IC-Grupo13

June 5, 2021

1 Interação e Concorrência

1.1 Trabalho Prático - Grupo 13

Grupo: - André Morandi A86912 - Ivo Lima A90214

[107]:
```
# importing Qiskit
from qiskit import Aer, IBMQ
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute, transpile

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

import matplotlib.pyplot as plt
%matplotlib inline

# Import measurement calibration functions
from qiskit.ignis.mitigation.measurement import (complete_meas_cal,␣
↪tensored_meas_cal,
                                                  CompleteMeasFitter,␣
↪TensoredMeasFitter)
```

Sabendo que o número do nosso grupo é $N = 13$

Temos de de usar um *quantum algorithm* para encontrar s numa lista não ordenada, tal que

$$s = Nmod8$$

[108]:
```
N = 13
s = N % 8
s
```

[108]: 5

Passando o valor de s para binário, ficamos com:

[109]:
```
w = bin(s)[2:]
w # winner
```

[109]: '101'

Portanto, como 5 em binário é 101, iremos precisar de preparar um circuito quântico de 3 qubits

[110]:
```
x = 3
print('Número de qubits: ', x)
```

```
Número de qubits:  3
```

[111]:
```
qr_x       = QuantumRegister(x, 'x')
cr         = ClassicalRegister(x, 'cr')
qc_Grover = QuantumCircuit(qr_x,cr) # circuito quântico
```

O algoritmo adotado pelo nosso grupo foi o algoritmo de $Grover$. Este é um algoritmo de pesquisa dividido em três fases, nomeadamente $inicializao$, $orculo$ e $amplificao$ ($diffuser$).

Portanto, iremos inicializar todos os estados com a mesma amplitude, isto é, inicializar todos os qubits com uma *gate* de $Hadammar$.

$$\sum_{x_i} |x_i\rangle$$

Para tal criamos a seguinte função:

[112]:
```
def init(qc_Grover):
    qc_Grover.h(0)
    qc_Grover.h(1)
    qc_Grover.h(2)
```

1.1.1 Implementação do Oracle

Para computar um *quantum algorithm* baseado em um determinada função, podemos implementar uma espécie de *black box* da função. Passamos um *input* x e recebemos um *output* $f(x)$.

Para resolver os problemas, podemos definir o oráculo da seguinte forma: marcaremos nossa solução (ou soluções) com uma fase negativa (-1). Desta forma, podemos usar o $Grover's\ algorithm$ para resolver.

$$U_w|x\rangle \Rightarrow x \neq w \rightarrow |x\rangle$$
$$U_w|x\rangle \Rightarrow x = w \rightarrow -|x\rangle$$

Tomando o valor w como sendo 101 resultará na seguinte matriz:

$$U_w = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

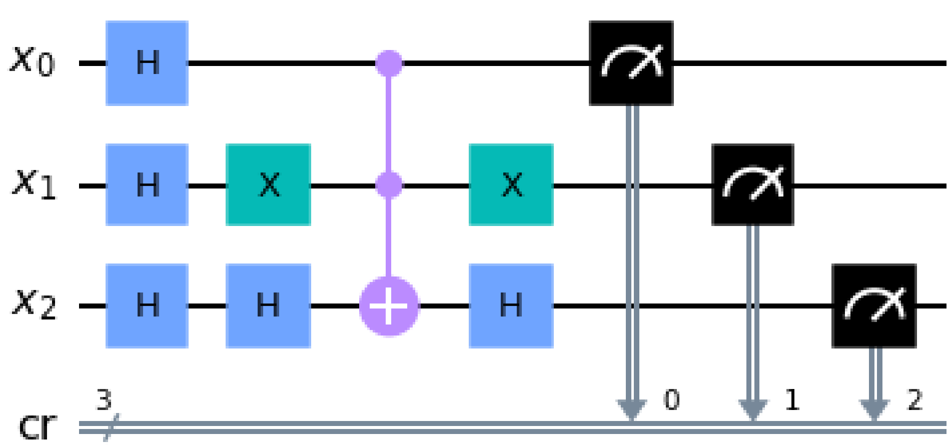
[113]:
```
def phase_oracle(qc_Grover, qr_x):
    qc_Grover.x(qr_x[1])
    qc_Grover.h(qr_x[2])
    qc_Grover.ccx(qr_x[0], qr_x[1], qr_x[2])
    qc_Grover.x(qr_x[1])
    qc_Grover.h(qr_x[2])
```

Se neste momento medíssemos a base $|x>$, a superposição colapsaria, de acordo com o que nos foi explicado nas aulas ($Schrdinger's\ cat$), teríamos em cada uma das bases uma probabilidade de $\frac{1}{N} = \frac{1}{2^n}$ e as nossas chances de encontrar o valor w, *priori*, estaria entre 1 e 2^n.

Aplica-se a reflexão do oráculo ao estado s. Esta transformação significa que a amplitude média à frente do estado de w tornar-se-á negativa (foi diminuída).

[114]:
```
init(qc_Grover)

phase_oracle(qc_Grover, qr_x)

qc_Grover.measure(qr_x,cr)

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

[114]:



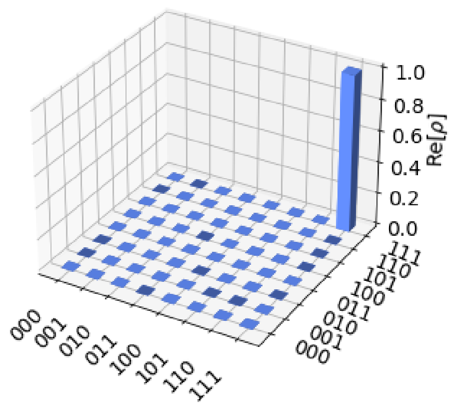
[115]:
```python
backend_state = Aer.get_backend('statevector_simulator') # the device to run on
```

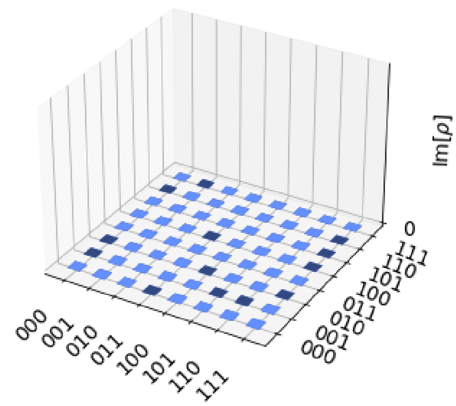
[116]:
```python
result = execute(qc_Grover, backend_state).result()
psi1  = result.get_statevector(qc_Grover)

print(psi1)
```

```
[ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j -0.+0.j  0.+0.j  1.+0.j]
```

[117]:
```python
plot_state_city(psi1)
```

[117]:





1.1.2 Diffuser

O computador quântico utiliza a amplificação para que possa aumentar a probabilidade. Este processo amplifica a amplitude do w diminuindo a dos outros. Isto faz com que a amplitude de w se destaque relativamente às outras, tornando a probabilidade de escolher-se o estado w muito maior.

[118]:
```python
def diffuser(qc_Grover,qr_x):
    qc_Grover.h(qr_x[0])
    qc_Grover.x(qr_x[0])
    qc_Grover.h(qr_x[1])
    qc_Grover.h(qr_x[2])
    qc_Grover.x(qr_x[1])
    qc_Grover.x(qr_x[2])
    qc_Grover.h(qr_x[2])
    qc_Grover.ccx(qr_x[0], qr_x[1], qr_x[2])
    qc_Grover.x(qr_x[0])
    qc_Grover.x(qr_x[1])
    qc_Grover.h(qr_x[2])
```

```
    qc_Grover.h(qr_x[0])
    qc_Grover.h(qr_x[1])
    qc_Grover.x(qr_x[2])
    qc_Grover.h(qr_x[2])
```

A fase do oráculo e *diffuser* terá de ser repetido aproximadamente $\sqrt{N}$ vezes para conseguirmos uma boa medição.

[119]:
```
import math as math

times= round(math.sqrt(2**x))
print(times)
```

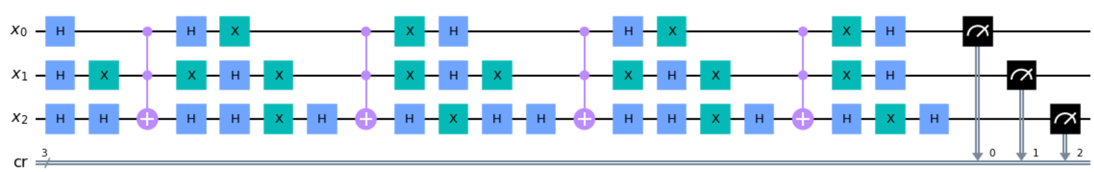
3

1.1.3 Implementação Completa em Qiskit

O *qc_Grover* vai inicializar o quantum circuit

[120]:
```
cr        = ClassicalRegister(x,'cr')
qc_Grover = QuantumCircuit(qr_x,cr)

init(qc_Grover)

for t in range(2):
    # phase oracle
    phase_oracle(qc_Grover, qr_x)
    # diffuser
    diffuser(qc_Grover,qr_x)

qc_Grover.measure(qr_x,cr)

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

[120]:



[121]:
```
backend_state = Aer.get_backend('statevector_simulator') # the device to run on
```

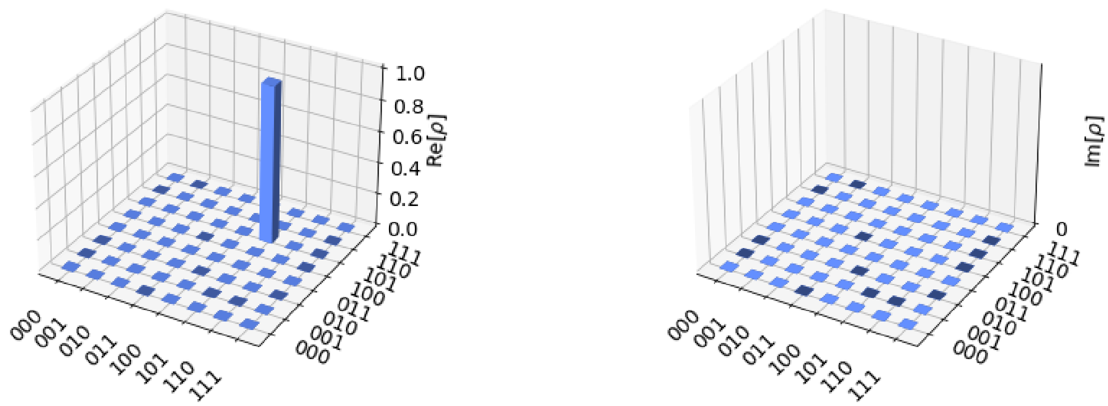
[122]:
```
result = execute(qc_Grover, backend_state).result()
psi2 = result.get_statevector(qc_Grover)
```

[123]:
```
print(psi2.real)
```

```
[-0. -0. -0. -0.  0.  1.  0. -0.]
```

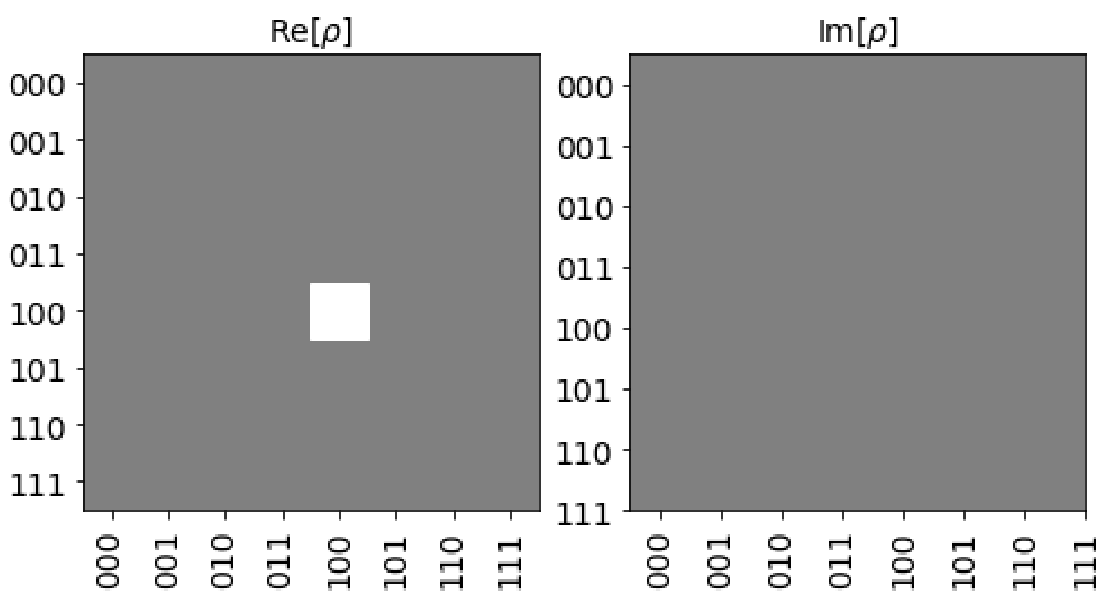
[124]:
```
plot_state_city(psi2)
```

[124]:



[125]:
```
plot_state_hinton(psi)
```

[125]:



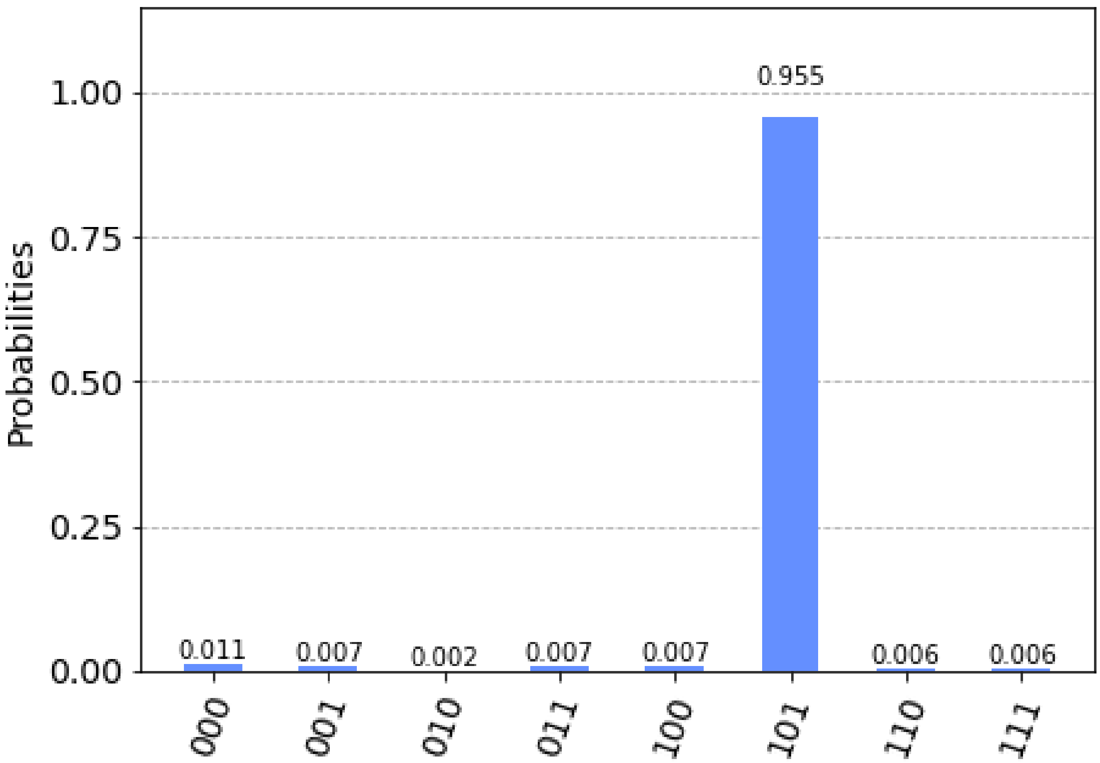
Agora iremos correr o circuito num simulador.

[126]:
```
backend = Aer.get_backend("qasm_simulator")
```

[127]:
```
shots=1024
result = execute(qc_Grover, backend, shots=shots).result()
counts_sim = result.get_counts(qc_Grover)
```

```
plot_histogram(counts_sim)
```

[127]:



Era espectável que após a execução de várias repetições tanto do oráculo como do difusor, a probabilidade de se escolher o estado w fosse de 100% (numa situação ideal), uma vez que a amplitude continuaria sempre a subir. Porém, é normal que nunca se atinja esta percentagem, uma vez que os outros estados também têm amplitude, possuindo uma pequena fatia deste 100%.

[128]:
```
qc_Grover.depth()
```

[128]: 22

Agora iremos testar o circuito numa máquina quantum de verdade

1.1.4 Correr num Quantum Computer (Noise Simulator)

[17]:
```
provider = IBMQ.load_account()
provider.backends()
```

[17]:
```
[<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')>]
```

[18]:
```
# Backend overview
import qiskit.tools.jupyter

%qiskit_backend_overview
```

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

[19]:
```
from qiskit.tools.monitor import backend_overview, backend_monitor

backend_overview()
```

```
ibmq_manila                 ibmq_quito                  ibmq_belem
-----------                 ----------                  ----------
Num. Qubits:  5             Num. Qubits:  5             Num. Qubits:  5
Pending Jobs: 0             Pending Jobs: 5             Pending Jobs: 0
Least busy:   True          Least busy:   False         Least busy:   False
Operational:  True          Operational:  True          Operational:  True
Avg. T1:      145.5         Avg. T1:      75.2          Avg. T1:      79.3
Avg. T2:      67.0          Avg. T2:      73.2          Avg. T2:      91.6



ibmq_lima                   ibmq_santiago               ibmq_athens
---------                   -------------               -----------
```

```
Num. Qubits:   5            Num. Qubits:   5            Num. Qubits:   5
Pending Jobs:  0            Pending Jobs:  6            Pending Jobs:  0
Least busy:    False        Least busy:    False        Least busy:    False
Operational:   True         Operational:   True         Operational:   True
Avg. T1:       69.2         Avg. T1:       146.5        Avg. T1:       85.2
Avg. T2:       64.9         Avg. T2:       136.4        Avg. T2:       120.6



ibmq_armonk                 ibmq_16_melbourne           ibmqx2
-----------                 -----------------           ------
Num. Qubits:   1            Num. Qubits:   15           Num. Qubits:   5
Pending Jobs:  28           Pending Jobs:  2            Pending Jobs:  0
Least busy:    False        Least busy:    False        Least busy:    False
Operational:   True         Operational:   True         Operational:   True
Avg. T1:       124.6        Avg. T1:       57.5         Avg. T1:       54.1
Avg. T2:       217.3        Avg. T2:       56.2         Avg. T2:       40.5
```

Escolhemos a *ibmq_santiago* devido ao Avgerage $T1$ *relaxation time* e $T2$ *coherence time* e também por causa da quantidade de qubits superior ou igual a 3.

[20]:
```
backend_device = provider.get_backend('ibmq_santiago')
print("Running on: ", backend_device)
```

```
Running on:  ibmq_santiago
```

[21]:
```
# See backend information
backend_device
```

```
VBox(children=(HTML(value="<h1 style='color:#ffffff;background-color:#000000;
↪padding-top: 1%;padding-bottom: 1…
```

[21]:
```
<IBMQBackend('ibmq_santiago') from IBMQ(hub='ibm-q', group='open',
project='main')>
```

[22]:
```
backend_monitor(backend_device)
```

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

```
    local: False
    simulator: False
    n_uchannels: 8
    supported_instructions: ['shiftf', 'measure', 'play', 'setf', 'rz', 'cx',
'u3', 'acquire', 'delay', 'id', 'reset', 'sx', 'u1', 'x', 'u2']
    coupling_map: [[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2], [3, 4], [4,
3]]
    memory: True
    description: 5 qubit device
    qubit_channel_mapping: [['u0', 'm0', 'd0', 'u1'], ['m1', 'u3', 'u0', 'd1',
'u1', 'u2'], ['u4', 'm2', 'u3', 'd2', 'u5', 'u2'], ['u4', 'u6', 'm3', 'u7',
'u5', 'd3'], ['u7', 'm4', 'd4', 'u6']]
    rep_times: [0.001]
    url: None
    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': 1.011137872642343, 'omegad1': 0.9860187056541215, 'omegad2': 1.0018026333654275, 'omegad3': 1.0073346201781475, 'omegad4': 1.0008689448135097, 'wq0': 30.369326284658154, 'wq1': 29.051000320192983, 'wq2': 30.288289457980554, 'wq3': 29.796805745616194, 'wq4': 30.261843869801826}}
    n_registers: 1
    allow_q_object: True
    dtm: 0.2222222222222222
    credits_required: True
    qubit_lo_range: [[4.33342839657397e+18, 5.333428396573969e+18], [4.1236103027229476e+18, 5.123610302722947e+18], [4.3205309850357484e+18, 5.320530985035748e+18], [4.242308922763805e+18, 5.242308922763806e+18], [4.316322038954131e+18, 5.316322038954131e+18]]
    allow_object_storage: True
    sample_name: family: Falcon, revision: 4, segment: L
    meas_levels: [1, 2]
    pulse_num_channels: 9
    meas_kernels: ['hw_qmfk']
    input_allowed: ['job']
    meas_map: [[0, 1, 2, 3, 4]]
    parametric_pulses: ['gaussian', 'gaussian_square', 'drag', 'constant']
    dynamic_reprate_enabled: True
    online_date: 2020-06-03 04:00:00+00:00
    dt: 0.2222222222222222
    discriminators: ['linear_discriminator', 'quadratic_discriminator', 'hw_qmfk']
    processor_type: {'family': 'Falcon', 'revision': 4, 'segment': 'L'}
    multi_meas_enabled: True
    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'}}
    pulse_num_qubits: 3
    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)}]]
    conditional_latency: []
    acquisition_latency: []
    quantum_volume: 32
    default_rep_delay: 250.0
    open_pulse: False
    rep_delay_range: [0.0, 500.0]
    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]]
    max_shots: 8192
    uchannels_enabled: True
    conditional: False
    max_experiments: 75

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

    Q0 / 4.83343 GHz / 162.34929 us / 240.32302 us / 0.00000 / 0.00027 / 0.00027 / 0.01770
    Q1 / 4.62361 GHz / 131.33661 us / 108.48683 us / 0.00000 / 0.00016 / 0.00016 / 0.00970
    Q2 / 4.82053 GHz / 140.37440 us / 97.48492 us / 0.00000 / 0.00022 / 0.00022 / 0.01010
    Q3 / 4.74231 GHz / 182.22586 us / 98.64605 us / 0.00000 / 0.00018 / 0.00018 / 0.00480
    Q4 / 4.81632 GHz / 115.98140 us / 136.96134 us / 0.00000 / 0.00044 / 0.00044 / 0.01720

Multi-Qubit Gates [Name / Type / Gate Error]
----
    cx4_3 / cx / 0.00610
    cx3_4 / cx / 0.00610
    cx2_3 / cx / 0.00567
    cx3_2 / cx / 0.00567
    cx2_1 / cx / 0.00592

```
cx1_2 / cx / 0.00592
cx0_1 / cx / 0.00610
cx1_0 / cx / 0.00610
```

[23]:
```
%qiskit_job_watcher
```

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

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

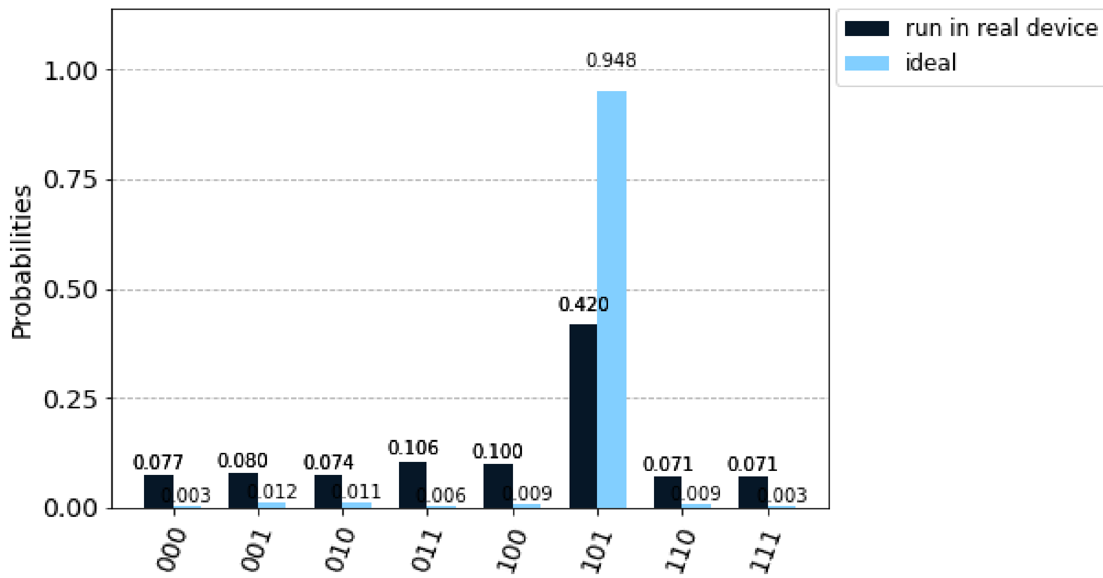
[24]:
```
job_r = execute(qc_Grover, backend_device, shots=shots)

jobID_r = job_r.job_id()

print('JOB ID: {}'.format(jobID_r))
```

```
JOB ID: 60bb51b55f4eaa46e7dae995
```

[26]:
```
job_get=backend_device.retrieve_job("60bb51b55f4eaa46e7dae995")

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

[27]:
```
plot_histogram([counts_run, counts_sim ], legend=[ 'run in real device',␣
↪'ideal'], color=['#061727','#82cfff'])
```

[27]:



Desta forma, concluímos que há uma maior chance de medir |101>. Os outros resultados ocorrem devido aos erros da computação quântica.

1.1.5 IGNIS

É uma calibração usada para diminuir os erros de medição.

1.1.6 Calibration Matrix

Como temos 3 qubits, precisamos de um circuito de calibração da ordem $2^3 = 8$

[28]:
```
# Generate the calibration circuits
qr = QuantumRegister(x)
meas_calibs, state_labels = complete_meas_cal(qubit_list=[0,1,2], qr=qr,␣
↪circlabel='mcal')
```

[29]:
```
state_labels
```

[29]:
```
['000', '001', '010', '011', '100', '101', '110', '111']
```

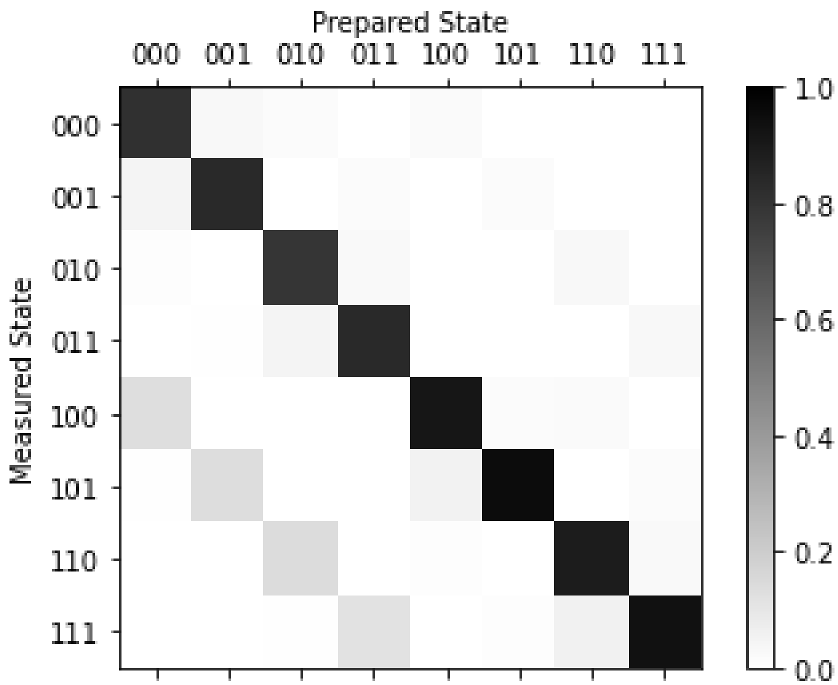
Num caso idealista onde não existiria barulho/erro, a matriz de calibração seria uma matriz identidade $8x8$. Mas, uma vez que estamos a aplicar num dispositvo quântico real, haverá sempre algum barulho/erro.

[30]:
```
job_ignis = execute(meas_calibs, backend=backend_device, shots=shots)

jobID_run_ignis = job_ignis.job_id()

print('JOB ID: {}'.format(jobID_run_ignis))
```

```
JOB ID: 60bb51da1eb02401eacee63d
```

[31]:
```
job_get=backend_device.retrieve_job("60b7883fdd5b829f163c1415")

cal_results = job_get.result()
```

[32]:
```
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')

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



1.1.7 Análise de Resultados

A *average assignment fidelity* é o traço da diagonal da matriz anterior.

[33]:
```
# Medida de fidelidade
print("Average Measurement Fidelity: %f" % meas_fitter.readout_fidelity())
```

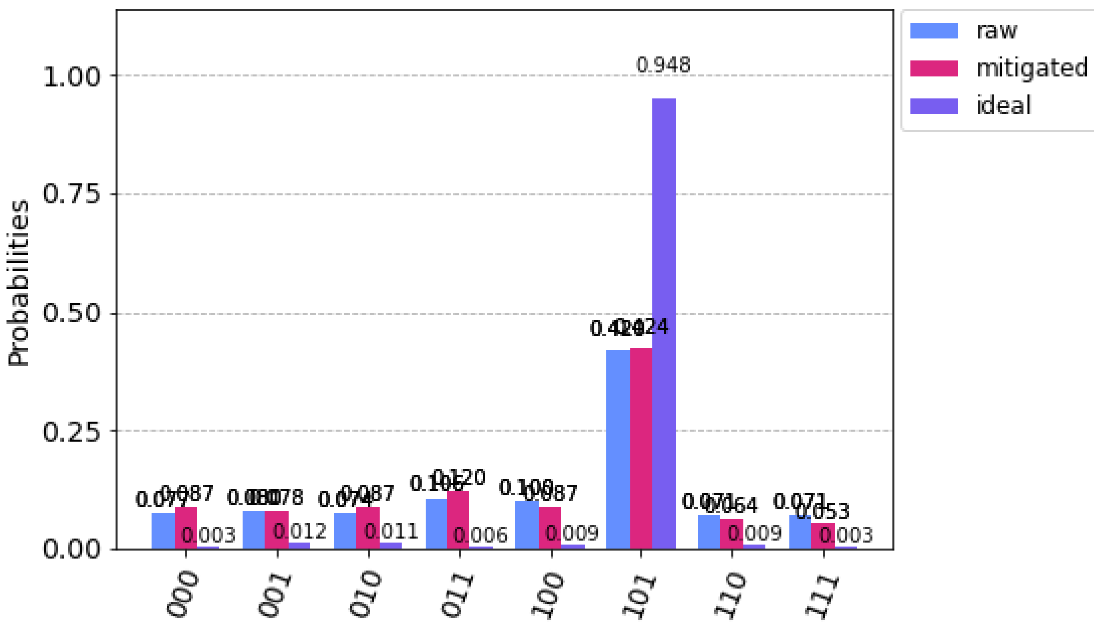
```
Average Measurement Fidelity: 0.868042
```

1.1.8 Calibração

[34]:
```
# Filtro
meas_filter = meas_fitter.filter

# Resultados (mitigation)
mitigated_results = meas_filter.apply(result_r)
mitigated_counts = mitigated_results.get_counts()
```

[35]:
```
plot_histogram([counts_run, mitigated_counts, counts_sim], legend=['raw',␣
↪'mitigated', 'ideal'])
```

[35]:



1.1.9 Conclusão

O algoritmo de Grover é relativamente simples, umas vez que a inserção das primeiras *Hadamard gates* colocam os qubits numa situação em que o estado das suas fases tem importância. O oráculo muda-as, já o difusor reorganiza-as para que mais tarde possam ser aplicadas novamente as *Hadamard gates* para assim obter o w esperado. Portanto podemos concluir que somente o oráculo é alterado e o *diffuser* mantém-se inalterado.

Além disso, a mitigação de erros foi capaz de aumentar ligeiramente a probabilidade de ocorrência no nosso estado marcado.

1.1.10 BIBLIOGRAFIA

Para a elaboração deste trabalho, consultamos as seguintes páginas da web, para o esclarecimento de dúvidas:

- Practical Guide
- Qiskit Documentation
- IBM Composer
- Grover's Algorithm
- Interação e Concorrência - Página da Disciplina

```
[ ]:
```