2024

Universidade do Minho
Instituto de Educação

Interação e Concorrência

EXERCICIOS PROPOSTOS

A98980 EDUARDO CUNHA

Exercício 1

Considere a seguinte função booleana $f : \{0,1\}^3 \mapsto \{0,1\}$:

Input	Output
000	1
001	1
010	1
011	1
100	0
101	0
110	0
111	0

Identifique o tipo de função booleana e implemente o algoritmo de Deutsch para identificar corretamente a função. Comente sobre os resultados esperados e obtidos.

Figura 1 - Exercício 1

Para resolver este problema, vamos definir a função booleana em "Qiskit" e, em seguida, implementar o algoritmo de "Deutsch-Jozsa" para identificar se se trata de uma função balanceada ou constante com apenas uma única chamada à função.

Analisando a tabela, foi possível perceber que a função retorna 1 para todas as entradas cujo primeiro bit é 0 e retorna 0 para todas as entradas cujo primeiro bit é 1, sendo, portanto, balanceada.

Após a execução do circuito de "Deutsch-Jozsa", medimos a função e, se obtivermos:

- O estado $|0\rangle$ - então é a função é constante
- O estado $|1\rangle$ - então é a função é balanceada

Sendo assim, tendo em conta tudo o que foi referido, é de esperar obter 1 após a execução do algoritmo, dado que se trata de uma função balanceada.

Resolução do exercício:

Definimos a função booleana então da seguinte forma:

```
def fun_exc1(circuit):
    circuit.cx(0, 3)
```

Figura 2 - fun_exc1

A função "fun_exc1" recebe um circuito quântico ("circuit") como argumento e aplica uma porta "CNOT" ("CX") ao circuito, utilizando o "qubit" 0 como controle e o qubit 3 (“ancilla”) como alvo.

Posto isto e tendo em conta o código estudado nas aulas práticas, foi aplicado o algoritmo de “Deutsch Josza” da seguinte forma:

```
def deutsch_josza(n_qubits=3):

    qr_input = QuantumRegister(n_qubits, 'input')
    qr_ancilla =  QuantumRegister(1, 'ancilla')

    # Para medir
    cr_output = ClassicalRegister(n_qubits, 'output')

    # Quantum circuit
    qc = QuantumCircuit(qr_input, qr_ancilla, cr_output)

    # H-gate para todos os qubits -ancilla
    qc.h(qr_input)

    #ancilla
    qc.x(qr_ancilla)
    qc.h(qr_ancilla)

    qc.barrier()

    # Aplicar a funcao - ORACLE
    fun_exc1(qc)

    qc.barrier()

    # H-gate para todos os qubits -ancilla, novamente
    qc.h(qr_input)

    qc.barrier()

    # Medicao
    qc.measure(qr_input, cr_output)

    return qc
```

Figura 3 - deutsch_josza

Com este código geramos o seguinte circuito:

```
qc = deutsch_josza(3)
qc.draw(output="mpl")
```

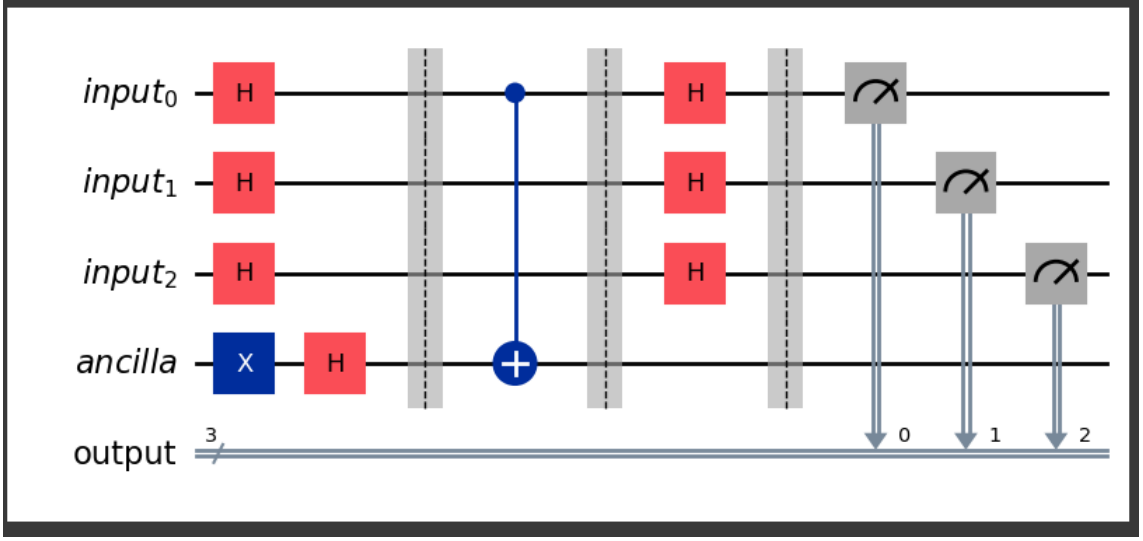
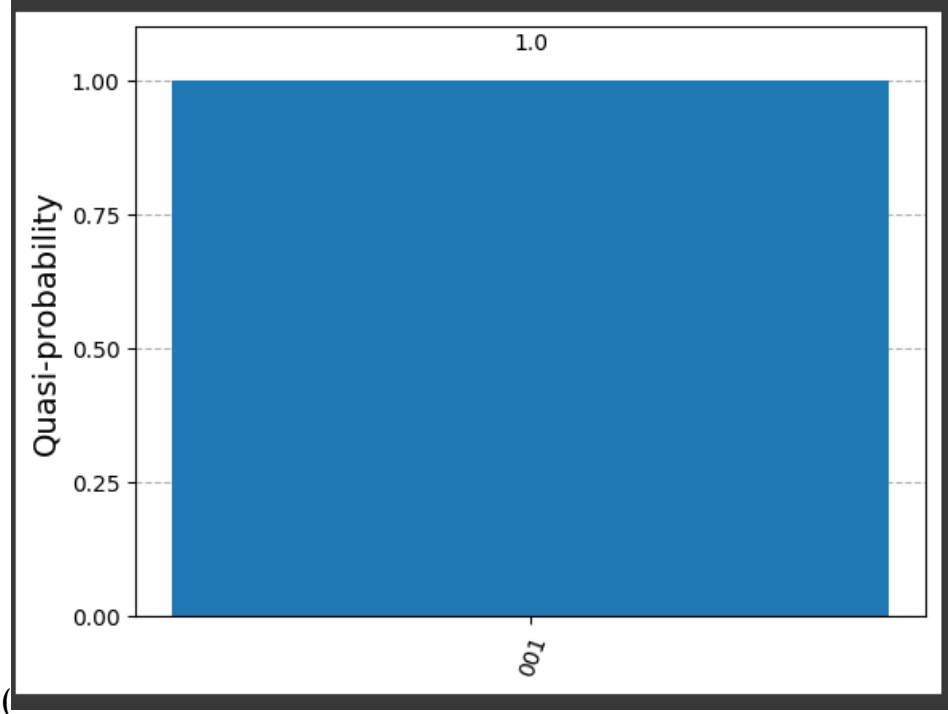


Figura 4 - Circuito gerado

E obtivemos as seguintes medidas:

(A função “execute_circuit” foi fornecida nas aulas práticas)

```
probs = execute_circuit(qc, shots=10000, binary=True, mode="sampler")
plot_histogram(probs)
```



(

Figura 5 - Medidas obtidas

Como é possível observar, obtivemos o resultado esperado (1), confirmando que se trata de uma função balanceada.

Exercício 2

a)

Considere uma base de dados com um total de N=16 elementos.

a) Implemente o algoritmo de Grover para encontrar o elemento indexado pelo estado $|0010\rangle$. Indique o número ótimo de iterações de Grover e a probabilidade expectável de medir o elemento. Comente sobre os resultados experimentais obtidos.

Figura 6 - Exercício 2 a)

A implementação do algoritmo de "Grover" foi realizada da seguinte forma:

Primeiramente, a implementação da função "Oracle" tem como objetivo marcar o estado desejado. Se esse estado corresponder à solução, o oráculo altera o seu sinal.

```
def oracle(qr, ancilla, solution=None):

    qc = QuantumCircuit(qr,ancilla)
    cx_gate = MCXGate(len(qr),ctrl_state=solution)
    qc = qc.compose(cx_gate)

    qc.barrier()
    return qc
```

Figura 7 - Função "oracle"

Seguidamente foi implementada a função "diffusion_operator" que tem como objetivo amplificar a amplitude do estado marcado pelo oráculo.

```
def diffusion_operator(qr, ancilla, n_qubits):

    qc = QuantumCircuit(qr,ancilla)

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

    cz = ZGate().control(n_qubits-1,ctrl_state="0"*(n_qubits-1))
    qc = qc.compose(cz)

    qc.x(qr[-1])

    qc.h(qr)

    qc.barrier()

    return qc
```

Figura 8 - Função "diffusion_operator"

Posto isto implementamos o circuito inteiro, de realçar que o número ótimo de iterações é dado por:

k = floor(pi/4 * sqrt(N))

Neste caso, N=16 (porque temos 4 qubits, resultando em 2^4 resultados possíveis).Então:

k = floor(pi/4 * sqrt(16)) = floor(pi) = 3

```
# Número de qubits
n = 4

# Nr de combinações possiveis
elements = 2**n

# Iterações do ciclo
iterations= int(np.floor(np.pi/4 * np.sqrt(elements)))


qr=QuantumRegister(n)

ancilla=QuantumRegister(1)

cr=ClassicalRegister(n)

qc = QuantumCircuit(qr,ancilla,cr)

qc.h(qr)

qc.x(ancilla)
qc.h(ancilla)

for j in range(iterations):
    qc = qc.compose(oracle(qr,ancilla,solution="0010"))
    qc = qc.compose(diffusion_operator(qr,ancilla,n))


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

Figura 9 - Circuito completo

Através do código anterior foi gerado o seguinte circuito.

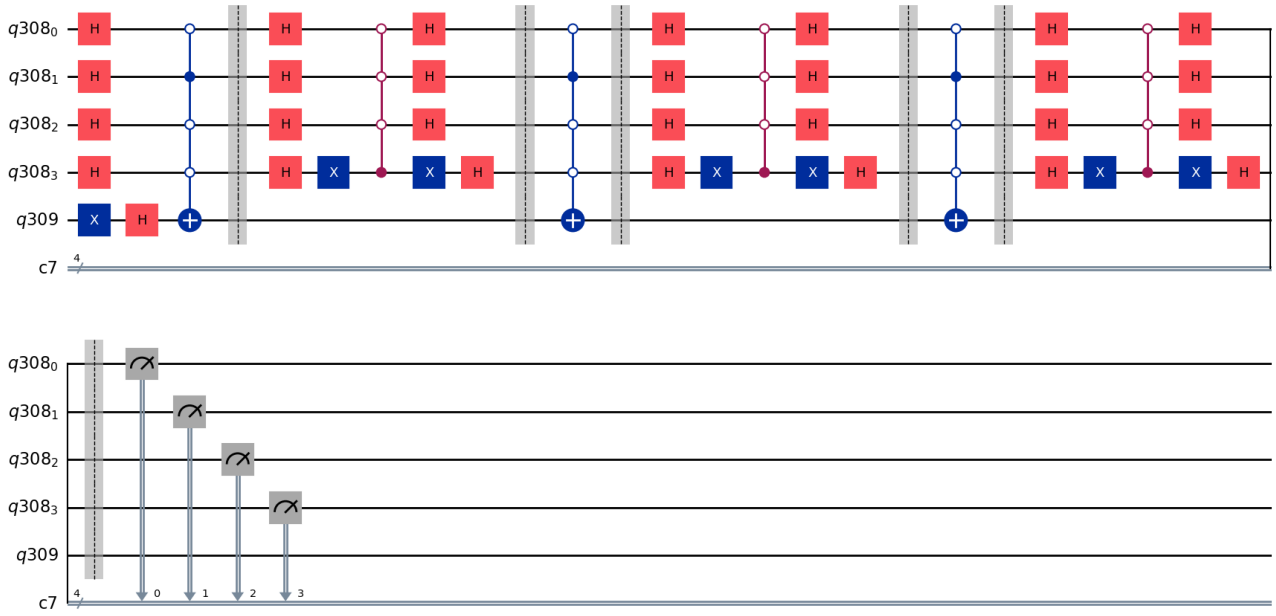


Figura 10 - Circuito gerado

E obtivemos os seguintes resultados:

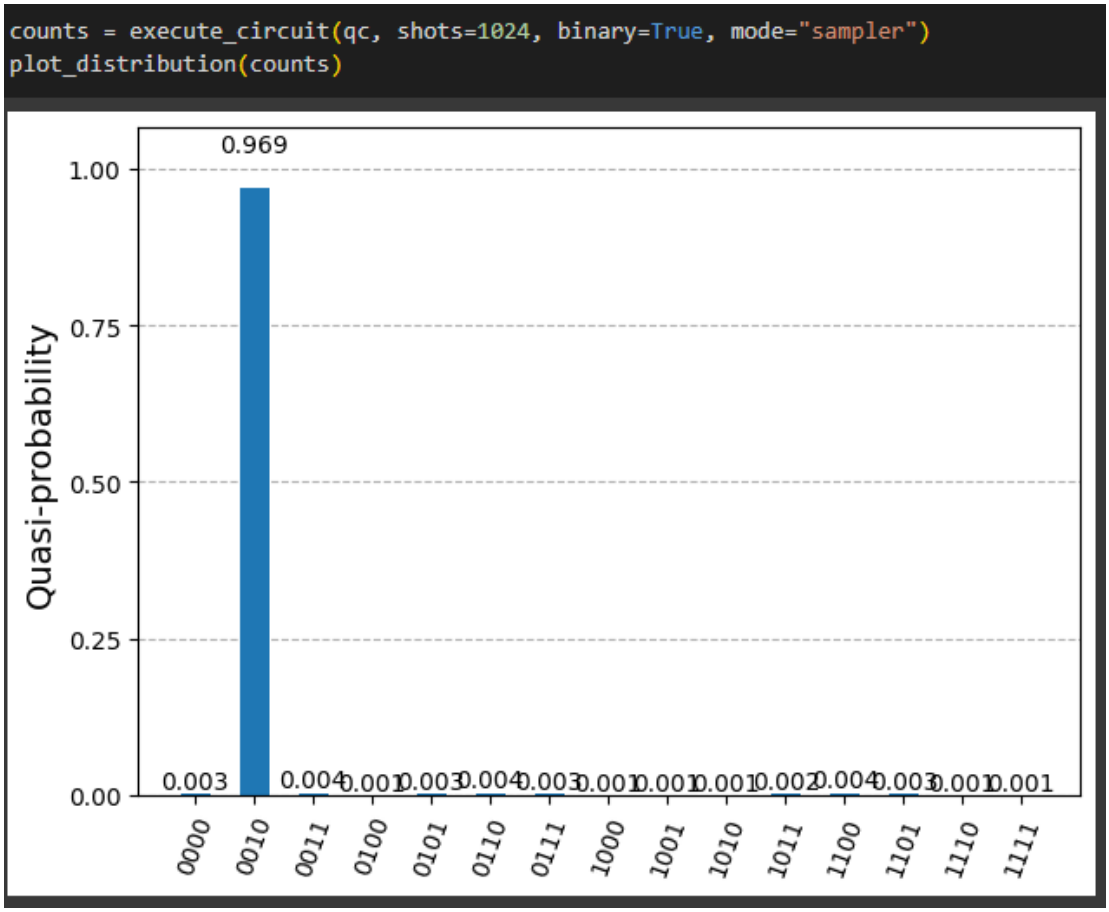


Figura 11 - Resultados Obtidos

Tendo em conta os slides teóricos relativos ao algoritmo de “Grover”, obtemos uma medição na base computacional que fornece a solução correta com uma probabilidade de (N-1)/N, ou seja, 15/16, o que é aproximadamente igual a 0.9375. Obtive o resultado de 0.969, o qual considero ser um valor bastante próximo do esperado.

b)

b) O que mudaria caso quisesse encontrar um dos seguintes elementos $\{0000,0101,1011,1110\}$? Comente sobre os resultados experimentais obtidos.

Figura 12 - Exercício 2 b)

Para encontrar um dos seguintes elementos fiz as seguintes alterações no código:

```
def grover(solutionInput):
  # Número de qubits
  n = 4

  # Nr de combinações possiveis
  elements = 2**n

  # Iterações do ciclo
  iterations= int(np.floor(np.pi/4 * np.sqrt(elements)))


  qr=QuantumRegister(n)

  ancilla=QuantumRegister(1)

  cr=ClassicalRegister(n)

  qc = QuantumCircuit(qr,ancilla,cr)

  qc.h(qr)

  qc.x(ancilla)
  qc.h(ancilla)

  for j in range(iterations):
      qc = qc.compose(oracle(qr, ancilla, solution=solutionInput))
      qc = qc.compose(diffusion_operator(qr,ancilla,n))

  return qc,qr,cr

qc,qr,cr = grover("0000")
qc.measure(qr,cr)
qc.draw(output="mpl")
```

Figura 13 - Função "Grover"

Como se pode verificar, agora, desenvolvi uma função que gera o circuito completo. Esta função recebe uma “string” indicando qual das soluções se pretende procurar e passa esta “string” como parâmetro da função "oracle" previamente definida.

```
counts = execute_circuit(qc, shots=1024, binary=True, mode="sampler")
plot_distribution(counts)
```

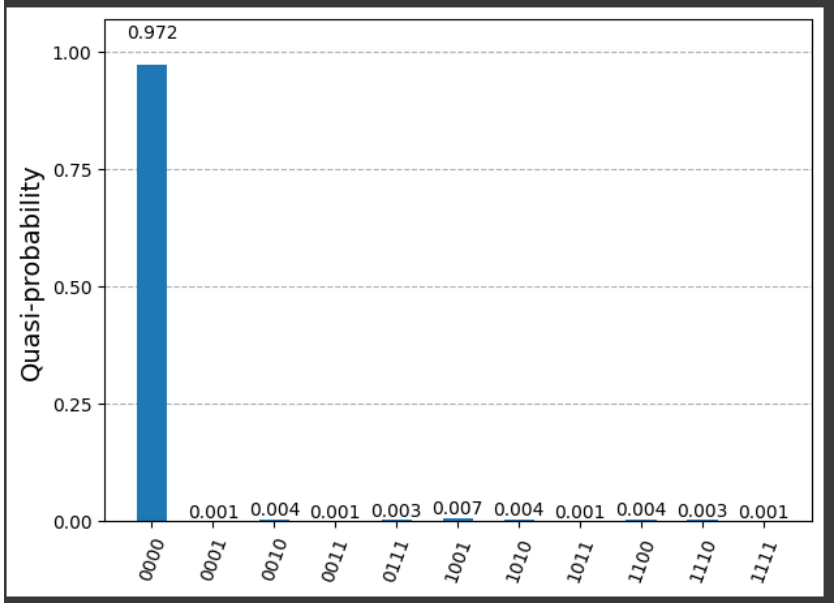


Figura 14 - Medição do circuito com entrada "0000"

```
qc,qr,cr = grover("0101")
qc.measure(qr,cr)
counts = execute_circuit(qc, shots=1024, binary=True, mode="sampler")
plot_distribution(counts)
```

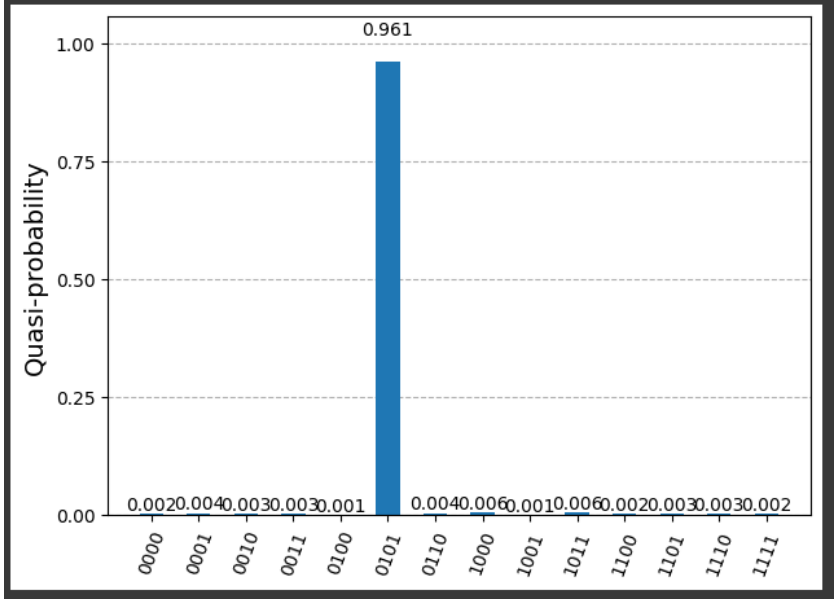


Figura 15 - Medição do o circuito com entrada "0101”

```
qc,qr,cr = grover("1101")
qc.measure(qr,cr)
counts = execute_circuit(qc, shots=1024, binary=True, mode="sampler")
plot_distribution(counts)
```

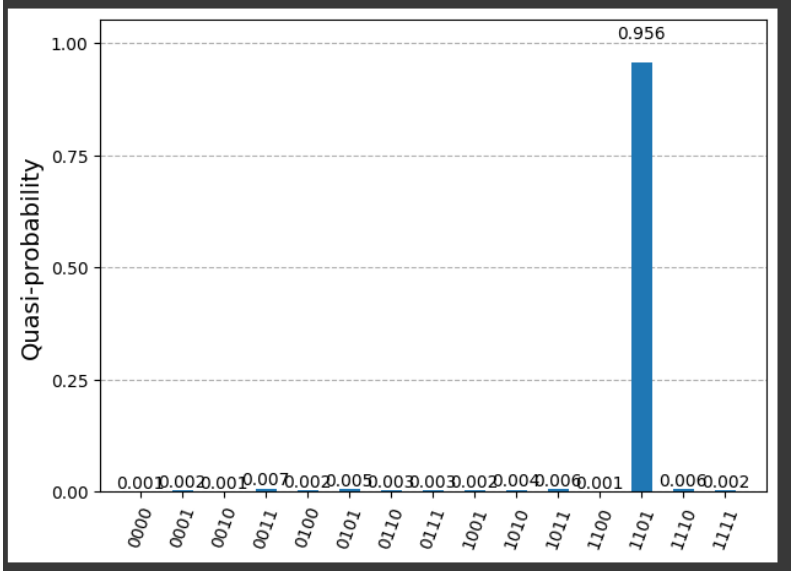


Figura 16 - Medição do o circuito com entrada "1101"

```
qc,qr,cr = grover("1110")
qc.measure(qr,cr)
counts = execute_circuit(qc, shots=1024, binary=True, mode="sampler")
plot_distribution(counts)
```

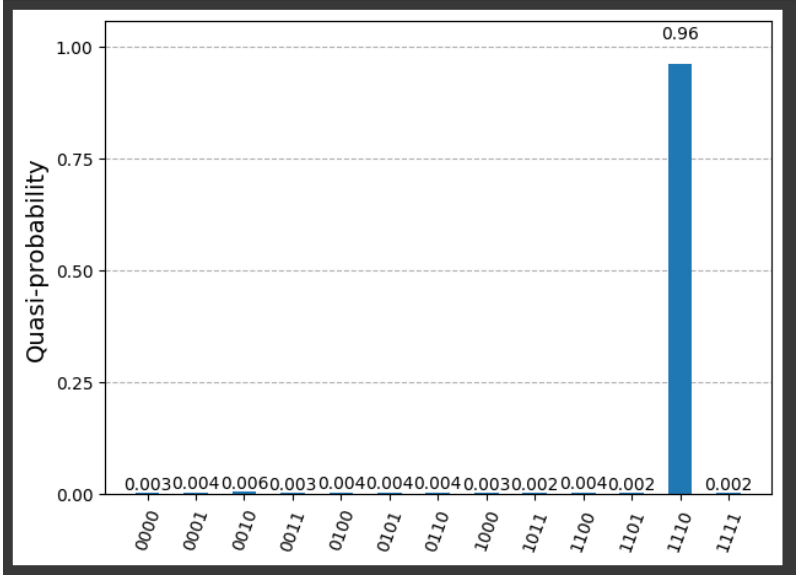
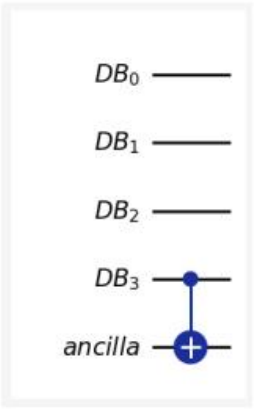


Figura 17 - Medição do o circuito com entrada "1110"

Estes foram os resultados obtidos e como se pode verificar, ao ser executado três vezes (como calculado na pergunta anterior), todas as medições se aproximaram bastante do valor esperado previamente calculado (0.9375), o que está de acordo com a teoria ensinada nas aulas. Isso confirma a eficácia e eficiência do algoritmo de “Grover”.

c)

c) Considere o oráculo representado pelo seguinte circuito:



Que elemento identifica o oráculo ? Comente sobre os resultados expectáveis do algoritmo de Grover e os resultados experimentais obtidos.

Figura 18 - Exercício 2 c)

Primeiramente implementamos o oráculo em questão:

```
def oracle(qr, ancilla):

    qc = QuantumCircuit(qr, ancilla)
    qc.cx(qr[-1], ancilla)
    qc.barrier()

    return qc
```

Figura 19 - Função "oracle"

Este oráculo aplica uma operação CNOT onde o último “qubit” é o “qubit” de controle e o “qubit ancilla” é o alvo. Isso significa que o oráculo inverte o estado do qubit “ancilla” se e somente se o último “qubit” for igual a 1, desta forma tenta marcar todos os estados terminados por 1 ou seja: 0001, 0011,0111,1111,0101,1001,1011,1101.

Desta forma, com a aplicação do algoritmo de “Grover”, seria de esperar obter percentagens mais altas do que as dos demais e bastante idênticas para esses estados.

Codificação do restante circuito:

```python
def diffusion_operator(qr, ancilla, n_qubits):

    qc = QuantumCircuit(qr,ancilla)

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

    cz = ZGate().control(n_qubits-1,ctrl_state="0"*(n_qubits-1))
    qc = qc.compose(cz)

    qc.x(qr[-1])

    qc.h(qr)

    qc.barrier()

    return qc
```

Figura 20 - Função "diffusion_operator"

```python
# Número de qubits
n = 4

# Nr de combinações possiveis
elements = 2**n

casos_possiveis = 2**(n-1)

# Iterações do ciclo
iterations= int(np.floor((np.pi/4) * (casos_possiveis / np.sqrt(elements))))
print(f"Iterações: {iterations}")


qr=QuantumRegister(n)

ancilla=QuantumRegister(1)

cr=ClassicalRegister(n)

qc = QuantumCircuit(qr,ancilla,cr)

qc.h(qr)

qc.x(ancilla)
qc.h(ancilla)

for j in range(iterations):
    qc = qc.compose(oracle(qr,ancilla))
    qc = qc.compose(diffusion_operator(qr,ancilla,n))


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

Figura 21 - Criação do circuito completo

Então, ao criar o circuito, com apenas uma alteração no número ideal de iterações, dado que agora há 8 estados possíveis de resposta, obtemos o seguinte circuito:

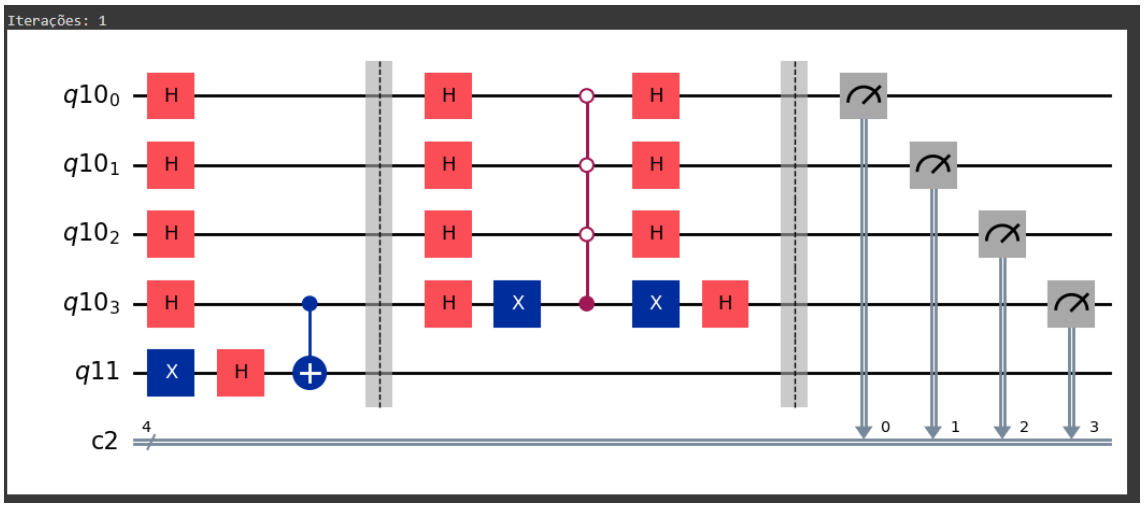


Figura 22 - Esquema do circuito gerado

Medimos os seguintes resultados:

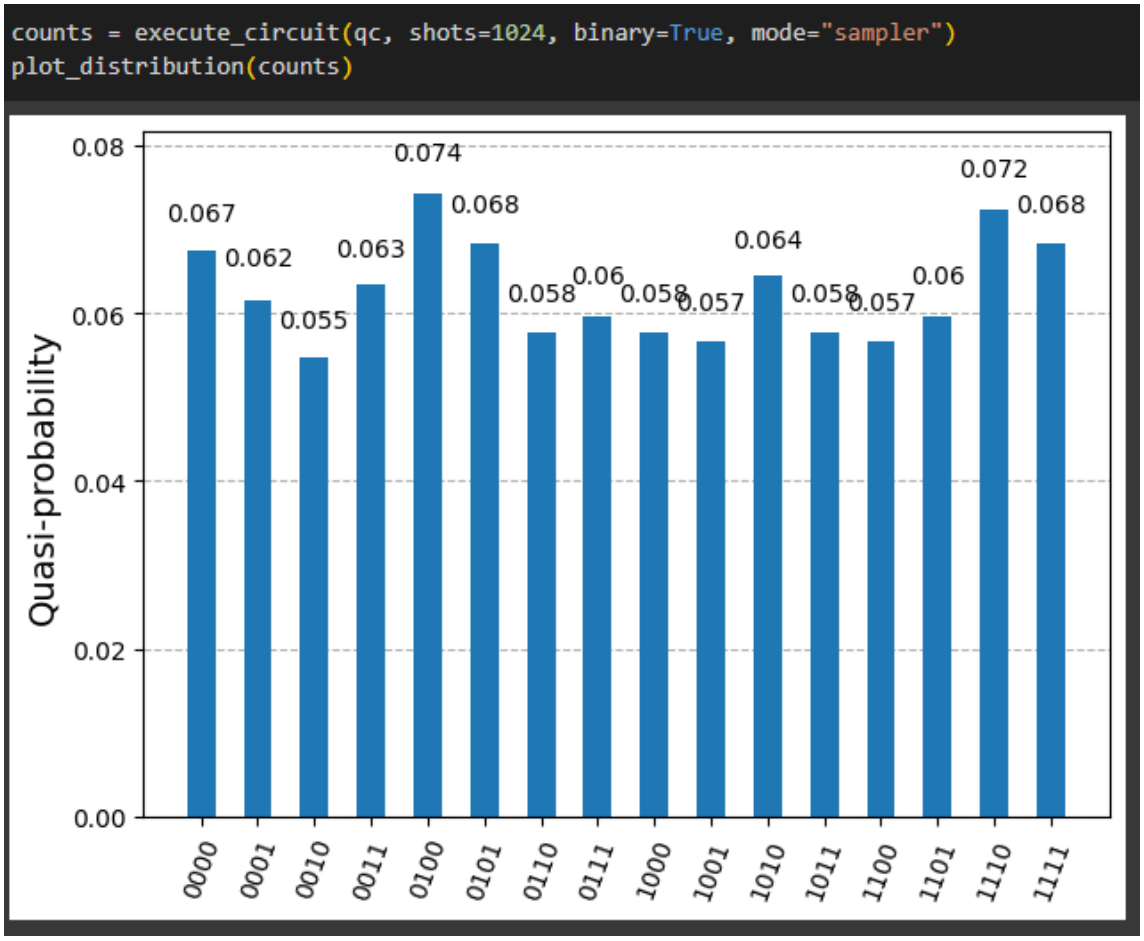


Figura 23 - Medição dos resultados

Isto vai contra o esperado, visto que todos os estados possuem uma percentagem alta. O problema parece estar no facto de que o oráculo atual apenas verifica o último “qubit”, sem considerar o estado dos outros “qubits”. Para que o algoritmo de “Grover” funcione corretamente, o oráculo precisa marcar especificamente o estado alvo (ou estados alvos) que desejamos amplificar e, desta forma, não está a marcar nenhum estado específico. Para marcar mais do que um estado, seria necessário marcar exatamente um estado de cada vez, como no exemplo a seguir, dada a função oráculo definida na alínea anterior.

```
for j in range(iterations):
    qc = qc.compose(oracle(qr,ancilla,"1111"))
    qc = qc.compose(oracle(qr,ancilla,"0001"))
    qc = qc.compose(diffusion_operator(qr,ancilla,n))
```

Figura 24 - Marcação de mais que 1 estado

Desta forma, caso fossem marcados todos os estados esperados seria possível obter os resultados inicialmente esperados.