Grover's Algorithm

Introduction

Grover's algorithm is a quantum search algorithm that was developed by Lor Grover in 1996. The purpose of this algorithm is to find a unique input to some function (a "black box") that gives a particular output. In other words, Grover's algorithm finds a marked item out of some search space.

The usefulness of this algorithm is that it applies to various NP-complete problems and does so in a faster time than a classical computation. NP complete refers to the set of decision problems (problems that can be posed as yes-or-no questions) whose solutions can be checked in polynomial time. Classically, we expect an unstructured search problem to be solved in $O(N)$ evaluations while the Grover's algorithm can be solved in $O(\sqrt{N})$ evaluations, which is significant as N gets larger, using principles of quantum mechanics—particularly superposition.

I will first go into more detail of this algorithm and then show how it applies to the specific problem that I have chosen to solve.

The Algorithm

The general idea behind this algorithm is that when one is given a particular search space, this algorithm can check whether a particular element in that search space is "correct". Examples would be having a list of elements (the search space) and checking if the particular element is equal to the one that you are searching for in the list of elements, or possible arrangements of people (the search space) and checking to see whether a particular arrangement of people satisfy the given constraints.

Gates

Below are images of the gates that are relevant in our circuit—their matrix representation and their visual representation, along with a short description of their essence.

Hadamard Gate

Symbol:

Matrix:

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

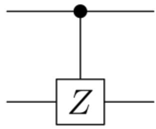
Description:

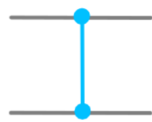
Acts on single qubits by putting them in superposition. It maps them as follows:

$$|0\rangle \mapsto \frac{|0\rangle + |1\rangle}{\sqrt{2}} \qquad |1\rangle \mapsto \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

CZ Gate

Symbol:





Matrix:

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

Description:

The CZ-gate is a Controlled-Z gate thus acting on 2 qubits. The mapping is:

$$|00> \rightarrow |00>$$

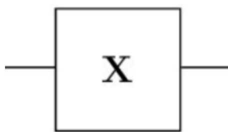
$$|01> \rightarrow |01>$$

$$|10> \rightarrow |10>$$

$$|11> \rightarrow -|11>$$

X (NOT) Gate

Symbol:



Matrix:

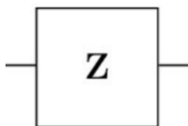
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Description:

The Pauli-X gate acts on a single qubit and rotates it around the x-axis of the Bloch Sphere by pi-radians. This is analogous to a NOT-gate for classical computers in which it flips the qubits—it maps the |0> basis ket to the |1> basis ket and the |1> basis ket to the |0> basis ket.

Z Gate

Symbol:



Matrix:

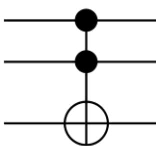
$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Description:

The Pauli-Z gate acts on a single qubit and rotates it as well. Specifically, it Pauli Z leaves the basis state |0> unchanged and maps |1> to -|1>. It is sometimes called phase-flip.

TOFFOLI/ CCNOT Gate

Symbol:



Truth table

INPUT			OUTPUT		
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

Matrix form

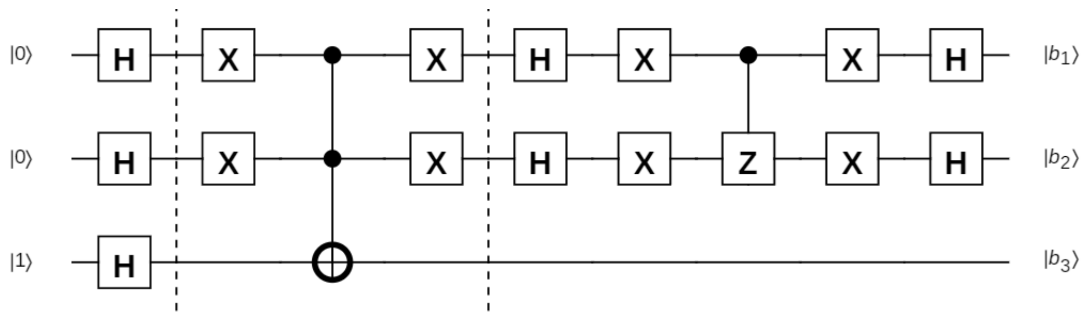
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Description:

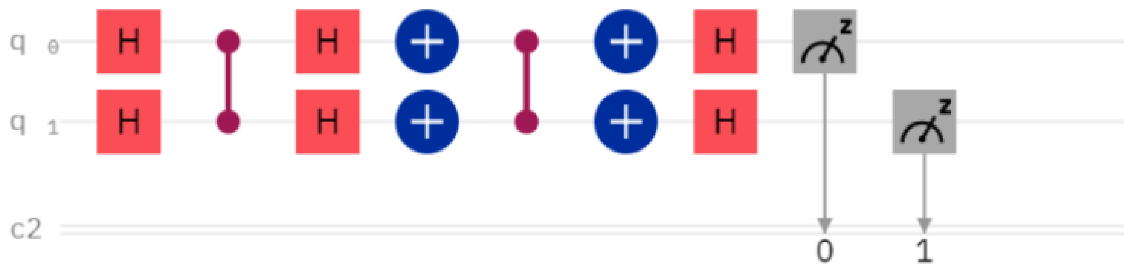
At the bottom of the truth table one can see how the two last rows have different output compared to their input as the qubit_3 changes from 0 to 1 or vice versa. This is the essence of the TOFFOLI Gate.

Full Circuit

While there are some variations of the algorithm as the combinations of various gates can lead to a similar state such as in the different use of CZ or Toffoli gates (analogously to classical programming where one can arrive to the same answer using different methods), the general algorithm is displayed below:



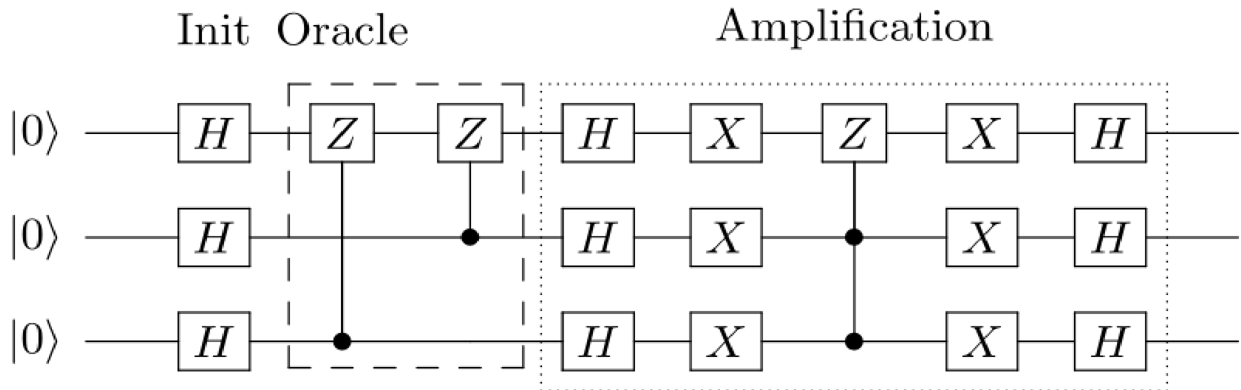
Alternative image of 2-qubit for comparison:



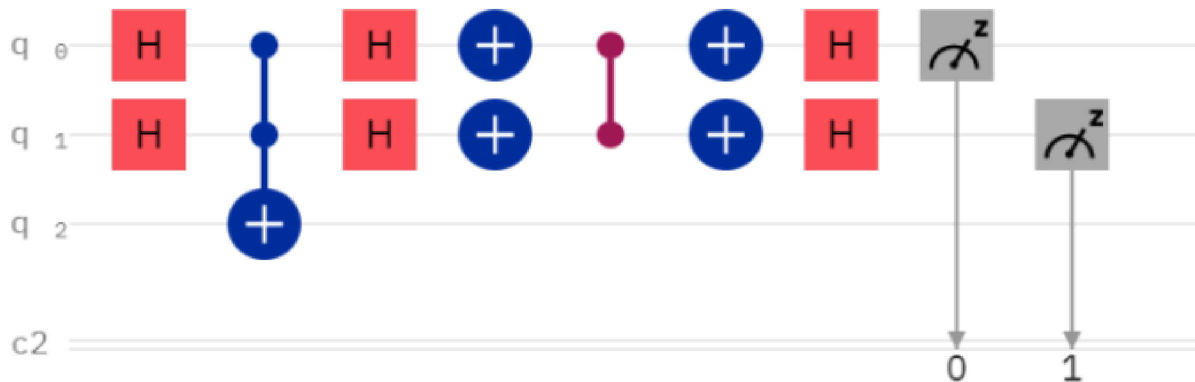
As one can see, the difference between the first and the second image is the placement of Pauli-X gates and a Toffoli gate on the above image versus a simple CZ gate on the lower image.

If we change the number of qubits or the number of iterations, similar patterns are observed therefore the image right above of the algorithm does not change too much. In particularly, altering the number of iterations simply duplicates (or triples etc.) the entire circuit, while altering the number of qubits extends another line similar as above, with the CZ and Toffoli connected to all qubits. One can see this when comparing the 2-qubit circuits above, with the three-qubit circuits below.

3-Qubit Circuit of Grover's Algorithm:



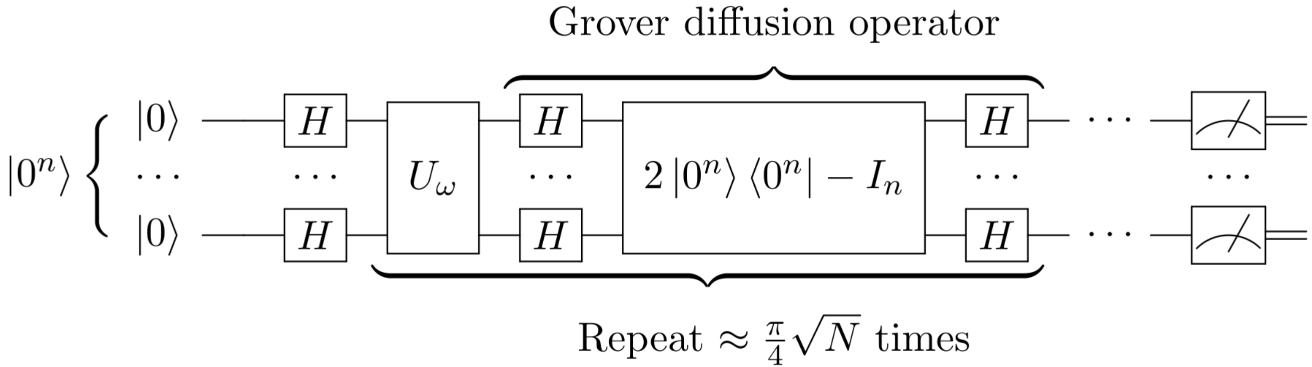
I will be using a slightly altered version of this circuit in the problems that I am trying to solve. The reason for this will be evident when I describe Problem #2.



The above is an alternative way of constructing a 3-qubit circuit. The difference between the previous 3-qubit circuit is that here they use a *Toffoli Gate* which naturally acts on 3 qubits (it is a 3 x 3 matrix).

How it works

Full General Algorithm:



Suppose you would like to locate the winner *w* from a list of *N* items.

After initializing the qubits into a uniform superposition over all states using the Hadamard gates, one then creates an Oracle which will add a negative phase to the solution state essentially marking the desired state.

This is represented with the U_w operator.

$$\begin{cases} U_\omega|x\rangle = -|x\rangle & \text{for } x = \omega, \text{ that is, } f(x) = 1, \\ U_\omega|x\rangle = |x\rangle & \text{for } x \neq \omega, \text{ that is, } f(x) = 0. \end{cases}$$

← The *f(x)* is a function that equals one if the proposed *x* is the winning state and 0 otherwise.

So, for example, if your winning/desired state *w* is the '11' qubit, then applying this operator, you will have:

$$U_\omega = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

← the |11> state

The next part of the circuit is doing amplitude amplification.

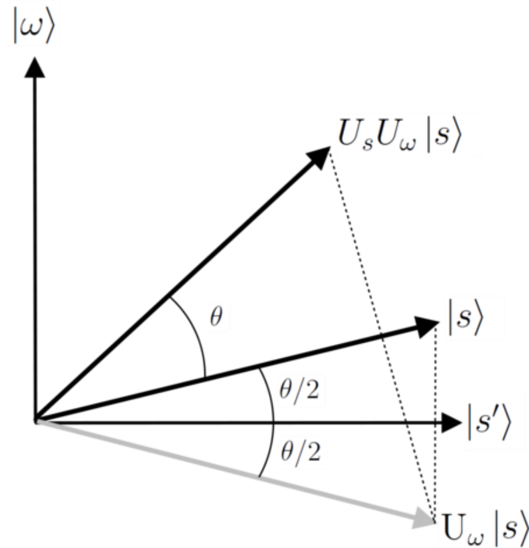
We begin with the superposition state:

$$|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$$

After applying the reflection operator U_w, we now have a state that has reflected across some hypothetical state vector $|s'>$—which we create to be orthogonal to our winning state $|w>$.

We then apply a second reflection, U_s, which reflects the $|s>$ back across the $|s'>$ axis and closer to the winning state, as the combination of two reflections create a rotation from the imaginary $|s'>$ state to the winning state. It is this action is what amplifies the amplitude of the state that we are attempting to find.

The above explanation is demonstrated visually (and is the geometric proof for the algorithm) in the image on the right →



The U_s operator is known as the *Diffusion Operator*:

$$U_s = 2|s\rangle\langle s| - 1$$

In summary, doing this process several times will bring the *s* state closer to our *w* winning state by amplifying the marked elements while the amplitude of the unmarked elements decreases. In fact, the number of such rotations are $\sqrt{N}$ times, hence speed up of Grover's search algorithm compared to classically.

Problems

Problem #1: Finding an element from a List

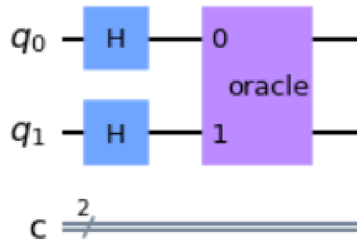
Before getting into a problem that shows an exact application of Grover's Algorithm, I want to show the basic idea of the algorithm by demonstrating the essence of this algorithm by using it to find an element from a list. This is the basics of Grover's Algorithm.

Right is an example of the result of the problem in classical terms. I have the list, from which the user picks a number and the oracle function identifies if the user's number matches the number in the list as it iterates through it. Classically, to find the number it will be on average *N/2* times, where *N* is the number of elements in the list.

```
[35, 48, 17, 82, 13, 65, 79, 26, 0, 99]

Please pick a number from the list:  2

Number selected is not in my list. Try again.

Please pick a number from the list:  17

Winner is found at index 2
3 calls to the Oracle used
```

In the code file I then proceed to show how one finds an element in a list using quantum computation. This will be analogous to the above example. The state we will search for is the |11> state. I build the circuit for Grover's Algorithm, and then display the probabilistic results.

First, I create the CZ gate which will be the Oracle for this 2-qubit example. I then also put Hadamard gates before the oracle to place the initial qubits in superposition. This is how this part of the circuit looks like:



After this part, I check to see what the state-vector is—which I expect it to be in a state of superposition. For 2 qubits it would be as:

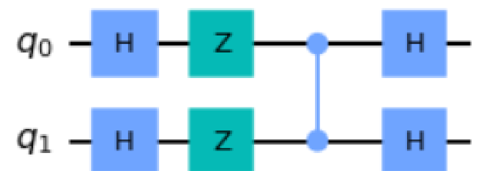
$$\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle)$$

Each super-positioned part will have a probability of 0.5, and the last state will have a negative sign representing the fact that our desired state was reflected (i.e. the purpose of the oracle (CZ gate)).

This is seen when I output the state-vector—the following array:

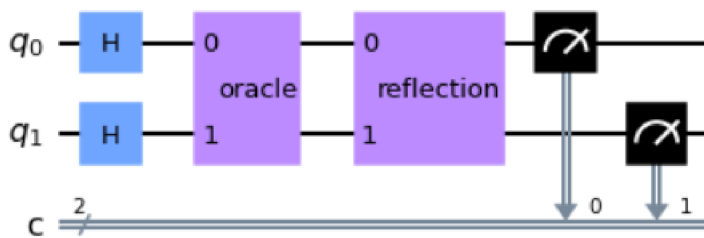
```
array([ 0.5+0.j,  0.5+0.j,  0.5+0.j, -0.5+0.j])
```

I then proceed to create the second part of the circuit which is the Diffuser Operator. Separately it looks as:



Solution to Problem #1

Putting all these pieces gives the full circuit:



Executing this circuit gives back the result of our desired state—the state we were searching for—in one call to the oracle.

```
{'11': 1}
```

A more direct example between a classical computation and a quantum search computation is seen in the next example which involves three qubits; however, the process is the same.

Problem #2: Arrangement of People

The problem that we wish to address an optimization problem called the *Satisfiability Problem*—problems that have specific conditions/constraints and you determine what is the most optimal outcome with those constraints.

We try to determine the best possible arrangement of friends with a set of constraints. The problem is:

Suppose you want to invite friends *A, B,* and *C*, to your dinner party. You order them two taxi's—taxi 0 and taxi 1. The constraints are that *A* and *B* are friends, while *A* and *C* are enemies, and *B* and *C* are also enemies. The question is: how do you maximize the friend pairs and minimizing enemy pairs in the taxi's?

To construct the solution, let's list the possible combinations. We can see all possible combinations if we write out in the following way:

[*A, B, C*] with each position holding 0 or 1. That is, if *A* is in taxi 0, *B* is in taxi 1 and *C* is in taxi 0, we can write [0, 1, 0].

All possible combinations:

[0, 0, 0] ; [1, 0, 0] ; [0, 1, 0] ; [0, 0, 1] ; [1, 1, 0] ; [1, 0, 1] ; [0, 1, 1] ; [1, 1, 1]

This is correct as $2^3 = 8$ possibilities.

Which of these combinations are acceptable?

Given the constraints, *A* and *C*, as well as *B* and *C* cannot be in the same taxi (since they are enemies) therefore if they have the same number, we can exclude these as satisfying possibilities.

By inspection, we see that there exist two possible states that satisfy the constraints: [0, 0, 1] & [1, 1, 0].

Therefore, let's set the winning state as $W_1 = 001$ and $W_2 = 110$. Now we must use Grover's Search Algorithm to locate these two states.

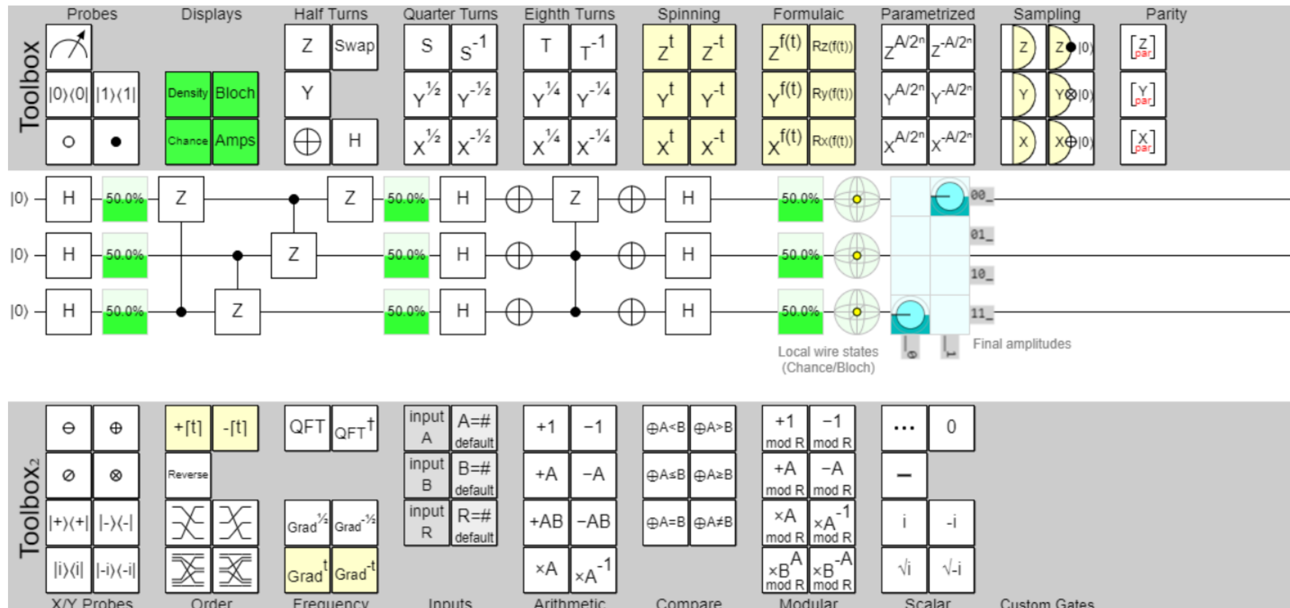
In my *Taxi-Code,* I first program this classically, as shown on the right. ➔

```
[[1, 1, 1], -1]
[[1, 1, 0], 1]
[[1, 0, 1], -1]
[[1, 0, 0], -1]
[[0, 1, 1], -1]
[[0, 1, 0], -1]
[[0, 0, 1], 1]
[[0, 0, 0], -1]
-1
Best Option(s):
[[[1, 1, 0], 1], [[0, 0, 1], 1]]
```

The -1 and +1 are values as this problem can be phrased as a game, that is, if the combination of friends A, B and C are such that they do not follow the conditions, then the 'score' is determined to be -1 for this value. However, if the combination is a good one (i.e. meets all the requirements) then it will be represented with a +1. Hence, one can see which combinations satisfy the conditions. The best options are shown to be the winning states as mentioned above: 001 and 110.

The second part of the code I implement the Qiskit library and use this to create the circuit visually and find the winning states using Grover's Algorithm.

Before doing so however, I had to determine how the circuit of the oracle will be such that the states 110 *and* 001 can be obtained. After playing around on a circuit simulator, I discovered the circuit that will do this looks as follows:

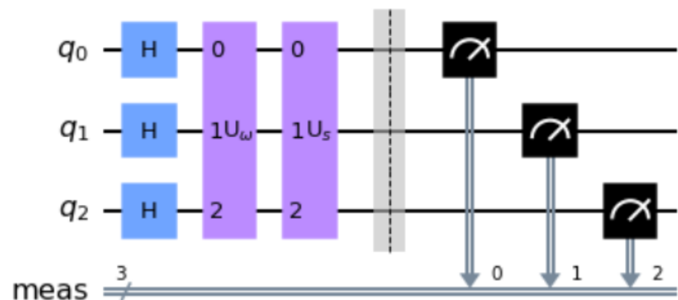


The main thing to note is that in order to get these specific winning states, we must have the oracle with a controlled-Z gate between qubit-1 and qubit-3, qubit-3 and qubit-2, and between qubit-2 and qubit-1, along with a Pauli-Z gate on qubit-1. The Diffuser operator remains standard (since that simply amplifies the result that we desire).

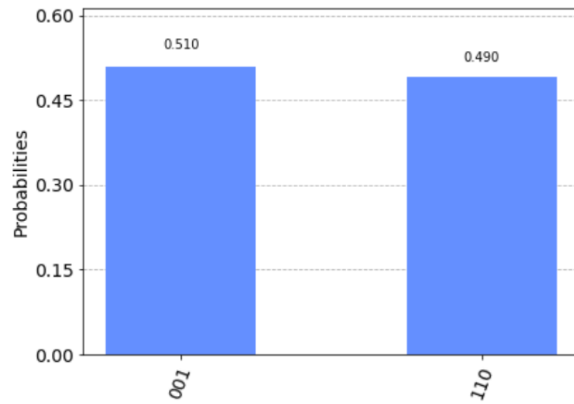
In my code, I initialize the qubits into a superposition by applying Hadamard gates onto all three qubits. I then create the oracle with the CZ- and Z-gates as already described to be necessary in order to find the two states that we are searching for, by flipping their sign. This results in a total wavefunction being:

$$\frac{1}{\sqrt{8}}(|000> -|001> +|011> +|100> +|101> -|110> +|111>)$$

I proceed to make a diffuser by creating a function that puts the following gates in the precise order: *Hadamard, Pauli-X, MCT (Multi-Controlled Toffoli), Pauli-X,* and *Hadamard*. I turn both the oracle and the diffuser into their own whole gates and draw the circuit, which looks as:



I proceed by first running the circuit on a backend simulator (i.e. not a real quantum computer) to see if I would get the out puts of |001> and |110> with a certain probability. ➔

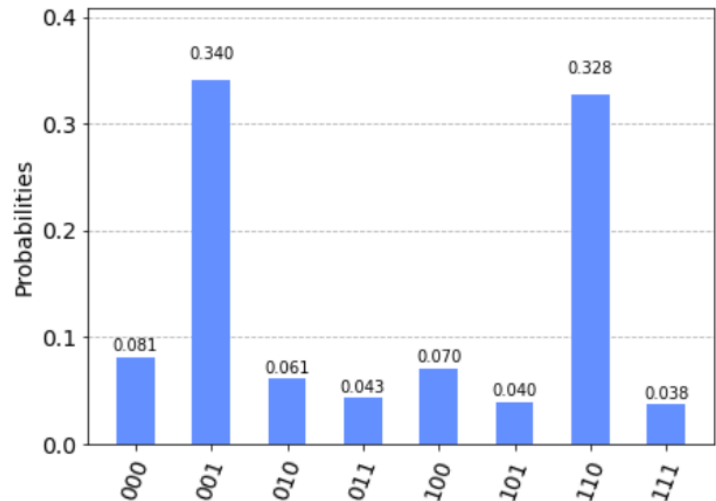


Solution to Problem #2

Finally, I run the circuit on a real quantum computer—in my run it was on the ibmq_lima device.

See diagram on the right.

```
Run on least busy device:  ibmq_lima
Job Status: job has successfully run
```



The outputs were just as expected-- |001> and |110> with higher probability amplitudes than the other states, indicating that I have found the two states that I have been searching for.

This is how Grover's Algorithm is implemented and the results you get.

Conclusion

I demonstrated two examples: a two-qubit circuit and a three-qubit circuit which use Grover's Algorithm to find a desired state. Thus, in essence, this is a search algorithm (as demonstrated with Problem #1). Grover's algorithm can be used as an optimization algorithm to solve a type of problem that is called the Satisfiability Problem. These are problems in which you must find the optimal states of a system given certain conditions. Problem #2 was an example of this. The problem was a re-defined from a popular problem called "Grover's Dinner Party"—where you wish to invite people to a dinner party however the conditions are such that some are enemies and some are friends—you must determine what would be the optimal combination(s).

I demonstrated how this algorithm is used using quantum circuits and applying python to create those circuits, as well as comparing classical examples of the problems to demonstrate why Grover's algorithm, and quantum computing in general, is a more optimal form of computation compared to classical computations—a major reason for the recent growth in quantum computing research in the past decade.

Citations

https://en.wikipedia.org/wiki/Grover%27s_algorithm

https://qiskit.org/textbook/ch-algorithms/grover.html

https://quantumzeitgeist.com/grovers-algorithm-an-intuitive-look/

https://quantumcomputinguk.org/tutorials/grovers-algorithm-with-code

https://demonstrations.wolfram.com/QuantumCircuitImplementingGroversSearchAlgorithm/

https://www.quantum-inspire.com/kbase/grover-algorithm/

https://arcb.csc.ncsu.edu/~mueller/qc/qc18/readings/dreher3.pdf

https://arxiv.org/pdf/quant-ph/0301079.pdf

https://arxiv.org/pdf/0811.4481.pdf

Grover's_Problem_2_Taxi

May 4, 2022

[12]:
```
#Classical Taxi Code---------------------------------------------
```

[6]:
```
scores = []

#List all states as list [A,B,C] w/ each friend able to take 0 (not in taxi) or␣
↪1 (in taxi)
states = [[1,1,1], [1,1,0], [1,0,1], [1,0,0], [0,1,1],
          [0,1,0],[0,0,1],[0,0,0]]

for i in range(0,8):
    individ_state = states[i]
    #print(current_config)

    #the state of each friend
    A = individ_state[0]
    B = individ_state[1]
    C = individ_state[2]

    score = 0

    if (A == 0 and B == 0) or (A == 1 and B == 1):
        score += 1

    if (A == 0 and C == 0) or (A == 1 and C == 1):
        score -= 1

    if (B == 0 and C == 0) or (B == 1 and C == 1):
        score -= 1

    status = [individ_state, score] #will output the state & if its good (+1)␣
↪or bad (-1)
                                    #outputs all scores
    scores.append(status)
    print(status)


highest_score = scores[0][1]
```

```
print(highest_score) #prints -1
best_options = []    #list of best state (and their score)

for i in range(1,8):
    if scores[i][1] >= highest_score:  #if score is > -1
        highest_score = scores[i][1]   #this becomes the new score
        best_options.append(scores[i])          #appends to list therefore␣
 ↪giving highest score

print("Best Option(s): ")
print(best_options)
```

```
[[1, 1, 1], -1]
[[1, 1, 0], 1]
[[1, 0, 1], -1]
[[1, 0, 0], -1]
[[0, 1, 1], -1]
[[0, 1, 0], -1]
[[0, 0, 1], 1]
[[0, 0, 0], -1]
-1
Best Option(s):
[[[1, 1, 0], 1], [[0, 0, 1], 1]]
```

[7]:

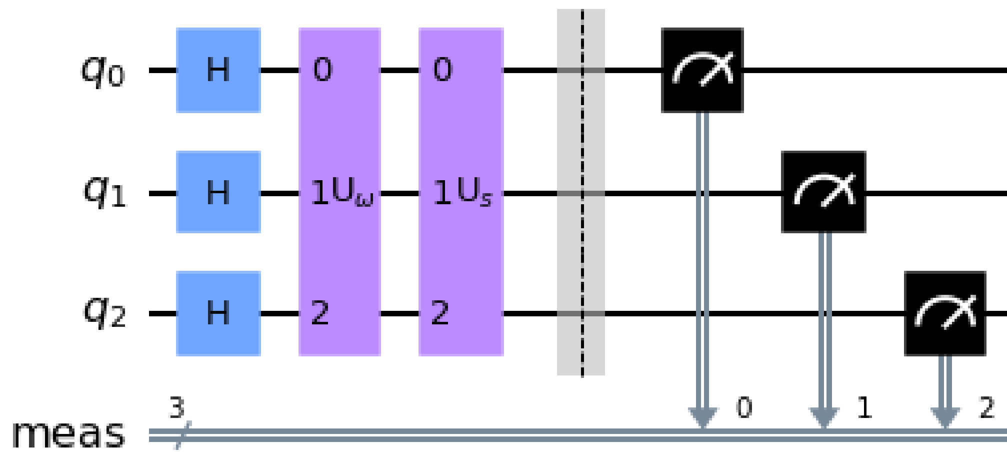
```
#Quantum Taxi Code--------------------------------------------------
```

[13]:

```
#Import python libraries & Qiskit
import matplotlib.pyplot as plt
import numpy as np
from qiskit import IBMQ, Aer, assemble, transpile
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit.providers.ibmq import least_busy
from qiskit.visualization import plot_histogram

#Put all initial qubits into superposition
def initialize(qc, qubits):
    for q in qubits:
        qc.h(q)
    return qc

#0,0,0 initial qubits
qc = QuantumCircuit(3)

#Adding CZ and Z gates appropriately in ORACLE
qc.cz(0, 2)
qc.cz(1, 2)
qc.cz(0,1)
```

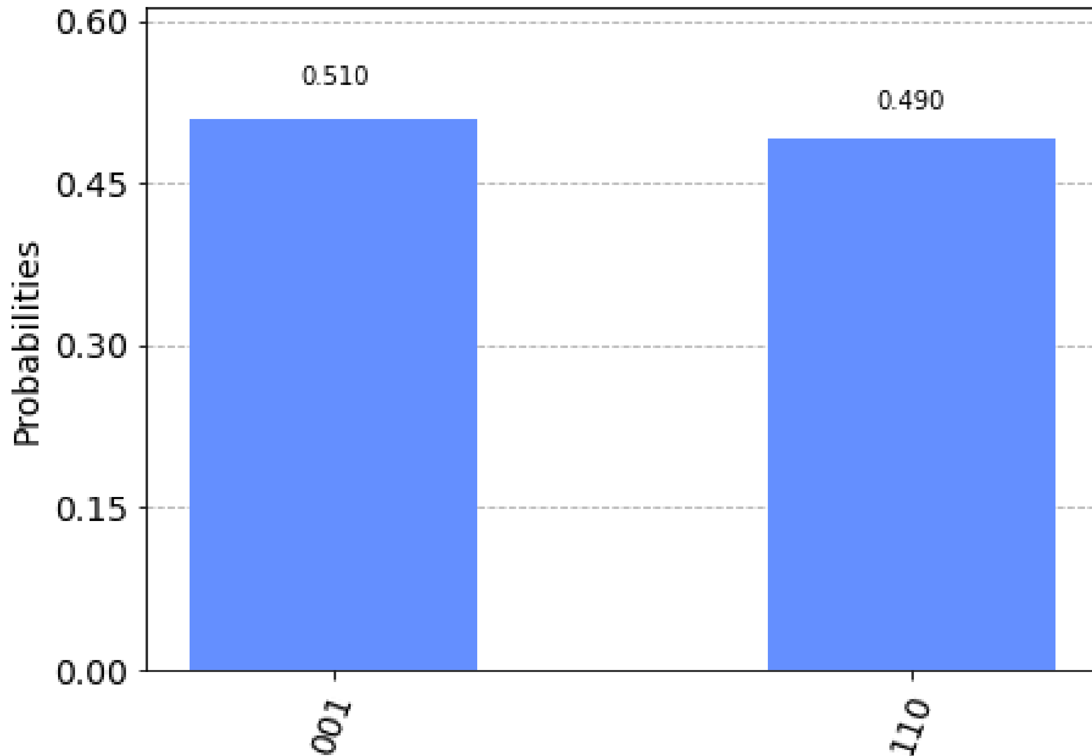
```
qc.z(0)
oracle = qc.to_gate()
oracle.name = "U$_\omega$"  #name of oracle--i.e. the U_omega operator

#Diffuser Algorithm
def diffuser(nqubits):
    qc = QuantumCircuit(nqubits)

    for qubit in range(nqubits):  #First part is applying H-gates
        qc.h(qubit)

    for qubit in range(nqubits):  #Second part is applying X-gates
        qc.x(qubit)

    qc.h(nqubits-1)
    qc.mct(list(range(nqubits-1)), nqubits-1)  #Applying Toffoli gates␣
 ↪(multi-CZ)
                                               #mct stands for␣
 ↪'multi-controlled-tofolli'
    qc.h(nqubits-1)
                                               #Applying X-gates and H-gates␣
 ↪again
    for qubit in range(nqubits):
        qc.x(qubit)

    for qubit in range(nqubits):
        qc.h(qubit)
                                            #Make the diffuser a gate (i.e. the␣
 ↪U_s operator), and return it
    U_s = qc.to_gate()
    U_s.name = "U$_s$"
    return U_s

#Setting up the circuit & acting it out
n = 3
grover_circ = QuantumCircuit(n)
grover_circ = initialize(grover_circ, [0,1,2])   #Initialized part acting on␣
 ↪the 3 qbits (H qubits)
grover_circ.append(oracle, [0,1,2])              #Oracle part acting on the 3␣
 ↪qbits
grover_circ.append(diffuser(n), [0,1,2])         #Diffuser part on 3 qbits

#Measure & Draw Circuit
grover_circ.measure_all()
grover_circ.draw()
```

[13]:



[10]:
```
#Simulator
aer_sim = Aer.get_backend('aer_simulator')
transpiled_grover_circuit = transpile(grover_circ, aer_sim)
qobj = assemble(transpiled_grover_circuit)
results = aer_sim.run(qobj).result()
counts = results.get_counts()
plot_histogram(counts)
```

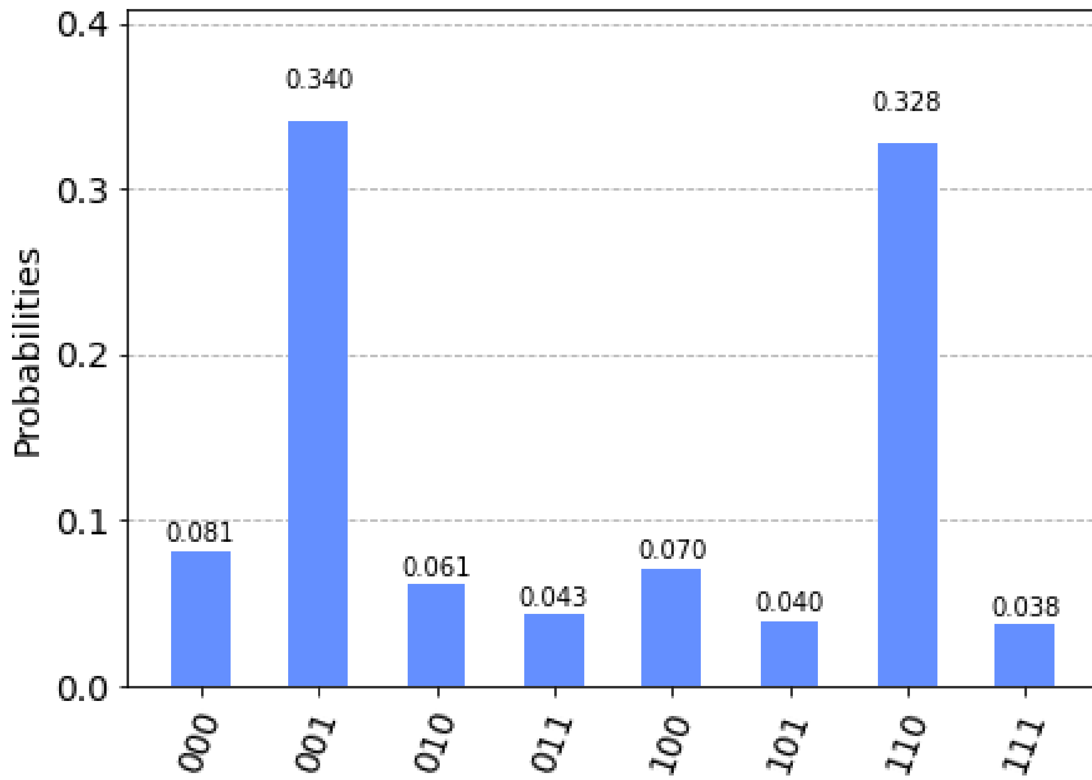
[10]:



[11]:
```
#Running Code on Real Device in the cloud at IBM

#First Load IBM Q account and get the least busy backend device
provider = IBMQ.load_account()
provider = IBMQ.get_provider("ibm-q")
device = least_busy(provider.backends(filters=lambda x: x.configuration().
↪n_qubits >= 3 and
                                        not x.configuration().simulator and x.
↪status().operational==True))
print("Run on least busy device: ", device)

#Run circuit and monitor the execution of the job in the queue
from qiskit.tools.monitor import job_monitor
transpiled_grover_circuit = transpile(grover_circ, device, optimization_level=3)
job = device.run(transpiled_grover_circuit)
job_monitor(job, interval=2)

#Results
results = job.result()
answer = results.get_counts(grover_circ)
plot_histogram(answer)
```

```
Run on least busy device:  ibmq_lima
Job Status: job has successfully run
```

[11]:



[ ]: