TP3G10

June 5, 2021

1 Quantum Practical Assignment

1.1 TP2 de Interação e Concorrência

1.1.1 Grupo 10

- Luís Miguel Mendonça Almeida - A84180
- João Pedro Martins Montrezol Camilo Antunes - A86813

1.2 Introdução

- breve introdução

[4]:
```
# importing Qiskit
from qiskit import Aer, IBMQ
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute, transpile

from qiskit.tools.visualization import plot_histogram, plot_state_city,␣
 ↪plot_state_hinton

import matplotlib.pyplot as plt
from math import floor
from IPython.display import HTML
%matplotlib inline
```

1.3 Grover Algorithm

Consideremos o problema de encontrar um item numa lista não estruturada (i.e. sem ordem). Uma abordagem num contexto de computação clássica seria percorrer toda esta lista até encontrar o item, o que significa que é possível resolver este problema em O(N) passos.

No entanto, utilizando um computador quântico, é possível resolver este problema com o algoritmo de Grover em $O(\sqrt{N})$ passos, o que é uma melhoria substancial em relação ao paradigma clássico de computação.

O primeiro passo para entendermos o algoritmo de Grover é entender como representar uma lista não estruturada no paradigma quântico. Podemos olhar para o espaço de estados representáveis com N qubits como sendo os elementos da nossa lista, obtendo assim $2^N - 1$ elementos. Por exemplo, suponhamos que temos 3 qubits. Assim, os elementos da lista serão:

$|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |111\rangle$. Ora, se ao medirmos o estado de cada um dos qubits tivermos a mesma probabilidade de obter qualquer um dos elementos da lista, então esta estará desordenada.

O algoritmo de Grover é constituído por 3 etapas:

- Inicialização do circuito para que todos os estados possíveis de *input* tenham a mesma amplitude
- Oráculo que altera a fase do estado (i.e. elemento) que queremos encontrar para um número negativo
- Amplificação da probabilidade de, ao medirmos o estado, este colapsar naquele que tem fase negativa, alterando-a para positiva

1.3.1 Inicialização

Para darmos a mesma amplitude para todos os estados possíveis de *input*, começamos por passar todos os qubits por uma gate de Hadamard

1.3.2 Oráculo

Nesta etapa queremos construir um oráculo que identifique corretamente o estado (i.e. elemento) que estamos a procurar e altere a sua fase de forma a que esta fique negativa, mantendo positivas as fases de todos os outros estados. No nosso caso queremos encontrar o valor $|010\rangle$, pelo que uma implementação do oráculo pedido seria:

1.3.3 Amplificação

Queremos agora amplificar a probabilidade de, ao medirmos o estado dos qubits, este colapsar no estado que tem fase negativa. Para este efeito, aplicamos uma gate de Hadamard, seguida de uma gate X, seguidos da gate $C^N Z$ (decomposta) e novamente uma gate X e H, para todos os qubits. Não nos devemos esquecer que esta etapa inverte novamente a fase do estado que queremos encontrar, pelo que no fim deste processo ficará positiva.

Assim, seguindo este raciocínio, podemos repetir as etapas de Oráculo e Amplificação para aumentarmos cada vez mais a probabilidade do estado dos qubits colapsar no elemento desejado. No entanto, temos de ter cuidado em não realizar iterações a mais pois tal pode resultar num decréscimo desta probabilidade. O número ideal de vezes que se repetem estas etapas é usualmente $\sqrt{(N)}$, podendo este número ser ligeiramente alterado em cada caso específico.

1.4 Código

A fim de aplicar o algoritmo descrito no último tópico, será então desenvolvida uma função de ordem superior. A qual chamaremos `oracleHandler(nQBits)`.

Esta função receberá como parâmetro o número de QBits que será utilizado no circuito *target* da aplicação final. Como resultado, tal função retornará uma outra função, esta última pronta a receber um circuito *target* e um *pattern* a ser procurado.

Prosseguiremos então por explicar as etapas da função de ordem superior desenvolvida:

[ ]:
```
applications = floor(2**(nQBits/2))
auxN = max(nQBits - 3,0)
```

Inicialmente calcula-se o número recomendado de aplicações do algoritmo. Como referido, o valor "ideal" teórico é $\sqrt{(N)}$, sendo no caso requerido, $N = 2^{nQbits}$.

Além disso é calculado neste ponto também, o número de QBits auxiliares para se realizar a aplicação do algoritmo. Este valor é calculado consoante a maneira com que os *gates* c^nZ são desenvolvidos.

Nas nossas aplicações o valor de gates auxiliares é dado simplesmente pela subtração de 3 a `nQbits`, em outras versões o valor a ser subtraído poderia ser 2, pelo uso explícito do gate cZ como apresentado mais adiante.

Passaremos agora ao desenvolvimento da função de aplicação do oráculo:

[3]:
```
    def searchPat(circuit,QBits,pattern):
        for i,t in enumerate(pattern):
            if t == '0':
                circuit.x(QBits[len(pattern)- 1 - i])
        pass
```

Esta primeira função aninhada dentro da função de ordem superior desenvolvida consistirá no início da aplicação do oráculo. Por estarmos lidando com funções de ordem superior, escalonáveis, esta função teve de ser definida internamente à função `oracleHandler(nQBits)`(assim como as próximas a serem apresentadas) para garantir sua parâmetrização consoante o número de bits do circuito.

Esta função tratará de "selecionar" os valores que ativarão os *gates* c^nZ a seguir. Assim, tal função irá aplicar um *gate* X negando os QBits que estejam em posições delimitadas por 0s no pattern a ser buscado. Tendo em atenção que pattern ainda não será um argumento da aplicação da função, mas sim, de uma aplicação futura da função resultante desta operação.

A função ainda leva em conta a "inversão" na ordem dos bits realizada, iterando os valores corretamente através do cálculo de índice realizado.

A seguir apresenta-se a função aninhada de maior importância na composição do algoritmo requerido,

[ ]:
```
    def fullCnZ(circuit,QBits,nQBits,auxQBits):

        auxLen = max(nQBits - 3,0)

        if (nQBits > 2):
            finals = [QBits[0],QBits[1]]

        for x in range(auxLen):
            circuit.ccx(finals[-2],finals[-1],auxQBits[x])
            finals += [auxQBits[x]]
            finals += [QBits[x+2]]
```

```
        #Z - CZ - CCZ
        #---------------------------------#
        switch={
            1: (lambda: circuit.x(QBits[0])),
            2: (lambda: circuit.cx(QBits[0],QBits[1])),
            'default': (lambda: circuit.
↪ccx(finals[-2],finals[-1],QBits[nQBits-1])),
            }

        circuit.h(QBits[nQBits-1])
        switch.get(nQBits,switch['default'])()
        circuit.h(QBits[nQBits-1])
        #---------------------------------#

        finals.pop()

        while len(finals) > 1:
            tgt = finals.pop()
            q2 = finals.pop()
            q1 = finals[-1]
            circuit.ccx(q1,q2,tgt)

        pass
```

A função consistirá em formar o *gate* $c^{nQBits}Z$ necessário para a aplicação do algoritmo. Para tanto, a função inicia por julgar se bits auxiliares serão necessários. Realizando uma de quatro estratégias,

- O circuito só possui 1 QBit: 1 único gate Z simples é aplicado;
- O circuito só possui 2 QBits: 1 único gate cZ é aplicado;
- O circuito só possui 3 QBits: 1 único gate ccZ é aplicado;
- O circuito necessita de QBits auxiliares: após reduzir os bits "verificados" a 3, 1 único gate ccZ é aplicado;

Para coêrencia algoritmica e a fim de reduzir o número de QBits utilizados, as gates Z serão representadas através da identidade:

$$Z = H \cdot X \cdot H$$

sendo os *gates* cZ e ccZ representados com o uso do cX e ccX respetivamente. A aplicação descrita acima é visível em:

[ ]:
```
        #Z - CZ - CCZ
        #---------------------------------#
        switch={
            1: (lambda: circuit.x(QBits[0])),
```

```
            2: (lambda: circuit.cx(QBits[0],QBits[1])),
            'default': (lambda: circuit.
↪ccx(finals[-2],finals[-1],QBits[nQBits-1])),
            }

        circuit.h(QBits[nQBits-1])
        switch.get(nQBits,switch['default'])()
        circuit.h(QBits[nQBits-1])
        #--------------------------------#
```

Dentre as formações de *gates* apresentadas, entretanto, a última é a que apresentaria a maior dificuldade, devido a "redução" mencionada. Esta, estaria referindo-se a algo semelhante à estratégia apresentada durante as aulas, a implementação Nielsen & Chuang, mas com uma pequena alteração.

Na aplicação original, o objetivo seria "reduzir" os bits de controlo a 2 auxiliares, tornando-se trivial a aplicação de uma gate cU.

Para a função desenvolvida, entretanto, o método desenvolvido foi implementar uma "redução" dos bits de controlo a 3 auxiliares, suficientes para a aplicação trivial de um ccZ como referido.

A redução consistirá em uma *stack* de QBits, inicialmente com o QBit Q_0 e Q_1. A *stack* avança ligando os últimos dois QBits por um ccX a um QBit Auxiliar, fazendo, a seguir um *push* do QBit Auxiliar resultante e o próximo QBit do sistema (no exemplo o Q_2). O processo é então repetido até que só hajam 2 QBits "livres" no circuito e 1 QBit auxiliar, totalizando 3 QBits de controlo.

o excerto de código da operação é demonstrado a seguir:

[ ]:
```
        for x in range(auxLen):
            circuit.ccx(finals[-2],finals[-1],auxQBits[x])
            finals += [auxQBits[x]]
            finals += [QBits[x+2]]
```

Por fim, a *stack* é consumida, realizando sucessivos *pop*s para reverter as operações ccX realizadas.

[ ]:
```
        finals.pop()

        while len(finals) > 1:
            tgt = finals.pop()
            q2 = finals.pop()
            q1 = finals[-1]
            circuit.ccx(q1,q2,tgt)
```

A seguir é apresentada uma pequena ilustração do desenvolvimento apresentado, é utilizado um circuito de 5 QBits, somados a 2 auxiliares. QBits marcados com uma bola vermelha são resultantes de um *push* na stack, e a azul a de um *pop*. o "x" vermelho representa simlesmente a aplicação do gate Z.

Por fim, a última função anônima a ser descrita será a função que será retornada. `genOracle(circuit,QBits,pattern="1"*nQBits,app=applications,auxQBits=None,`

`phaseTests=False)`. Esta função fara um *wrap* das duas funções acima demonstradas, realizando enfim a aplicação parametrizada do oráculo e do difusor.

Os parâmetros a serem utilizados serão o padrão a ser procurado, e o número de aplicações associadas ao oráculo e ao difusor. Além claro, do circuito, QBits e QBits Auxiliares necessários para a aplicação.

Um parâmetro a ser ressaltado será `phaseTests`, uma flag para que a função retorne uma lista com as *phases* do sistema em cada aplicação e passo dado na constução do circuito

[ ]:
```
    def␣
↪genOracle(circuit,QBits,pattern="1"*nQBits,app=applications,auxQBits=None,␣
↪phaseTests=False):
        phaseRes = []
        for ap in range(app):

            #Oracle
            searchPat(circuit,QBits,pattern)
            fullCnZ(circuit,QBits,nQBits,auxQBits)
            searchPat(circuit,QBits,pattern)

            if phaseTests:
                backend_state = Aer.get_backend('statevector_simulator')
                result = execute(circuit, backend_state).result()
                state  = result.get_statevector(circuit)
                phaseRes += [('oracle',state)]


            #Diffuser
            circuit.h(QBits)
            circuit.x(QBits)
            fullCnZ(circuit,QBits,nQBits,auxQBits)
            circuit.x(QBits)
            circuit.h(QBits)

            if phaseTests:
                backend_state = Aer.get_backend('statevector_simulator')
                result = execute(circuit, backend_state).result()
                state  = result.get_statevector(circuit)
                phaseRes += [('diffuser',state)]


        print(f'Grover Algorithm applied {app} times...\n To search for␣
↪{pattern}...\n')

        return phaseRes
```

Por Fim, apresenta-se a Função de ordem superior criada. Esta função portanto, tomará como parâmetro o número de QBits de um circuito, e produzirá uma função que gerará um oráculo com

o tamanho necessário a partir de parâmetros a serem passados.

[5]:
```
def oracleHandler(nQBits):
    print(f"Generating Oracle Handler to circuit with {nQBits} QBits...\n")
    applications = floor(2**(nQBits/2))
    print(f"Recomended {applications} applications of the Gate...\n")
    auxN = max(nQBits - 3,0)
    print(f"{auxN} auxiliar QBits required to execute circuit...\n")

    def searchPat(circuit,QBits,pattern):
        for i,t in enumerate(pattern):
            if t == '0':
                circuit.x(QBits[len(pattern)- 1 - i])
        pass


    def fullCnZ(circuit,QBits,nQBits,auxQBits):

        auxLen = max(nQBits - 3,0)

        if (nQBits > 2):
            finals = [QBits[0],QBits[1]]

        for x in range(auxLen):
            circuit.ccx(finals[-2],finals[-1],auxQBits[x])
            finals += [auxQBits[x]]
            finals += [QBits[x+2]]


        #Z - CZ - CCZ
        #---------------------------------#
        switch={
            1: (lambda: circuit.x(QBits[0])),
            2: (lambda: circuit.cx(QBits[0],QBits[1])),
            'default': (lambda: circuit.
↪ccx(finals[-2],finals[-1],QBits[nQBits-1])),
            }

        circuit.h(QBits[nQBits-1])
        switch.get(nQBits,switch['default'])()
        circuit.h(QBits[nQBits-1])
        #---------------------------------#

        finals.pop()

        while len(finals) > 1:
            tgt = finals.pop()
```

```
            q2 = finals.pop()
            q1 = finals[-1]
            circuit.ccx(q1,q2,tgt)

        pass



    def␣
↪genOracle(circuit,QBits,pattern="1"*nQBits,app=applications,auxQBits=None,␣
↪phaseTests=False):
        phaseRes = []
        if phaseTests:
            backend_state = Aer.get_backend('statevector_simulator')
            result = execute(circuit, backend_state).result()
            state  = result.get_statevector(circuit)
            phaseRes += [('start',state)]

        for ap in range(app):

            #Oracle
            searchPat(circuit,QBits,pattern)
            fullCnZ(circuit,QBits,nQBits,auxQBits)
            searchPat(circuit,QBits,pattern)

            if phaseTests:
                backend_state = Aer.get_backend('statevector_simulator')
                result = execute(circuit, backend_state).result()
                state  = result.get_statevector(circuit)
                phaseRes += [('oracle',state)]


            #Diffuser
            circuit.h(QBits)
            circuit.x(QBits)
            fullCnZ(circuit,QBits,nQBits,auxQBits)
            circuit.x(QBits)
            circuit.h(QBits)

            if phaseTests:
                backend_state = Aer.get_backend('statevector_simulator')
                result = execute(circuit, backend_state).result()
                state  = result.get_statevector(circuit)
                phaseRes += [('diffuser',state)]
```

```
        print(f'Grover Algorithm applied {app} times...\n To search for␣
↪{pattern}...\n')

        return phaseRes


    if auxN:
        print(f"- returned:\n␣
↪ (circuit,QBits,pattern='{'1'*nQBits}',app={applications}, auxQBits=None,␣
↪phaseTests=False)\n")
    else:
        print(f"- returned:\n␣
↪ (circuit,QBits,pattern='{'1'*nQBits}',app={applications},␣
↪phaseTests=False)\n")
    return genOracle
```

1.5 Primeiro Circuito

A seguir, trataremos portanto de solucionar o problema proposto. A aplicação será a de um oráculo e difusor de 3 Bits, sendo $|010$ o padrão a ser procurado (Grupo $10 \equiv_8 2 = 010$). Apliquemos então as funções desenvolvidas

[6]:
```
nBits = 3
pat = '010'
```

[7]:
```
handle = oracleHandler(nBits)
```

```
Generating Oracle Handler to circuit with 3 QBits…

Recomended 2 applications of the Gate…

0 auxiliar QBits required to execute circuit…

- returned:
  (circuit,QBits,pattern='111',app=2, phaseTests=False)
```

[8]:
```
qr = QuantumRegister(nBits,'qr')
cr = ClassicalRegister(nBits,'cr')
qc_Grov = QuantumCircuit(qr,cr)
_ = qc_Grov.h(qr)
```

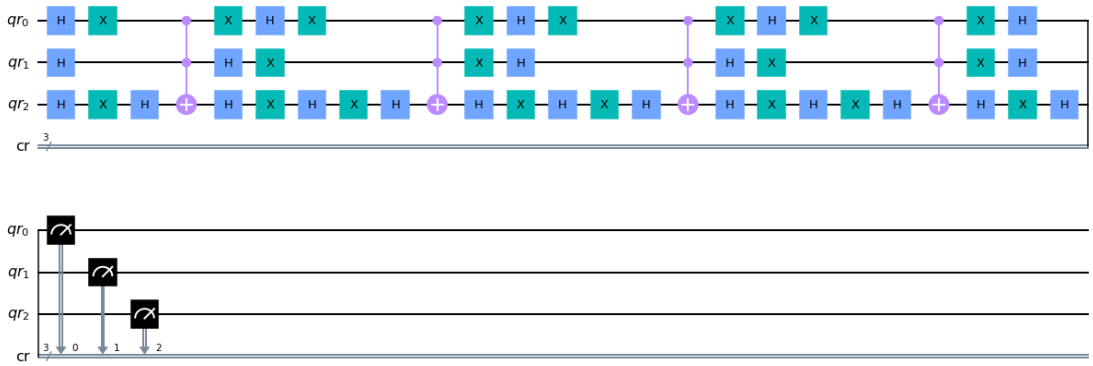
Aplicaremos, como o recomendado, $\lfloor\sqrt{(2^3)}\rfloor = 2$ iterações:

[9]:
```
_ = handle(qc_Grov,qr,pattern=pat)
```

```
Grover Algorithm applied 2 times…
 To search for 010…
```

[10]:
```
qc_Grov.measure(qr,cr)
qc_Grov.draw(output='mpl')
```

[10]:



1.6 Simulações de fases no circuito

Para agora verificarmos as propriedades descritas no algoritmo enunciado, serão simuladas as *phases* em cada iteração do oráculo e difusor,isto é, na etapa de "seleção" e "amplificação" de fase.

Será executada a função utilizada na geração do circuito apresentado, com o parâmetro `phaseTests = True`:

[11]:
```
nBits=3
handle = oracleHandler(nBits)
qr = QuantumRegister(nBits,'qr')
cr = ClassicalRegister(nBits,'cr')
qc_PhaseTest = QuantumCircuit(qr,cr)
qc_PhaseTest.h(qr)


tests = handle(qc_PhaseTest,qr,pattern= '010', phaseTests=True)
```

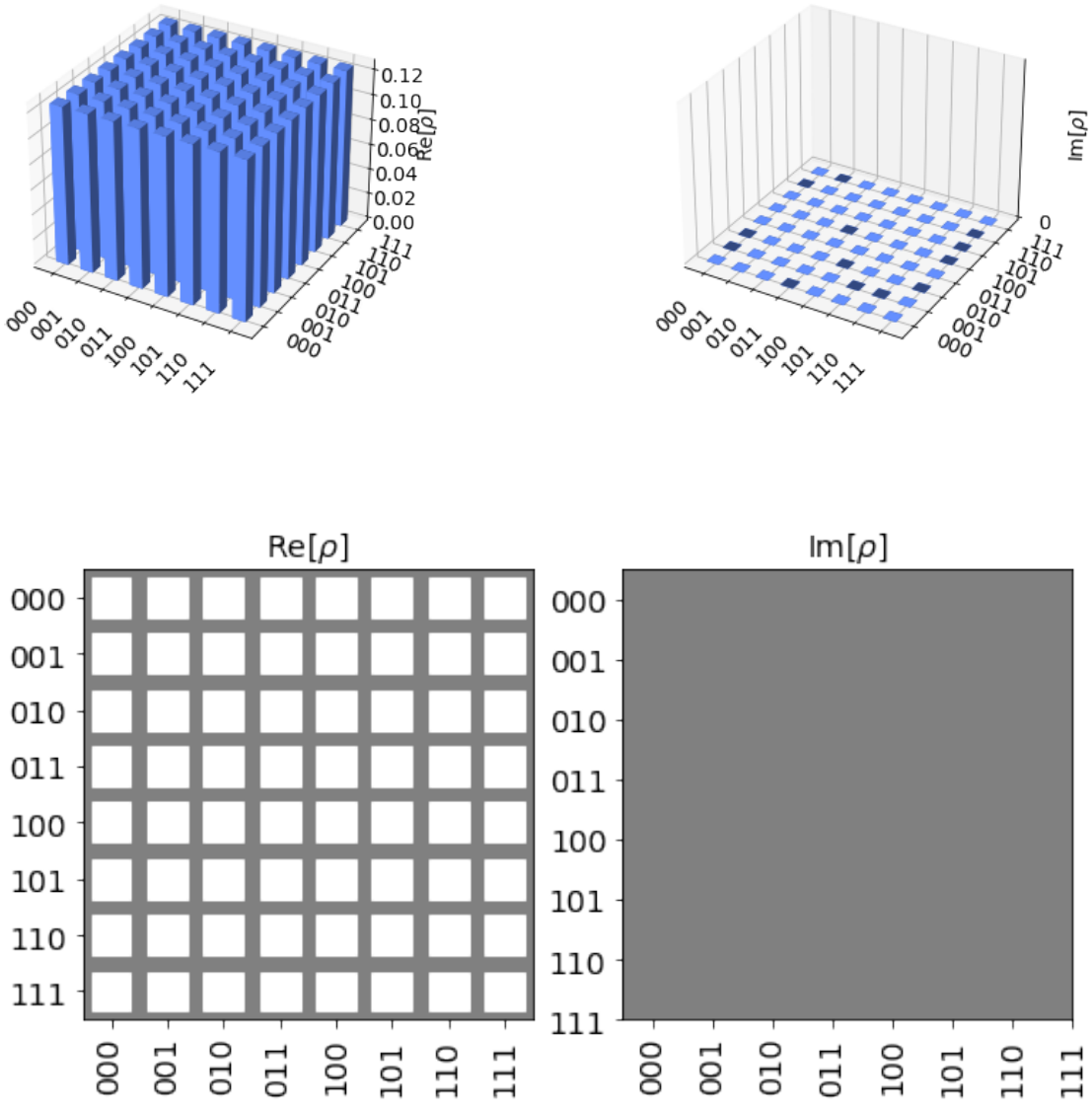
```
Generating Oracle Handler to circuit with 3 QBits…

Recomended 2 applications of the Gate…

0 auxiliar QBits required to execute circuit…

- returned:
  (circuit,QBits,pattern='111',app=2, phaseTests=False)

Grover Algorithm applied 2 times…
 To search for 010…
```

A variável `tests` apresentará então uma lista de tuplos `(etapa,estado)`, em que "etapa" corresponde ao identificador do passo dado, e "estado" a medição da fase após a execução da etapa.

Iremos então verificar um a um os *steps* de execução:

[12]:
```
(step,state) = tests.pop(0)
display(HTML(f'<h1>{step}</h1>'))
display(plot_state_city(state))
display(plot_state_hinton(state))
```

```
<IPython.core.display.HTML object>
```

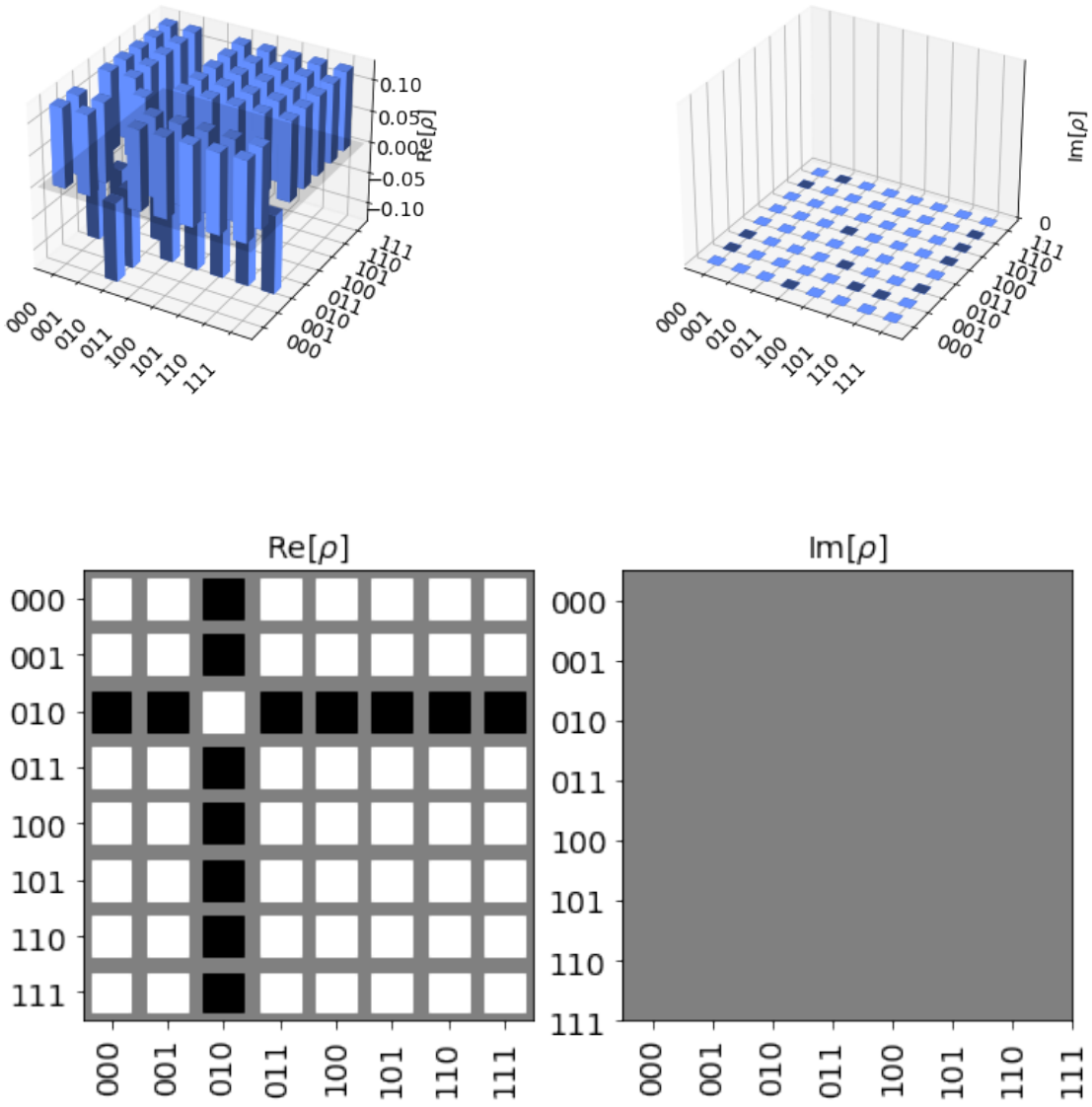


A medição apresentada acima trata-se do estado inicial do circuito,em que os fatores encontram-se todos igualmente distribuídos, A seguir apresentamos as *phases* após a aplicação do primeiro oráculo.

Como descrito anteriormente, nesta aplicação, a *phase* será invertida para valores correspondentes com o pattern procurado:

[13]:
```
(step,state) = tests.pop(0)
display(HTML(f'<h1>{step}</h1>'))
display(plot_state_city(state))
display(plot_state_hinton(state))
```

```
<IPython.core.display.HTML object>
```

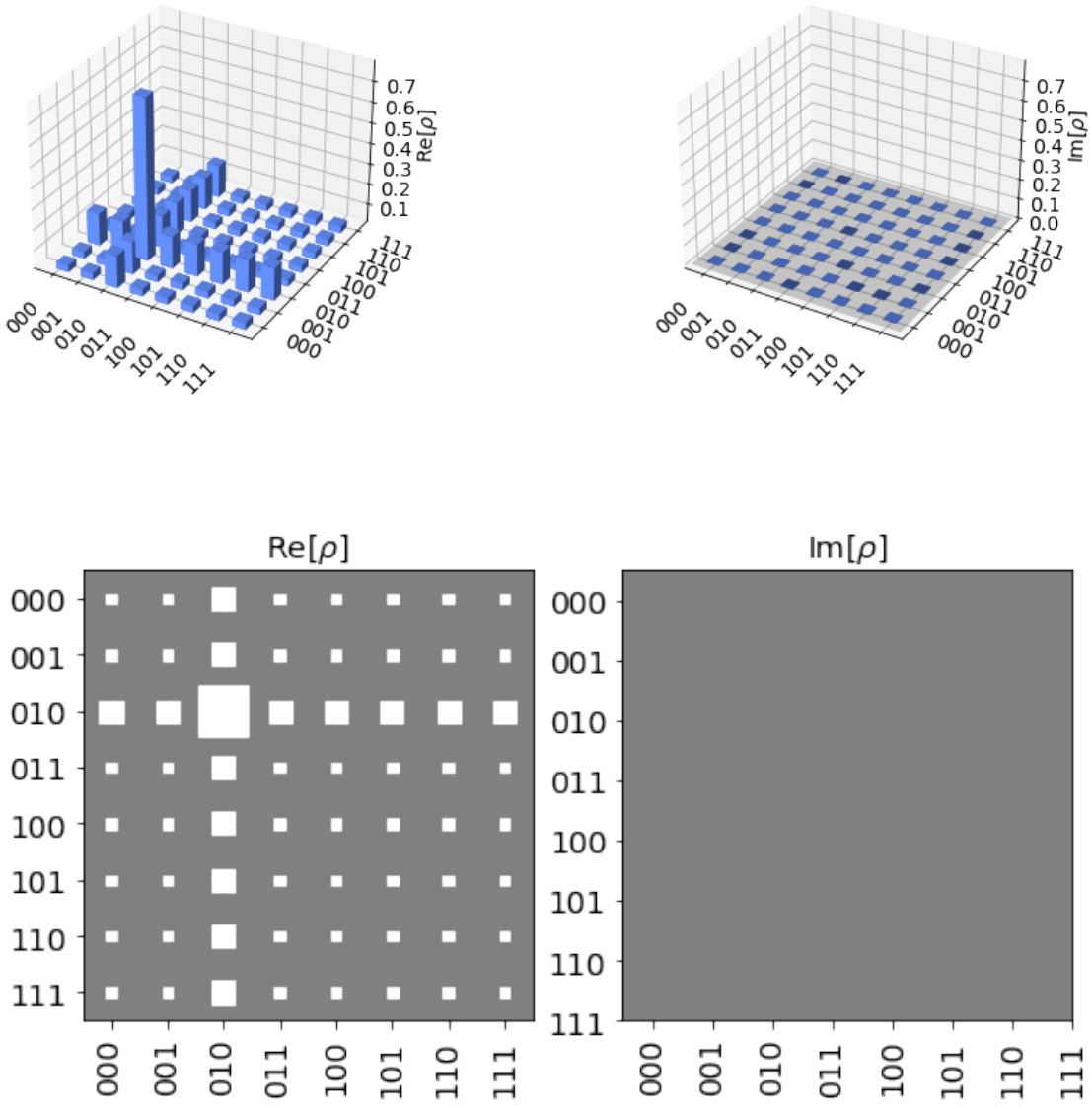


Em seguida, é aplicado o difusor, verificaremos então o comportamento requerido na especificação. Isto é, o fator a ser selecionado deve ser reinvertido e ampliado.

Verifiquemos:

[14]:
```
(step,state) = tests.pop(0)
display(HTML(f'<h1>{step}</h1>'))
display(plot_state_city(state))
display(plot_state_hinton(state))
```

```
<IPython.core.display.HTML object>
```

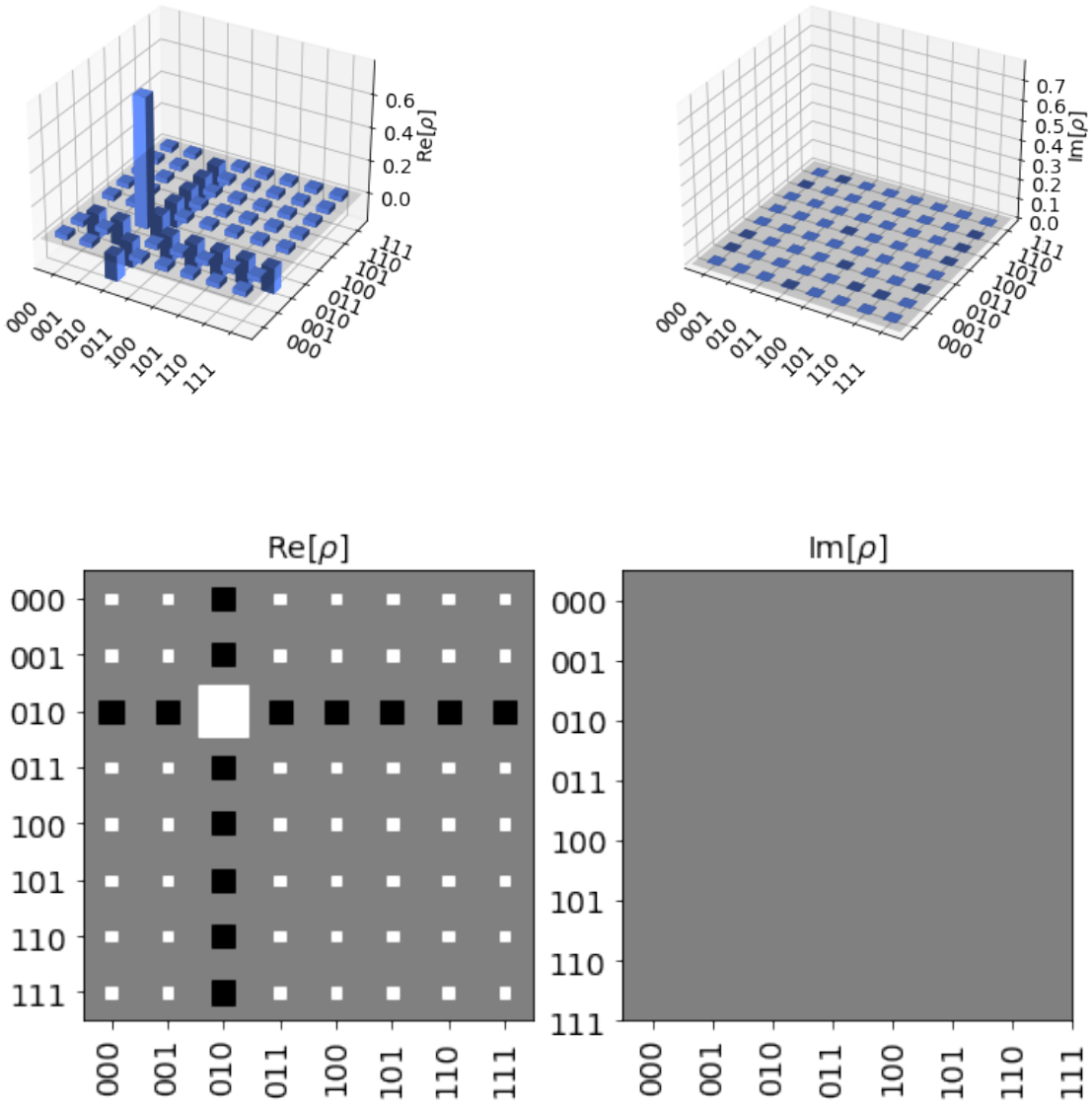


As aplicações apresentadas são então repetidas, iteremos então sobre o final da lista apresentada,

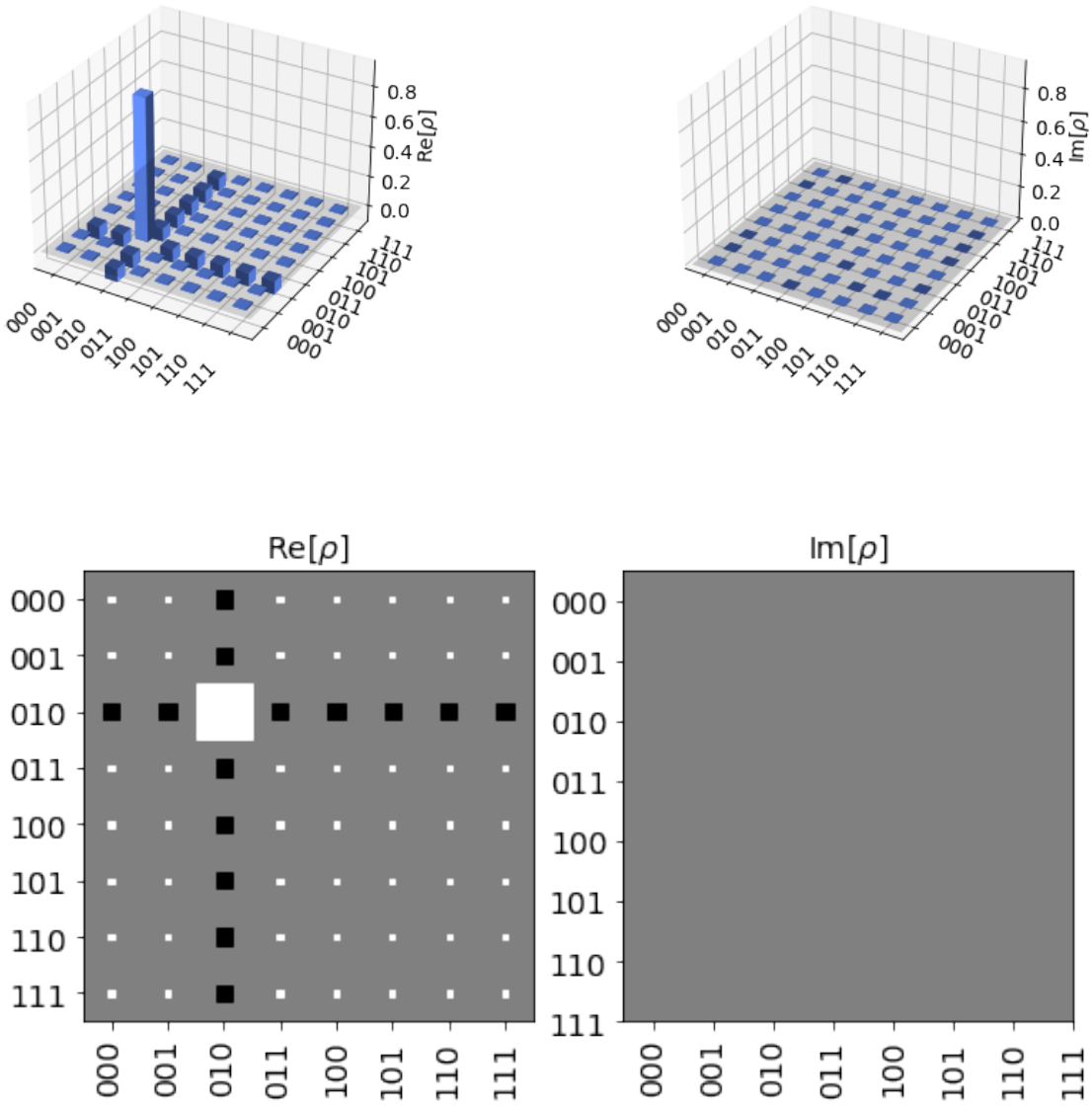
para verificarmos o bom funcionamento das iterações seguintes:

[15]:
```
for (step,state) in tests:
    display(HTML(f'<h1>{step}</h1>'))
    display(plot_state_city(state))
    display(plot_state_hinton(state))
```

```
<IPython.core.display.HTML object>
```



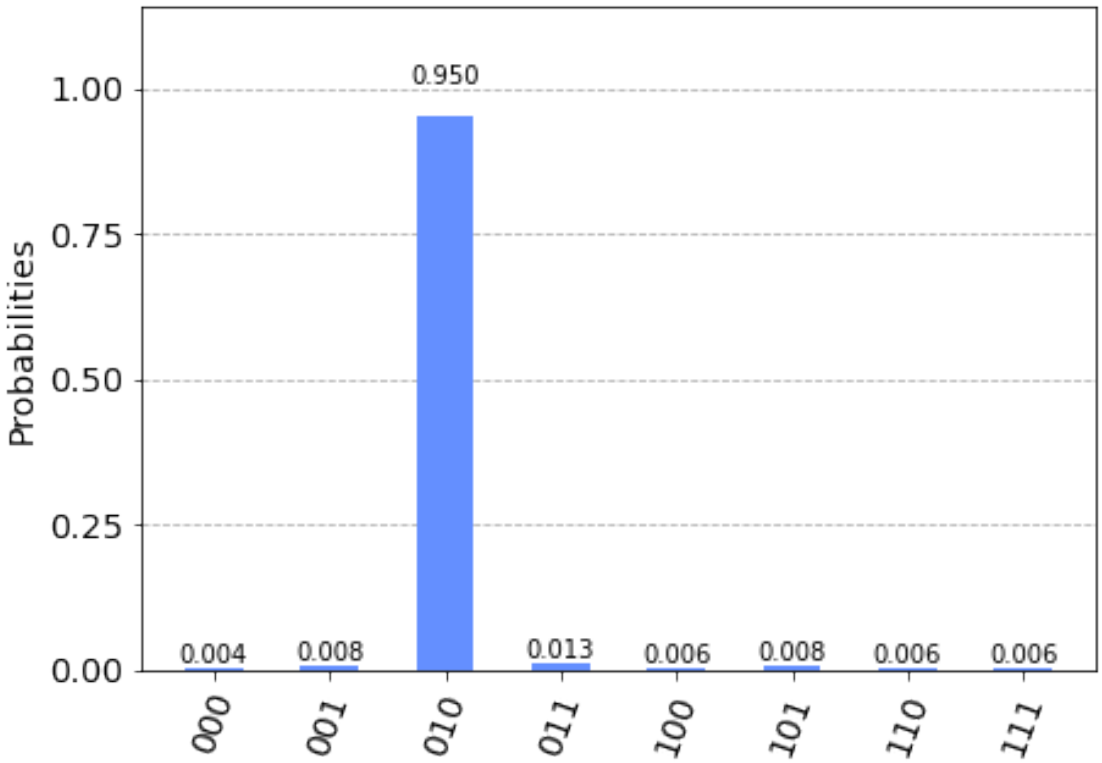

```
<IPython.core.display.HTML object>
```



1.7 Simulação do circuito apresentado

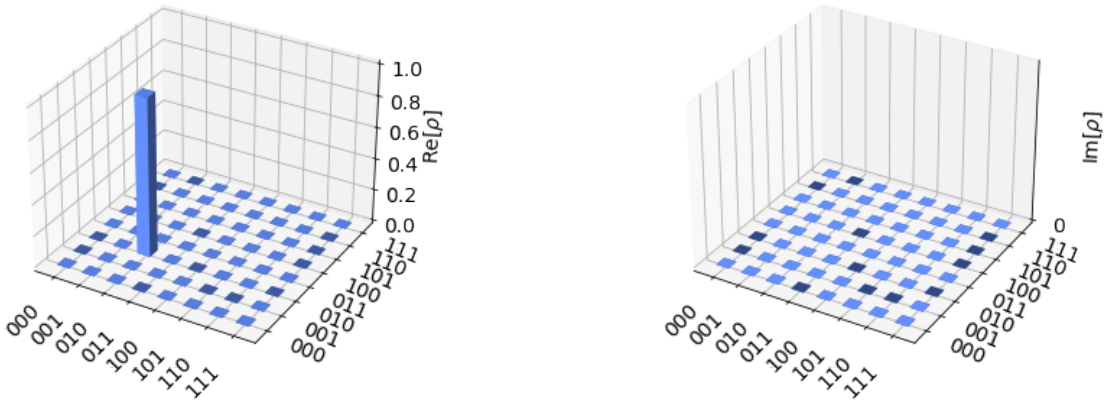
[16]:
```
backend = Aer.get_backend("qasm_simulator")
shots=1024
result = execute(qc_Grov, backend, shots=shots).result()
counts_sim = result.get_counts(qc_Grov)
plot_histogram(counts_sim)
```

[16]:



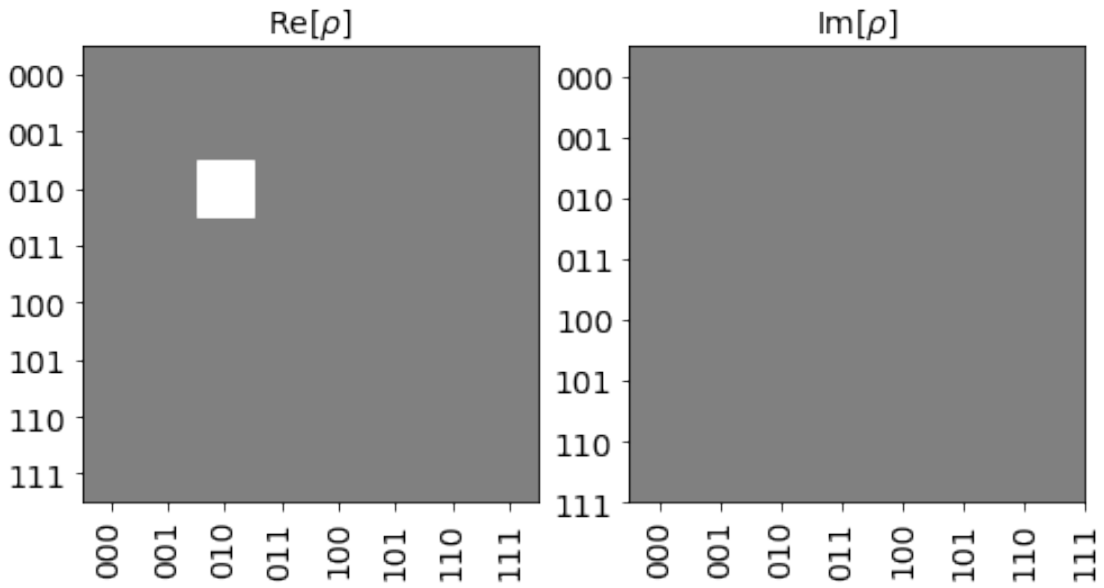
[17]:
```
backend_state = Aer.get_backend('statevector_simulator')
result = execute(qc_Grov, backend_state).result()
state  = result.get_statevector(qc_Grov)
plot_state_city(state)
```

[17]:



[18]:
```
plot_state_hinton(state)
```

[18]:



1.8 Testes em *Real Devices*

Para então utilizarmos o circuito na prática, podemos escolher um *backend* da IBM para realizar a aplicação.

[19]:
```
provider = IBMQ.load_account()
provider.backends()
```

[19]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmqx2') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_armonk') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_athens') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_santiago') 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')>]
```

[20]:
```
import qiskit.tools.jupyter

%qiskit_backend_overview
```

```
VBox(children=(HTML(value="<h2 style ='color:#ffffff; background-color:#000000;
↪padding-top: 1%; padding-bottom…
```

1.8.1 Escolha do Device

[21]:
```
from qiskit.tools.monitor import backend_overview, backend_monitor

backend_overview()
```

```
ibmq_manila
-----------
Num. Qubits:  5
Pending Jobs: 23
Least busy:   False
Operational:  True
Avg. T1:      169.0
Avg. T2:      62.0

ibmq_quito
----------
Num. Qubits:  5
Pending Jobs: 7
Least busy:   True
Operational:  True
Avg. T1:      85.7
Avg. T2:      77.5

ibmq_belem
----------
Num. Qubits:  5
Pending Jobs: 9
Least busy:   False
Operational:  True
Avg. T1:      73.7
Avg. T2:      82.0

ibmq_lima
---------
Num. Qubits:  5
Pending Jobs: 10
Least busy:   False
Operational:  True
Avg. T1:      68.9
Avg. T2:      74.2

ibmq_santiago
-------------
Num. Qubits:  5
Pending Jobs: 10
Least busy:   False
Operational:  True
Avg. T1:      109.9
Avg. T2:      115.2

ibmq_athens
-----------
Num. Qubits:  5
Pending Jobs: 10
Least busy:   False
Operational:  True
Avg. T1:      100.4
Avg. T2:      116.7

ibmq_armonk
-----------
Num. Qubits:  1
Pending Jobs: 35

ibmq_16_melbourne
-----------------
Num. Qubits:  15
Pending Jobs: 16

ibmqx2
------
Num. Qubits:  5
Pending Jobs: 557
```

```
Least busy:   False          Least busy:   False          Least busy:   False
Operational:  True           Operational:  True           Operational:  True
Avg. T1:      179.6          Avg. T1:      58.7           Avg. T1:      58.3
Avg. T2:      223.8          Avg. T2:      56.1           Avg. T2:      38.4
```

O *device* escolhido tratou-se do `ibmq_santiago`. Para realizar a escolha, procuramos equilibrar a taxa de erro dos gates cX de grande importância no algoritmo apresentado, com as medidas de T1 e T2, e com o tamanho da *queue* de *jobs*.

O `ibmq_santiago` possuia uma fila pequena, com taxas de erros aceitáveis, logo fora o escolhido. A segunda opção viria a ser `ibmq_athens`, mas este apresentava uma taxa de erro elevada de cX em alguns pares de QBits.

O `ibmq_athens`, entretanto, possuía menores medições de T1 e T2, o que, por exeperiências passadas colocou o grupo em dúvida na escolha entre tais *devices*.

O T1 corresponde ao tempo que o ruído leva para alterar o estado de um qubit aleatoriamente (por exemplo de $|0\rangle$ para $|1\rangle$), enquanto o T2 corresponde ao tempo que o ruído leva para alterar a amplitude de cada uma das bases de um qubit em sobreposição, podendo assim causar um enviesamento para um determinado estado.

Assim o T2, que possui maior impacto na amplitude (de muita importância no algoritmo apresentado), seria o *target* (entre T1 e T2) na escolha (entre Santiago e Athens).

Levando em conta a última informação, optamos pelo `ibmq_santiago` devido à diferença entre o T2 dos dois *backends* em questão, não ser tanta comparada à diferença no erro dos *gates* cX.

A seguir apresentamos informações acerca dos dois dispositivos em questão.

[22]:
```
backend_device = provider.get_backend('ibmq_santiago')
print("Running on: ", backend_device)
```

```
Running on:  ibmq_santiago
```

[25]:
```
backend_monitor(backend_device)
```

```
ibmq_santiago
=============
Configuration
-------------
    n_qubits: 5
    operational: True
    status_msg: active
    pending_jobs: 8
    backend_version: 1.3.22
    basis_gates: ['id', 'rz', 'sx', 'x', 'cx', 'reset']
    local: False
    simulator: False
    meas_kernels: ['hw_qmfk']
```

```
    discriminators: ['quadratic_discriminator', 'hw_qmfk',
'linear_discriminator']
    url: None
    meas_levels: [1, 2]
    n_registers: 1
    credits_required: True
    conditional_latency: []
    memory: True
    dynamic_reprate_enabled: True
    allow_q_object: True
    sample_name: family: Falcon, revision: 4, segment: L
    coupling_map: [[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2], [3, 4], [4,
3]]
    hamiltonian: {'description': 'Qubits are modeled as Duffing oscillators. In
this case, the system includes higher energy states, i.e. not just |0> and |1>.
The Pauli operators are generalized via the following set of
transformations:\n\n$(\\mathbb{I}-\\sigma_{i}^z)/2 \\rightarrow O_i \\equiv
b^\\dagger_{i} b_{i}$,\n\n$\\sigma_{+} \\rightarrow b^\\dagger$,\n\n$\\sigma_{-}
\\rightarrow b$,\n\n$\\sigma_{i}^X \\rightarrow b^\\dagger_{i} +
b_{i}$.\n\nQubits are coupled through resonator buses. The provided Hamiltonian
has been projected into the zero excitation subspace of the resonator buses
leading to an effective qubit-qubit flip-flop interaction. The qubit resonance
frequencies in the Hamiltonian are the cavity dressed frequencies and not
exactly what is returned by the backend defaults, which also includes the
dressing due to the qubit-qubit interactions.\n\nQuantities are returned in
angular frequencies, with units 2*pi*GHz.\n\nWARNING: Currently not all system
Hamiltonian information is available to the public, missing values have been
replaced with 0.\n', 'h_latex': '\\begin{align} \\mathcal{H}/\\hbar = & \\sum_{i
=0}^{4}\\left(\\frac{\\omega_{q,i}}{2}(\\mathbb{I}-\\sigma_i^{z})+\\frac{\\Delta
_{i}}{2}(O_i^2-O_i)+\\Omega_{d,i}D_i(t)\\sigma_i^{X}\\right) \\\\ & +
J_{0,1}(\\sigma_{0}^{+}\\sigma_{1}^{-}+\\sigma_{0}^{-}\\sigma_{1}^{+}) +
J_{3,4}(\\sigma_{3}^{+}\\sigma_{4}^{-}+\\sigma_{3}^{-}\\sigma_{4}^{+}) +
J_{2,3}(\\sigma_{2}^{+}\\sigma_{3}^{-}+\\sigma_{2}^{-}\\sigma_{3}^{+}) +
J_{1,2}(\\sigma_{1}^{+}\\sigma_{2}^{-}+\\sigma_{1}^{-}\\sigma_{2}^{+}) \\\\ & +
\\Omega_{d,0}(U_{0}^{(0,1)}(t))\\sigma_{0}^{X} +
\\Omega_{d,1}(U_{1}^{(1,0)}(t)+U_{2}^{(1,2)}(t))\\sigma_{1}^{X} \\\\ & +
\\Omega_{d,2}(U_{3}^{(2,1)}(t)+U_{4}^{(2,3)}(t))\\sigma_{2}^{X} +
\\Omega_{d,3}(U_{6}^{(3,4)}(t)+U_{5}^{(3,2)}(t))\\sigma_{3}^{X} \\\\ & +
\\Omega_{d,4}(U_{7}^{(4,3)}(t))\\sigma_{4}^{X} \\\\ \\end{align}', 'h_str':
['_SUM[i,0,4,wq{i}/2*(I{i}-Z{i})]', '_SUM[i,0,4,delta{i}/2*O{i}*O{i}]',
'_SUM[i,0,4,-delta{i}/2*O{i}]', '_SUM[i,0,4,omegad{i}*X{i}||D{i}]',
'jq0q1*Sp0*Sm1', 'jq0q1*Sm0*Sp1', 'jq3q4*Sp3*Sm4', 'jq3q4*Sm3*Sp4',
'jq2q3*Sp2*Sm3', 'jq2q3*Sm2*Sp3', 'jq1q2*Sp1*Sm2', 'jq1q2*Sm1*Sp2',
'omegad1*X0||U0', 'omegad0*X1||U1', 'omegad2*X1||U2', 'omegad1*X2||U3',
'omegad3*X2||U4', 'omegad4*X3||U6', 'omegad2*X3||U5', 'omegad3*X4||U7'], 'osc':
{}, 'qub': {'0': 3, '1': 3, '2': 3, '3': 3, '4': 3}, 'vars': {'delta0':
-2.1481278490714906, 'delta1': -2.0623435150768743, 'delta2':
-2.1429828509850863, 'delta3': -2.137118237032298, 'delta4': -2.154596484455155,
```

'jq0q1': 0.007378105608801839, 'jq1q2': 0.007268700678758498, 'jq2q3':
0.007255936195908655, 'jq3q4': 0.006881064755295536, 'omegad0':
1.0122660261853713, 'omegad1': 0.9882668447893664, 'omegad2': 1.005255042398994,
'omegad3': 1.010215348492549, 'omegad4': 1.003471829038666, 'wq0':
30.36932716381236, 'wq1': 29.051015143106522, 'wq2': 30.288332873692667, 'wq3':
29.79683200369637, 'wq4': 30.261867686111472}}
    default_rep_delay: 250.0
    meas_lo_range: [[6.952624018e+18, 7.952624018e+18], [6.701014434e+18,
7.701014434e+18], [6.837332258e+18, 7.837332258e+18], [6.901770712e+18,
7.901770712e+18], [6.775814414e+18, 7.775814414e+18]]
    u_channel_lo: [[{'q': 1, 'scale': (1+0j)}], [{'q': 0, 'scale': (1+0j)}],
[{'q': 2, 'scale': (1+0j)}], [{'q': 1, 'scale': (1+0j)}], [{'q': 3, 'scale':
(1+0j)}], [{'q': 2, 'scale': (1+0j)}], [{'q': 4, 'scale': (1+0j)}], [{'q': 3,
'scale': (1+0j)}]]
    qubit_channel_mapping: [['u0', 'd0', 'm0', 'u1'], ['u3', 'd1', 'u2', 'u0',
'm1', 'u1'], ['u3', 'u2', 'd2', 'u4', 'm2', 'u5'], ['m3', 'd3', 'u6', 'u4',
'u7', 'u5'], ['d4', 'm4', 'u6', 'u7']]
    dtm: 0.2222222222222222
    rep_times: [0.001]
    pulse_num_qubits: 3
    online_date: 2020-06-03 04:00:00+00:00
    processor_type: {'family': 'Falcon', 'revision': 4, 'segment': 'L'}
    qubit_lo_range: [[4.3334285364957076e+18, 5.333428536495707e+18],
[4.123612661862908e+18, 5.123612661862908e+18], [4.3205378948609393e+18,
5.320537894860939e+18], [4.242313101867062e+18, 5.242313101867061e+18],
[4.3163258294375373e+18, 5.316325829437537e+18]]
    conditional: False
    uchannels_enabled: True
    supported_instructions: ['u3', 'measure', 'setf', 'u2', 'id', 'delay', 'cx',
'acquire', 'sx', 'x', 'reset', 'u1', 'shiftf', 'play', 'rz']
    max_shots: 8192
    rep_delay_range: [0.0, 500.0]
    description: 5 qubit device
    meas_map: [[0, 1, 2, 3, 4]]
    pulse_num_channels: 9
    open_pulse: False
    acquisition_latency: []
    multi_meas_enabled: True
    max_experiments: 75
    allow_object_storage: True
    parametric_pulses: ['gaussian', 'gaussian_square', 'drag', 'constant']
    quantum_volume: 32
    n_uchannels: 8
    dt: 0.2222222222222222
    backend_name: ibmq_santiago
    channels: {'acquire0': {'operates': {'qubits': [0]}, 'purpose': 'acquire',
'type': 'acquire'}, 'acquire1': {'operates': {'qubits': [1]}, 'purpose':
'acquire', 'type': 'acquire'}, 'acquire2': {'operates': {'qubits': [2]},

'purpose': 'acquire', 'type': 'acquire'}, 'acquire3': {'operates': {'qubits':
[3]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire4': {'operates':
{'qubits': [4]}, 'purpose': 'acquire', 'type': 'acquire'}, 'd0': {'operates':
{'qubits': [0]}, 'purpose': 'drive', 'type': 'drive'}, 'd1': {'operates':
{'qubits': [1]}, 'purpose': 'drive', 'type': 'drive'}, 'd2': {'operates':
{'qubits': [2]}, 'purpose': 'drive', 'type': 'drive'}, 'd3': {'operates':
{'qubits': [3]}, 'purpose': 'drive', 'type': 'drive'}, 'd4': {'operates':
{'qubits': [4]}, 'purpose': 'drive', 'type': 'drive'}, 'm0': {'operates':
{'qubits': [0]}, 'purpose': 'measure', 'type': 'measure'}, 'm1': {'operates':
{'qubits': [1]}, 'purpose': 'measure', 'type': 'measure'}, 'm2': {'operates':
{'qubits': [2]}, 'purpose': 'measure', 'type': 'measure'}, 'm3': {'operates':
{'qubits': [3]}, 'purpose': 'measure', 'type': 'measure'}, 'm4': {'operates':
{'qubits': [4]}, 'purpose': 'measure', 'type': 'measure'}, 'u0': {'operates':
{'qubits': [0, 1]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u1':
{'operates': {'qubits': [1, 0]}, 'purpose': 'cross-resonance', 'type':
'control'}, 'u2': {'operates': {'qubits': [1, 2]}, 'purpose': 'cross-resonance',
'type': 'control'}, 'u3': {'operates': {'qubits': [2, 1]}, 'purpose': 'cross-
resonance', 'type': 'control'}, 'u4': {'operates': {'qubits': [2, 3]},
'purpose': 'cross-resonance', 'type': 'control'}, 'u5': {'operates': {'qubits':
[3, 2]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u6': {'operates':
{'qubits': [3, 4]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u7':
{'operates': {'qubits': [4, 3]}, 'purpose': 'cross-resonance', 'type':
'control'}}

Qubits [Name / Freq / T1 / T2 / RZ err / SX err / X err / Readout err]

Q0 / 4.83343 GHz / 108.98769 us / 177.14257 us / 0.00000 / 0.00018 / 0.00018
/ 0.01990
Q1 / 4.62361 GHz / 141.50940 us / 98.38286 us / 0.00000 / 0.00017 / 0.00017
/ 0.01180
Q2 / 4.82054 GHz / 78.29741 us / 76.97528 us / 0.00000 / 0.00021 / 0.00021 /
0.01390
Q3 / 4.74231 GHz / 108.87670 us / 81.31520 us / 0.00000 / 0.00018 / 0.00018
/ 0.00620
Q4 / 4.81633 GHz / 111.62712 us / 142.06860 us / 0.00000 / 0.00029 / 0.00029
/ 0.01900

Multi-Qubit Gates [Name / Type / Gate Error]
---

cx4_3 / cx / 0.00510
cx3_4 / cx / 0.00510
cx2_3 / cx / 0.00499
cx3_2 / cx / 0.00499
cx2_1 / cx / 0.00587
cx1_2 / cx / 0.00587
cx0_1 / cx / 0.00592
cx1_0 / cx / 0.00592

[44]:
```
backend_monitor(provider.get_backend('ibmq_athens'))
```

```
ibmq_athens
===========
Configuration
-------------
    n_qubits: 5
    operational: True
    status_msg: active
    pending_jobs: 10
    backend_version: 1.3.19
    basis_gates: ['id', 'rz', 'sx', 'x', 'cx', 'reset']
    local: False
    simulator: False
    meas_kernels: ['hw_qmfk']
    discriminators: ['quadratic_discriminator', 'hw_qmfk',
'linear_discriminator']
    url: None
    meas_levels: [1, 2]
    n_registers: 1
    credits_required: True
    conditional_latency: []
    memory: True
    dynamic_reprate_enabled: True
    allow_q_object: True
    sample_name: family: Falcon, revision: 4, segment: L
    coupling_map: [[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2], [3, 4], [4,
3]]
    hamiltonian: {'description': 'Qubits are modeled as Duffing oscillators. In
this case, the system includes higher energy states, i.e. not just |0> and |1>.
The Pauli operators are generalized via the following set of
transformations:\n\n$(\\mathbb{I}-\\sigma_{i}^z)/2 \\rightarrow O_i \\equiv
b^\\dagger_{i} b_{i}$,\n\n$\\sigma_{+} \\rightarrow b^\\dagger$,\n\n$\\sigma_{-}
\\rightarrow b$,\n\n$\\sigma_{i}^X \\rightarrow b^\\dagger_{i} +
b_{i}$.\n\nQubits are coupled through resonator buses. The provided Hamiltonian
has been projected into the zero excitation subspace of the resonator buses
leading to an effective qubit-qubit flip-flop interaction. The qubit resonance
frequencies in the Hamiltonian are the cavity dressed frequencies and not
exactly what is returned by the backend defaults, which also includes the
dressing due to the qubit-qubit interactions.\n\nQuantities are returned in
angular frequencies, with units 2*pi*GHz.\n\nWARNING: Currently not all system
Hamiltonian information is available to the public, missing values have been
replaced with 0.\n', 'h_latex': '\\begin{align} \\mathcal{H}/\\hbar = & \\sum_{i
=0}^{4}\\left(\\frac{\\omega_{q,i}}{2}(\\mathbb{I}-\\sigma_i^{z})+\\frac{\\Delta
_{i}}{2}(O_i^2-O_i)+\\Omega_{d,i}D_i(t)\\sigma_i^{X}\\right) \\\\ & +
J_{1,2}(\\sigma_{1}^{+}\\sigma_{2}^{-}+\\sigma_{1}^{-}\\sigma_{2}^{+}) +
J_{3,4}(\\sigma_{3}^{+}\\sigma_{4}^{-}+\\sigma_{3}^{-}\\sigma_{4}^{+}) +
```

```
J_{0,1}(\\sigma_{0}^{+}\\sigma_{1}^{-}+\\sigma_{0}^{-}\\sigma_{1}^{+}) +
J_{2,3}(\\sigma_{2}^{+}\\sigma_{3}^{-}+\\sigma_{2}^{-}\\sigma_{3}^{+}) \\\\ & +
\\Omega_{d,0}(U_{0}^{(0,1)}(t))\\sigma_{0}^{X} +
\\Omega_{d,1}(U_{1}^{(1,0)}(t)+U_{2}^{(1,2)}(t))\\sigma_{1}^{X} \\\\ & +
\\Omega_{d,2}(U_{3}^{(2,1)}(t)+U_{4}^{(2,3)}(t))\\sigma_{2}^{X} +
\\Omega_{d,3}(U_{6}^{(3,4)}(t)+U_{5}^{(3,2)}(t))\\sigma_{3}^{X} \\\\ & +
\\Omega_{d,4}(U_{7}^{(4,3)}(t))\\sigma_{4}^{X} \\\\ \\end{align}', 'h_str':
['_SUM[i,0,4,wq{i}/2*(I{i}-Z{i})]', '_SUM[i,0,4,delta{i}/2*O{i}*O{i}]',
'_SUM[i,0,4,-delta{i}/2*O{i}]', '_SUM[i,0,4,omegad{i}*X{i}||D{i}]',
'jq1q2*Sp1*Sm2', 'jq1q2*Sm1*Sp2', 'jq3q4*Sp3*Sm4', 'jq3q4*Sm3*Sp4',
'jq0q1*Sp0*Sm1', 'jq0q1*Sm0*Sp1', 'jq2q3*Sp2*Sm3', 'jq2q3*Sm2*Sp3',
'omegad1*X0||U0', 'omegad0*X1||U1', 'omegad2*X1||U2', 'omegad1*X2||U3',
'omegad3*X2||U4', 'omegad4*X3||U6', 'omegad2*X3||U5', 'omegad3*X4||U7'], 'osc':
{}, 'qub': {'0': 3, '1': 3, '2': 3, '3': 3, '4': 3}, 'vars': {'delta0':
-2.1117934764003934, 'delta1': -2.0894421352015744, 'delta2':
-2.1179183671068604, 'delta3': -2.0410045431261215, 'delta4':
-2.1119885565086776, 'jq0q1': 0.010495754104003914, 'jq1q2':
0.01078171551120001, 'jq2q3': 0.008920779377814226, 'jq3q4':
0.008985191651087791, 'omegad0': 0.96949166086591, 'omegad1':
0.9786118546687342, 'omegad2': 0.9473767537036452, 'omegad3':
0.9723569587342615, 'omegad4': 0.9809400192634314, 'wq0': 32.517884262974626,
'wq1': 33.09492014632308, 'wq2': 31.7454748985563, 'wq3': 30.510705418353474,
'wq4': 32.1607203205431}}
    default_rep_delay: 250.0
    meas_lo_range: [[6.714107437e+18, 7.714107437e+18], [6.860372122e+18,
7.860372122e+18], [6.798419517e+18, 7.798419517e+18], [6.663115298e+18,
7.663115298e+18], [6.911673028e+18, 7.911673028e+18]]
    u_channel_lo: [[{'q': 1, 'scale': (1+0j)}], [{'q': 0, 'scale': (1+0j)}],
[{'q': 2, 'scale': (1+0j)}], [{'q': 1, 'scale': (1+0j)}], [{'q': 3, 'scale':
(1+0j)}], [{'q': 2, 'scale': (1+0j)}], [{'q': 4, 'scale': (1+0j)}], [{'q': 3,
'scale': (1+0j)}]]
    qubit_channel_mapping: [['u0', 'd0', 'm0', 'u1'], ['u2', 'u1', 'm1', 'u0',
'd1', 'u3'], ['u2', 'd2', 'u5', 'u4', 'u3', 'm2'], ['u6', 'u7', 'u4', 'd3',
'u5', 'm3'], ['m4', 'u6', 'd4', 'u7']]
    dtm: 0.2222222222222222
    rep_times: [0.001]
    pulse_num_qubits: 3
    online_date: 2020-03-13 04:00:00+00:00
    processor_type: {'family': 'Falcon', 'revision': 4, 'segment': 'L'}
    qubit_lo_range: [[4.675382019342564e+18, 5.675382019342564e+18],
[4.76722013251887e+18, 5.76722013251887e+18], [4.5524492509049196e+18,
5.55244925090492e+18], [4.35592958455163e+18, 5.35592958455163e+18],
[4.618537612410399e+18, 5.618537612410399e+18]]
    conditional: False
    uchannels_enabled: True
    supported_instructions: ['u2', 'u1', 'setf', 'x', 'acquire', 'sx', 'cx',
'rz', 'reset', 'measure', 'delay', 'shiftf', 'id', 'u3', 'play']
    max_shots: 8192
```

```
    rep_delay_range: [0.0, 500.0]
    description: 5 qubit device
    meas_map: [[0, 1, 2, 3, 4]]
    pulse_num_channels: 9
    open_pulse: False
    acquisition_latency: []
    multi_meas_enabled: True
    max_experiments: 75
    allow_object_storage: True
    parametric_pulses: ['gaussian', 'gaussian_square', 'drag', 'constant']
    quantum_volume: 32
    n_uchannels: 8
    dt: 0.2222222222222222
    backend_name: ibmq_athens
    channels: {'acquire0': {'operates': {'qubits': [0]}, 'purpose': 'acquire',
'type': 'acquire'}, 'acquire1': {'operates': {'qubits': [1]}, 'purpose':
'acquire', 'type': 'acquire'}, 'acquire2': {'operates': {'qubits': [2]},
'purpose': 'acquire', 'type': 'acquire'}, 'acquire3': {'operates': {'qubits':
[3]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire4': {'operates':
{'qubits': [4]}, 'purpose': 'acquire', 'type': 'acquire'}, 'd0': {'operates':
{'qubits': [0]}, 'purpose': 'drive', 'type': 'drive'}, 'd1': {'operates':
{'qubits': [1]}, 'purpose': 'drive', 'type': 'drive'}, 'd2': {'operates':
{'qubits': [2]}, 'purpose': 'drive', 'type': 'drive'}, 'd3': {'operates':
{'qubits': [3]}, 'purpose': 'drive', 'type': 'drive'}, 'd4': {'operates':
{'qubits': [4]}, 'purpose': 'drive', 'type': 'drive'}, 'm0': {'operates':
{'qubits': [0]}, 'purpose': 'measure', 'type': 'measure'}, 'm1': {'operates':
{'qubits': [1]}, 'purpose': 'measure', 'type': 'measure'}, 'm2': {'operates':
{'qubits': [2]}, 'purpose': 'measure', 'type': 'measure'}, 'm3': {'operates':
{'qubits': [3]}, 'purpose': 'measure', 'type': 'measure'}, 'm4': {'operates':
{'qubits': [4]}, 'purpose': 'measure', 'type': 'measure'}, 'u0': {'operates':
{'qubits': [0, 1]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u1':
{'operates': {'qubits': [1, 0]}, 'purpose': 'cross-resonance', 'type':
'control'}, 'u2': {'operates': {'qubits': [1, 2]}, 'purpose': 'cross-resonance',
'type': 'control'}, 'u3': {'operates': {'qubits': [2, 1]}, 'purpose': 'cross-
resonance', 'type': 'control'}, 'u4': {'operates': {'qubits': [2, 3]},
'purpose': 'cross-resonance', 'type': 'control'}, 'u5': {'operates': {'qubits':
[3, 2]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u6': {'operates':
{'qubits': [3, 4]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u7':
{'operates': {'qubits': [4, 3]}, 'purpose': 'cross-resonance', 'type':
'control'}}

Qubits [Name / Freq / T1 / T2 / RZ err / SX err / X err / Readout err]
--------------------------------------------------------------------
    Q0 / 5.17538 GHz / 97.15297 us / 172.18687 us / 0.00000 / 0.00031 / 0.00031
/ 0.01000
    Q1 / 5.26722 GHz / 89.40801 us / 126.47599 us / 0.00000 / 0.00036 / 0.00036
/ 0.01350
    Q2 / 5.05245 GHz / 126.57496 us / 156.45419 us / 0.00000 / 0.00034 / 0.00034
```

```
/ 0.02820
    Q3 / 4.85593 GHz / 84.73470 us / 26.87493 us / 0.00000 / 0.00019 / 0.00019 /
0.01150
    Q4 / 5.11854 GHz / 103.98292 us / 101.31362 us / 0.00000 / 0.00024 / 0.00024
/ 0.01680

Multi-Qubit Gates [Name / Type / Gate Error]
------------------------------------------
    cx4_3 / cx / 0.00631
    cx3_4 / cx / 0.00631
    cx2_3 / cx / 0.00853
    cx3_2 / cx / 0.00853
    cx1_2 / cx / 0.00818
    cx2_1 / cx / 0.00818
    cx1_0 / cx / 0.01001
    cx0_1 / cx / 0.01001
```

[26]:
```
%qiskit_job_watcher
```

```
Accordion(children=(VBox(layout=Layout(max_width='710px', min_width='710px')),),␣
↪layout=Layout(max_height='500…
```

```
<IPython.core.display.Javascript object>
```

1.9 Resultados

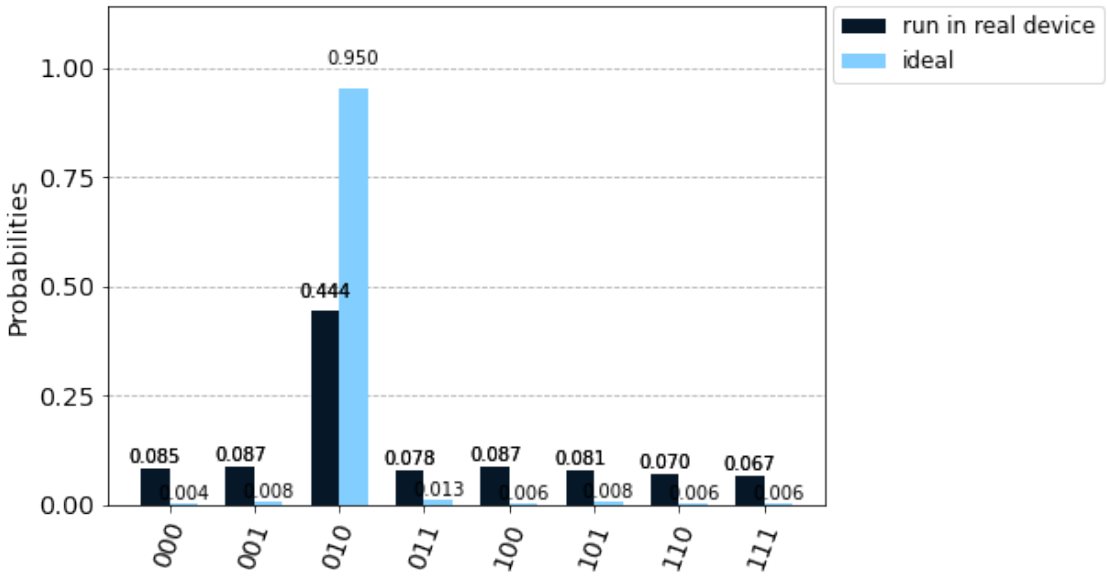
[27]:
```
job_r = execute(qc_Grov, backend_device, shots=shots)
jobID_r = job_r.job_id()
print(f'JOB ID: {jobID_r}')
```

```
JOB ID: 60b8e4b19acb6d21ddcc4c0f
```

[28]:
```
job_get=backend_device.retrieve_job("60b8e4b19acb6d21ddcc4c0f")

result_r = job_get.result()
counts_run = result_r.get_counts(qc_Grov)
```

[29]:
```
plot_histogram([counts_run, counts_sim ], legend=[ 'run in real device',␣
↪'ideal'], color=['#061727','#82cfff'])
```

[29]:



1.10 Variante do circuito

Além do resultado apresentado, acreditávamos ser possível diminuir a taxa de erro diminuindo a depth do circuito e reduzindo probabilidades de ruído. Apresenta-se então o mesmo circuito realizado no tópico anterior, mas com a aplicação de uma única iteração do algoritmo.

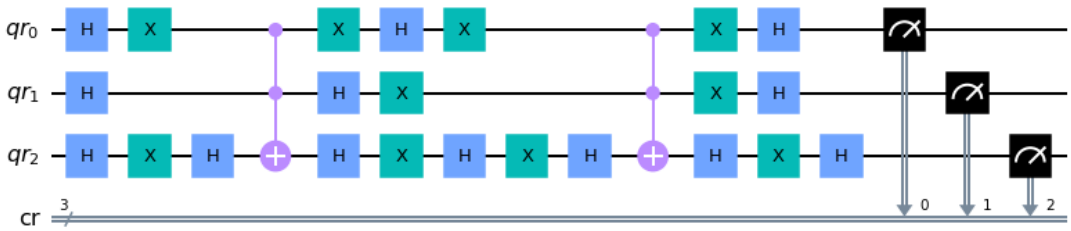
[30]:
```
qr = QuantumRegister(nBits,'qr')
cr = ClassicalRegister(nBits,'cr')
qc_Grov = QuantumCircuit(qr,cr)
_ = qc_Grov.h(qr)
```

[31]:
```
_ = handle(qc_Grov,qr,pattern= '010',app=1)
```

```
Grover Algorithm applied 1 times…
 To search for 010…
```

[32]:
```
qc_Grov.measure(qr,cr)
qc_Grov.draw(output='mpl')
```

[32]:



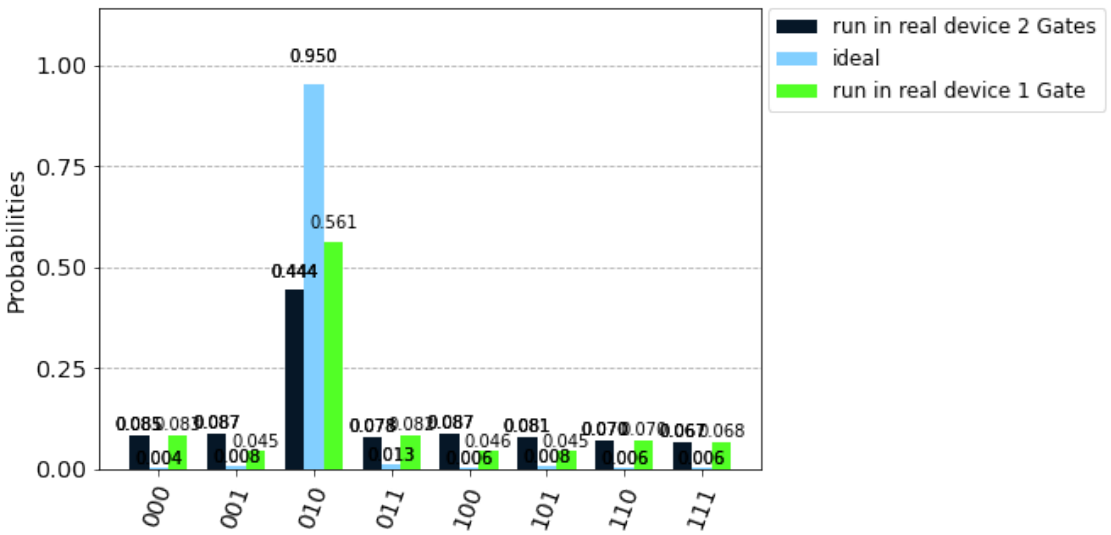
[33]:
```
job_r = execute(qc_Grov, backend_device, shots=shots)
jobID_r = job_r.job_id()
print(f'JOB ID: {jobID_r}')
```

```
JOB ID: 60b8e4e50731dfe980bde1b3
```

[34]:
```
job_get=backend_device.retrieve_job("60b8e4e50731dfe980bde1b3")

result_r = job_get.result()
countsRunSingleGate = result_r.get_counts(qc_Grov)
```

[35]:
```
plot_histogram([counts_run, counts_sim, countsRunSingleGate ], \
               legend=[ 'run in real device 2 Gates', 'ideal','run in real␣
↪device 1 Gate'], \
               color=['#061727','#82cfff','#52ff26'])
```

[35]:



1.11 Mitigação de Erros de medida com IGNIS

Os dispositivos quânticos que estamos a utilizar têm ruído, o que influencia o funcionamento do nosso circuito quântico. A título de exmplo, podemos citar os já referidos T1 e T2.

Para mitigar o ruído em medições podemos utilizar calibrações próprias. O IGNIS providencia-nos as ferramentas necessárias para fazermos estas calibrações e produzir um filtro de mitigação para o ruído nas medições.

[36]:
```
from qiskit.ignis.mitigation.measurement import (complete_meas_cal,␣
↪tensored_meas_cal,
```

```
                                                    CompleteMeasFitter,␣
↪TensoredMeasFitter)
```

Começamos por gerar uma lista de circuitos de calibração de medições, um para cada estado possível no nosso circuito. Como estamos a utilizar 3 qubits, vamos precisar de 2^3 circuitos de calibração.

[37]:
```
qr = QuantumRegister(nBits)

meas_calibs, state_labels = complete_meas_cal(qubit_list=list(range(nBits)),␣
↪qr=qr, circlabel='mcal')
```

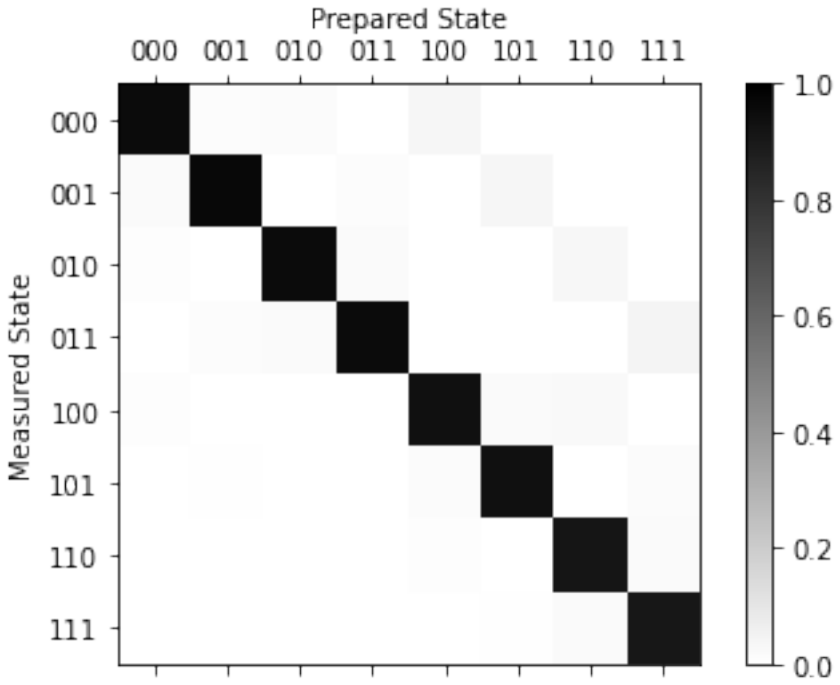
[38]:
```
job_ignis = execute(meas_calibs, backend=backend_device, shots=shots)
jobID_run_ignis = job_ignis.job_id()

print(f'JOB ID: {jobID_run_ignis}')
```

```
JOB ID: 60b8e5259acb6d80d2cc4c17
```

[39]:
```
job_get=backend_device.retrieve_job("60b8e5259acb6d80d2cc4c17")

cal_results = job_get.result()
```

omo estamos a correr o nosso circuito num dispositivo quântico real vai existir ruído, pelo que a matriz de calibração será diferente da identidade. Procedemos ao cálculo desta matriz:

[40]:
```
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')

# Plot the calibration matrix
meas_fitter.plot_calibration()
```



[41]:
```
print(f"Average Measurement Fidelity: {meas_fitter.readout_fidelity()*100: .
↪2f}%")
```

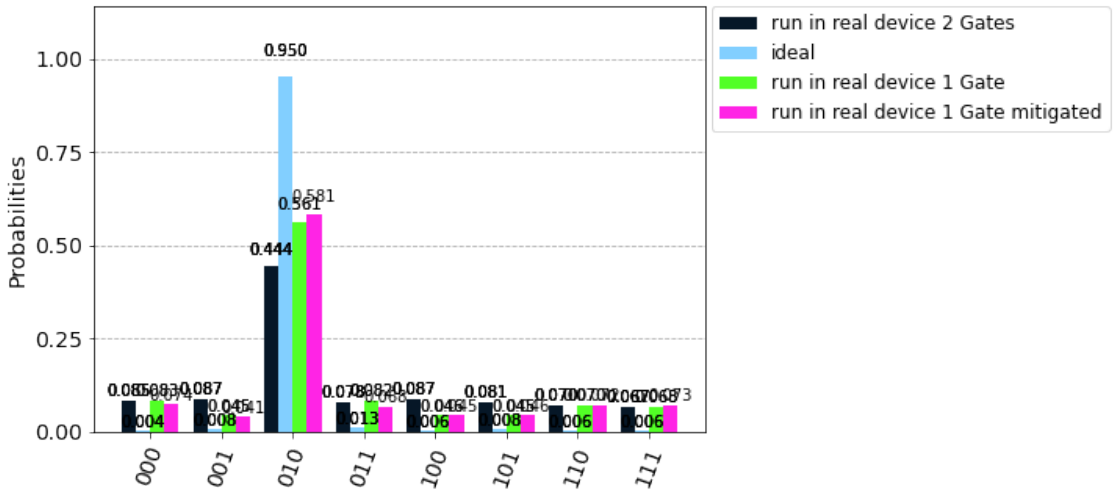
```
Average Measurement Fidelity:  94.10%
```

De seguida aplicamos o filtro de mitigação produzido aos resultados que obtivemos nos dispositivos quânticos:

[42]:
```
meas_filter = meas_fitter.filter

mitigated_results = meas_filter.apply(result_r)
mitigated_counts = mitigated_results.get_counts()
```

[43]:
```
plot_histogram([counts_run, counts_sim, countsRunSingleGate,mitigated_counts ],␣
↪\
               legend=[ 'run in real device 2 Gates', 'ideal','run in real␣
↪device 1 Gate','run in real device 1 Gate mitigated' ], \
               color=['#061727','#82cfff','#52ff26', '#ff24e5'])
```

[43]:



1.12 Demonstração da função geradora

A seguir, fazemos então apenas uma pequena demonstração da função de ordem superior desenvolvida.

Tomaremos um circuito de 8 QBits e procuraremos por um padrão escolhido aleatóriamente. Novamente recorreremos a função de ordem superior para retornar um gerador do algoritmo para circuitos de N QBits, e de seguida aplicaremos o padrão a ser procurado.

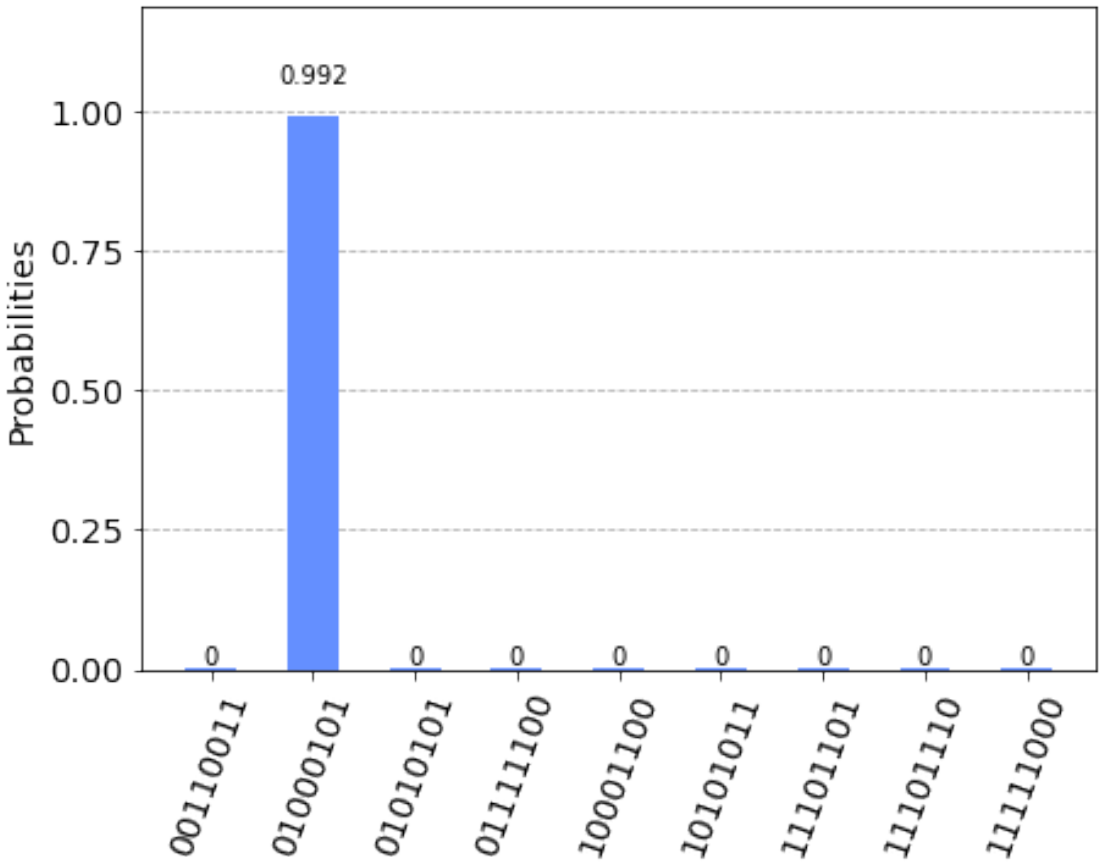
O número ideal de procuras foi apontado como 16, foi entretanto escolhida a aplicação de 11 iterações.

[37]:
```
nBits = 8

handle = oracleHandler(nBits)
qr = QuantumRegister(nBits,'qr')
cr = ClassicalRegister(nBits,'cr')
qrAux = QuantumRegister(nBits-3,'qrA')
qc_Grov = QuantumCircuit(qr,qrAux,cr)
qc_Grov.h(qr)

print("====================================")

handle(qc_Grov,qr,pattern= '01000101',auxQBits=qrAux,app=11)
qc_Grov.measure(qr,cr)

print("====================================")

backend = Aer.get_backend("qasm_simulator")
shots=1024
result = execute(qc_Grov, backend, shots=shots).result()
```

```
counts_sim = result.get_counts(qc_Grov)
print("higher:",max(counts_sim.keys(), key=lambda t: counts_sim[t]))
plot_histogram(counts_sim)
```

```
Generating Oracle Handler to circuit with 8 QBits…

Recomended 16 applications of the Gate…

5 auxiliar QBits required to execute circuit…

- returned:
  (circuit,QBits,pattern='11111111',app=16, auxQBits=None, phaseTests=False)

====================================
Grover Algorithm applied 11 times…
 To search for 01000101…

====================================
higher: 01000101
```

[37]:



A execução teórica foi um sucesso. A execução real entretanto torna-se inviável como descrito a seguir. O circuito é muito profundo, suficiente para que não seja possível a execução em um dispositivo real.

[9]:
```
from qiskit.dagcircuit import DAGCircuit
from qiskit.converters import circuit_to_dag
from qiskit.compiler import transpile

transpile_circuit = transpile(qc_Grov, basis_gates=['u3', 'cx'] )
dag_circuit = circuit_to_dag(transpile_circuit)
dag_circuit.count_ops_longest_path()
```

[9]:
```
{'u3': 991, 'cx': 1122, 'measure': 1}
```

[8]:
```
qc_Grov.depth()
```

[8]:
```
310
```

1.13 Exemplo

A seguir, entretanto, apresenta-se gráficamente um circuito resultante de uma única iteração das 11 acima requeridas.

[38]:
```
handle = oracleHandler(nBits)
qr = QuantumRegister(nBits,'qr')
cr = ClassicalRegister(nBits,'cr')
qrAux = QuantumRegister(nBits-3,'qrA')
qc_Grov = QuantumCircuit(qr,qrAux,cr)
qc_Grov.h(qr)

print("====================================")

handle(qc_Grov,qr,pattern= '01000101',auxQBits=qrAux,app=1)
qc_Grov.measure(qr,cr)

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

```
Generating Oracle Handler to circuit with 8 QBits…

Recomended 16 applications of the Gate…

5 auxiliar QBits required to execute circuit…

- returned:
  (circuit,QBits,pattern='11111111',app=16, auxQBits=None, phaseTests=False)

====================================
Grover Algorithm applied 1 times…
```

To search for 01000101…

[38]: