Projeto

June 23, 2022

1 Grupo 5

Pedro Paulo Costa Pereira - A88062

Tiago André Oliveira Leite - A91693

[2]:
```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, Aer,␣
↪execute, IBMQ
from qiskit.tools.visualization import plot_histogram, visualize_transition
from qiskit.tools import job_monitor

from qiskit.utils import QuantumInstance
from qiskit.algorithms import Grover, AmplificationProblem
from qiskit.circuit.library import PhaseOracle

import matplotlib.pyplot as plt
import numpy as np
```

1.1 Formula booleana 3-SAT

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

1.1.1 Verificar que tem solução

Seja:

$f_1 = A \vee B \vee C$

$f_2 = A \vee B \vee \neg C$

$f_3 = A \vee \neg B \vee C$

$f_4 = A \vee \neg B \vee \neg C$

$f_5 = \neg A \vee B \vee C$

$f_6 = \neg A \vee B \vee \neg C$

$f_7 = \neg A \vee \neg B \vee \neg C$

A	B	C	f_1	f_2	f_3	f_4	f_5	f_6	f_7	F
1	1	1	0	1	1	1	1	1	1	0
1	1	0	1	1	1	1	1	1	0	0
1	0	1	1	1	1	1	1	0	1	0
1	0	0	1	1	1	1	0	1	1	0
0	1	1	1	1	1	0	1	1	1	0
0	1	0	1	1	0	1	1	1	1	0
0	0	1	1	0	1	1	1	1	1	0
0	0	0	1	1	1	1	1	1	1	1

Solução:

$A = 0 \qquad B = 0 \qquad C = 0$

1.2 Algoritmo de resolução

1.2.1 Defnir estado inicial

No estado inicial aplicamos uma porta *Hadamard* aos 3 *qubits* que representam os literais e aplicamos uma porta X seguida de uma porta *Hadamard* aos *qubits* ancilla.

Por cada cláusula da fórmula existe um qubit *ancilla*, sendo que ainda existe um *qubit ancilla* extra para representar a fórmula F na sua totalidade.

[3]:
```
def init(n,a):
    qr = QuantumRegister(n)
    ancilla = QuantumRegister(a)
    cr = ClassicalRegister(n)
    qc = QuantumCircuit(qr,ancilla,cr)

    qc.h(qr)
    qc.x(ancilla)
    qc.h(ancilla)
    qc.barrier()

    return qc, qr, ancilla, cr
```

1.2.2 Definir oráculo

Uma vez que o Qiskit não possui uma porta que disponibilize o operador lógico OR, é feita uma dupla negação em cada umas das cláusulas da fórmula F de modo a remover as dijunções.

Fica-se assim com a seguinte fórmula:

$F = \neg(A \wedge B \wedge C) \wedge \neg(\neg A \wedge \neg B \wedge C) \wedge \neg(\neg A \wedge B \wedge \neg C) \wedge \neg(\neg A \wedge B \wedge C) \wedge \neg(A \wedge \neg B \wedge \neg C) \wedge \neg(A \wedge \neg B \wedge C) \wedge \neg(A \wedge B \wedge \neg C)$

Assim sendo para cada clausula da formula resultante usa-se a seguinte estratégia:

1. Se o literal estiver negado, aplicamos uma porta X.

2. Aplicamos uma porta *Multi-Control Toffoli* com os 3 *qubits* que representam os literais da cláusula e em que o *target* é o qubit *ancilla* que representa a clausula em questão.
3. Aplicamos uma porta X ao *qubit ancilla* que representa a cláusula pois esta agora está negada.
4. Revertemos as portas aplicadas no ponto 1.

Depois de aplicar este prodecimento a todas as cláusulas aplicamos uma porta *Multi-Control Toffoli* com os *qubits ancilla* que representam cada cláusula e em que o *target* é o ultimo *qubit ancilla* que representa a fórmula F.

[4]:
```
def oracle(qr, ancilla):
    qc = QuantumCircuit(qr,ancilla)

    # f1
    qc.mcx(qr,ancilla[0])
    qc.x(ancilla[0])

    #f2
    qc.x(qr[0])
    qc.x(qr[1])
    qc.mcx(qr,ancilla[1])
    qc.x(ancilla[1])
    qc.x(qr[0])
    qc.x(qr[1])

    #f3
    qc.x(qr[0])
    qc.x(qr[2])
    qc.mcx(qr,ancilla[2])
    qc.x(ancilla[2])
    qc.x(qr[0])
    qc.x(qr[2])

    #f4
    qc.x(qr[0])
    qc.mcx(qr,ancilla[3])
    qc.x(ancilla[3])
    qc.x(qr[0])


    #f5
    qc.x(qr[1])
    qc.x(qr[2])
    qc.mcx(qr,ancilla[4])
    qc.x(ancilla[4])
    qc.x(qr[1])
    qc.x(qr[2])

    #f6
```

```
    qc.x(qr[1])
    qc.mcx(qr,ancilla[5])
    qc.x(ancilla[5])
    qc.x(qr[1])

    #f7
    qc.x(qr[2])
    qc.mcx(qr,ancilla[6])
    qc.x(ancilla[6])
    qc.x(qr[2])


    #F
    qc.mcx(ancilla[:-1],ancilla[-1])

    qc.barrier()
    return qc
```

1.2.3 Definir difusor

O operador de difusão atua sobre os 3 *qubits* que representam os literais, procedendo da seguinte forma:

1. Aplica uma porta *Hadamard* aos 3 *qubits* para remover o estado de sobreposição.
2. Aplica um porta X aos 3 *qubits*.
3. Aplica uma porta *Multi-Control Toffoli* dos dois primeiros *qubits* ao terceiro.
4. Aplica uma porta *Hadamard* aos 3 *qubits*.

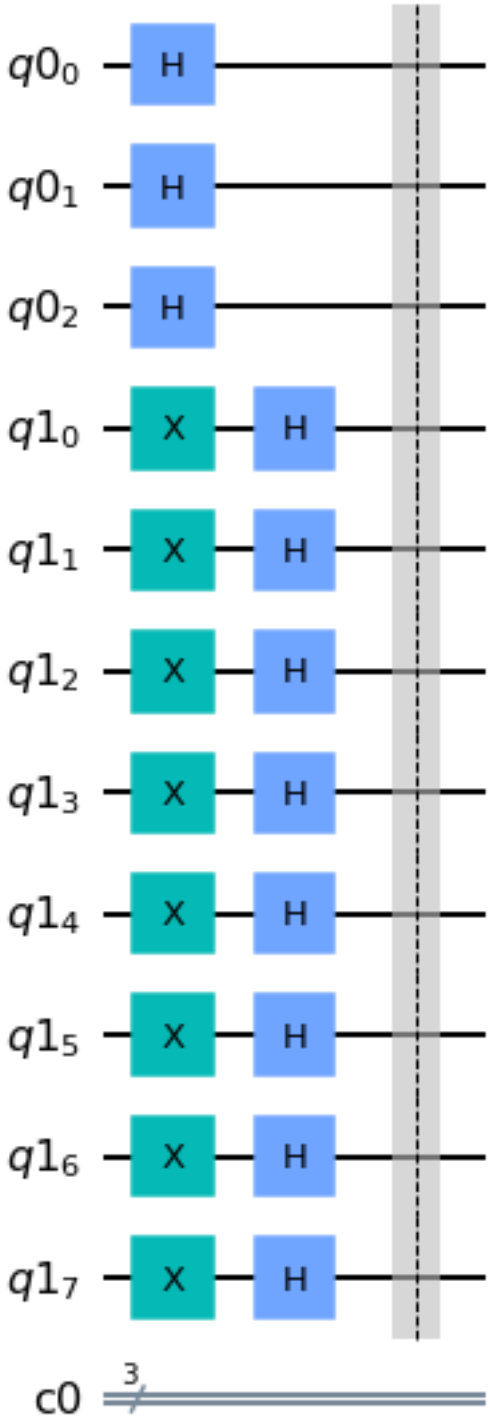
[5]:
```
def diffusion_operator(qr, ancilla):

    qc = QuantumCircuit(qr,ancilla)

    qc.h(qr)

    qc.x(qr)
    qc.h(qr[-1])

    qc.mcx(qr[:-1],qr[-1])

    qc.h(qr[-1])
    qc.x(qr)

    qc.h(qr)

    qc.barrier()

    return qc
```

