Stony Brook University

ESE 561: Theory of Artificial Intelligence
Spring 2025

QUANTUM RANDOM WALK FOR GRAPH SEARCH

PROJECT REPORT

AUTHOR

Mahnoor Fatima
mahnoor.fatima@stonybrook.edu

DATE OF SUBMISSION

May 17, 2025

Contents

List of Figures

List of Tables

LISTINGS

Section 1

INTRODUCTION

In the course *ESE561: Theory of Artificial Intelligence*, we have studied various search algorithms for graph-based problems (maps, trees, etc.). Moreover, we have studied about Markov processes and single and multi-agent (stochastic) environments for decision making. In all these algorithms, an agent can only explore one solution at a time by using a certain strategy.

What if an agent had the option to explore more than one solutions simultaneously? This is the motivation behind this project: searching a graph using Quantum Random Walks. This project utilizes a lot of concepts already explored in ESE561 (stochastic environments, Markov chains, etc.) while expanding on the decision-making algorithms by adding a quantum spin.

1.1 Quantum Random Walk

Quantum random walks are a fundamental framework for exploring quantum systems, offering a powerful extension to classical random walks by leveraging quantum superposition and interference. Unlike classical random walks, where a walker moves between discrete states based on probabilistic transitions, quantum random walks evolve according to unitary transformations, enabling faster propagation and richer dynamics.

Quantum random walks can be categorized into discrete-time and continuous-time models:

- **Discrete-Time Quantum Random Walk:** This algorithm involves a quantum coin operator that controls the movement of the walker. These are widely used in quantum algorithms, such as quantum search and graph traversal.
- **Continuous-Time Quantum Random Walk:** This algorithm models natural quantum processes governed by the system's evolution operator (called a Hamiltonian), making them useful for physical simulations.

In this project, Discrete QRW has been used for graph search due to its conceptual similairty to "classic" random walks.

1.2 The Quantum Magic

Quantum random walks are, algorithmically, the same as classical random walks, i.e., based on a coin toss, the next step of the walker is determined. However, there are two principle properties of quantum computing that render QRWs starkly different from classical random walks.

1.2.1 Quantum Superposition

A quantum object can exist in more than one states simultaneously until it is measured.

This can best be understood by the famous Scrödinger's cat (fig. 1.1), a poor cat shut in a box with a radioactive substance triggering a kill switch with a 50% probability; the cat is, rather stragely, both alive and dead until the box is opened.

It is quantum superposition allows the quantum walker to exist at more than one nodes simultaneously; this will be discussed further in section 2.

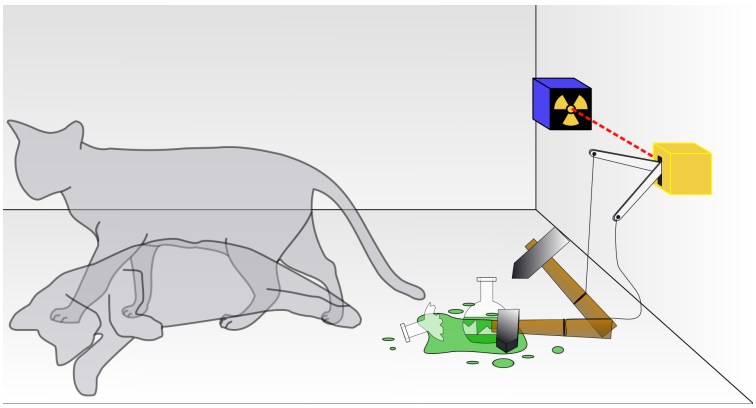


Figure 1.1: Scrödinger's cat. Source: *Wikipedia*

1.2.2 Quantum Interference

Quantum states can interact with each other, either constructively or destructively, to influence the outcome.

In classical computing, the bits can either have a state 0 or 1. However, in quantum computing, the qubits can not only be in a combination of 0 and 1 states (thanks to quantum superposition), but can also have a phase associated with their state. This phase appears because quantum computing uses complex-valued states instead of the commonly used binary-valued states in classical computing and ranges from $-\pi$ to π.

To understand the effect of phases for quantum computing, consider a qubit in state $|0\rangle + |1\rangle$ and another in $|0\rangle - |1\rangle$. Adding them together will cause a constructive interference of the $|0\rangle$ states and destructive interference of the $|1\rangle$ states, thus resulting in a state $|0\rangle$.

Quantum interference is used in a lot of quantum algorithms to amplify the probability of the desired outcome and diminish the probability of the undesired outcomes. This project will make a similar use of quantum interference to find the desired graph node.

1.3 Symbolic Notation

The quantum comuting's bra-ket notation has been used in this report. It uses a ket ($|\rangle$) to depict a column vector and a bra ($\langle|$) to depict a row vector. Thus, an inner product of two states ϕ and ψ can be written as $\langle\phi|\psi\rangle$ and their outer product can be written as $|\phi\rangle\langle\psi|$.

Section 2

METHODOLOGY

This project has been implemented using the algorithm described in [2].

2.1 Problem Formulation

This problem deals with the traversal of an n-dimensional hypercube with 2^n nodes with each node connected with n other nodes. This is illustrated in fig. 2.1. For this project, the use-case has been restricted to $n = 3$ to allow for extensive testing and improved understanding of the project.

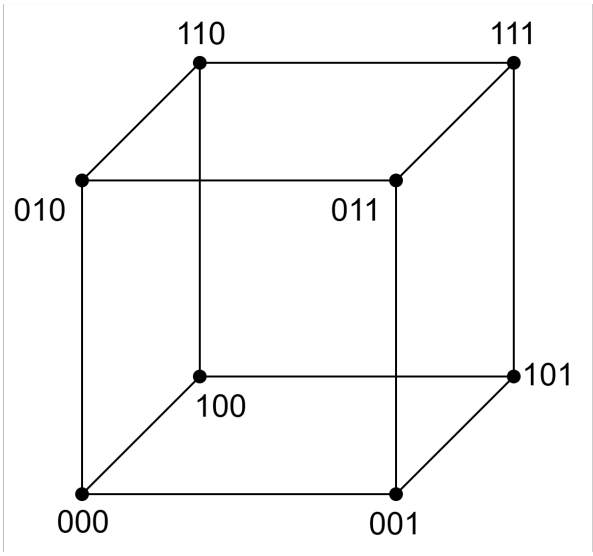


Figure 2.1: Graph hypercube

The elementary basis vectors of the 3D-hypercube are as follows:

$$\vec{e_0} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$$
$$\vec{e_1} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$$
$$\vec{e_2} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$$
$$\vec{e_3} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$$

The null vector e_0 has been included in the algorithm to complete the mapping of all states $\leq n$ states generated by the quantum coin.

2.1.1 Graph Traversal

The next state of the "walker" is chosen by taking an XOR of $\vec{x}$ with the eigenvector $\vec{e_d}$ corresponding to the state d of the $(n+1)$-state coin. This ensures that only one bit-flip occurs during one step of the quantum random walk. This allows for adequate quantum error correction and mitigation techniques to be used.

The traversal of a quantum walker at node '0' is depicted in fig. 2.2.

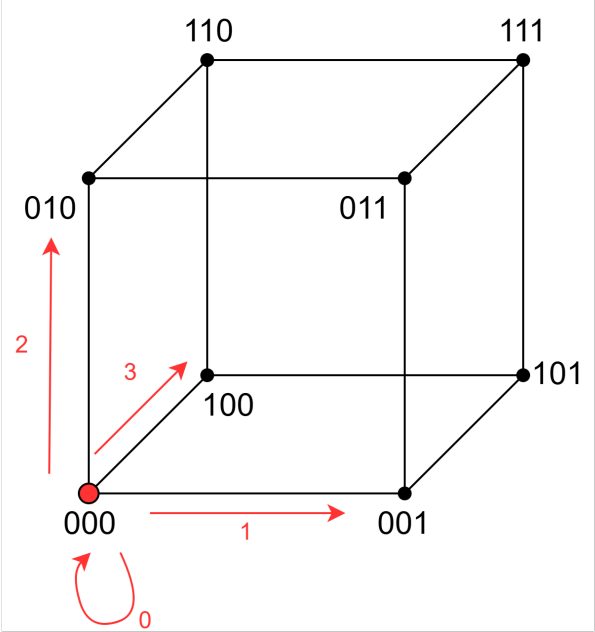


Figure 2.2: Graph traversal from node '0' based on the state of the coin

2.2 Solution—Quantum Random Walk

2.2.1 The Quantum Coin

The quantum coin distributes the quantum states uniformly across all states of the quantum coin. This is computed as

