The Applications and Limits of Grover's Algorithm

Xiaoxuan Lu, Xiaoyu Wu, Yiming Li

Abstract

Classical algorithms for searching in unstructured data are mostly inefficient. However, quantum algorithms such as Grover's algorithm can speed up the search process by utilizing quantum properties. The superiority of Grover's algorithm, especially in the case of large datasets, has brought the algorithm massive attention in the field of quantum information. In this paper, we explore the application of Grover's algorithm in search and satisfiability problems, showing how quantum circuits are built compared with the classical approach and what the results entail after completing the circuits. We demonstrate that Grover's algorithm can obtain solutions with a quadratic speed-up by using amplitude amplification. In the end, we discuss the limits and challenges of achieving quantum speedups in real quantum computers.

Keywords: Grover's algorithm, unstructured search, satisfiability problem, quadratic speed-up, Qiskit, quantum circuit

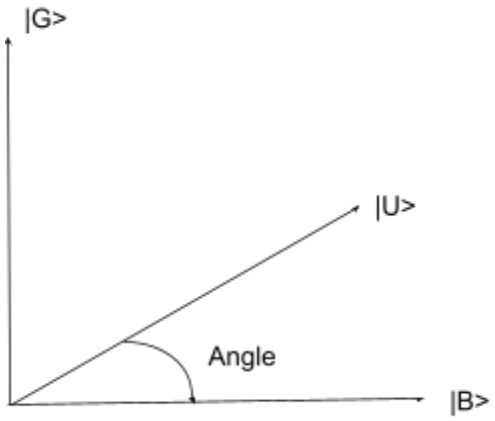
Introduction

Searching for an ample search space with many elements is a complex problem. A classical approach to a search problem is to iterate through each element successively. For example, imagine we have an unstructured linear list with N numbers. We need to find a unique number inside the list, called **P**. In the worst case, the time of finding **P** would be N steps. This occurs when **P** is the last element in the list. Grover's amplitude amplification method reveals that **P** can be found in √N steps on a quantum computer. Unlike other quantum algorithms, such as Shor's algorithm, which yields an exponential speedup for specific factoring tasks, Grover's search yields a quadratic speedup. With this speed advantage, Grover's algorithms can be combined with many classical algorithms in solving difficult problems, such as approximate counting, shortest path problems, minimum spanning trees, collision problems, and estimating the mean value and median value of a set (Grover, 1996; Ambainis, 2004; Fürer, 2008; Zhuang et al., 2014). Grover's algorithm tackles NP-complete problems by doing exhaustive searches across all feasible solutions. Even though it does not give a polynomial-time solution, this would result in a significant speedup over conventional methods (Zhuang et al., 2014).

In the project, we have implemented Grover's algorithm by using an open-source software development kit, Qiskit. Examples of a two-qubit quantum circuit and a four-qubit quantum circuit are shown to solve trivial search problems. After that, we discuss the application of Grover's algorithm to a set of NP-complete problems, namely the satisfiability problem. Lastly, we discuss the obstacles to achieving quantum speedup in real-life quantum computers.

Implementing Grover's Algorithm

Let's first have a look at the case of Grover's algorithm with 2 qubits. In this particular case, only one rotation is required to rotate the initial state $|U>$ to the required $|G>$. In general, let m be the number of marked states and n be the number of qubits. We have:



Where a*ngle* $\theta = arcsin(\sqrt{\frac{m}{2^n}})$. Therefore, in the case where $n = 2$, we have $\theta = arcsin(\sqrt{\frac{1}{2^2}}) = \frac{\pi}{6}$.

After $t$steps, we should have $(U_d U_o)^t |U> = sin\theta_t |G> + cos\theta_t |B>$ where,

$$\theta_t = (2t + 1)\theta$$

In order to obtain $|G>$, we need $\theta_t = \frac{\pi}{2}$, which with $\theta = \frac{\pi}{6}$ results to $t = 1$. This implies that after $t = 1$ rotation (i.e., iteration), the searched element is found. Let's first implement an example using a specific oracle $|G> = |11>$. The oracle function U_o should satisfy :

$$U_o|U> = U_o \frac{1}{2}(|00> + |01> + |10> + |11>) = \frac{1}{2}(|00> + |01> + |10> - |11>)$$

which can be achieved by a controlled-Z gate. Thus, in this case, our oracle is simply the controlled Z gate (which is made by one CNOT gate between two Hadmanard gates). The following diffuser U_d should satisfy $2|U><U| - I$, and it can be represented with a series of operators $U_d = H^{\otimes 2} Z^{\otimes 2} U_o H^{\otimes 2}$. Figure 1 illustrates this quantum circuit arrangement to output the desired $|G> = |11>$.

