  quantum_sigma_x(i, reg);
    }

}

void inversion(quantum_reg *reg)
{
  int i;

  for(i=0;i<reg->width;i++)
    quantum_sigma_x(i, reg);

  quantum_hadamard(reg->width-1, reg);
```

```
if(reg->width==3)
  quantum_toffoli(0, 1, 2, reg);

else
  {
    quantum_toffoli(0, 1, reg->width+1, reg);

    for(i=1;i<reg->width-1;i++)
{
  quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
}

    quantum_cnot(reg->width+i, reg->width-1, reg);

    for(i=reg->width-2;i>0;i--)
{
  quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
}

    quantum_toffoli(0, 1, reg->width+1, reg);
  }

quantum_hadamard(reg->width-1, reg);
```

```
  for(i=0;i<reg->width;i++)
    quantum_sigma_x(i, reg);
}


void grover(int target, quantum_reg *reg)
{
  int i;

  oracle(target, reg);

  for(i=0;i<reg->width;i++)
    quantum_hadamard(i, reg);

  inversion(reg);

  for(i=0;i<reg->width;i++)
    quantum_hadamard(i, reg);

}

int main(int argc, char **argv)
```

```
{
  quantum_reg reg;
  int i, N, width=0;

  srand(time(0));

  if(argc==1)
    {
      printf("Usage: grover [number] [[qubits]]\n\n");
      return 3;
    }

  N=atoi(argv[1]);

  if(argc > 2)
    width = atoi(argv[2]);

  if(width < quantum_getwidth(N+1))  //For the setup.
  //In case the number we are searching for is bigger than
     the register width.
    width = quantum_getwidth(N+1);
  printf ("The width is now: %d\n",width);
  reg = quantum_new_qureg(0, width);
```

```
quantum_sigma_x(reg.width, &reg);

for(i=0;i<reg.width;i++)
  quantum_hadamard(i, &reg);

quantum_hadamard(reg.width, &reg);

printf("Iterating %i times\n", (int) (pi/4*sqrt(1<<reg.
   width)));

for(i=1; i<=pi/4*sqrt((1 << reg.width)); i++)
  {
    printf("Iteration #%i\n", i);
    grover(N, &reg);
  }

quantum_hadamard(reg.width, &reg);

reg.width++;

quantum_bmeasure(reg.width-1, &reg);
```

```
  for(i=0; i<reg.size; i++)
    {
    //  if(reg.node[i].state == N)
  printf("State: %d  ",reg.node[i].state);
  printf("  Amplitude: %f\n",creal(reg.node[i].amplitude));
  printf("  Probability: %f\n",creal(reg.node[i].amplitude)
                                  *creal(reg.node[i].amplitude)
                                    );
  //printf("\nFound %i with a probability of %f\n\n",
                                                reg.node[i].state
                                                    ,
    //      quantum_prob(reg.node[i].amplitude));
//  printf("\nFound %i with a probability of %f\n\n", N,
//          quantum_prob(reg.node[i].amplitude));
    }

  quantum_delete_qureg(&reg);

  return 0;
}
```

The following is the code for using the values only.

```
/* grover.c: Implementation of Grover's search algorithm

   Copyright 2003 Bjoern Butscher, Hendrik Weimer

   This file is part of libquantum

   libquantum is free software; you can redistribute it and/
      or modify
   it under the terms of the GNU General Public License as
      published
   by the Free Software Foundation; either version 3 of the
      License,
   or (at your option) any later version.

   libquantum is distributed in the hope that it will be
      useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
      the GNU
   General Public License for more details.
```

```
   You should have received a copy of the GNU General Public
      License
   along with libquantum; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
   MA 02110-1301, USA

*/
//Modified by Sean extensively for his thesis.
#include <quantum.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <complex.h>

#ifdef M_PI
#define pi M_PI
#else
#define pi 3.141592654
#endif


void quantum_multiple_once(quantum_reg *reg)
```

```
{
  //first reset all multiple values.
  //printf ("size: %i\n",reg->size);
  int i,j;
  for (i=0;i<reg->size;i++)
  {
      reg->node[i].multiples=0;
  }


  //next update the multiple field based on reccurence
  for (i=0;i<reg->size;i++)
  {
    for (j=i+1;j<reg->size;j++)
    {
      if (reg->node[i].state==reg->node[j].state)
      {
        reg->node[j].multiples++;
      }
    }
  }

}
```

```
void oracle(int state, quantum_reg *reg)
{
  int i;

  for(i=0;i<reg->width;i++)
    {
      if(!(state & (1 << i)))
  {
    quantum_sigma_x(i, reg);
  }
    }

  quantum_toffoli(0, 1, reg->width+1, reg);

  for(i=1;i<reg->width;i++)
    {
      quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
    }

  quantum_cnot(reg->width+i, reg->width, reg);

  for(i=reg->width-1;i>0;i--)
```

```
    {
      quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
    }

  quantum_toffoli(0, 1, reg->width+1, reg);

  for(i=0;i<reg->width;i++)
    {
      if(!(state & (1 << i)))
  quantum_sigma_x(i, reg);
    }

}

void inversion(quantum_reg *reg)
{
  int i;

  for(i=0;i<reg->width;i++)
    quantum_sigma_x(i, reg);

  quantum_hadamard(reg->width-1, reg);
```

```
if(reg->width==3)
  quantum_toffoli(0, 1, 2, reg);

else
  {
    quantum_toffoli(0, 1, reg->width+1, reg);

    for(i=1;i<reg->width-1;i++)
{
  quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
}

    quantum_cnot(reg->width+i, reg->width-1, reg);

    for(i=reg->width-2;i>0;i--)
{
  quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
}

    quantum_toffoli(0, 1, reg->width+1, reg);
  }

quantum_hadamard(reg->width-1, reg);
```

```
  for(i=0;i<reg->width;i++)
    quantum_sigma_x(i, reg);
}


void grover(int target, quantum_reg *reg)
{
  int i;

  //oracle(target, reg);
  oracle(5,reg);

  for(i=0;i<reg->width;i++)
    quantum_hadamard(i, reg);

  inversion(reg);

  for(i=0;i<reg->width;i++)
    quantum_hadamard(i, reg);

}
```

```
int main(int argc, char **argv)
{
  quantum_reg reg;
  int i, N, width=0;

  srand(time(0));

  if(argc==1)
    {
      printf("Usage: grover [number] [[qubits]]\n\n");
      return 3;
    }

  N=atoi(argv[1]);

  if(argc > 2)
    width = atoi(argv[2]);

  if(width < quantum_getwidth(N+1))  //For the setup.
  //In case the number we are searching for is bigger than
     the register width.
    width = quantum_getwidth(N+1);
  printf ("The width is now: %d\n",width);
```

```
  reg = quantum_new_qureg(0, width);

  quantum_sigma_x(reg.width, &reg);

  for(i=0;i<reg.width;i++)
    quantum_hadamard(i, &reg);
  reg.node[0].state=8;
  reg.node[1].state=9;
reg.node[2].state=10;
reg.node[3].state=11;
reg.node[4].state=12;
reg.node[5].state=13;
reg.node[6].state=13;
reg.node[7].state=14;
quantum_multiple_once(&reg);
  quantum_hadamard(reg.width, &reg);
/*
  for(i=0; i<reg.size; i++)
    {
  printf("BEFORE State: %d",reg.node[i].state);
  printf("  Amplitude: %f",creal(reg.node[i].amplitude));
  printf("  multiples: %i\n",reg.node[i].multiples);
```

```
    }
*/

  printf("Iterating␣%i␣times\n", (int) (pi/4*sqrt(1<<reg.
     width)));
  int k;//number of solutions
  k=2;

  for(i=1; i<=pi/4*sqrt((1 << reg.width)/k); i++)
    {
      printf("Iteration␣#%i\n", i);
      grover(N, &reg);
    }

  quantum_hadamard(reg.width, &reg);

  reg.width++;

  quantum_bmeasure(reg.width-1, &reg);
  for(i=0; i<reg.size; i++)
    {
  printf("State:␣%d",reg.node[i].state);
  printf("␣␣Amplitude:␣%f",creal(reg.node[i].amplitude));
```

```
  printf("  Probability: %f",creal(reg.node[i].amplitude)
                                  *creal(reg.node[i].amplitude)
                                    );
  printf("  multiples: %i\n",reg.node[i].multiples);

   }
  quantum_delete_qureg(&reg);

  return 0;
}
```

A.3 Third Method of Implementation

The following is the code for using both the values and position within the quantum register.

```
/* grover.c: Implementation of Grover's search algorithm

   Copyright 2003 Bjoern Butscher, Hendrik Weimer

   This file is part of libquantum

   libquantum is free software; you can redistribute it and/
      or modify
```

```
   it under the terms of the GNU General Public License as
      published
   by the Free Software Foundation; either version 3 of the
      License,
   or (at your option) any later version.

   libquantum is distributed in the hope that it will be
      useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
      the GNU
   General Public License for more details.

   You should have received a copy of the GNU General Public
      License
   along with libquantum; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
   MA 02110-1301, USA

*/
/*
  final try to mod this program... going to go all out.
  Idea: throw in position on top of all the superpositions
```

```
    that will be created. Now, there is no need for multiples
    because each field will be unique. The only question now
    is what happens during the inversion about mean.
  This will pretty much result in twice the qubit sizes.
*/
#include <quantum.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <complex.h>

#ifdef M_PI
#define pi M_PI
#else
#define pi 3.141592654
#endif
static inline unsigned int
quantum_hash64(MAX_UNSIGNED key, int width)
{
  unsigned int k32;

  k32 = (key & 0xFFFFFFFF) ^ (key >> 32);
```

```
  k32 *= 0x9e370001UL;
  k32 = k32 >> (32-width);

  return k32;
}
static inline void
quantum_add_hash(MAX_UNSIGNED a, int pos, quantum_reg *reg)
{
  int i, mark = 0;

  i = quantum_hash64(a, reg->hashw);

  while(reg->hash[i])
    {
      i++;
      if(i == (1 << reg->hashw))
  {
    if(!mark)
      {
        i = 0;
        mark = 1;
      }
```

```
    else
      printf("Error.\n");;
  }
    }

  reg->hash[i] = pos+1;

}
void quantum_multiple_once(quantum_reg *reg)
{
  //first reset all multiple values.
  //printf ("size: %i\n",reg->size);
  int i,j;
  for (i=0;i<reg->size;i++)
  {
      reg->node[i].multiples=0;
  }


  //next update the multiple field based on reccurence
  for (i=0;i<reg->size;i++)
  {
    for (j=i+1;j<reg->size;j++)
```

```
    {
      if (reg->node[i].state==reg->node[j].state)
      {
        reg->node[j].multiples++;
      }
    }
  }

}

void oracle(int state,int positionwidth, quantum_reg *reg)
{
  int i;
  int newwidth=(reg->width)-positionwidth;
  //use the value and flag bit for the oracle.
  for(i=0;i<newwidth;i++)
    {
      if(!(state & (1 << i)))
  {
    quantum_sigma_x(i, reg);
  }
    }
```

```
  quantum_toffoli(0, 1, reg->width+1, reg);

  for(i=1;i<newwidth;i++)
    {
      quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
    }

  quantum_cnot(reg->width+i, reg->width, reg);

  for(i=newwidth-1;i>0;i--)
    {
      quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
    }

  quantum_toffoli(0, 1, reg->width+1, reg);

  for(i=0;i<newwidth;i++)
    {
      if(!(state & (1 << i)))
  quantum_sigma_x(i, reg);
    }

}
```

```
void inversion(int positionwidth, quantum_reg *reg)
{
  int i;
  int newwidth=(reg->width)-positionwidth;

  for(i=0;i<reg->width;i++)
  //for(i=0;i<newwidth;i++)
    quantum_sigma_x(i, reg);

  quantum_hadamard(reg->width-1, reg);
/*
  for(i=0; i<reg->size; i++)
    {
  printf("BEFORE State:   %d",reg->node[i].state);
  printf("  BEFORE Amplitude: %f",creal
          (reg->node[i].amplitude)*creal(reg->node[i].
             amplitude));
  printf("  BEFORE multiples: %i\n",reg->node[i].multiples);

   }
*/
//Am leaving this as reg->width so that it performs
```

```
//the inversion with respect to all the
//values, flag state, and position values.
//Getting a much better result this way.
  if(reg->width==3)
    //if(newwidth==3)
  {
  printf("Yep this ran\n");
    quantum_toffoli(0, 1, 2, reg);
//quantum_sigma_z(2,reg);
  }
  else
    {
      quantum_toffoli(0, 1, reg->width+1, reg);

      for(i=1;i<reg->width-1;i++)
  {
    quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
  }

      quantum_cnot(reg->width+i, reg->width-1, reg);

      for(i=reg->width-2;i>0;i--)
  {
```

```
    quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
  }

      quantum_toffoli(0, 1, reg->width+1, reg);
    }
/*
  printf("\n\n\n");
  for(i=0; i<reg->size; i++)
    {

  printf("AFTER State: %d",reg->node[i].state);
  printf("  AFTER Amplitude: %f",
    creal(reg->node[i].amplitude)*creal(reg->node[i].
      amplitude));
  printf("  AFTER multiples: %i\n",reg->node[i].multiples);

   }
*/
  quantum_hadamard(reg->width-1, reg);

  for(i=0;i<reg->width;i++)
 // for(i=0;i<newwidth;i++)
    quantum_sigma_x(i, reg);
```

```
/*
  for(i=0; i<reg->size; i++)
    {
  printf("State: %d",reg->node[i].state);
  printf("  Amplitude: %f",
    creal(reg->node[i].amplitude)*creal(reg->node[i].
      amplitude));
  printf("  multiples: %i\n",reg->node[i].multiples);

   }
*/
}


void grover(int target, int positionwidth, quantum_reg *reg)
{
  int i;

  oracle(target,positionwidth, reg);
  //oracle(1,reg);
  //oracle(target,reg);
  //oracle(6,reg);
```

```
/*
  for(i=0;i<reg->width;i++)
    quantum_hadamard(i, reg);

  inversion(positionwidth, reg);

  for(i=0;i<reg->width;i++)
    quantum_hadamard(i, reg);
*/
}
//now mod this to accept an array instead.
int begin_array()
{
  //do all the things we did in main,
    //but populate an array instead for testing.
}

int main(int argc, char **argv)
{
  quantum_reg reg;
  int i, N, width=0;

  srand(time(0));
```

```
  if(argc==1)
    {
      printf("Usage: grover [number] [[qubits]]\n\n");
      return 3;
    }
//take in a 3rd argument that handles the value of the
   positions.
  N=atoi(argv[1]);

  if(argc > 2)
    width = atoi(argv[2]);

  int positionwidth;
  positionwidth=atoi(argv[3]);
//now, mod this to be twice of N +1.
//First 3 bits = value, next 3 = position.
//now i need to modify to accept value, position on
//the main program and the other parts.
  if(width < quantum_getwidth(N+1))
//For the setup. In case the number we are searching
//for is bigger than the register width.
    width = quantum_getwidth(N+1);
```

```
  width=positionwidth+width;
  printf ("The width is now: %d\n",width);
  reg = quantum_new_qureg(0, width);

  quantum_sigma_x(reg.width, &reg);

  for(i=0;i<reg.width;i++)
      quantum_hadamard(i, &reg);
//ok.. so now we have 3 first bits as values, then next 3
//bits as positions of that value in the database.
//Final bit as the flag bit which gets
//superimposed in the next hadamard
//now populate the new values.

/*
  for(i=0; i<reg.size; i++)
    {
  int mask=0,j=0,k=0;

  printf("BEFORE State: %d\n",reg.node[i].state);
  for (j=0;j<((reg.width-1)/2)+1;j++)
  {
    mask = 1 << j;
```

```
        k = reg.node[i].state & mask;
        if( k == 0)
            printf("0");
        else
            printf("1");
    }

    printf("\nAmplitude: %f",creal(reg.node[i].amplitude));
    printf("   multiples: %i\n",reg.node[i].multiples);

    }
*/
quantum_hadamard(reg.width,&reg);

    int k;//number of solutions
    k=1;
    printf("Iterating %i
 times\n", (int) (pi/4*sqrt(1<<reg.width))/k);

    quantum_hadamard(reg.width, &reg);

    quantum_hadamard(reg.width, &reg);
```

```
reg.width++;


quantum_bmeasure(reg.width-1, &reg);
//i have to mod this bmeasure to preserve the multiples.
printf("\n\n\n\n");
for(i=0; i<reg.size; i++)
  {
int mask=0,j=0,k=0;


printf("State: %d ",reg.node[i].state);
for (j=(reg.width/2)-1;j>=0;j--)
{
  //printf("j is %d\n",j);
  mask = 1 << j;
       k = reg.node[i].state & mask;
       if( k == 0)
            printf("0");
       else
            printf("1");
}
printf(" Amplitude: %f",creal(reg.node[i].amplitude));
printf(" Probability: %f\n",
```

```
    creal(reg.node[i].amplitude)*creal(reg.node[i].amplitude)
        );

  quantum_delete_qureg(&reg);

  return 0;
}
```

A.4 The Dijkstra's Implementation Code

```
//important note: I've modified quantum get state
//to include a field called multiple.
/*
If there is a reoccurence of a state, the
multiple field for that state will be incremented by 1.
*/

/*
Sean's disclaimer: Part of the problem with using
grover for this pathfind is that it only works if
you know exactly what you are looking for, and you
are trying to see if it exists in the search space.
Hence, it is pretty much guaranteed that the
```

```
quantum section(using grover) is NOT more efficient,
in fact, it is definitely slower. Then again, my
thesis isn't about making it more efficient, but
whether or not it can be implemented using a
quantum algorithm.
*/
/*Much thanks goes out to libquantum.de's Bjoern
Butscher, Hendrik Weimer for their quantum
simulator and implementation of grover's algorithm.
Also to Jeffrey Hunter from www.iDevelopment.info
for his queue library.
*/
#include <GL/glut.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>

#include "grover.h"
//contains grover's algorithm for
//the quantum section
#include "queue.h"
```

```
//uses a queue for data storage implementation

int method;
//temporary variable. Will fix later
// for a proper interface
//to do: break up this pathfind
//section into another c-file soon. ;)
Queue Qx;//stores the x position
Queue Qy;//stores the y position
Queue distance;//stores the distance from begin point

int choice_of_array;
const int dist=1;
//Node->edges

int node_distance[16][16];
//Set the initial array of 15x15 to -1.
//This allows us to keep track of each
//distance from starting to end point.

//variables only for the unsorted array
int unsorted_minx, unsorted_miny,
unsorted_mindistance;
```

```
int startingx,startingy;
int endingx,endingy;



float xsphere=-3.0;
float ysphere=0.05;
float zsphere=6.0;

float nextxsphere;
float nextzsphere;

int nextx;
int nexty;
int xhistory[16][16];
int yhistory[16][16];


void dequeue_Quantumfindmin(int iteration_count, int
   final_distance)
//this function modifies the global variables:
// unsorted_minx, unsorted_miny, unsorted_mindistance
{
```

```
//first try a classical method but with
  //precalculating the minimum distance first.
/*
  The data only works if i don't ommit the out
      of boundary values for the grid. (I am ommitting)
  Hence, just choose a starting point away
      from the boundary for the quantum part to work.
  1 to 4 - minimum value of 1
  5-20 - min of 2
  21-52 - min of 3
  53-100 - min of 4
  IF the final distance is equal to the value
      above, skip the if statement.
*/
printf ("\nQuantum method being used. \n");
int current_distance;
  if ((iteration_count<=4) && (iteration_count>=1))
{
  current_distance=1;
}
else if ((iteration_count<=20) && (iteration_count>=5))
{
```

```
  current_distance=2;
}
else if ((iteration_count<=52) && (iteration_count>=21))
{
  current_distance=3;
}
else if ((iteration_count<=100) && (iteration_count>=53))
{
  current_distance=4;
}
else
{
  current_distance=5;
}
//now dequeue a temporary queue to find the smallest
   distance
Queue tempx;
tempx=CreateQueue(125);
//tempx=Qx;
Queue tempy;
tempy=CreateQueue(125);
//tempy=Qy;
Queue tempdist;
```

```
tempdist=CreateQueue(125);
//tempdist=distance;
int smallestdistance=999999;
int smallestx,smallesty;
//now sort
printf ("Size of Queue is : %d \n",ReturnSize(Qx));
while (!IsEmpty(Qx))
{
  if (Front(distance)<smallestdistance)
  {
    smallestdistance=Front(distance);
    smallestx=Front(Qx);
    smallesty=Front(Qy);
    Dequeue(Qx);
    Dequeue(Qy);
    Dequeue(distance);
  }
  else
  {
    Enqueue(Front(Qx),tempx);
      Dequeue(Qx);
    Enqueue(Front(Qy),tempy);
    Dequeue(Qy);
```