$$C = -I + 2\sum_{i=0}^{n} |i\rangle \langle i| .$$

2.2.2 The Shift Operator

The shift operator is defined as

$$S = \sum_{d}\sum_{x} \begin{pmatrix} d \\ x \oplus e_d \end{pmatrix} \begin{pmatrix} d & x \end{pmatrix},$$

i.e., it transforms the state $\begin{pmatrix} d & x \end{pmatrix}$ to $\begin{pmatrix} d & e_d \oplus x \end{pmatrix}$, thus "shifting" the state x to $x \oplus e_d$.

This transformation ensures that only one bit-flip occurs during one step of the quantum random walk. It's an economical design choice for quantum error correction by using unit-distance code.

2.2.3 The Quantum Random *Walk*

Each step of the Quantum Random Walk includes the implementation of the coin operator C followed by the shift operator S.

$$U = S.C$$

The entire transformation of the "walker" state during QRW is the successive application of the U operator.

2.3 Graph Search with Quantum Random Walk

To use QRW for graph search, the only significant change made is modifying the quantum coin to mark the desired state. This takes $\lceil \frac{\pi}{2}\sqrt{2^{n-1}} \rceil$ steps to find the optimal solution (and walking for more steps actually takes us away from the optimal solution).

2.3.1 Modified Coin Operator

For graph search, the state of the desired node m is marked by modifying the original coin operator C as follows:

$$C' = C \otimes I - (C - I) \otimes |m\rangle \langle m|$$

This results in the state $|m\rangle$ being marked with a negative phase which amplifies the m^{th} node's probability each time the quantum walker hits the node.

2.3.2 Algorithm

1. Create an equal superposition of all states of the coin.
2. Create the marked coin operator C' with the node you want to detect as described in 2.3.1.
3. Implement the $U' = SC'$ transformation for $\lceil \frac{\pi}{2}\sqrt{2^{n-1}} \rceil$ times to find the marked element.
4. Measure the $c \otimes s$ qubits and record the simulation result.
5. Find the element with the highest probability of detection. The shift state x corresponding to this element is the marked element detected by the quantum circuit.

2.4 Code Implementation

The project has been implemented using Qiskit [4], IBM's open-source quantum computing SDK. The choice of the SDK was based on the popularity and widespread use of Qiskit in the quantum computing community and the availability of tech support by Qiskit's team and its open-source community.

Fig. 2.3 shows the quantum circuit implementation of the QRW algorithm for three steps in Qiskit. The qubits c_0 and c_1 are assigned to the quantum coin and the qubits s_0, s_1, and s_2 are assigned for the shift register. The module *coin* implements the coin operator C while the $shift\ operator$ module implemets the S operator. The rightmost part of the circuit (with grey meter-like components) are the measurement devices which meaure the quantum states of the registers on the classical *meas* register.

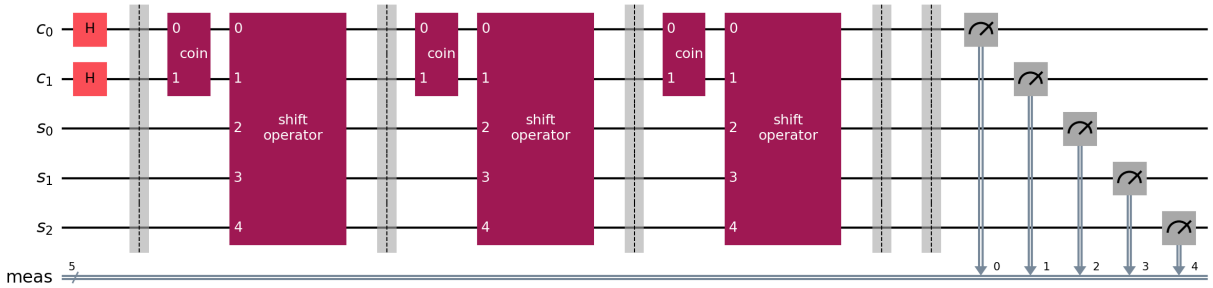


Figure 2.3: Quantum Circuit for Quantum Random Walk

Fig. 2.4 shows the quantum circuit implementation of the QRW algorithm for graph search for three steps in Qiskit. The only change from the Quantum Random Circuit is the replacement of the *coin* module with the *marked coin* module

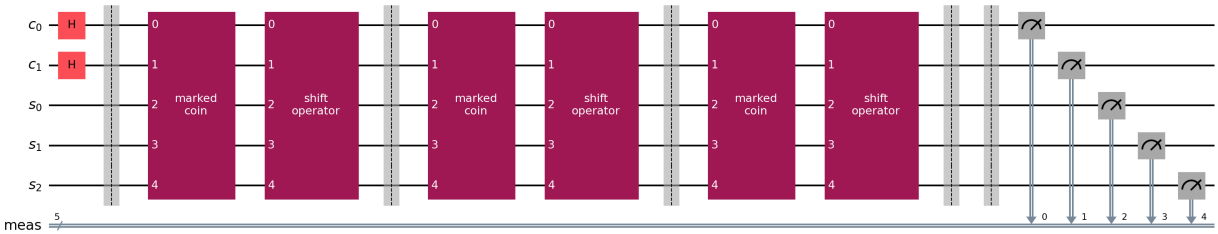


Figure 2.4: Quantum Circuit for Quantum Random Walk for graph search

The project code is documented in Appendix A. The code, along with other project resources, is also available at this public GitHub repository.

Section 3

RESULTS

3.1 Quantum Random Walk

Fig. 3.1 depicts the simulation results of QRW when a state $|s = 0\rangle$ takes its first step. The results in fig. 3.1 are tabulated in tab. 3.1. It can be verified that the steps taken by the "walker" based on the state of the coin are as depicted in fig. 2.2.

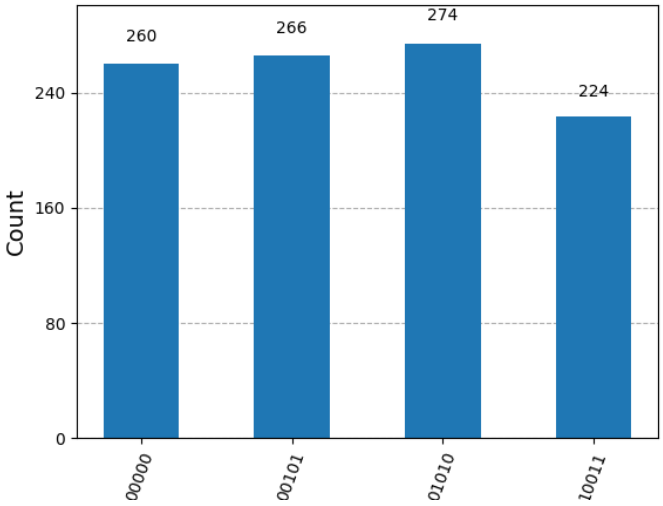


Figure 3.1: Quantum Random Walk - one step from node '000'

Node	Coin	Count
000	00	260
001	01	266
010	10	274
100	11	224

Table 3.1: Tabulated results of QRW simulation

3.2 Search Accuracy

According to [2], the QRW search algorithm has an expected accuracy of $\frac{1}{2} - O(n)$. For $n = 3$, the search accuracy should, thus, range from $\approx 16.67\%$ to 50%. The simulation results reported in tab. 3.2 show that for a step size $k = 4$, the search accuracy is 62.5% and 50% for step sizes 3 and 5.

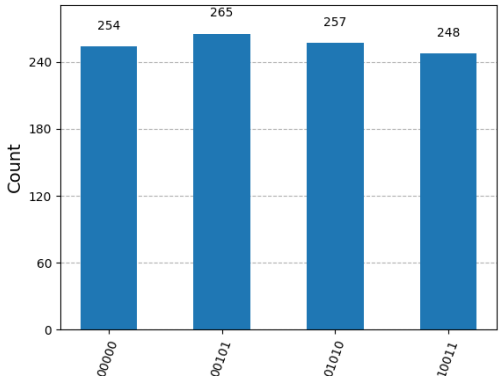
Steps	3	4	5
0	✓	✓	✓
1	✓	✓	×
2	✓	✓	×
3	×	×	✓
4	✓	×	×
5	×	×	✓
6	×	✓	✓
7	×	✓	×
Accuracy	0.5	0.625	0.5

Table 3.2: Accuracy of graph search for different number of steps

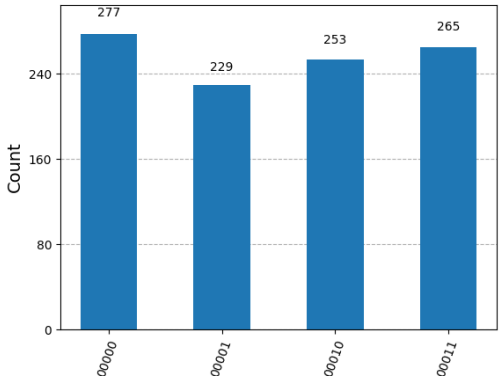
3.3 Periodicity of Quantum Random Walk

The period of this QRW algorithm is $2\pi\sqrt{2^{n-1}}$. For $n = 3$, this equals $12.57 \approx 13$.

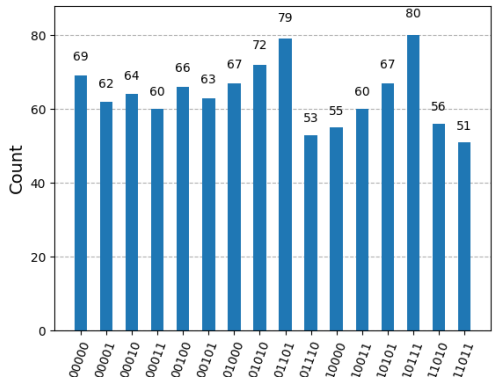
This periodicity of QRW is illustrated in fig. 3.2. It is worth noting that the indices are lined up like points on a circle, with the simulation results for a QRW with steps $k = i$ being the same as a QRW with steps $12 - i$. The slight discrepancy to this rule at higher indices can be attributed to the inexactness of the period—due to the discreteness of this QRW algorithm, exact periodicity in the results cannot be achieved. However, a decent approximation with a reduced error can be expected for higher values of n.



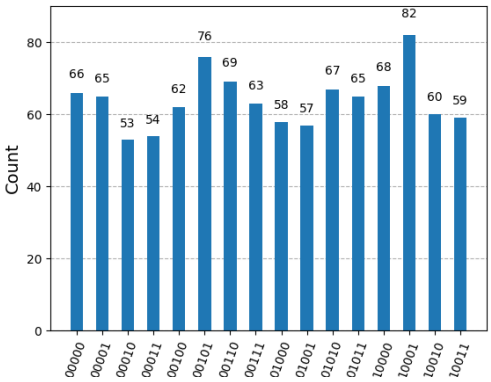
(a) k=1



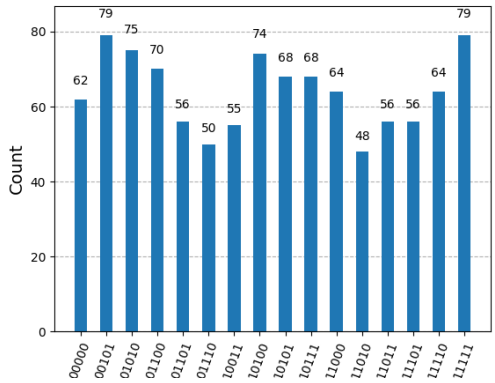
(b) k=12



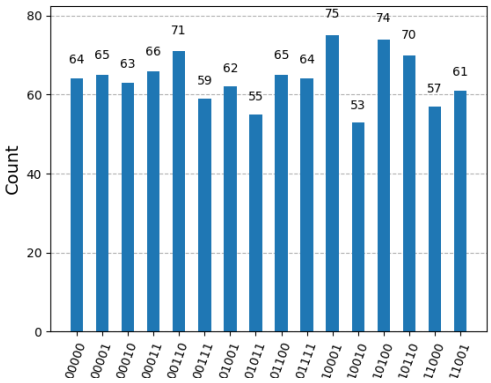
(c) k=2



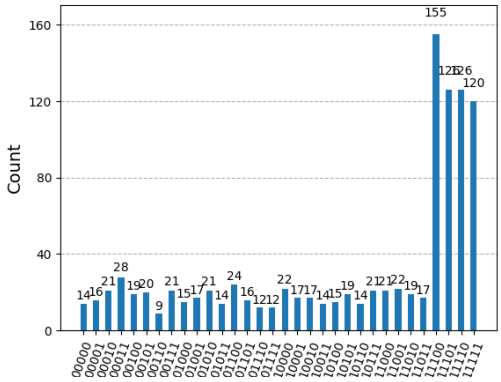
(d) k=11



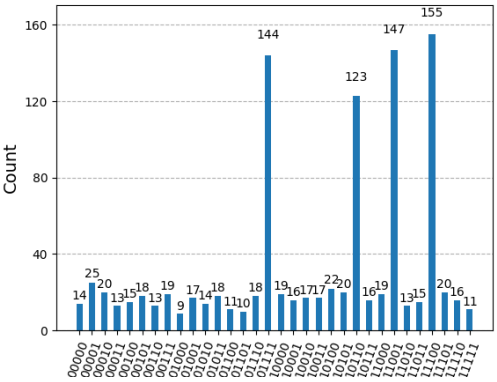
(e) k=3



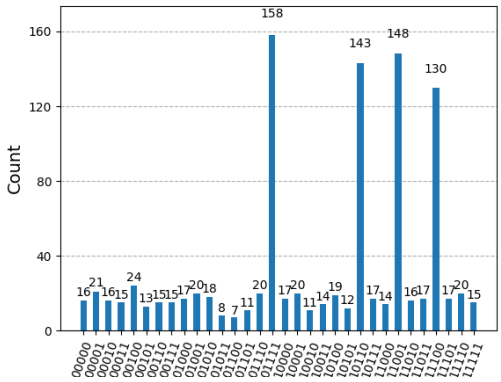
(f) k=10



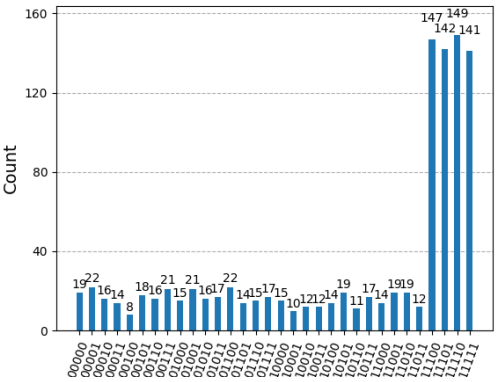
(g) k=4



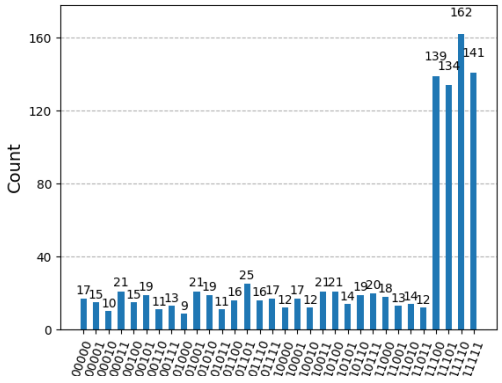
(h) k=9



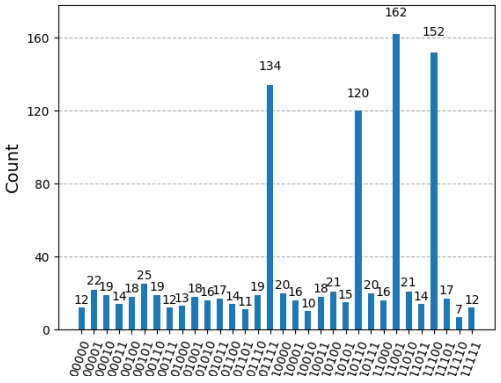
(i) k=5



(j) k=8



(k) k=6



(l) k=7

Figure 3.2: Periodicity of Quantum Random Walk

Section 4

CONCLUSION

This project implements the Quantum Random Walk algorithm to search an n-dimensional hypercube graph as described in [2] using IBM's Qiskit SDK, and has been implemented as a course requirement for ESE561: Theory of Artficial Intelligence. The entire project code was developed independently without consulting any large language models (LLMs), automated code generation tools, or external resources and the expected results as described in [2] were successfully obtained.

Despite the project achieving the expected results, the algorithm accuracy was not very high (62.5% for $n = 3$). This can be attributed to the inexact implementation of Discrete Quantum Random Walks. Better quantum algorithms are already available, like Grover's algorithm [1] (which uses operators which are very similar to QRW). Still, this project allowed for the opportunity to understand the concepts learned in the course ESE561 from a counterintuitive lens, thus providing a fresh perspective and a deeper understanding.

There are many avenues for improvement in the project, which include better visualizations and better Quantum Random Walk algorithms (like [3] and [5]).

Appendix A

PROJECT CODE

A.1 Quantum Random Walk

```
# import libraries
from qiskit import QuantumCircuit, QuantumRegister, transpile
from qiskit.circuit.library import UnitaryGate
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
from numpy import pi, array
import numpy as np
from helper import print_table_from_dict

# Define the coin
n = 3 # dimension of graph
n_d = int(np.ceil(np.log2(n))) # dimension of coin

# declare the shift operator
ed = np.array([[0, 0, 0],
               [0, 0, 1],
               [0, 1, 0],
               [1, 0, 0]])

S = np.zeros((2**(n+n_d), 2**(n+n_d)))
for d in range(2**n_d):
    for x in range(2**n):
        vd = np.zeros((1, 2**n_d))
        vd[0, d] = 1
        vx = np.zeros((1, 2**n), dtype=np.int64)
        vx[0, x] = 1
        ved = np.zeros((1, 2**n), dtype=np.int64)
        ed_decimal = int("".join(map(str, ed[d])), 2)
        ved[0, np.bitwise_xor(ed_decimal, x)] = 1
        col  = np.kron(ved, vd).reshape(2**(n+n_d), 1)
        row  = np.kron(vx, vd).reshape(1, 2**(n+n_d))
        S += np.dot(col, row)
np.set_printoptions(threshold=np.inf)
S_gate = UnitaryGate(S, label='shift\noperator')
print("Shift operator:\n", S_gate.params)

# Declare the coin operator
C = UnitaryGate(2/2**n_d*np.ones((2**n_d, 2**n_d)) - np.eye(2**n_d), label='coin')
print("Coin operator:\n", C.params)

# Create the quantum circuit
qc  = QuantumCircuit()
steps = 3
# Add quantum registers to the circuit
shift = QuantumRegister(n, 's') # qubits associated with the graph nodes
coin = QuantumRegister(n_d, 'c') # qubit associated with the coin
qc.add_register(coin)
qc.add_register(shift)
```

```
# Create equal superposition of coin and walker
[qc.h(coin[i]) for i in range(n_d)]
qc.barrier()
for _ in range(steps):
    qc.append(C, coin)
    # Apply the walk
    qc.append(S_gate, range(qc.num_qubits))
    qc.barrier()
# Measure the state of the walker
qc.measure_all()
# Display the quantum circuit
qc.draw('mpl', filename=f"qrw_circuit_{steps}.png")

# Simulate the quantum circuit
sim_ideal = AerSimulator()
tqc = transpile(qc, sim_ideal)
result_ideal = sim_ideal.run(tqc).result()
plot_histogram(result_ideal.get_counts(), filename=f"qrw_sim_results_{steps}.png")

print_table_from_dict(result_ideal.get_counts(), n, n_d)
```

Listing A.1: Code for `qrw.py`

A.2 Helper file

```
# function to present simulation results as a table
def print_table_from_dict(mydict, n, n_d):
    print("----------------------------")
    print("Node\t Coin\t Probability")
    print("----------------------------")
    for count in mydict:
        print(count[0:n], '\t', count[n:n+n_d],'\t', mydict[count]/1000)
    print("----------------------------")
```

Listing A.2: Code for `helper.py`

A.3 Quantum Random Walk for Search

```
# import libraries
from qiskit import QuantumCircuit, QuantumRegister, transpile
from qiskit.circuit.library import UnitaryGate
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
from numpy import pi, array
import numpy as np
from helper import print_table_from_dict

# Define the variables
n = 3 # dimension of graph
n_d = int(np.ceil(np.log2(n))) # dimension of coin

# declare the shift operator
ed = np.array([[0, 0, 0],
               [0, 0, 1],
               [0, 1, 0],
               [1, 0, 0]])

S = np.zeros((2**(n+n_d), 2**(n+n_d)))
for d in range(2**n_d):
    for x in range(2**n):
        vd = np.zeros((1, 2**n_d))
        vd[0, d] = 1
        vx = np.zeros((1, 2**n), dtype=np.int64)
```

```
        vx[0, x] = 1
        ved = np.zeros((1, 2**n), dtype=np.int64)
        ed_decimal = int("".join(map(str, ed[d])), 2)
        ved[0, np.bitwise_xor(ed_decimal, x)] = 1
        col  = np.kron(ved, vd).reshape(2**(n+n_d), 1)
        row  = np.kron(vx, vd).reshape(1, 2**(n+n_d))
        S += np.dot(col, row)
np.set_printoptions(threshold=np.inf)
S_gate = UnitaryGate(S, label='shift\noperator')

# Declare the coin operator
C = UnitaryGate(2/2**n_d*np.ones((2**n_d, 2**n_d)) - np.eye(2**n_d), label='coin')
marked_state = 3
marked_state_mat = np.zeros((2**n, 2**n))
marked_state_mat[marked_state, marked_state] = 1
C1 = np.kron(np.eye(2**n), C.to_matrix()) - np.kron(marked_state_mat, (C-np.eye(2**n_d)))
C1_gate = UnitaryGate(C1,  label='marked\ncoin')

# Create the quantum circuit
qc_search  = QuantumCircuit()
steps = 5
# Add quantum registers to the circuit
shift = QuantumRegister(n, 's') # qubits associated with the graph nodes
coin = QuantumRegister(n_d, 'c') # qubit associated with the coin
qc_search.add_register(coin)
qc_search.add_register(shift)
# Create equal superposition of coin and walker
[qc_search.h(coin[i]) for i in range(n_d)]
qc_search.barrier()
for _ in range(steps):
    qc_search.append(C1_gate, range(qc_search.num_qubits))
    # Apply the walk
    qc_search.append(S_gate, range(qc_search.num_qubits))
    qc_search.barrier()
# Measure the state of the walker
qc_search.measure_all()
# Display the quantum circuit
qc_search.draw('mpl',  filename=f"qrw_search_{steps}.png")

# Run the simulation
sim_ideal = AerSimulator()
tqc_search = transpile(qc_search, sim_ideal)
result_ideal = sim_ideal.run(tqc_search).result()
search_counts =  result_ideal.get_counts()
# Display the results
plot_histogram(search_counts, filename=f"qrw_search_{steps}")
# Print the simulation data
print_table_from_dict(search_counts, n, n_d)
# Print the marked element
max_key = max(search_counts, key=search_counts.get)
print("Marked element:", int(max_key[0:3], 2))
```

Listing A.3: Code for `qrw_search.py`

BIBLIOGRAPHY

[1] Lov K Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 212–219.

[2] Neil Shenvi, Julia Kempe, and K Birgitta Whaley. "Quantum random-walk search algorithm". In: *Physical Review A* 67.5 (2003), p. 052307.

[3] Mario Szegedy. "Quantum speed-up of Markov chain based algorithms". In: *45th Annual IEEE symposium on foundations of computer science*. IEEE. 2004, pp. 32–41.

[4] Robert Wille, Rod Van Meter, and Yehuda Naveh. "IBM's Qiskit tool chain: Working with and developing for real quantum computers". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1234–1240.

[5] Thomas G Wong and Andris Ambainis. "Quantum search with multiple walk steps per oracle query". In: *Physical Review A* 92.2 (2015), p. 022338.