Quantum Computing - Final project

Solving satisfiability problems using Grover's Algorithm

- André Oliveira Barbosa
- Miguel Angelo Mesquita Rego

Tasks

- Design a solvable 3-SAT boolean formula;
- Implement Grover's algorithm for solving the satisfiability problem;
- Assess the quality of the solution employed by the quantum algorithm;
- Study the complexity associated with the algorithm applied to your problem, I.e., the optimal number of Grover iterations needed to reach a solution.

Imports & Installs

```
pip install qiskit
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: qiskit in /usr/local/lib/python3.9/dist-packages (0.42.1)
Requirement already satisfied: qiskit-ibmq-provider==0.20.2 in /usr/local/lib/python3.9/dist-packages (from qiskit) (0.2
Requirement already satisfied: qiskit-terra==0.23.3 in /usr/local/lib/python3.9/dist-packages (from qiskit) (0.23.3)
Requirement already satisfied: qiskit-aer==0.12.0 in /usr/local/lib/python3.9/dist-packages (from qiskit) (0.12.0)
Requirement already satisfied: numpy>=1.16.3 in /usr/local/lib/python3.9/dist-packages (from qiskit-aer==0.12.0->qiskit)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.9/dist-packages (from qiskit-aer==0.12.0->qiskit) (1
Requirement already satisfied: python-dateutil>=2.8.0 in /usr/local/lib/python3.9/dist-packages (from qiskit-ibmq-provic
Requirement already satisfied: websocket-client>=1.5.1 in /usr/local/lib/python3.9/dist-packages (from qiskit-ibmq-provi
Requirement already satisfied: urllib3>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from qiskit-ibmq-provider==0.2
Requirement already satisfied: websockets>=10.0 in /usr/local/lib/python3.9/dist-packages (from qiskit-ibmq-provider==0.
Requirement already satisfied: requests-ntlm<=1.1.0 in /usr/local/lib/python3.9/dist-packages (from qiskit-ibmq-provider
Requirement already satisfied: requests>=2.19 in /usr/local/lib/python3.9/dist-packages (from qiskit-ibmq-provider==0.20
Requirement already satisfied: ply>=3.10 in /usr/local/lib/python3.9/dist-packages (from qiskit-terra==0.23.3->qiskit) (
Requirement already satisfied: symengine>=0.9 in /usr/local/lib/python3.9/dist-packages (from qiskit-terra==0.23.3->qisk
Requirement already satisfied: psutil>=5 in /usr/local/lib/python3.9/dist-packages (from qiskit-terra==0.23.3->qiskit) (
Requirement already satisfied: stevedore>=3.0.0 in /usr/local/lib/python3.9/dist-packages (from qiskit-terra==0.23.3->qi
Requirement already satisfied: sympy>=1.3 in /usr/local/lib/python3.9/dist-packages (from qiskit-terra==0.23.3->qiskit)
Requirement already satisfied: dill>=0.3 in /usr/local/lib/python3.9/dist-packages (from qiskit-terra==0.23.3->qiskit) (
Requirement already satisfied: rustworkx>=0.12.0 in /usr/local/lib/python3.9/dist-packages (from qiskit-terra==0.23.3->c
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.9/dist-packages (from python-dateutil>=2.8.0->qiskit-i
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests>=2.19->qiskit
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests>=2.19-
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests>=2.19->qiskit-ibmq-
Requirement already satisfied: cryptography>=1.3 in /usr/local/lib/python3.9/dist-packages (from requests-ntlm<=1.1.0->c
Requirement already satisfied: ntlm-auth>=1.0.2 in /usr/local/lib/python3.9/dist-packages (from requests-ntlm<=1.1.0->qi
Requirement already satisfied: pbr!=2.1.0,>=2.0.0 in /usr/local/lib/python3.9/dist-packages (from stevedore>=3.0.0->qisk
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.9/dist-packages (from sympy>=1.3->qiskit-terra==0.
Requirement already satisfied: cffi>=1.12 in /usr/local/lib/python3.9/dist-packages (from cryptography>=1.3->requests-nt
Requirement already satisfied: pycparser in /usr/local/lib/python3.9/dist-packages (from cffi>=1.12->cryptography>=1.3->
```

```
pip install pylatexenc
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: pylatexenc in /usr/local/lib/python3.9/dist-packages (2.10)
```

```
import qiskit.tools.jupyter
```

```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, Aer, execute
from qiskit.tools.visualization import plot_histogram, plot_distribution
import matplotlib.pyplot as plt
import numpy as np
```

Design a solvable 3-SAT boolean formula

Como pedido na secção de tarefas dada no enunciado, começamos por construir uma fórmula 3-SAT. Assim, tivemos em conta que esta formula tem as especificidades:

- Tem de ser satisfeita por meio de uma atribuição de valores de verdade para as suas variáveis de modo a que a tornem verdadeira;

- Tem que conter 3 variáveis em cada claúsula.

Assim, consideramos a seguinte fórmula f:

$$f(v1, v2, v3) = (v1 \lor v2 \lor v3) \land (v1 \lor \neg v2 \lor v3) \land (\neg v1 \lor v2 \lor v3) \land (v1 \lor v2 \lor \neg v3) \land (\neg v1 \lor v2 \lor \neg v3) \\ \land (\neg v1 \lor \neg v2 \lor \neg v3) \land (\neg v1 \lor \neg v2 \lor v3)$$

De modo a verificar a satisfabilidade desta fórmula, elaboramos a seguinte tabela:

v1	v2	v3	((v1 ∨ (v2 ∨ v3)) ∧ ((v1 ∨ (¬v2 ∨ v3)) ∧ ((¬v1 ∨ (v2 ∨ v3)) ∧ ((v1 ∨ (v2 ∨ ¬v3)) ∧ ((¬v1 ∨ (v2 ∨ ¬v3)) ∧ ((¬v1 ∨ (¬v2 ∨ ¬v3)) ∧ (¬v1 ∨ (¬v2 ∨ v3)))))))
F	F	F	**F**
F	F	T	**F**
F	T	F	**F**
F	T	T	**T**
T	F	F	**F**
T	F	T	**F**
T	T	F	**F**
T	T	T	**F**

Podemos assim concluir, com base nos pontos descritos anteriormente, que estamos perante uma fórmula 3-SAT.

Algoritmo de Grover

```
def execute_circuit(qc, shots=1024, decimal=False):
    device = Aer.get_backend('qasm_simulator')
    counts = device.run(qc, shots=shots).result().get_counts()

    if decimal:
        counts = dict((int(a[::-1],2),b) for (a,b) in counts.items())
    else:
        counts = dict((a[::-1],b) for (a,b) in counts.items())

    return counts
```

Inicialização é responsável por criar uma superposição uniforme de todas as possíveis soluções do problema, permitindo que o algoritmo comece a busca de forma equilibrada.

```
def create_circuit():
  #3 qubits que são as variaveis que vamos usar na nossa formula
  qr = QuantumRegister(3)

  #vamos usar 7 data qubits + 1 ancilla qubit
  ancilla = QuantumRegister(8)

  #Vai ser usado para guardar os valores finais quando o estado colapsar
  cr = ClassicalRegister(3)

  #construção do circuito quantico
  qc = QuantumCircuit(qr, ancilla, cr)

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


  return qc, qr, ancilla, cr
```

Diffusion tem como objetivo aumentar a amplitude da solução e diminuir a amplitude das outras entradas, aumentando asssim a probabilidade de encontrar a resposta correta na próxima iteração.

```
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
```

Oracle é responsável por identificar se uma determinada entrada é uma solução para o problema em questão ou não, marcando a amplitude correspondente à solução.

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

  for k in list(range(0,7)):
    #vai realizar as gates para obtermos as subformulas em cada iteração
    if k == 0:
      qc.mcx(qr,ancilla[k])
      qc.x(ancilla[k])

    if k == 1:
      qc.x(qr[1])
      qc.mcx(qr,ancilla[k])
      qc.x(ancilla[k])
      qc.x(qr[1])

    if k == 2:
      qc.x(qr[1])
      qc.x(qr[2])
      qc.mcx(qr,ancilla[k])
      qc.x(ancilla[k])
      qc.x(qr[1])
      qc.x(qr[2])

    if k == 3:
      qc.x(qr[2])
      qc.mcx(qr,ancilla[k])
      qc.x(ancilla[k])
      qc.x(qr[2])

    if k == 4:
      qc.x(qr[0])
      qc.x(qr[1])
      qc.mcx(qr,ancilla[k])
      qc.x(ancilla[k])
      qc.x(qr[0])
      qc.x(qr[1])

    if k == 5:
      qc.x(qr[0])
      qc.x(qr[1])
      qc.x(qr[2])
      qc.mcx(qr,ancilla[k])
      qc.x(ancilla[k])
      qc.x(qr[0])
      qc.x(qr[1])
      qc.x(qr[2])

    if k == 6:
      qc.x(qr[0])
      qc.x(qr[2])
      qc.mcx(qr,ancilla[k])
      qc.x(ancilla[k])
      qc.x(qr[0])
      qc.x(qr[2])

    #calcula o valor final da formula
  qc.mcx(ancilla[:-1], ancilla[-1])
  qc.x(ancilla[-1])

  qc.barrier()
  return qc
```

Double-click (or enter) to edit

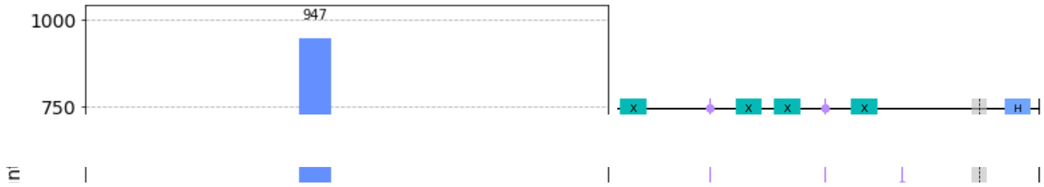
```
#Grover algorithm
def grover(qc, qr, ancilla, oracle):
  num_iterations = int(np.floor(np.pi /4 * np.sqrt(2**3)))

  for i in range(num_iterations):
    qc = qc.compose(oracle(qr,ancilla))
    qc = qc.compose(diffusion_operator(qr,ancilla))

  return qc


qc, qr, ancilla, cr = create_circuit()
qc = grover(qc,qr,ancilla,oracle)
qc.measure(qr,cr)
qc.draw(output = "mpl")
```

```
plot_histogram(execute_circuit(qc,1024))
```



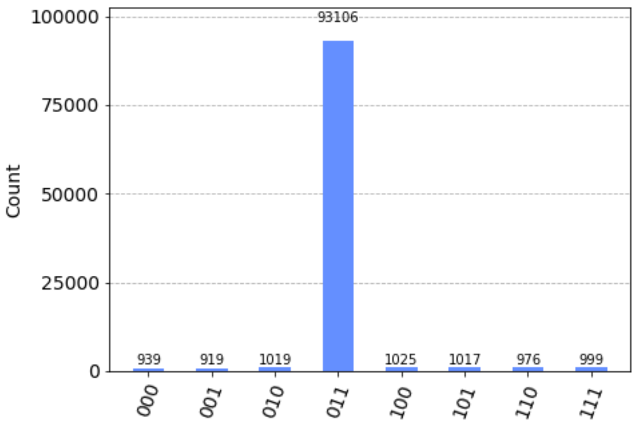
Solução obtida com 1 iteração

```
#Com 1 iteração
def groverone(qc, qr, ancilla, oracle):
  num_iterations = int(np.floor(np.pi /4 * np.sqrt(2**3)))

  qc = qc.compose(oracle(qr,ancilla))
  qc = qc.compose(diffusion_operator(qr,ancilla))
  return qc

qc, qr, ancilla, cr = create_circuit()
qc = grover(qc,qr,ancilla,oracle)
qc.measure(qr,cr)
qc.draw(output = "mpl")

plot_histogram(execute_circuit(qc, 100000))
```



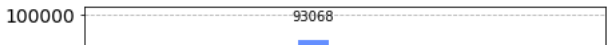
Solução obtida com 3 iterações

```
#Com 3 iterações
def groverthree(qc, qr, ancilla, oracle):
  num_iterations = int(np.floor(np.pi /4 * np.sqrt(2**3)))

  for j in range(3):
    qc = qc.compose(oracle(qr,ancilla))
    qc = qc.compose(diffusion_operator(qr,ancilla))
  return qc

qc, qr, ancilla, cr = create_circuit()
qc = grover(qc,qr,ancilla,oracle)
qc.measure(qr,cr)
qc.draw(output = "mpl")

plot_histogram(execute_circuit(qc, 100000))
```



Conclusão

Pelo resultado acima podemos concluir que o algoritmo tem uma probabilidade acima de obter a solução correta é de acima de 95% o que demonstra a excelente precisão do algoritmo.

O grau de complexidade é, tal como mostrado nas aulas, de $O(\sqrt{N})$, e o número de iterações ideal é 2, dado que é nesse caso que a sua precisão é maior (face à precisão de 93% no caso de 1 e 3 iterações.).

Consideramos que este trabalho prático foi essencial para consolidar os nossos conhecimentos nesta área e que foi um desafio bastante interessante. Foi também enriquecedor entender as vantagens o funcionamento deste algoritmo quantico. Por fim, entendemos que fomos ao encontro de todas as tarefas propostas no enunciado do projeto e que as cumprimos com sucesso.

Referências:

- Satisfiability with Grover - https://qiskit.org/textbook/ch-applications/satisfiability-grover.html
- Grover's algorithm - Qiskit - https://qiskit.org/textbook/ch-algorithms/grover.html
- BlackBoard

Double-click (or enter) to edit