ECE 510 Introduction to Computational Intelligence
Quantum Computing Assignment
John Gebbie
27 November 2007

Introduction

The purpose of this assignment is to learn the fundamentals of quantum computing in a hands on manner. The assignment calls for the design, implementation, and simulation of a quantum computer. This paper discusses the design in-depth and also talks about the relevant pieces of the implementation.

This quantum computer is not a general purpose computing platform. Instead, it is tailored to solve a specific problem. The problem chosen for this quantum computer to solve is the maximum independent set problem in graph theory. The results produced by simulating this problem the quantum computer are presented and discussed.

Background

Quantum Computing is an entirely new way of computing. It leverages the quantum nature of subatomic particles to perform parallel calculations. When these particles are in a *superposition* state, they have a certain probability of collapsing to a zero or one state when observed. The act of observing and collapsing the particle causes it to lose information – namely its superposition as well as its phase. A quantum computer is built around the idea of keeping particles in a state of superposition and manipulating them there in order to use this extra information as part of its computations.

Two quantum particles both in superposition states can become entangled. This essentially means the extra information that each particle contains is linked with the other one. A quantum circuit uses these entangled particles to perform parallel computations. The Grover algorithm is an example such of a quantum computer.

Overview of Tasks

This section gives an overview of the tasks that were performed for this project. At a high level, the project broke down into several major parts:

- implementation of a Grover algorithm simulator
- development of a framework for building quantum circuits
- design and implementation of an oracle for determining if a set of graph nodes is independent
- design and implementation of an oracle for comparing the size of an independent set to a threshold value

The Grover simulator hosts an arbitrary oracle circuit and searches for good solutions using that oracle. The simulator can be thought of as a test harness for oracles; except that the test harness is the quantum computer itself. The simulator can also be used to verify that a given oracle circuit functions correctly. Lastly, seeing the simulator function properly will indicate the Grover algorithm has been properly implemented in Matlab code.

The framework code facilitates the construction of complex oracles and other circuits from simpler components. The framework can be thought of as a set of software tools for transforming arbitrary quantum circuits into a form which is consumable by the Grover algorithm.

Each oracle implements the logic for testing a solution to a particular problem. For this project, several small oracles were built that each implement a part of an algorithm for solving the "maximum independent set" mapping problem. These parts were then put together into a final oracle that implemented the full algorithm.

Grover

This section talks about the design of the Grover simulator.

The simulator takes the following inputs:

- the oracle circuit
- the number of input wires
- extra iterations (explained later)

To enable us to use the Grover simulator with multiple oracles, the oracle circuit is a matrix which is passed in as a parameter. The size of the oracle matrix dictates the number of wires the oracle circuit uses (t). The other parameter is the number of inputs the oracle itself takes (n) which are all the wires excluding the ancilla bits. This number is always less than the total number of wires in the oracle circuit ($n < t$).

The wires of the oracle circuit are grouped by inputs first, followed by the oracle bit, and then the ancilla bits. All bits are initially given the value of $|0\rangle$. The following diagram graphically shows the Grover algorithm. Note the position of the wires going into and out of the circuit.

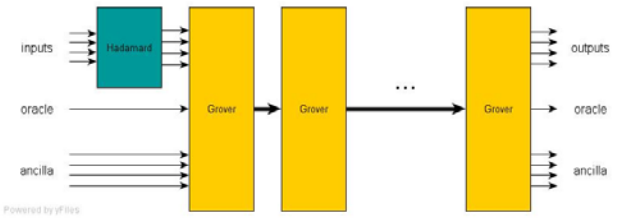

Figure 1. Flow of the Grover Algorithm

After initializing all the input wires, just the inputs are passed through Hadamard gates. This puts each of the input wires into a superposition state (initially equal probability of collapsing to a $|0\rangle$ or $|1\rangle$). At this stage, the oracle and ancilla bits are passed through unmodified.

Next the wires are passed to the first Grover circuit. The Grover circuit uses two Hadamard gates sandwiched around a zero-state phase shift gate; which is then fed into the oracle circuit. The following diagram illustrates this:

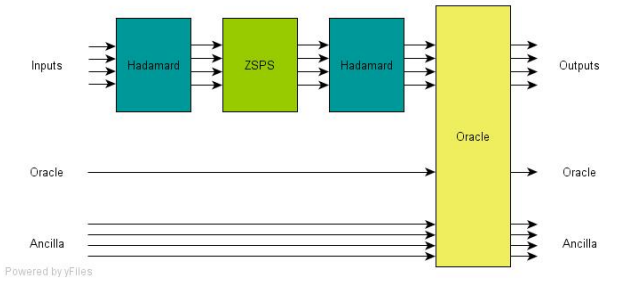

Figure 2. Composition of Grover Matrix

Notice the Hadamard and zero-state phase shift gates are only applied to the *input* wires; not the oracle or ancilla wires. But the oracle is passed all wires.

The number of times the Grover circuit is applied is given by the following formula:

$$\left\lceil \frac{\pi\sqrt{N}}{4} \right\rceil$$

Formula 1. Number of Grover Iterations

where N is the number of inputs. In practice, it was found that the number of iterations required to produce valid solutions from the algorithm varied slightly from the above algorithm. The amount of variation was at most two iterations. This was the reason for adding the *niterExtra* parameter to *grover.m*. The reason for this discrepancy is unknown and may be due to a design error.

The output of the algorithm is a vector of entangled qubits all in superposition. When observed, the entangled vector snaps into a given state. Every possible state has associated with it a probability. Note that the state includes all wires, not just input wires. In a real quantum computer, the implementor would need to measure the output of the Grover circuit multiple times to determine the probability distribution. If the algorithm was successful at finding a solution, the states that contain that solution will have higher probability than the other, non-solution states and so will be observed more often.

The simulator, however, produces a vector of probabilities. So, instead of having to measure to determine the probability distribution, all that needs to be done is add up the appropriate probabilities. Since all wires output from of the Grover circuit are part of the probabilistic state, we must group together the probabilities common for every state of just the input wires. In other words, if the oracle takes 10 wires but the top four are "inputs" to the oracle, then we are interested in just every combination of the four input wires, not every combination of all wires. After grouping the probabilities for each combination of inputs, we can sort by probability to determine if the algorithm produced any solutions and, if so, what those solutions are.

The code for the Grover algorithm is contained in the file *grover.m*. The merging and printing code are in the files:

- merge_probs.m
- printresults.m.
- bubblesort.m

Maximum Independent Set

The quantum computer that was built for this project solves a mapping problem called *Maximum Independent Set*. What is an independent set? An independent set is a set of nodes where no two nodes in the set are connected by ane edge. A

Maximal independent set is an independent set that is not a subset of any other independent set.

Consider the following example graph.

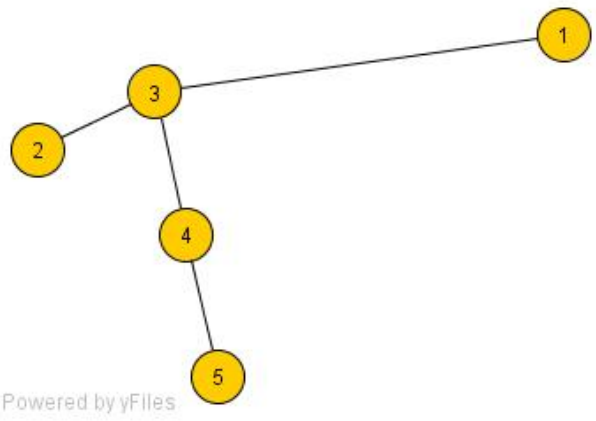

Figure 3. Example Graph

In the above graph, $\{5,3\}$ is an independent set but $\{5,4\}$ is not. Since $\{5,3\}$ is not a subset of any other independent set, we can say it is *maximal*. However, it is not the *maximum* independent set. Consider the set $\{4,2\}$. This set is independent but not maximal. However, the set $\{4,1,2\}$ is maximal. And since $\{4,1,2\}$ is bigger than $\{5,3\}$ we know $\{5,3\}$ is not maximum. In this trivially simple example, we can see that $\{4,1,2\}$ is one of the maximum independent sets, but in a larger graph this would be less clear. Also, $\{4,1,2\}$ is not the *only* maximum independent set. There is also $\{5,1,2\}$.

Finding a *maximal* independent set can be done relatively easily. Starting at given node, we can successively visit each node in the graph and either add it to the set if it is non-adjacent to any node already in the set, or skip it if it is adjacent. Finding the *maximum* independent set is much more difficult because it involves back tracing to see if prior decisions caused poor final results. Finding maximum independent sets is an NP-complete problem.

Framework

To facilitate the development of complex quantum circuits, a software framework was developed for this project. This framework provided the "glue" necessary for linking smaller circuits together into bigger, compositional circuits. It also provided ways of displaying results of the Grover simulator.

The Subgate

The framework builds on several ideas from class about reversibility and wire swapping and fits them into software constructs accessible through Matlab. The core construct is the *subgate*. This term is not intended to be related to any outside concept.

A subgate is a reversible circuit and consists of an ordered list of matrices. It has a size which is equal to the number of wires entering and leaving it. Each matrix has a size of $2^n \times 2^n$ where *n* is the size. A subgate is designed to be able to be dropped into a bigger circuit and connected to an arbitrary set of wires from the bigger circuit.

When putting a smaller circuit into a bigger circuit, the smaller circuit is wrapped in a subgate that *routes* particular wires to its inputs. This is done by swapping wires multiple times to shuffle the wires into the proper place. After the subgate, the wires need to be un-shuffled back into their original positions. A subgate always places the wrapped circuit in the top wires of the containing circuit and routes the wires up to it rather than placing it somewhere in the middle. The former approach is taken because it is simpler but may actually require more swaps.

When a wrapped circuit covers the top wires of a containing circuit there are usually extra wires at the bottom of the containing circuit that need to pass "underneath" the wrapped circuit. This entails adding wires to the matrices of the wrapped circuit by applying the Kronecker product multiple times to a 2×2 identity matrix (which is the matrix for a wire) to increase the size of the wrapped circuit until it exactly fits inside the containing circuit.

One of the motivations for representing a subgate as a list of matrices is it makes subgates compose-able. In other words, if subgate A is nested inside of B which is nested inside of C, the matrices of A are still valid, perhaps bigger, but basically unchanged inside C's list. Surrounding A's matrices are matrices that do the routing from B's environment to A's and vice-versa. Likewise bookending B's matrices are routing matrices that make B a part of C's environment.

A common need in quantum circuit design is to create logic that computes some value on an ancilla bit wire and then returns all the other wires to their original state. To do this, the circuit that computes the final result can be mirrored over the last step which essentially un-computes the values on the other wires. Mirroring a subgate is easy because it just involves making a copy of the list and reversing the order of its matrices.

To use a subgate in computations, it is necessary to collapse all the matrices down to a single matrix. This is done by computing the dot-product of all the matrices in the list in reverse order.

The files that implement subgate are:

- subgate.m
- mat_swap_wires.m
- mat_move_wire.m
- mirror_gate.m
- collapse_gate.m

Oracles and Circuits

This section talks about the design of the maximum independent set oracle used in the project. The oracle is not a general-purpose oracle for determining the maximum independent set of an arbitrary graph. Rather, it is designed to operate on a particular graph. However, using the principals embodied in this design, it would be possible to adapt this oracle to work with any arbitrary graph.

The test graph this oracle is designed to analyze is represented in the following figure.

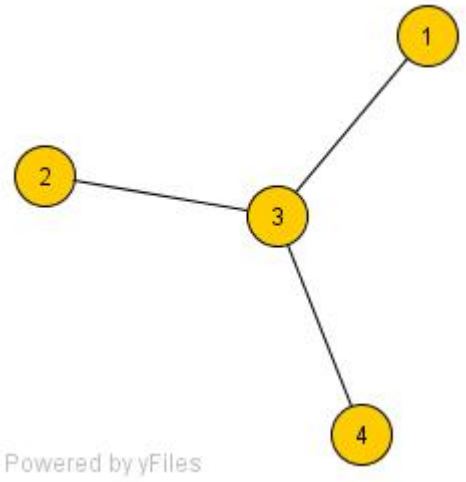

Figure 4. Test Graph For This Project

Composition

Several smaller circuits comprise this oracle. Each of these components focuses on a part of the overall algorithm of the oracle. This divide and conquer approach has advantages because it logically breaks down the algorithm into more manageable pieces which can be developed, debugged, and tested more readily.

Conceptually, we can think of the oracle as having two main parts: the independent set circuit and the threshold calculation circuit. The former is concerned with determining whether a set (of nodes) is independent or not; and the latter focuses on determining whether the set size is greater than some threshold value.

The rest of this section will describe each of these oracles in depth. Each sub section describes the purpose behind each component and explains the role it plays. It will then discuss how these parts fit together into the final full oracle.

Independent Set Circuit

This circuit is at the heart of the maximum independent set algorithm. It determines whether a set is independent or not. Recall that a set is independent if none of its nodes are adjacent. This circuit does not make any claims about the maximality of the set; just its independence. If a set is non-independent (has adjacent nodes) it must be excluded as a possible solution. This circuit outputs a boolean value of |1> if the graph is independent or |0> if not.

The following figure shows this quantum circuit:

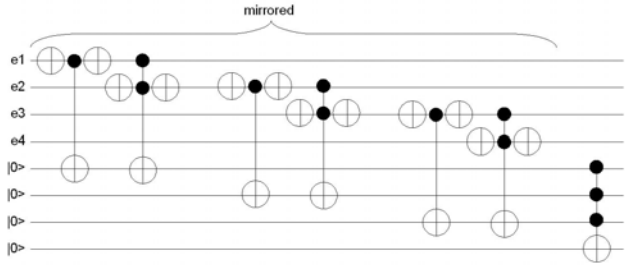

Figure 5. Graph Oracle Quantum Circuit

Each of the four edges are passed in on a separate wire. The value of the wire indicates whether it is in the test set (|1>) or not (|0>). This part of the algorithm wants to know whether nodes are not adjacent. So, for each edge we NAND the

nodes at the endpoint of that edge together. If either or neither of the nodes are in the test set, then that edge does not violate the independence constraint. If, however, both of the nodes are |1> then we know the entire set is not independent because there is an edge that connects two of the nodes.

In the test graph in figure 4, there are three edges. Correspondingly, there are three NAND gates in the circuit. These are the three repeated structures starting at the left and moving to the right. Each of these NAND gates outputs its value on an ancilla bit. At the far right a Toffoli gate ANDs these values together so that the oracle returns a |1> only if all edges satisfy the independence constraint.

At the right edge of this circuit, the ancilla bits carry the result of the NAND operations. Since this circuit will be part of a larger circuit, we need to return these wires to their original value of |0>. To do this, we mirror the whole circuit – excluding the Toffoli gate – on the right after the Toffoli gate. This will leave the oracle value (|0> or |1>) on the last wire but all other wires will appear unchanged to any circuits that may come after this circuit.

Node Counter Circuit

This circuit counts the number of wires that have the value |1>. It returns this as a binary number. The range of values is, of course, zero through four. This means that at most three bits are required to represent the output.

Logically, this is represented in the figure below as a non-quantum circuit. The binary number that is produced is the value "c3 s3 s2".

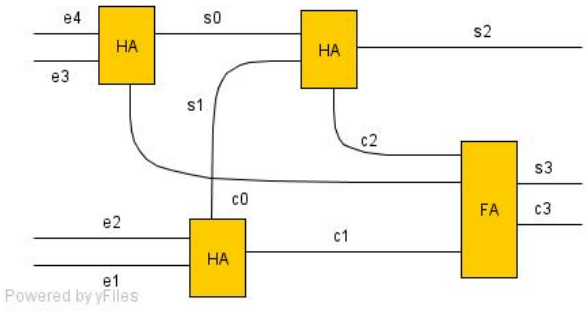

Figure 6. Node Counter Regular Circuit

When we transform this into a quantum circuit, we require some ancilla bits – one for each adder. The following figure shows this circuit.

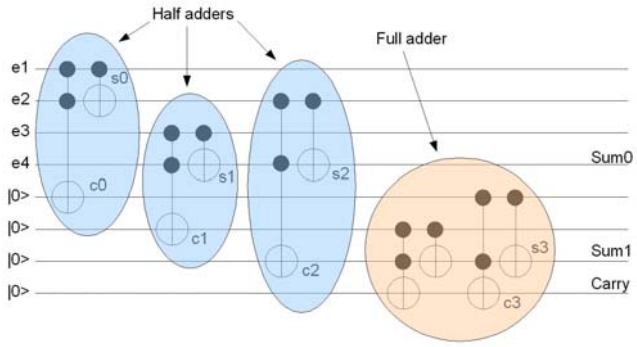

Figure 7. Node Counter Quantum Circuit

Notice how the half adders re-use the carry lines from the adders before them. This saves using unnecessary ancilla bits. The output of this circuit is a three bit value on the 4th, 7th and 8th wires. This circuit is intended to be used as part of a bigger circuit and then re-applied in its mirrored form to restore the original input values. Since this circuit actually overwrites one of the edge input values, the circuit that comes after this must not need this information. The next circuit is the Comparator Circuit.

Threshold Comparator Circuit

This circuit compares the number of nodes in the test set to some predefined threshold value. This circuit takes both of these inputs as 3-bit binary numbers and outputs a single bit indicating whether they are equal.

To test for equality, we must first determine if any of the individual bits between the two binary numbers differ. The logical operation that will test for this on each bit is the XOR operation which outputs a |1> if two bits differ or a |0> if they are the same. This is actually the inverse of what we want so we use the ~XOR instead. The ~XOR returns |1> if both bits are the same and |0> if they are not. If we apply this operation to each of the bits we get a vector of equality values. Since we are interested in equality over the entire number we use a simple AND operation to get a single output value for the total equality.

The following diagram shows this logical flow.

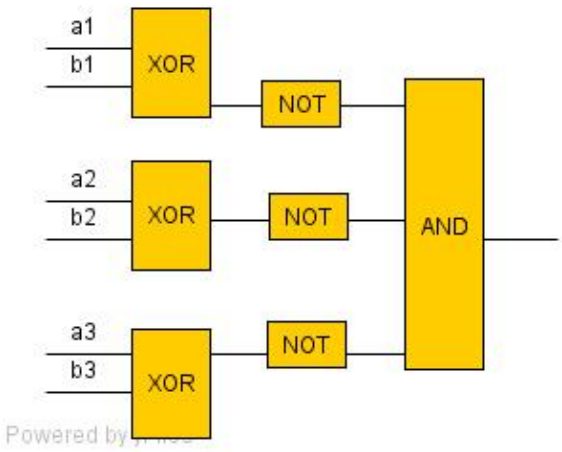

Figure 8. Set Size Threshold Comparator Regular Circuit

To translate this into a quantum circuit, we overwrite the second number with the ~XOR values and then output the resulting AND value on an ancilla bit wire.

The following figure shows the above regular circuit translated into a quantum circuit.

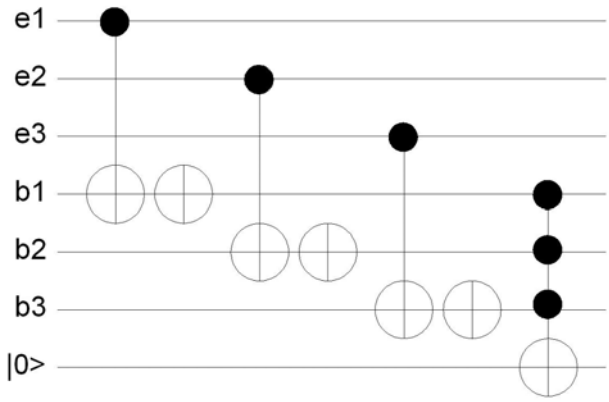

Figure 9. Set Size Threshold Comparator Quantum Circuit

Only if $e1=b1$, $e2=b2$, and $e3=b3$ then the last wire will contain a value of |1>. Since this circuit overwrites its inputs, it will need to be mirrored again to reverse its changes.

Full Maximum Independent Set Oracle

We now have circuits for testing whether a set of nodes is independent. We also have circuits for counting nodes in a test set and for comparing that binary number to a threshold value. The last step is to combine these together into a single circuit that outputs a |1> if the test set is independent and has a given number of nodes.

The following figure shows all the circuits discussed up to this point together in a single full circuit.

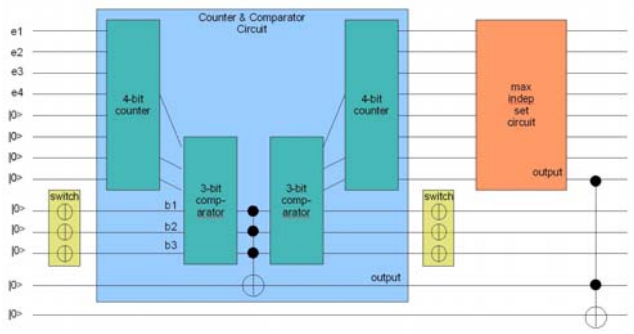

Figure 10. Full Oracle Quantum Circuit

Notice that all circuits are reversed in-place except for the last Toffoli gate whose value remains on the last wire after the circuit finishes executing. This is because this is an oracle circuit and needs a single output value. This is a requirement for a circuit to be an oracle.

Another requirement is that the oracle output wire must come immediately after the inputs but before the ancilla bits. So something that is not shown here but is part of this circuit is the bottom wire is moved into position 5 after the Toffoli gate at the right. This puts the circuit into a form that is consumable by the Grover algorithm.

In the circuit, the switch boxes encode the threshold value. This value is not a true input to the oracle but rather is encoded in the oracle with inverter gates as it is constructed. The advantage of this approach is it reduces the number of inputs which allows the circuit to be simulated much faster.

The big blue box contains the 4-bit counter and 3-bit comparator circuits. These are followed by a single Toffoli gate and then are mirrored again afterwards. The mirroring undoes any changes to the other wires so that the independent set circuit can take these wires as inputs without having to know about any changes. The counter and comparator outputs a boolean value for whether the test set (represented by the four input wires at the top) matches the threshold. The independent set oracle outputs a boolean for whether the test set

is independent. The last Toffoli gate ANDs these two booleans together and puts the result on the last wire.

Normally the entire shown circuit, excluding the last Toffoli gate, would be mirrored after the Toffoli gate. This is not done here because the remaining values on all the output wires (except the last) are identical to the input. So the requirement of an oracle circuit to have only a single output without modifying any other wires is already met; and so mirroring the circuit about the Toffoli gate would be redundant.

Basic Algorithm

So, how does this oracle solve the maximum independent set problem? Running the oracle inside the Grover simulator will tell us what test sets match a *particular* threshold. But we are interested in the *maximum* threshold. So, one possible algorithm would start with a large threshold value and decrease it until Grover returns a result. We would then be sure that we found the largest (and thus maximum) independent set.

Another approach would be to use a binary search algorithm that would start with a threshold somewhere in the "middle" and progressively hone in on the maximum threshold value.

A third approach would be to start at zero and increase the threshold until no more independent sets are found. This last approach was used in this project even though with such a trivially small graph this was not technically advantageous in any way.

Results

This section shows the results of running the maximum independent set oracle in the Grover simulator. The threshold value is not an input to the oracle, so a slightly different oracle was built for each possible threshold value and the Grover simulator re-ran. The results showed that the oracle successfully found *all* the possible independent sets at every threshold value. As the threshold value passes the size of the actual maximum independent set in the graph, we observe that nothing is found. Based on the algorithm described above, we can declare that the last threshold value is whatever the last threshold value to return a result was.

For the raw output from the algorithm, please refer to *main.m*.

Threshold Size Zero

When searching for a threshold size of zero, the empty set is returned. This was expected. The following shows a graph with no nodes chosen.

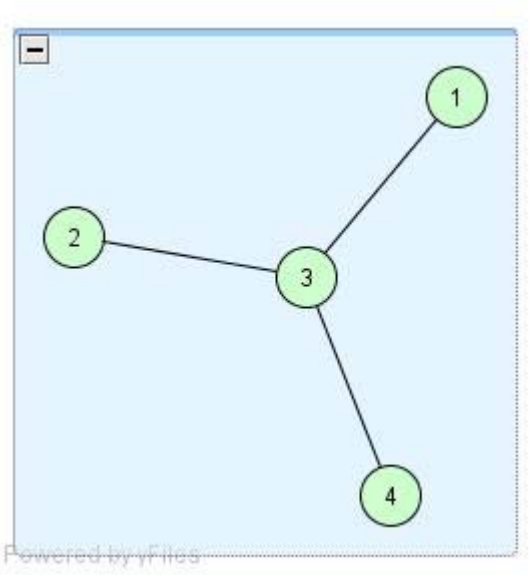

Figure 11. Result Size Zero

The Grover algorithm gave the probability of observing this graph on the output as 0.512. The next most probable output was 0.033. There were no extra Grover iterations.

Threshold Size One

When searching for a threshold size of one, four sets were returned. This was expected. The following shows the chosen sets.

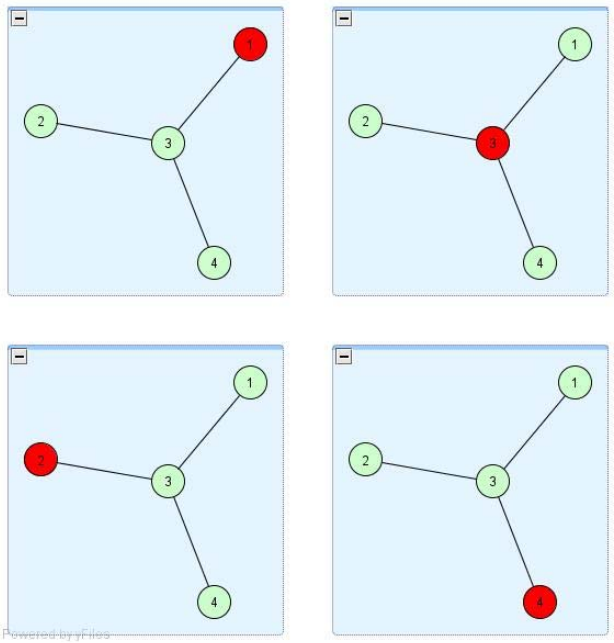

Figure 12. Result Size One

The Grover algorithm gave equal probability of observing each of these sets with a value of 0.156. The next most probable output was 0.031. There was one extra Grover iteration needed to produce this result.

Threshold Size Two

When searching for a threshold size of two, three sets were returned. This was expected. The following shows the chosen sets.

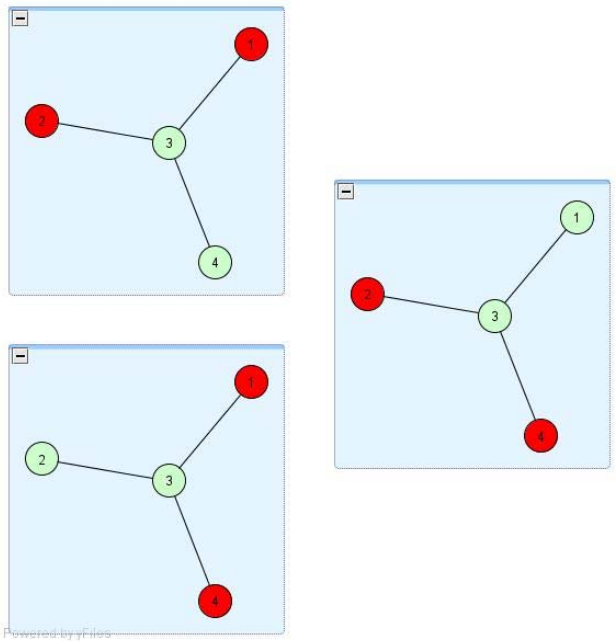

Figure 13. Result Size Two

The Grover algorithm gave equal probability of observing each of these sets with a value of 0.190. The next most probable output was 0.033. There were two extra Grover iterations needed to produce this result.

Threshold Size Three

When searching for a threshold size of three, one set was returned. This was expected. The following shows the chosen set.

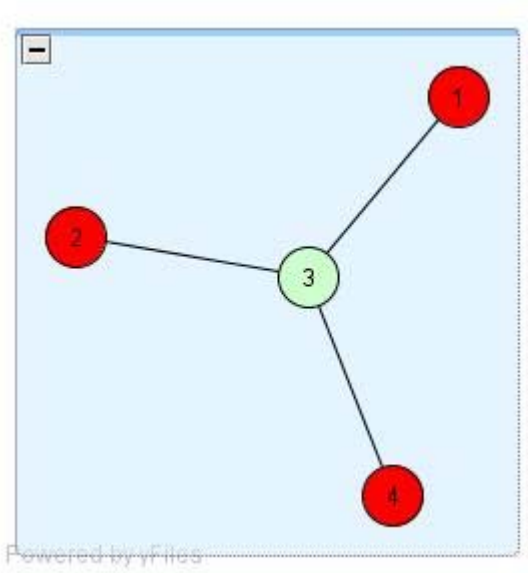

Figure 14. Result Size Thee

The Grover algorithm gave the probability of observing the chosen set with a value of 0.322. The next most probable output was 0.045. There was one extra Grover iteration needed to produce this result.

Threshold Size Four

When searching for a threshold size of four, no results with any appreciable probability were returned. This was expected. The Grover algorithm gave equal probability to all its (incorrect) outputs as 0.062. Adjusting the number of extra Grover iterations did not change these results.

Performance

The performance of the simulator had a big impact on this project and was a limiting factor for creating more complex circuits. Ideally it would have been nice to simulate the algorithm on a larger graph to see if the algorithm properly extrapolated. There seemed to be a wall at about 9-10 qubits before the simulations took many minutes to complete instead of a few seconds.

The build-in profiler in Matlab was used to hone in on the problem. The reason for the wall had to do with the size of matrices being computed. Each matrix for a circuit of n qubits has a size of $2^n \times 2^n$. The total number of elements in this matrix is $(2^n)^2 = 4^n$ giving a space complexity of $O(4^n)$.

A dot-product performs n multiplications for every element in the matrix. This means the number of multiplications to compute a dot product of is $n4^n$ meaning the time complexity is $O(n4^n)$. Clearly, this was the cause of the wall I was experiencing.