1.3 Solução

Inicialização do circuito

[6]:
```
n_qubits = 3
n_ancillas = 8
qc, qr, ancilla, cr = init(n_qubits, n_ancillas)
qc.draw("mpl")
```

[6]:



Cálculo no número de iterações necessárias

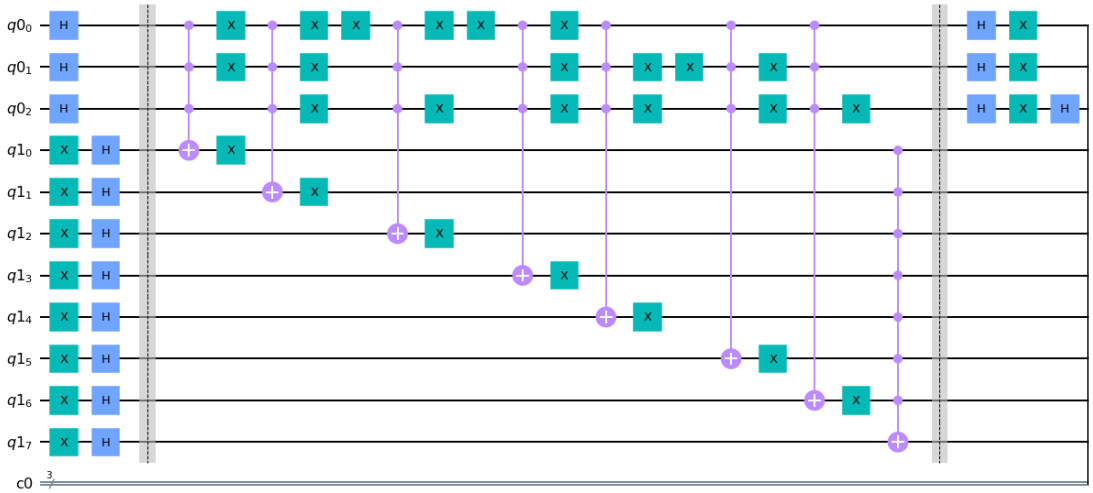
[7]:
```
elements = 2**n_qubits
iterations= int(np.floor(np.pi/4 * np.sqrt(elements)))
iterations
```

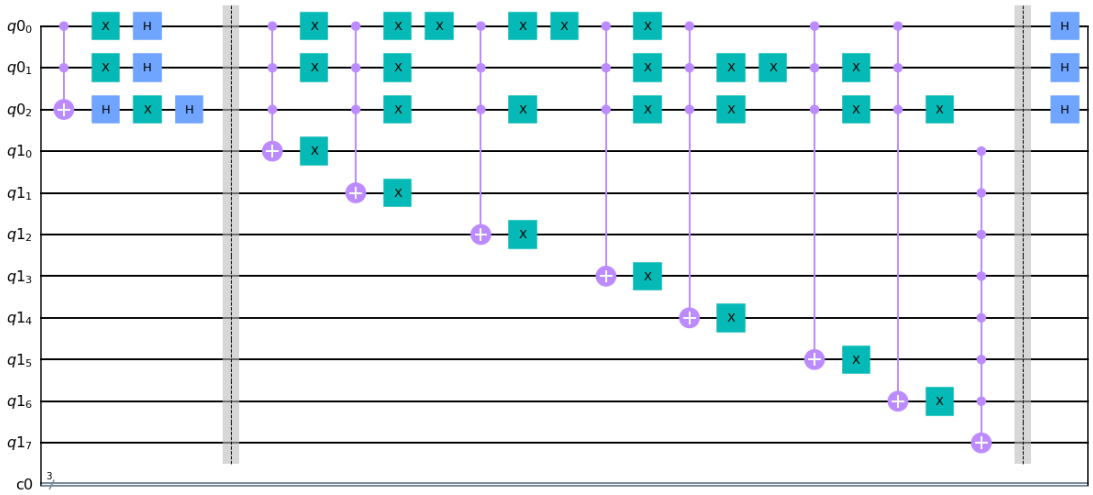
[7]: 2

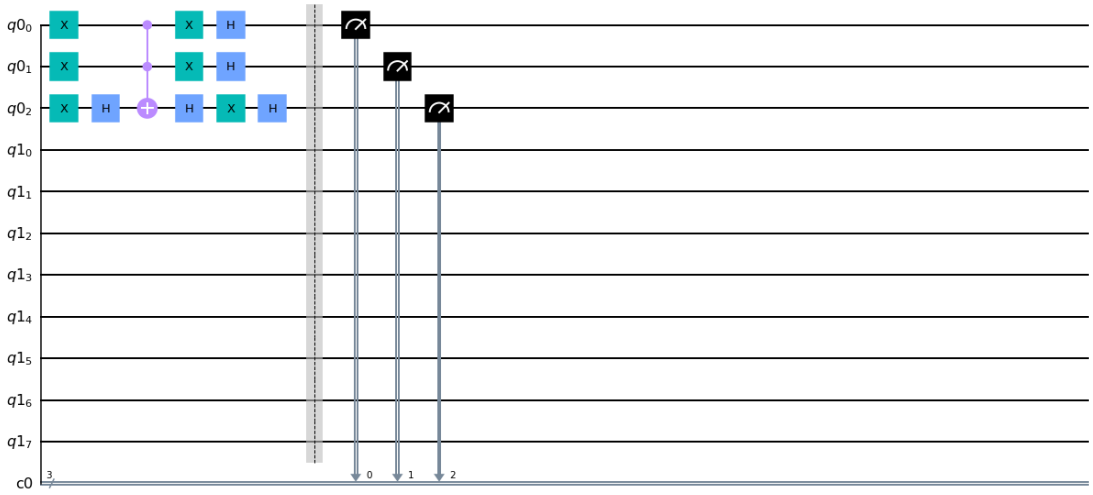
Composição do circuito com oráculo e difusor

[8]:
```
for i in range(iterations):
    qc = qc.compose(oracle(qr,ancilla))
    qc = qc.compose(diffusion_operator(qr,ancilla))

qc.measure(qr,cr)
qc.draw(output="mpl")
```

[8]:


q0₀
q0₁
q0₂
q1₀
q1₁
q1₂
q1₃
q1₄
q1₅
q1₆
q1₇
c0
3


q0₀
q0₁
q0₂
q1₀
q1₁
q1₂
q1₃
q1₄
q1₅
q1₆
q1₇
c0
3

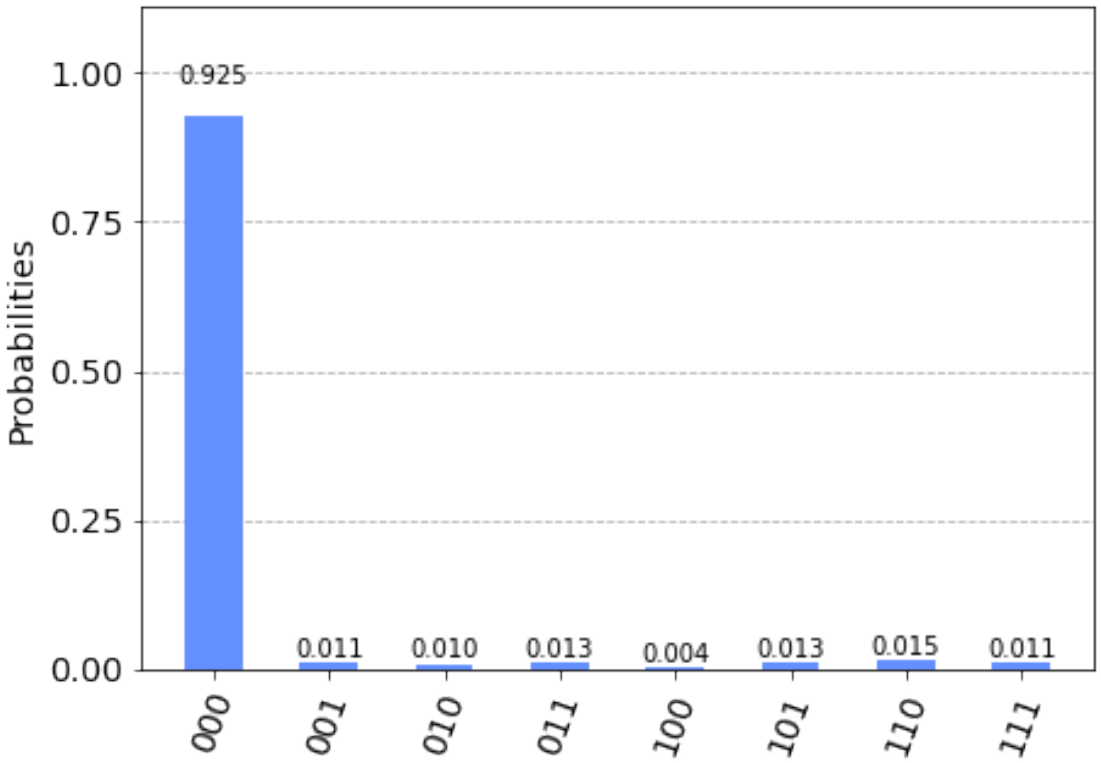

q0₀
q0₁
q0₂
q1₀
q1₁
q1₂
q1₃
q1₄
q1₅
q1₆
q1₇
c0
3
0
1
2

1.3.1 Simulador local

Executar circuito

[9]:
```
backend = Aer.get_backend('qasm_simulator')
counts = backend.run(qc, shots=1024).result().get_counts()
plot_histogram(counts)
```

[9]:



1.3.2 IBM Quantum

Autenticação no IBM Quantum

[9]:
```
#token = 'copiar token da conta IBMQ'
#IBMQ.active_account()
#IBMQ.save_account(token, overwrite=True)
IBMQ.load_account()
```

[9]:
```
<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

Máquinas disponiveis

[10]:
```
#IBMQ.providers()
provider = IBMQ.get_provider()
provider.backends(operational=True)
```

[10]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_armonk') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q',
group='open', project='main')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibm_nairobi') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibm_oslo') from IBMQ(hub='ibm-q', group='open', project='main')>]
```

Selecionar máquina Uma vez que nenhuma das máquinas reais disponiveis tem mais que 7 qubits, teremos que usar um dos simuladores do IBM Quantum.

[11]:
```
backend = provider.get_backend('simulator_statevector')
backend.configuration().n_qubits
```

[11]:
```
32
```

Executar circuito

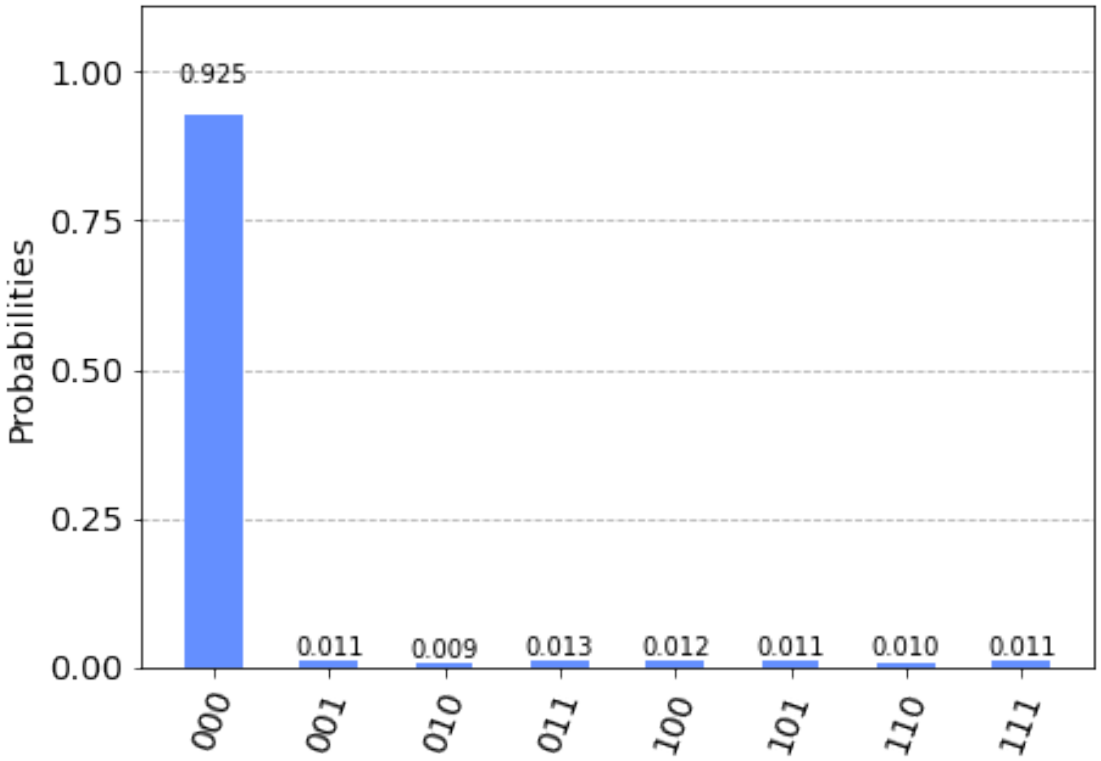
[12]:
```
job = execute(qc,backend)
```

[13]:
```
job_monitor(job)
```

```
Job Status: job has successfully run
```

[15]:
```
result = job.result()
counts = result.get_counts()
plot_histogram(counts)
```

[15]:



1.4 Análise da complexidade do algoritmo

Uma vez que a fŕomula possui 3 literais, a resolução deste tipo de problemas pela forma clássica terá, no pior caso, uma complexidade de $2^3 = 8$.

No entanto, como se pode verificar, a obtenção de uma solução com o algoritmo de *Grover* apenas necessitou de 2 iterações.

Para o caclulo do número de iterações necessarias, temos que ter em conta a seguinte equação:

$\frac{\phi}{2} + r \times \phi = \frac{\pi}{2}$, com $\phi = \arcsin\frac{1}{\sqrt{N}}$ e N $= 2^3$

Resolvendo a equação em relação a r e considerando $\phi \approx sin(\phi)$ para valores pequenos de ϕ , obtemos:

$r \approx \frac{\pi}{4} \times \sqrt{N}$

$\approx O(\sqrt{N})$

Ou seja, para a formula escolhida, a complexidade do algoritmo é aproximadamente $\sqrt{8} \approx 2.83$

1.5 Resolução com Qiskit Aqua

[60]:
```
with open('formula.dimacs', 'r') as f:
    dimacs = f.read()
print(dimacs)
```

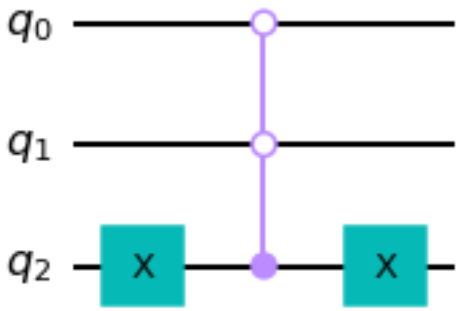
```
c example DIMACS CNF 3-SAT
p cnf 3 5
-1 -2 -3 0
1 2 -3 0
1 -2 3 0
1 -2 -3 0
-1 2 3 0
-1 2 -3 0
-1 -2 3 0
```

1.5.1 Simulador Local

Obter oráculo

[61]:
```
oracle = PhaseOracle.from_dimacs_file('formula.dimacs')
oracle.draw("mpl")
```

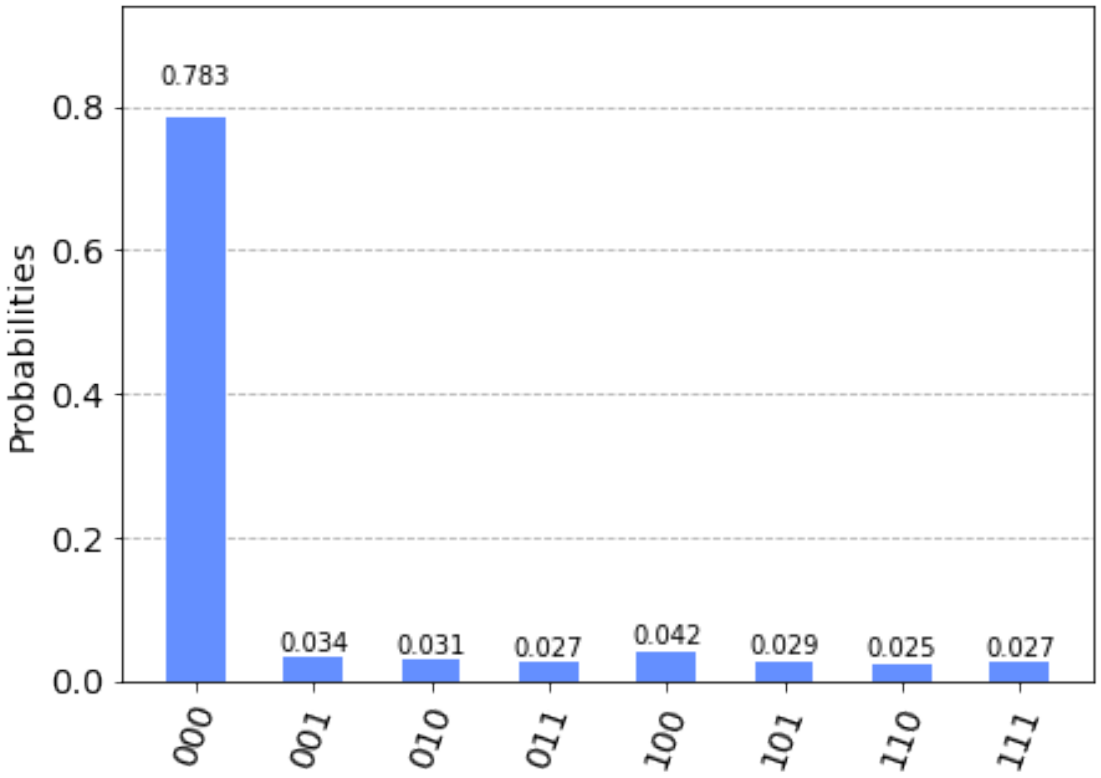
[61]:



Compor e excutar circuito

[62]:
```
backend = Aer.get_backend('aer_simulator')
quantum_instance = QuantumInstance(backend, shots=1024)
problem = AmplificationProblem(oracle=oracle)
grover = Grover(quantum_instance=quantum_instance)
result = grover.amplify(problem)
plot_histogram(result.circuit_results)
```

[62]:



1.5.2 Máquina Real

Construir circuito

[63]:
```
qc = grover.construct_circuit(problem, max(result.iterations))
qc.measure_all()
```

Selecionar máquina

[68]:
```
backend = provider.get_backend('ibm_oslo')
backend.configuration().n_qubits
```

[68]: 7

Executar circuito

[69]:
```
job = execute(qc,backend)
```

[70]:
```
job_monitor(job)
```

```
Job Status: job has successfully run
```

[71]:
```
result = job.result()
counts = result.get_counts()
plot_histogram(counts)
```

[71]: