

Grover's Algorithm

In this section, we introduce Grover's algorithm and how it can be used to solve unstructured search problems. We then implement the quantum algorithm using Qiskit, and run on a simulator and device.

Contents

1. [Introduction](#)
2. [Example: 2 Qubits](#)
 - 2.1 [Simulation](#)
 - 2.2 [Device](#)
3. [Example: 3 Qubits](#)
 - 3.1 [Simulation](#)
 - 3.2 [Device](#)
4. [Problems](#)
5. [Solving Sudoku using Grover's Algorithm](#)
6. [References](#)

1. Introduction

You have likely heard that one of the many advantages a quantum computer has over a classical computer is its superior speed searching databases. Grover's algorithm demonstrates this capability. This algorithm can speed up an unstructured search problem quadratically, but its uses extend beyond that; it can serve as a general trick or subroutine to obtain quadratic run time improvements for a variety of other algorithms. This is called the amplitude amplification trick.

Unstructured Search

Suppose you are given a large list of N items. Among these items there is one item with a unique property that we wish to locate; we will call this one the winner w . Think of each item in the list as a box of a particular color. Say all items in the list are gray except the winner w , which is purple.



To find the purple box -- the *marked item* -- using classical computation, one would have to check on average $N/2$ of these boxes, and in the worst case, all N of them. On a quantum computer, however, we can find the marked item in roughly \sqrt{N} steps with Grover's amplitude amplification trick. A quadratic speedup is indeed a substantial time-

saver for finding marked items in long lists. Additionally, the algorithm does not use the list's internal structure, which makes it *generic*; this is why it immediately provides a quadratic quantum speed-up for many classical problems.

Creating an Oracle

For the examples in this textbook, our 'database' is comprised of all the possible computational basis states our qubits can be in. For example, if we have 3 qubits, our list is the states $|000\rangle, |001\rangle, \dots |111\rangle$ (i.e the states $|0\rangle \rightarrow |7\rangle$).

Grover's algorithm solves oracles that add a negative phase to the solution states. I.e. for any state $|x\rangle$ in the computational basis:

$$U_{\omega}|x\rangle = \begin{cases} -|x\rangle & \text{if } x = \omega \\ |x\rangle & \text{if } x \neq \omega \end{cases}$$

This oracle will be a diagonal matrix, where the entry that correspond to the marked item will have a negative phase. For example, if we have three qubits and $\omega = |101\rangle$, our oracle will have the matrix:

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

What makes Grover's algorithm so powerful is how easy it is to convert a problem to an oracle of this form. There are many computational problems in which it's difficult to *find* a solution, but relatively easy to *verify* a solution. For example, we can easily verify a solution to a [sudoku](#) by checking all the rules are satisfied. For these problems, we can create a function f that takes a proposed solution x , and returns $f(x) = 0$ if x is not a solution ($x \neq \omega$) and $f(x) = 1$ for a valid solution ($x = \omega$). Our oracle can then be described as:

$$U_{\omega}|x\rangle = (-1)^{f(x)}|x\rangle$$

and the oracle's matrix will be a diagonal matrix of the form:

$$U_{\omega} = \begin{bmatrix} (-1)^{f(0)} & 0 & \dots & 0 \\ 0 & (-1)^{f(1)} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & (-1)^{f(2^n-1)} \end{bmatrix}$$

► Circuit Construction of a Grover Oracle (click to expand)

For the next part of this chapter, we aim to teach the core concepts of the algorithm. We will create example oracles where we know ω beforehand, and not worry ourselves with whether these oracles are useful or not. At the end of the chapter, we will cover a short example where we create an oracle to solve a problem (sudoku).

Amplitude Amplification

So how does the algorithm work? Before looking at the list of items, we have no idea where the marked item is. Therefore, any guess of its location is as good as any other, which can be expressed in terms of a uniform superposition: $|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$.

If at this point we were to measure in the standard basis $\{|x\rangle\}$, this superposition would collapse, according to the fifth quantum law, to any one of the basis states with the same probability of $\frac{1}{N} = \frac{1}{2^n}$. Our chances of guessing the right value w is therefore 1 in 2^n , as could be expected. Hence, on average we would need to try about $N/2 = 2^{n-1}$ times to guess the correct item.

Enter the procedure called amplitude amplification, which is how a quantum computer significantly enhances this probability. This procedure stretches out (amplifies) the amplitude of the marked item, which shrinks the other items' amplitude, so that measuring the final state will return the right item with near-certainty.

This algorithm has a nice geometrical interpretation in terms of two reflections, which generate a rotation in a two-dimensional plane. The only two special states we need to consider are the winner $|w\rangle$ and the uniform superposition $|s\rangle$. These two vectors span a two-dimensional plane in the vector space \mathbb{C}^N . They are not quite perpendicular because $|w\rangle$ occurs in the superposition with amplitude $N^{-1/2}$ as well. We can, however, introduce an additional state $|s'\rangle$ that is in the span of these two vectors, which is perpendicular to $|w\rangle$ and is obtained from $|s\rangle$ by removing $|w\rangle$ and rescaling.

Step 1: The amplitude amplification procedure starts out in the uniform superposition $|s\rangle$, which is easily constructed from $|s\rangle = H^{\otimes n} |0\rangle$.



The left graphic corresponds to the two-dimensional plane spanned by perpendicular vectors $|w\rangle$ and $|s'\rangle$ which allows to express the initial state as $|s\rangle = \sin \theta |w\rangle + \cos \theta |s'\rangle$, where $\theta = \arcsin \frac{1}{\sqrt{N}}$. The right graphic is a bar graph of the amplitudes of the state $|s\rangle$.

Step 2: We apply the oracle reflection U_f to the state $|s\rangle$.



Geometrically this corresponds to a reflection of the state $|s\rangle$ about $|s'\rangle$. This transformation means that the amplitude in front of the $|w\rangle$ state becomes

negative, which in turn means that the average amplitude (indicated by a dashed line) has been lowered.

Step 3: We now apply an additional reflection (U_s) about the state $|s\rangle$: $U_s = 2|s\rangle\langle s| - I$. This transformation maps the state to $U_s U_f |s\rangle$ and completes the transformation.



Two reflections always correspond to a rotation. The transformation $U_s U_f$ rotates the initial state $|s\rangle$ closer towards the winner $|w\rangle$. The action of the reflection U_s in the amplitude bar diagram can be understood as a reflection about the average amplitude. Since the average amplitude has been lowered by the first reflection, this transformation boosts the negative amplitude of $|w\rangle$ to roughly three times its original value, while it decreases the other amplitudes. We then go to **step 2** to repeat the application. This procedure will be repeated several times to zero in on the winner.

After t steps we will be in the state $|\psi_t\rangle$ where: $|\psi_t\rangle = (U_s U_f)^t |s\rangle$.

How many times do we need to apply the rotation? It turns out that roughly \sqrt{N} rotations suffice. This becomes clear when looking at the amplitudes of the state $|\psi_t\rangle$. We can see that the amplitude of $|w\rangle$ grows linearly with the number of applications $\sim t^{1/2}$. However, since we are dealing with amplitudes and not probabilities, the vector space's dimension enters as a square root. Therefore it is the amplitude, and not just the probability, that is being amplified in this procedure.

In the case that there are multiple solutions, M , it can be shown that roughly $\sqrt{(N/M)}$ rotations will suffice.



2. Example: 2 Qubits

Let's first have a look at the case of Grover's algorithm for $N=4$ which is realized with 2 qubits. In this particular case, only **one rotation** is required to rotate the initial state $|s\rangle$ to the winner $|w\rangle$ [3]:

1. Following the above introduction, in the case $N=4$ we have $\theta = \arcsin \frac{1}{\sqrt{2}} = \frac{\pi}{6}$.
2. After t steps, we have $(U_s U_\omega)^t |s\rangle = \sin \theta_t |\omega\rangle + \cos \theta_t |s'\rangle$, where $\theta_t = (2t+1)\theta$.
3. In order to obtain $|\omega\rangle$ we need $\theta_t = \frac{\pi}{2}$, which with $\theta = \frac{\pi}{6}$ inserted above results to $t=1$. This implies that after $t=1$ rotation the searched element is found.

We will now follow through an example using a specific oracle.

Oracle for $|w\rangle = |11\rangle$

Let's look at the case $|w\rangle = |11\rangle$. The oracle U_ω in this case acts as follows:

$$U_\omega |s\rangle = U_\omega \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle + |10\rangle - |11\rangle)$$

or:

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

which you may recognise as the controlled-Z gate. Thus, for this example, our oracle is simply the controlled-Z gate:



Reflection U_s

In order to complete the circuit we need to implement the additional reflection $U_s = 2|s\rangle\langle s| - I$. Since this is a reflection about $|s\rangle$, we want to add a negative phase to every state orthogonal to $|s\rangle$.

One way we can do this is to use the operation that transforms the state $|s\rangle \rightarrow |0\rangle$, which we already know is the Hadamard gate applied to each qubit:

$$H^{\otimes n} |s\rangle = |0\rangle$$

Then we apply a circuit that adds a negative phase to the states orthogonal to $|0\rangle$:

$$U_0 \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{\sqrt{2}}(|00\rangle - |01\rangle - |10\rangle - |11\rangle)$$

i.e. the signs of each state are flipped except for $|00\rangle$. As can easily be verified, one way of implementing U_0 is the following circuit:



Finally, we do the operation that transforms the state $|0\rangle \rightarrow |s\rangle$ (the H-gate again):

$$H^{\otimes n} U_0 H^{\otimes n} = U_s$$

The complete circuit for U_s looks like this:



Full Circuit for $|w\rangle = |11\rangle$

Since in the particular case of $N=4$ only one rotation is required we can combine the above components to build the full circuit for Grover's algorithm for the case $|w\rangle = |11\rangle$:



2.1 Qiskit Implementation

We now implement Grover's algorithm for the above case of 2 qubits for $|w\rangle = |11\rangle$.

```
In [1]: #initialization
import matplotlib.pyplot as plt
import numpy as np

# importing Qiskit
from qiskit import IBMQ, Aer, assemble, transpile
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit.providers.ibmq import least_busy

# import basic plot tools
from qiskit.visualization import plot_histogram
```

We start by preparing a quantum circuit with two qubits:

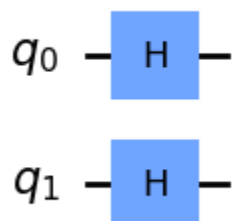
```
In [2]: n = 2
grover_circuit = QuantumCircuit(n)
```

Then we simply need to write out the commands for the circuit depicted above. First, we need to initialize the state $|s\rangle$. Let's create a general function (for any number of qubits) so we can use it again later:

```
In [3]: def initialize_s(qc, qubits):
        """Apply a H-gate to 'qubits' in qc"""
        for q in qubits:
            qc.h(q)
        return qc
```

```
In [4]: grover_circuit = initialize_s(grover_circuit, [0,1])
grover_circuit.draw()
```

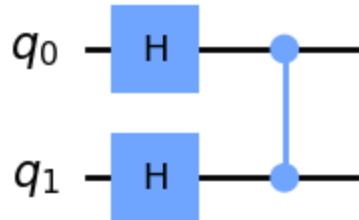
Out[4]:



Apply the Oracle for $|w\rangle = |11\rangle$. This oracle is specific to 2 qubits:

```
In [5]: grover_circuit.cz(0,1) # Oracle  
grover_circuit.draw()
```

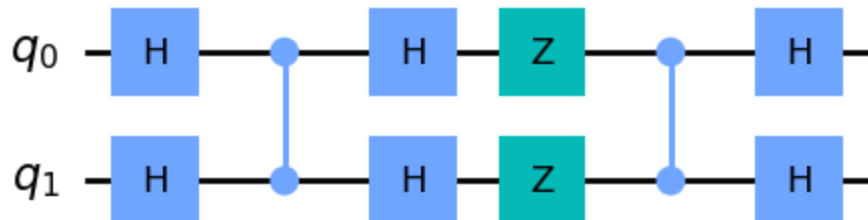
Out[5]:



We now want to apply the diffuser (U_s). As with the circuit that initializes $|s\rangle$, we'll create a general diffuser (for any number of qubits) so we can use it later in other problems.

```
In [6]: # Diffusion operator (U_s)  
grover_circuit.h([0,1])  
grover_circuit.z([0,1])  
grover_circuit.cz(0,1)  
grover_circuit.h([0,1])  
grover_circuit.draw()
```

Out[6]:



This is our finished circuit.

2.1.1 Experiment with Simulators

Let's run the circuit in simulation. First, we can verify that we have the correct statevector:

```
In [7]: sim = Aer.get_backend('aer_simulator')  
# we need to make a copy of the circuit with the 'save_statevector'  
# instruction to run on the Aer simulator  
grover_circuit_sim = grover_circuit.copy()  
grover_circuit_sim.save_statevector()  
qobj = assemble(grover_circuit_sim)  
result = sim.run(qobj).result()  
statevec = result.get_statevector()
```

```
from qiskit_textbook.tools import vector2latex
vector2latex(statevec, pretext="|\\psi\\rangle =")
```

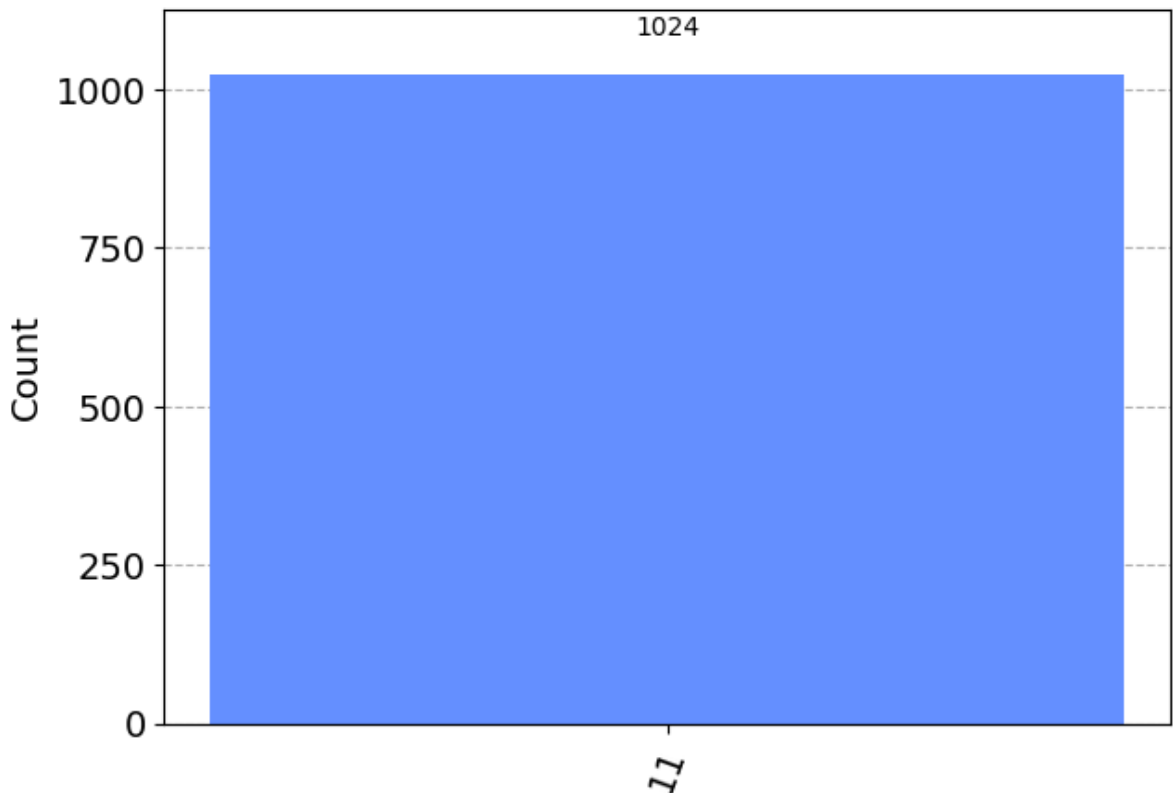
$$|\psi\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

As expected, the amplitude of every state that is not $|11\rangle$ is 0, this means we have a 100% chance of measuring $|11\rangle$:

```
In [8]: grover_circuit.measure_all()

aer_sim = Aer.get_backend('aer_simulator')
qobj = assemble(grover_circuit)
result = aer_sim.run(qobj).result()
counts = result.get_counts()
plot_histogram(counts)
```

Out[8]:



2.1.2 Experiment with Real Devices

We can run the circuit a real device as below.

```
In [9]: # 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: int(x.configuration().n_qubits) >= 5 and
not x.configuration().simulator and x.status() != 'BUSY')))
print("Running on current least busy device: ", device)
```



```
/tmp/ipykernel_13906/3474636535.py:2: DeprecationWarning: The qiskit.IBMQ entrypoint and the qiskit-ibmq-provider package (accessible from 'qiskit.providers.ibmq') are deprecated and will be removed in a future release. Instead you should use the qiskit-ibm-provider package which is accessible from 'qiskit_ibm_provider'. You can install it with 'pip install qiskit_ibm_provider'. Just replace 'qiskit.IBMQ' with 'qiskit_ibm_provider.IBMProvider'
provider = IBMQ.load_account()
```

Running on current least busy device: ibmq_lima

```
In [ ]: # Run our circuit on the least busy backend. Monitor the execution of the job
from qiskit.tools.monitor import job_monitor
transpiled_grover_circuit = transpile(grover_circuit, device, optimization_level=1)
job = device.run(transpiled_grover_circuit)
job_monitor(job, interval=2)
```

Job Status: job is queued (24)

```
In [ ]: # Get the results from the computation
results = job.result()
answer = results.get_counts(grover_circuit)
plot_histogram(answer)
```

We confirm that in the majority of the cases the state $|11\rangle$ is measured. The other results are due to errors in the quantum computation.

3. Example: 3 Qubits

We now go through the example of Grover's algorithm for 3 qubits with two marked states $|101\rangle$ and $|110\rangle$, following the implementation found in Reference [2]. The quantum circuit to solve the problem using a phase oracle is:



1. Apply Hadamard gates to 3 qubits initialized to $|000\rangle$ to create a uniform superposition: $|\psi_1\rangle = \frac{1}{\sqrt{8}} (|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle)$
2. Mark states $|101\rangle$ and $|110\rangle$ using a phase oracle: $|\psi_2\rangle = \frac{1}{\sqrt{8}} (|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle - |101\rangle - |110\rangle + |111\rangle)$
3. Perform the reflection around the average amplitude:
 - A. Apply Hadamard gates to the qubits $|\psi_{3a}\rangle = \frac{1}{2} (|000\rangle + |011\rangle + |100\rangle - |111\rangle)$
 - B. Apply X gates to the qubits $|\psi_{3b}\rangle = \frac{1}{2} (|000\rangle + |011\rangle + |100\rangle + |111\rangle)$
 - C. Apply a doubly controlled Z gate between the 1, 2 (controls) and 3 (target) qubits $|\psi_{3c}\rangle = \frac{1}{2} (|000\rangle + |011\rangle + |100\rangle + |111\rangle)$

$+\lvert 100\rangle - \lvert 111\rangle \rangle$

D. Apply X gates to the qubits $\lvert \psi_{3d} \rangle = \frac{1}{2} \lvert 000 \rangle + \lvert 011 \rangle + \lvert 100 \rangle - \lvert 111 \rangle$

E. Apply Hadamard gates to the qubits $\lvert \psi_{3e} \rangle = \frac{1}{\sqrt{2}} \lvert 101 \rangle - \lvert 110 \rangle$

4. Measure the 3 qubits to retrieve states $\lvert 101 \rangle$ and $\lvert 110 \rangle$

Note that since there are 2 solutions and 8 possibilities, we will only need to run one iteration (steps 2 & 3).

3.1 Qiskit Implementation

We now implement Grover's algorithm for the above [example](#) for 3-qubits and searching for two marked states $\lvert 101 \rangle$ and $\lvert 110 \rangle$. **Note:** Remember that Qiskit orders it's qubits the opposite way round to this resource, so the circuit drawn will appear flipped about the horizontal.

We create a phase oracle that will mark states $\lvert 101 \rangle$ and $\lvert 110 \rangle$ as the results (step 1).

```
In [ ]: qc = QuantumCircuit(3)
        qc.cz(0, 2)
        qc.cz(1, 2)
        oracle_ex3 = qc.to_gate()
        oracle_ex3.name = "U_\omega"
```

In the last section, we used a diffuser specific to 2 qubits, in the cell below we will create a general diffuser for any number of qubits.

► Details: Creating a General Diffuser (click to expand)

```
In [ ]: def diffuser(nqubits):
        qc = QuantumCircuit(nqubits)
        # Apply transformation |s> -> |00..0> (H-gates)
        for qubit in range(nqubits):
            qc.h(qubit)
        # Apply transformation |00..0> -> |11..1> (X-gates)
        for qubit in range(nqubits):
            qc.x(qubit)
        # Do multi-controlled-Z gate
        qc.h(nqubits-1)
        qc.mct(list(range(nqubits-1)), nqubits-1) # multi-controlled-toffoli
        qc.h(nqubits-1)
        # Apply transformation |11..1> -> |00..0>
        for qubit in range(nqubits):
            qc.x(qubit)
        # Apply transformation |00..0> -> |s>
        for qubit in range(nqubits):
            qc.h(qubit)
        # We will return the diffuser as a gate
```

```

U_s = qc.to_gate()
U_s.name = "U$_s$"
return U_s

```

We'll now put the pieces together, with the creation of a uniform superposition at the start of the circuit and a measurement at the end. Note that since there are 2 solutions and 8 possibilities, we will only need to run one iteration.

```

In [ ]: n = 3
grover_circuit = QuantumCircuit(n)
grover_circuit = initialize_s(grover_circuit, [0,1,2])
grover_circuit.append(oracle_ex3, [0,1,2])
grover_circuit.append(diffuser(n), [0,1,2])
grover_circuit.measure_all()
grover_circuit.draw()

```

3.1.1 Experiment with Simulators

We can run the above circuit on the simulator.

```

In [ ]: aer_sim = Aer.get_backend('aer_simulator')
transpiled_grover_circuit = transpile(grover_circuit, aer_sim)
qobj = assemble(transpiled_grover_circuit)
results = aer_sim.run(qobj).result()
counts = results.get_counts()
plot_histogram(counts)

```

As we can see, the algorithm discovers our marked states $|101\rangle$ and $|110\rangle$.