To partially alleviate this problem, the sparse matrix feature of Matlab was leveraged to allow the creation and computation of the dot product of much larger matrices. The confounding factor was the $n \times n$ Hadamard matrix used inside of Grover. This matrix is huge and also *not* sparse, so turning on the sparse matrix feature does not help here. This seems to make intuitive sense because what the Grover algorithm is doing leveraging the entangled state of all qubits to perform massively parallel computations. So the fact that simulating this is hard is not surprising.

Conclusion

Much was learned about quantum computing in this assignment. I feel I have a better understanding of the kind of problem contexts this technology will be well suited for, as well as some of the challenges and limitations associated with designing a quantum computer. Admittedly, even though my Grover algorithm implementation seemed to function correctly, some of the higher level concepts of why the Grover algorithm works are still elusive to me.

This project was time consuming mainly due to development of the supporting framework. With more time and computing resources, it would have been possible to adapt the code to simulate a slightly larger graph, perhaps 5 or 6 nodes large. Also, by playing around with the adder it would be possible to compute an inequality which would make the threshold value a little more interesting.

The Grover algorithm is one technique for doing quantum computing, and perhaps other techniques address this differently, but it seems that creating circuits in the form of oracles does not lead

to the greatest control over the structure of algorithms. For example, it is unclear how control flow, or memory, or I/O would be implemented.

Code

<attached>

```
function [ result ] = bubblesort( data, funcmphand, stop_after )
%BUBBLESORT bubble sort
%   data is a cell array of
%   sort the data using comparator funcmphand
%   stop_after is the number of elements to stop sorting after

    s = size(data,2);
    for i=1:(s-1)
        if i > stop_after
            break
        end
        for j=(i+1):s
            d1 = data{i};
            d2 = data{j};
            if funcmphand(d1,d2) < 0
                data{i} = d2;
                data{j} = d1;
            end
        end
    end
    result = data;
end
```

Published with MATLAB® 7.4

```
function [ result ] = collapse_gate( mat )
%COLLAPSE_GATE collapse a list of matricies into a single matrix.
%   multiply all component matricies together (in reverse order) into a
%   single matrix.

    m = mat_wire(log2(size(mat{1,1}.ptr,1)));
    for i=1:size(mat,2)
        m = mat{1,i}.ptr * m;
    end
    result = m;
end
```

Published with MATLAB® 7.4

```
function [ u ] = format_prob_vec( v, n )
%FORMAT_PROB_VEC pretty print output vector
%   format the merged output vector into pretty string
%   v is a horizontal cell array of structures

    f = strcat('%',int2str(n+1),'s --> %4.3f');
    u = '';
    for i = 1:size(v,2)
        b = v{i}.bits;
        p = v{i}.prob;
        s = sprintf(f, b, p);
        u = strvcat(u, s);
    end
    result = u;
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_four_bit_counter()
%GATE_FOUR_BIT_COUNTER four bit counter
%   produces a 3-bit number for count of input bits with value 1.
%   returns a 2^8 by 2^8 matrix with the following configuration
%            Input       Output
%   1 -      e1
%   2 -      e2
%   3 -      e3
%   4 -      e4            R0
%   5 -      |0>
%   6 -      |0>
%   7 -      |0>           R1
%   8 -      |0>           Cout

    n = 8;

    % a cell array of pointers. pointers are used because matlab doesn't
    %   know how to store sparse arrays in cells. making the strucutres
    %   gets around this problem.
    m = {};

    m = horzcat(m, subgate(n, gate_half_adder(), 1,2,5));
    m = horzcat(m, subgate(n, gate_half_adder(), 3,4,6));
    m = horzcat(m, subgate(n, gate_half_adder(), 2,4,7));
    m = horzcat(m, subgate(n, gate_full_adder(), 5,6,7,8));

    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_full_adder()
%GATE_FULL_ADDER full adder
%   returns a 2^4 by 2^4 matrix for full adder.
%   wires:
%            Input       Output
%       1 -   a           a
%       2 -   b           b
%       3 -  |0>          Sum
%       4 -  |0>          Carry

    n = 4;

    % a cell array of pointers. pointers are used because matlab doesn't
    %   know how to store sparse arrays in cells. making the strucutres
    %   gets around this problem.
    m = {};

    m = horzcat(m, subgate(n, gate_toffoli(3), 2, 3, 4));
    m = horzcat(m, subgate(n, gate_toffoli(2), 2, 3));
    m = horzcat(m, subgate(n, gate_toffoli(3), 1, 3, 4));
    m = horzcat(m, subgate(n, gate_toffoli(2), 1, 3));

    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_graph_indepset_n4e3_r1()
%GATE_GRAPH_INDEPSET_N4E3_R1 graph with 4 nodes and 3 edges
%   edges:  1,3  2,3  3,4
%   returns a 2^8 by 2^8 matrix with top 5 wires input and bottom
%   five being |0> ancilla bits and last being the single bit output.
%           Input       Output
%   1 -      e1           e1
%   2 -      e2           e2
%   3 -      e3           e3
%   4 -      e4           e4
%   5 -      |0>          |0>
%   6 -      |0>          |0>
%   7 -      |0>          |0>
%   8 -      |0>         x XOR result

    n = 8;

    % a cell array of pointers. pointers are used because matlab doesn't
    %   know how to store sparse arrays in cells. making the strucutres
    %   gets around this problem.
    m1 = {};
    m2 = {};

    % edge 1,3 --> working bit 5
    m1 = horzcat(m1, subgate(n, gate_nand(), 1,3,5));
    % edge 2,3 --> working bit 6
    m1 = horzcat(m1, subgate(n, gate_nand(), 2,3,6));
    % edge 3,4 --> working bit 7
    m1 = horzcat(m1, subgate(n, gate_nand(), 3,4,7));

    % toffoli gate anding all edges together
    m2 = horzcat(m2, subgate(n, gate_toffoli(4), 5,6,7,8));

    m = horzcat(m1, m2, mirror_gate(m1));
    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_half_adder()
%GATE_HALF_ADDER half adder
%   returns a 2^3 by 2^3 matrix for half adder.
%   wires:
%            Input       Output
%       1 -   a           a
%       2 -   b          Sum
%       3 -  |0>         Carry

    n = 3;

    % a cell array of pointers. pointers are used because matlab doesn't
    %   know how to store sparse arrays in cells. making the strucutres
    %   gets around this problem.
    m = {};

    m = horzcat(m, subgate(n, gate_toffoli(3), 1,2,3));
    m = horzcat(m, subgate(n, gate_toffoli(2), 1,2));

    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_inverter()
%GATE_INVERTER inverter gate (1-input, 1-output)
%   inverts a value

    result = { wrapptr([ 0 1 ; 1 0 ]) };
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_nand()
%GATE_NAND nand gate
%   returns an 8x8 matrix with top two wires being inputs and bottom wire
%   being result XOR'ed to the input of that wire.

    n = 3;

    m = {};

    m = horzcat(m, subgate(n, gate_inverter(), 1));
    m = horzcat(m, subgate(n, gate_toffoli(2), 1,3));
    m = horzcat(m, subgate(n, gate_inverter(), 1));

    m = horzcat(m, subgate(n, gate_inverter(), 2));
    m = horzcat(m, subgate(n, gate_toffoli(3), 1,2,3));
    m = horzcat(m, subgate(n, gate_inverter(), 2));

    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_three_bit_comparator()
%GATE_THREE_BIT_COMPARATOR compare two 3-bit bit strings
%   returns a 2^6 by 2^6 matrix with the following configuration
%           Input       Output
%   1 -       a0          a0
%   2 -       a1          a1
%   3 -       a2          a2
%   4 -       b0          XOR(a0,b0)
%   5 -       b1          XOR(a1,b1)
%   6 -       b2          XOR(a2,b2)

    n = 6;

    % a cell array of pointers. pointers are used because matlab doesn't
    %   know how to store sparse arrays in cells. making the strucutres
    %   gets around this problem.
    m = {};

    m = horzcat(m, subgate(n, gate_toffoli(2), 1,4));
    m = horzcat(m, subgate(n, gate_inverter(), 4));

    m = horzcat(m, subgate(n, gate_toffoli(2), 2,5));
    m = horzcat(m, subgate(n, gate_inverter(), 5));

    m = horzcat(m, subgate(n, gate_toffoli(2), 3,6));
    m = horzcat(m, subgate(n, gate_inverter(), 6));

    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_threshold( b3, b2, b1 )
%GATE_THRESHOLD count 4 bits and compare result with another 3
%   counts 1-bits and compares to 3-bit number.
%   returns a 2^12 by 2^12 matrix with the following configuration
%            Input       Output
%   1 -       e1           e1
%   2 -       e2           e2
%   3 -       e3           e3
%   4 -       e4           e4
%   5 -       |0>          |0>
%   6 -       |0>          |0>
%   7 -       |0>          |0>
%   8 -       |0>          |0>
%   9 -       |0>          |0>
%   10 -      |0>          |0>
%   11 -      |0>          |0>
%   12 -      x          x XOR result

    n = 12;

    m1 = {};
    m2 = {};

    % put inverters on the wires for each of the 1 'b' bits.
    if b1
        m1 = horzcat(m1, subgate(n, gate_inverter(), 5));
    end
    if b2
        m1 = horzcat(m1, subgate(n, gate_inverter(), 6));
    end
    if b3
        m1 = horzcat(m1, subgate(n, gate_inverter(), 7));
    end

    m1 = horzcat(m1, subgate(n, gate_four_bit_counter(), 1,2,3,4,8,9,10,11)); % 4,7,8 -> 4,10,11
    m1 = horzcat(m1, subgate(n, gate_three_bit_comparator(), 4,10,11,5,6,7));
    m2 = horzcat(m2, subgate(n, gate_toffoli(4), 5,6,7,12));

    result = horzcat(m1, m2, mirror_gate(m1));
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_toffoli( n )
%GATE_TOFFOLI toffoli gate (n-inputs, n-outputs)
%   returns a 2^n by 2^n matrix for a toffoli gate

    s = 2^n;
    r = speye(s);
    if n > 1
        r(s-1, s-1) = 0;
        r(s, s) = 0;
        r(s, s-1) = 1;
        r(s-1, s) = 1;
    end
    result = { wrapptr(r) };
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_wire( n )
%GATE_WIRE wire gate (n-input, n-output)
%   returns a 2^n by 2^n gate for a set of plain wires

    result = { wrapptr(mat_wire(n)) };
end
```

Published with MATLAB® 7.4

```
function [result, niter] = grover(oracle, n, niterExtra)
%GROVER grover algorithm
%   apply grover algorithm to oracle matrix with n input qubits.
%   return final output vector and number of grover iterations.
%   all work qubits are initialized to zero.
%   oracle matrix dimensions encompass input, oracle, and work qubits.
%   output vector is t by 1 where t is one dimension of oracle matrix
%       (oracle matrix is square).

    % figure out how many qubits we need for this calculation (this is t)
    t = log2(size(oracle,1));

    % start off with all zeros for input, oracle, and working bits
    % v is the vector we will be operating on
    v = kronpow(qubit(0),t);

    % compute hadamard matrix
    mat_had_n = kronpow((1/sqrt(2)) .* [ 1 1 ; 1 -1 ], n);

    % compute the inital circuit that applies hadamards to all the
    % inputs but not the oracle and work qubits
    hadamards_inputs_circuit = kron(mat_had_n, mat_wire(t-n));

    % apply the inital hadamard circuit to the vector v
    v = hadamards_inputs_circuit * v;

    % compute the grover circuit we will apply to v on each iteration
    h = kron(mat_had_n, mat_wire(t-n));
    z = kron(mat_zsps(n), mat_wire(t-n));
    grover_circuit = h * (z * (h * oracle)); % takes a LONG time for inputs > 9

    % compute the number of times we need to apply grover
    niter = ceil(pi*sqrt(t)/4) + niterExtra;

    % do the grover iterations
    for i=1:niter
        v = grover_circuit * v;
    end

    % return the vector
    result = v;
end
```

Published with MATLAB® 7.4

```
function [ result ] = kronchain( varargin )
%KRONCHAIN kronecker product
%   apply kronecker product on multiple matricies (in order specified)

    m = varargin{1};
    for i=2:size(varargin,2)
        m = kron(m,varargin{i});
    end
    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = kronpow( mat, n )
%KRONPOW kronecker^n
%   apply a kronecker matrix operation n times

    k = [ 1 ];
    for i=1:n
        k = kron(mat, k);
    end
    result = k;
end
```

Published with MATLAB® 7.4

Contents

search for 0-node independent sets

```
clear;
num_inputs = 4;
oracle = oracle_indep_set(0,0,0);
printresults(grover(oracle, num_inputs, 0), num_inputs, 10)
```

```
ans =

 0000 --> 0.512
 1100 --> 0.033
 1010 --> 0.033
 1001 --> 0.033
 1101 --> 0.033
 1111 --> 0.033
 1000 --> 0.033
 0011 --> 0.033
 0101 --> 0.033
 0110 --> 0.033
```

search for 1-node independent sets

```
clear;
num_inputs = 4;
oracle = oracle_indep_set(0,0,1);
printresults(grover(oracle, num_inputs, 1), num_inputs, 10)
```

```
ans =

 0001 --> 0.156
 1000 --> 0.156
 0100 --> 0.156
 0010 --> 0.156
 1110 --> 0.031
 1111 --> 0.031
 1101 --> 0.031
 1100 --> 0.031
 0101 --> 0.031
 0011 --> 0.031
```

search for 2-node independent sets

```
clear;
num_inputs = 4;
oracle = oracle_indep_set(0,1,0);
printresults(grover(oracle, num_inputs, 2), num_inputs, 10)
```

```
ans =

 0101 --> 0.190
 1001 --> 0.190
 1100 --> 0.190
 1101 --> 0.033
 1000 --> 0.033
```

```
 1011 --> 0.033
 1110 --> 0.033
 0111 --> 0.033
 0100 --> 0.033
 1010 --> 0.033
```

search for 3-node independent sets

```
clear;
num_inputs = 4;
oracle = oracle_indep_set(0,1,1);
printresults(grover(oracle, num_inputs, 1), num_inputs, 10)
```

```
ans =

 1101 --> 0.322
 1011 --> 0.045
 1100 --> 0.045
 1010 --> 0.045
 1110 --> 0.045
 1111 --> 0.045
 0110 --> 0.045
 1001 --> 0.045
 1000 --> 0.045
 0001 --> 0.045
```

search for 4-node independent sets

```
clear;
num_inputs = 4;
oracle = oracle_indep_set(1,0,0);
printresults(grover(oracle, num_inputs, 0), num_inputs, 10)
```

```
ans =

 1111 --> 0.062
 1101 --> 0.062
 1110 --> 0.062
 1100 --> 0.062
 0111 --> 0.062
 1011 --> 0.062
 0001 --> 0.062
 0000 --> 0.062
 1000 --> 0.062
 1001 --> 0.062
```

Published with MATLAB® 7.4

```
function [ result ] = mat_move_wire( n, from, to )
%MAT_MOVE_WIRE move a wire to a different position
%   n    - total number of wires
%   from - starting position of wire
%   to   - ending position of wire

    m = mat_wire(n);
    if from < to
        % walk forward
        for i = from:(to-1)
            t1 = mat_wire(i-1);
            t2 = mat_swap();
            t3 = mat_wire(n-(i+1));
            m = kronchain(t1, t2, t3) * m;
        end
    elseif from > to
        % walk backward
        for i = from:-1:(to+1)
            t1 = mat_wire(i-2);
            t2 = mat_swap();
            t3 = mat_wire(n-i);
            m = kronchain(t1, t2, t3) * m;
        end
    end
    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = mat_swap()
%MAT_SWAP swap gate (2-input, 2-output)
%   returns a 4 by 4 matrix for swapping two qubits

    result = [ 1 0 0 0 ; 0 0 1 0 ; 0 1 0 0 ; 0 0 0 1 ];
end
```

Published with MATLAB® 7.4

```
function [ result ] = mat_swap_wires( n, wire1, wire2 )
%MAT_SWAP_WIRES swap wires
%   swap position of 2 wires
%   n     - total number of wires in circuit
%   wire1 - position of wire1
%   wire2 - position of wire2

    m = mat_wire(n);

    if wire1 < wire2
        m = mat_move_wire(n, wire1, wire2) * m; % wire2 now in wire2-1 pos
        m = mat_move_wire(n, wire2-1, wire1) * m;
    elseif wire1 > wire2
        m = mat_move_wire(n, wire2, wire1) * m; % wire1 now in wire1-1 pos
        m = mat_move_wire(n, wire1-1, wire2) * m;
    end
    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = mat_wire( n )
%MAT_WIRE matrix for 'n' wires
%   return a 2^n by 2^n matrix (sparse) for n wires. this is the same as
%   the identity matrix.

    result = speye(2^max(n,0));
end
```

Published with MATLAB® 7.4

```
function [ result ] = gate_zsps( n )
%GATE_ZSPS zero-state phase shift gate (n-input, n-output)
%   returns a 2^n by 2^n matrix for a zero-state phase shift gate

    m = mat_wire(n);
    m(1,1) = -1;

    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = merge_probs( v, n )
%MERGE_PROBS merge probabilities of grover output vector
%   merge non-input qubits in output vector (v) together and return a vector
%   of probabilities for just the n input qubits.
%   returns a horizontal cell array of structures.

    t = log2(size(v,1));
    u = {};
    for i = 1:(2^n)
        s = 0;
        for j = 1:(2^(t-n))
            idx = (i-1)*(2^(t-n)) + j;
            s = s + abs(v(idx))^2;
        end
        r.bits = dec2bin(i-1,n);
        r.prob = s;
        u = [ u r ];
    end
    result = u;
end
```

Published with MATLAB® 7.4

```
function [ result ] = mirror_gate( gate )
%MIRROR_GATE compute mirror of gate
%   return the mirror of the gate parameter. the mirror is the same as the
%   gate itself but with all its element matricies in reverse order.

    m = {};
    for i=size(gate,2):-1:1
        m = horzcat(m, gate{1,i});
    end
    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = oracle_and( n )
%ORACLE_AND oracle that ANDs inputs
%   returns matrix is 2^(n+1) by 2^(n+1) for the oracle performing
%   the AND operation. the last bit is the oracle bit. there are no
%   working bits.
%   NOTE: used to test the grover algorithm functions correctly

    result = collapse_gate(gate_toffoli(n+1));
end
```

Published with MATLAB® 7.4

```
function [ result ] = oracle_indep_set( b3, b2, b1 )
%ORACLE_INDEP_SET oracle for finding independent sets of given size
%   return an oracle (matrix) for testing whether a given independent set
%   if of a given size specified by the encoded 3-bit input.
%   returns a 2^13 by 2^13 matrix with the following configuration
%            Input       Output
%   1 -       e1           e1
%   2 -       e2           e2
%   3 -       e3           e3
%   4 -       e4           e4
%   5 -       |0>          |0>
%   6 -       |0>          |0>
%   7 -       |0>          |0>
%   8 -       |0>          |0>
%   9 -       |0>          |0>
%   10 -      |0>          |0>
%   11 -      |0>          |0>
%   12 -      |0>          |0>
%   13 -      x           x XOR result

    n = 13;

    % a cell array of pointers. pointers are used because matlab doesn't
    %   know how to store sparse arrays in cells. making the strucutres
    %   gets around this problem.
    m1 = {};
    m2 = {};

    m1 = horzcat(m1, subgate(n, gate_threshold(b3, b2, b1), ...
        1,2,3,4,5,6,7,8,9,10,11,12));
    m1 = horzcat(m1, subgate(n, gate_graph_indepset_n4e3_r1(), ...
        1,2,3,4,5,6,7,8));
    m2 = horzcat(m2, subgate(n, gate_toffoli(3), 8,12,13));

    % build circuit
    m = horzcat(m1, m2, mirror_gate(m1));

    % take the output and put in on line 5, right below the inputs
    m = subgate(n, m, 1,2,3,4,6,7,8,9,10,11,12,13,5);

    % return the result
    result = collapse_gate(m);
end
```

Published with MATLAB® 7.4

```
function [ result ] = oracle_just_comparator( b3, b2, b1 )
%ORACLE_JUST_COMPARATOR test oracle for gate_three_bit_comparator

    n = 7;

    m1 = {};
    m2 = {};

    % put inverters on the wires for each of the 1 'b' bits.
    if b1
        m1 = horzcat(m1, subgate(n, gate_inverter(), 4));
    end
    if b2
        m1 = horzcat(m1, subgate(n, gate_inverter(), 5));
    end
    if b3
        m1 = horzcat(m1, subgate(n, gate_inverter(), 6));
    end

    m1 = horzcat(m1, subgate(n, gate_three_bit_comparator(), 1,2,3,4,5,6));
    m2 = horzcat(m2, subgate(n, gate_toffoli(4), 4,5,6,7));

    m = horzcat(m1, m2, mirror_gate(m1));
%     m = subgate(n, m, 1,2,3,5,6,7,4);

    % return result
    result = collapse_gate(m);
end
```

Published with MATLAB® 7.4

```
function [ result ] = oracle_just_counter_comp(b3, b2, b1)
%ORACLE_JUST_COUNTER_COMP test oracle for gate_threshold

    n = 12;

    m = gate_threshold(b3, b2, b1);
    m = subgate(n, m, 1,2,3,4,6,7,8,9,10,11,12,5);

    % return result
    result = collapse_gate(m);
end
```

Published with MATLAB® 7.4

```
function [ result ] = oracle_just_graph()
%ORACLE_JUST_GRAPH test oracle for gate_graph_indepset_n4e3_r1

    % get graph gate
    m = gate_graph_indepset_n4e3_r1();

    % take last bit and put it in position 5
    m = collapse_gate(subgate(8, m, 1,2,3,4,6,7,8,5));

    % return result
    result = m;
end
```

Published with MATLAB® 7.4

```
function [ result ] = oracle_nor( n )
%ORACLE_NOR oracle that NORs inputs
%   returns matrix is 2^(n+1) by 2^(n+1) for the oracle performing
%   the NOR operation. the last bit is the oracle bit. there are no
%   working bits.
%   NOTE: used to test the grover algorithm functions correctly

    m1 = {};
    m2 = {};

    for i=1:n
        m1 = horzcat(m1, subgate(n+1, gate_inverter(), i));
    end
    m2 = horzcat(m2, subgate(n+1, gate_toffoli(n+1), 1:n+1));

    m = horzcat(m1, m2, mirror_gate(m1));
    result = collapse_gate(m);
end
```

