University of Basel
Quantum Computation by Dr. James Wootton

The quantum walk and its search algorithm

Marcel Köberlin
June 2022

Contents

1 The 1D classical random walk **2**

2 The 1D Quantum Walk on $\mathbb{Z}$ **3**

3 Quantum walk search algorithm **13**

4 Remark **23**

5 Conclusion **23**

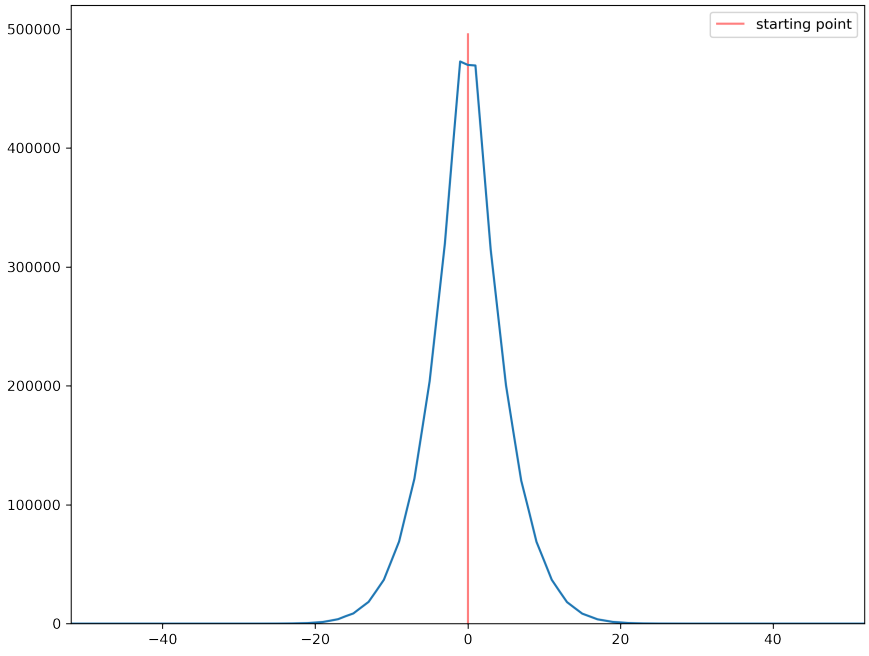
6 Sources **23**

1 The 1D classical random walk

For the classical random walk, we flip a coin and either walk right or left, depending on the outcome. This can easily be implemented in python with just loops and the random function:

```
def classicwalk(steps,iterations):

    result={} #Create an empty dictionary that will hold the results
    for number in range(-iterations,iterations+1): #initialize the
   dictionary by setting everything to 0
        result[number]=0
    for number in range(0,iterations):
        start=0
        for step in range(0,steps):
            if round(np.random.random())==1:
                start=start+1
            else:
                start=start-1
            result[start]+=1
    return result
```

In this example we do 100000 iterations with 50 steps each. The outcome is (close to a) gaussian distribution:



In other words, the most probable outcome is ending at our starting position 0 or anywhere close to this. This is due to the fact that each combination of steps is equally likely, but for positions close to the start, there are more combinations.

2 The 1D Quantum Walk on $\mathbb{Z}$

We consider the discrete time quantum walk on the infinite integer line $\mathbb{Z}$. We denote the position of the walker's position with $|j\rangle$ where $j \in \mathbb{Z}$. The walker will now only move on the nodes of this graph, whereas the word "node" and "state" will be used interchangeably. A coin will decide the movement of the walker, the basis is chosen to be $\{|0\rangle, |1\rangle\}$.
In this model, two states and two operators are required: The position state representing the position and the coin-state that decides the movement. The space that we are working with is $\mathcal{H} = \mathcal{H}_C \otimes \mathcal{H}_\mathcal{P}$ where $\mathcal{H}_C$ refers to the Hilbert space of the coin and may be written as

$$\mathcal{H}_C = \{c_0 |0\rangle_C + c_1 |1\rangle_C : c_0, c_1 \in \mathbb{C}\}$$

whereas the position Hilbert space is

$$\mathcal{H}_P = \left\{ \sum_{x\in\mathbb{Z}} \alpha_x |x\rangle_P : \sum_{x\in\mathbb{Z}} |\alpha_x| < \infty \right\}$$

The two operators needed are the coin operator $\mathcal{C}$ and the shifting operator $\mathcal{S}$ that actually moves the walker. A common choice for the coin operator is the Hadamard gate

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

that puts the coin state in an equal superposition:

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \qquad \text{and} \qquad H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

Measuring the coin state at this point would yield $|0\rangle$ and $|1\rangle$ with equal probability.

Another possibility is the Grover diffusion operator known from Grover's algorithm:

$$G = \begin{pmatrix} \frac{2}{n}-1 & \frac{2}{n} & \cdots & \frac{2}{n} & \\ \frac{2}{n} & \frac{2}{n}-1 & \cdots & \frac{2}{n} & \\ \vdots & & \cdots & & \ddots & \vdots \\ \frac{2}{n} & \frac{2}{n} & \cdots & \frac{2}{n}-1 & \end{pmatrix}$$

While it also puts the walker into a superposition, it is not an equal superposition. To test this for three qubits, initially in the state $|000\rangle$, we must find a way to write G in terms of gates (this was done in exercise 9):

$$\boxed{G = H^{\otimes 3} \cdot X^{\otimes 3} \cdot \mathrm{CCZ} \cdot X^{\otimes 3} \cdot H^{\otimes 3}}$$

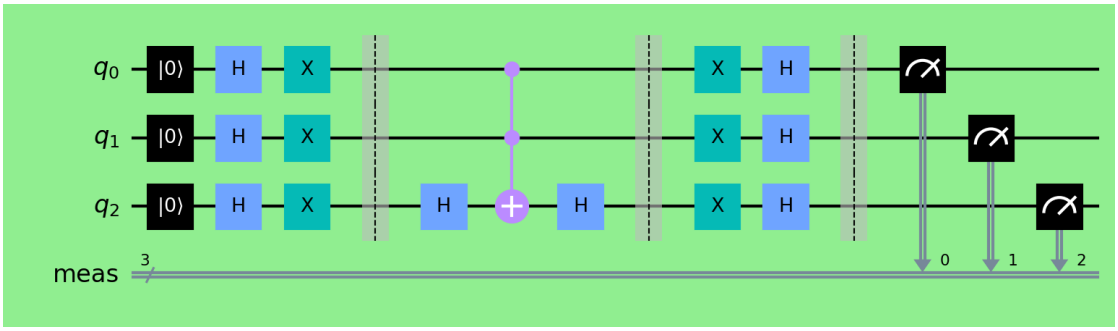
We can rewrite the CCZ gate in terms of the CCX gate and two hadamards:

$$\mathrm{CCZ} = (\mathbb{1} \otimes \mathbb{1} \otimes H) \cdot \mathrm{CCX} \cdot (\mathbb{1} \otimes \mathbb{1} \otimes H)$$

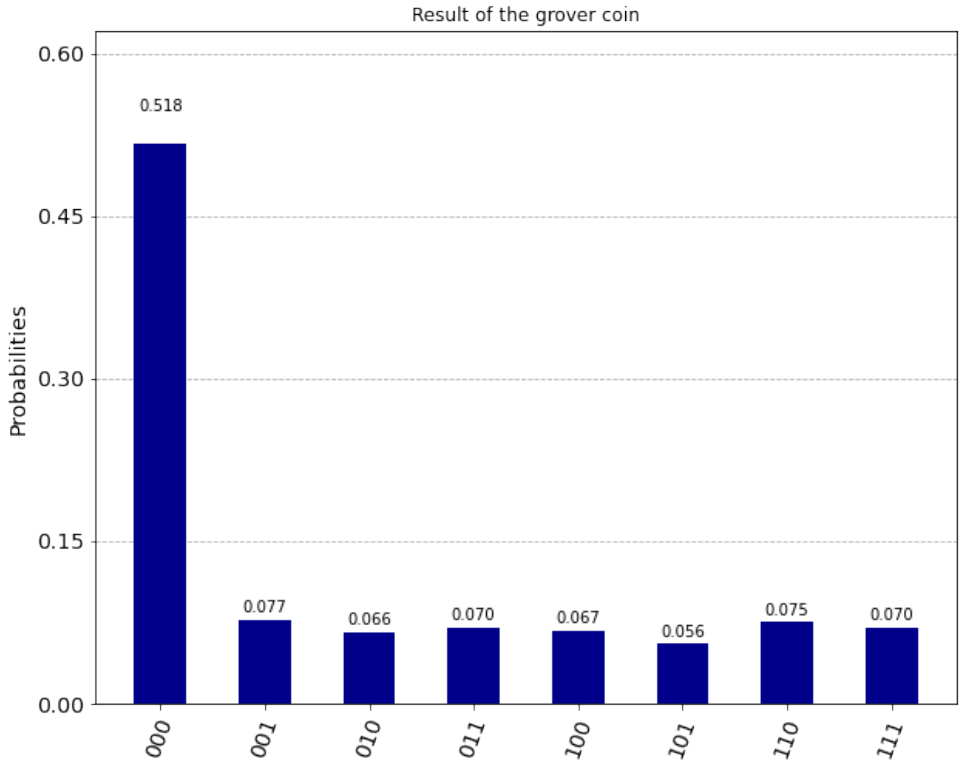
which works as $Z = HXH$. With Qiskit we can then build and test the circuit:

```
qc = QuantumCircuit(3)
qc.reset([0,1,2]) #Make sure all qubits are in the 0 state
qc.h([0,1,2]) #Apply Hadamard to all qubits
qc.x([0,1,2]) #Apply X to all qubits

qc.h(2) #Creating the CCZ gate with a toffoli gate
qc.toffoli(0,1,2)#Creating the CCZ gate with a toffoli gate
qc.h(2)#Creating the CCZ gate with a toffoli gate

qc.x([0,1,2]) #Apply X to all qubits
qc.h([0,1,2]) #Apply Hadamard to all qubits

qc.measure_all() #Measure all qubits
aer_sim = Aer.get_backend('aer_simulator') #Simulate the circuit
qobj = assemble(qc)
result = aer_sim.run(qobj).result()
counts = result.get_counts()
```

The circuit build looks like this:



The Hadamard and the CCX gate between the two barriers make up the CCZ gate. The results after measurement are seen in the next figure:



It can be seen that the $|000\rangle$ state has the highest probability, while all other state are almost equally likely. This is fundamentally differently from the Hadamard.

We now have two possible operators for the coin but what we are still missing is the shift operator. We choose the convention that the coin state $|0\rangle_c$ moves the walker to the right side, while the coin state $|1\rangle_c$ moves it to the left. We can express this behaviour as

$$\mathcal{S}\left[|0\rangle_C \otimes |j\rangle_P\right] = |0\rangle_C \otimes |j+1\rangle_P$$

and

$$\mathcal{S}\left[|1\rangle_C \otimes |j\rangle_P\right] = |0\rangle_C \otimes |j-1\rangle_P$$

The indices C and P will be dropped from now on.

The shift operator can be realized by

$$\mathcal{S} = |0\rangle\langle 0| \otimes \sum_j |j+1\rangle\langle j| + |1\rangle\langle 1f| \otimes \sum_j |j-1\rangle\langle j|$$

With our two states and two operators, we may define a new, *unitary* operator

$$\mathcal{U} = \mathcal{SC}$$

This unitary operator allows us to find the state $|\Psi(t)\rangle = \mathcal{U}^t |\Psi(0)\rangle$.

For example the initial state $|\Psi(0)\rangle = |0\rangle \otimes |0\rangle$ and the Hadamard coin H coin yields:

$$|0\rangle \otimes |0\rangle \xrightarrow{H} \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle \xrightarrow{\mathcal{S}} \frac{1}{\sqrt{2}}(|0\rangle \otimes |1\rangle + |1\rangle \otimes |-1\rangle)$$

In order words, if the walker starts at position $|0\rangle$ and a "fair" Hadamard coin is used for the initial coin state $|0\rangle$, the walker will be either in position $|-1\rangle$ or $|1\rangle$ with equal probability after applying the unitary once. Applying this unitary multiple times, one could expect the classical result, but as it turns out, the quantum case yields wildy different results.

Let us now find a way to implement this unitary in Qiskit. The idea of the unitary is quite simple:

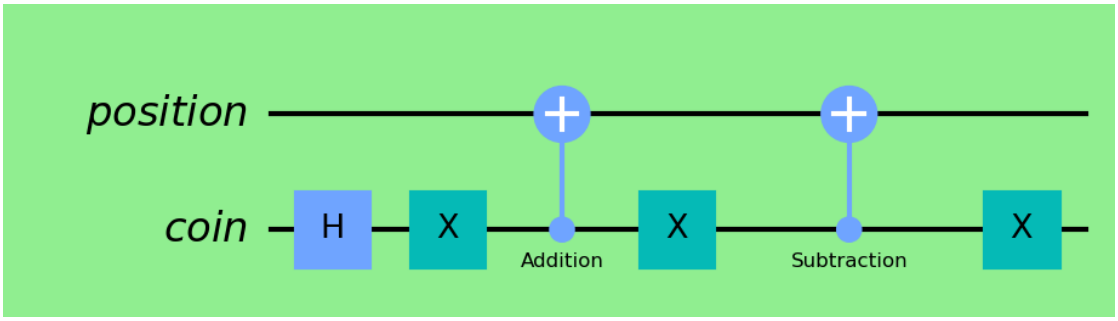


Figure 1: The idea of the unitary operator where the controlled gates represent the addition and subtraction operator, respectively.

The upper "lane" will be our position state with N qubits, thus allowing us to depict numbers from 0 to $2^N - 1$. The lower lane will be our "coin lane". The two controlled gates are the addition and subtraction operators. We see that a Hadamard is first applied to our coin qubit and thus putting it into a superposition. If the coin is in state $|0\rangle$ (after applying the Hadamard), the (controlled) addition operator will be applied and the subtraction operator will do nothing. If the coin is $|1\rangle$ however, the subtraction operator will be applied instead.

Since the addition operator $\mathcal{A}$ is unitary

$$\mathcal{A}^\dagger \mathcal{A} |j\rangle = \mathcal{A}^\dagger |j+1\rangle = |j\rangle$$

or equivalently, if writing the subtraction operator as $\mathcal{K}$

$$\mathcal{A}^\dagger = \mathcal{K},$$

we can write $\mathcal{A}$ as a product of unitaries U_i:

$$\mathcal{A} = U_1 U_2 ... U_n \qquad \Longleftrightarrow \qquad \mathcal{K} = \mathcal{A}^\dagger = U_n^\dagger ... U_2^\dagger U_1^\dagger = U_n ... U_2 U_1 \tag{1}$$

The last step is allowed if and only if the addition operator only uses hermitian gates.

Chosen convention: The highest qubit q_0 will be the first binary value.
So $q_0 q_1 q_2$=100 (bin) corresponds to 4 (dec).

Lets start with the addition operator. All we have to implement is binary addition:

```
def addition(circuit,N):
    number=N-1
    while number>=0:
        circuit.append(MCXGate(N-number),list(range(N,number-1,-1)))
    #MCXGate allows us to create multicontrolled gates
        if number!=0:
            circuit.x(number)
        number=number-1
```

As we can see, only X and (multi-controlled) CX gates are used, those are hermitian and thus the last step in (1) is allowed. From formula (1) we can easily find subtraction by just doing the reversed order of operations:

```
def subtraction(circuit,N):
    number=0
    while number<=N-1:
        if number!=0:
            circuit.x(number)
        circuit.append(MCXGate(N-number),list(range(N,number-1,-1)))
    #MCXGate allows us to create multicontrolled gates
        number=number+1
```

The following figures show the addition and subtraction operators implemented in Qiskit:

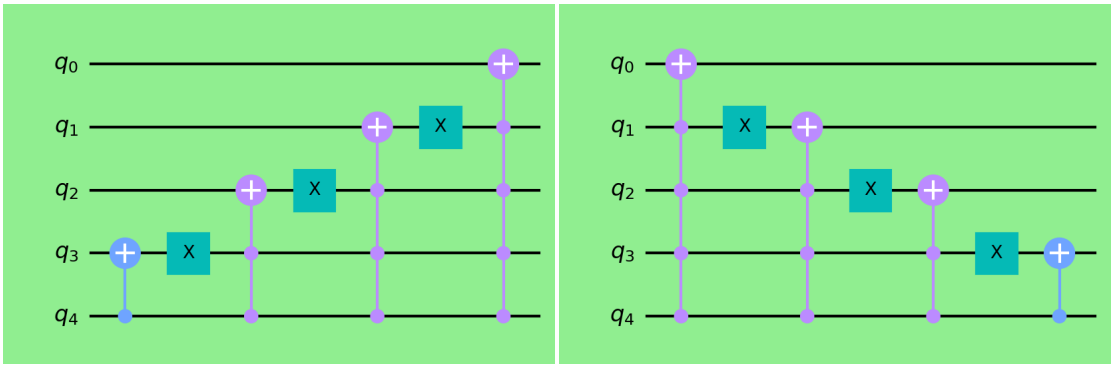


Figure 2: The addition operator (left) and the subtraction operator (right side) in Qískit.

Keeping figure (1) in mind, the unitary is easy to code:

```
def unitary(circuit,N):

    circuit.h(N) #Apply a Hadamard to the "coin-qubit"
    circuit.x(N) #Apply an X gate to the "coin-qubit"

    addition(circuit,N) #add the addition operator

    circuit.x(N) #Apply an X gate to the "coin-qubit"

    subtraction(circuit,N) #add the subtraction operator
```

We have N "position qubits" and are thus able to write all numbers up to $2^N - 1$. Since we want to start roughly in the middle of the binary number $11111...$ (N bits), we start at $100000...$ (N bits), by applying an X gate to the first qubit:

```
def initial(circuit,N):#Sets the initial state of the qubits

    circuit.reset(list(range(0,N+1))) #Make sure all qubits are set
    to |0>
    circuit.x(0) #This is roughly the half of 2^N-1
```

Keep in mind that we start in the state $|0\rangle\,|1000000...\rangle$, so with the coin in the $|0\rangle$ state. Having the unitary and the initial state, all that must be done is applying the unitary multiple times in a row to make the walker move.

For this, first a circuit with $(N+1)$ qubits must be initialized, then the *initial* function must be applied and lastly the *unitary* multiple times:

```
def walk(N):
    steps = 2**(N-1)
    circuit=QuantumCircuit(N+1) #Create a new empty circuit

    initial(circuit,N) #Input the initial state
    for times in range(0,steps): #Apply the unitary operator SC (
   steps)-times

        unitary(circuit,N)

    circuit.measure_all() #Measure them all

    backend=Aer.get_backend('aer_simulator')
    job = execute(circuit, backend, shots=30000) #Do 30'000 shots
    result=job.result() #Gather the results
    count=result.get_counts() #Get information of results

    return result
```

(Actually the original code is longer, because the coin-qubit is measured too, so we must "clean up" the bitstring and then convert it to decimal.)

To summarize: The N-bit position can depict numbers up to $2^N - 1$, so we start walking at $\approx 2^{N-1}$ and thus we only do a maximum of 2^{N-1} steps.

Before looking at the results, let us consider the first few iterations of the state $|\Psi(0)\rangle = |0\rangle\,|0\rangle$ by hand:

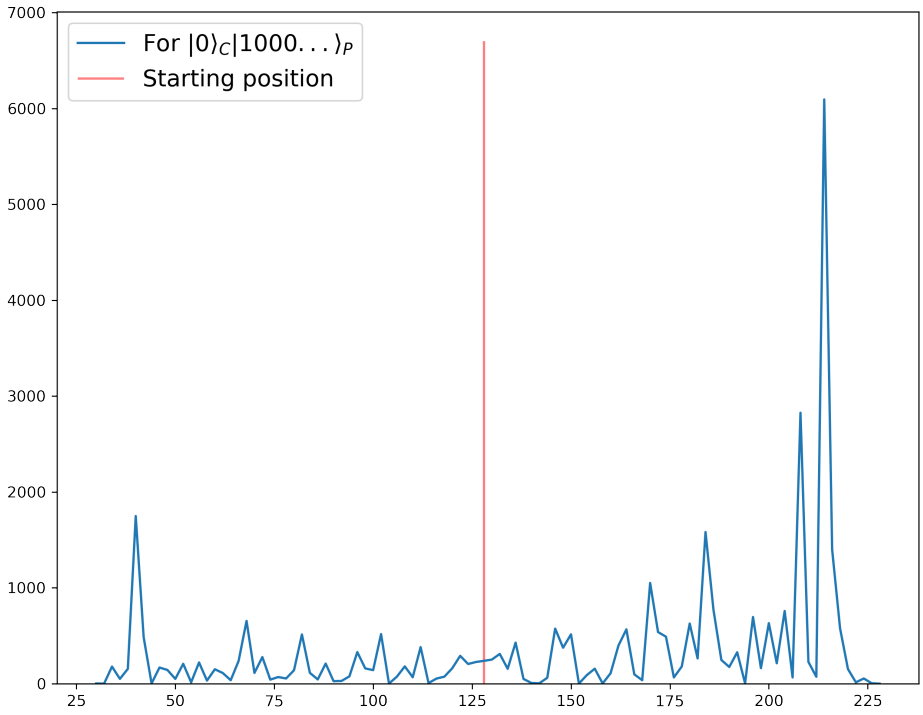
$$|\Psi(1)\rangle \propto (|0\rangle\,|1\rangle + |1\rangle\,|-1\rangle)$$

$$|\Psi(2)\rangle \propto (|0\rangle\,|2\rangle + |1\rangle\,|0\rangle + |0\rangle\,|0\rangle - |1\rangle\,|-2\rangle)$$

$$|\Psi(3)\rangle \propto (|0\rangle\,|3\rangle + |1\rangle\,|1\rangle + 2\cdot|0\rangle\,|1\rangle + |0\rangle\,|-1\rangle + |1\rangle\,|-3\rangle)$$

While after the second step, the probability is symmetrically distributed, the third step already shows a bias to the right side. The probability to land on $|1\rangle$ is three times as high as landing on $|-1\rangle$. Let us check whether that holds for more steps.

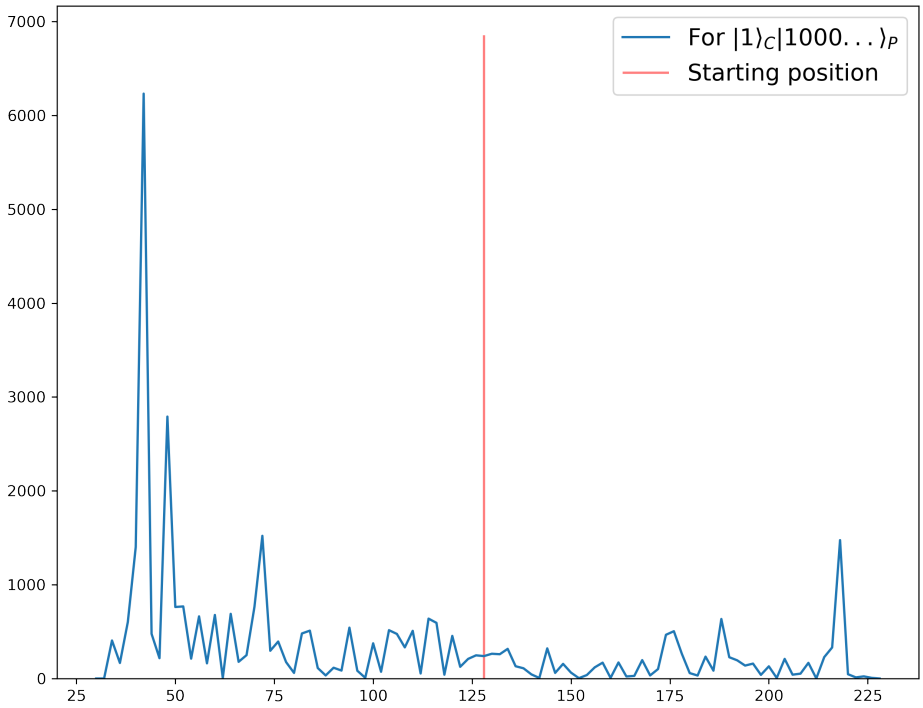
After doing 30000 shots for the coin state $|0\rangle$, we find a staggering result:



The resulting graph is not what one would expect: Instead of being symmetric and having its peak at starting position, the peak is shifted to the right with a small peak on the left and the graph is not symmetric at all! Is our coin not fair?

The reason for this behaviour is interference of different states that cause the graph to bias to the right.

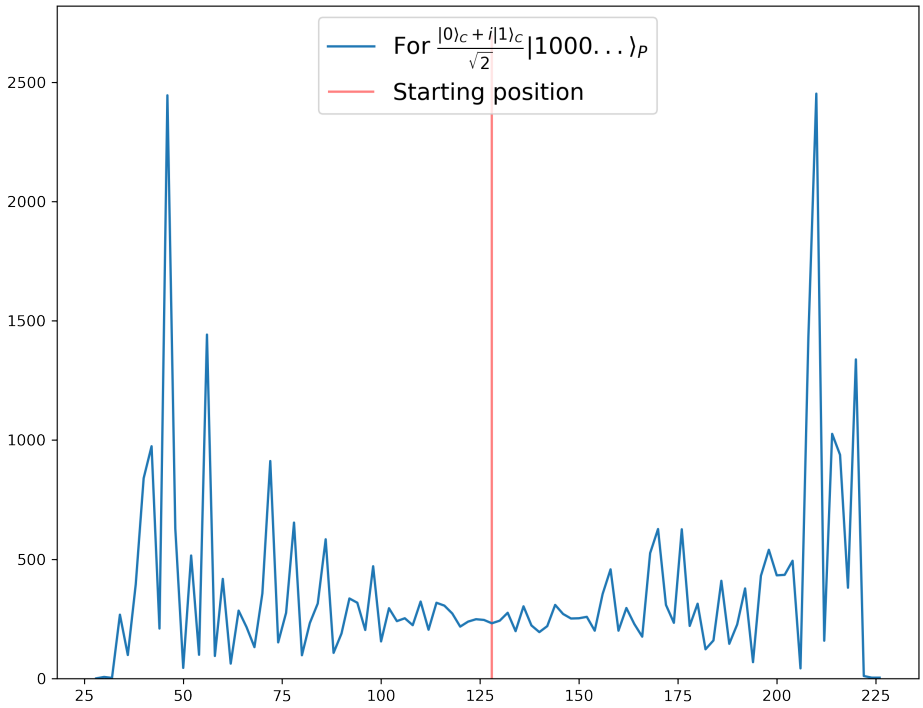
What happens if the coin state is initially $|1\rangle$? We find that the graph is basically the same thing but mirrored:



This time the peak is on the left side with a small peak on the right side

If we were to theoretically add these two states somehow, we could get a symmetric distribution. Adding just $|0\rangle$ and $|1\rangle$ will not yield the desired result, instead we must include a complex number: To "combine" these two graphs, we can create a superposition by applying the H and the S gate:

$$S \cdot H\,|0\rangle = S\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) = \frac{|0\rangle + i\,|1\rangle}{\sqrt{2}}$$



We finally have our symmetric graph and everything is good. Or is it? We used a coin that puts the coin state in an equal superposition and yet we do not end up at our starting position most of the time. As it turns out, the quantum mechanical case of a random walk is fundamentally different from the classical case, even though our intuition tells us otherwise.

Let us now put the quantum walk to a use:

3 Quantum walk search algorithm

Before talking about the quantum walk seach algorithm, we must first discuss how to generalize the idea of a quantum walk onto more dimensions. Instead of a line, we have a n-dimensional hypercube with $N = 2^n$ nodes. The Hilbert space in that case is

$$\mathcal{H} = \mathcal{H}_C^n \otimes \mathcal{H}_P^{2^n}$$

Any state in that space can be written as superposition of

$$|\Phi\rangle = \{|a, \vec{v}\rangle, 0 \geq a \geq n-1, \vec{v} \in \{(0...0), ..., (1...1)\}\}$$

The value of a decides our movement: If $a = 0$, the walker walks to position where only the first binary value differs from the current position. For $a = 1$, second binary value changes and so on. The movement is again executed by our shift operator $\mathcal{S}$:

$$\mathcal{S}\,|a\rangle\,|\vec{v}\rangle = |a\rangle\,|\vec{v} \oplus \vec{e_a}\rangle \qquad \text{where} \qquad |\vec{e_a}\rangle = |0...1_a...0\rangle \tag{2}$$

We can thus write $\mathcal{S}$ as

$$\mathcal{S} = \sum_{a=0}^{n-1} \sum_{|\vec{v}|=0}^{2^n-1} |a\rangle\,|\vec{v} \oplus \vec{e_a}\rangle\,\langle a|\,\langle \vec{v}| \tag{3}$$

Our unitary can then again be written as

$$\mathcal{U} = \mathcal{SC}$$

The problem is now the following: We have a graph where some vertices are marked. Our goal is to find the marked vertices starting from an arbitrarily chosen position by walking until we find a marked vertex.

The set of marked vertices is denoted by $|M|$ and the transition matrix P keeps all probabilities to move between nodes. We start by defining:

$$|p_x\rangle = \sum_y \sqrt{P_{xy}}\,|y\rangle$$

which corresponds to a superposition over all neighbouring states $|y\rangle$ of $|x\rangle$. P_{xy} corresponds to the probability of walking from to$|x\rangle$ to $|y\rangle$

If $|x\rangle\,|y\rangle$ is our basis state, we can subdivide all states into good G and bad B:

$$|G\rangle = \frac{1}{\sqrt{|M|}} \sum_{x \in M} |x\rangle\,|p_x\rangle$$

and

$$|B\rangle = \frac{1}{\sqrt{N - |M|}} \sum_{x \notin M} |x\rangle\,|p_x\rangle$$

We define the fraction of all marked states as $\epsilon = \frac{|M|}{N}$ and $\theta = \arcsin(\sqrt{\epsilon})$.

Let us now implement this algorithm on the 4-dimensional hypercube, e.g. nodes of degree 4 and $N = 2^4 = 16$ nodes. We remind ourselves that our Hilbert space is written as $\mathcal{H} = \mathcal{H}_C^4 \otimes \mathcal{H}_P^{16}$.

Let us define the behaviour corresponding to the coin state:

$\|00\rangle_C$	add 1 to q_0	$\|0000\rangle \overset{\text{shift}}{\Longrightarrow} \|1000\rangle$
$\|01\rangle_C$	add 1 to q_1	$\|0000\rangle \overset{\text{shift}}{\Longrightarrow} \|0100\rangle$
$\|10\rangle_C$	add 1 to q_2	$\|0000\rangle \overset{\text{shift}}{\Longrightarrow} \|0010\rangle$
$\|11\rangle_C$	add 1 to q_3	$\|0000\rangle \overset{\text{shift}}{\Longrightarrow} \|0001\rangle$

The general algorithm consists of the following steps:

1) Initialize a uniform superposition over all edges:

$$|U\rangle = \frac{1}{\sqrt{N}} \sum_x |x\rangle |p_x\rangle = \sin\theta |G\rangle + \cos\theta |B\rangle$$

   This can be realized by applying Hadamard gates to all qubits.

2) We repeat the following process $\mathcal{O}\left(1/\sqrt{\epsilon}\right)$ times:

   a) Reflect our state through the bad state $|B\rangle$ by using a phase oracle.

   b) Reflect our resulting state through the initial state $|U\rangle$ by applying phase estimation on our unitary operation.

3) We measure the result.

Step 1 Uniform superposition

This can be done by applying Hadamard gates to all 4 position qubits and the 2 coin qubits:

```
circuit.h([Position[0]]) #Apply Hadamard to position qubit
circuit.h([Position[1]]) #Apply Hadamard to position qubit
circuit.h([Position[2]]) #Apply Hadamard to position qubit
circuit.h([Position[3]]) #Apply Hadamard to position qubit

circuit.h([Coin[0]]) #Apply Hadamard to coin qubit
circuit.h([Coin[1]]) #Apply Hadamard to coin qubit
```

Step 2a Reflection through $|B\rangle$

This will be done by our phase oracle known from Grover's algorithm. The way we could do this, is by creating 16 different phase oracles for each possible position state to mark. This would of course be very tedious so I instead used a function of Qiskit to turn a matrix into an operator. As a reminder (and example), the phase oracle U_{11} that marks the state$|11\rangle$ would turn $(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$ into $(|00\rangle + |01\rangle + |10\rangle - |11\rangle)$.

So we can start with a 16×16 matrix (4-qubit gate) and turn the corresponding $+1$ to -1:

```
def PhaseOracle(statestomark): #Will create our Phase Oracle with
    choice of states to search

    PhaseOracleMatrix=np.identity(16) #We create a 16x16 identity

    for binary in statestomark: #For the binaries in the given list
        if len(binary)!=4:
            return False
        else:
            #We create the PhaseOracle from the given binary by
    flipping the corresponding entry
            PhaseOracleMatrix[int(binary,2),int(binary,2)]=-1

    PhaseOracleOperator=qi.Operator(PhaseOracleMatrix) #We let Qiskit
     create the corresponding gate

    return PhaseOracleOperator #This function will return us a 4-
    qubit gate
```

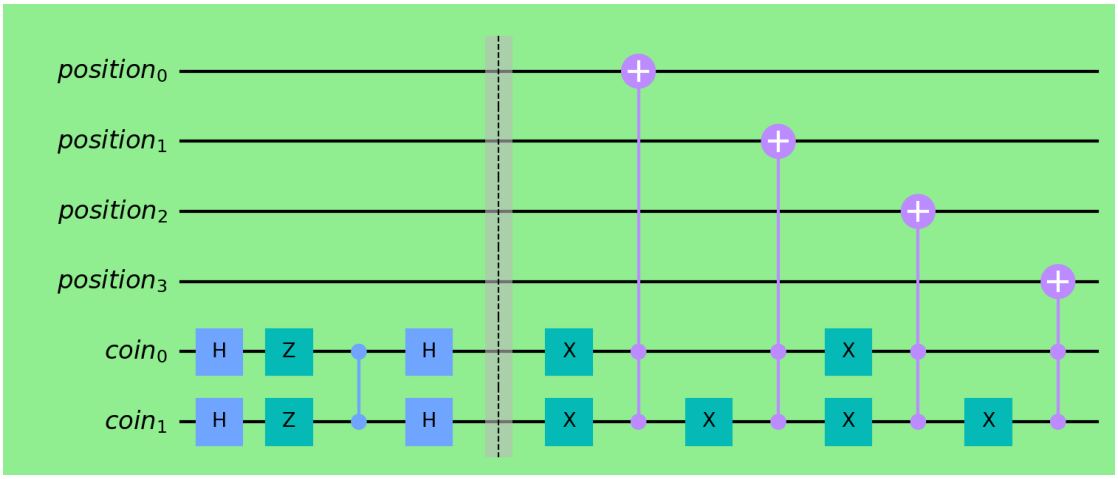
Thus we can choose which states to mark, as the function will create the right gate.

Step 2b Reflection through $|U\rangle$

This reflection works by applying a phase estimation on the unitary that consists of the coin and shift operator. We first define the unitary with an argument *inverse* that allows us to reverse the unitary. We will further output the unitary as controlled version:

```
def OneStep(inverse):
    circuit = QuantumCircuit(6, name='One Step') #Create

    circuit.h([4,5]) #Apply the coin
    circuit.z([4,5]) #Apply the coin
    circuit.cz(4,5) #Apply the coin
    circuit.h([4,5]) #Apply the coin

    circuit.x([4,5]) #Apply the shift operator
    circuit.ccx(4,5,0) #Apply the shift operator
    circuit.x(5) #Apply the shift operator
    circuit.ccx(4,5,1) #Apply the shift operator
    circuit.x([4,5]) #Apply the shift operator
    circuit.ccx(4,5,2) #Apply the shift operator
    circuit.x(5) #Apply the shift operator
    circuit.ccx(4,5,3) #Apply the shift operator

    if inverse==True:
        return circuit.inverse().control() #inverse and controlled
   version of the circuit
    elif inverse==False:
        return circuit.control() #controlled version of the circuit
```

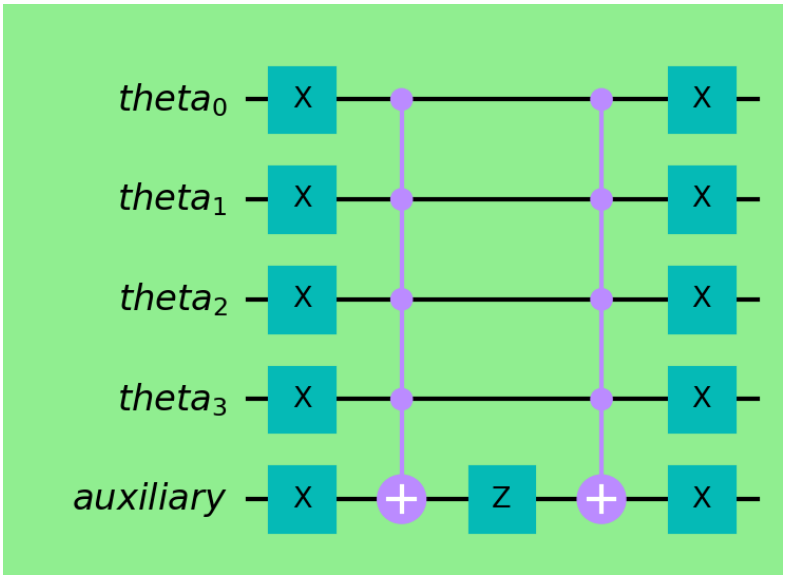
Without the control this is a 7 qubit gate, but this is what actually happens to the 6 qubits (renamed to make it clearer):



To actually do the rotation around $|U\rangle$, we also need an auxiliary gate that rotates an auxiliary qubit if and only if the others are non zero. This corresponds to checking whether the angle θ to $|U\rangle$ is 0 or not:

```
def MarkAuxiliary(): #Will help us by marking an auxiliary gate
    circuit = QuantumCircuit(5, name='Auxiliary')

    circuit.x([0,1,2,3,4]) #Apply X to all 5 qubits
    circuit.append(MCXGate(4),[0,1,2,3,4]) #Apply controlled X gate,
   with qubit 0-3 controlling X
    circuit.z(4) #Apply Z gate to lower qubit
    circuit.append(MCXGate(4),[0,1,2,3,4]) #Apply controlled X gate,
   with qubit 0-3 controlling X
    circuit.x([0,1,2,3,4]) #Apply X to all 5 qubits

    return circuit.to_instruction() #Return a gate of the circuit
```

This is what the function applies to 5 qubits:

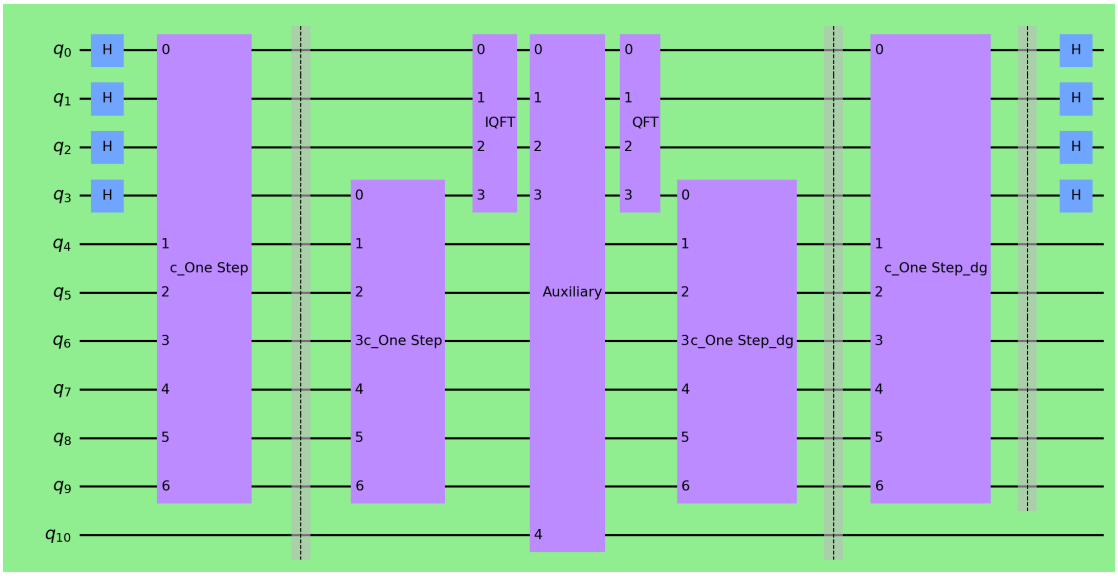


We now have all the tools to do our phase estimation: We start by creating an 11-qubit circuit and first apply 4 hadamards to the theta qubits. After that, we recursively apply the controlled unitary. We then apply the inverse quantum Fourier transformation to the theta qubits. Now we use the auxiliary gate that rotate the auxiliary qubit if $\theta \neq 0$. Now the quantum Fourier transformation reverses the phase estimation and we then recursively apply the *inverse* controlled unitary. Afterwards, 4 hadamards are again applied to the theta qubits.

```
def PhaseEstimation(): #The complete Phase Estimation
    circuit = QuantumCircuit(11, name='Phase Estimation')

    circuit.h([0,1,2,3]) #Apply Hadamard to qubit 0-3
    for i in range(0,4): #Apply the (coin,shift)-operator to qubit
    4-9 (position qubits) with i controlling the gate
        stop = 2**i
        for j in range(0,stop):
            circuit.append(OneStep(inverse=False), [i,4,5,6,7,8,9])

    circuit.append(QFT(4, inverse=True).to_instruction(), [0,1,2,3])
    #Apply the inverse QFT to the qubits 0-3

    #Apply the "mark auxiliary"-gate to qubits 0-3 with qubit 10
    being the auxiliary gate
    circuit.append(MarkAuxiliary(), [0,1,2,3,10])

    circuit.append(QFT(4, inverse=False).to_instruction(), [0,1,2,3])
     #Apply QFT to the qubits 0-3
    for i in range(3,-1,-1): #Apply the (shift,coin)-operator to
    qubit 4-9 (position qubits) with i controlling the gate
        stop = 2**i
        for j in range(0,stop):
            circuit.append(OneStep(inverse=True), [i,4,5,6,7,8,9])

    circuit.barrier(range(0,10))
    circuit.h([0,1,2,3]) #Apply Hadamard to qubit 0-3

    return circuit.to_instruction() #Return the Phase Estimation as
    11 qubit gate
```

The function will output an 11-qubit gate that does all these steps.

If we print the phase estimation circuit and place a barrier where the unitary ("One Step") and its inverse ("One Step dg") is recursively applied, we get the following:



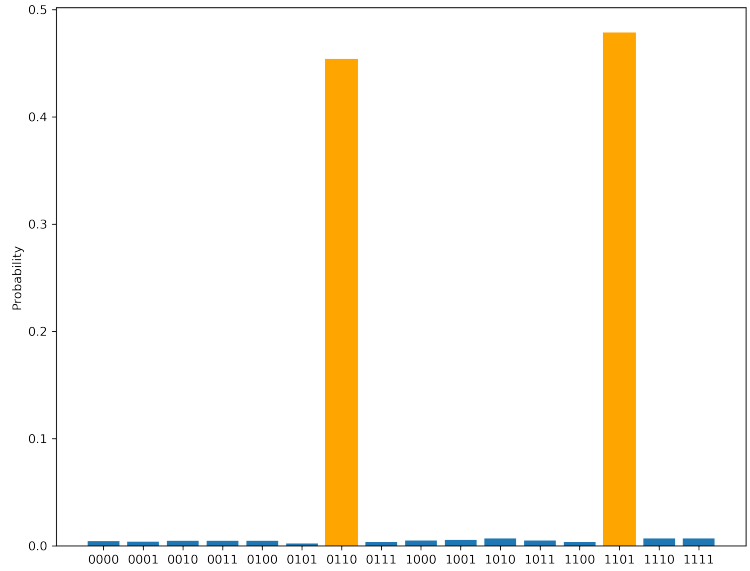
We finally have all the tools to create the quantum walk search algorithm. We start by creating a 15-qubit circuit: 4 for theta, 4 for the positon, 2 for the coin, 1 for the auxiliary qubit and 4 for the measurement result.

We start by applying a Hadamard to each position qubit and do the same to the coin qubits. We then repeat the application of our phase oracle and the phase estimation 1 or 2 times, depending on the number of marked states (actually its $\mathcal{O}(\sqrt{1/\epsilon})$). At the very last we can measure the 4 position qubits.

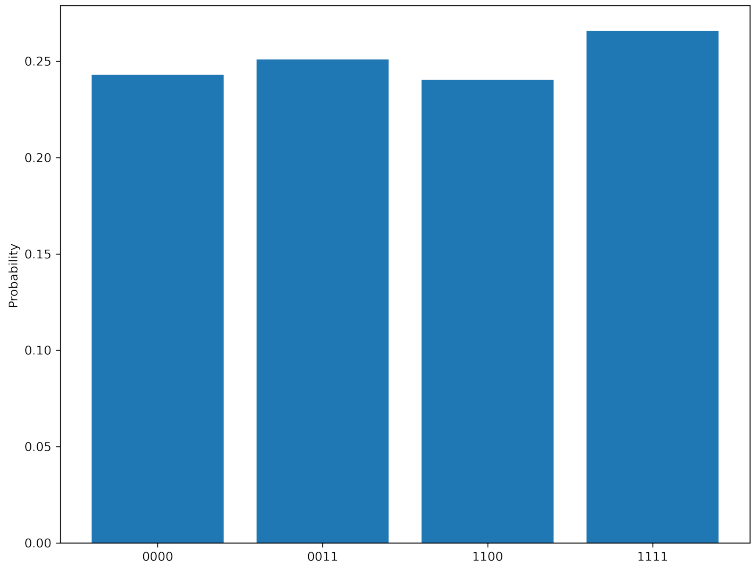
The full code is then:

```
def QuantumWalkSearch(StatesToSearch): #Our full algorithm

    Theta = QuantumRegister(4, 'theta')
    Position = QuantumRegister(4, 'position') #Keeps track of
   position
    Coin = QuantumRegister(2, 'coin') #The 2 coin qubits where the
   coin operator is applied to
    Auxiliary = QuantumRegister(1, 'auxiliary') #The auxiliary gate
   that is connected to theta
    ClasRegister = ClassicalRegister(4, 'c') #A classical register
   for the measurement
    circuit = QuantumCircuit(Theta, Position, Coin, Auxiliary,
   ClasRegister)

    circuit.h([Position[0]]) #Apply Hadamard to position qubit
    circuit.h([Position[1]]) #Apply Hadamard to position qubit
    circuit.h([Position[2]]) #Apply Hadamard to position qubit
    circuit.h([Position[3]]) #Apply Hadamard to position qubit
    circuit.h([Coin[0]]) #Apply Hadamard to coin qubit
    circuit.h([Coin[1]]) #Apply Hadamard to coin qubit
    if len(StatesToSearch) in [1,2]:
        iterations = 2
    elif len(StatesToSearch) in [3,4]:
        iterations =1

    for i in range(0,iterations): #Apply the PhaseOracle and Phase
   Estimation twice
        try:
            circuit.append(PhaseOracle(StatesToSearch),[4,5,6,7]) #
   Set Phase Oracle to mark given states
            circuit.append(PhaseEstimation(),
   [0,1,2,3,4,5,6,7,8,9,10]) #Apply Phase Estimation operator
        except:
            return print("States must be 4-bit state.")
    circuit.measure(Position[0], ClasRegister[0]) #Measure the 4
   position qubits
    circuit.measure(Position[1], ClasRegister[1]) #Measure the 4
   position qubits
    circuit.measure(Position[2], ClasRegister[2]) #Measure the 4
   position qubits
    circuit.measure(Position[3], ClasRegister[3]) #Measure the 4
   position qubits
    backend = Aer.get_backend('qasm_simulator')
    job = execute( circuit, backend, shots=3000 )
    hist = job.result().get_counts()

    return plot_histogram(hist)
```

Let us test our code by marking the states $|0110\rangle$ and $|1101\rangle$:



We can clearly see that the circuit almost always collapses to these marked states. Testing it with four marked states $|0000\rangle$, $|1100\rangle$,$|0011\rangle$ and $|1111\rangle$:



In this case, all 3000 measurements revealed that the circuit collapsed into one of the marked states.

4 Remark

As we have seen, the quantum walk search algorithm is very similar to Grover's search algorithm. Namely that we also start in a superposition, the runtime is $\mathcal{O}(\sqrt{n})$, we mark states and use the Grover diffusion operator. And yet, after all these similarities, there is a difference:

1) While Grover uses a two-dimensional subspace spanned by non-target states and target states, the quantum walk search spans a two-dimensional subspace out of a superposition of all states and an approximation of the target state.

2) The final states contains contributions from neighbouring nodes, the Grover search outputs only the final state.

5 Conclusion

We have seen that the quantum walk is fundamentally different than the classical random walk. The final probability distribution depends on the coin state chosen. We saw that $\frac{|0\rangle+i|1\rangle}{\sqrt{2}}$ resulted in a symmetric distribution with two peaks instead of one. The quantum walk can be used to search for marked states in a similar manner as Grover's Algorithm. Having implemented this on the 4D-hypercube, we were able to successfully use the algorithm.

6 Sources

1) https://Qiskit.org/textbook/ch-algorithms/quantum-walk-search-algorithm.html

2) https://medium.com/Qískit/studying-quantum-walks-on-near-term-quantum-computers-f60fd2395f04

3) https://vicpina.com/walk-the-quantum-walk/

4) https://windowsontheory.org/2018/12/23/introduction-to-quantum-walks/

5) Quantum Random Walk Search and Grover's Algorithm -An Introduction and Neutral-Atom Approach by Anna Maria Houk