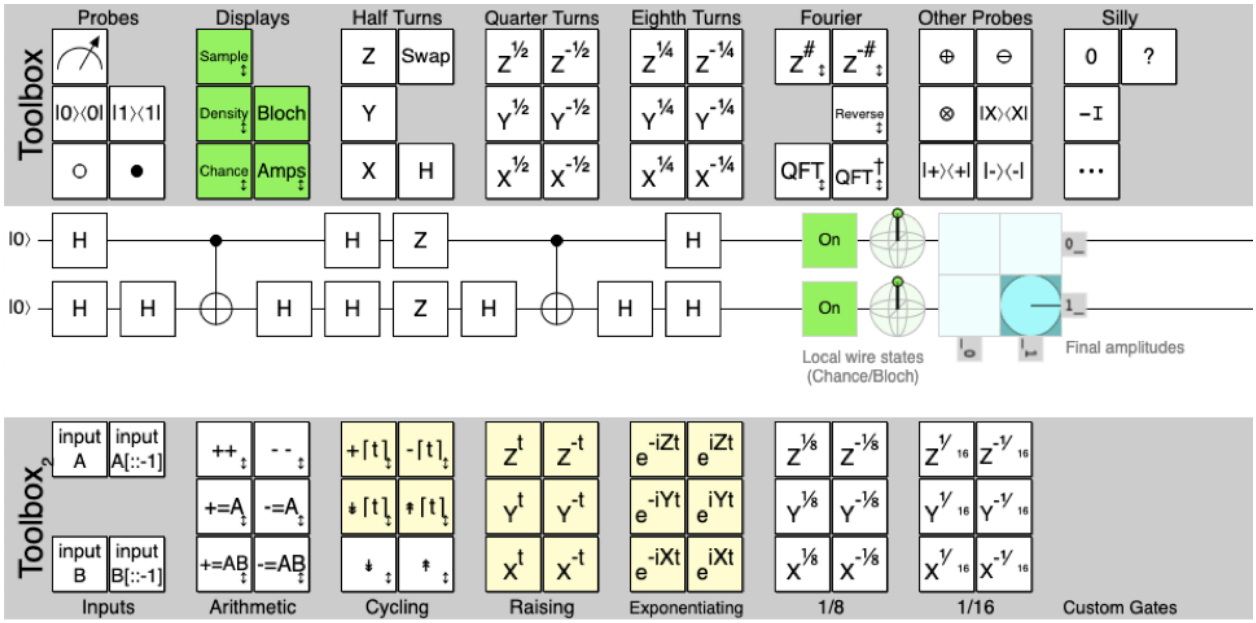


Figure 1: Quantum circuit for Grover's search algorithm to find $|11>$ with Craig's interface.

Now, We can use Qiskit packages to implement Grover's algorithm for the above case of 2 qubits, and the steps are as follows: First, we prepare a quantum circuit with two qubits and initialize the circuit by creating a superposition for the qubits. Then we apply the oracle function U_o and the diffuser U_d accordingly, to complete our circuit. The detailed procedures are shown in Appendix I. Finally, we can run the circuit in simulation. As expected, the amplitude of every state that is not $|11>$ is 0, this means we have a 100% chance of measuring $|11>$:

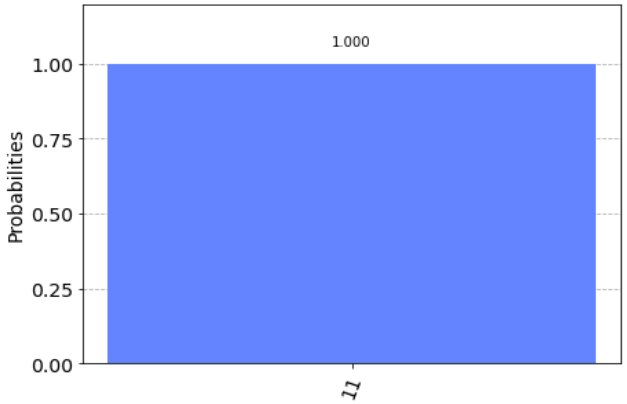


Figure 2: The success probability of finding the marked state $|11>$.

Similarly, we can solve the example of Grover's algorithm for 4 qubits with marked state $|1101>$ by following the quantum circuit shown in Figure 3.

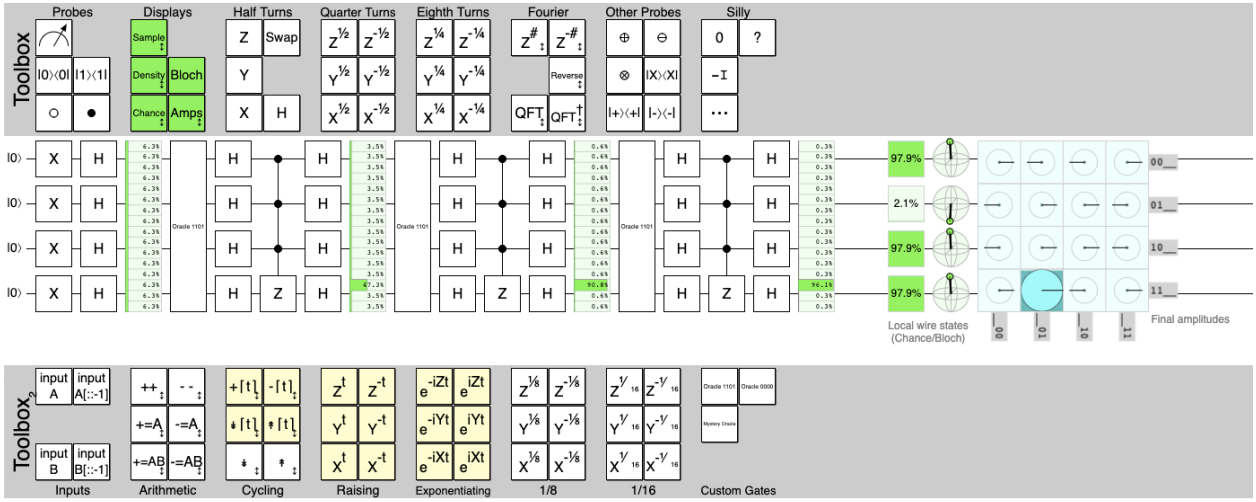


Figure 3: Quantum circuit for Grover's search algorithm to find $|1101>$.

We can follow the same procedure of initializing the qubits first and then constructing a working oracle function and a diffuser. Appendix II contains the code implementation for finding the state $|1101>$. It is worth noting that the algorithm discovers our marked state $|1101>$ only with the probability of 45.3%. To enhance this probability, we need to find the optimal number of iterations. Here $m = 1$, $n = 4$, $\theta = arcsin(\sqrt{\frac{1}{2^4}})$ and we also have $(2t + 1)\theta = \theta_t = \frac{\pi}{2}$, so we anticipate the optimal iteration number would be $t = 3$ times. As a result, the simulation produces $|1101>$ with a probability of 97%.

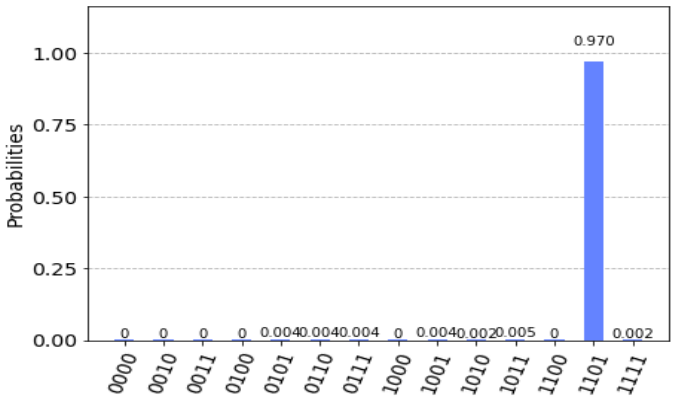


Figure 4: The probability of finding each state.

Additionally, It is worth comparing these 2-qubits and 4-qubits quantum approaches of searching elements to the classical approach. This way, we can demonstrate the advantage of quantum algorithms in time complexity over the classical ones. To do so, we first translate the problem in the classical term. The example with 2 qubits is a search problem of finding the last element ($|11>$) in a list which $N = 4$. The 4-qubits example is set up aiming to find the 13th element ($|1101>$) when $N = 16$. Then, we set up the oracle in the classical case. This oracle is simply doing interactions over every element in the list one at a time and returns TRUE when it finds the target.

To apply the oracle over these two examples, we see the classical algorithm was called 4 times and 13 times to find the target (see Appendix III). While in the quantum case, the oracle only gets called 1 time in the case of 2 qubits to find the target ($|11>$) with 100% probability and 3 times for it to have the probability of 97% in finding the target ($|1101>$). In both cases, it is evident that the glover algorithm performs better since the oracle is called fewer times than it was needed in the classical approach. On average, the complexity of the search problem in the classical approach requires the oracle to be called N/2 times. In the worst-case scenario, the oracle needs to be called N times. On the other hand, by using Glover's algorithm, we see that the quantum approach takes O(√N) times**.** It reflects the point that for solving search problems, the quantum algorithm has more advantages in terms of complexity.

So far, we have only searched for one marked state; hence, we also include another interesting implementation to find two desired elements in Appendix IV, and Grover's algorithm again produces the desired outputs with just two iterations.

Solving Satisfiability Problems using Grover's Algorithm

Despite using Grover's algorithm in search problems where we know what we expect, we can also use it to solve satisfiability problems for which we do not necessarily know the solutions beforehand.

The satisfiability (SAT) problem asks that, given a boolean expression, if there is a solution that when all variables in the expression are replaced by boolean values (TRUE or FALSE), the expression would still evaluate to TRUE. In fact, SAT is the first problem that was proven to be NP-complete and is important in the fields such as artificial intelligence and circuit design (Roli & Milano, 2011).

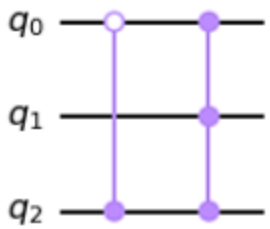
We look more into one specific class of SAT problems, namely the k-SAT problem. The k-sat problems refer to logical expressions that satisfy the structure consisting of conjunctions of individual clauses and within each clause are exactly k variables that are connected by the disjunction operator. The number of variables in each clause is normally reflected in the name. For example, 2-SAT means that within each clause, there are 2 variables. It is worth noting that this does not guarantee information about how many variables there are in the entire expression.

The advantages of quantum speedup in solving unstructured search problems, as we mentioned in the previous section, should still be applicable here with SAT. Here we face the problem of finding an assignment of one of 2 values (TRUE or FALSE) to each of n variables so that the time complexity is O($2^{n/2}$) (Cerf et al., 2000). However, it does not guarantee that Grover's algorithm is the fastest compared to all classical algorithms. For example, for 3-SAT problems, the fastest algorithm is a classical algorithm with a complexity of O(1.307^n) at the moment. Still, Grover's algorithm can outperform classical ones in the general case (k-SAT) and theoretically there is also potential of using Grover's algorithms in combination with classical algorithms to achieve a better speedup altogether (Hansen et al., 2019).

To demonstrate a simple application of Grover's algorithm for a 2-SAT problem, we set up a logical expression with three variables as below,

$$(A \vee C) \wedge (\neg A \vee B) \wedge (\neg B \vee C)$$

With this example problem input, we create the corresponding oracle for our Glover search as follows:



Then, we finish constructing the circuit with the initialization step and the diffuser, as shown in Figure 5, and run it. The results as shown in Figure 6 where all combinations of the variables are evaluated. Solutions that satisfy the logical expression are the ones with high probabilities associated with them. All implementation code in Qiskit is attached in Appendix V.

The results can be interpreted with the state of each qubit representing the value of the corresponding variable. The suggested solution $|100>$ means that C should be set to TRUE, B and A should be set to FALSE. Similarly, $|110>$ equals $\{A = FALSE, B = TRUE, C = TRUE\}$ and $|111>$ means all three variables should be set to TRUE. If we check the proposed solutions traditionally (e.g., with truth tables), we can see that these are indeed the satisfying solutions.

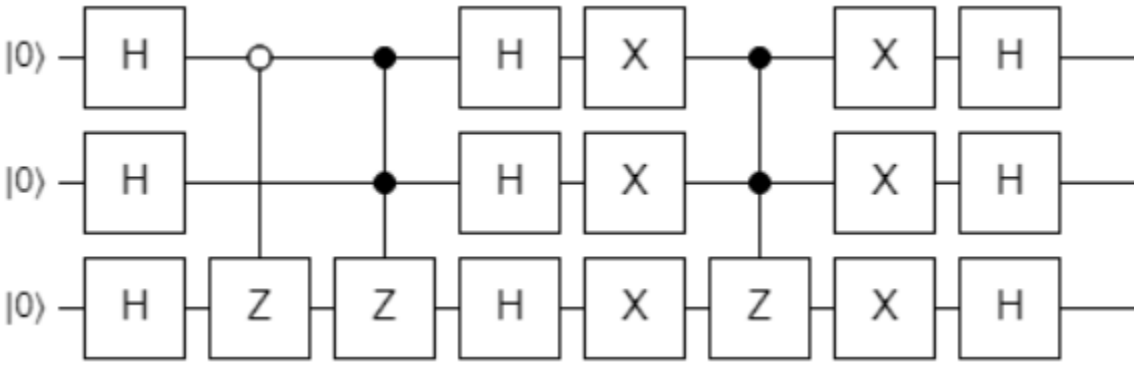


Figure 5: Overview of the circuit for the example 2-SAT problem

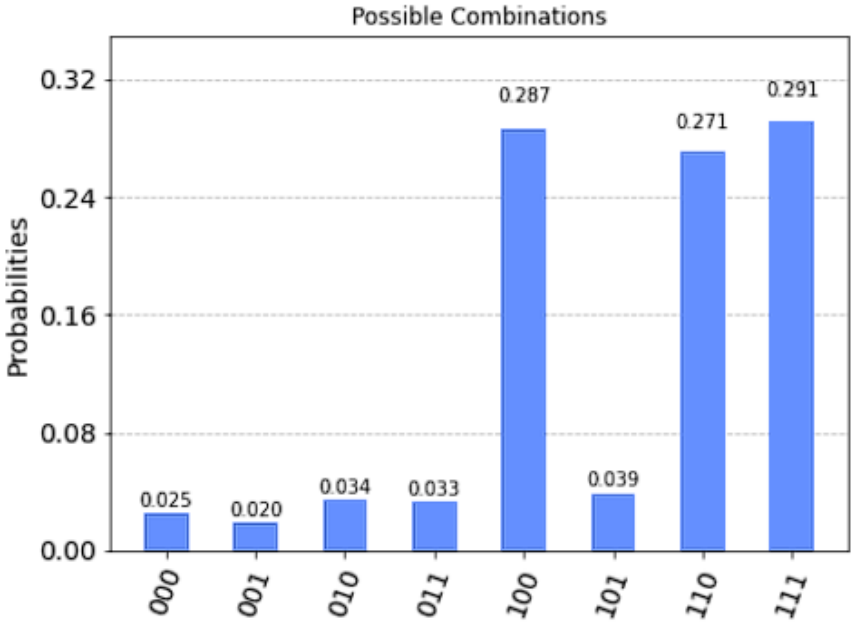


Figure 6: the results of running circuit simulations

Harsh Reality: the Limits of Quantum Computers

Using quantum algorithms like Grover's algorithm, quantum computers can be incredibly fast at specialized jobs such as unstructured search. However, it appears that there are real issues standing in the way of them being more efficient than today's classical computers. The primary issue is that quantum error correction and device operating time contribute considerable constant-factor algorithm runtime slowdowns. The practical realization of useful fault-tolerant devices is hampered by these substantial overhead costs (Babbush et al., 2021). Babbush et al. (2021) made numerous optimistic assumptions about the quantum computer but nonetheless concluded that the probability of quadratic speedups with current error-correcting codes and architectures outperforming conventional computers in time-to-solution is low. To achieve a quantum advantage with decent fault-tolerant resources, one must either go beyond quadratic speedups or significantly improve error-correction algorithms, or both.

Specifically, we can define the runtime for quantum and classical algorithms as $T_q = \sqrt{N} * t_q$ and $T_c = N * t_c$, where T is the total runtime of an algorithm. It is known in Grover's algorithm, it takes $\sqrt{N}$ steps comparing to the classical algorithms with N steps in the worst case. For achieving a quantum advantage, it must satisfy $T_q < T_c$, thus $N > (\frac{t_q}{t_c})^2$. Therefore, it will require sufficient calls N for a quantum advantage to be attainable, suggesting that the classic speed and the correction mechanism of quantum computers will make the quantum computer slower than classical ones if N is small. Possible solutions are to make T_q smaller by building better quantum computers, achieving better error-correction algorithms, or inventing search algorithms that go beyond quadratic speedups.

Conclusion and Discussion

We saw that employing techniques of Grover's algorithm in quantum computing can accelerate the process in unstructured search. Grover's algorithm can deliver a high probability of generating certain outputs by performing only $O(\sqrt{N})$ function evaluations, where N is the size of the function's domain. In this research, we used Qiskit to implement a 2 qubits' search, two 4 qubits' searches, and also a satisfiability problem. In each example, we confirmed the effectiveness of using Grover's algorithm on search problems compared with classical approaches in an ideal setting. Nonetheless, Grover's algorithm still has many limitations that are yet to be solved to truly achieve a significant speed-up.

Our reseach also has drawbacks. For example, we only use a small number of qubits to construct our quantum circuits. In future research. we could try to include more qubits to give a more comprehensive view on the algorithm's capability. Additionally, our quantum circuits were evaluated using a fixed simulator (i.e., AER simulator for unstructured search and QASM simulator for satisfiability problems), and it might be worthwhile to compare the success probability by using different simulators or run the circuits in different shots to generalize the results.

Reference

1. Ambainis, A. (2004). Quantum search algorithms. SIGA. https://doi.org/10.1145/992287.992296
2. Babbush, R., McClean, J. R., Newman, M., Gidney, C., Boixo, S., & Neven, H. (2021). Focus beyond Quadratic Speedups for Error-Corrected Quantum Advantage. *PRX Quantum*, *2*(1). https://doi.org/10.1103/prxquantum.2.010103
3. Cerf, N. J., Grover, L. K., & Williams, C. P. (2000). Nested Quantum Search and NP-Hard Problems. *Applicable Algebra in Engineering, Communication and Computing*, *10*(4-5), 311–338. https://doi.org/10.1007/s002000050134
4. Fürer, M. (2008). Solving NP-complete Problems with Quantum Search. In E. S. Laber, C. Bornstein, L. T. Nogueira, & L. Faria (Eds.), *LATIN 2008: Theoretical Informatics* (pp. 784–792). Springer Berlin Heidelberg.
5. Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. *ArXiv:quant-Ph/9605043*. https://arxiv.org/abs/quant-ph/9605043
6. Hansen, T. D., Kaplan, H., Zamir, O., & Zwick, U. (2019). *Faster k-SAT algorithms using biased-ppsz*. 578–589. https://doi.org/10.1145/3313276.3316359
7. Roli, A., & Milano, M. (2011). Solving the satisfiability problem through boolean networks. *CoRR*, *abs/1101.6009*. http://arxiv.org/abs/1101.6009
8. Singhal, A., & Chatterjee, A. (2018). *Grover's algorithm*. https://doi.org/10.13140/RG.2.2.30860.95366
9. Zhuang, J., Zhao, J., Xu, F., Hu, H., & Qiao, P. (2014). Analysis and Simulation of Grover's Search Algorithm. *International Journal of Machine Learning and Computing*, 21–23. https://doi.org/10.7763/ijmlc.2014.v4.380
10. The Qiskit Team. (2022, April 28). *Grover's Algorithm*. Qiskit.org; Data 100 at UC Berkeley. https://qiskit.org/textbook/ch-algorithms/grover.html#6.-References-

Appendix I

The procedures for implementing 2-qubit's Grover algorithm with Qiskit (The Qiskit Team, 2022):

```
#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
```

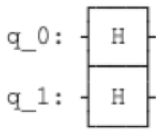
We start by preparing a quantum circuit with two qubits:

```
# prepare a quantum circuit with 2 qubits

n = 2
grover_circuit = QuantumCircuit(n)
```

Then we prepare a superposition for the qubits by applying each of them a Hadamard gate:

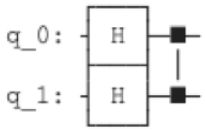
```
def initialize_s(qc, qubits):
    """Apply a H-gate to 'qubits' in the circuit"""
    for q in qubits:
        qc.h(q)
    return qc
```

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



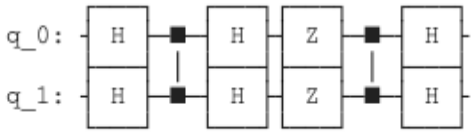
Apply the oracle U_o for |G> = |11>, note that there is a CZ gate in the Qiskit register, so we don't need to implement H-CNOT-H gates separately.

```
# apply oracle function

grover_circuit.cz(0,1) # Oracle
grover_circuit.draw()
```



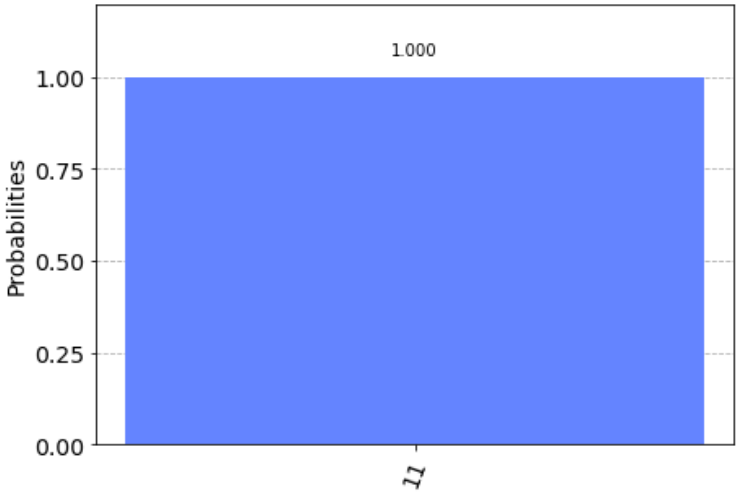
We now want to apply the diffuser U_d

```
# amplification step (after one iteration)
grover_circuit.h([0,1])
grover_circuit.z([0,1])
grover_circuit.cz(0,1) # H-CNOT-H
grover_circuit.h([0,1])
grover_circuit.draw()
```



This is our finished circuit, and we can run the circuit on the aer_simulator.

```
# run on simulations

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)
```



Appendix II

The procedures for implementing 4-qubit's Grover algorithm to find state |1101> with Qiskit:

The first step is still initializing the qubits:

```
def new_initialize_s(qc, qubits):
    """Apply a X-gate and H-gate to 'qubits' in the circuit"""
    for q in qubits:
        qc.x(q)
        qc.h(q)
    return qc
```

Then we build an oracle function that searches |1101>, and gives it the name U_omega.

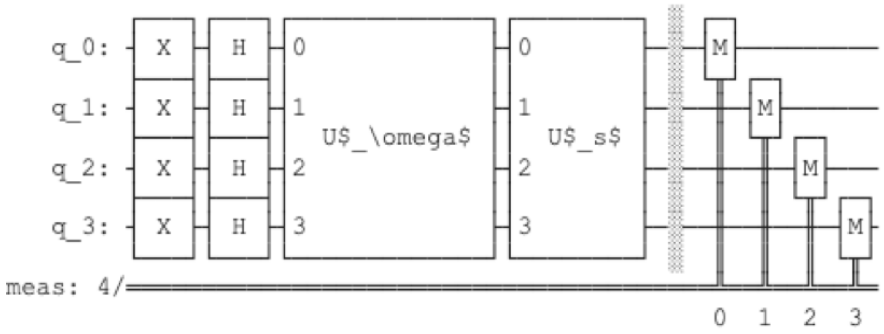
```
# build oracle function |1101>

qc = QuantumCircuit(4)
qc.x(1)
qc.h(3)
qc.mct(list(range(3)), 3)  # multi-controlled-toffoli
qc.h(3)
qc.x(1)
oracle_ex3 = qc.to_gate()
oracle_ex3.name = "U$_\omega$"
```

We will create a general diffuser for any number of qubits in this setting, and name is U_s.

```
# build diffuser

def diffuser(nqubits):
    qc = QuantumCircuit(nqubits)
    # Apply transformation |s> -> |00..0> (H-gates)
    for qubit in range(nqubits):
        qc.h(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 |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
```

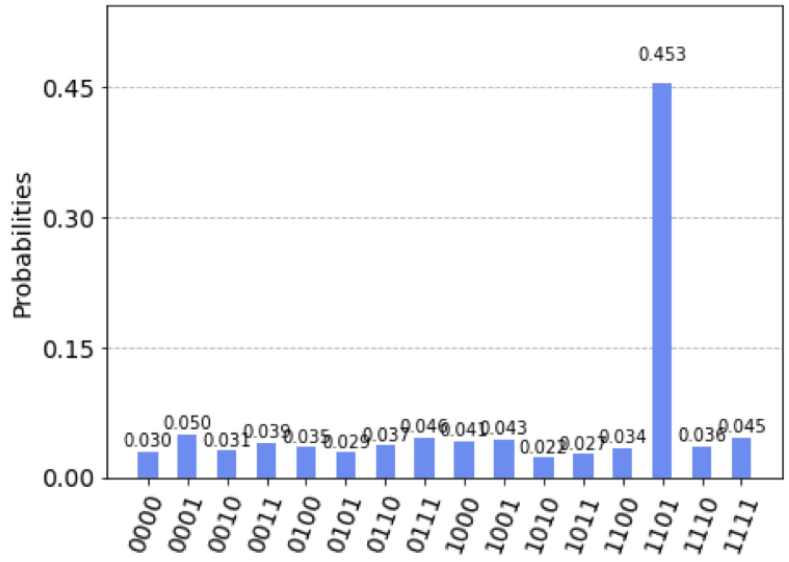
We'll now put the pieces together, with the creation of a uniform superposition at the start of the circuit and measurement at the end. Note that this is a circuit that only runs for one iteration.

```
n=4
grover_circuit = QuantumCircuit(n)
grover_circuit = new_initialize_s(grover_circuit, [0,1,2,3])

# first iteration
grover_circuit.append(oracle_ex3, [0,1,2,3])
grover_circuit.append(diffuser(n), [0,1,2,3])
grover_circuit.measure_all()
grover_circuit.draw()
```



We can run the above circuit on the simulator.

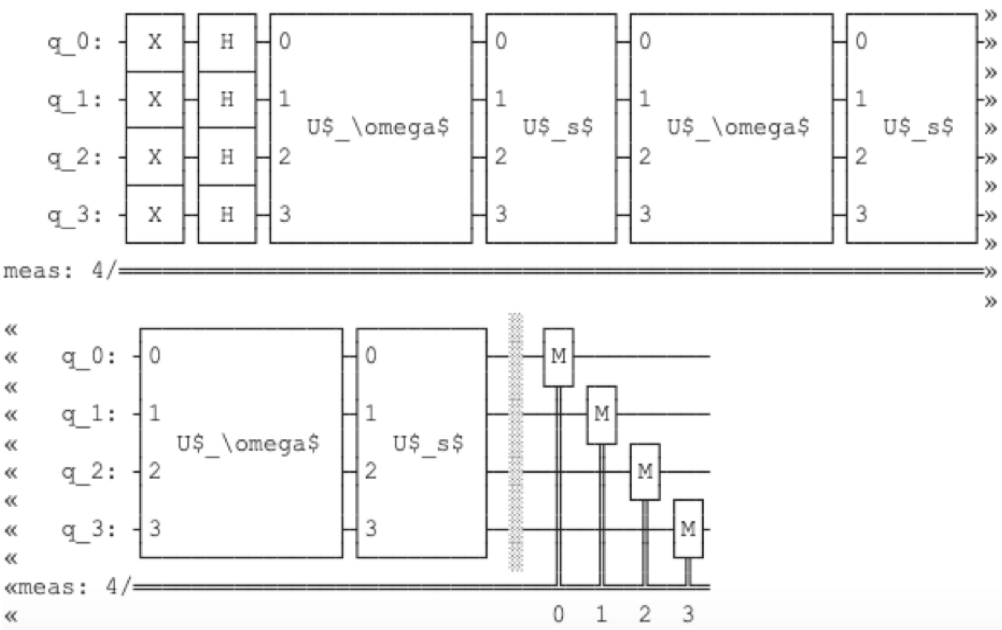
```
# experiment with simulator (with one iteration)

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)
```



To enhance the probability, we iterate three times the same circuit:

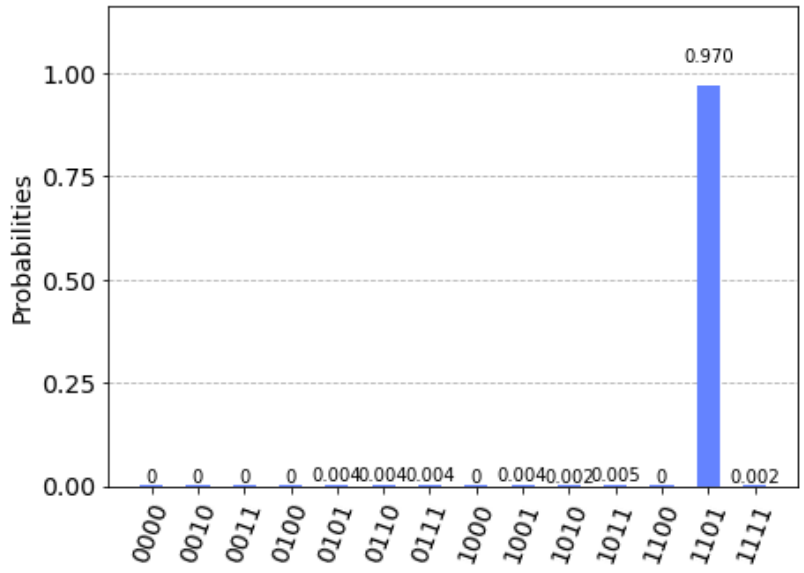
```
# optimal numbers of iterations (3 times?)

n=4
grover_circuit = QuantumCircuit(n)
grover_circuit = new_initialize_s(grover_circuit, [0,1,2,3])

counter = 0

while counter < 3:
    grover_circuit.append(oracle_ex3, [0,1,2,3])
    grover_circuit.append(diffuser(n), [0,1,2,3])
    counter += 1

if counter == 3:
    print ("iteration completed")

grover_circuit.measure_all()
grover_circuit.draw()
```

```
iteration completed
```



Then, we run the above circuit on the simulator again.

```
aer_sim = Aer.get_backend('aer_simulator',shots=1024)
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)
```



Appendix III

In the classical case, the oracle is defined as follows in code,

```
def oracle(input, target):
    if input is target:
        return True
    else:
        return False
```

In the example of finding the target in a list (|11>) when N=4, the oracle needs to be called a total of 4 times as demonstrated below.

```
lst_4 = [1, 2, 3, 4]
TARGET_4 = 4
```

```
for index, element in enumerate(lst_4):
    if oracle(element, TARGET_4) is True:
        print("Target found at index %i."%index)
        print("Oracle is called %i times."%(index+1))
        break
```

```
Target found at index 3.
Oracle is callsed 4 times.
```

In the case of N=16, the classical oracle needs to be called 13times in order to find the target (|1101>) as shown in the following.

```
lst_16 = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
TARGET_16 = 13

for index, element in enumerate(lst_16):
    if oracle(element, TARGET_16) is True:
        print("Target found at index %i."%index)
        print("Oracle is called %i times."%(index+1))
        break
```

```
Target found at index 12.
Oracle is called 13 times.
```

Appendix IV

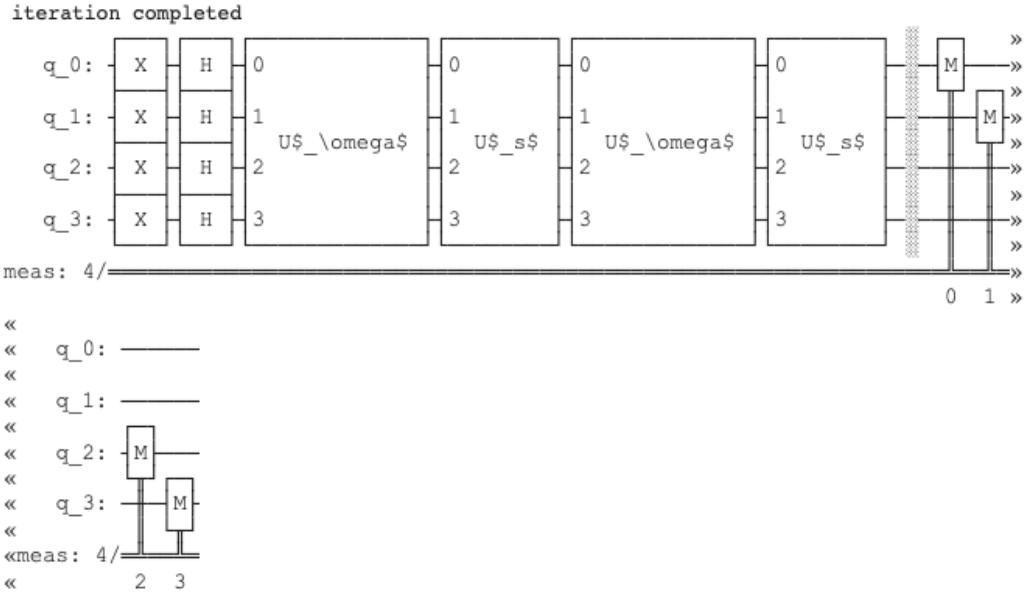
The procedures for implementing 4-qubit's Grover algorithm to two marked states: |1100> and |1110> are as follows:

We use the "new_initialize_s" function mentioned in the above example to create a superposition for all qubits, and then create our oracle functions separately for |1100> and |1110>, and pass the general diffuser in the last example to set up the operations for one iteration. Finally, we need to determine the number of iterations needed to complete the circuit. When m=2, n=4, we have $\theta = arcsin(\sqrt{\frac{2}{2^4}})$ and (2t+1)θ= $\theta_t = \frac{\pi}{2}$, so t=2 in this case. Therefore, we set the "iteration" to be two and loop the operation twice to get the finished circuit.

```
n=4
grover_circuit = QuantumCircuit(n)
grover_circuit = new_initialize_s(grover_circuit, [0,1,2,3])

counter = 0
iteration = 2

# build oracle functions |1100> and |1110>

# for |1100>
qc = QuantumCircuit(4)
qc.x(0)
qc.x(1)
# CZ gates for 4 qubits
qc.h(3)
qc.mct(list(range(3)), 3)  # multi-controlled-toffoli
qc.h(3)
qc.x(0)
qc.x(1)

# for |1110>
qc.x(0)
qc.h(3)
qc.mct(list(range(3)), 3)  # multi-controlled-toffoli
qc.h(3)
qc.x(0)

oracle_ex4 = qc.to_gate()
oracle_ex4.name = "U$_\omega$"
```

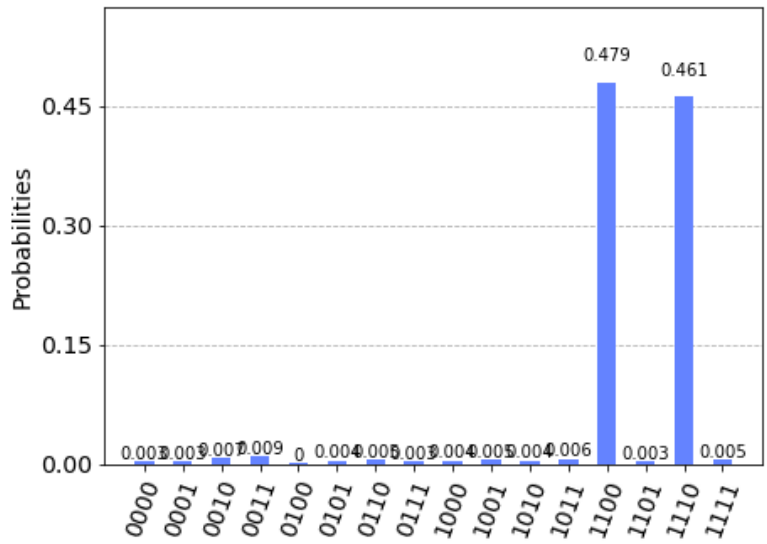
```
while counter < iteration:
    grover_circuit.append(oracle_ex4, [0,1,2,3])
    grover_circuit.append(diffuser(n), [0,1,2,3])
    counter += 1

if counter == iteration:
    print ("iteration completed")

grover_circuit.measure_all()
grover_circuit.draw()
```

```
iteration completed
```



Running the above circuit on the simulator gives two marked state, with 47.9% of finding |1100> and 46.1% of finding |1110>, and all other states can be neglected because of their low probabilities.

```
aer_sim = Aer.get_backend('aer_simulator',shots=1024)
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)
```



Appendix V

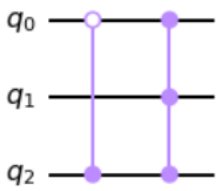
The complete code of implementing a Grover solution to a 2-SAT problem in Qiskit.

```
from qiskit import Aer
from qiskit.algorithms import Grover, AmplificationProblem
from qiskit.circuit.library import PhaseOracle
from qiskit.tools.visualization import plot_histogram
from qiskit.utils import QuantumInstance
```

```
backend = Aer.get_backend('qasm_simulator')
qinstance = QuantumInstance(backend, shots=1024)
```

```
oracle = PhaseOracle('((~A | B)) & (A | C)  & (~B | C)')
problem = AmplificationProblem(oracle=oracle, is_good_state=oracle.evaluate_bitstring
grover = Grover(quantum_instance=qinstance)
result = grover.amplify(problem)
```

```
oracle.draw(output='mpl')
```




```
plot_histogram(result.circuit_results, title='Possible Combinations', bar_labels=True
```