Published with MATLAB® 7.4

```
function [ result ] = printresults( v, n, t )
%PRINTRESULTS print results vector v
%   v is the results vector (size 2^t by 1).
%   n is the number of input qubits (n < t).
%   t is the number of results to output

    % merge the probabilites into single vector of just inputs
    merged = merge_probs(v,n);

    % number of entries to print
    num_print = min(t,2^n);

    function [ result ] = cmp_probs(r1,r2)
        if r1.prob < r2.prob
            result = -1;
        elseif r1.prob > r2.prob
            result = 1;
        else
            result = 0;
        end
    end

    % sort the array
    sorted = bubblesort(merged, @cmp_probs, num_print);

    % slice off the top few elements of the array
    sliced = sorted(1:num_print);

    % print out the top few results
    result = format_prob_vec(sliced,n);
end
```

Published with MATLAB® 7.4

```
function [ result ] = qubit( v )
%QUBIT a v-valued qubit
%   returns a 2 by 1 vector for qubit of value 0 or 1

    if v == 0
        z = [ 1 ; 0 ];
    else
        z = [ 0 ; 1 ];
    end
    result = z;
end
```

Published with MATLAB® 7.4

```
function [ result ] = subgate( varargin )
%SUBGATE use gate as sub gate as part of a containing gate
%   param n     - size containing gate
%   param g     - gate to use as sub gate
%   param 3...  - wires from containing gate to apply to sub gate.
%                   the number of wires must equal size of subgate.
%   returns a cell array of matrix pointers of size n-by-n.

    % n - size of circuit
    n = varargin{1};
    % g - sub gate (cell array of matrix pointers)
    g = varargin{2};
    % p - all inputs to gate
    p = [varargin{1,3:size(varargin,2)}];
    % t - input gate matrix
    t = size(p,2); % number of inputs to gate
    % q - size of sub-gate
    q = log2(size(g{1,1}.ptr,1));

    % check dimensions of gate match input count
    if q ~= size(p,2)
        error 'gate size mismatch with input count';
    end
    if q > n
        error 'subgate larger than containing gate';
    end

    m1 = {}; % in circuit order
    m2 = {}; % in circuit order

    % shuffle bits into topmost position
    for i=1:t
        if p(i) ~= i
            % move position p(i) into position i; swap p(i) and i
            m1 = horzcat(m1, wrapptr(mat_swap_wires(n,p(i),i)));
            for j=(i+1):t
                % if i contained one of our elements, rename it in p
                if p(j) == i
                    p(j) = p(i);
                end
            end
        end
    end


    % append the sub-gate matricies expanded to fit containing circuit
    for i=1:size(g,2)
        m2 = horzcat(m2, wrapptr(kron(g{1,i}.ptr, mat_wire(n-q))));
    end


%     % undo the shuffling from above (apply swaps in reverse order)
%     for i=size(swaps,2):-1:1
%         op = swaps{i};
%         m = horzcat(m, wrapptr(mat_swap_wires(n,op.pos1,op.pos2)));
%     end

    result = horzcat(m1, m2, mirror_gate(m1));
end
```

Published with MATLAB® 7.4

Contents

grover just counter comparator

```
clear;
num_inputs = 4;
oracle = oracle_just_counter_comp(0,1,0);
printresults(grover(oracle, num_inputs, 2), num_inputs, 10)
```

```
ans =

 0101 --> 0.088
 1001 --> 0.088
 0011 --> 0.088
 1010 --> 0.088
 0110 --> 0.088
 1100 --> 0.088
 1110 --> 0.047
 1111 --> 0.047
 1101 --> 0.047
 0100 --> 0.047
```

grover just comparator

```
clear;
num_inputs = 3;
oracle = oracle_just_comparator(1,1,0);
printresults(grover(oracle, num_inputs, -1), num_inputs, 10)
```

```
ans =

 011 --> 0.535
 010 --> 0.066
 101 --> 0.066
 001 --> 0.066
 100 --> 0.066
 000 --> 0.066
 110 --> 0.066
 111 --> 0.066
```

grover just graph

```
clear;
num_inputs = 4;
oracle = oracle_just_graph();
printresults(grover(oracle, num_inputs, 1), num_inputs, 10)
```

```
ans =
```

```
 0100 --> 0.084
 0101 --> 0.084
 1001 --> 0.084
 1000 --> 0.084
 0000 --> 0.084
 1100 --> 0.084
 1101 --> 0.084
 0001 --> 0.084
 0010 --> 0.084
 0110 --> 0.035
```

grover NOR

```
clear;
num_inputs = 5;
oracle = oracle_nor(num_inputs);
printresults(grover(oracle, num_inputs, 2), num_inputs, 10)
```

```
ans =

 00000 --> 0.515
 11010 --> 0.016
 11001 --> 0.016
 11111 --> 0.016
 11110 --> 0.016
 11101 --> 0.016
 11100 --> 0.016
 10111 --> 0.016
 11000 --> 0.016
 11011 --> 0.016
```

grover AND

```
clear;
num_inputs = 5;
oracle = oracle_and(num_inputs);
printresults(grover(oracle, num_inputs, 2), num_inputs, 10)
```

```
ans =

 11111 --> 0.515
 10000 --> 0.016
 10010 --> 0.016
 10110 --> 0.016
 10001 --> 0.016
 11101 --> 0.016
 11110 --> 0.016
 00101 --> 0.016
 00100 --> 0.016
 00000 --> 0.016
```

test half adder

```
clear;
n = 3;
a = kronchain(qubit(1),qubit(1),qubit(0));
b = mat_swap_wires(n, 2, 3) * collapse_gate(subgate(n, gate_half_adder(), 1:n)) * a;
strvcat(...
    printresults(a, n, 1),...
    printresults(b, n, 1))
```

```
ans =
```

```
 110 --> 1.000
 110 --> 1.000
```

test full adder

```
clear;
n = 4;
a = kronchain(qubit(1),qubit(1),qubit(1),qubit(0));
b = mat_swap_wires(n, 3, 4) * collapse_gate(subgate(n, gate_full_adder(), 1:n)) * a;
strvcat(...
    printresults(a, n, 1),...
    printresults(b, n, 1))
```

```
ans =

 1110 --> 1.000
 1111 --> 1.000
```

test counter

```
clear;
n = 8;
a = kronchain(qubit(0),qubit(1),qubit(1),qubit(0),qubit(0),qubit(0),qubit(0),qubit(0));
b = mat_move_wire(n, 6,7) * ...
    mat_move_wire(n, 4,8) * ...
    collapse_gate(subgate(n, gate_four_bit_counter(), 1:n)) * a;
strvcat(...
    printresults(a, n, 1),...
    printresults(b, n, 1))
```

```
ans =

 01100000 --> 1.000
 01100010 --> 1.000
```

test comparator itself

```
clear;
n = 6;
a = kronchain(qubit(0),qubit(0),qubit(1),qubit(0),qubit(0),qubit(1));
b = collapse_gate(subgate(n, gate_three_bit_comparator(), 1:n)) * a;
strvcat(...
    printresults(a, n, 1),...
    printresults(b, n, 1))
```

```
ans =

 001001 --> 1.000
 001111 --> 1.000
```

test comparator oracle

```
clear;
n = 7;
a = kronchain(qubit(0),qubit(1),qubit(1),qubit(0),qubit(0),qubit(0),qubit(0));
b = oracle_just_comparator(0,1,1) * a;
strvcat(...
    printresults(a, n, 1),...
    printresults(b, n, 1))
```

```
ans =

 0110000 --> 1.000
 0110000 --> 1.000
```

Published with MATLAB® 7.4

```
function [ result ] = wrapptr( input )
%WRAPPTR wrap input in a pointer structure
%   wrap input in a structure with a single field called 'ptr'.
%   this is used to get around matlab limitation of not being
%   able to store sparse arrays in cell arrays.

    r.ptr = input;
    result = r;
end
```

Published with MATLAB® 7.4