```
        Enqueue(Front(distance),tempdist);
        Dequeue(distance);
      }
      }
   Qx=tempx;
   Qy=tempy;
   distance=tempdist;
//   DisposeQueue(tempx);
//   DisposeQueue(tempy);
//   DisposeQueue(tempdist);
   unsorted_minx=smallestx;
   unsorted_miny=smallesty;
   unsorted_mindistance=smallestdistance;

}


void dequeue_findmin()
//this function modifies the global variables:
//unsorted_minx, unsorted_miny, unsorted_mindistance
{
   //now dequeue a temporary queue to find the smallest
      distance
```

```
Queue tempx;
tempx=CreateQueue(125);
//tempx=Qx;
Queue tempy;
tempy=CreateQueue(125);
//tempy=Qy;
Queue tempdist;
tempdist=CreateQueue(125);
//tempdist=distance;
int smallestdistance=999999;
int smallestx,smallesty;
//now sort
while (!IsEmpty(Qx))
{
  if (Front(distance)<smallestdistance)
  {
    smallestdistance=Front(distance);
    smallestx=Front(Qx);
    smallesty=Front(Qy);
    Dequeue(Qx);
    Dequeue(Qy);
    Dequeue(distance);
  }
```

```
    else
    {
      Enqueue(Front(Qx),tempx);
        Dequeue(Qx);
      Enqueue(Front(Qy),tempy);
      Dequeue(Qy);
      Enqueue(Front(distance),tempdist);
      Dequeue(distance);
    }
    }
  Qx=tempx;
  Qy=tempy;
  distance=tempdist;
//  DisposeQueue(tempx);
//  DisposeQueue(tempy);
//  DisposeQueue(tempdist);
  unsorted_minx=smallestx;
  unsorted_miny=smallesty;
  unsorted_mindistance=smallestdistance;
  printf ("Size of Queue is : %d \n",ReturnSize(Qx));
  printf ("Minimum distance is : %d \n",smallestdistance);
}
```

```
//Neighbour function
//Finds the neighbouring points and puts it in the queue
void find_neighbour()
{
  //calculate distance <---- not needed if
    //we list all the vertex and distance
  int path_distance=abs(unsorted_minx-startingx)+abs(
     unsorted_miny-startingy)+1;
  if (unsorted_minx!=1)//first neighbour -left
  {
       Enqueue(unsorted_minx-1,Qx);
       Enqueue(unsorted_miny,Qy);
       Enqueue(path_distance,distance);
  }
  if (unsorted_miny!=1)//2nd neighbour - bottom
  {
    Enqueue(unsorted_minx,Qx);
    Enqueue(unsorted_miny-1,Qy);
    Enqueue(path_distance,distance);
  }
  if (unsorted_minx!=15)//3rd neighbour - right
  {
    Enqueue(unsorted_minx+1,Qx);
```

```
    Enqueue(unsorted_miny,Qy);
    Enqueue(path_distance,distance);
  }
  if (unsorted_miny!=15)//4th neighbour - up
  {
    Enqueue(unsorted_minx,Qx);
    Enqueue(unsorted_miny+1,Qy);
    Enqueue(path_distance,distance);
  }
}

//By feeding this function a begin and end point,
//it will return the shortest path from the begin
//to the end in a vector, along with their distances.
//This should update the struct with the new values
//Does a fixed 15x15 grid, however, write this generic
//enough to do any grid size
int found=0;//while this is zero, don't draw the animated
   sphere also.
void pathfind(int beginx,int beginy, int endx, int endy)
{
  method=2;
  //First find the neighbours of the begin point.
```

```
  //we have a limit of 1-15 for x and y atm.
  startingx=beginx;
  startingy=beginy;

  unsorted_minx=beginx;
  unsorted_miny=beginy;


  node_distance[beginx][beginy]=0;
  find_neighbour();
//first 4 neighbours go into the queue. yay!
  found=0;

//The final distance and iteration count
//is only used for the quantumfind method
  int final_distance=abs(startingx-endingx)+abs(startingy-
     endingy);
  int iteration_count=1;
  while (!IsEmpty(Qx)&&found==0)
  {
    //next step, find the minimum value from the queue. (This
        is an unsorted array)
    if (method==1)
```

```
    {
      printf ("The final distance is : %d\n",final_distance);
      dequeue_findmin();
    }
    else
    {

      printf ("The final distance is : %d\n",final_distance);
      dequeue_Quantumfindmin(iteration_count,final_distance);
    }
    printf("Total iteration count: %d \n",iteration_count);
    iteration_count++;
//after that, unsorted_minx/y/distance
//is updated with the new minimum distance to travel.
//now we update the grid values(array) with the new
//distances. If it is -1, update, if it is zero skip,
// if it is less than the existing value, update.
    if ((node_distance[unsorted_minx][unsorted_miny]==-1)
||(((node_distance[unsorted_minx][unsorted_miny])!=0)
&&(node_distance[unsorted_minx][unsorted_miny]>
   unsorted_mindistance)))
    {
```

```
        node_distance[unsorted_minx][unsorted_miny]=
            unsorted_mindistance;
        //if indeed the distance is the shortest, then we
create a history of this point(add old history) to store
        if (node_distance[unsorted_minx-1][unsorted_miny]==
            unsorted_mindistance-1)
        {
          xhistory[unsorted_minx][unsorted_miny]=unsorted_minx
              -1;
          yhistory[unsorted_minx][unsorted_miny]=unsorted_miny;
        }
        else if (node_distance[unsorted_minx][unsorted_miny
            -1]==unsorted_mindistance-1)
        {
          xhistory[unsorted_minx][unsorted_miny]=unsorted_minx;
          yhistory[unsorted_minx][unsorted_miny]=unsorted_miny
              -1;
        }
        else if (node_distance[unsorted_minx+1][unsorted_miny
            ]==unsorted_mindistance-1)
        {
          xhistory[unsorted_minx][unsorted_miny]=unsorted_minx
              +1;
```

```
        yhistory[unsorted_minx][unsorted_miny]=unsorted_miny;
      }
      else if (node_distance[unsorted_minx][unsorted_miny
         +1]==unsorted_mindistance-1)
      {
        xhistory[unsorted_minx][unsorted_miny]=unsorted_minx;
        yhistory[unsorted_minx][unsorted_miny]=unsorted_miny
           +1;
      }
//see if current minimum point is the end point.
// If it is, end it and pop the path queue (the found path)

      if ((unsorted_minx==endx)&&(unsorted_miny==endy))
      {
        found=1;
        //set the coordinates up to draw the animated sphere
        xsphere=(endx/2.5)-3.2;
        zsphere=(endy/2.5)-0.2;
        printf ("Beginx = %d\n",beginx);
        printf ("Beginy = %d\n",beginy);
        printf ("Xsphere and zsphere = %f %f\n",xsphere,
           zsphere);
        nextx=xhistory[endx][endy];
```

```
        nexty=yhistory[endx][endy];
        nextxsphere=(xhistory[endx][endy]/2.5)-3.2;
        nextzsphere=(yhistory[endx][endy]/2.5)-0.2;
      }
      else
//if it isn't the end point, then push
//it in a found path and and determine
//it's according neighbours.
      {
        find_neighbour();
      }
    }
  }
}

void initialise_array()
{
  int i,j;
  for (i=1; i<16;i++)
  {
    for (j=1;j<16;j++)
    {
      node_distance[i][j]=-1;
```

```
    }
  }
}


static float angle=0.0,ratio;
static float x=0.0f,y=25.0f, z=35.0f;
static float lx=0.0f,ly=-0.50f,lz=-0.65f;

//mousekeys
int lastx;
int lasty;

int spheretest[15][15];

int program_state; //1 when it begins. 2 when begin point is
   entered, 3 when end point is entered.

void changeSize(int w, int h)
{

  // Prevent a divide by zero, when window is too short
  // (you cant make a window of zero width).
```

```
    if(h == 0)
      h = 1;

    ratio = 1.0f * w / h;
    // Reset the coordinate system before modifying
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Set the viewport to be the entire window
    glViewport(0, 0, w, h);

    // Set the clipping volume
    gluPerspective(10,ratio,1,1000);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(x, y, z,
              x + lx,y + ly,z + lz,
             0.0f,1.0f,0.0f);
}

void drawgrid()
{
```

```
float redMaterial[]={1,0,0,1};
GLUquadricObj *cylquad = gluNewQuadric();
float counter;
glPushMatrix();
//  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, redMaterial);
  glColor3f(1.0f, 0.5f, 0.5f);
  glTranslatef(-3.0f,0.0f,0.0f);
  glRotatef(0.0f,0.0f,0.0f,0.0f);
  gluCylinder(cylquad, 0.05, 0.05, 6, 100, 2 );
  glRotatef(90.0f,0.0f,1.0f,0.0f);
  gluCylinder(cylquad, 0.05, 0.05, 6, 100, 2 );
  //this draws everything going down
  for (counter=1;counter<16;counter+=1)
  {
    glTranslatef(-0.4f,0.0f,0.0f);
    gluCylinder(cylquad,0.05,0.05,6,100,2);
  }
glPopMatrix();



glPushMatrix();
  glColor3f(1.0f, 0.5f, 0.5f);
  glTranslatef(3.0f,0.0f,0.0f);
```

```
    glRotatef(0.0f,0.0f,0.0f,0.0f);
    gluCylinder(cylquad, 0.05, 0.05, 6, 100, 2 );
    for (counter=1;counter<15;counter+=1)
    {
      glTranslatef(-0.4f,0.0f,0.0f);
      gluCylinder(cylquad,0.05,0.05,6,100,2);
    }
  glPopMatrix();
}


void drawsphere()
{
  //glEnable(GL_TEXTURE_2D);
  //glBindTexture(GL_TEXTURE_2D,2005);
  if (found==1)
  {
  glPushMatrix();
    glColor3f(0.0f,0.0f,0.0f);
    glTranslatef(xsphere, ysphere, zsphere);
    glutSolidSphere(0.1f,10,10);
  glPopMatrix();
  }
```

```
  //glDisable(GL_TEXTURE_2D);
}

void drawspheretest(int testx,int testy)
{
  //testx=location on grid x. (15 possibilities)
  //testy=location on grid y. (also 15 possibilities)
  float xresult=0;
  float yresult=-0.05;
  float zresult=0;
 //next convert the testx,testy to locations on the 3d space
  xresult=(testx*0.4)-3.4;
  zresult=(testy*0.4)-0.4;
/*
  switch( testx )
  {
  case 1:
    xresult=-3.0;
  break;
  case 15:
    xresult=2.6;
  break;
  default:
```

```
  xresult=0;
  }
  switch (testy)
  {
  case 1:
    zresult=0;
  break;
  case 15:
    zresult=5.6;
  break;
  default:
  zresult=0;
  }
*/

        //glEnable(GL_TEXTURE_2D);
        //glBindTexture(GL_TEXTURE_2D,2005);
        glPushMatrix();
          glColor3f(1.0f,1.0f,0.0f);
/*
    glBegin(GL_QUADS);
      glVertex3f(xresult,yresult,zresult);
      glVertex3f(xresult+0.4,yresult,zresult);
```

```
        glVertex3f(xresult, yresult, zresult+0.4);
        glVertex3f(xresult+0.4, yresult, zresult+0.4);
    glEnd();
*/
            glTranslatef(xresult+0.2, yresult, zresult+0.2);
    glutSolidCube(0.4);
//                  glutSolidSphere(0.1f,10,10);
//    glutSwapBuffers();
        glPopMatrix();
//  glutSwapBuffers();
        //glDisable(GL_TEXTURE_2D);
}

void animatesphere()
{
//for now one direction
  if (xsphere<=nextxsphere)
  {
    xsphere+=0.05;
  }
  else if (zsphere<=nextzsphere)
  {
    zsphere+=0.05;
```

```
  }
  else
  {
    //update nextspheres to the ones in the array.
    if (node_distance[nextx][nexty]!=1)
    {
      nextxsphere=xhistory[nextx][nexty];
      nextzsphere=yhistory[nextx][nexty];
      nextx=nextxsphere;
      nexty=nextzsphere;
      nextxsphere=(nextx/2.5)-3.2;
      nextzsphere=(nexty/2.5)-0.2;
    }
    else
    {
      nextxsphere=(startingx/2.5)-3.2;
      nextzsphere=(startingy/2.5)-0.2;
    }
  }
  glutPostRedisplay();
}

void renderScene(void)
```

```
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glColor3f(0.9, 0.9, 0.9);
  glBegin(GL_QUADS);
    glVertex3f(-100.0f, 0.0f, -100.0f);
    glVertex3f(-100.0f, 0.0f,  100.0f);
    glVertex3f( 100.0f, 0.0f,  100.0f);
    glVertex3f( 100.0f, 0.0f, -100.0f);
  glEnd();
  glPopMatrix();
  drawgrid();
  drawsphere();
  int counter;
  int counter2;
  for (counter=1;counter<=15;counter++)
  {
    for (counter2=1;counter2<=15;counter2++)
    {
      if (spheretest[counter][counter2]==1)
      {
        drawspheretest(counter,counter2);
      }
```

```
    }
  }
  glutSwapBuffers();
}

void orientMe(float ang) {


  lx = sin(ang);
  lz = -cos(ang);
  glLoadIdentity();
  gluLookAt(x, y, z,
            x + lx,y + ly,z + lz,
           0.0f,1.0f,0.0f);
}


void moveMeFlat(int i) {
  x = x + i*(lx)*0.1;
  z = z + i*(lz)*0.1;
  glLoadIdentity();
  gluLookAt(x, y, z,
            x + lx,y + ly,z + lz,
```

```
        0.0f,1.0f,0.0f);
}

void processNormalKeys(unsigned char key, int x, int y) {

  if (key == 27)
    exit(0);
}



void inputKey(int key, int x, int y) {

  switch (key) {
    case GLUT_KEY_LEFT : angle -= 0.01f;orientMe(angle);break
       ;
    case GLUT_KEY_RIGHT : angle +=0.01f;orientMe(angle);break
       ;
    case GLUT_KEY_UP : moveMeFlat(1);break;
    case GLUT_KEY_DOWN : moveMeFlat(-1);break;
  }
}
void initialise_lights()
{
```

```
  GLfloat lightIntensity[]={0.2,0.2,0.2,1.0f};
    GLfloat lightPosition[]={0.0f,10.0f,0.0f,0.0f};
    glLightfv(GL_LIGHT0,GL_POSITION,lightPosition);
    glLightfv(GL_LIGHT0,GL_AMBIENT,lightIntensity);
  glColorMaterial(GL_FRONT_AND_BACK,GL_DIFFUSE);
  glEnable(GL_COLOR_MATERIAL);
}

void mouseInput(int button, int state, int xmouse, int ymouse
   )
{
    if(state == GLUT_DOWN)
    {
        lastx=xmouse;
        lasty=ymouse;
        if(button == GLUT_LEFT_BUTTON)
        {
    int store_temporaryx,store_temporaryy;
    printf ("Pressed left button\n");
    printf("X=%d , Y=%d",xmouse,ymouse);
    //implement some sort of mouse to array conversion here.
  //x213,y292 draw at top left hand corner
  //x637,y509 draw at
```

```
    store_temporaryx=(xmouse-80)/38;
    store_temporaryy=(ymouse-180)/25;
    //if ((xmouse>100&&xmouse<230) && (ymouse>250&&ymouse
       <310))
    if (store_temporaryx<1 || store_temporaryy<1 ||
 store_temporaryx>15||store_temporaryy>15)
    {
      program_state=0;
//invalid position, skip next step and reenter again.
    }

    //for initial, store starting point
    if (program_state==1)
    {
      printf ("Entering state 1\n");
      spheretest[store_temporaryx][store_temporaryy]=1;
      startingx=store_temporaryx;
      startingy=store_temporaryy;
      program_state++;
      printf ("Starting x has value of: %d\n",startingx);
      printf ("Starting y has value of: %d\n",startingy);
    }
//for second, store end point and
```

```
//then run the pathfind function and start drawing
    else if (program_state==2)
    {
      printf ("Entering state 2\n");
      spheretest[store_temporaryx][store_temporaryy]=1;
      endingx=store_temporaryx;
      endingy=store_temporaryy;
      program_state++;
      printf ("Ending x has value of : %d\n",endingx);
      printf ("Ending y has value of : %d\n",endingy);

      //after 2nd point has been selected, we run the
         pathfind algorithm.
      choice_of_array=1;
      Qx=CreateQueue(125);
      Qy=CreateQueue(125);
      distance=CreateQueue(125);
      initialise_array();
      //printf ("Using point 5,5 as begin point.\n");
      //printf ("Using point 10,15 as end point.\n");
      switch (choice_of_array)
      {
        case 1:
```

```
            printf ("Using an unsorted array.\n");
            pathfind(startingx, startingy, endingx, endingy);
        break;
        case 2:
            printf ("Using a priority queue.\n");
        break;
        case 3:
            printf ("Using grover's search.\n");
        break;
        default:
        break;
      }
    }
//haven't figured out what to add yet..
// maybe loop program to run again.
    else if (program_state==3)
    {
    }
    else //if invalid coordinates are entered.
    {
      program_state=1;
    }
        }
```

```
    }
}
int main(int argc, char **argv)
{

  method=2;//1 for traditional unsorted array. 2 for quantum
     method.
  //maingrover(4,3);
  program_state=1; //begin our program state
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
  glutInitWindowPosition(100,100);
  glutInitWindowSize(800,800);
  glutCreateWindow("Dijkstra's pathfind using Quantum
     algorithms");

  glutKeyboardFunc(processNormalKeys);
  glutMouseFunc(mouseInput);
  glutSpecialFunc(inputKey);

  glutDisplayFunc(renderScene);
  glutIdleFunc(renderScene);
  glutIdleFunc(animatesphere);
```

```
    glutReshapeFunc(changeSize);

    //turn on the lights
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glShadeModel(GL_SMOOTH);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_NORMALIZE);
    initialise_lights();
    glutMainLoop();


    return(0);
}
```

REFERENCES

[1] A.Pittenger. *An Introduction to Quantum Computing Algorithms.* Birkhauser, Boston, Basel, Berlin, 1st edition, 2000.

[2] B.Butscher and H.Weimer. Libquantum. http://www.libquantum.de/, 2008.

[3] C.Shannon. A Mathematical Theory of Communication. http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html, 2006.

[4] D.Divincenzo. The Physical Implementation of Quantum Computation. *quant-ph/0002077*, 2003.

[5] E.Weisstein. Dirac Notation. http://mathworld.wolfram.com, 2009.

[6] H.Lucien. Quantum Theory From Five Reasonable Axioms. *quant-ph/0101012v4*, 2001.

[7] IBM. IBM's Test-Tube Quantum Computer Makes History. *IBM Press Release*, December 2001.

[8] J.Townsend. *A Modern Approach To Quantum Mechanics.* University Science Books, Sausalito California, 1st edition, 2000.

[9] K.Dorai and A.Kumar. Implementation of a Deutsch-Like Quantum Algorithm Utilising Entanglement at the Two-Qubit Level on an NMR Quantum Informa-

tion Processor. http://eprints.iisc.ernet.in/archive/00000300/01/Deutsch.pdf, 2004.

[10] L.Grover. What's a Quantum Phone Book? http://www.bell-labs.com/user/feature/archives/lkgrover/, 2002.

[11] I.Chuang M.Nielsen. *Quantum Computation and Quantum Information.* Cambridge, Cambridge UK, 1st edition, 2000.

[12] P.Horodecki. Separability Criterion and Inseparable Mixed States With Positive Partial Transposition. *Physics Letters A,232:333-339,* August 1997.

[13] Q.Norton. The Father of Quantum Computing. http://www.wired.com/science/discoveries/news/2007/02/72734, 2007.

[14] J.Bub. Quantum Entanglement and Information. http://plato.stanford.edu/entries/qt-entangle/, Aug 13 2001.

[15] R.Landauer. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development, 5:183-191*, 1961.

[16] R.Perry. The Temple of Quantum Computing. http://www.toqc.com/, 2006.

[17] S.Banerjee. Quantum Computation and Information Theory - Lecture 1. http://www.cse.iitd.ernet.in/nsuban/quantum/lectures/lecture1.pdf, 2004.

[18] S.Lloyd. Quantum Search Without Entanglement. *Physical Review A, 61(1):010301(4)*, December 1999.

[19] Y.Orlov. Quantumlike Bits and Logic Gates Based on Classical Oscillators. *Physical Review A, 66(5):052324(4)*, 2002.

[20] Z.Meglicki. *Quantum Computing Without Magic.* The MIT press, Cambridge Massachusetts, London England, 1st edition, 2008.