3.1.2 Experiment with Real Devices

We can run the circuit on the real device as below.

```

In [ ]: backend = least_busy(provider.backends(filters=lambda x: int(x.configuration()
not x.configuration().simulator and x.sta
print("least busy backend: ", backend)

```

```

In [ ]: # Run our circuit on the least busy backend. Monitor the execution of the job
from qiskit.tools.monitor import job_monitor
transpiled_grover_circuit = transpile(grover_circuit, device, optimization_level=1)
job = device.run(transpiled_grover_circuit)
job_monitor(job, interval=2)

```

```

In [ ]: # Get the results from the computation
results = job.result()
answer = results.get_counts(grover_circuit)
plot_histogram(answer)

```

As we can (hopefully) see, there is a higher chance of measuring $|101\rangle$ and $|110\rangle$. The other results are due to errors in the quantum computation.

4. Problems

The function `grover_problem_oracle` below takes a number of qubits (`n`), and a `variant` and returns an `n`-qubit oracle. The function will always return the same oracle for the same `n` and `variant`. You can see the solutions to each oracle by setting `print_solutions = True` when calling `grover_problem_oracle`.

```
In [ ]: from qiskit_textbook.problems import grover_problem_oracle
        ## Example Usage
        n = 4
        oracle = grover_problem_oracle(n, variant=1) # 0th variant of oracle, with
        qc = QuantumCircuit(n)
        qc.append(oracle, [0,1,2,3])
        qc.draw()
```


1. `grover_problem_oracle(4, variant=2)` uses 4 qubits and has 1 solution.
 - a. How many iterations do we need to have a > 90% chance of measuring this solution?
 - b. Use Grover's algorithm to find this solution state. c. What happens if we apply more iterations the number we calculated in problem 1a above? Why?
2. With 2 solutions and 4 qubits, how many iterations do we need for a >90% chance of measuring a solution? Test your answer using the oracle `grover_problem_oracle(4, variant=1)` (which has two solutions).
3. Create a function, `grover_solver(oracle, iterations)` that takes as input:
 - A Grover oracle as a gate (`oracle`)
 - An integer number of iterations (`iterations`)and returns a `QuantumCircuit` that performs Grover's algorithm on the '`oracle`' gate, with '`iterations`' iterations.

5. Solving Sudoku using Grover's Algorithm

The oracles used throughout this chapter so far have been created with prior knowledge of their solutions. We will now solve a simple problem using Grover's algorithm, for which we do not necessarily know the solution beforehand. Our problem is a 2x2 binary sudoku, which in our case has two simple rules:

- No column may contain the same value twice
- No row may contain the same value twice

If we assign each square in our sudoku to a variable like so:

 2x2 binary sudoku, with each square allocated to a different variable

we want our circuit to output a solution to this sudoku.

Note that, while this approach of using Grover's algorithm to solve this problem is not practical (you can probably find the solution in your head!), the purpose of this example is to demonstrate the conversion of classical [decision problems](#) into oracles for Grover's algorithm.

5.1 Turning the Problem into a Circuit

We want to create an oracle that will help us solve this problem, and we will start by creating a circuit that identifies a correct solution. Similar to how we created a classical adder using quantum circuits in [The Atoms of Computation](#), we simply need to create a *classical* function on a quantum circuit that checks whether the state of our variable bits is a valid solution.

Since we need to check down both columns and across both rows, there are 4 conditions we need to check:

```
v0 ≠ v1    # check along top row
v2 ≠ v3    # check along bottom row
v0 ≠ v2    # check down left column
v1 ≠ v3    # check down right column
```

Remember we are comparing classical (computational basis) states. For convenience, we can compile this set of comparisons into a list of clauses:

```
In [ ]: clause_list = [[0,1],
                       [0,2],
                       [1,3],
                       [2,3]]
```

We will assign the value of each variable to a bit in our circuit. To check these clauses computationally, we will use the `XOR` gate (we came across this in the atoms of computation).

```
In [ ]: def XOR(qc, a, b, output):
        qc.cx(a, output)
        qc.cx(b, output)
```

Convince yourself that the `output0` bit in the circuit below will only be flipped if `input0` \neq `input1`:

```
In [ ]: # We will use separate registers to name the bits
in_qubits = QuantumRegister(2, name='input')
out_qubit = QuantumRegister(1, name='output')
```

```
qc = QuantumCircuit(in_qubits, out_qubit)
XOR(qc, in_qubits[0], in_qubits[1], out_qubit)
qc.draw()
```

This circuit checks whether `input0 == input1` and stores the output to `output0`. To check each clause, we repeat this circuit for each pairing in `clause_list` and store the output to a new bit:

```
In [ ]: # Create separate registers to name bits
var_qubits = QuantumRegister(4, name='v') # variable bits
clause_qubits = QuantumRegister(4, name='c') # bits to store clause-checks

# Create quantum circuit
qc = QuantumCircuit(var_qubits, clause_qubits)

# Use XOR gate to check each clause
i = 0
for clause in clause_list:
    XOR(qc, clause[0], clause[1], clause_qubits[i])
    i += 1

qc.draw()
```

The final state of the bits `c0`, `c1`, `c2`, `c3` will only all be `1` in the case that the assignments of `v0`, `v1`, `v2`, `v3` are a solution to the sudoku. To complete our checking circuit, we want a single bit to be `1` if (and only if) all the clauses are satisfied, this way we can look at just one bit to see if our assignment is a solution. We can do this using a multi-controlled-Toffoli-gate:

```
In [ ]: # Create separate registers to name bits
var_qubits = QuantumRegister(4, name='v')
clause_qubits = QuantumRegister(4, name='c')
output_qubit = QuantumRegister(1, name='out')
qc = QuantumCircuit(var_qubits, clause_qubits, output_qubit)

# Compute clauses
i = 0
for clause in clause_list:
    XOR(qc, clause[0], clause[1], clause_qubits[i])
    i += 1

# Flip 'output' bit if all clauses are satisfied
qc.mct(clause_qubits, output_qubit)

qc.draw()
```

The circuit above takes as input an initial assignment of the bits `v0`, `v1`, `v2` and `v3`, and all other bits should be initialized to `0`. After running the circuit, the state of the `out0` bit tells us if this assignment is a solution or not; `out0 = 0` means the assignment is *not* a solution, and `out0 = 1` means the assignment is a solution.

Important: Before you continue, it is important you fully understand this circuit and are convinced it works as stated in the paragraph above.

5.2 Uncomputing, and Completing the Oracle

We can now turn this checking circuit into a Grover oracle using [phase kickback](#). To recap, we have 3 registers:

- One register which stores our sudoku variables (we'll say $x = v_3, v_2, v_1, v_0$)
- One register that stores our clauses (this starts in the state $|0000\rangle$ which we'll abbreviate to $|0\rangle$)
- And one qubit ($|\text{out}\rangle$) that we've been using to store the output of our checking circuit.

To create an oracle, we need our circuit (U_ω) to perform the transformation:

$$U_\omega |x\rangle |0\rangle |\text{out}\rangle = |x\rangle |0\rangle |\text{out} \oplus f(x)\rangle$$

If we set the `out0` qubit to the superposition state $|-\rangle$ we have:

$$\begin{aligned} U_\omega |x\rangle |0\rangle |-\rangle &= \\ U_\omega |x\rangle |0\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) &= \\ |x\rangle |0\rangle \otimes \frac{1}{\sqrt{2}}(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle) & \end{aligned}$$

If $f(x) = 0$, then we have the state:

$$\begin{aligned} &= |x\rangle |0\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \\ |x\rangle |0\rangle |-\rangle & \end{aligned}$$

(i.e. no change). But if $f(x) = 1$ (i.e. $x = \omega$), we introduce a negative phase to the $|-\rangle$ qubit:

$$\begin{aligned} &= |x\rangle |0\rangle \otimes \frac{1}{\sqrt{2}}(|1\rangle - |0\rangle) = \\ |x\rangle |0\rangle \otimes -\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) &= -|x\rangle |0\rangle |-\rangle \end{aligned}$$

This is a functioning oracle that uses two auxiliary registers in the state $|0\rangle |-\rangle$:

$$U_\omega |x\rangle |0\rangle |-\rangle = \begin{aligned} &|x\rangle |0\rangle |-\rangle \quad \text{for } x \neq \omega \\ &-|x\rangle |0\rangle |-\rangle \quad \text{for } x = \omega \end{aligned}$$

To adapt our checking circuit into a Grover oracle, we need to guarantee the bits in the second register (`c`) are always returned to the state $|0000\rangle$ after the computation. To do this, we simply repeat the part of the circuit that computes the clauses which guarantees `c0 = c1 = c2 = c3 = 0` after our circuit has run. We call this step '*uncomputation*'.

```
In [ ]: var_qubits = QuantumRegister(4, name='v')
        clause_qubits = QuantumRegister(4, name='c')
        output_qubit = QuantumRegister(1, name='out')
        cbits = ClassicalRegister(4, name='cbits')
        qc = QuantumCircuit(var_qubits, clause_qubits, output_qubit, cbits)

def sudoku_oracle(qc, clause_list, clause_qubits):
    # Compute clauses
    i = 0
    for clause in clause_list:
        XOR(qc, clause[0], clause[1], clause_qubits[i])
        i += 1

    # Flip 'output' bit if all clauses are satisfied
    qc.mct(clause_qubits, output_qubit)

    # Uncompute clauses to reset clause-checking bits to 0
    i = 0
    for clause in clause_list:
        XOR(qc, clause[0], clause[1], clause_qubits[i])
        i += 1

sudoku_oracle(qc, clause_list, clause_qubits)
qc.draw()
```

In summary, the circuit above performs:

$$U_{\omega} |x\rangle |0\rangle |\text{out}_0\rangle = \Bigg\{ \begin{aligned} &|x\rangle |0\rangle |\text{out}_0\rangle \quad \text{for } x \neq \omega \\ &|x\rangle |0\rangle |1\rangle \quad \text{for } x = \omega \end{aligned} \Bigg\}$$

and if the initial state of $|\text{out}_0\rangle = |-\rangle$:

$$U_{\omega} |x\rangle |0\rangle |-\rangle = \Bigg\{ \begin{aligned} &|x\rangle |0\rangle |-\rangle \quad \text{for } x \neq \omega \\ &|x\rangle |0\rangle |+\rangle \quad \text{for } x = \omega \end{aligned} \Bigg\}$$

5.3 The Full Algorithm

All that's left to do now is to put this oracle into Grover's algorithm!

```
In [ ]: var_qubits = QuantumRegister(4, name='v')
        clause_qubits = QuantumRegister(4, name='c')
        output_qubit = QuantumRegister(1, name='out')
        cbits = ClassicalRegister(4, name='cbits')
        qc = QuantumCircuit(var_qubits, clause_qubits, output_qubit, cbits)

# Initialize 'out0' in state |->
qc.initialize([1, -1]/np.sqrt(2), output_qubit)

# Initialize qubits in state |s>
qc.h(var_qubits)
```



```

qc.barrier() # for visual separation

## First Iteration
# Apply our oracle
sudoku_oracle(qc, clause_list, clause_qubits)
qc.barrier() # for visual separation
# Apply our diffuser
qc.append(diffuser(4), [0,1,2,3])

## Second Iteration
sudoku_oracle(qc, clause_list, clause_qubits)
qc.barrier() # for visual separation
# Apply our diffuser
qc.append(diffuser(4), [0,1,2,3])

# Measure the variable qubits
qc.measure(var_qubits, cbits)

qc.draw(fold=-1)

```

```

In [ ]: # Simulate and plot results
aer_simulator = Aer.get_backend('aer_simulator')
transpiled_qc = transpile(qc, aer_simulator)
qobj = assemble(transpiled_qc)
result = aer_sim.run(qobj).result()
plot_histogram(result.get_counts())

```

There are two bit strings with a much higher probability of measurement than any of the others, `0110` and `1001`. These correspond to the assignments:

```

v0 = 0
v1 = 1
v2 = 1
v3 = 0

```

and

```

v0 = 1
v1 = 0
v2 = 0
v3 = 1

```

which are the two solutions to our sudoku! The aim of this section is to show how we can create Grover oracles from real problems. While this specific problem is trivial, the process can be applied (allowing large enough circuits) to any decision problem. To recap, the steps are:

1. Create a reversible classical circuit that identifies a correct solution
2. Use phase kickback and uncomputation to turn this circuit into an oracle
3. Use Grover's algorithm to solve this oracle

6. References

1. L. K. Grover (1996), "A fast quantum mechanical algorithm for database search", Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC 1996), [doi:10.1145/237814.237866](https://doi.org/10.1145/237814.237866), [arXiv:quant-ph/9605043](https://arxiv.org/abs/quant-ph/9605043)
2. C. Figgatt, D. Maslov, K. A. Landsman, N. M. Linke, S. Debnath & C. Monroe (2017), "Complete 3-Qubit Grover search on a programmable quantum computer", Nature Communications, Vol 8, Art 1918, [doi:10.1038/s41467-017-01904-7](https://doi.org/10.1038/s41467-017-01904-7), [arXiv:1703.10535](https://arxiv.org/abs/1703.10535)
3. I. Chuang & M. Nielsen, "Quantum Computation and Quantum Information", Cambridge: Cambridge University Press, 2000.

```
In [ ]: import qiskit.tools.jupyter
        %qiskit_version_table
```