EXPLORING GRAPHS WITH DEEPSEEK AND QUANTUM COMPUTERS

Samir Lipovaca

To my wife Besima

TABLE OF CONTENTS

Introduction

This book delves into the exploration of graph theory through the lens of DeepSeek, an advanced AI chatbot, and quantum computers, offering hopefully a fresh perspective on this timeless discipline.

Graph theory, a branch of mathematics, has long been a cornerstone in the study of networks and relationships. Its applications span various fields, from computer science and biology to social sciences and logistics. The main contributions of graph theory are manifold. It provides a robust framework for modeling relationships and interactions within a network. Graph theory has been instrumental in the development of algorithms for searching, sorting, and optimizing data. It also plays a crucial role in the study of connectivity, helping to solve problems related to network design, communication, and transportation.

DeepSeek is an AI chatbot designed to analyze and interpret complex data structures. Deepseek serves multiple purposes, making it a valuable tool across various AI-driven applications. It enhances natural language processing (NLP) by improving human-language comprehension and interaction, creating more natural AI conversations. Developers use it for code generation, automating tasks, and streamlining workflows. It's also a powerful content creation tool, assisting in writing blog posts, emails, and social media updates. For education, Deepseek supports learning and tutoring by explaining complex concepts and aiding with homework and coding tasks. Additionally, it plays a crucial role in building, optimizing, and deploying AI applications, facilitating machine learning and data analysis. Overall, Deepseek is a versatile solution designed to improve efficiency and innovation in AI-related tasks.

Quantum computers are a new type of computing device that leverage the principles of quantum mechanics to perform calculations in ways that classical computers cannot. Unlike classical computers, which use bits (0s and 1s), quantum computers use **qubits**, which can exist in multiple states of 0s and 1s simultaneously due to **superposition**. They also utilize **entanglement**, allowing qubits to be correlated in ways that enable faster and more complex computations.

Superposition and entanglement are concepts fundamental to the power of quantum computing so let’s break down these concepts in simple terms.

Superposition

Imagine you have a coin. In the classical world, the coin can either be in a state of heads or tails. However, in the quantum world, the coin can be in a state where it is both heads and tails at the same time. This is called superposition. It's like having a coin that is spinning so fast that it is simultaneously showing both sides. Only when you stop the coin and look at it (measure it) does it settle into one of the two states (heads or tails).

Entanglement

Now, let's say you have two coins that are entangled. In the classical world, if you flip two coins, the outcome of one coin flip doesn't affect the other. But in the quantum world, if two coins are entangled, the state of one coin is directly related to the state of the other, no matter how far apart they are. It's like having two magic coins: if you flip one and it lands on heads, the other one will instantly land on tails, even if it's on the other side of the universe. This connection happens instantaneously, faster than the speed of light.

In summary, superposition is like a coin being both heads and tails at the same time, and entanglement is like having two coins that are magically linked, so the state of one instantly affects the state of the other.

Quantum Computing Frameworks

Several frameworks and programming languages have been developed to write code for quantum computers. Some of the most widely used ones include:

- **Qiskit** – IBM's open-source quantum computing SDK, primarily written in Python, used for building and executing quantum circuits.
- **Cirq** – Google's Python framework designed for creating and optimizing quantum circuits, particularly for Noisy Intermediate-Scale Quantum (NISQ) devices.
- **Q#** – Microsoft's quantum programming language, designed specifically for quantum algorithms and integrated with Azure Quantum.
- **PyQuil** – Rigetti's Python library for constructing and running quantum programs using the Quil language.
- **PennyLane** – A cross-platform Python library for quantum computing, quantum machine learning, and quantum chemistry.
- **Amazon Braket** – A Python SDK for interacting with quantum devices on Amazon's cloud-based quantum computing service.

These frameworks provide tools for simulating quantum circuits, interfacing with quantum processors, and running quantum algorithms. As quantum computing continues to evolve, these frameworks will play a crucial role in making quantum programming more accessible.

Quantum computers offer several advantages over classical computers, making them a promising technology for solving complex problems. Here are some key advantages:

1. **Speed**: Quantum computers can perform certain calculations exponentially faster than classical computers. This speedup is particularly beneficial for tasks such as factoring large numbers, simulating quantum systems, and solving optimization problems.
2. **Parallelism**: Quantum computers leverage the principles of superposition and entanglement, allowing them to process multiple possibilities simultaneously. This parallelism enables quantum computers to explore vast solution spaces more efficiently.

3. **Complex Problem Solving**: Quantum computers excel at solving complex problems that are intractable for classical computers. Examples include optimization problems, cryptography, and simulating molecular structures for drug discovery.
4. **Scalability**: As data sets grow larger and more complex, quantum computing offers a scalable solution. Quantum computers can handle increasingly complex calculations without a significant loss in performance.
5. **Enhanced Pattern Recognition**: Quantum computing allows for more accurate and insightful pattern recognition. Quantum algorithms can detect correlations and patterns in data that are difficult to identify with classical methods.
6. **Real-time Analysis**: Quantum computers can perform real-time analysis of dynamic systems, making them useful for applications such as financial modeling, weather forecasting, and social network analysis.

Overall, quantum computers have the potential to revolutionize various fields by providing faster, more efficient, and scalable solutions to complex problems.

This book offers a curated exploration of Graph Theory, focusing on selected concepts chosen by the author’s curiosity. These concepts are examined using DeepSeek, which initially provided respective qiskit version of the code. The author then modified or updated this code as needed until it effectively illustrated the concept.

Here is an overview of each chapter.

1. **Bipartite Graphs**: This chapter discusses bipartite graphs, which consist of two distinct sets of vertices with edges connecting vertices from different sets. It highlights their practical applications in job matching, recommendation systems, and modeling biological interactions. The chapter also covers methods to determine if a graph is bipartite using classical and quantum algorithms.
2. **The Shortest Path In a Graph**: This chapter focuses on the shortest path problem, which involves finding the least costly route between nodes in a graph. It explains Dijkstra's Algorithm, a well-known method for solving this problem, and explores alternative approaches using quantum computing, such as quantum walk algorithms and Grover's algorithm.
3. **Palindrome Binary Numbers and Symmetric Graphs**: This chapter explores the relationship between palindrome binary numbers and symmetric graphs. It discusses how symmetric graphs can be represented using palindrome binary numbers and the implications of this representation in various applications.
4. **The SYK Model and Black Hole Information Scrambling**: This chapter delves into the SYK model, a quantum many-body system that provides insights into black hole physics, specifically information scrambling. It explains how the SYK model exhibits chaotic behavior, where local information spreads rapidly throughout the system, similar to how black holes mix information.

5. **Random Graphs and Information Spreading Inside a Black Hole**: This chapter discusses the use of random graphs, particularly the Erdős-Rényi model, to simulate quantum information spreading in black holes. It explains how researchers apply entangling gates to qubits represented in a random graph to study information scrambling, with entanglement entropy serving as a measure of this process.
6. **Spin Glass Model With A Randomly Generated Interaction Graph**: This chapter covers the spin glass model, its complexity, and the use of the Quantum Approximate Optimization Algorithm (QAOA) to determine the ground state of a spin glass model using Qiskit. It details the parameters of the QAOA algorithm and how it optimizes the ground state configuration.
7. **Deutsch-Jozsa Algorithm and Eulerian/half-Odd-half-Even Graphs**: This chapter explains the Deutsch-Jozsa algorithm, a quantum algorithm used to solve specific problems faster than classical algorithms. It also discusses Eulerian and half-odd-half-even graphs, their properties, and their relevance to the Deutsch-Jozsa algorithm.

Please feel free to download a pdf version of the book at [samlip-blip/ExploringGraphsWithDeepSeekAndQuantumComputers]

Let's dive in!

Bipartite Graphs

Imagine you have a group of people at a party, and you want to pair them up for a dance. A bipartite graph is like a dance chart where you have two separate groups of people, and each person from one group can only dance with someone from the other group.

In simpler terms, it's a way to connect two sets of things where connections only happen between the sets, not within the same set. For example, if you have a group of students and a group of books, a bipartite graph would show which student is reading which book, but it wouldn't show connections between students or between books.

Bipartite graphs have many practical applications in real life. Here are a few examples:

1. **Job Matching**: Imagine you have a group of job seekers and a group of job openings. A bipartite graph can be used to match job seekers to job openings based on their skills and qualifications. Each job seeker is connected to the job openings they are qualified for, helping employers find the best candidates and job seekers find suitable positions.
2. **Recommendation Systems**: In online platforms like Netflix or Amazon, bipartite graphs can be used to recommend movies or products to users. One set of nodes represents users, and the other set represents items (movies, products, etc.). Connections between users and items indicate preferences or purchases, allowing the system to suggest items that similar users have liked.
3. **Network Flow Problems**: Bipartite graphs are used in network flow problems to optimize the flow of resources. For example, in transportation networks, they can help determine the most efficient way to route goods from suppliers to consumers, ensuring that supply meets demand.
4. **Biological Networks**: In biology, bipartite graphs can represent interactions between different types of entities, such as proteins and genes. This helps researchers understand how these entities interact and can lead to discoveries about biological processes and disease mechanisms.
5. **Social Networks**: Bipartite graphs can model relationships between two different types of entities in social networks, such as people and events. For example, they can show which people attended which events, helping to analyze social interactions and community structures.

How can we determine if a given graph is a bipartite graph?

Determining if a graph is bipartite can be explained with a simple analogy. Imagine you have a group of people who want to form two teams for a game. The rule is that friends can only be on different teams, not the same team. To figure out if it's possible to divide everyone into two teams following this rule, you can use the following steps:

1. **Start with any person**: Pick any person and assign them to Team A.

2. **Assign their friends to the other team**: Assign all of their friends to Team B.
3. **Continue the process**: For each person in Team B, assign their friends to Team A, and for each person in Team A, assign their friends to Team B.
4. **Check for conflicts**: If at any point you find that a person needs to be assigned to both teams (because they have friends in both teams), then it's not possible to divide the group into two teams following the rule. This means the graph is not bipartite.
5. **No conflicts**: If you can assign everyone to one of the two teams without any conflicts, then the graph is bipartite.

In more technical terms, you can use a method called "graph coloring" where you try to color the graph using two colors. If you can color the graph such that no two adjacent nodes (connected by an edge) have the same color, then the graph is bipartite. If you encounter a situation where two adjacent nodes need to be the same color, then the graph is not bipartite.

In quantum computing, determining if a graph is bipartite can be approached using various quantum algorithms. Here are a few notable ones:

Quantum Walk Algorithms

Quantum walk algorithms are a quantum analog of classical random walk algorithms. They can be used to explore the structure of a graph efficiently. For bipartite graph detection, quantum walks can help identify the two distinct sets of vertices by examining the graph's connectivity and structure.

Quantum Annealing

Quantum annealing is a quantum optimization technique that can be used to solve combinatorial problems, including graph bipartiteness. By mapping the problem onto a quantum system, quantum annealing can find the minimum energy configuration that corresponds to the bipartite structure of the graph.

Grover's Algorithm

Grover's algorithm is a quantum search algorithm that can be used to search for specific properties within a graph. While it is not specifically designed for bipartite graph detection, it can be adapted to search for the bipartite property by checking subsets of vertices and their connections.

Quantum Machine Learning

Quantum machine learning algorithms can be trained to recognize bipartite graphs. By using quantum neural networks or other quantum learning techniques, these algorithms can classify graphs based on their bipartite nature.

Practical Application

These quantum algorithms have practical applications in various fields, such as:

- **Network Analysis**: Identifying bipartite structures in social networks or communication networks.
- **Optimization Problems**: Solving complex optimization problems that involve bipartite graphs.
- **Quantum Computing Research**: Advancing the understanding of quantum algorithms and their applications in graph theory.

DeepSeek solution

DeepSeek was tasked with generating qiskit code to determine if a given graph is bipartite. The initial code did not function correctly. For example, one of the technical issues was that the circuit contained non-Clifford gates, such as multi-controlled phase operations, which were not supported by the stabilizer simulator. After several iterations and adjustments, the code was successfully modified to illustrate the bipartite concept.

To ascertain if a graph is bipartite using a quantum algorithm, the code employs **Grover's algorithm** to identify a valid **bipartition**. The qiskit implementation builds the oracle for verifying bipartitions and utilizes Grover's algorithm to enhance the solution states.

SIDE NOTE: Grover's search algorithm *is a quantum algorithm that helps find a specific item in a large, unsorted list much faster than classical algorithms. Imagine you have a huge box of mixed-up puzzle pieces, and you need to find one particular piece. Normally, you'd have to look at each piece one by one, which could take a long time. Grover's algorithm, however, uses the principles of quantum mechanics to significantly speed up this search process.*

Here's a simple analogy:

1. ***Classical Search****: Imagine you have a deck of cards, and you need to find the Ace of Spades. You'd go through each card one by one until you find it. If there are 52 cards, on average, you'd check about half of them before finding the Ace of Spades.*
2. ***Grover's Algorithm****: Now, imagine you have a magical tool that can look at multiple cards simultaneously and quickly narrow down the possibilities. This tool uses quantum mechanics to amplify the probability of finding the Ace of Spades, allowing you to find it much faster than by checking each card individually.*

In essence, Grover's algorithm can search through a list of (N) items in about $\sqrt{N}$ *steps, which is a significant improvement over the classical method that requires (N) steps. This makes it incredibly useful for tasks like database searches, optimization problems, and more.*

SIDE NOTE: *A* ***bipartition*** *is a way of dividing something into two distinct parts. Imagine you have a group of people at a party, and you want to split them into two separate groups so that no one in the same group knows each other. This is similar to what a bipartition does in a graph.*

In a graph, a bipartition means dividing the set of nodes (or points) into two groups in such a way that no two nodes within the same group are directly connected by an edge (or line). Instead, all the connections (edges) are between nodes in different groups. This kind of graph is called a bipartite graph.

Think of it like a school dance where you want to pair up boys and girls for dancing. You can divide the students into two groups: boys and girls. The rule is that boys can only dance with girls and vice versa, but boys can't dance with boys, and girls can't dance with girls. This ensures that all the connections (dances) are between the two groups.

***SIDE NOTE:** Imagine a **quantum circuit** as a recipe for making a special dish, but instead of using ingredients like flour and sugar, you're using quantum bits (qubits) and quantum gates.*

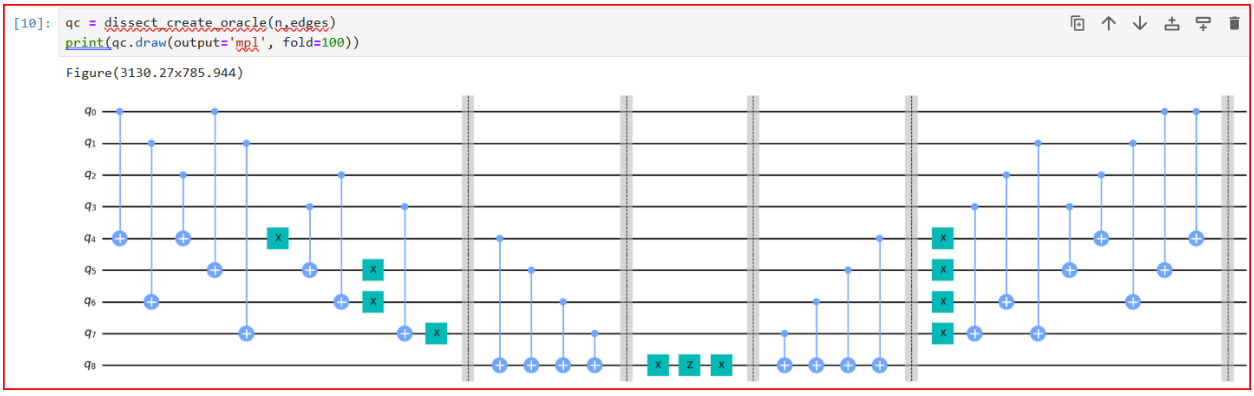
Here's a simple analogy:

1. ***Qubits**: Think of qubits as the basic ingredients. In classical computing, we use bits that can be either 0 or 1. Qubits are like magical ingredients that can be both 0 and 1 at the same time, thanks to a property called superposition.*
2. ***Quantum Gates**: These are like the steps in your recipe. Just as you mix, bake, or stir ingredients in cooking, quantum gates manipulate qubits in specific ways. There are different types of gates, each performing a unique operation on the qubits.*
3. ***Quantum Circuit**: This is the complete recipe. It consists of a sequence of quantum gates applied to qubits to achieve a desired outcome. The circuit starts with qubits in a certain state, applies various gates, and ends with a measurement to get the final result.*

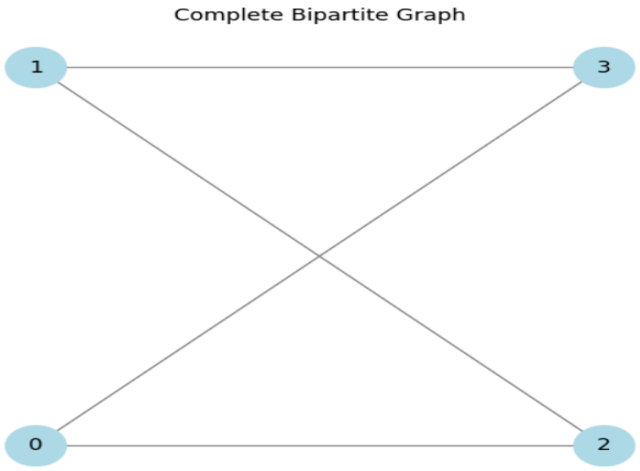
Imagine you're making a smoothie. You start with fruits (qubits), add some milk and yogurt (quantum gates), blend everything together (more gates), and finally pour it into a glass (measurement). The quantum circuit is like the entire process of making the smoothie, from start to finish.

In summary, a quantum circuit is a series of operations performed on qubits to solve a problem or perform a computation, much like following a recipe to make a delicious dish.

To familiarize yourself here is an example of respective Grover's Oracle circuit for Bipartite Problem:



This oracle circuit is built for the following 4-nodes bipartite graph:



Let's now examine the Qiskit code for the oracle circuit. The code is contained within the create_oracle function provided below.

create_oracle function

The *create_oracle* function constructs the quantum oracle that identifies valid bipartitions and returns respective oracle circuit.

```
def create_oracle(n_vertices, edges):
    """Creates quantum oracle for bipartite checking"""
    num_edges = len(edges)
    total_qubits = n_vertices + num_edges + 1
    qc = QuantumCircuit(total_qubits, name="Oracle")

    edge_ancillas = list(range(n_vertices, n_vertices + num_edges))
    flag_qubit = n_vertices + num_edges

    # Check each edge for same-color violation
    for i, (u, v) in enumerate(edges):
        edge_anc = edge_ancillas[i]
        qc.cx(u, edge_anc)
        qc.cx(v, edge_anc)
        qc.x(edge_anc)

    # Aggregate violations using XOR operation
    for anc in edge_ancillas:
        qc.cx(anc, flag_qubit)

    # Apply phase flip when no violations (flag=0)
    qc.x(flag_qubit)
    qc.z(flag_qubit)
    qc.x(flag_qubit)

    # Uncompute operations
    for anc in reversed(edge_ancillas):
        qc.cx(anc, flag_qubit)

    for i, (u, v) in reversed(list(enumerate(edges))):
        edge_anc = edge_ancillas[i]
```

```
    qc.x(edge_anc)
    qc.cx(v, edge_anc)
    qc.cx(u, edge_anc)

return qc
```

Initially, the *create_oracle* function calculates the required number of qubits to represent the graph and the ancilla qubits. We need one ancilla qubit per edge plus one for the overall flag, resulting in a total number of qubits equal to the sum of vertices, edges, and one additional qubit. In the Figure 1 oracle circuit, qubits q0 to q3 represent the graph, q4 to q7 serve as ancilla qubits, and q8 is designated as the overall flag qubit.

*SIDE NOTE: In simple terms, an **ancilla qubit** is like a helper or assistant in a quantum computer.*

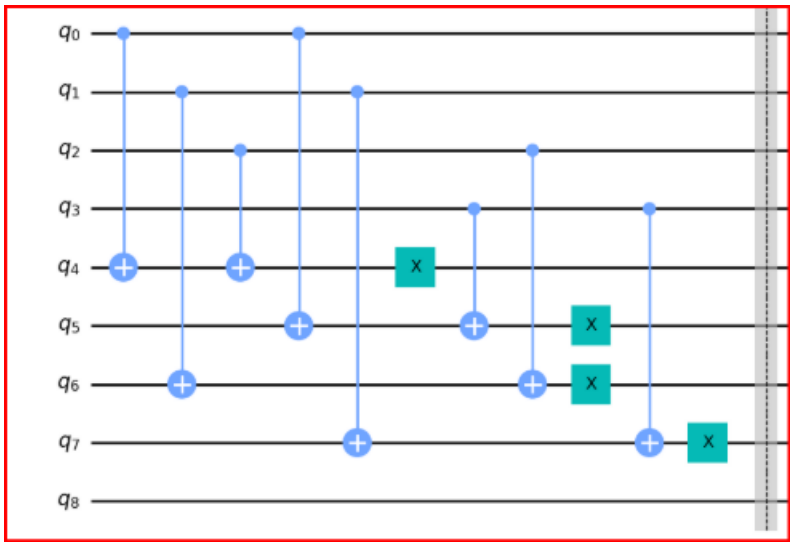
Imagine you have a main task to do, but you need an extra hand to help you complete it. The ancilla qubit is that extra hand. It's not the main focus of the computation, but it helps the main qubits (the ones doing the primary work) to perform their tasks more efficiently.

Ancilla qubits are often used for things like error correction, where they help detect and fix mistakes that might happen during the computation. They can also be used to prepare certain states or to assist in more complex operations.

So, think of ancilla qubits as the helpful assistants that make sure everything runs smoothly in a quantum computer.

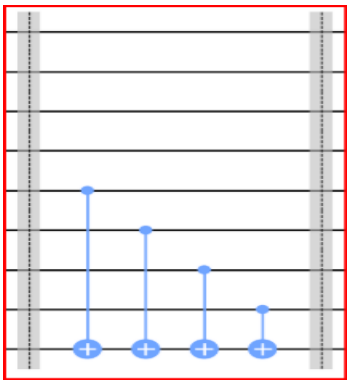
Next, the function checks each edge for same-color violation. For each edge (u,v) it marks edge ancilla if u and v have the same color (violation):

```
# Check each edge for same-color violation
for i, (u, v) in enumerate(edges):
    edge_anc = edge_ancillas[i]
    qc.cx(u, edge_anc)
    qc.cx(v, edge_anc)
    qc.x(edge_anc)
```



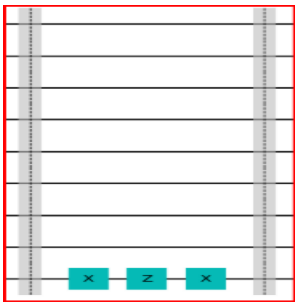
The flag qubit is set to 1 if any edge violates bipartition.

```
# Aggregate violations using XOR operation
for anc in edge_ancillas:
    qc.cx(anc, flag_qubit)
```



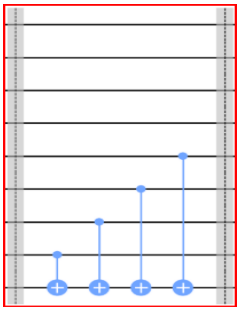
A phase flip (-1) is applied to the overall flag qubit only when there are no violations, meaning that the overall flag qubit is 0:

```
# Apply phase flip when no violations (flag=0)
qc.x(flag_qubit)
qc.z(flag_qubit)
qc.x(flag_qubit)
```

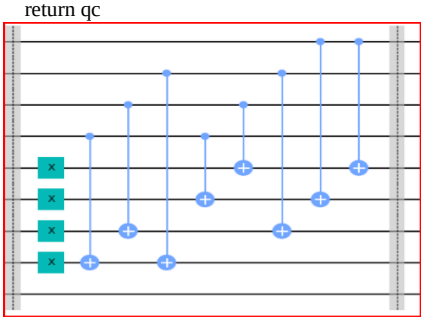


Finally, the function performs uncompute operations and returns the corresponding oracle circuit (qc).

```
# Uncompute operations
for anc in reversed(edge_ancillas):
    qc.cx(anc, flag_qubit)
```



```
for i, (u, v) in reversed(list(enumerate(edges))):
    edge_anc = edge_ancillas[i]
    qc.x(edge_anc)
    qc.cx(v, edge_anc)
    qc.cx(u, edge_anc)

return qc
```



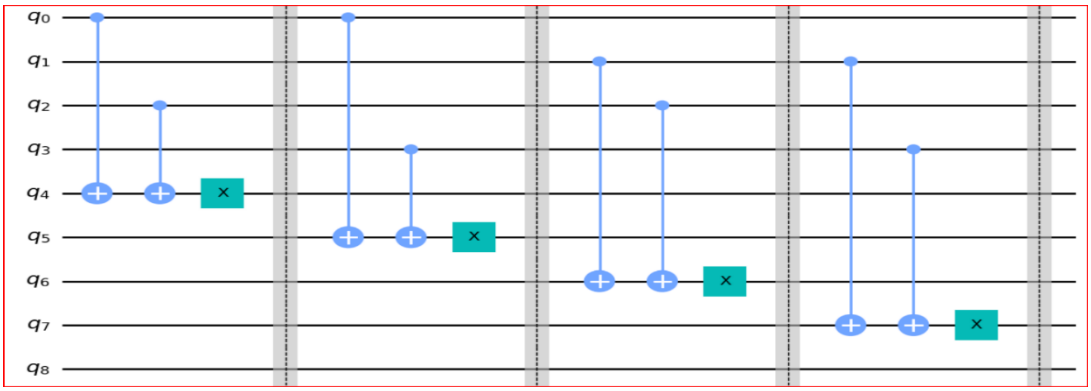
When we use ancilla qubits in a quantum circuit, **uncompute operations** play a crucial role in ensuring that the ancilla qubits are returned to their original state after they have assisted in the computation. This is important for several reasons:

1. **Error Correction**: Ancilla qubits are often used to detect and correct errors in the main qubits. After they have done their job, uncomputing ensures that they are reset and ready to be used again for further error detection.
2. **State Preparation**: Sometimes, ancilla qubits are used to prepare specific quantum states that are needed for certain operations. Uncomputing helps to clean up the ancilla qubits after they have been used, so they do not interfere with subsequent operations.
3. **Resource Management**: Quantum resources are precious and limited. By uncomputing, we ensure that ancilla qubits are not left in an entangled or altered state, which could affect the accuracy and efficiency of the quantum circuit.

4. **Reversibility**: Quantum computations are inherently reversible. Uncomputing is a way to reverse the operations performed on the ancilla qubits, ensuring that the overall computation remains consistent and accurate.

In summary, uncompute operations help to reset the ancilla qubits to their original state, ensuring that they do not introduce errors or interfere with the main computation, and allowing them to be reused efficiently.

When the function checks each edge for same-color violations, it requires two CNOT gates and one X gate per edge to effectively encode the task. The control qubits are the vertices of the edge, such as u and v, while the respective ancilla qubit serves as the target qubit for both CNOT gates.

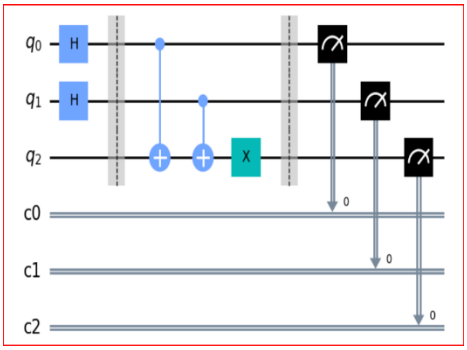


Using our graph from Figure 2 we see that

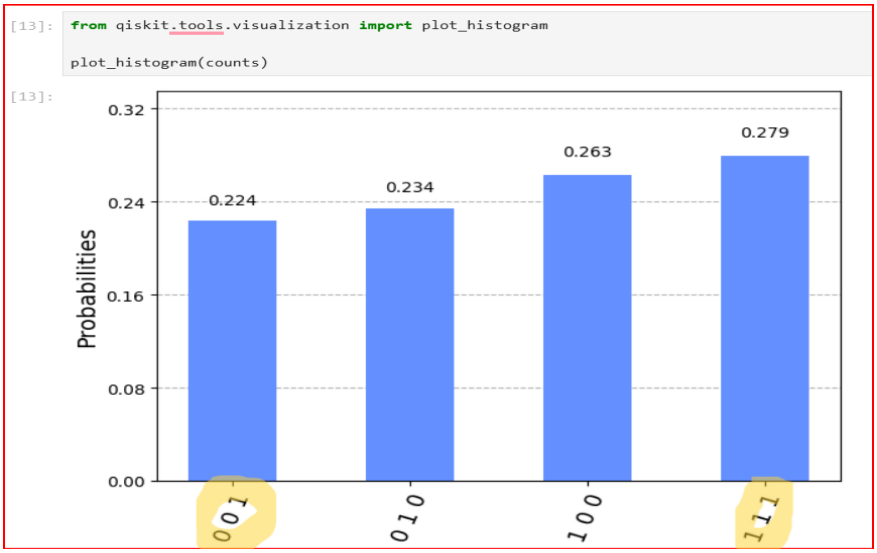
```
[14]: #edges
      for i, (u, v) in enumerate(edges):
          print(f"edge# {i} edge ({u}, {v})")

      edge# 0 edge (0, 2)
      edge# 1 edge (0, 3)
      edge# 2 edge (1, 2)
      edge# 3 edge (1, 3)
```

(0, 2) edge is encoded as CNOT(0,4) and CNOT(2,4). Then X gate as added to qubit 4. The same pattern is repeated for all other edges.

Let's confirm that this specific combination of CNOT, CNOT, and X gates effectively checks for same-color violations. Let's examine this simplified circuit.



The circuit simulates the (0, 1) edge of a graph, with qubits q0 and q1 encoding this edge. Hadamard gates are employed to generate all possible combinations of q0 and q1 input states. As expected, the q2 qubit, serving as an ancilla qubit, is set to 1 only when both q0 and q1 qubits are either 0 or 1, indicating a same-color violation.



Let's prove this mathematically as well. Our 3-qubit input state after 2 Hadamard gates is

$$|q_0q_1q_2 \geq \frac{1}{\sqrt{2}}(|0>+|1>)\frac{1}{\sqrt{2}}(|0>+|1>)|0>=$$

$$=\frac{1}{2}(|000>+|010>+|100>+|110>).$$

After the first CNOT, the state is, remembering that a CNOT gate flips the target qubit only if the control qubit is 1,

$$|q_0q_1q_2>_{CNOT1}=\frac{1}{2}(|000>+|010>+|101>+|111>).$$

Then after the second CNOT we have

$$|q_0q_1q_2>_{CNOT1-CNOT2}=\frac{1}{2}(|000>+|011>+|101>+|110>).$$

The final state, after the X gate, is

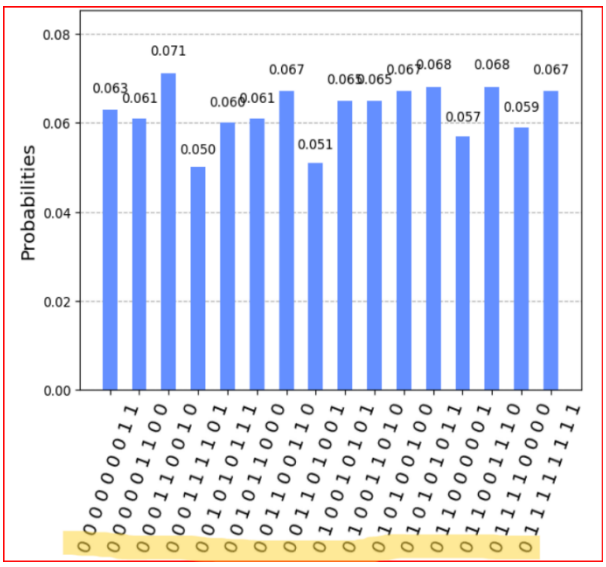
$$|q_0 q_1 q_2 >_{CNOT1-CNOT2-X} = \frac{1}{2}(|001> + |010> + |100> + |111>).$$

The q2 qubit, acting as an ancilla qubit, is set to 1 only when both q0 and q1 qubits are either 0 or 1, indicating a same-color violation. So, the simplified circuit essentially implements XNOR (Exclusive NOR) operation.

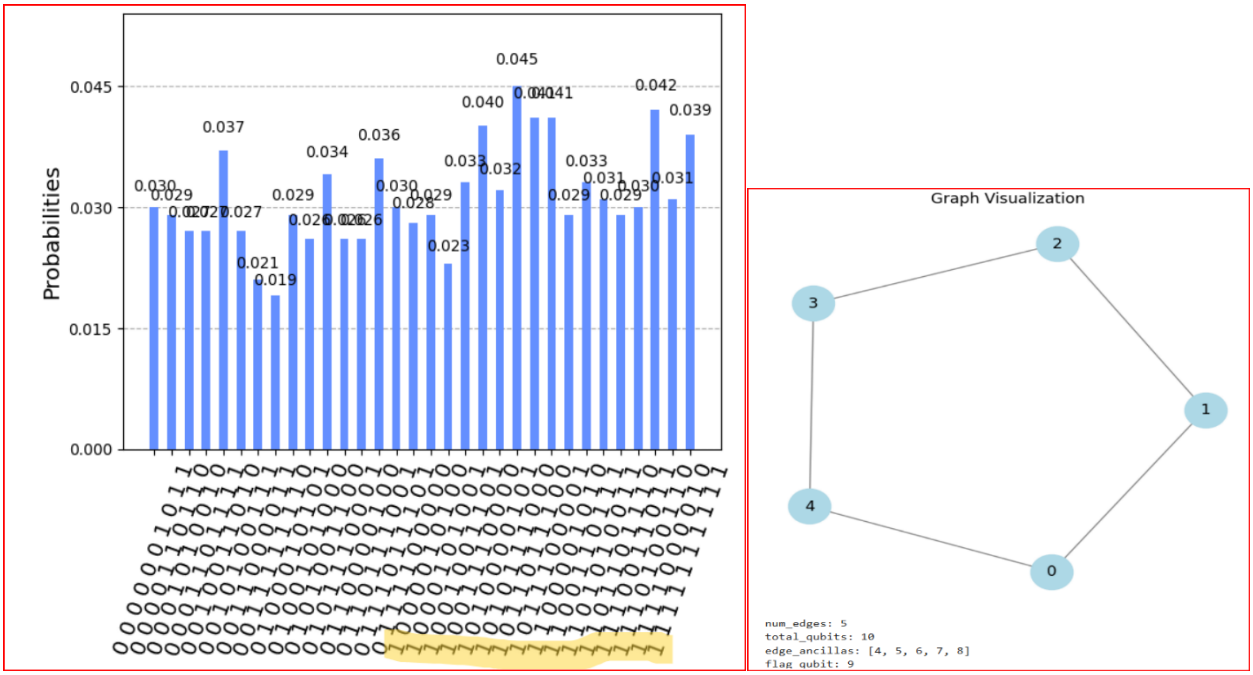
A	B	A ⊙ B
0	0	1
0	1	0
1	0	0
1	1	1

The XNOR gate outputs true (1) only when the inputs are the same. If the inputs are different, the output is false (0).

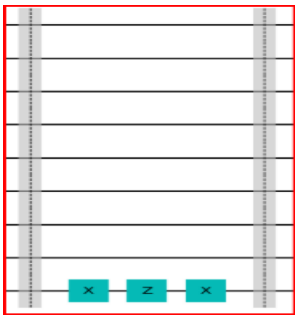
The overall flag qubit is set to 1 if any edge violates bipartition. For our graph from Figure 2 that is not the case, as expected, since the graph is bipartite.



But, if a graph is not bipartite, the flag is set to 1. For example, this graph is not bipartite:



Let's check now a phase flip (-1) is applied to the overall flag qubit only when there are no violations, meaning that the overall flag qubit is 0:



We have

$$Z|0>=|0>$$

$$Z|1>=-|1>$$

$$X|0>=|1>$$

$$X|1>=|0>.$$

Thus, as expected -1 phase is applied only to |0> overall flag stats:

$$XZX|0>=XZ|1>=-X|1>=-|0>$$

$$XZX|1>=XZ|0>=X|0>=|1>.$$

This concludes our review of the *create_oracle* function.

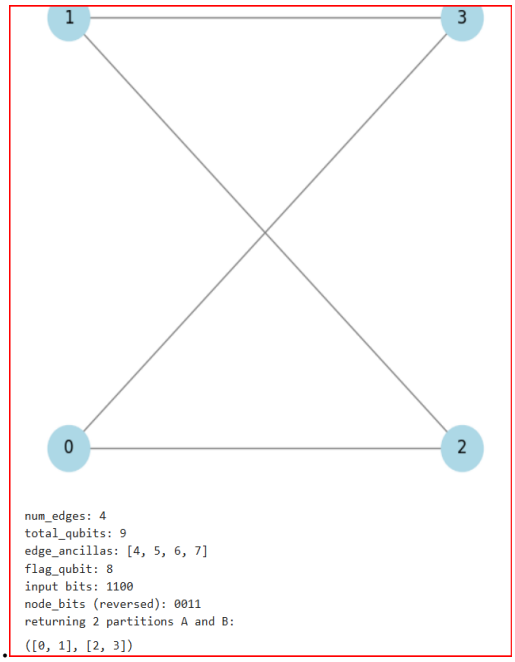
Let's now examine the remaining functions.

get_partitions

```
def get_partitions(bitstring, n_vertices):
    """Extracts partitions from measurement result"""
    reversed_bits = bitstring[::-1]
    node_bits = reversed_bits[:n_vertices]
    return (
        [i for i, bit in enumerate(node_bits) if bit == '0'],
        [i for i, bit in enumerate(node_bits) if bit == '1']
    )
```

The function converts a measured bitstring into node partitions. It initially corrects Qiskit's reverse bit-ordering by reversing the bits. The function then generates two partitions: partition A (nodes with bit=0) and partition B (nodes with bit=1). For instance, in

our 4-node bipartite graph, these partitions are obtained.



is_valid

```
def is_valid(bitstring, edges, n_vertices):
    """Validates bipartition against edges"""
    set_a, set_b = get_partitions(bitstring, n_vertices)
    return all((u in set_a) != (v in set_a) for u, v in edges)
```

The function classically verifies if a measured bitstring represents a valid bipartition. It uses *get_partitions* to split nodes and checks that all edges connect nodes from different partitions. For our 4-nodes bipartite graph we get:

```
[13]: def is_valid_dissect(bitstring, edges, n_vertices):
          """Validates bipartition against edges"""
          set_a, set_b = get_partitions(bitstring, n_vertices)
          print('set_a:',set_a,'set_b:',set_b)
          print("verify if all edges connect nodes from different partitions:")
          return all((u in set_a) != (v in set_a) for u, v in edges)

      is_valid_dissect('1100', edges, n_vertices)

      set_a: [0, 1] set_b: [2, 3]
      verify if all edges connect nodes from different partitions:
[13]: True
```

grover_iterations

```
# For n nodes and M valid solutions:

def grover_iterations(n, M=2):

    N = 2 ** n

    R_raw = (math.pi/4) * math.sqrt(N/M) - 0.5

    return max(0, int(math.floor(R_raw)))
```

To determine the optimal number of Grover iterations for the bipartite graph checking circuit, the function does use the following formula

$$R = \frac{\pi}{4}\sqrt{\frac{N}{M}} - \frac{1}{2}$$

where $N = 2^n$ is the total number of possible bipartitions and n is the number of nodes, $M$is the number of valid bipartitions. For a **connected** bipartite graph, $M = 2$. For a graph with c connected bipartite components, $M = 2^c$.

SIDE NOTE: Imagine you have a school dance where students are divided into two groups: Group A and Group B. Group A consists of students who prefer dancing, while Group B consists of students who prefer watching. To make the dance enjoyable for everyone, you want to ensure that every student in Group A has at least one student in Group B to dance with, and vice versa. This means there are connections (dance partnerships) between students in Group A and students in Group B, but no dance partnerships within the same group.

*A **connected bipartite graph** is like this school dance setup. It's a way to organize things into two groups where every item in one group is connected to at least one item in the other group, and*

there are no connections within the same group. The "connected" part means that you can reach any item from any other item by following the connections, ensuring that the whole setup is linked together.

Example Calculation

For our 4-node connected bipartite graph

$$N = 2^4 = 16, M = 2$$

$$R = \frac{\pi}{4}\sqrt{\frac{16}{2}} - \frac{1}{2} = 1.721 = 1.$$

Thus, 1 Grover iteration is optimal (we did floor 1.721 to avoid over-amplification). Grover iterations balance probability amplification without over-rotation.

SIDE NOTE: Let's use a simple analogy to explain ***Grover iterations****.*

Imagine you have a huge library with thousands of books, and you are looking for one specific book. Normally, you would have to check each book one by one until you find the one you're looking for, which could take a long time.

Now, imagine you have a magical librarian who can help you find the book much faster. This librarian has a special trick: they can narrow down the search by eliminating many books at once, making the search process much quicker.

Grover iterations work like this magical librarian. In the world of quantum computing, Grover's algorithm helps you find a specific item in a large list much faster than traditional methods. It uses a series of steps (iterations) to narrow down the possibilities and quickly zero in on the item you're looking for. Instead of checking each item one by one, Grover's algorithm uses quantum mechanics to speed up the search process, making it much more efficient.

In technical jargon, Grover's algorithm amplifies the probability of measuring valid bipartitions by iteratively rotating the quantum state toward the solution subspace. The formula ensures maximal success probability while minimizing circuit depth.

check_if_bipartite

```
def check_if_bipartite(n,edges):
    # Create quantum components
    oracle = create_oracle(n, edges)
    state_prep = QuantumCircuit(oracle.num_qubits)
    state_prep.h(range(n))  # Initialize superposition
```

```
# Configure Grover's algorithm
iterations = grover_iterations(n, M=2)
print("iterations:",iterations)

grover = Grover(iterations=iterations)
problem = AmplificationProblem(
  oracle=oracle,
  state_preparation=state_prep,
  is_good_state=lambda x: is_valid(x, edges, n)
)

# Construct and measure circuit
circuit = grover.construct_circuit(problem)
circuit.measure_all()

circuit.draw(output='mpl', fold=100)

# Execute on QASM simulator
backend = Aer.get_backend('qasm_simulator')
result = execute(circuit, backend, shots=1000).result()
counts = result.get_counts()

# Analyze results
valid_results = [bs for bs in counts if is_valid(bs, edges, n)]

if valid_results:
    best_result = max(valid_results, key=lambda x: counts[x])
    part_a, part_b = get_partitions(best_result, n)
    print("Bipartite Graph Detected ✅")
    print(f"Partition A ({len(part_a)} nodes): {sorted(part_a)}")
    print(f"Partition B ({len(part_b)} nodes): {sorted(part_b)}")
    print(f"Confidence: {counts[best_result]/1000:.1%}")
else:
    print("❌ Graph is not bipartite")
```

This is the last function to review which for a given graph determines whether the graph is bipartite or not. The input to the function is the number of vertices and a list of respective edges. The

function orchestrates the quantum algorithm and classical post-processing. As you can glance from the above code listing, the function first creates quantum components: respective oracle circuit and prepares the state in equal superposition for determined number of qubits.

`grover = Grover(iterations=iterations)`

- This line creates an instance of the Grover class or function with a specified number of iterations. The variable iterations specifies how many times the algorithm should run to achieve the desired result.

`problem = AmplificationProblem(oracle=oracle, state_preparation=state_prep, is_good_state=lambda x: is_valid(x, edges, n))`

- This line creates an instance of the AmplificationProblem class or function, which is used to define the problem that Grover's algorithm will solve.
- `oracle=oracle`: The oracle is a function or object that marks the correct solutions to the problem.
- `state_preparation=state_prep`: The state_prep is a function or object that prepares the initial quantum state for the algorithm.
- `is_good_state=lambda x: is_valid(x, edges, n):` This lambda function defines the condition for a "good" state, which is a valid solution to the problem. It uses the *is_valid* function with parameters x, edges, and n.

Essentially, these two lines of code set up Grover's algorithm with the necessary parameters and problem definition to perform the quantum search.

`circuit = grover.construct_circuit(problem):`

- This line constructs the quantum circuit for Grover's algorithm based on the defined problem. The problem includes the oracle, state preparation, and the condition for a "good" state. The `grover.construct_circuit(problem)` method creates the necessary quantum gates and operations to perform the search.

`circuit.measure_all():`

- This line adds measurement operations to all qubits in the quantum circuit. Measuring the qubits collapses their quantum states to classical bits, allowing the results of the quantum computation to be read and analyzed.

So these two lines set up the quantum circuit for Grover's algorithm and prepare it for measurement to obtain the results.

Then respective quantum circuit is drawn for a visual presentation. Finally, the remaining code uses a QASM simulator to execute the quantum circuit for Grover's algorithm and analyze the results to determine if a graph is bipartite. Here's a step-by-step explanation:

1. **Execute on QASM simulator**:
   - `backend = Aer.get_backend('qasm_simulator'):` This line sets up the backend for the quantum simulation, specifically using the QASM simulator from the Aer library.

- `result = execute(circuit, backend, shots=1000).result()`: This line executes the quantum circuit (circuit) on the QASM simulator backend, running it for 1000 shots (iterations), and retrieves the results.
- `counts = result.get_counts()`: This line gets the counts of the measurement results from the execution, which represents the frequency of each possible outcome.

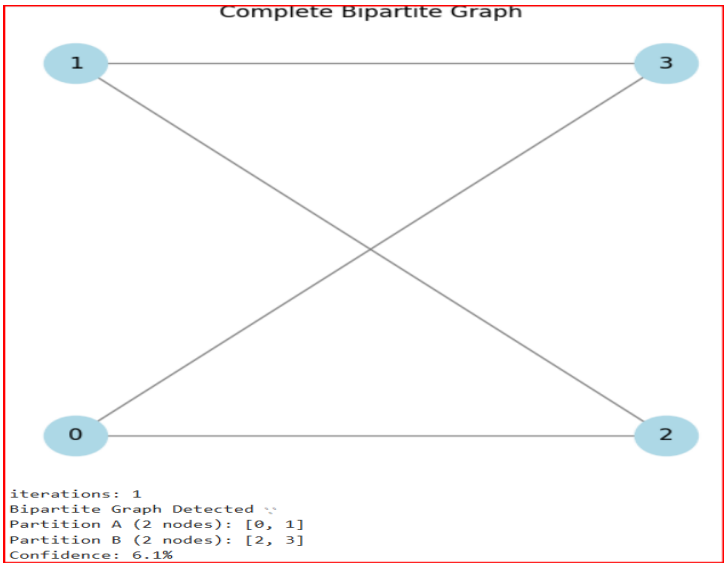
2. **Analyze results**:
   - `valid_results = [bs for bs in counts if is_valid(bs, edges, n)]`: This line filters the measurement results to find valid solutions. It uses the `is_valid` function to check if each bitstring (bs) is a valid solution based on the graph's edges and the number of nodes (n).
3. **Determine if the graph is bipartite**:
   - `if valid_results:`: This conditional statement checks if there are any valid results.
   - `best_result = max(valid_results, key=lambda x: counts[x])`: If there are valid results, this line finds the best result, which is the bitstring with the highest count (most frequent occurrence).
   - `part_a, part_b = get_partitions(best_result, n)`: This line partitions the nodes into two sets (part_a and part_b) based on the best result.
   - `print("Bipartite Graph Detected ")`: This line prints a message indicating that a bipartite graph has been detected.
   - `print(f"Partition A ({len(part_a)} nodes): {sorted(part_a)}")`: This line prints the nodes in partition A.
   - `print(f"Partition B ({len(part_b)} nodes): {sorted(part_b)}")`: This line prints the nodes in partition B.
   - `print(f"Confidence: {counts[best_result]/1000:.1%}")`: This line prints the confidence level of the detection, calculated as the frequency of the best result divided by the total number of shots.
4. **Handle the case where the graph is not bipartite**:
   - `else:`: If there are no valid results, this conditional statement is executed.
   - `print("❌ Graph is not bipartite")`: This line prints a message indicating that the graph is not bipartite.

In summary, the remaining code executes a quantum circuit on a QASM simulator, analyzes the results to find valid solutions, and determines if the graph is bipartite based on the measurement outcomes. If a bipartite graph is detected, it prints the partitions and the confidence level; otherwise, it indicates that the graph is not bipartite.

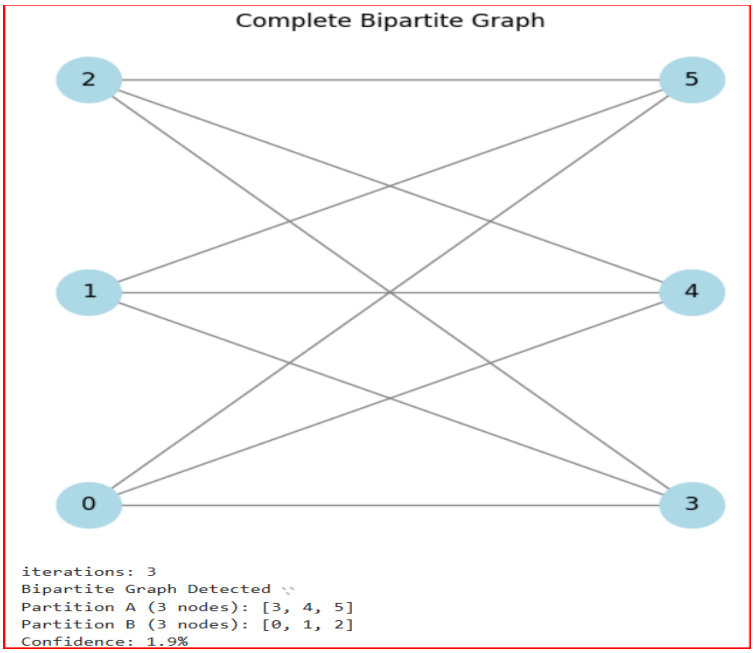
Testing

The above quantum code was tested successfully on a few examples of bipartite graphs listed below, although the confidence level was surprisingly low.

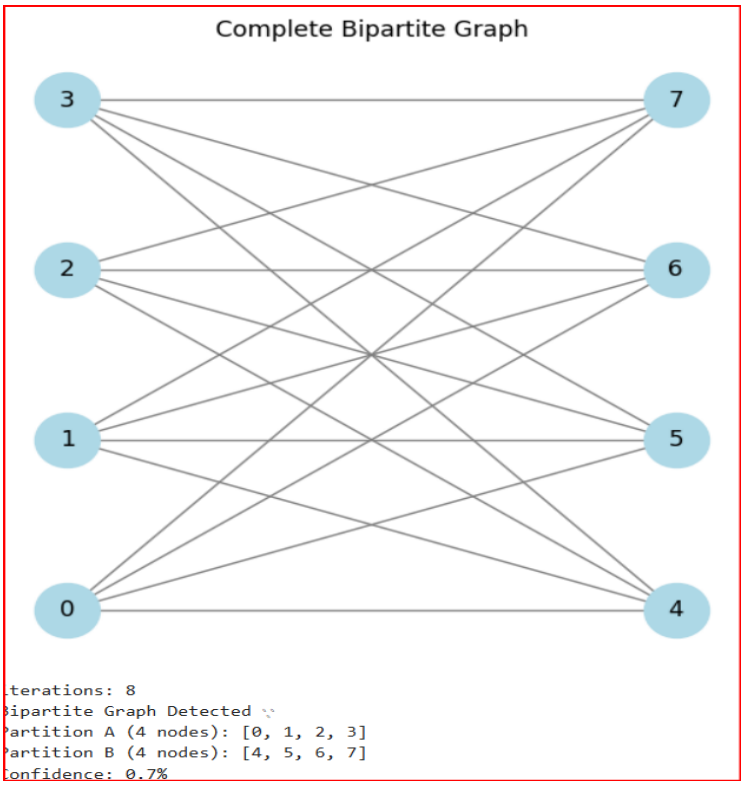
Test 1:



Test 2:

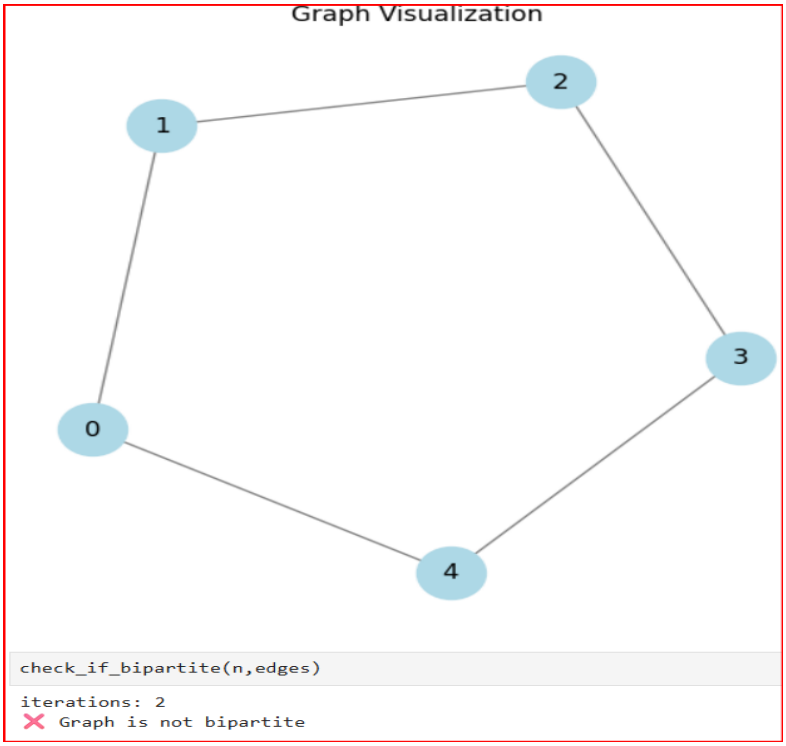


Test 3:



The code also successfully identified that a graph is not bipartite.

Test 4:



Summary:

To determine if a graph is bipartite using a quantum algorithm, the code leverages Grover's algorithm to find a valid bipartition. The Qiskit implementation constructs the oracle for verifying bipartitions and utilizes Grover's algorithm to amplify the solution states. This approach exemplifies a hybrid quantum-classical method, combining quantum search acceleration with classical validation of results. Running the code on actual quantum hardware would require error mitigation for accurate results. Scaling the code would demonstrate Grover's quadratic speedup compared to classical ($O(2^N)$) brute-force methods.

All the code and analysis is in this notebook:
deepseekAndBipartite/DeepSeekAndBipartiteGraphs.ipynb at main · samlip-blip/deepseekAndBipartite

Please feel free to download the notebook and explore.

In the next chapter we focus on the shortest path problem, which involves finding the least costly route between nodes in a graph. It explains Dijkstra's Algorithm, a well-known method for solving this problem, and explores alternative approaches using quantum computing, such as quantum walk algorithms and Grover's algorithm.

The Shortest Path In a Graph

Imagine you have a map with several cities connected by roads. You want to find the quickest route from one city to another. In graph theory, each city is called a "node" and each road is called an "edge." The shortest path is the route that takes the least amount of time or distance to travel from one node to another.

Example

Let's say you have a map with the following cities and roads:

- **City A** to **City B**: 5 miles
- **City A** to **City C**: 10 miles
- **City B** to **City C**: 3 miles
- **City B** to **City D**: 2 miles
- **City C** to **City D**: 4 miles

If you want to travel from **City A** to **City D**, you can take the following routes:

1. **City A** -> **City B** -> **City D** (5 miles + 2 miles = 7 miles)
2. **City A** -> **City C** -> **City D** (10 miles + 4 miles = 14 miles)
3. **City A** -> **City B** -> **City C** -> **City D** (5 miles + 3 miles + 4 miles = 12 miles)

The shortest path is the first route, which is 7 miles.

Practical Application

The concept of the shortest path is used in various real-world applications, such as:

- **Navigation Systems**: GPS devices use algorithms to find the quickest route to your destination.
- **Internet Routing**: Data packets travel through the shortest path in a network to reach their destination quickly.
- **Transportation Planning**: City planners use shortest path algorithms to design efficient public transportation routes.
- **Social Networks**: Finding the shortest path between two users can help in analyzing connections and relationships.

Calculating the shortest path in a graph involves using algorithms designed to find the most efficient route between nodes. One of the most well-known algorithms for this purpose is **Dijkstra's Algorithm**. Here's a simplified explanation of how it works:

Dijkstra's Algorithm

1. **Initialization**: Start by setting the distance to the starting node to 0 and the distance to all other nodes to infinity. Mark all nodes as unvisited. Set the starting node as the current node.
2. **Visit Neighbors**: For the current node, consider all its unvisited neighbors. Calculate the tentative distance to each neighbor through the current node. If this tentative distance is less than the known distance, update the shortest distance.
3. **Mark as Visited**: Once all neighbors of the current node have been considered, mark the current node as visited. A visited node will not be checked again.
4. **Select the Next Node**: Select the unvisited node with the smallest tentative distance and set it as the new current node.
5. **Repeat**: Repeat steps 2-4 until all nodes have been visited or the shortest path to the target node has been determined.
6. **Path Reconstruction**: To find the actual shortest path, backtrack from the target node to the starting node using the recorded shortest distances.

Example

Let's use the same example as before with cities and roads:

- **City A** to **City B**: 5 miles
- **City A** to **City C**: 10 miles
- **City B** to **City C**: 3 miles
- **City B** to **City D**: 2 miles
- **City C** to **City D**: 4 miles

If you want to find the shortest path from **City A** to **City D**:

1. Start at **City A** with a distance of 0.
2. Visit neighbors: **City B** (5 miles) and **City C** (10 miles).
3. Mark **City A** as visited.
4. Select **City B** (5 miles) as the next node.
5. Visit neighbors: **City C** (5 + 3 = 8 miles) and **City D** (5 + 2 = 7 miles).
6. Mark **City B** as visited.

7. Select **City D** (7 miles) as the next node.
8. Since **City D** is the target, the shortest path is **City A** -> **City B** -> **City D** (7 miles).

In quantum computing, similar to determining if a graph is bipartite, determining the shortest path in a graph can be approached using various quantum algorithms. For example:

Quantum Walk Algorithms

Quantum walk algorithms are a quantum analog of classical random walk algorithms. They can be used to explore the structure of a graph efficiently. For shortest path determination, quantum walks can help identify the most efficient route by examining the graph's connectivity and structure.

Quantum Annealing

Quantum annealing is a quantum optimization technique that can be used to solve combinatorial problems, including shortest path determination. By mapping the problem onto a quantum system, quantum annealing can find the minimum energy configuration that corresponds to the shortest path in the graph.

Grover's Algorithm

Grover's algorithm is a quantum search algorithm that can be used to search for specific properties within a graph. While it is not specifically designed for shortest path determination, it can be adapted to search for the shortest path by checking subsets of vertices and their connections.

Quantum Machine Learning

Quantum machine learning algorithms can be trained to recognize shortest paths in graphs. By using quantum neural networks or other quantum learning techniques, these algorithms can classify graphs based on their shortest path properties.

DeepSeek Solution

The qiskit code produced by DeepSeek was an approach that incorporated the high-level idea of parametrized quantum walks and adaptive tuning. The core idea is to use tunable parameters theta for mixing, and gamma for phase feedback. Essentially, fixed Grover iterations are replaced with adaptive steps.

DeepSeek also generated qiskit code that was pretty much a standard implementation of Grover's search algorithm based on an Oracle and an iterative application of the Oracle and diffusion operator, no surprises here.

For a small 4-nodes weighted sample graph, both codes generated the correct smallest path between specified nodes.

For a 10-node weighted graph, some incorrect smallest paths for both codes prompted some code tweaks and updates to resolve these issues. However, for the same 10-nodes weighted graphs, there

were some start and end node pairs for which both updated codes could not generate the correct smallest path, so further investigation is needed into these issues.

The code that implements the high-level idea of parametrized quantum walks and adaptive tuning for the shortest path in a graph

```
from qiskit import Aer, QuantumCircuit, execute
from qiskit.circuit import Parameter
from qiskit.extensions import UnitaryGate  # For Qiskit 0.18.1
from qiskit.aqua.components.optimizers import COBYLA
from qiskit.aqua import QuantumInstance
import networkx as nx
import numpy as np
from math import pi
import matplotlib.pyplot as plt


class QuantumShortestPathSolver:
    def __init__(self, graph, start, end, add_additional_phase=False, additional_phase=0):
        self.graph = graph
        self.start = start
        self.end = end
        self.num_nodes = len(graph.nodes())
        self.node_map = {node: idx for idx, node in enumerate(graph.nodes())}
        self.reverse_map = {idx: node for node, idx in self.node_map.items()}
        self.add_additional_phase=add_additional_phase
        self.additional_phase=additional_phase

        # Parameters
        self.theta = Parameter('θ')
        self.gamma = Parameter('γ')

        # Create circuit
        self.circuit = QuantumCircuit(self.num_nodes, self.num_nodes)
        self.build_circuit()
```

```
def initialize_state(self):
        """Initialize quantum state with start node"""
        start_idx = self.node_map[self.start]
        self.circuit.x(start_idx)
        self.circuit.h(range(self.num_nodes))


    def quantum_walk_step(self):
        """Implement a quantum walk step with parameters"""
        # Mixing operator
        for i in range(self.num_nodes):
            self.circuit.ry(self.theta, i)


        # Mark end node
        end_idx = self.node_map[self.end]
        self.circuit.z(end_idx)


    def apply_phase_feedback(self):
        """Apply parameterized phase feedback"""
        # Create the 2x2 unitary matrix that applies the phase shift on |1>
        phase_matrix = np.array([[1, 0],[0, np.exp(1j * self.additional_phase)]])
        # Create the custom UnitaryGate
        phase_gate = UnitaryGate(phase_matrix, label=f"Phase({self.additional_phase:.2f})")
        for i in range(self.num_nodes):
            self.circuit.rz(self.gamma, i)
            if self.add_additional_phase == True:
                self.circuit.append(phase_gate, [i])


    def build_circuit(self, iterations=2):
        """Build the complete quantum circuit"""
        self.initialize_state()
        for _ in range(iterations):
            self.quantum_walk_step()
            self.apply_phase_feedback()
        self.circuit.measure(range(self.num_nodes), range(self.num_nodes))
```

```
    def is_valid_path(self, path):
        """Check if path is valid"""
        if not path or path[0] != self.start or path[-1] != self.end:
            return False
        for i in range(len(path)-1):
            if not self.graph.has_edge(path[i], path[i+1]):
                return False
        return True

    def calculate_path_length(self, path):
        """Calculate path length"""
        return sum(self.graph[path[i]][path[i+1]]['weight'] for i in range(len(path)-1))

class ParameterTuner:
    def __init__(self, solver, shots=512):
        self.solver = solver
        self.shots = shots
        self.backend = Aer.get_backend('qasm_simulator')
        self.quantum_instance = QuantumInstance(self.backend, shots=shots)

    def objective_function(self, params):
        """Objective function for optimization"""
        # Convert parameters to dictionary for binding
        param_dict = {self.solver.theta: params[0],
                      self.solver.gamma: params[1]}

        try:
            # Bind parameters
            bound_circuit = self.solver.circuit.bind_parameters(param_dict)

            # Execute
            job = execute(bound_circuit, self.backend, shots=self.shots)
            result = job.result()
            counts = result.get_counts()
```

```
            # Calculate score based on valid paths
            score = 0
            for bitstring, count in counts.items():
                path = [self.solver.reverse_map[i] for i, bit in enumerate(reversed(bitstring)) if bit == '1']
                if self.solver.is_valid_path(path):
                    length = self.solver.calculate_path_length(path)
                    score += (count/self.shots) * (1/length)

            return -score  # Return negative for minimization

        except Exception as e:
            print(f"Error during execution: {e}")
            return float('inf')  # Return worst possible score

    def tune_parameters(self, initial_params=None, maxiter=80):#maxiter=20
        """Parameter tuning using Qiskit 0.18.1 COBYLA"""
        if initial_params is None:
            initial_params = [pi/4, pi/8]  # Default initial values

        # Create optimizer
        optimizer = COBYLA(maxiter=maxiter, tol=0.01)

        # Optimization
        ret = optimizer.optimize(num_vars=2,
                                 objective_function=self.objective_function,
                                 initial_point=initial_params)

        optimal_params = ret[0]
        optimal_value = ret[1]

        print(f"Optimization results:")
        print(f"Best θ: {optimal_params[0]:.4f}")
        print(f"Best γ: {optimal_params[1]:.4f}")
        print(f"Best score: {-optimal_value:.4f}")

        return optimal_params[0], optimal_params[1]
```

```
def run_example(G,start,end,add_additional_phase=False, additional_phase=0):
    # Initialize solver
    solver = QuantumShortestPathSolver(G, start, end, add_additional_phase, additional_phase)

    # Run parameter tuning
    print("Starting parameter tuning...")
    tuner = ParameterTuner(solver, shots=2048) #shots=512
    best_theta, best_gamma = tuner.tune_parameters(maxiter=60) #maxiter=15

    # Solve with tuned parameters
    print("\nRunning with tuned parameters...")
    param_dict = {solver.theta: best_theta, solver.gamma: best_gamma}
    bound_circuit = solver.circuit.bind_parameters(param_dict)
    result = execute(bound_circuit, Aer.get_backend('qasm_simulator'), shots=1024).result()
    counts = result.get_counts()

    # Process results
    solutions = []
    for bitstring, count in counts.items():
        path = [solver.reverse_map[i] for i, bit in enumerate(reversed(bitstring)) if bit == '1']
        if solver.is_valid_path(path):
            length = solver.calculate_path_length(path)
            solutions.append((path, length, count))

    # Sort by length then by count
    solutions.sort(key=lambda x: (x[1], -x[2]))

    # Display results
    if solutions:
        print("\nTop solutions:")
        for i, (path, length, count) in enumerate(solutions[:3], 1):
            print(f"{i}. Path: {path}, Length: {length}, Count: {count}")
    else:
        print("No valid paths found.")
```

```
    # Draw circuit
    print("\nFinal circuit:")
    bound_circuit.draw(output='mpl', fold=100)
    plt.show()

def run_custom_tuning_example(G,start,end,theta, gamma,add_additional_phase=False, additional_phase=0):
    # Initialize solver
    solver = QuantumShortestPathSolver(G, start, end, add_additional_phase, additional_phase)

    # Run parameter tuning
    print("Starting parameter tuning...")
    tuner = ParameterTuner(solver, shots=2048) #shots=512
    best_theta, best_gamma = tuner.tune_parameters([theta, gamma], maxiter=60) #maxiter=15

    # Solve with tuned parameters
    print("\nRunning with tuned parameters...")
    param_dict = {solver.theta: best_theta, solver.gamma: best_gamma}
    bound_circuit = solver.circuit.bind_parameters(param_dict)
    result = execute(bound_circuit, Aer.get_backend('qasm_simulator'), shots=1024).result()
    counts = result.get_counts()

    # Process results
    solutions = []
    for bitstring, count in counts.items():
        path = [solver.reverse_map[i] for i, bit in enumerate(reversed(bitstring)) if bit == '1']
        if solver.is_valid_path(path):
            length = solver.calculate_path_length(path)
            solutions.append((path, length, count))

    # Sort by length then by count
    solutions.sort(key=lambda x: (x[1], -x[2]))
```

```
# Display results
if solutions:
    print("\nTop solutions:")
    for i, (path, length, count) in enumerate(solutions[:3], 1):
        print(f"{i}. Path: {path}, Length: {length}, Count: {count}")
else:
    print("No valid paths found.")

# Draw circuit
print("\nFinal circuit:")
bound_circuit.draw(output='mpl', fold=100)
plt.show()
```

Here's a breakdown of what each part of the code is doing:

Imports

The code imports various libraries:

- `qiskit`: For quantum computing operations.
- `networkx`: For graph operations.
- `numpy`: For numerical operations.
- `math`: For mathematical functions.
- `matplotlib.pyplot`: For plotting.

QuantumShortestPathSolver Class

This class is designed to solve the shortest path problem using quantum computing.

Initialization

The `__init__` method initializes the solver with the graph, start node, and end node. It also sets up parameters for the quantum circuit and maps nodes to indices.

Methods

- `initialize_state`: Initializes the quantum state with the start node.
- `quantum_walk_step`: Implements a quantum walk step with parameters.
- `apply_phase_feedback`: Applies parameterized phase feedback.
- `build_circuit`: Builds the complete quantum circuit with a specified number of iterations.
- `is_valid_path`: Checks if a given path is valid.

- calculate_path_length: Calculates the length of a given path.

ParameterTuner Class

This class is designed to tune the parameters of the quantum circuit to optimize the shortest path solution.

Initialization

The __init__ method initializes the tuner with the solver and sets up the quantum instance.

Methods

- objective_function: Defines the objective function for optimization, which calculates the score based on valid paths.
- tune_parameters: Tunes the parameters using the COBYLA optimizer.

run_example Function

This function runs an example of the quantum shortest path solver.

Steps

1. Initializes the solver with the graph, start node, and end node.
2. Runs parameter tuning to find the best parameters.
3. Solves the shortest path problem with the tuned parameters.
4. Processes and displays the results.
5. Draws the final quantum circuit.

Summary

The code sets up a quantum circuit to solve the shortest path problem in a graph, tunes the parameters using optimization techniques, and executes the circuit to find the shortest path. The results are then processed and displayed.

Let's go through each function in the code and describe what they do:

QuantumShortestPathSolver Class

__init__(self, graph, start, end, add_additional_phase=False, additional_phase=0)

This is the initializer method for the QuantumShortestPathSolver class. It sets up the graph, start node, end node, and initializes parameters for the quantum circuit. It also maps nodes to indices and sets up additional phase parameters if specified.

initialize_state(self)

This method initializes the quantum state with the start node. It sets the start node to the state |1> and applies a Hadamard gate to all nodes to create a superposition.

quantum_walk_step(self)

This method implements a quantum walk step with parameters. It applies a rotation around the Y-axis (RY gate) to each node and marks the end node with a Z gate.

apply_phase_feedback(self)

This method applies parameterized phase feedback. It applies a rotation around the Z-axis (RZ gate) to each node and, if specified, applies an additional custom phase shift using a unitary gate.

build_circuit(self, iterations=2)

This method builds the complete quantum circuit. It initializes the state, performs quantum walk steps, applies phase feedback, and measures the quantum state. The number of iterations can be specified.

is_valid_path(self, path)

This method checks if a given path is valid. It ensures the path starts at the start node, ends at the end node, and all edges in the path exist in the graph.

calculate_path_length(self, path)

This method calculates the length of a given path by summing the weights of the edges in the path.

ParameterTuner Class

__init__(self, solver, shots=512)

This is the initializer method for the ParameterTuner class. It sets up the solver and the quantum instance with the specified number of shots.

objective_function(self, params)

This method defines the objective function for optimization. It binds the parameters to the quantum circuit, executes the circuit, and calculates the score based on valid paths. The score is negative for minimization.

tune_parameters(self, initial_params=None, maxiter=80)

This method tunes the parameters using the COBYLA optimizer. It optimizes the parameters to minimize the objective function and returns the best parameters and score.

run_example(G, start, end, add_additional_phase=False, additional_phase=0)

This function runs an example of the quantum shortest path solver. It initializes the solver, runs parameter tuning, solves the shortest path problem with tuned parameters, processes and displays the results, and draws the final quantum circuit.

run_custom_tuning_example(G, start, end, theta, gamma, add_additional_phase=False, additional_phase=0)

This function runs an example of the quantum shortest path solver with custom initial parameters for tuning. It follows similar steps as run_example but allows specifying initial values for the parameters.

Now, let's explore further.

Qubit are used to represent nodes (1 qubit = 1 node), maintaining compatibility with small graphs. For parametrized quantum walk, tunable parameters theta and gamma are introduced, theta for mixing, gamma for phase feedback.

What is a parametrized quantum walk?

*SIDE NOTE: Let's break down the concept of a **parametrized quantum walk** in simple terms.*

Imagine you are walking on a path, and at each step, you have to decide whether to go left or right. In a classical walk, you might flip a coin to make this decision. However, in a quantum walk, you use the principles of quantum mechanics to make these decisions, which means you can be in a superposition of going both left and right at the same time.

Now, a parametrized quantum walk adds another layer of complexity. It involves adjusting certain parameters that influence the probabilities of taking different paths. These parameters can be tuned to control the behavior of the quantum walk, making it more flexible and adaptable to different situations.

Example:

Let's say you are trying to find the shortest path through a maze. In a classical walk, you would explore each possible path one by one, which could take a long time. In a quantum walk, you can explore multiple paths simultaneously, speeding up the process. By adjusting the parameters of the quantum walk, you can optimize the search process, making it even faster and more efficient.

In summary, a parametrized quantum walk is like a supercharged version of a quantum walk, where you can fine-tune the parameters to achieve better results in various applications, such as searching, optimization, and problem-solving.

Let's consider now an example of a parametrized quantum walk to find the shortest path between two nodes in a graph. Imagine you have a graph with several nodes connected by edges, and you want to find the shortest path between node A and node B. In a classical approach, you might use algorithms like Dijkstra's to explore each possible path one by one, which can be time-consuming.

In a quantum walk, you can leverage the principles of quantum mechanics to explore multiple paths simultaneously. Here's how it works:

1. **Initialization**: You start by initializing the quantum state to represent all possible paths from node A to node B. This state is a superposition of all paths, meaning it contains information about every possible route. As you probably guessed, to create a superposition of all paths, we need to apply Hadamard (H) gate to each qubit (1 qubit = 1 node).
2. **Parametrization**: You introduce parameters that influence the probabilities of taking different paths. These parameters can be adjusted to control the behavior of the quantum

walk. For example, you might set parameters to favor shorter paths or paths with fewer obstacles.

3. **Quantum Walk**: The quantum walk begins, and the quantum state evolves according to the parameters you set. The walk explores multiple paths simultaneously, with the probabilities of each path being influenced by the parameters.
4. **Measurement**: After a certain number of steps, you measure the quantum state to determine the most likely path. The measurement collapses the superposition into a single path, which is the shortest path between node A and node B.

By adjusting the parameters, you can optimize the quantum walk to find the shortest path more efficiently than classical algorithms. This approach can be particularly useful in complex graphs with many nodes and edges, where classical methods might struggle.

In summary, a parametrized quantum walk allows you to fine-tune the parameters to achieve better results in finding the shortest path between two nodes, leveraging the power of quantum mechanics to explore multiple paths simultaneously and optimize the search process.

Let's dive into the parameters theta (for mixing) and gamma (for phase feedback) used in a parametrized quantum walk to find the shortest path between two nodes in a graph.

theta (Mixing Parameter):

The mixing parameter theta is used to control the probabilities of transitioning between different nodes in the graph. In a quantum walk, the state of the system can be in a superposition of multiple paths. The mixing parameter theta helps to adjust the balance between these paths, influencing how the quantum walk explores the graph.

- **Role**: theta determines the amplitude distribution among the possible paths. By tuning theta, you can control how much the quantum walk favors certain paths over others.
- **Effect**: A well-chosen theta can enhance the probability of finding the shortest path by optimizing the exploration process. It ensures that the quantum walk does not get stuck in local minima and can efficiently explore the graph.

gamma (Phase Feedback Parameter):

The phase feedback parameter gamma is used to adjust the phase of the quantum state during the walk. In quantum mechanics, the phase of a state can influence the interference patterns, which are crucial for the behavior of the quantum walk.

- **Role**: gamma controls the phase shifts applied to the quantum state as it evolves. These phase shifts can enhance constructive interference for desirable paths and destructive interference for less optimal paths.
- **Effect**: By tuning gamma, you can optimize the interference patterns to favor the shortest path. Proper phase feedback can amplify the probability of the quantum walk collapsing to the shortest path upon measurement.

Example:

Imagine you have a graph with nodes A, B, C, and D, and you want to find the shortest path from A to D. By adjusting theta and gamma, you can influence the quantum walk to explore the paths more efficiently:

1. **Initialization**: Start with a superposition of all possible paths from A to D.
2. **Mixing**: Use theta to adjust the probabilities of transitioning between nodes, favoring shorter paths.
3. **Phase Feedback**: Apply gamma to control the phase shifts, enhancing constructive interference for the shortest path.
4. **Measurement**: After several steps, measure the quantum state to collapse it to the most likely shortest path.

In summary, theta and gamma are crucial parameters that help fine-tune the behavior of a parametrized quantum walk, optimizing the search process to find the shortest path between two nodes in a graph.

What is the role COBYLA optimizer in the above code?

COBYLA (Constrained Optimization BY Linear Approximations) is an optimization algorithm that adjusts the parameters theta (for mixing) and gamma (for phase feedback) in a parametrized quantum walk to find the shortest path between two nodes in a graph. Here's how it works:

Iterative Adjustment Process:

1. Initialization: COBYLA starts with initial values for theta and gamma. These values can be chosen randomly or based on prior knowledge.
2. **Objective Function**: Define an objective function that measures the performance of the quantum walk in finding the shortest path. This function could be based on the probability of successfully identifying the shortest path or the efficiency of the walk.
3. Linear Approximation: COBYLA approximates the objective function with a linear model. This approximation helps to simplify the optimization process, making it easier to adjust the parameters.
4. Parameter Adjustment: COBYLA iteratively adjusts theta and gamma to improve the objective function. In each iteration, it finds a new set of parameters that better optimize the function based on the linear approximation.
5. Constraints Handling: COBYLA can handle constraints on the parameters. For example, you might want theta and gamma to stay within certain bounds to ensure the quantum walk remains stable and efficient.

6. Convergence: The algorithm continues to iterate until it converges to a set of parameters that optimize the objective function. This means that theta and gamma are tuned to achieve the best performance in finding the shortest path.

Example:

Imagine you have a graph with nodes A, B, C, and D, and you want to find the shortest path from A to D using a parametrized quantum walk. You can use COBYLA to tune theta and gamma as follows:

1. Objective Function: Define the objective function as the probability of the quantum walk successfully identifying the shortest path from A to D.
2. Initialization: Start with initial values for theta and gamma.
3. Iteration: Use COBYLA to iteratively adjust theta and gamma to improve the probability of finding the shortest path.
4. Constraints: Ensure (\theta) and (\gamma) stay within certain bounds to maintain stability.
5. Convergence: Continue iterating until COBYLA converges to the optimal values of theta and gamma.

By using COBYLA, you can efficiently tune the parameters theta and gamma to optimize the performance of the parametrized quantum walk in finding the shortest path between two nodes in a graph.

What is the Objective Function?

The objective_function evaluates the performance of the quantum walk with given parameters (\theta) and (\gamma). It binds these parameters to the quantum circuit, executes the circuit, and calculates a score based on the probability of finding valid paths and their lengths. The function returns the negative score to facilitate minimization by the optimization algorithm.

Let's break down the objective_function step by step:

Purpose:

The objective_function is designed to optimize the parameters theta and gamma for a parametrized quantum walk. The goal is to find the best values for these parameters to maximize the probability of finding the shortest path between two nodes in a graph.

Steps:

1. **Convert Parameters to Dictionary**:

```
param_dict = {self.solver.theta: params,
        self.solver.gamma: params}
```

This line converts the input parameters params into a dictionary param_dict where params is assigned to self.solver.theta and params is assigned to self.solver.gamma.

2. **Bind Parameters**:

```
bound_circuit = self.solver.circuit.bind_parameters(param_dict)
```

The parameters in param_dict are bound to the quantum circuit self.solver.circuit. This means the circuit is now configured with the specific values of (\theta) and (\gamma).

3. **Execute the Circuit**:

```
job = execute(bound_circuit, self.backend, shots=self.shots)

result = job.result()

counts = result.get_counts()
```

The bound circuit is executed on the quantum backend specified by self.backend, with the number of shots (repetitions) defined by self.shots. The results of the execution are retrieved, and the counts of different measurement outcomes are obtained.

4. **Calculate Score Based on Valid Paths**:

```
score = 0

for bitstring, count in counts.items():

   path = [self.solver.reverse_map[i] for i, bit in enumerate(reversed(bitstring)) if bit == '1']

   if self.solver.is_valid_path(path):

      length = self.solver.calculate_path_length(path)

     score += (count/self.shots) * (1/length)
```

The function iterates over the measurement outcomes (bitstring) and their counts. For each bitstring, it constructs a path by mapping the bits to nodes using self.solver.reverse_map. If the path is valid (checked by self.solver.is_valid_path), the length of the path is calculated using self.solver.calculate_path_length. The score is then updated based on the count of the bitstring and the inverse of the path length.

5. **Return Negative Score for Minimization**:

```
return -score  # Return negative for minimization
```

The function returns the negative of the score because optimization algorithms typically minimize the objective function. By returning the negative score, the algorithm effectively maximizes the original score.

Details about tuning parameters theta and gamma

The tune_parameters function uses the COBYLA optimizer to find the best values for the parameters theta and gamma in a parametrized quantum walk. It starts with initial values, performs optimization, retrieves the optimal parameters and value, prints the results, and returns the optimal parameters. This process helps to maximize the probability of finding the shortest path between two nodes in a graph.

Let's break down the tune_parameters function step by step:

Purpose:

The tune_parameters function is designed to optimize the parameters theta (for mixing) and gamma (for phase feedback) in a parametrized quantum walk using the COBYLA (Constrained Optimization BY Linear Approximations) algorithm. The goal is to find the best values for these parameters to maximize the probability of finding the shortest path between two nodes in a graph.

Steps:

1. **Function Definition and Default Parameters**:

```
def tune_parameters(self, initial_params=None, maxiter=80):
"""Parameter tuning using Qiskit 0.18.1 COBYLA"""
   if initial_params is None:
      initial_params = [pi/4, pi/8]  # Default initial values
```

The function tune_parameters takes two optional arguments: initial_params and maxiter.

If initial_params is not provided, it defaults to $\frac{\pi}{4}$ for theta and $\frac{\pi}{8}$ for gamma.

2. **Create Optimizer**:

```
optimizer = COBYLA(maxiter=maxiter, tol=0.01)
```

An instance of the COBYLA optimizer is created with a maximum number of iterations (maxiter) and a tolerance (tol) of 0.01. The maxiter parameter controls how many iterations the optimizer will perform.

3. **Optimization**:

```
ret = optimizer.optimize(num_vars=2,
                         objective_function=self.objective_function,
                         initial_point=initial_params)
```

The optimize method of the COBYLA optimizer is called with the following arguments:

- `num_vars=2`: The number of variables to optimize (in this case, theta and gamma).
- `objective_function=self.objective_function`: The objective function to be minimized, which evaluates the performance of the quantum walk with given parameters.

- initial_point=initial_params: The initial values for theta and gamma.

4. **Retrieve Optimal Parameters and Value**:

```
optimal_params = ret[0]

optimal_value = ret[1]
```

The results of the optimization are stored in ret, which is a tuple. ret contains the optimal values for theta and gamma and ret contains the optimal value of the objective function.

5. **Print Optimization Results**:

```
print(f"Optimization results:")

print(f"Best θ: {optimal_params:.4f}")

print(f"Best γ: {optimal_params:.4f}")

print(f"Best score: {-optimal_value:.4f}")
```

The function prints the optimization results, including the best values for theta and gamma, and the best score (note that the score is negated to convert it back to a maximization problem).

6. **Return Optimal Parameters**:

```
return optimal_params, optimal_params
```

The function returns the optimal values for theta and gamma.

is_valid_path function step by step

Purpose:

The is_valid_path function is designed to check whether a given path is valid within a graph. It ensures that the path starts and ends at specified nodes and that each consecutive pair of nodes in the path is connected by an edge in the graph.

Steps:

1. Function Definition and Docstring:

```
def is_valid_path(self, path):

  """Check if path is valid"""
```

The function is_valid_path takes a single argument path, which is a list of nodes representing a path in the graph. The docstring provides a brief description of the function's purpose.

2. Initial Validity Checks:

```
if not path or path != self.start or path[-1] != self.end:

  return False
```

The function first checks if the path is empty or if the first node in the path is not equal to self.start or if the last node in the path is not equal to self.end. If any of these conditions are true, the function returns False, indicating that the path is invalid.

3. Edge Validity Checks:

```
for i in range(len(path)-1):
    if not self.graph.has_edge(path[i], path[i+1]):
        return False
```

The function then iterates through the path to check if each consecutive pair of nodes is connected by an edge in the graph. It uses the has_edge method of self.graph to verify the existence of an edge between path[i] and path[i+1]. If any pair of nodes is not connected by an edge, the function returns False, indicating that the path is invalid.

4. Return True for Valid Path:

```
return True
```

If all the checks pass, the function returns True, indicating that the path is valid.

Summary:

The is_valid_path function performs the following checks to determine the validity of a path:

- It ensures the path is not empty and starts at self.start and ends at self.end.
- It verifies that each consecutive pair of nodes in the path is connected by an edge in the graph.

If all these conditions are met, the function returns True, indicating that the path is valid. Otherwise, it returns False.

calculate_path_length step by step

Purpose:

The calculate_path_length function is designed to calculate the total length of a given path in a graph. The length is determined by summing the weights of the edges that make up the path.

Steps:

1. **Function Definition and Docstring**:

```
def calculate_path_length(self, path):
    """Calculate path length"""
```

The function calculate_path_length takes a single argument path, which is a list of nodes representing a path in the graph. The docstring provides a brief description of the function's purpose.

2. **Calculate Path Length**:

```
return sum(self.graph[path[i]][path[i+1]]['weight'] for i in range(len(path)-1))
```

This line of code calculates the total length of the path by summing the weights of the edges between consecutive nodes in the path. Here's a detailed breakdown of how it works:

- `range(len(path)-1)`: This generates a range of indices from 0 to the length of the path minus 1. This range is used to iterate over the nodes in the path.
- `self.graph[path[i]][path[i+1]]['weight']`: For each index i in the range, this expression retrieves the weight of the edge between the node at position i and the node at position i+1 in the path. The self.graph object represents the graph, and self.graph[path[i]][path[i+1]] accesses the edge between the nodes path[i] and path[i+1]. The ['weight'] part retrieves the weight of that edge.
- sum(...): The sum function adds up all the weights of the edges between consecutive nodes in the path.

Example:

Let's say you have a graph with nodes A, B, C, and D, and the following edges with weights:

- A to B: weight 2
- B to C: weight 3
- C to D: weight 4

If the path is [A, B, C, D], the function will calculate the total length as follows:

- Weight of edge A to B: 2
- Weight of edge B to C: 3
- Weight of edge C to D: 4

The total length of the path is (2 + 3 + 4 = 9).

Summary:

The calculate_path_length function calculates the total length of a given path in a graph by summing the weights of the edges between consecutive nodes in the path. It uses a list comprehension to iterate over the nodes in the path and retrieve the weights of the edges and then sums these weights to get the total length.

Quantum Circuit

Let’s now analyze respective quantum circuit.

```
# Create circuit
    self.circuit = QuantumCircuit(self.num_nodes, self.num_nodes)
    self.build_circuit()
```

The first line creates a quantum circuit with self.num_nodes qubits and self.num_nodes classical bits. The quantum circuit is stored in self.circuit. The build_circuit method is then called to construct the complete quantum circuit.

```
def initialize_state(self):
    """Initialize quantum state with start node"""
    start_idx = self.node_map[self.start]
    self.circuit.x(start_idx)
    self.circuit.h(range(self.num_nodes))
```

The initialize_state method initializes the quantum state of the circuit. Here's what happens in detail:

- start_idx = self.node_map[self.start]: This line retrieves the index of the start node from self.node_map, which is a mapping of nodes to their corresponding indices.
- self.circuit.x(start_idx): This applies the X (Pauli-X) gate to the qubit corresponding to the start node. The X gate flips the state of the qubit from $|0\rangle$ to $|1\rangle$.
- self.circuit.h(range(self.num_nodes)): This applies the Hadamard gate to all qubits in the circuit. The Hadamard gate creates a superposition state, allowing the quantum walk to explore multiple paths simultaneously.

```
def build_circuit(self, iterations=2):
    """Build the complete quantum circuit"""
    self.initialize_state()
    for _ in range(iterations):
        self.quantum_walk_step()
        self.apply_phase_feedback()
    self.circuit.measure(range(self.num_nodes), range(self.num_nodes))
```

The build_circuit method constructs the complete quantum circuit. Here's what happens in detail:

- self.initialize_state(): This initializes the quantum state as described above.
- for _ in range(iterations): self.quantum_walk_step(); self.apply_phase_feedback(): This loop performs the quantum walk and applies phase feedback for a specified number of

iterations. The quantum_walk_step method likely implements the steps of the quantum walk, while the apply_phase_feedback method adjusts the phase of the quantum state to optimize the walk.

- self.circuit.measure(range(self.num_nodes), range(self.num_nodes)): This measures all qubits in the circuit and stores the results in the corresponding classical bits. Measurement collapses the quantum state to a classical state, allowing the results to be read.

Summary:

- **Creating the Quantum Circuit**: Initializes a quantum circuit with a specified number of qubits and classical bits.
- **Initializing the Quantum State**: Sets the start node to |1⟩ and applies Hadamard gates to create a superposition state.
- **Building the Complete Quantum Circuit**: Performs the quantum walk and phase feedback for a specified number of iterations, then measures the qubits to obtain the results.

Let's focus on the **quantum_walk_step** function now. The quantum_walk_step function is designed to implement a single step of a quantum walk on a graph. This step involves applying a mixing operator to the qubits in the quantum circuit, which helps to explore different paths in the graph.

Steps:

1. **Function Definition and Docstring**:

```
def quantum_walk_step(self):
"""Implement a quantum walk step with parameters"""
```

The function quantum_walk_step is defined without any parameters. The docstring provides a brief description of the function's purpose, indicating that it implements a quantum walk step with parameters.

2. **Mixing Operator**:

```
for i in range(self.num_nodes):
    self.circuit.ry(self.theta, i)
```

This loop applies a mixing operator to each qubit in the quantum circuit. Here's what happens in detail:

- for i in range(self.num_nodes): This loop iterates over all the qubits in the circuit. self.num_nodes represents the total number of qubits (or nodes) in the graph.

- self.circuit.ry(self.theta, i): This line applies the ry gate (a rotation around the Y-axis) to the qubit at index i. The angle of rotation is specified by self.theta, which is a parameter that can be tuned to optimize the quantum walk.
-

3. The Z gate is used to mark the end node of a graph in a quantum circuit.

```
# Mark end node

    end_idx = self.node_map[self.end]

    self.circuit.z(end_idx)
```

Summary:

The quantum_walk_step function performs the following actions:

- It iterates over all the qubits in the quantum circuit.
- It applies the ry gate to each qubit, with the rotation angle specified by the parameter self.theta.
- The Z gate is used to mark the end node of a graph in a quantum circuit.

The ry gate is a crucial component of the mixing operator in a quantum walk. By rotating the qubits around the Y-axis, the ry gate helps to create superpositions and explore different paths in the graph. This step is essential for the quantum walk to effectively search for the shortest path between nodes.

Role of the ry Gate:

1. **Creating Superpositions**: The ry gate is a rotation gate that rotates the state of a qubit around the Y-axis by a specified angle theta. When applied to a qubit, it creates a superposition of the $|0\rangle$ and $|1\rangle$ states. This superposition is essential for the quantum walk, as it allows the qubit to simultaneously explore multiple paths in the graph.
2. **Controlling Probabilities**: The angle theta used in the ry gate determines the probabilities of the qubit being in the $|0\rangle$ or $|1\rangle$ state. By tuning theta, you can control the balance between these probabilities, influencing how the quantum walk explores different paths. This tuning helps optimize the search process for finding the shortest path.
3. **Mixing Operator**: In the context of a quantum walk, the ry gate acts as a mixing operator. It mixes the amplitudes of the qubit states, ensuring that the quantum walk does not get stuck in local minima and can efficiently explore the graph. The mixing operator is essential for maintaining the coherence and effectiveness of the quantum walk.

Example:

Imagine you have a qubit initially in the |0⟩ state. Applying an ry gate with angle theta will rotate the qubit to a superposition state:

$$ry(\theta)\}0> = \cos\left(\frac{\theta}{2}\right)|0> + \sin\left(\frac{\theta}{2}\right)|1>$$

This superposition allows the quantum walk to explore both the |0⟩ and |1⟩ states simultaneously, increasing the chances of finding the optimal path in the graph.

Role of the Z Gate:

1. **Phase Flip**: The Z gate, also known as the Pauli-Z gate, flips the phase of the qubit state. Specifically, it changes the phase of the |1⟩ state to -|1⟩ while leaving the |0⟩ state unchanged. This phase flip is crucial for distinguishing the end node from other nodes in the graph.
2. **Marking the End Node**: By applying the Z gate to the qubit corresponding to the end node, you effectively mark this node with a unique phase. This marking helps to identify the end node during the quantum walk and ensures that the quantum algorithm can recognize when it has reached the destination.
3. **Interference Patterns**: The phase flip introduced by the Z gate influences the interference patterns in the quantum walk. Proper phase marking can enhance constructive interference for paths leading to the end node and destructive interference for other paths. This optimization helps to increase the probability of the quantum walk collapsing to the shortest path upon measurement.

Example:

Imagine you have a graph with nodes A, B, C, and D, and you want to find the shortest path from A to D. By applying the Z gate to the qubit corresponding to node D, you mark this node with a unique phase. During the quantum walk, this marking helps to identify node D as the destination and optimize the search process.

Summary:

The ry gate is essential for creating superpositions, controlling probabilities, and acting as a mixing operator in the quantum walk step. By rotating the qubits around the Y-axis, it enables the quantum walk to explore multiple paths simultaneously and optimize the search process for finding the shortest path between nodes.

The Z gate is used to mark the end node of a graph by flipping the phase of the corresponding qubit. This phase marking helps to distinguish the end node, optimize interference patterns, and increase the probability of finding the shortest path during the quantum walk.

Let's break down the **apply_phase_feedback** function step by step to understand what it does.

Purpose:

The apply_phase_feedback function is designed to apply parameterized phase feedback to the qubits in a quantum circuit. This phase feedback helps to optimize the quantum walk by adjusting the phases of the qubits based on the parameter gamma.

Steps:

1. **Function Definition and Docstring**:

```
def apply_phase_feedback(self):
    """Apply parameterized phase feedback"""
```

The function apply_phase_feedback is defined without any parameters. The docstring provides a brief description of the function's purpose, indicating that it applies parameterized phase feedback.

2. **Applying Phase Feedback**:

```
for i in range(self.num_nodes):
    self.circuit.rz(self.gamma, i)
```

This loop applies the rz gate (a rotation around the Z-axis) to each qubit in the quantum circuit. Here's what happens in detail:

   - for i in range(self.num_nodes): This loop iterates over all the qubits in the circuit. self.num_nodes represents the total number of qubits (or nodes) in the graph.
   - self.circuit.rz(self.gamma, i): This line applies the rz gate to the qubit at index i. The angle of rotation is specified by self.gamma, which is a parameter that can be tuned to optimize the quantum walk.

Role of the rz Gate:

The rz gate is a rotation gate that rotates the state of a qubit around the Z-axis by a specified angle (gamma. In the context of a quantum walk, the rz gate plays the following roles:

1. **Phase Adjustment**: The rz gate adjusts the phase of the qubit's state. By tuning the angle gamma, you can control the phase shifts applied to the qubits. These phase shifts influence the interference patterns in the quantum walk, which are crucial for optimizing the search process.
2. **Constructive and Destructive Interference**: Proper phase feedback can enhance constructive interference for desirable paths and destructive interference for less optimal paths. This helps to amplify the probability of the quantum walk collapsing to the shortest path upon measurement.

Summary:

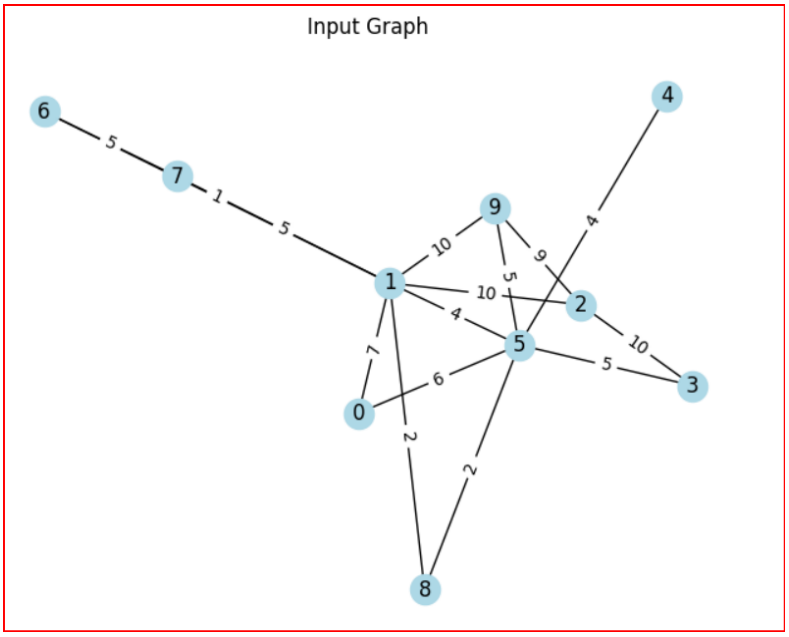
The apply_phase_feedback function performs the following actions:

- It iterates over all the qubits in the quantum circuit.
- It applies the rz gate to each qubit, with the rotation angle specified by the parameter (gamma.

The rz gate is essential for adjusting the phases of the qubits, optimizing the interference patterns, and enhancing the performance of the quantum walk in finding the shortest path between nodes.

Testing

10-node weighted graph:



The shortest path between 1 and 9 nodes is 1-5-9. This is confirmed by Dijkstra's Algorithm.

In [12]:
```python
import heapq

def dijkstra(graph, start, end):
    # Priority queue to store (cost, node)
    queue = [(0, start)]
    # Dictionary to store the shortest path to each node
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    # Dictionary to store the path
    previous_nodes = {node: None for node in graph}

    while queue:
        current_distance, current_node = heapq.heappop(queue)

        # If we reached the end node, reconstruct the path
        if current_node == end:
            path = []
            while previous_nodes[current_node] is not None:
                path.insert(0, current_node)
                current_node = previous_nodes[current_node]
            path.insert(0, start)
            return path, distances[end]

        # If a shorter path to current_node has been found, skip processing
        if current_distance > distances[current_node]:
            continue

        # Explore neighbors
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_nodes[neighbor] = current_node
                heapq.heappush(queue, (distance, neighbor))

    return None, float('inf')
```

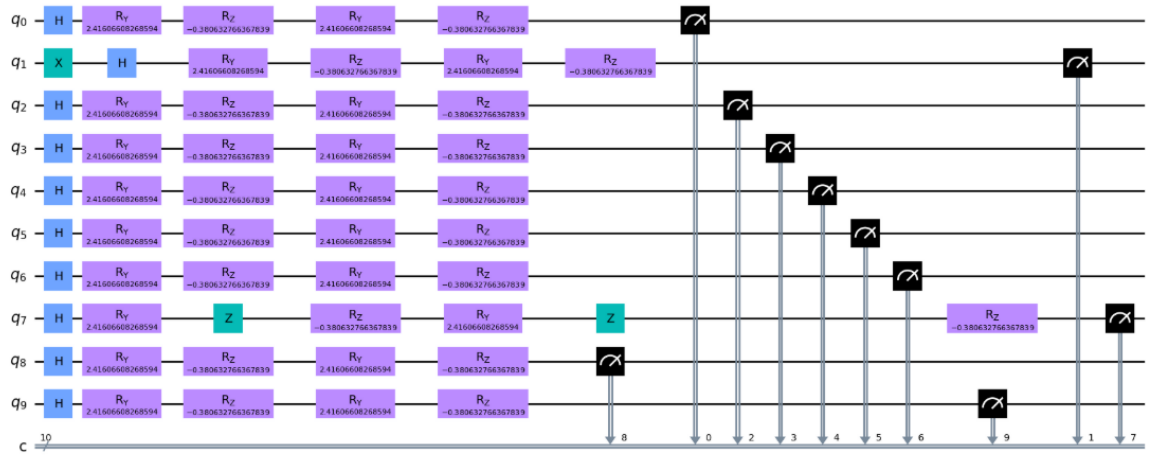
In [13]:
```python
start_node = '1'
end_node = '9'
path, cost = dijkstra(graph_dict, start_node, end_node)

if path:
    print(f"The shortest path from {start_node} to {end_node} is: {' -> '.join(path)} with a cost of {cost}")
else:
    print(f"No path found from {start_node} to {end_node}")
```

```
The shortest path from 1 to 9 is: 1 -> 5 -> 9 with a cost of 9
```

The above quantum code did return the same path.

```
Optimization results:
Best θ: 2.4161
Best γ: -0.3806
Best score: 0.0425

Running with tuned parameters...

Top solutions:
1. Path: [1, 5, 9], Length: 9, Count: 8
2. Path: [1, 9], Length: 10, Count: 396
3. Path: [1, 2, 9], Length: 19, Count: 5
```

```
Final circuit:
C:\Users\slipo\anaconda3\envs\old_qiskit\lib\site-packages\qiskit\circuit\tools\pi_check.py:49: SymPyDeprecationWarning:

The expr_free_symbols property is deprecated. Use free_symbols to get
the free symbols of an expression.

See https://docs.sympy.org/latest/explanation/active-deprecations.html#deprecated-expr-free-symbols
for details.

This has been deprecated since SymPy version 1.9. It
will be removed in a future version of SymPy.

  if not hasattr(inpt._symbol_expr, "expr_free_symbols"):
```



Another example for the same 10-node graph, the shortest path between 0 and 8 nodes is 0-5-8

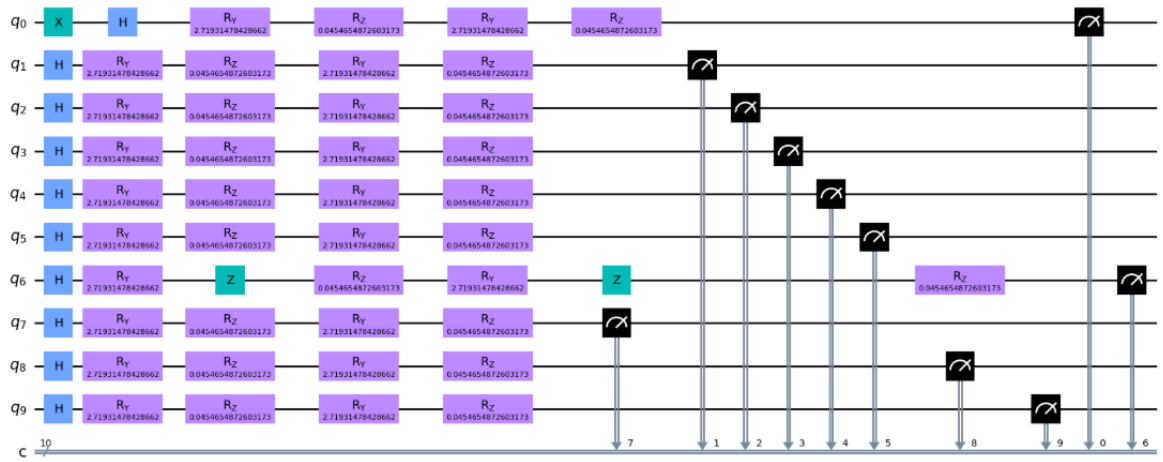
```
start_node = '0'
end_node = '8'
path, cost = dijkstra(graph_dict, start_node, end_node)

if path:
    print(f"The shortest path from {start_node} to {end_node} is: {' -> '.join(path)} with a cost of {cost}")
else:
    print(f"No path found from {start_node} to {end_node}")
```

```
The shortest path from 0 to 8 is: 0 -> 5 -> 8 with a cost of 8
```

which is confirmed by the above quantum code:

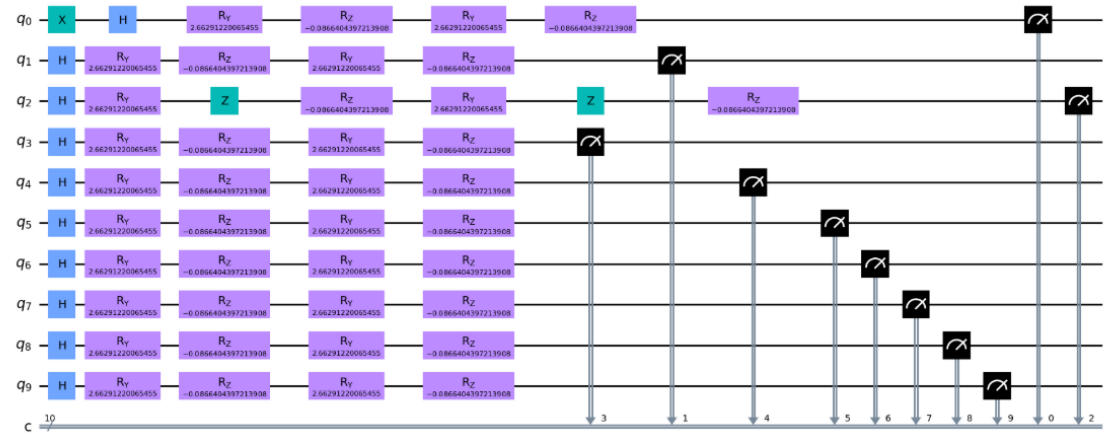
```
Starting parameter tuning...
Optimization results:
Best θ: 2.7193
Best γ: 0.0455
Best score: 0.0051

Running with tuned parameters...

Top solutions:
1. Path: [0, 5, 8], Length: 8, Count: 22
2. Path: [0, 1, 8], Length: 9, Count: 24
3. Path: [0, 1, 5, 8], Length: 13, Count: 2
```

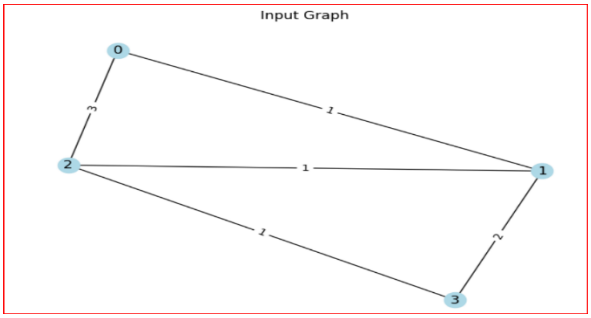
```
Final circuit:
C:\Users\slipo\anaconda3\envs\old_qiskit\lib\site-packages\qiskit\circuit\tools\pi_check.py:49: SymPyDeprecationWarning:

The expr_free_symbols property is deprecated. Use free_symbols to get
the free symbols of an expression.

See https://docs.sympy.org/latest/explanation/active-deprecations.html#deprecated-expr-free-symbols
for details.

This has been deprecated since SymPy version 1.9. It
will be removed in a future version of SymPy.

  if not hasattr(inpt._symbol_expr, "expr_free_symbols"):
```



2 more examples.

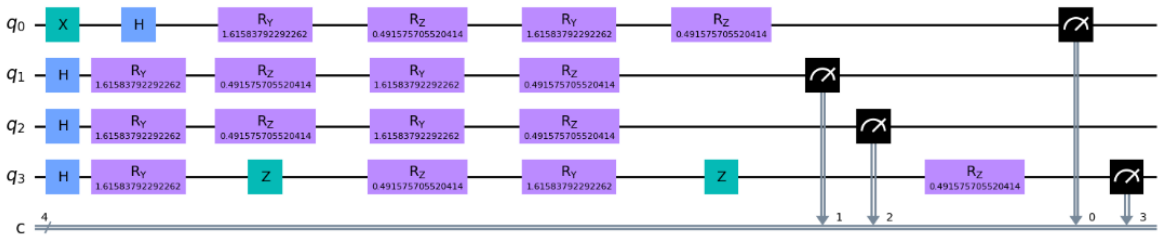
```
In [26]: run_example(G, 0, 2)

Starting parameter tuning...
Optimization results:
Best θ: 2.6629
Best γ: -0.0866
Best score: 0.0013

Running with tuned parameters...

Top solutions:
1. Path: [0, 1, 2], Length: 17, Count: 24

Final circuit:
```





The shortest path between 0 and 3 nodes is 1-3:

```
Optimization results:
Best θ: 1.6158
Best γ: 0.4916
Best score: 0.0598

Running with tuned parameters...

Top solutions:
1. Path: [0, 1, 3], Length: 3, Count: 67
2. Path: [0, 1, 2, 3], Length: 3, Count: 63
3. Path: [0, 2, 3], Length: 4, Count: 72

Final circuit:
C:\Users\slipo\anaconda3\envs\old_qiskit\lib\site-packages\qiskit\circuit\tools\pi_check.py:49: SymPyDeprecationWarning:

The expr_free_symbols property is deprecated. Use free_symbols to get
the free symbols of an expression.

See https://docs.sympy.org/latest/explanation/active-deprecations.html#deprecated-expr-free-symbols
for details.

This has been deprecated since SymPy version 1.9. It
will be removed in a future version of SymPy.

  if not hasattr(inpt._symbol_expr, "expr_free_symbols"):
```



The provided Qiskit code for solving the shortest path problem using quantum computing has several limitations:

1. **Scalability**: The code is designed to work with a quantum circuit whose size is proportional to the number of nodes in the graph. As the number of nodes increases, the quantum circuit becomes larger and more complex, making it difficult to scale to larger graphs.

2. **Noise and Errors**: Quantum computers are prone to noise and errors, which can affect the accuracy of the results. The code uses a quantum simulator (qasm_simulator), which does not account for the noise present in real quantum hardware. Therefore, the results obtained from the simulator may not be representative of those obtained from actual quantum hardware.

3. **Parameter Optimization**: The code uses the COBYLA optimizer to tune the parameters of the quantum circuit. While COBYLA is a popular choice for optimization, it may not always find the global optimum, especially in the presence of noise and errors. Additionally, the optimization process can be time-consuming, particularly for larger graphs.
4. **Initialization**: The code initializes the quantum state by setting the start node to the $|1\rangle$ state and applying a Hadamard gate to all qubits. This initialization may not be optimal for all graph structures and could affect the performance of the algorithm.
5. **Phase Feedback**: The code includes an option to apply an additional phase shift to the quantum circuit. While this can be useful for certain problems, it adds complexity to the circuit and may not always lead to improved results. The effectiveness of the additional phase shift depends on the specific graph and problem instance.
6. **Classical Post-Processing**: After executing the quantum circuit, the code performs classical post-processing to interpret the measurement results and identify valid paths. This step can be computationally expensive, especially for large graphs with many possible paths.
7. **Graph Representation**: The code uses the NetworkX library to represent the graph. While NetworkX is a powerful and flexible library, it may not be the most efficient choice for very large graphs. The performance of the code could be improved by using a more specialized graph representation.
8. **Limited Iterations**: The build_circuit method constructs the quantum circuit with a fixed number of iterations (default is 2). The number of iterations can significantly impact the performance of the algorithm, and finding the optimal number of iterations may require additional experimentation.

Overall, while the code provides a good starting point for solving the shortest path problem using quantum computing, it has several limitations that need to be addressed to improve its scalability, accuracy, and performance.

The code that was pretty much a standard implementation of Grover's search algorithm

1. Oracle Construction:
    - The oracle identifies all valid paths between the start and end nodes.
    - marks paths with the minimal total weight (shortest paths)
    - Uses multi-controlled X gates to flip the phase of solution states
2. Grover's algorithm:

- creates uniform superposition of all possible paths
- applies the oracle and diffusion operator iteratively
- the number of iterations is calculated based on the estimated number of solutions

3. Result Interpretation:
    - converts measureent results back to path representations
    - filters valid paths and removes duplicates
    - returns the shortest paths found

Limitations and Considerations:

1. Graphs Size: works best for small graphs (3-5) nodes due to exponential growth of quantum resources needed
2. Path Encoding: uses one qubit per node, which may not be the most efficient for all graphs
3. Classical Preprocessing: the oracle creation uses classical computation to find all paths, which limits the quantum advantage
4. Optimal Iterations: the calculation of optimal Grover iterations assumes knowledge of the number of solutions.

```
from qiskit import QuantumCircuit, Aer, execute
from qiskit.circuit.library import GroverOperator
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_histogram
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
from math import ceil, pi

def create_graph(vertices, weighted_edges):
    """Create a sample graph for testing"""
    G = nx.Graph()
    G.add_nodes_from(vertices)
    G.add_weighted_edges_from(weighted_edges)
    return G
```

```
def path_to_bitstring(path, num_nodes):
    """Convert a path to a bitstring representation"""
    bitstring = ['0'] * num_nodes
    for node in path:
        bitstring[node] = '1'
    return ''.join(bitstring)


def is_valid_path(G, path, start, end):
    """Check if the path is valid (connects start to end through edges)"""
    print ("path:",path, "start:",start,"end:",end)
    if not path or path[0] != start or path[-1] != end:
        return False
    for i in range(len(path)-1):
        if not G.has_edge(path[i], path[i+1]):
            return False
    return True


def create_oracle(G, start, end, num_nodes):
    """Create an oracle that marks valid paths from start to end"""
    # Generate all possible paths (for small graphs)
    all_paths = []
    for path in nx.all_simple_paths(G, start, end):
        all_paths.append(path)

    # Find the minimal path length
    if not all_paths:
        raise ValueError("No paths exist between the specified nodes")

    min_length = min(sum(G[path[i]][path[i+1]]['weight'] for i in range(len(path)-1)) for path in all_paths)

    # Mark states that represent valid paths with minimal length
    marked_states = []
    for path in all_paths:
        path_length = sum(G[path[i]][path[i+1]]['weight'] for i in range(len(path)-1))
        if path_length == min_length:
            bitstring = path_to_bitstring(path, num_nodes)
```

```
            marked_states.append(bitstring)

    # Create oracle circuit
    oracle = QuantumCircuit(num_nodes)
    for state in marked_states:
        # Flip the phase of marked states
        oracle.x([i for i, bit in enumerate(reversed(state)) if bit == '0'])
        oracle.h(num_nodes-1)
        oracle.mcx(list(range(num_nodes-1)), num_nodes-1)
        oracle.h(num_nodes-1)
        oracle.x([i for i, bit in enumerate(reversed(state)) if bit == '0'])

    return oracle, len(marked_states)

def grover_shortest_path(G, start, end):
    """Find shortest path using Grover's algorithm"""
    num_nodes = len(G.nodes)

    # Create oracle
    oracle, num_solutions = create_oracle(G, start, end, num_nodes)

    # Determine optimal number of Grover iterations
    iterations = ceil(pi/4 * np.sqrt(2**num_nodes / num_solutions))
    print("iterations:",iterations)

    # Create Grover operator
    grover_operator = GroverOperator(oracle)

    # Construct the full circuit
    qc = QuantumCircuit(num_nodes, num_nodes)
    qc.h(range(num_nodes))  # Apply Hadamard to all qubits

    # Apply Grover iterations
    for _ in range(iterations):
        qc.append(grover_operator, range(num_nodes))
```

```
# Measure all qubits
qc.measure(range(num_nodes), range(num_nodes))

# Execute the circuit
backend = Aer.get_backend('qasm_simulator')
shots = 1024
result = execute(qc, backend, shots=shots).result()
counts = result.get_counts()

# Display results
print("Measurement results:")
print(counts)
plot_histogram(counts)
plt.show()

qc.draw(output='mpl', fold=100)

# Find the most probable solutions
sorted_counts = sorted(counts.items(), key=lambda x: x[1], reverse=True)

# Interpret results
solutions = []
#for bitstring, count in sorted_counts[:5]:  # Top 5 results
for bitstring, count in sorted_counts:  # All results
    path = [i for i, bit in enumerate(bitstring) if bit == '1']
    if is_valid_path(G, path, start, end):
        path_length = sum(G[path[i]][path[i+1]]['weight'] for i in range(len(path)-1))
        solutions.append((path, path_length))

# Remove duplicates
unique_solutions = []
seen = set()
for solution in solutions:
    path_str = str(solution[0])
    if path_str not in seen:
        seen.add(path_str)
```

```
            unique_solutions.append(solution)

    return unique_solutions

def run_GroverGraphShortestPath(G, start, end):
    # Choose start and end nodes
    start = start
    end = end

    # Draw the graph
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='lightblue')
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
    plt.title("Input Graph")
    plt.show()

    # Find shortest path using Grover's algorithm
    solutions = grover_shortest_path(G, start, end)
    if solutions:
        print("\nFound shortest paths:")
        for path, length in solutions:
            print(f"Path: {path}, Length: {length}")
    else:
        print("No valid path found.")
```

Let’s dive into the grover_shortest_path function.

1. **Purpose**: This function aims to find the shortest path between two points in a graph using Grover's algorithm, which is a quantum algorithm.
2. **Graph and Nodes**: The function takes a graph G and two points, start and end. It first counts the number of nodes (points) in the graph.
3. **Oracle Creation**: An oracle is created, which helps identify the correct paths between the start and end points. The oracle also tells how many valid solutions (paths) exist.
4. **Grover Iterations**: The function calculates the optimal number of times to apply Grover's algorithm based on the number of nodes and solutions.

5. **Quantum Circuit**: A quantum circuit is constructed with the same number of qubits as nodes. Initially, all qubits are set to a superposition state using Hadamard gates.
6. **Grover Operator**: The Grover operator is applied to the quantum circuit for the calculated number of iterations. This operator helps amplify the probability of finding the correct path.
7. **Measurement**: The quantum circuit is measured, and the results are collected. These results show the probability distribution of different paths.
8. **Result Analysis**: The function sorts the measurement results to find the most probable paths. It then checks if these paths are valid and calculates their lengths.
9. **Unique Solutions**: Duplicate paths are removed, and the function returns a list of unique valid paths along with their lengths.

In essence, this function uses quantum computing to efficiently find the shortest path in a graph, leveraging Grover's algorithm to speed up the search process.

This function, while powerful, has several limitations:

1. **Quantum Hardware**: The function relies on quantum computing, which is still in its early stages. Access to quantum hardware is limited, and current quantum computers have noise and error rates that can affect the accuracy of results.
2. **Scalability**: The function's performance is tied to the number of nodes in the graph. As the number of nodes increases, the quantum circuit becomes more complex, requiring more qubits and computational resources.
3. **Oracle Creation**: The function assumes that an oracle can be created to identify valid paths. Designing and implementing such an oracle can be challenging, especially for complex graphs.
4. **Grover Iterations**: The number of Grover iterations is based on an estimate of the number of solutions. If this estimate is inaccurate, the function may not find the optimal path.
5. **Classical Post-Processing**: After the quantum computation, the function performs classical post-processing to interpret the results and find valid paths. This step can be computationally intensive, especially for large graphs.
6. **Graph Representation**: The function assumes that the graph is represented in a specific way, with nodes and edges having weights. If the graph representation differs, the function may need modifications.
7. **Simulation Limitations**: The function uses a quantum simulator for execution. Simulators have limitations in terms of the number of qubits they can handle and the accuracy of the simulation.
8. **Path Validity**: The function checks the validity of paths after measurement. If the graph has many invalid paths, the function may need to perform additional checks, increasing the computational load.

These limitations highlight the challenges of using quantum algorithms for practical problems, but they also underscore the potential for significant advancements as quantum technology evolves.

The formula used to determine the number of Grover iterations is:

$$R = \frac{\pi}{4}\sqrt{\frac{2^n}{M}}$$

where:

- n is the number of qubits (or nodes in the graph).
- M is the number of solutions (valid paths).
- R is rounded to the nearest integer.

Why This Formula?

1. **Grover's Algorithm Basics**: Grover's algorithm is designed to search an unsorted database or solve an unstructured search problem. It provides a quadratic speedup over classical algorithms. The algorithm works by amplifying the probability of the correct solutions through a series of iterations.
2. **Optimal Number of Iterations**: The formula calculates the optimal number of iterations needed to maximize the probability of finding a correct solution. If you perform too few iterations, the probability of finding the correct solution is low. If you perform too many iterations, the probability starts to decrease again. The formula ensures that you stop at the point where the probability is highest.
3. **Quadratic Speedup**: The term $\sqrt{\frac{2^n}{M}}$ represents the quadratic speedup. In classical search algorithms, you would need to check 2^n possibilities. Grover's algorithm reduces this to approximately $\sqrt{2^n}$ possibilities. The factor$\frac{1}{\sqrt{M}}$ adjusts for the number of solutions, ensuring that the algorithm is efficient even if there are multiple valid solutions.
4. **Ceiling Function**: The ceiling function ensures that the number of iterations is an integer. Since you can't perform a fractional number of iterations, rounding up ensures that you perform enough iterations to maximize the probability of success.

In summary, this formula is used to balance the trade-off between too few and too many iterations, ensuring that Grover's algorithm is both efficient and effective in finding the shortest path in the graph.

Let's break down the function of the multi-controlled gates in the oracle circuit.

```
# Create oracle circuit
    oracle = QuantumCircuit(num_nodes)
```

```
for state in marked_states:
    # Flip the phase of marked states
    oracle.x([i for i, bit in enumerate(reversed(state)) if bit == '0'])
    oracle.h(num_nodes-1)
    oracle.mcx(list(range(num_nodes-1)), num_nodes-1)
    oracle.h(num_nodes-1)
    oracle.x([i for i, bit in enumerate(reversed(state)) if bit == '0'])
```

Oracle Circuit

The oracle circuit is a crucial part of Grover's algorithm. It is responsible for marking the "correct" or "desired" states by flipping their phase. This marking helps the algorithm amplify the probability of these states during the Grover iterations.

Multi-Controlled Gates

In the above code snippet, the multi-controlled gate (mcx) is used to flip the phase of the marked states. Here's how it works:

1. **Initialization**: The oracle circuit is initialized with the same number of qubits as nodes in the graph.
2. **Marking States**: For each marked state (a state that represents a valid path in the graph):
    - **X Gates**: The x gates are applied to the qubits that are in the 0 state in the marked state. This effectively inverts the qubits, preparing them for the multi-controlled gate.
    - **Hadamard Gate**: A Hadamard gate (h) is applied to the last qubit (target qubit). This puts the target qubit into a superposition state, making it ready for the multi-controlled gate.
    - **Multi-Controlled X Gate (mcx)**: The mcx gate is a multi-controlled NOT gate (also known as a Toffoli gate). It flips the target qubit if all the control qubits are in the 1 state. In this context, it flips the phase of the marked state.
    - **Hadamard Gate**: Another Hadamard gate is applied to the target qubit to revert it back from the superposition state.
    - **X Gates**: The x gates are applied again to revert the qubits to their original state.
    -

Purpose of Multi-Controlled Gates

The multi-controlled gates are used to conditionally flip the phase of the target qubit based on the state of the control qubits. This conditional flipping is what marks the desired states in Grover's

algorithm. By marking these states, the algorithm can amplify their probability during the Grover iterations, making it more likely to find the correct solution.

In summary, the multi-controlled gates in the oracle circuit are essential for marking the desired states by flipping their phase, which is a key step in Grover's algorithm for finding the shortest path in the graph.

The provided Qiskit code for solving the shortest path problem using Grover's algorithm has several limitations:

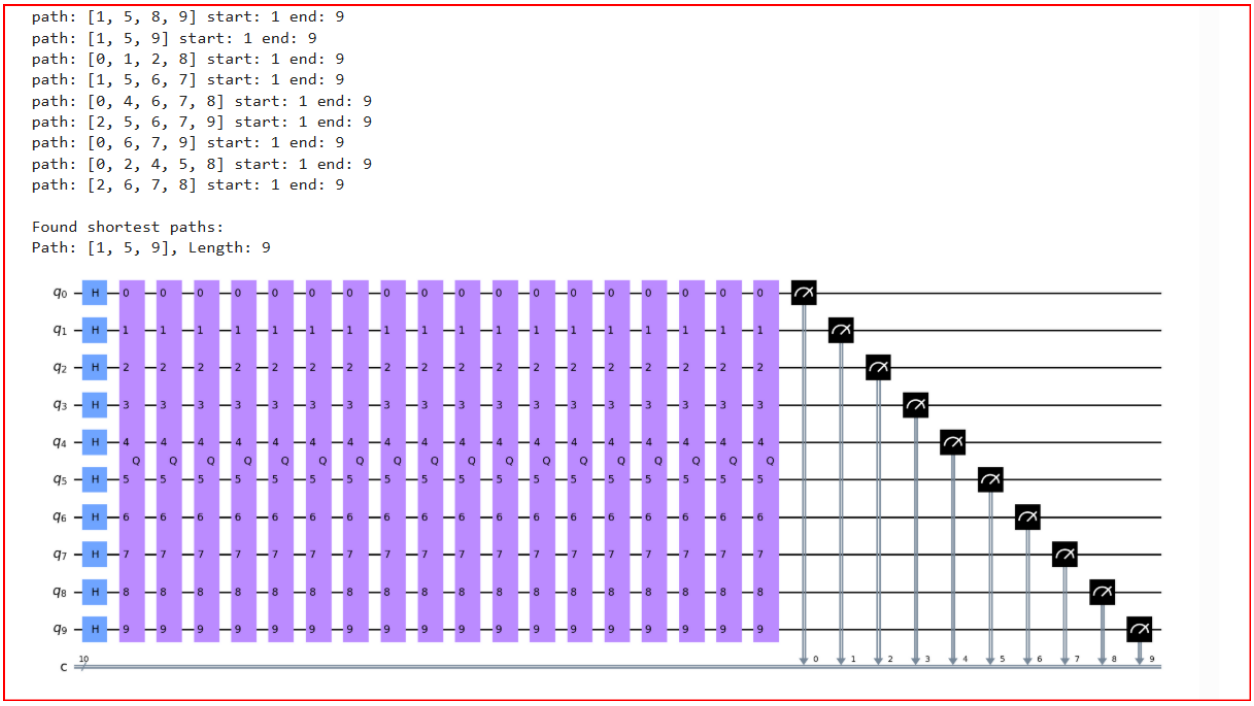
1. **Scalability**: The code is designed to work with a quantum circuit whose size is proportional to the number of nodes in the graph. As the number of nodes increases, the quantum circuit becomes larger and more complex, making it difficult to scale to larger graphs.
2. **Noise and Errors**: Quantum computers are prone to noise and errors, which can affect the accuracy of the results. The code uses a quantum simulator (qasm_simulator), which does not account for the noise present in real quantum hardware. Therefore, the results obtained from the simulator may not be representative of those obtained from actual quantum hardware.
3. **Oracle Construction**: The code constructs an oracle that marks valid paths with minimal length. This process involves generating all possible paths between the start and end nodes, which can be computationally expensive for larger graphs. Additionally, the oracle circuit itself can become quite complex, especially for graphs with many nodes and edges.
4. **Grover Iterations**: The code determines the optimal number of Grover iterations based on the number of solutions. While this approach is theoretically sound, it may not always yield the best results in practice, especially in the presence of noise and errors. The number of iterations can significantly impact the performance of the algorithm, and finding the optimal number may require additional experimentation.
5. **Initialization**: The code initializes the quantum state by applying a Hadamard gate to all qubits. This initialization may not be optimal for all graph structures and could affect the performance of the algorithm.
6. **Classical Post-Processing**: After executing the quantum circuit, the code performs classical post-processing to interpret the measurement results and identify valid paths. This step can be computationally expensive, especially for large graphs with many possible paths.
7. **Graph Representation**: The code uses the NetworkX library to represent the graph. While NetworkX is a powerful and flexible library, it may not be the most efficient choice for very large graphs. The performance of the code could be improved by using a more specialized graph representation.
8. **Limited Iterations**: The grover_shortest_path method constructs the quantum circuit with a fixed number of iterations. The number of iterations can significantly impact the

performance of the algorithm, and finding the optimal number of iterations may require additional experimentation.

Overall, while the code provides a good starting point for solving the shortest path problem using Grover's algorithm, it has several limitations that need to be addressed to improve its scalability, accuracy, and performance.

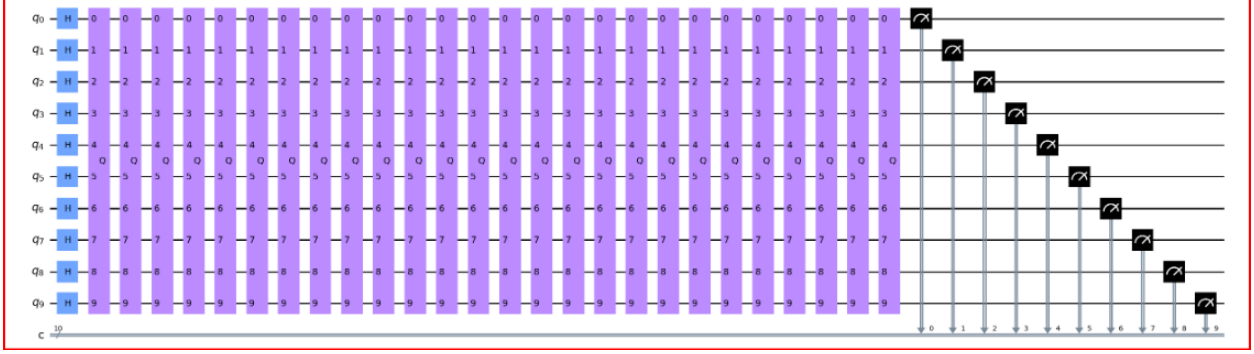
Testing

For the same above 10-node graph the Grover search code identified correctly the shortest path between 1 and 9 nodes.



Another correctly identified shortest path example:

```
path: [0, 1, 2] start: 0 end: 2
path: [0, 2, 3, 5, 7, 9] start: 0 end: 2
path: [4, 8, 9] start: 0 end: 2
path: [3, 5, 6, 8, 9] start: 0 end: 2
path: [0, 4, 5, 6, 8] start: 0 end: 2
path: [2, 3, 4, 5, 8] start: 0 end: 2
path: [1, 2, 3, 4, 5, 6, 7, 8, 9] start: 0 end: 2
path: [0, 3, 5, 7, 9] start: 0 end: 2
path: [6, 9] start: 0 end: 2

Found shortest paths:
Path: [0, 1, 2], Length: 17
```



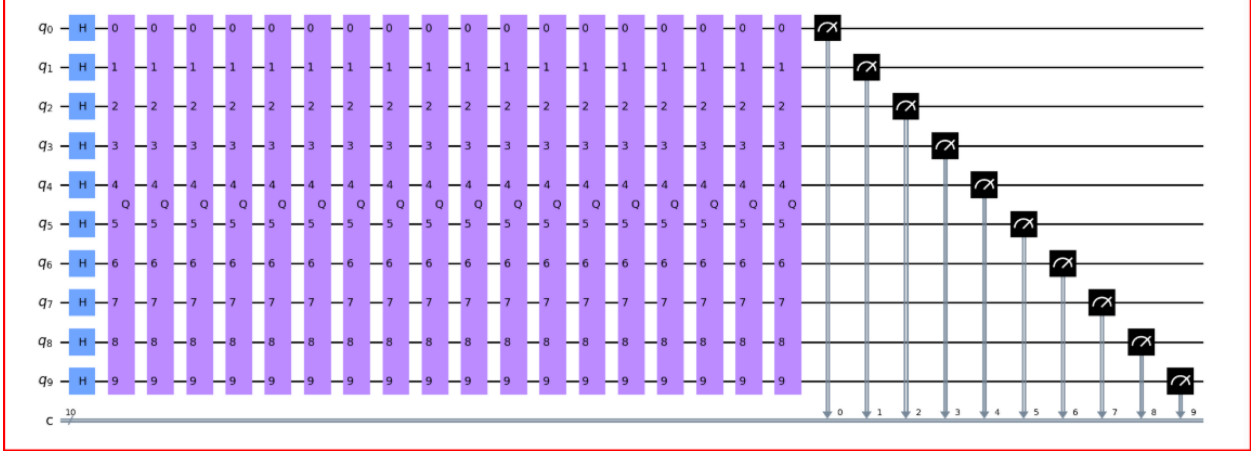
Summary:

Both approaches for solving the shortest path problem provide a good starting point. However, both approaches have several limitations that need to be addressed to improve its scalability, accuracy, and performance.

For example, both approaches failed to find the shortest path between 9 and 6 nodes for the above 10-node graph.

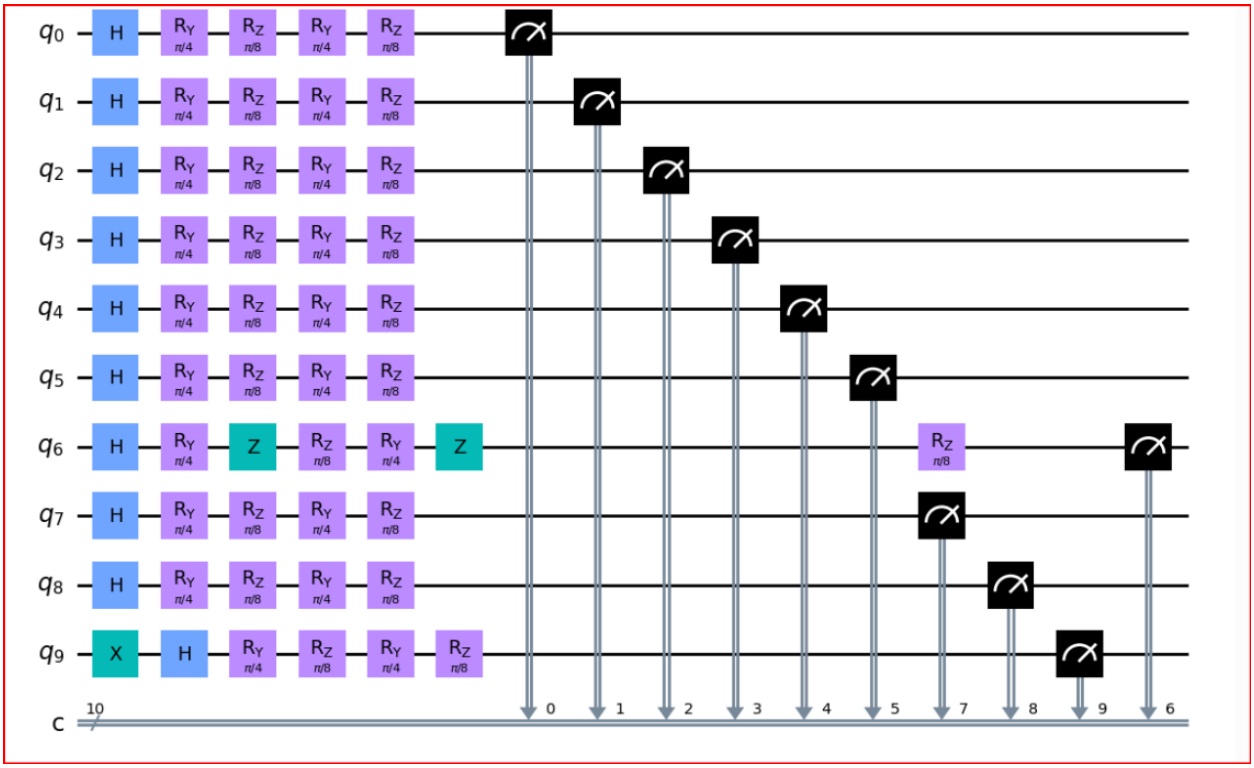
The shortest path from 9 to 6 is: 9 -> 5 -> 1 -> 6 with a cost of 10.

```
path: [1, 5, 6, 8, 9] start: 9 end: 6
path: [1, 5, 6, 9] start: 9 end: 6
path: [1, 2, 3, 7, 8, 9] start: 9 end: 6
path: [0, 1, 5, 8, 9] start: 9 end: 6
No valid path found.
```



```
run_example(G, 9, 6)
```

```
Starting parameter tuning...
Optimization results:
Best θ: 0.7854
Best γ: 0.3927
Best score: -0.0000

Running with tuned parameters...
No valid paths found.
```



All the code and analysis is in this notebook:

https://github.com/samlip-blip/deepseekandgraphs/blob/main/DeepSeekAndShortestPath.ipynb

Please feel free to download the notebook and explore.

In the next chapter we explore the relationship between palindrome binary numbers and symmetric graphs. The chapter discusses how symmetric graphs can be represented using palindrome binary numbers and the implications of this representation in various applications.

Palindrome Binary Numbers and Symmetric Graphs

Palindrome binary numbers can indeed be connected to graphs in various ways, particularly in the context of graph theory and computer science.

Graph Representation:

In graph theory, a graph is a collection of nodes (vertices) connected by edges. Binary numbers can be used to represent the structure of a graph. For example, each bit in a binary number can represent the presence or absence of an edge between nodes.

Palindrome Binary Numbers:

A palindrome binary number is a binary number that reads the same forwards and backwards. For example, 101 and 1001 are palindrome binary numbers.

Connection to Graphs:

Symmetric Graphs: Palindrome binary numbers can be used to represent symmetric graphs. In a symmetric graph, the adjacency matrix (a matrix representing connections between nodes) is symmetric. This means that the graph looks the same when viewed from different directions, like how a palindrome reads the same forwards and backwards.

Symmetric graphs are used in various practical applications. One notable example is in network design. In telecommunications and computer networks, symmetric graphs can represent network topologies where each node has the same number of connections, ensuring uniformity and balance. This can help in optimizing network performance and reliability.

Another example is in chemistry, where symmetric graphs can represent molecular structures. For instance, the benzene molecule (C_6H_6) can be represented as a symmetric graph, where each carbon atom is connected to two other carbon atoms and one hydrogen atom, forming a hexagonal ring. This symmetry helps in understanding the molecule's stability and reactivity.

Symmetric graphs are also used in social network analysis to model relationships between individuals or groups. In these graphs, nodes represent people, and edges represent interactions or connections. Symmetric graphs can help identify patterns and clusters within the network, aiding in the study of social dynamics and influence.

Here are some key aspects:

1. **Representation of Relationships**: In social networks, nodes represent individuals or entities, and edges represent interactions or connections between them. Symmetric graphs ensure that the connections are bidirectional, meaning if person A is connected to person

B, then person B is also connected to person A. This symmetry helps in accurately modeling mutual relationships.

2. **Identification of Patterns**: Symmetric graphs can reveal patterns and clusters within the network. For example, they can help identify tightly-knit groups or communities where individuals have strong mutual connections. These clusters can be important for understanding social dynamics, influence, and the spread of information.
3. **Analysis of Centrality**: Symmetric graphs can be used to analyze centrality measures, which indicate the importance or influence of individuals within the network. Common centrality measures include degree centrality (number of connections), betweenness centrality (control over information flow), and closeness centrality (proximity to other nodes). Symmetric graphs ensure that these measures are balanced and accurately reflect mutual relationships.
4. **Visualization**: Symmetric graphs can be visually represented to provide insights into the structure of the social network. Visualization tools can highlight key individuals, clusters, and the overall connectivity of the network. This can be useful for researchers, marketers, and policymakers to understand and leverage social interactions.
5. **Applications**: Symmetric graphs are used in various applications such as marketing (to identify influential customers), epidemiology (to track the spread of diseases), and organizational behavior (to study communication patterns within companies). By analyzing symmetric graphs, stakeholders can make informed decisions based on the structure and dynamics of social networks.

Graph Encoding: Binary numbers, including palindromes, can be used to encode the structure of a graph. For example, a binary number can represent the adjacency matrix of a graph, where each bit indicates the presence or absence of an edge. Palindrome binary numbers can represent graphs with specific symmetrical properties.

Graph Algorithms: Certain graph algorithms may leverage the properties of palindrome binary numbers. For example, algorithms that search for symmetrical patterns in graphs or optimize paths based on symmetrical properties might use palindrome binary numbers as part of their computations.

Example:

Consider a graph with four nodes (A, B, C, D) and the following edges:

- A to B
- B to C
- C to D
- D to A

The adjacency matrix for this graph can be represented as a binary number. If the graph is symmetric, the binary number might be a palindrome, indicating that the graph has symmetrical properties.

In summary, palindrome binary numbers can be connected to graphs through their representation, encoding, and use in algorithms, particularly in the context of symmetric graphs and graph theory.

Quantum code to verify if a binary number is a palindrome

```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, execute, Aer
import matplotlib.pyplot as plt


def create_palindrome_circuit(binary_str):
    n = len(binary_str)
    num_pairs = n // 2  # Number of bit pairs to compare

    # Quantum registers
    input_reg = QuantumRegister(n, 'input')  # Input bits
    ancilla = QuantumRegister(num_pairs, 'ancilla')  # Comparison results
    result = QuantumRegister(1, 'result')  # Final answer (1 = palindrome)
    cr = ClassicalRegister(1, 'cr')  # Classical register for measurement
    qc = QuantumCircuit(input_reg, ancilla, result, cr)



    # Step 1: Initialize input qubits to the given binary number
    for i, bit in enumerate(binary_str):
        if bit == '1':
```

```
            qc.x(input_reg[i])

    # Step 2: Compare symmetric bit pairs (e.g., first and last bit)
    for i in range(num_pairs):
        left = i
        right = n - 1 - i
        # XOR each pair into ancilla qubits (ancilla=0 if bits match)
        qc.cx(input_reg[left], ancilla[i])
        qc.cx(input_reg[right], ancilla[i])

    # Step 3: Check if all ancillas are 0 (i.e., all pairs matched)
    qc.x(ancilla)  # Invert ancillas (now 1 means "valid pair")
    qc.mcx(ancilla, result)  # Flip result qubit if ALL ancillas are 1
    qc.x(ancilla)  # Revert ancillas to original state

    # Step 4: Measure the result
    qc.measure(result, cr)
    return qc

def run_palindrome_circuit(binary_number):
    qc = create_palindrome_circuit(binary_number)
    qc.draw(output='mpl', fold=100)

    # Simulate the circuit
    simulator = Aer.get_backend('qasm_simulator')
    result = execute(qc, simulator, shots=1024).result()
    counts = result.get_counts()

    print(f"Results for binary number '{binary_number}':", counts)
    if '1' in counts:
        print("✅ The number is a palindrome!")
    else:
        print("❌ The number is NOT a palindrome.")
```

The above Qiskit code is designed to create and run a quantum circuit that checks if a given binary number is a palindrome.

The create_palindrome_circuit function creates a quantum circuit to check if the input binary string is a palindrome. It does the following:

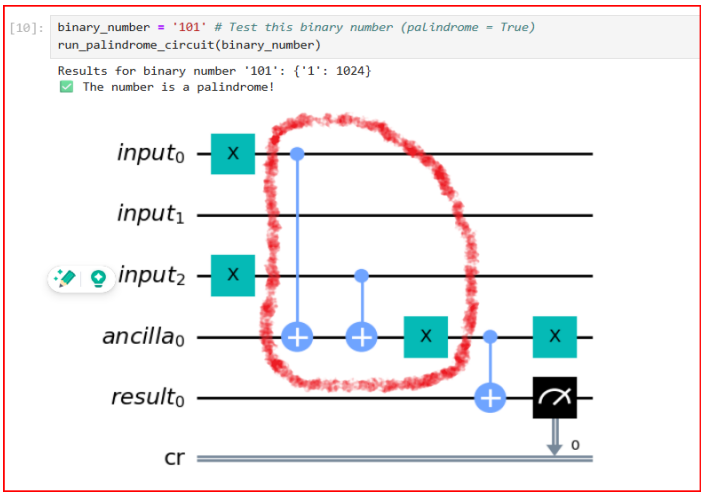
- Initializes quantum registers for input bits, ancilla bits (for comparison results), and a result bit.
- Sets the input qubits to the given binary number.
- Compares symmetric bit pairs using XOR operations.
- Checks if all ancilla bits are 0 (indicating all pairs matched).
- Measures the result.

The run_palindrome_circuit function runs the palindrome circuit by:

  - Creating the circuit using the create_palindrome_circuit function.
  - Drawing the circuit.
  - Simulating the circuit using Qiskit's qasm_simulator.
  - Printing the results and determining if the binary number is a palindrome based on the measurement results.

In summary, the Qiskit code creates a quantum circuit to check if a binary number is a palindrome and simulates the circuit to determine the result. If the result qubit is measured as 1, the number is a palindrome; otherwise, it is not.

For each symmetric bit pair, an ancilla qubit is used to store the result of the comparison by employing two CNOT (XOR) gates followed by an X gate. Let's consider a simple example: the X gates for input0 and input2 both encode 1 for the binary value 101.

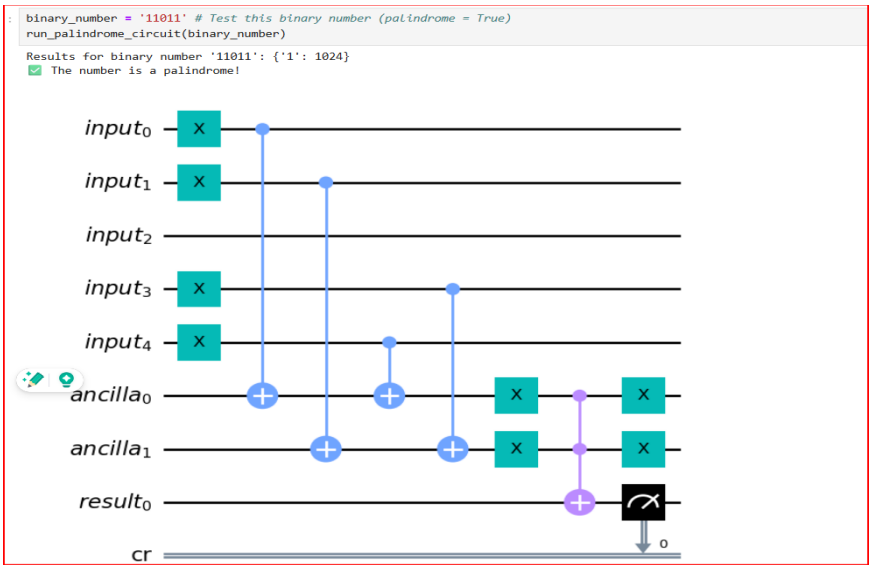


Consequently, the combination of CNOT1-CNOT2-X gates sets ancilla0 to 1 when both inputs have the same bit value, as illustrated in the table below for all possible inputs.

Input0	Input2	Input0-X	Input2-X	Ancilla0-CNOT1	Ancilla0-CNOT2	Ancilla0-X
0	**0**	**1**	**1**	**1**	**0**	**1**
0	1	1	0	1	1	0
1	0	0	1	0	1	0
1	1	**0**	**0**	**0**	**0**	**1**

Interestingly, we observed the same CNOT-CNOT-X gates pattern in the Bipartite Graphs chapter when we encoded a check for same-color violations.

Finally, the result0 qubit is set to 1 by the CNOT gate because ancilla0 is set to 1, and then ancilla0 is reverted to its original state of 0. For larger binary numbers, additional ancilla qubits are used to verify if all ancilla bits are 1, which is accomplished using a Toffoli gate, followed by reverting all ancillas to their original 0 state, as demonstrated in the example below.



The above quantum code is an exploratory code and is scalable to a certain extent, but there are some considerations to keep in mind:

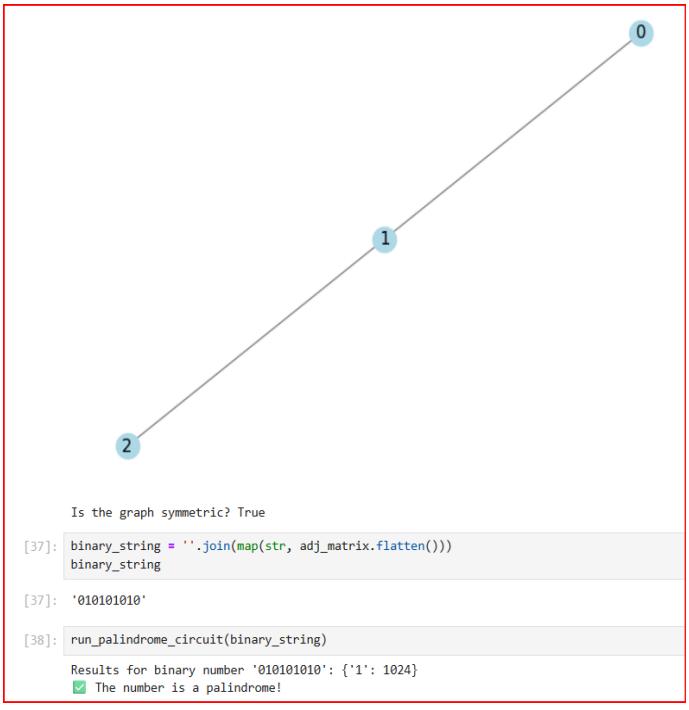
1. **Quantum Register Size**: The size of the quantum registers (input_reg, ancilla, and result) will increase with the length of the binary string. For very large binary numbers, this could become a limitation due to the finite number of qubits available on current quantum hardware.
2. **Gate Operations**: The number of gate operations (CNOT, X, and Toffoli gates) will also increase with the length of the binary string. This can lead to longer execution times and higher error rates, especially on noisy quantum devices.

3. **Simulation**: When simulating the circuit using a classical simulator like qasm_simulator, the computational resources required will grow with the size of the quantum circuit. This can lead to longer simulation times and higher memory usage.
4. **Measurement**: The measurement step is straightforward and scales linearly with the number of qubits, but the complexity of the circuit before measurement can impact the overall scalability.

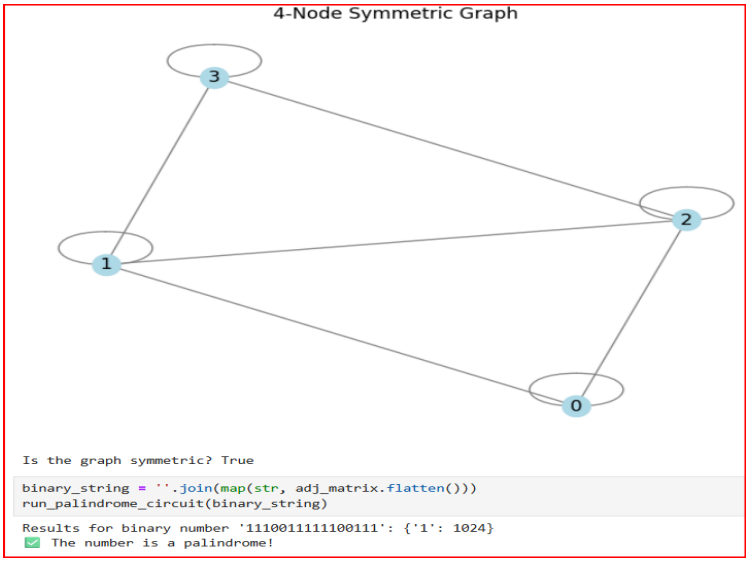
Overall, while the code is scalable for moderate-sized binary strings, it may face challenges with very large binary numbers due to hardware limitations and increased complexity. Optimizing the circuit and using error mitigation techniques can help improve scalability.

Testing

If you have the adjacency matrix of a graph, you can use it to obtain the corresponding binary number and check if it is a palindrome. For instance, with this 3-node symmetric graph, we have:



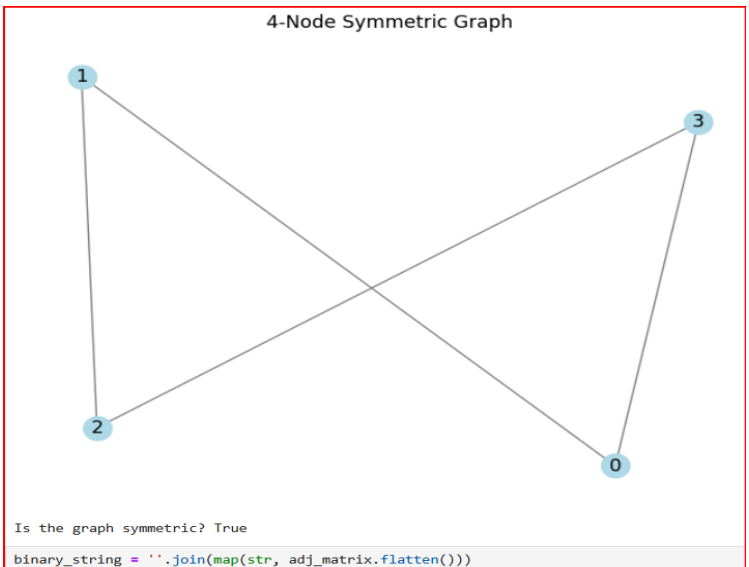
Another 4-node symmetric graphs:




```
Is the graph symmetric? True
```

```
binary_string = ''.join(map(str, adj_matrix.flatten()))
run_palindrome_circuit(binary_string)
```

```
Results for binary number '1110011111100111': {'1': 1024}
✅ The number is a palindrome!
```




```
Is the graph symmetric? True
```

```
binary_string = ''.join(map(str, adj_matrix.flatten()))
run_palindrome_circuit(binary_string)
```

```
Results for binary number '0101101001011010': {'1': 1024}
✅ The number is a palindrome!
```

For a non-symmetric graph, the quantum code returns “NOT a palindrome” as expected.

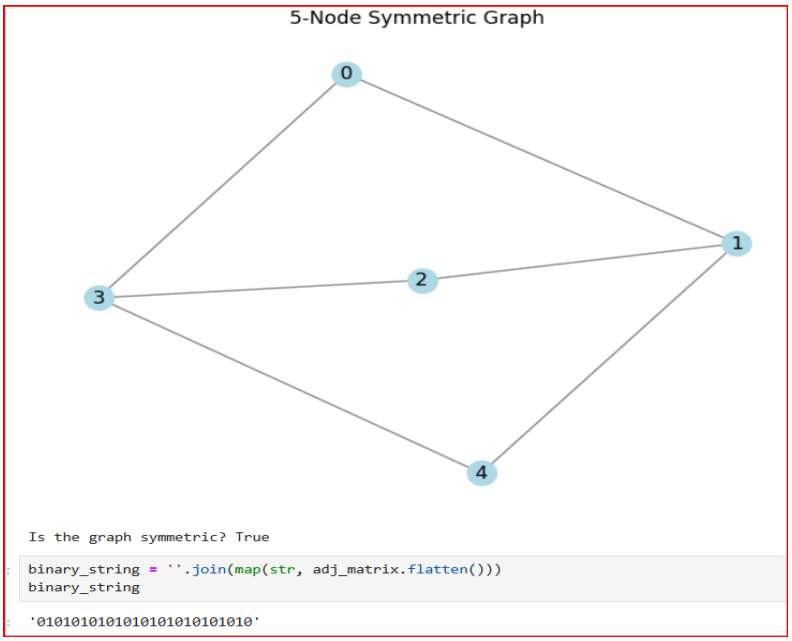
```
[23]: # Create a non-symmetric graph
non_symmetric_graph = nx.DiGraph()
non_symmetric_graph.add_edges_from([(1, 2), (2, 3), (3, 1)])
adj_matrix = nx.adjacency_matrix(non_symmetric_graph).todense()
print(adj_matrix)
```

```
[[0 1 0]
 [0 0 1]
 [1 0 0]]
```

```
[24]: binary_string = ''.join(map(str, adj_matrix.flatten()))
run_palindrome_circuit(binary_string)
```

```
Results for binary number '010001100': {'0': 1024}
❌ The number is NOT a palindrome.
```

Naturally, the quantum code is merely a toy code that quickly encounters memory issues with the QASM simulator as the graph size increases.



QiskitError: 'ERROR: [Experiment 0] QasmSimulator: **Insufficient memory for 37-qubit circuit using "statevector" method.** You could try using the "matrix_product_state" or "extended_stabilizer" method instead. , ERROR: QasmSimulator: Insufficient memory for 37-qubit circuit using "statevector" method. You could try using the "matrix_product_state" or "extended_stabilizer" method instead.'

Summary:

Overall, the code provides a good demonstration of quantum computing concepts but faces challenges related to scalability, error rates, resource constraints, noise, and practical applicability.

Limitations:

1. **Scalability**:
    - The code's complexity increases with the length of the binary string. For very long strings, the number of qubits and gates required can become impractical, leading to increased resource consumption and longer execution times.
2. **Error Rates**:
    - Quantum gates, especially multi-controlled gates, are prone to errors. The accuracy of the results may be affected by the error rates of the quantum hardware used for simulation.
    -

3. **Resource Constraints**:
    - The code requires a significant number of qubits, especially for long binary strings. Current quantum hardware may have limitations on the number of available qubits, which can restrict the length of the binary strings that can be processed.
4. **Noise and Decoherence**:
    - Quantum circuits are susceptible to noise and decoherence, which can affect the reliability of the results. The simulation may not fully capture these effects, leading to discrepancies between simulated and actual hardware results.
5. **Measurement Overhead**:
    - The measurement process introduces overhead and potential errors. The accuracy of the final result depends on the precision of the measurement.
6. **Limited Practical Applications**:
    - While the code demonstrates the principles of quantum computing, its practical applications are limited. Checking for palindromes can be efficiently done using classical algorithms, making the quantum approach more of a theoretical exercise.

By converting the adjacency matrix of a graph into a binary string, we can use this representation as input for the quantum code. Flattening the adjacency matrix and transforming it into a binary string creates a sequence of bits that the quantum circuit can process. This method enables us to utilize the existing quantum code to examine the symmetry of the graph's adjacency matrix and ascertain whether the graph is symmetric.

Please feel free to download the notebook and further explore:

palindromes/Palindrome.ipynb at main · samlip-blip/palindromes

In the next chapter we delve into the SYK model, a quantum many-body system that provides insights into black hole physics, specifically information scrambling. The chapter explains how the SYK model exhibits chaotic behavior, where local information spreads rapidly throughout the system, similar to how black holes mix information.

The SYK Model and Black Hole Information Scrambling

Now, let's embark on an exciting journey into the captivating physics of black holes, which intriguingly intersects with graph theory, especially random graphs. Our focus will be on information scrambling. When an object falls into a black hole, the information about it gets mixed up and spread across the black hole's event horizon (the boundary surrounding the black hole). This phenomenon is known as information scrambling. Imagine a blender mixing fruits, making it nearly impossible to identify the original ingredients. The SYK model replicates this information scrambling in black holes.

First, we'll provide an overview of the SYK model in the context of information scrambling. Although it's a bit technical, don't worry, there's a dictionary of difficult terms to ease your understanding, followed by an intriguing quantum code that simulates black hole information scrambling.

Let's begin our thrilling adventure!

Introduction: The Sachdev–Ye–Kitaev (SYK) model is an exactly solvable quantum many-body model that has gained prominence for its unexpected connections to black hole physics [1]. Initially conceived in a condensed matter context by Subir Sachdev and Jinwu Ye, and refined by Alexei Kitaev, the SYK model consists of N Majorana fermions with all-to-all random interactions [1]. Despite its simplicity, this model provides deep insights into strongly correlated quantum matter and has a close relationship to a discrete version of the AdS/CFT correspondence [1]. In recent years, it has emerged as a theoretical bridge between quantum chaos in many-body systems and the physics of black holes. In particular, SYK offers a rare **"solvable" window** into how black holes scramble information – a process at the heart of the black hole information paradox. As Juan Maldacena noted, *"the SYK model gives us a glimpse into the interior of an extremal black hole … a feature this model has that no other model has"* [2]. Below, we outline the theoretical framework underlying this connection, including the concepts of quantum chaos, out-of-time-order correlators (OTOCs), and the holographic AdS/CFT duality.

Quantum Chaos and Information Scrambling

In classical physics, chaotic systems exhibit extreme sensitivity to initial conditions – the famed "butterfly effect." In **quantum chaos**, we seek analogous behavior in quantum systems: small perturbations rapidly spread through a system's degrees of freedom, effectively hiding or **scrambling** the initial information. Information scrambling refers to the process by which local information (for example, a quantum bit or a localized excitation) gets delocalized into the many-body degrees of freedom of a quantum system [3]. A perfectly scrambled state is one where the original local information can no longer be retrieved by any local measurement, having been smeared out into highly nonlocal correlations (often manifested as entanglement).

Black holes are conjectured to be *optimal* scramblers of information. In a seminal conjecture, Sekino and Susskind proposed that *"black holes are the fastest scramblers in nature,"* meaning they generate entanglement and mix information at the maximal rate allowed by quantum mechanics [4]. In fact, for an N-degree-of-freedom system, they argued a black hole can scramble information in a time on the order of $\sim \ln N$ (logarithmic in the number of degrees of freedom) – much faster than typical quantum systems which might take times polynomial in N. This **fast scrambling** conjecture is rooted in the need for unitary black hole evaporation: if black holes eventually return information via Hawking radiation, they must first *thoroughly mix* (scramble) that information internally [4]. In other words, to avoid being a permanent information sink, a black hole rapidly smears any infalling data across its horizon degrees of freedom, so that it can later leak out in a seemingly random (but ultimately information-carrying) form.

Quantum chaos in many-body systems is often diagnosed by the loss of initial memory and the approach to thermal equilibrium (quantum thermalization) [5]. Recent studies have revealed striking connections between this notion of chaos in microscopic quantum systems and the dynamics of black holes [5]. A central idea is that quantum chaos leads to information delocalization (scrambling) which closely parallels how black holes would thermalize and hide perturbations behind their event horizon. A key quantity to make this notion precise is the **out-of-time-order correlator**, which we discuss next.

Out-of-Time-Order Correlators (OTOCs) as a Measure of Scrambling

To quantify information scrambling and chaos in a quantum system, physicists use **out-of-time-order correlators** (OTOCs). An OTOC is a correlation function that, roughly, measures the growth of the commutator of two initially independent operators W and V as one of them is evolved in time. A common definition is:

$$C(t) = \langle [W(t), V(0)]\dagger\, [W(t), V(0)] \rangle,$$

where $W(t) = e^{iHt} W e^{-iHt}$ is an operator W evolved Heisenberg-style under the system's Hamiltonian H. Here $V(0)$ is some operator at the initial time. If the system's dynamics *do not* scramble information, then a localized operator at time t will still commute with an independent operator V at time 0, and the commutator [W(t), V(0)] will remain small (possibly zero). However, if the perturbation created by W at time t spreads through the system, it will fail to commute with $V(0)$ – the commutator grows, and so does $C(t)$ [3]. In essence, the OTOC captures how a perturbation applied at one time (or location) affects the value of an observable measured at another time (or location), thereby diagnosing the scrambling of information. When $C(t)$ grows toward a large value, it indicates that the information initially encoded in V (at time 0) has been smeared into the degrees of freedom that $W(t)$ acts on [3]. From an information-theoretic perspective, OTOCs thus **"determine the degree to which local information becomes hidden in nonlocal degrees of freedom, leading to effective memory loss of initial conditions"** [3]. In chaotic quantum systems, OTOCs typically exhibit an early-time exponential growth, analogous to the exponential divergence of trajectories in classical chaos. One often finds

$$C(t) \sim e^{\lambda t},$$

for some rate λ at intermediate times before saturation. This λ is identified as a **quantum Lyapunov exponent** [3], representing the rate of scrambling. Importantly, there is a fundamental limit on how large λ can be. Maldacena, Shenker, and Stanford (MSS) showed that for thermal quantum systems, λ **has an upper bound** $\boldsymbol{2\pi k_B \frac{T}{\hbar}}$, where T is the temperature [5]. In units where $k_B = \hbar = 1$, this gives $\lambda \leq 2\pi T$. This is often called the **MSS bound** or the *universal bound on chaos*. Physically, it means there is a maximum speed at which information can be scrambled in a system at temperature T. Notably, **black holes** in Einstein gravity are conjectured to *saturate this bound*, with a Lyapunov exponent $\lambda = 2\pi T_{BH}$ [5]. This maximal chaos is one hallmark of black hole dynamics: if one perturbs a black hole (for instance, by throwing in a particle), the perturbation's effects on later observables (such as outgoing Hawking radiation) grow as fast as quantum mechanics permits.

The SYK model garnered attention when it was found to exhibit *precisely* this kind of behavior. Kitaev (building on earlier work by Sachdev and Ye) predicted that the SYK model at low temperature would also saturate the chaos bound, showing Lyapunov growth with $\lambda = 2\pi T$ [5]. This was a surprise because aside from black holes, no other known quantum system reached this extreme limit of chaos [5]. The saturation of the bound in SYK was a strong hint that the model might be capturing the same scrambling dynamics as an actual black hole.

The Sachdev–Ye–Kitaev Model: Definition and Chaotic Properties

Model Definition: The SYK model is defined by a simple but rich Hamiltonian. In its best-known form, it consists of N Majorana fermion operators χ_i (with $i = 1, \ldots, N$) that satisfy $\{\chi_i, \chi_j\} = 2\delta_{ij}$. The Hamiltonian involves random all-to-all interactions among these fermions. For example, one common version is:

$$HSYK = \frac{1}{4!} \sum_{i,j,k,l=1}^{N} J_{ijkl}\, \chi_i \chi_j \chi_k \chi_l,$$

where J_{ijkl} are random coupling constants for each distinct quadruplet (i, j, k, l) [1]. These couplings are typically drawn from a Gaussian distribution with mean zero and a variance scaling as $\frac{J^2}{N^3}$ (so that the model has a sensible large-N limit) [2]. The case shown is a quartic ($q = 4$) interaction in Majorana fermions; more generally one can consider q-body interactions. The simplicity of the SYK Hamiltonian is deceiving – it has no spatial structure (every fermion interacts equally with every other in a random manner), which places it in an "infinite-range" or all-to-all interaction class. Such models are akin to mean-field spin glasses, and indeed the SYK model has deep roots in earlier work on disordered quantum systems and spin liquids [2].

Solvability: The great virtue of SYK is that in the limit of large N (and appropriate scaling of J_{ijkl}, it becomes analytically tractable. One can average over the disorder and perform a path integral analysis or Schwinger-Dyson equation analysis to solve for correlation functions at leading order in $1/N$. The randomness effectively washes out individual microscopic details, leaving behind *universal* behavior. Sachdev and Ye discovered that a similar model exhibited a

nontrivial conformal symmetry at low energies [2]. **Kitaev further showed that the SYK model possesses an emergent reparameterization symmetry in the low-energy limit, making it solvable and providing a candidate for a 0+1-dimensional quantum system with properties resembling a black hole horizon.** In the deep infrared (low energy, $E \to 0$), the SYK model enjoys approximate conformal invariance in time (i.e., it is almost a CFT_1), with that symmetry being both spontaneously and explicitly broken – leaving behind a single collective mode associated with time reparameterizations. The effective action for this mode is the so-called Schwarzian action. This solvable structure is what allows SYK's dynamics to be computed and compared to gravitational results.

Chaotic Features: The SYK model is chaotic in multiple senses:

- **No Quasiparticles:** SYK has no long-lived quasiparticle excitations; it is an example of a non-Fermi liquid state. Instead of well-defined particle-like modes, the system's excitations are highly collective and decaying. This is evidenced by a power-law decay of two-point functions at low energies rather than oscillations or simple exponential decays. The lack of quasiparticles is often a feature of strongly chaotic quantum systems (as opposed to integrable ones). As a result, SYK behaves like a strongly coupled quantum soup, **similar to the dynamic chaos expected in a black hole's interior [6].**

- **Random Matrix Spectra:** A hallmark of quantum chaos is that the energy level statistics follow random matrix theory (RMT) rather than Poisson statistics. The SYK model, for large N, shows spectral correlations that conform to the Wigner-Dyson statistics of random matrices [7]. This indicates that SYK, like a quantum chaotic system, essentially "forgets" initial structure in its spectrum – reminiscent of the idea that a black hole's energy levels (if one could quantize them) should look random (the black hole has "no hair", meaning no additional quantum numbers to characterize its microstates beyond global charges). In this sense, SYK's Hamiltonian is a kind of random matrix, and indeed can undergo transitions between chaotic and integrable phases by tuning parameters [7]. The **universality** of its spectral fluctuations is another parallel with black holes, which are believed to have spectra that at long times produce universal random-matrix-like correlators (as seen in the late-time behavior of the boundary theory correlators).

- **Maximal Lyapunov Exponent:** As mentioned, the SYK model has an out-of-time-order four-point function that can be computed analytically. Kitaev's analysis, later refined by Maldacena and Stanford, showed that the four-point OTOC exhibits exponential growth with a Lyapunov exponent $\lambda = 2\pi T$ (in units with $k_B = \hbar = 1$) at low temperatures [3]. In other words, SYK saturates the MSS chaos bound, qualifying as a **maximally chaotic** system. This behavior is quantitatively identical to that of a black hole in Anti-de Sitter space, which also gives $\lambda = 2\pi T_{BH}$ for its scrambling rate. Numerical simulations of finite-size SYK systems have given strong evidence of this exponential OTOC growth. In one study, researchers observed that as the temperature is lowered, the SYK model's dynamics cross over from those of a typical quantum system to those mirroring a **"quantum black hole,"** with the measured chaos rate approaching the theoretical upper limit [5]. At high temperatures SYK behaves more like a regular interacting quantum system, but at low

temperatures it enters a regime of **extreme quantum chaos** identical to black hole behavior [5].

- **Fast Scrambling:** The large- N limit of SYK is effectively a mean-field description, which suggests that local perturbations become distributed over all N modes in a short time. Although SYK is a 0-dimensional system (no spatial structure), one can define a scrambling time t_* *as the time when OTOCs become* $O(1)$ *(signifying near-complete scrambling). For maximally chaotic systems,* $t_* \sim \frac{1}{\lambda} lnN$. Since SYK has $\lambda \approx 2\pi T$, we get $t_* \sim \frac{1}{2\pi T} lnN$. **This scaling (logarithmic in N) is parametrically the same fast-scrambling scaling conjectured for black holes.** Thus, SYK not only has a large Lyapunov exponent but also spreads information in the minimal time possible up to constants. In the language of Sekino and Susskind, it behaves as a fast scrambler.

- **Thermodynamics and Entropy:** Another striking parallel between SYK and black holes lies in thermodynamics. The SYK model at low temperatures has an entropy that approaches a constant value as $T \to 0$ (when $q > 2$) – a finite zero-temperature entropy (sometimes called "residual entropy"). This is unusual for ordinary quantum systems (which typically have entropy vanishing as $T \to 0$ if the ground state is non-degenerate), but is a known property of extremal black holes which possess a finite horizon area even at zero temperature. In fact, SYK's entropy at low T matches the form expected from the Bekenstein–Hawking entropy of a 1+1D black hole in AdS_2 [8].Subir Sachdev has noted that the SYK model has led to an understanding of how charged AdS black holes realize their large ground-state entropy in a manner consistent with quantum mechanics [8]. **In short, SYK provides a concrete example of a unitary quantum system with a ground-state degeneracy (a large number of microstates at essentially the same energy) analogous to an extremal black hole's horizon microstates. This helps demystify the notion of black hole entropy by showing how it can emerge from an ensemble of quantum states, without violating quantum principles.**

To summarize these points, it's helpful to list the **key parallels between the SYK model and black hole physics**:

- **Maximal Chaos:** The SYK model exhibits a **Lyapunov exponent** that saturates the Maldacena–Shenker–Stanford bound, $\lambda = 2\pi T$, just like an AdS black hole does [5]. This indicates **quantum chaos at the theoretical limit**, a property previously unique to black holes.

- **Fast Scrambling:** SYK scrambles information in a **minimal time** $t_* \sim \frac{1}{2\pi T} lnN$, scaling logarithmically with system size. Black holes are likewise conjectured to scramble in $t_* \sim \frac{1}{2\pi T_{BH}} lnS$ (with S the entropy $\sim N$). Both systems spread local information nearly instantaneously over all degrees of freedom.

- **Residual Entropy:** The SYK model has an extensive **ground-state entropy**, reflecting a huge number of nearly degenerate microstates [8]. This mirrors the **horizon entropy of an**

extremal black hole, providing a testbed for understanding black hole microstate counting in a quantum mechanical model [8].

- **Random Matrix Spectra:** At long times, the SYK model's spectral correlations follow random matrix universality (Wigner-Dyson statistics) [7]. Similarly, black hole energy levels (in a holographic context) are believed to exhibit random matrix statistics – a manifestation of the so-called *Eigenstate Thermalization* and the idea that black holes have no hair (no special spectral features) [7].
- **Holographic Duality:** SYK is believed to be **dually equivalent to a 1+1 dimensional black hole** system (Jackiw–Teitelboim gravity in AdS_2). Many SYK correlation functions, thermodynamic properties, and even quantum phase transitions correspond to processes or structures in a dual AdS_2black hole spacetime [9] [6]. This duality situates SYK firmly in the context of the AdS/CFT correspondence and quantum gravity.

Each of these parallels reinforces the idea that the SYK model is not just an abstract statistical model, but effectively a **"toy model" of a black hole**. We next discuss the AdS/CFT holographic correspondence that underlies this connection.

AdS/CFT Correspondence and the SYK–Black Hole Duality

The **AdS/CFT correspondence** (also known as holographic duality) is a theoretical framework that posits a one-to-one mapping between a gravitational theory in a bulk Anti-de Sitter (AdS) spacetime and a conformal field theory (CFT) living on that spacetime's boundary. In the original example by Maldacena, a 5-dimensional AdS black hole geometry is dual to a 4-dimensional $N = 4$ supersymmetric Yang–Mills theory. In general, black holes in AdS space correspond to finite-temperature states in the dual field theory. Crucially, since the boundary field theory is an ordinary quantum system obeying unitary time evolution, the duality implies that **black hole evolution (in AdS) is also unitary**, providing an elegant resolution to the information paradox in the context of AdS/CFT. In other words, any information that falls into the black hole is not destroyed, but rather encoded in subtle correlations in the Hawking radiation, which correspond to the entangled state of the dual CFT.

For SYK, the relevant version of this correspondence is AdS_2/CFT_1 – a holographic duality between a 1-dimensional quantum system (a quantum mechanics, which plays the role of the "CFT_1") and a 2-dimensional AdS spacetime with gravity. AdS_2 gravity is closely related to the near-horizon geometry of extremal black holes in higher dimensions. For instance, an extremal charged black hole in 4 dimensions often has an $AdS_2 \times S^2$ near-horizon throat. The **Jackiw–Teitelboim (JT) gravity** theory is a simple model of 2D gravity that captures the dynamics of such AdS_2 throats, including the nearly zero-temperature thermodynamics and the pattern of information scrambling.

It has been established that the **low-energy, strong-coupling limit of the SYK model is dual to JT gravity in AdS_2** [9]. In practical terms, this means that for processes occurring on timescales long compared to the inverse of the interaction strength (but short compared to the total quantum lifetime of the system), SYK's correlation functions can be mapped to correlation functions

computed in a 2D black hole spacetime. The SYK model's emergent reparameterization mode (the Schwarzian action) is precisely the boundary action one obtains for nearly-AdS_2 gravity. JT gravity provides a concrete description of the AdS_2 black hole's dynamics, including how it scrambles information and how its entropy behaves. The fact that SYK and JT gravity share the same effective Schwarzian description at low energy is a strong piece of evidence that they are dual to each other.

Holographic dictionary: In this duality, classical concepts of the black hole have counterparts in the SYK model:

- The **thermal AdS_2 black hole** (with near-extremal temperature) corresponds to the SYK model in a thermal state (with the same temperature). Thermal correlation functions of SYK (like two-point Green's functions) have been shown to match those computed from AdS_2 gravity (including one-loop corrections) [3]. Notably, SYK's four-point function (OTOC) at large N can be mapped to a shockwave calculation in the AdS_2 gravity side, yielding the same Lyapunov exponent of $2\pi T$.

- The **Bekenstein–Hawking entropy** of the AdS_2 black hole (including its famous proportionality to horizon area) corresponds to the entropy of the SYK model. SYK reproduces the entropy of an extremal black hole (which is proportional to the extremal horizon area) plus the linear in T correction for near-extremal black holes [8]. In fact, the SYK model helped elucidate the *universal low-energy density of states of near-extremal black holes*, which rises roughly exponentially with a power-law prefactor, by providing an explicit example of such a spectrum [8].

- The **dynamics of the black hole horizon** (how it fluctuates and responds to perturbations) are mirrored by the **collective modes in SYK**. For example, an AdS_2 black hole has a set of so-called **boundary graviton** modes – related to time reparameterizations – which control how the black hole returns to equilibrium after a perturbation. These are directly encoded by the Schwarzian effective theory on the SYK side.

- **Spatial structure:** While SYK has no spatial extent, one can consider coupled SYK models to mimic an AdS wormhole or multiple black hole system. For instance, two SYK models coupled in a certain way have been shown to be dual to an eternal traversable wormhole in AdS_2 – literally two black holes connected by a throat (an example of the ER=EPR concept). This, though beyond the scope of this discussion, highlights how versatile the SYK-as-CFT picture is in exploring quantum gravity scenarios.

Perhaps one of the most compelling holographic demonstrations is that one can use the dual SYK model to **reconstruct the black hole interior** – a task which directly addresses the information paradox. Recent research explicitly showed how an observer's measurements in the SYK model can be translated into observations behind the horizon of the dual AdS_2 black hole. In the paper *"Seeing behind black hole horizons in SYK,"* Gao and colleagues outline a procedure to recover interior information using only boundary (SYK) degrees of freedom. This remarkable result leverages the SYK model's control to perform thought-experiments that would be impossible with

real black holes, thereby deepening our understanding of how information might not be lost behind horizons.

Implications for the Information Paradox: The SYK model's duality to a black hole provides strong evidence that black hole information loss is avoided in quantum gravity. Since the SYK model is a manifestly unitary quantum system (it has a well-defined Hilbert space and time evolution), any process in SYK – no matter how scrambling – is fundamentally information-preserving. This maps to the statement that the AdS_2 black hole, described by JT gravity, is also unitary: information that falls in will be encoded in later correlation functions and can in principle be recovered. The AdS/CFT correspondence thus assures that black holes *do not* violate quantum mechanics. Instead, they rapidly scramble information, making it practically inaccessible but not destroyed. The SYK model serves as a concrete example where we can **see explicitly how unitarity is preserved**: the initial information is hidden in complicated many-body entanglement (analogous to Hawking radiation carrying information in extremely scrambled form).

Conclusion

The Sachdev–Ye–Kitaev model has revolutionized our theoretical approach to quantum chaos and black hole physics. By providing a solvable model that mirrors the most extreme scrambling properties of black holes, SYK allows physicists to test ideas about gravity and information in a controlled setting. We see that concepts such as **quantum chaos** (characterized by OTOC growth and Lyapunov exponents) and **information scrambling** (the delocalization of quantum information) take on precise, calculable forms in SYK – and these align almost exactly with what we expect for a quantum black hole. Through the lens of **AdS/CFT holography**, SYK's behavior is understood as that of a dual $(1+1)-$dimensional black hole in AdS_2, specifically a Jackiw-Teitelboim gravity system [9]. Virtually every hallmark of black hole physics – rapid thermalization, maximal entropy, lack of quasiparticles (no hair), maximal Lyapunov spectrum, fast scrambling – finds its counterpart in the SYK model [6] [5].

This convergence has significant implications. It bolsters the idea that black hole interiors and quantum gravitational dynamics can be decoded by studying quantum many-body systems. It suggests that the mysterious aspects of black holes (like the information paradox) are not paradoxes at all when viewed correctly – they are natural consequences of quantum chaos and complexity, as epitomized by SYK. Indeed, the SYK model shows in detail how a unitary theory can thermalize and scramble information in a way that, to a naive observer, looks thermal and irreversible (much as a black hole's Hawking radiation looks thermal, yet in the broader picture must encode the information of what fell in).

In summary, **the SYK model mimics black hole information scrambling by being a maximally chaotic, fast-scrambling quantum system with a holographic gravity dual**. Its theoretical framework ties together random matrix theory, quantum chaos diagnostics like OTOCs, and the AdS/CFT correspondence into one coherent picture. The insight gleaned is profound: even the most enigmatic object in physics – a black hole – can be studied and understood through the physics of quantum entanglement and chaos in systems like SYK [5]. This synergy between condensed matter theory and high-energy physics is a prime example of how progress in one

domain (here, strongly correlated electrons or spins) can illuminate the deep mysteries of another (quantum gravity). As research continues, SYK and its variants will likely remain at the forefront of exploring how spacetime geometry, quantum information, and chaos are interwoven at the fundamental level [5] [9].

Dictionary of difficult items in no particular order:

Majorana fermions are special particles that are their own antiparticles. This means that they can annihilate themselves, which is quite unique compared to other particles.

The SYK model (named after its creators Sachdev, Ye, and Kitaev) is a theoretical model used in physics to study complex systems. It's particularly interesting because it helps scientists understand quantum mechanics and chaos.

In the context of the SYK model, Majorana fermions are used to represent the particles in the system. The SYK model uses these fermions to explore how particles interact with each other in a chaotic environment. This model has been useful in studying black holes and quantum computers because it provides insights into how information behaves in these systems.

So, Majorana fermions are unique particles that help scientists use the SYK model to understand complex and chaotic systems, which has implications for studying black holes and advancing quantum computing.

Quantum chaos is a field of study that looks at how chaotic systems behave at the quantum level. Chaos, in general, refers to systems that are highly sensitive to initial conditions, meaning that small changes can lead to vastly different outcomes. Think of it like the butterfly effect, where a butterfly flapping its wings could eventually cause a tornado.

In the quantum world, things get even more interesting. Quantum chaos examines how particles, like electrons, behave in systems that are chaotic. Unlike classical chaos, where we can predict the behavior of systems using equations, quantum chaos involves the strange and unpredictable behavior of particles governed by the rules of quantum mechanics.

One key aspect of quantum chaos is the idea that even though quantum systems are governed by deterministic equations, their behavior can still appear random and unpredictable due to the complex interactions between particles. This has implications for understanding phenomena like the behavior of electrons in atoms, the dynamics of black holes, and even the development of quantum computers.

So, quantum chaos is the study of how particles behave in chaotic systems at the quantum level, revealing the unpredictable and fascinating nature of the quantum world.

A **thermal AdS_2 black hole** is a special type of black hole that exists in a **two-dimensional** version of anti-de Sitter (AdS) space. AdS space is a curved universe with a negative cosmological constant, meaning it has a kind of "gravitational well" that keeps things from escaping too easily.

Breaking it Down Simply

1. **Why AdS_2?**
    - In higher dimensions, black holes have event horizons that trap light and matter.
    - In **AdS_2**, which is just a **1+1 dimensional** spacetime (one space and one time dimension), black holes behave differently but still follow similar thermodynamic rules.
2. **What Makes It "Thermal"?**
    - A thermal AdS_2 black hole is one that has **temperature**, meaning it emits radiation (like Hawking radiation).
    - This temperature is related to how fast the black hole scrambles information and how it interacts with its surroundings.
3. **Why Is It Important?**
    - AdS_2 black holes are closely related to **quantum gravity** and **holography** (AdS/CFT correspondence).
    - They help physicists understand **how black holes process and scramble information**, which is crucial for solving the **black hole information paradox**.

Real-World Analogy

Imagine a **tiny whirlpool** in a deep ocean. The whirlpool traps water inside, but over time, some water slowly leaks out due to turbulence. The thermal AdS_2 black hole is like that whirlpool—it traps information but also lets some escape in a highly structured way.

Bekenstein–Hawking entropy is a concept from physics that helps us understand black holes. Imagine a black hole as a giant vacuum cleaner in space that sucks in everything around it, including light. Now, scientists wanted to figure out how much "information" is hidden inside a black hole.

Bekenstein–Hawking entropy tells us that the amount of information inside a black hole is proportional to the surface area of its event horizon (the boundary beyond which nothing can escape). In simpler terms, it's like saying the more "stuff" a black hole has sucked in, the bigger its surface area gets, and the more information it contains.

So, Bekenstein–Hawking entropy is a way to measure the "hidden secrets" inside a black hole based on its size. It's a fascinating concept that combines ideas from quantum mechanics and general relativity!

Let's break down **holographic duality** in simple terms.

Imagine you have a hologram, like those cool 3D images you see on credit cards or movie posters. A hologram is a flat surface that can display a three-dimensional image. Now, think of the universe as a giant hologram.

Holographic duality is a concept in physics that suggests that everything happening in a certain space (like inside a black hole) can be described by information on a lower-dimensional boundary (like the surface of the black hole). In other words, the complex, three-dimensional events inside space can be represented by simpler, two-dimensional information on the boundary.

This idea helps physicists understand how gravity and quantum mechanics work together. It's like having a cheat sheet that simplifies the complex interactions inside a space by using information on its surface. This concept has been particularly useful in studying black holes and the nature of the universe.

So, in layman's terms, holographic duality is the idea that the complex events happening in a space can be represented by simpler information on its boundary, much like how a hologram displays a 3D image on a flat surface.

Let's break down the **Information Paradox** in simple terms.

Imagine you have a book with all the information about a certain topic. Now, let's say you throw that book into a black hole. According to classical physics, once something falls into a black hole, it gets crushed and lost forever. But here's the problem: the information in that book can't just disappear. This is because one of the fundamental principles of physics is that information must be conserved.

The **Information Paradox** arises because black holes seem to destroy information, which contradicts the principle of information conservation. Scientists have been puzzled by this for a long time. If information is truly lost in a black hole, it would mean that the laws of physics are incomplete.

To solve this paradox, physicists have proposed various theories. One idea is that the information might be stored on the surface of the black hole (the event horizon) and could be released back into the universe when the black hole evaporates. Another theory suggests that information might be encoded in the radiation emitted by the black hole (Hawking radiation).

So, the Information Paradox is the puzzle of how information can be conserved when it seems to be lost in a black hole. Scientists are still working on understanding this mystery and finding ways to reconcile it with the laws of physics.

The principle of physics that states information must be conserved is known as the **principle of information conservation**. This principle is closely related to the **law of conservation of energy** and the **law of conservation of mass**, which state that energy and mass cannot be created or destroyed, only transformed from one form to another.

In the context of information, this principle means that the information about the state of a physical system must be preserved over time. In other words, if you know the complete information about a system at one point in time, you should, in theory, be able to determine its past and future states. This is a fundamental concept in quantum mechanics and classical physics.

The Information Paradox challenges this principle because it suggests that information could be lost in a black hole, which would violate the idea that information must be conserved.

Let's break down **black hole information scrambling** in simple terms.

Imagine you have a blender, and you throw in a bunch of different fruits to make a smoothie. Once you blend everything together, it's almost impossible to separate the individual fruits again. This is similar to what happens with information in a black hole.

When something falls into a black hole, the information about that object gets mixed up and spread out across the black hole's event horizon (the boundary around the black hole). This process is called **information scrambling**. It's like the blender mixing up the fruits, making it very difficult to figure out what went in originally.

Information scrambling is important because it helps explain how information might be preserved in a black hole, despite the Information Paradox. Even though the information gets mixed up, it's still there, just in a very scrambled form. Scientists believe that this scrambled information could eventually be released back into the universe through Hawking radiation (the radiation emitted by black holes).

So, in layman's terms, black hole information scrambling is the process of mixing up and spreading out information across the event horizon of a black hole, making it difficult to figure out what went in originally, but still preserving the information in a scrambled form.

Let's break down **entanglement entropy** in simple terms.

Imagine you have two particles that are like best friends. When these particles become "entangled," it means that whatever happens to one particle instantly affects the other, no matter how far apart they are. This special connection is called "quantum entanglement."

Now, "entanglement entropy" is a way to measure how much information is shared between these two entangled particles. Think of it like this: if you have two friends who share a lot of secrets, the amount of secrets they share is like the entanglement entropy. The more secrets (or information) they share, the higher the entanglement entropy.

The maximum value of entanglement entropy depends on the number of particles or subsystems involved. In general, for a system with (N) particles, the maximum entanglement entropy is given by ($log_2(N)$). This means that the more particles or subsystems you have, the higher the maximum entanglement entropy can be.

In simple terms, entanglement entropy tells us how strongly two particles are connected and how much information they share with each other. It's a fascinating concept that helps scientists understand the mysterious world of quantum mechanics!

The quantum code for the SYK model and Black hole information scrambling

DeepSeek initially provided code that utilized the Suzuki-Trotter method. However, this step resulted in a non-unitary final circuit, so the code was modified to exclude it.

*SIDE NOTE: **The Suzuki-Trotter method** is a technique used in quantum computing to simulate the evolution of a quantum system over time. Let's break it down in simple terms:*

Imagine you have a very complex recipe that involves mixing several ingredients in a specific way. However, instead of following the entire recipe all at once, you decide to break it down into smaller, simpler steps. You mix a few ingredients together, then mix another set, and so on, until you've completed the recipe. This makes the process more manageable and easier to follow.

In the context of quantum computing, the Suzuki-Trotter method does something similar. It takes a complex quantum operation (which describes how a quantum system evolves over time) and breaks it down into a series of simpler operations. These simpler operations are easier to implement on a quantum computer.

By repeating these simpler operations many times, the Suzuki-Trotter method approximates the overall effect of the complex operation. This approach helps in accurately simulating the behavior of quantum systems, such as how particles interact and evolve, without having to deal with the full complexity all at once.

In summary, the Suzuki-Trotter method is like breaking down a complicated recipe into smaller, more manageable steps, making it easier to simulate the evolution of a quantum system on a quantum computer.

The unitarity of a quantum circuit is crucial for several reasons:

1. **Preservation of Probability**: In quantum mechanics, the total probability of all possible outcomes must always sum to one. Unitary operations ensure that this condition is met by preserving the norm of the quantum state vector.
2. **Reversibility**: Unitary operations are reversible, meaning that if you apply a unitary operation to a quantum state, you can always apply its inverse to return to the original state. This is essential for quantum computing, where operations must be reversible to ensure accurate computation and error correction.

3. **Consistency with Quantum Theory**: Quantum mechanics is fundamentally based on unitary evolution. The Schrödinger equation, which governs the time evolution of quantum states, describes unitary transformations. Therefore, any quantum circuit must be unitary to be consistent with the principles of quantum mechanics.
4. **Information Preservation**: Unitary operations preserve information, which is vital for quantum algorithms and quantum communication. This ensures that no information is lost during the computation process, allowing for reliable and accurate results.

To give you a full picture, let's dive into how unitary operations are implemented in quantum circuits.

In quantum computing, unitary operations are represented by unitary matrices. These matrices are used to transform quantum states, which are represented by vectors in a complex vector space. The key properties of unitary matrices are that they preserve the inner product of vectors and their inverse is equal to their conjugate transpose.

Quantum Gates

Quantum gates are the building blocks of quantum circuits, analogous to classical logic gates in traditional computing. Each quantum gate corresponds to a unitary matrix that performs a specific operation on qubits (quantum bits). Here are some common quantum gates:

1. **Hadamard Gate (H)**: This gate creates superposition by transforming a qubit from the basis state $|0\rangle$ to $(|0\rangle + |1\rangle)/\sqrt{2}$ and from $|1\rangle$ to $(|0\rangle - |1\rangle)/\sqrt{2}$.
2. **Pauli-X Gate (X)**: This gate acts like a classical NOT gate, flipping the state of a qubit from $|0\rangle$ to $|1\rangle$ and vice versa.
3. **Pauli-Y Gate (Y)**: This gate performs a bit-flip and a phase-flip simultaneously.
4. **Pauli-Z Gate (Z)**: This gate flips the phase of the qubit, changing the sign of the $|1\rangle$ state.
5. **CNOT Gate (CX)**: This is a two-qubit gate that flips the state of the target qubit if the control qubit is in the state $|1\rangle$.

Quantum Circuits

A quantum circuit is a sequence of quantum gates applied to a set of qubits. The overall operation of the circuit is described by the product of the unitary matrices representing each gate. Because the product of unitary matrices is also unitary, the entire circuit remains unitary.

Implementation

To implement unitary operations in a quantum circuit, you need to:

1. **Define the Quantum Gates**: Specify the unitary matrices for the gates you will use.
2. **Apply the Gates**: Arrange the gates in the desired sequence to form the quantum circuit.

3. **Measure the Qubits**: After applying the gates, measure the qubits to obtain the output state.

Now, we are ready to break the final quantum code down in the context of the SYK model and black hole information scrambling:

1. **Setting Up the Quantum System**: The code starts by creating a quantum system that represents the SYK model. The SYK model is a theoretical model used to study complex interactions between particles, and it's known for mimicking the behavior of black holes, especially how they scramble information.
2. **Random Interactions**: In the SYK model, particles interact with each other in random ways. The code generates these random interactions, which are like the random mixing of ingredients in a blender. This randomness is crucial for studying how information gets scrambled.
3. **Building the Hamiltonian**: The Hamiltonian is a mathematical representation of the energy and interactions within the system. Think of it as a recipe that tells us how the particles interact and evolve over time.
4. **Converting to Qubits**: The Hamiltonian is then converted into a form that can be used with qubits, the basic units of quantum information. This step is like translating the recipe into a language that a quantum computer can understand.
5. **Evolving the System**: The code then evolves the quantum system over time, applying a series of operations that simulate the interactions described by the Hamiltonian. This evolution is like running the blender for a certain amount of time, mixing the ingredients thoroughly.
6. **Measuring Entanglement**: Finally, the code measures the entanglement entropy, which tells us how much the information has been mixed up and shared between different parts of the system. In the context of black holes, this is like measuring how well the blender has mixed the fruits, making it difficult to figure out what went in originally.

In summary, this code sets up a quantum system based on the SYK model, evolves it over time, and measures how the information within the system gets scrambled. It's a way to study the complex behavior of black holes and understand how they mix up information.

Parameters

```
# Parameters

N_majorana = 6  # 6 Majorana fermions (3 qubits)

J_strength = 10000.0  # Coupling strength

t = 8 * np.pi    # Evolution time

trotter_steps = 20 # Trotter steps for accuracy
```

Let's break down these parameters for the SYK (Sachdev-Ye-Kitaev) model:

1. **N_majorana = 6**: This parameter indicates that there are 6 Majorana fermions in the system. Majorana fermions are particles that are their own antiparticles. In the context of quantum computing, 6 Majorana fermions can be mapped to 3 qubits, as each qubit can be represented by a pair of Majorana fermions.
2. **J_strength = 10000.0**: This parameter represents the coupling strength between the Majorana fermions. The coupling strength determines the interaction energy between the fermions and plays a crucial role in the dynamics of the system. A higher coupling strength typically leads to stronger interactions.
3. **t = 8 * π**: This parameter specifies the evolution time of the system. In quantum mechanics, the evolution time determines how long the system evolves under a given Hamiltonian (the operator corresponding to the total energy of the system). The value 8 * π indicates that the system evolves for a time period of 8π units.
4. **trotter_steps = 20**: This parameter refers to the number of Trotter steps used for the simulation. The Trotter-Suzuki decomposition is a method used to approximate the exponential of a sum of non-commuting operators. By breaking the evolution time into smaller steps (Trotter steps), the accuracy of the simulation is improved. In this case, 20 Trotter steps are used to achieve a more accurate approximation of the system's evolution.

These parameters for the SYK model have a significant impact on the simulation in various ways:

1. **N_majorana (Number of Majorana Fermions)**: The number of Majorana fermions determines the complexity of the system. With 6 Majorana fermions, the system can be mapped to 3 qubits. The more Majorana fermions you have, the more complex the interactions and the larger the Hilbert space you need to consider. This affects the computational resources required for the simulation.
2. **J_strength (Coupling Strength)**: The coupling strength influences the interaction energy between the Majorana fermions. A higher coupling strength means stronger interactions, which can lead to more pronounced quantum effects and potentially more chaotic behavior. This parameter is crucial for studying the dynamics and properties of the system, such as entanglement and information scrambling.
3. **t (Evolution Time)**: The evolution time determines how long the system evolves under the given Hamiltonian. Longer evolution times allow you to observe the system's behavior over extended periods, which can reveal long-term dynamics and stability. However, longer evolution times also require more computational steps to accurately simulate the system's evolution.
4. **trotter_steps (Trotter Steps)**: The number of Trotter steps affects the accuracy of the simulation. The Trotter-Suzuki decomposition approximates the exponential of a sum of non-commuting operators by breaking the evolution time into smaller steps. More Trotter steps result in a more accurate approximation, but they also increase the computational complexity. Finding the right balance between accuracy and computational efficiency is key.

Overall, these parameters collectively determine the behavior, complexity, and accuracy of the SYK model simulation. Adjusting them allows researchers to explore different aspects of the system and gain insights into its quantum properties.

As previously mentioned, the Suzuki-Trotter method resulted in a non-unitary final circuit, so the code was modified to exclude it. In the updated code, the parameter **trotter_steps** determines the length of the unitary evolution:

```
u_pauli = Operator(evolution_gate)
    for i in range(trotter_steps-1):
      u_pauli = Operator(evolution_gate) @ u_pauli
```

This replaces the following step:

```
evolution_gate = PauliEvolutionGate(
     sparse_pauli_op,
     time=t,
     label=label,  # Custom label with coefficients
    #    synthesis=SuzukiTrotter(order=2, reps=trotter_steps) # this step does not produce a unitary circuit!
    )
```

Below is the complete code listing.

```
def get_transpiled_circuit_custom(data,n):
    unitary_matrix=data
    # Create a UnitaryGate from the matrix
    unitary_gate = UnitaryGate(Operator(unitary_matrix))

    # Create a QuantumCircuit and add the unitary gate
    qc = QuantumCircuit(n)  # Adjust the number of qubits as needed
    qc.append(unitary_gate, range(n))

    # Transpile the circuit to decompose the unitary into standard gates
    transpiled_circuit = transpile(qc, basis_gates=['u3', 'cx'], optimization_level=3)

    # Print the transpiled circuit
    #print("transpiled circuit depth:",transpiled_circuit.depth())
    #print("Transpiled Circuit Display:")
    #print(transpiled_circuit)
    return transpiled_circuit

# Parameters
```

```
N_majorana = 6  # 6 Majorana fermions (3 qubits)
J_strength = 10000.0  # Coupling strength
t = 8 * np.pi    # Evolution time
trotter_steps = 20 # Trotter steps for accuracy

def syk6_scramble():
    # Generate random SYK couplings (Gaussian distribution)
    terms = list(combinations(range(N_majorana), 4))
    J = {}
    for term in terms:
        J_val = np.random.normal(scale=J_strength * np.sqrt(6) / (N_majorana**1.5))
        J[term] = J_val

    # Build SYK Hamiltonian
    H = MajoranaOperator()
    for indices, J_val in J.items():
        H += MajoranaOperator(
        indices,
        coefficient=J_val
        )
    # Convert to qubit Hamiltonian via Jordan-Wigner
    H_qubit = jordan_wigner(H)

    # Convert QubitOperator to SparsePauliOp
    pauli_terms = []
    for term, coeff in H_qubit.terms.items():
        pauli_str = ['I'] * 3
        for (index, op) in term:
            pauli_str[index] = op
        pauli_terms.append((''.join(pauli_str), coeff))

    # Handle empty Hamiltonian
    if pauli_terms:
        sparse_pauli_op = SparsePauliOp.from_list(pauli_terms).simplify()
    else:
        sparse_pauli_op = SparsePauliOp(['III'], coeffs=[0])
```

```
# Initialize quantum circuit
qc = QuantumCircuit(3)
qc.h([0, 1, 2])  # Prepare |+++> initial state

# Apply evolution with custom labels
if sparse_pauli_op.size > 0 and not np.allclose(sparse_pauli_op.coeffs, 0):
    pauli_list = sparse_pauli_op.paulis
    coeffs = sparse_pauli_op.coeffs

    # Generate custom label
    label = "H = "
    for pauli, coeff in zip(pauli_list, coeffs):
        label += f"{coeff.real:.2f} {pauli.to_label()} +"
    label[:-1] #drop last +

    evolution_gate = PauliEvolutionGate(
      sparse_pauli_op,
      time=t,
      label=label,  # Custom label with coefficients
    #    synthesis=SuzukiTrotter(order=2, reps=trotter_steps) # this step does not produce a unitary circuit!
    )

    u_pauli = Operator(evolution_gate)
    for i in range(trotter_steps-1):
        u_pauli = Operator(evolution_gate) @ u_pauli

    transc = get_transpiled_circuit_custom(u_pauli.data,3)
    qc.compose(transc, [0, 1, 2], inplace=True)
else:
    print("No evolution applied.")

# Compute entanglement entropy
state = Statevector(qc)
rho = partial_trace(state,[1,2])
entanglement_entropy = entropy(rho, base=2)
```

```
#print(f"Entanglement entropy: {entanglement_entropy:.4f}")

return entanglement_entropy
```

get_transpiled_circuit_custom(data, n)

This function takes a unitary matrix (data) and the number of qubits (n) as inputs and performs the following steps:

1. **Create a Unitary Gate**: It creates a UnitaryGate from the provided unitary matrix.
2. **Create a Quantum Circuit**: It initializes a quantum circuit with n qubits and appends the unitary gate to the circuit.
3. **Transpile the Circuit**: It transpiles the quantum circuit to decompose the unitary gate into standard quantum gates (u3 and cx) with an optimization level of 3. Transpiling is the process of converting a quantum circuit into an equivalent circuit that can be executed on a quantum computer.
4. **Return the Transpiled Circuit**: It returns the transpiled quantum circuit.

syk6_scramble()

This function simulates the scrambling dynamics of the SYK model with 6 Majorana fermions (mapped to 3 qubits) and performs the following steps:

1. **Generate Random SYK Couplings**: It generates random couplings for the SYK model using a Gaussian distribution.
2. **Build SYK Hamiltonian**: It constructs the SYK Hamiltonian using the generated couplings.
3. **Convert to Qubit Hamiltonian**: It converts the SYK Hamiltonian to a qubit Hamiltonian using the Jordan-Wigner transformation.
4. **Convert to SparsePauliOp**: It converts the qubit Hamiltonian to a SparsePauliOp representation.
5. **Initialize Quantum Circuit**: It initializes a quantum circuit with 3 qubits and prepares the initial state $|+++\rangle$.
6. **Apply Evolution with Custom Labels**: It applies the evolution operator to the quantum circuit using the PauliEvolutionGate with custom labels. If the Suzuki-Trotter method is not used, it manually constructs the unitary evolution operator.
7. **Transpile and Compose Circuit**: It uses the get_transpiled_circuit_custom function to transpile the unitary evolution operator and composes it into the quantum circuit.
8. **Compute Entanglement Entropy**: It computes the entanglement entropy of the resulting quantum state by tracing out two of the three qubits.

These functions are used to simulate the dynamics of the SYK model and analyze the entanglement properties of the system.

Entanglement Entropy

As previously mentioned, "entanglement entropy" is a way to measure how much information is shared between entangled particles.

```
# Compute entanglement entropy
  state = Statevector(qc)
  rho = partial_trace(state,[1,2])
  entanglement_entropy = entropy(rho, base=2)
```

Let's break down what these three lines of code do in the context of computing entanglement entropy:

1. **state = Statevector(qc)**: This line creates a Statevector object from the quantum circuit qc. The Statevector represents the quantum state of the entire system after the quantum circuit has been applied. Essentially, it captures the final state of the qubits in the circuit.
2. **rho = partial_trace(state, [1, 2])**: This line computes the partial trace of the Statevector over qubits 1 and 2. The partial trace is a mathematical operation that traces out (or ignores) certain qubits, leaving a reduced density matrix for the remaining qubits. In this case, it traces out qubits 1 and 2, leaving the reduced density matrix rho for qubit 0. This reduced density matrix describes the state of qubit 0 after ignoring the other qubits.
3. **entanglement_entropy = entropy(rho, base=2)**: This line calculates the entanglement entropy of the reduced density matrix rho. **The entanglement entropy is a measure of the quantum entanglement between the traced-out qubits (qubits 1 and 2) and the remaining qubit (qubit 0).** The entropy function computes the von Neumann entropy of rho, which quantifies the amount of entanglement. The base=2 parameter specifies that the entropy should be calculated using base-2 logarithms, resulting in the entanglement entropy being measured in bits.

In summary, these lines of code compute the entanglement entropy of the quantum state represented by the quantum circuit qc, focusing on the entanglement between qubit 0 and the rest of the system. This measure provides insight into the degree of quantum entanglement present in the system.

Higher amounts of entanglement entropy in the context of black hole information scrambling indicate a greater degree of information scrambling within the black hole. As previously mentioned, information scrambling refers to the process by which information that falls into a black hole becomes distributed and entangled among the particles inside the black hole, making it difficult to retrieve or reconstruct.

In practical terms, higher entanglement entropy means that the information initially localized in one part of the system (such as the information carried by an infalling particle) becomes more thoroughly mixed and spread out across the entire system. This has several implications:

1. **Increased Complexity**: Higher entanglement entropy signifies a more complex and intertwined quantum state within the black hole. The information becomes more deeply embedded in the quantum correlations among the particles, making it harder to extract.

2. **Enhanced Information Scrambling**: Greater entanglement entropy indicates stronger information scrambling, where the information is rapidly and extensively distributed throughout the black hole. This is relevant for understanding the dynamics of black holes and their role in quantum chaos.

3. **Challenges for Information Retrieval**: Higher entanglement entropy makes it more challenging to retrieve information from the black hole. The more entangled the state, the more difficult it is to reconstruct the original information from the scrambled quantum correlations.

4. **Implications for the Black Hole Information Paradox**: The black hole information paradox arises from the question of whether information that falls into a black hole is lost forever or can be recovered. Higher entanglement entropy suggests that while information may not be lost, it becomes extremely difficult to recover due to the extensive scrambling.

Overall, higher entanglement entropy in the context of black hole information scrambling indicates a more complex and thoroughly mixed quantum state, making information retrieval more challenging and providing insights into the nature of quantum chaos and the black hole information paradox.

Testing

To illustrate the corresponding quantum circuit:

In [16]:
```
import matplotlib.pyplot as plt

qc.draw(output='mpl', fold=100)
```

Out[16]:

In [14]:
```
# Measurement
qc.measure_all()

# Simulate
simulator = Aer.get_backend('qasm_simulator')
result = execute(qc, simulator, shots=2000).result()
counts = result.get_counts(qc)
print("Measurement counts:", counts)

# Circuit info
print("\nCircuit depth:", qc.depth())
print("Number of Hamiltonian terms:", sparse_pauli_op.size)
```

```
Measurement counts: {'100': 127, '010': 273, '000': 287, '110': 287, '011': 366, '101': 47, '111': 364, '001': 249}

Circuit depth: 41
Number of Hamiltonian terms: 15
```

Notice a high circuit depth (41). A circuit depth of 41 indicates the number of layers of gates in the quantum circuit. In other words, it represents the longest path from the input to the output of the circuit, measured in terms of the number of gates applied sequentially.

Higher circuit depth can have several implications:

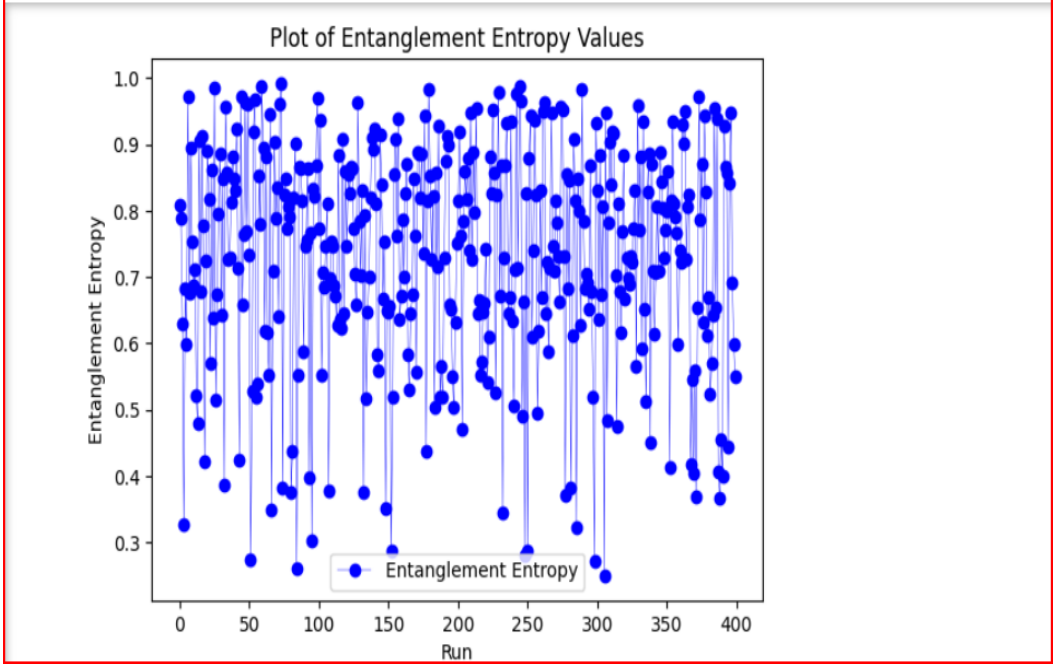
1. **Complexity**: A greater depth often means a more complex circuit, which may be necessary for performing more sophisticated quantum operations.
2. **Execution Time**: The depth of the circuit can affect the execution time on a quantum computer. Deeper circuits may take longer to run and may be more susceptible to errors due to decoherence and other noise factors.
3. **Resource Requirements**: Higher depth circuits may require more computational resources, both in terms of qubits and classical processing for error correction and optimization.
4. **Accuracy**: While deeper circuits can perform more complex operations, they also need to be carefully optimized to minimize errors and ensure accurate results.

The **syk6_scramble()** function was executed 400 times, and in each iteration, the calculated entanglement entropy was appended to a list (ent_list):

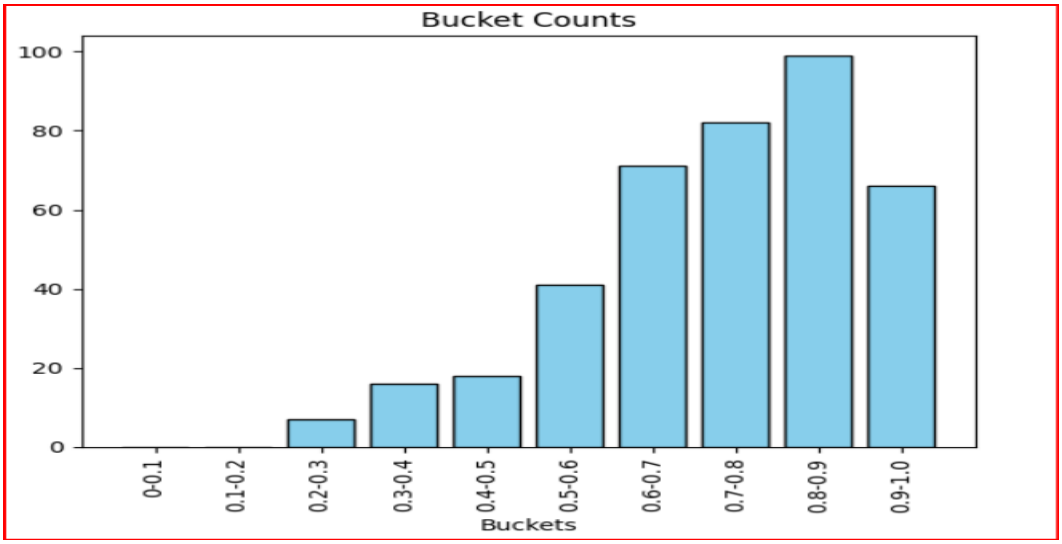
```
[64]: ent_list=[]
      for i in range(400):
          ent_list.append(round(syk6_scramble(),4))
```

As expected, the plot below illustrates the prevalence of high values of entanglement entropy.

```
[66]: import matplotlib.pyplot as plt

# Plot the values
plt.plot(ent_list, marker='o', linestyle='-', color='b', linewidth=0.3, label='Entanglement Entropy')

# Add labels and title
plt.xlabel('Run')
plt.ylabel('Entanglement Entropy')
plt.title('Plot of Entanglement Entropy Values')
plt.legend()

# Show the plot
plt.show()
```



Values in the higher buckets (0.6 and above) are prevalent.



The prevalence of high values of entanglement entropy illustrates how the SYK model replicates the information scrambling in black holes. In the context of black holes, as previously mentioned, information scrambling refers to the process by which information that falls into a black hole becomes rapidly and extensively distributed throughout the system, making it difficult to retrieve. The initial state $|{+}{+}{+}\rangle$ did not have any entanglement entropy before scrambling.

```
[9]: # Compute entanglement entropy before scambling
     state = Statevector(qc)
     print(state)
     rho = partial_trace(state,[1,2])
     entanglement_entropy = entropy(rho, base=2)
     print(f"Entanglement entropy: {entanglement_entropy:.4f}")

     Statevector([0.35355339+0.j, 0.35355339+0.j, 0.35355339+0.j,
                  0.35355339+0.j, 0.35355339+0.j, 0.35355339+0.j,
                  0.35355339+0.j, 0.35355339+0.j],
                 dims=(2, 2, 2))
     Entanglement entropy: 0.0000
```

The SYK model replicates this process by demonstrating how information initially localized in one part of the system becomes entangled and spread out across the entire system over time.

In the context of black holes, the SYK model serves as a toy model for understanding aspects of quantum gravity and holography. It shares similarities with the near-horizon dynamics of extremal black holes, particularly in how information is scrambled and lost to an observer outside the event horizon. The model's chaotic behavior mimics the way black holes process and obscure information, reinforcing ideas from the AdS/CFT correspondence—a theoretical framework linking quantum gravity and quantum field theory.

Summary:

Overall, the code provides a useful framework for simulating the SYK model and analyzing information scrambling, but it has limitations in terms of scalability, accuracy, and computational complexity. Further improvements and optimizations could enhance its effectiveness for larger and more complex systems.

The code aims to simulate the scrambling dynamics of the SYK model with 6 Majorana fermions (mapped to 3 qubits) and compute the entanglement entropy. Here's a breakdown of the key steps:

1. **Random SYK Couplings**: The syk6_scramble() function generates random couplings for the SYK model using a Gaussian distribution. These couplings are used to build the SYK Hamiltonian, which describes the interactions between the Majorana fermions.

2. **Hamiltonian Construction**: The SYK Hamiltonian is constructed using the generated couplings and then converted to a qubit Hamiltonian via the Jordan-Wigner transformation. This transformation maps the Majorana fermions to qubits.

3. **SparsePauliOp Representation**: The qubit Hamiltonian is converted to a SparsePauliOp representation, which is a compact way to represent the Hamiltonian in terms of Pauli operators.

4. **Quantum Circuit Initialization**: A quantum circuit with 3 qubits is initialized, and the initial state $|+++\rangle$ is prepared using Hadamard gates.

5. **Evolution Operator**: The evolution operator is applied to the quantum circuit using the PauliEvolutionGate with custom labels. **The Suzuki-Trotter method is excluded to ensure the final circuit remains unitary.**

6. **Transpilation**: The get_transpiled_circuit_custom() function is used to transpile the unitary evolution operator into standard quantum gates (u3 and cx) and compose it into the quantum circuit.

7. **Entanglement Entropy Calculation**: The entanglement entropy is computed by tracing out two of the three qubits and calculating the von Neumann entropy of the reduced density matrix.

Limitations of the Code

1. **Scalability**: The code is limited to simulating a small number of qubits (3 qubits in this case). Scaling up to larger systems with more Majorana fermions and qubits would require significant computational resources and may not be feasible with the current approach.

2. **Accuracy of Trotter Steps**: While the code uses 20 Trotter steps for accuracy, the Trotter-Suzuki decomposition is an approximation. Increasing the number of Trotter steps improves accuracy but also increases computational complexity. Finding the optimal balance between accuracy and efficiency is challenging.

3. **Handling Empty Hamiltonian**: The code includes a check for an empty Hamiltonian and handles it by setting the SparsePauliOp to a trivial operator. This is a necessary safeguard, but it may not fully capture the dynamics of the system in cases where the Hamiltonian is sparse or has very small coefficients.

4. **Transpilation Complexity**: The transpilation process decomposes the unitary gate into standard gates, which can be computationally intensive. The optimization level (set to 3) aims to reduce the circuit depth, but this may not always result in the most efficient circuit for execution on a quantum computer.

5. **Limited Analysis**: The code focuses on computing entanglement entropy as a measure of information scrambling. While entanglement entropy is a valuable metric, additional analyses (such as studying other quantum properties or performing more detailed statistical analyses) could provide deeper insights into the SYK model and information scrambling.

Please feel free to download the notebook and further explore:

syk/SYK6.ipynb at main · samlip-blip/syk

In the next chapter we discuss the use of random graphs, particularly the Erdős-Rényi model, to simulate quantum information spreading in black holes. The chapter explains how researchers apply entangling gates to qubits represented in a random graph to study information scrambling, with entanglement entropy serving as a measure of this process.

Random Graphs and Information Spreading Inside a Black Hole

We are continuing our thrilling journey from the previous chapter, where we will delve into how quantum information spreads within a black hole using random graphs. We can model qubit interactions with an Erdős-Rényi random graph, where each node represents a qubit and edges indicate where entangling gates (e.g., CNOT) are applied. The resulting entanglement entropy measures the spread of information, similar to the scrambling process in black holes. But first, let's explore what random graphs are.

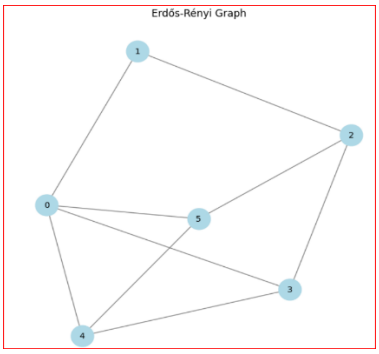
Random Graphs

Random graphs are a type of graph where edges between nodes are formed randomly. One common model for generating random graphs is the Erdős-Rényi model, where each possible edge between a pair of nodes is included with a fixed probability.

Applications of Random Graphs:

1. Network Theory: Random graphs are used to model and analyze various types of networks, such as social networks, communication networks, and biological networks.
2. Epidemiology: They help in understanding the spread of diseases by modeling how infections can propagate through a population.
3. Computer Science: Random graphs are used in algorithms and complexity theory, particularly in the study of randomized algorithms and the behavior of networks.
4. Physics: They are used to study phenomena such as percolation and phase transitions.
5. Quantum Computing: As mentioned earlier, random graphs can model qubit interactions and help understand quantum information spread and entanglement.

Here is an example of an Erdős-Rényi random graph (6 nodes or vertices, p = 0.6 edge fixed probability):



Random graphs are used to model the spread of quantum information inside a black hole by representing the interactions between qubits. Here's how it works:

1. Erdős-Rényi Random Graphs: In this model, each node represents a qubit, and edges between nodes are formed randomly with a certain probability. This randomness helps simulate the complex and chaotic environment inside a black hole.
2. Entangling Gates: The edges in the graph indicate where entangling gates, such as Controlled-NOT (CNOT) gates, are applied. These gates create entanglement between qubits, which is essential for quantum information processing.
3. Entanglement Entropy: As the entangling gates are applied, the qubits become increasingly entangled. The entanglement entropy, which measures the degree of entanglement, quantifies how information spreads through the system. In the context of a black hole, this spread of information is analogous to the scrambling process, where information that falls into the black hole becomes rapidly mixed and dispersed.
4. Scrambling: Scrambling is a key concept in understanding black holes and quantum information. It refers to the process by which information that enters a black hole becomes so thoroughly mixed that it is effectively lost to an outside observer. By using random graphs to model qubit interactions and entanglement, researchers can study how quickly and efficiently information scrambles inside a black hole.

This approach provides valuable insights into the behavior of quantum information in extreme environments and helps bridge the gap between quantum mechanics and general relativity.

Quantum Code

Now, let's examine the quantum code that models the spread of quantum information inside a black hole.

```
import numpy as np

import networkx as nx

import random

from qiskit import QuantumCircuit, Aer, execute

from qiskit.quantum_info import Statevector, partial_trace, entropy


def run_random_graphs_for_black_hole(n, p, subsystem_size):

    # Parameters

    #n: Number of qubits

    #p: Edge probability

    #subsystem_size: # Qubits to keep (rest are traced out)
```

```
# Generate random graph
G = nx.erdos_renyi_graph(n, p)

# Create quantum circuit
qc = QuantumCircuit(n)

# Initialize qubits in superposition (product state)
qc.h(range(n))

# Apply CNOTs, swaps, and cz gates for each edge in random order
edges = list(G.edges())
random.shuffle(edges)
for (control, target) in edges:
    qc.cx(control, target)
    qc.swap(control, target)
    qc.cz(control, target)

# Simulate and compute entanglement entropy
backend = Aer.get_backend('statevector_simulator')
statevector = execute(qc, backend).result().get_statevector()

# Trace out the complement of the subsystem
subsystem = list(range(subsystem_size, n))  # Qubits to trace out
rho = partial_trace(statevector, subsystem)
entropy_val = entropy(rho, base=2)

print(f"Entanglement entropy: {entropy_val:.4f}")
```

This code models the spread of quantum information inside a black hole using random graphs and quantum circuits. Here's a step-by-step explanation:

1. Imports: The code imports necessary libraries, including `numpy`, `networkx`, `random`, and `qiskit`.
2. Function Definition: The function `run_random_graphs_for_black_hole` takes three parameters:
    - `n`: Number of qubits.
    - `p`: Edge probability for the random graph.
    - subsystem_size: Number of qubits to keep (the rest are traced out).

3. Generate Random Graph: The code generates an Erdős-Rényi random graph G with n nodes (qubits) and edge probability p. This graph represents the interactions between qubits.
4. Create Quantum Circuit: A quantum circuit qc with n qubits is created.
5. Initialize Qubits: All qubits are initialized in a superposition state using Hadamard gates ($qc.h(range(n))$).
6. Apply Entangling Gates: The code applies a series of entangling gates (CNOT, SWAP, and CZ) to the qubits based on the edges of the random graph. The edges are shuffled to introduce randomness in the order of gate application.
7. Simulate Quantum Circuit: The quantum circuit is simulated using the statevector simulator backend from Qiskit. The resulting statevector represents the quantum state of the system.
8. Compute Entanglement Entropy: The code traces out the qubits that are not part of the subsystem (specified by subsystem_size). The partial trace operation reduces the statevector to a density matrix representing the subsystem. The entanglement entropy of this density matrix is then calculated, which quantifies the spread of quantum information.
9. Output: The entanglement entropy value is printed, providing a measure of how information has spread within the system.

In the context of a black hole, this code helps model how quantum information spreads and becomes entangled, analogous to the scrambling process inside a black hole. The entanglement entropy serves as a measure of this information spread.

The model of using random graphs to simulate the spread of quantum information inside a black hole has several significant implications for quantum information theory:

1. Understanding Scrambling: This model helps researchers understand the process of scrambling, where information that falls into a black hole becomes rapidly mixed and dispersed. Scrambling is a key concept in quantum information theory and black hole physics, and this model provides a way to study it quantitatively.
2. Entanglement and Information Spread: By modeling qubit interactions and calculating entanglement entropy, the model provides insights into how quantum information spreads and becomes entangled in complex systems. This is crucial for understanding the behavior of quantum systems and the nature of quantum entanglement.
3. Quantum Chaos: The use of random graphs introduces an element of chaos into the system, which is analogous to the chaotic environment inside a black hole. Studying quantum chaos is important for understanding the behavior of quantum systems in the presence of disorder and randomness.
4. Quantum Computing: The model has practical implications for quantum computing, as it provides a way to simulate and study the behavior of quantum circuits with a large number

of qubits. This can help in the development of more efficient quantum algorithms and error correction techniques.

5. Bridging Quantum Mechanics and General Relativity: By providing a framework to study the behavior of quantum information in extreme environments, this model helps bridge the gap between quantum mechanics and general relativity. This is a significant step towards a unified theory of quantum gravity.
6. Experimental Verification: The model can be used to design experiments that test the predictions of quantum information theory in the context of black holes. This can lead to new insights and discoveries in both theoretical and experimental physics.

Overall, this model provides a powerful tool for exploring the fundamental principles of quantum information theory and their implications for our understanding of the universe.

Let's revisit the concept of entanglement entropy.

Entanglement Entropy

Imagine you have two particles that are like best friends. When these particles become "entangled," it means that whatever happens to one particle instantly affects the other, no matter how far apart they are. This special connection is called "quantum entanglement."

Now, "entanglement entropy" is a way to measure how much information is shared between these two entangled particles. Think of it like this: if you have two friends who share a lot of secrets, the amount of secrets they share is like the entanglement entropy. The more secrets (or information) they share, the higher the entanglement entropy.

The maximum value of entanglement entropy depends on the number of particles or subsystems involved. In general, for a system with N particles, the maximum entanglement entropy is given by $log_2(N)$. This means that the more particles or subsystems you have, the higher the maximum entanglement entropy can be.

In simple terms, entanglement entropy tells us how strongly two particles are connected and how much information they share with each other. It's a fascinating concept that helps scientists understand the mysterious world of quantum mechanics!

Let's delve deeper into what these lines of code do:

```
# Trace out the complement of the subsystem
subsystem = list(range(subsystem_size, n))  # Qubits to trace out
rho = partial_trace(statevector, subsystem)
entropy_val = entropy(rho, base=2)
```

1. **Trace out the complement of the subsystem**: This means we are focusing on a specific part of the system (subsystem) and ignoring the rest. In the context of a black hole, this helps us understand how information spreads within a particular region while disregarding the rest of the system.

2. **subsystem = list(range(subsystem_size, n))**: Here, we are defining the subsystem by specifying which qubits to keep. The subsystem_size determines the number of qubits we are interested in, and n is the total number of qubits. The range(subsystem_size, n) creates a list of qubits that we will ignore (trace out).

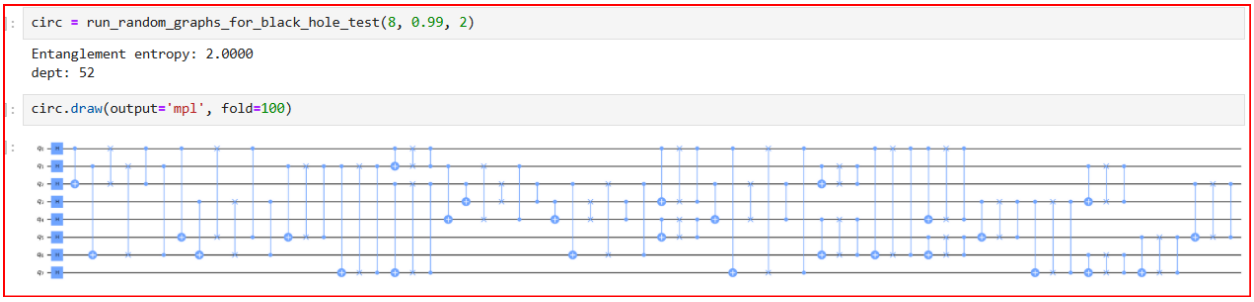
3. **rho = partial_trace(statevector, subsystem)**: This line performs a mathematical operation called "partial trace" on the statevector (which represents the quantum state of the system). By tracing out the qubits in the subsystem list, we obtain a reduced representation (density matrix) of the remaining qubits. This helps us focus on the part of the system we are interested in.

4. **entropy_val = entropy(rho, base=2)**: Here, we calculate the entanglement entropy of the reduced representation (density matrix) obtained from the partial trace. Entanglement entropy measures how much information is spread and entangled within the subsystem. In the context of a black hole, it quantifies the degree of information scrambling.

5. **print(f"Entanglement entropy: {entropy_val:.4f}")**: Finally, this line prints the calculated entanglement entropy value, giving us a numerical measure of how information has spread and become entangled within the subsystem.

These lines of code help us understand how quantum information spreads and becomes entangled inside a black hole by focusing on a specific part of the system and calculating the entanglement entropy. This provides valuable insights into the behavior of information in extreme environments.

To satisfy your curiosity, here is the corresponding quantum circuit. Note the significant depth of 52.



As previously mentioned, the depth of a quantum circuit refers to the number of layers of gates applied to the qubits. In the context of information spreading inside a black hole, the circuit depth has several important implications:

1. **Complexity of Entanglement**: A higher circuit depth means more layers of entangling gates are applied to the qubits. This increases the complexity of entanglement, leading to a more thorough mixing and spreading of quantum information. In a black hole, this is analogous to the rapid scrambling of information.

2. **Information Propagation**: As the depth increases, the quantum information propagates more extensively throughout the system. This means that information initially localized to

a few qubits becomes distributed across many qubits, simulating the chaotic environment inside a black hole.

3. **Measurement of Entanglement Entropy**: The entanglement entropy, which quantifies the spread of information, tends to increase with circuit depth. This provides a measure of how effectively information is being scrambled and dispersed within the system.

4. **Simulation of Black Hole Dynamics**: A deeper circuit can better simulate the dynamics of a black hole, where information is rapidly mixed and lost to an outside observer. This helps researchers understand the behavior of quantum information in extreme environments and contributes to the study of quantum gravity.

Overall, the depth of the quantum circuit plays a crucial role in modeling the spread of quantum information inside a black hole, providing valuable insights into the nature of entanglement and scrambling.

The circuit consistently produces elevated levels of entanglement entropy, suggesting enhanced scrambling, similar to black hole dynamics. Each run is conducted for a random graph realization.

```
[74]: for i in range(20):
          run_random_graphs_for_black_hole(8, 0.99, 2)

Entanglement entropy: 2.0000
Entanglement entropy: 1.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 1.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
Entanglement entropy: 2.0000
```

Summary:

Explanation

1. Random Graph: The Erdős-Rényi graph G defines qubit interactions. Each edge adds a CNOT, SWAP, CZ gates, simulating chaotic interactions in a black hole.

2. Quantum Circuit:
    - Qubits start in a superposition (no entanglement).
    - CNOT, SWAP, CZ gates (edges) entangle qubits, spreading information.
3. Entanglement Entropy:
    - After tracing out part of the system, the entropy measures how much information is shared between subsystems.
    - Higher entropy indicates greater scrambling, similar to black hole dynamics.

Interpretation

- Connectivity (p): Higher p increases entanglement entropy, mimicking faster information scrambling.
- Graph Variability: Run multiple trials to average over random graph realizations.
- Extensions: Use more qubits, deeper circuits, or different gates (e.g., CZ, SWAP) for richer behavior.

This approach links random graph theory to quantum information dynamics, offering a simplified model of black hole scrambling.

Benefits:

1. Random Graph Generation: The code uses the networkx library to generate random graphs based on the Erdős-Rényi model, which is useful for simulating various graph structures.
2. Quantum Circuit Creation: The code initializes qubits in a superposition state and applies a series of quantum gates (CNOT, swap, and CZ) based on the edges of the random graph. This approach helps in exploring the entanglement properties of different graph structures.
3. Entanglement Entropy Calculation: The code calculates the entanglement entropy by tracing out a subset of qubits, which is a valuable metric for understanding quantum information scrambling and black hole dynamics.
4. Modularity: The `run_random_graphs_for_black_hole` function is well-structured with clear parameters, making it easy to modify and extend for different experiments.

Potential Issues:

1. Gate Application Order: The code applies CNOT, swap, and CZ gates in a fixed order for each edge. This might not fully explore the entanglement properties of the graph. Randomizing the order of gate application could provide more diverse results.
2. Subsystem Size: The choice of `subsystem_size` is crucial for the entanglement entropy calculation. If the size is too small or too large, it might not provide meaningful insights. Experimenting with different sizes could be beneficial.

3. Performance: The code uses the statevector_simulator backend, which is suitable for small-scale simulations. However, for larger graphs and circuits, this approach might become computationally expensive. Using more efficient simulation techniques or hardware backends could improve performance.

4. Error Handling: The code does not include error handling mechanisms. Adding checks for valid input parameters and handling exceptions during execution would make the code more robust.

Overall, the code provides a solid foundation for exploring entanglement entropy in quantum circuits based on random graphs, but there are opportunities for optimization and enhancement.

Please feel free to download the notebook and further explore:

[randomGraphsAndblackHoles/RandomGraphsAndBlackHoles.ipynb at main · samlip-blip/randomGraphsAndblackHoles]

In the notebook, you'll find the modified code that randomizes the order of gate application, resulting in more diverse outcomes, including a higher frequency of 2's compared to the original version.

```
# Apply CNOTs, swaps, and cz gates for each edge in random order
    edges = list(G.edges())
    random.shuffle(edges)
    gate_list = ['cx','swap','cz']
    for (control, target) in edges:
        random.shuffle(gate_list)
        for gate in gate_list:
            apply_gate(qc, gate, control, target)
```

Can you experiment with different subsystem sizes?

The next chapter covers the spin glass model, its complexity, and the use of the Quantum Approximate Optimization Algorithm (QAOA) to determine the ground state of a spin glass model

using Qiskit. It details the parameters of the QAOA algorithm and how it optimizes the ground state configuration.

Spin Glass Model With A Randomly Generated Interaction Graph

In this chapter, we will introduce a Qiskit implementation designed to determine the ground state of a spin glass model with a randomly generated interaction graph. This implementation employs the Quantum Approximate Optimization Algorithm (QAOA) to address the combinatorial optimization problem represented by the spin glass Hamiltonian. Let's begin.

Imagine you have a box filled with tiny magnets, each one pointing in a different direction. These magnets represent atoms with their own magnetic fields. In a regular magnet, all the tiny magnets align in the same direction, creating a strong overall magnetic field. Now, in a spin glass, the tiny magnets are randomly arranged and "frozen" in place, pointing in different directions. This randomness and frustration (where some magnets want to align with their neighbors but can't) create a complex and disordered system. The spin glass model helps scientists understand how these disordered systems behave and how they can transition between different states. In simpler terms, think of a spin glass as a chaotic and frozen mess of tiny magnets, each trying to align but unable to do so perfectly.

In the context of spin glasses, a randomly generated interaction graph refers to a network where the connections (or interactions) between the tiny magnets (spins) are assigned randomly. Here's a simplified explanation:

Imagine you have a bunch of tiny magnets, each one represented as a node in a graph. The edges (connections) between these nodes represent the interactions between the magnets. In a randomly generated interaction graph, these edges are assigned randomly, meaning that each magnet can interact with any other magnet in a random manner.

This randomness introduces a lot of complexity and frustration into the system because some magnets will want to align with their neighbors, but the random interactions make it difficult for them to do so. This results in a disordered and chaotic system, which is characteristic of spin glasses.

The spin glass model with a randomly generated interaction graph is used to study how these complex and disordered systems behave, and it has applications in various fields such as physics, materials science, and even optimization problems in computer science.

Spin Glasses and Quantum Computers

Spin glasses are complex systems with disordered magnetic interactions, making them challenging to model using classical computers. However, quantum computers offer promising approaches to simulate and understand these systems.

One way to model spin glasses on a quantum computer is through the use of quantum annealing. Quantum annealing is a technique that leverages quantum fluctuations to find the ground state of

a system, which is the state of lowest energy. This method is particularly useful for solving optimization problems, including those found in spin glasses.

In quantum annealing, the system starts in a superposition of all possible states and gradually evolves towards the ground state. The quantum computer uses qubits to represent the spins in the spin glass, and quantum gates to simulate the interactions between these spins. By carefully controlling the evolution of the system, the quantum computer can find the configuration of spins that minimizes energy, effectively modeling the behavior of the spin glass.

Another approach is using the Quantum Approximate Optimization Algorithm (QAOA). QAOA is a hybrid quantum-classical algorithm that aims to find approximate solutions to combinatorial optimization problems. It involves applying a series of quantum gates to the qubits, followed by classical optimization steps to adjust the parameters of these gates. This iterative process helps in finding the optimal configuration of spins in the spin glass.

Both quantum annealing and QAOA are promising techniques for modeling spin glasses, and ongoing research continues to explore and refine these methods to better understand the complex behavior of these systems.

QAOA

Let's break it down in simple terms:

Imagine you have a puzzle where you need to find the best arrangement of pieces to get the highest score. This puzzle is very complicated, and there are many possible ways to arrange the pieces. Solving it by trying every possible arrangement would take a very long time.

QAOA helps by using the unique properties of quantum computers to find a good solution more efficiently. Here's how it works:

1. Quantum Superposition: Quantum computers can explore many possible solutions at once. Think of it as being able to look at multiple puzzle arrangements simultaneously.
2. Quantum Gates: These are like instructions that guide the quantum computer on how to manipulate the pieces of the puzzle. The quantum computer applies a series of these instructions to the puzzle pieces.
3. Classical Optimization: After applying the quantum instructions, the quantum computer measures the arrangement of the pieces. It then uses classical (regular) computer techniques to adjust the instructions and improve the arrangement.
4. Iteration: The process is repeated multiple times, with the quantum computer exploring different arrangements and the classical computer fine-tuning the instructions. Each iteration brings the solution closer to the best possible arrangement.

In essence, QAOA combines the strengths of quantum and classical computing to tackle complex optimization problems more effectively than classical methods alone. It's like having a super-smart assistant that can quickly sift through many possibilities and help you find a good solution to a tricky puzzle.

QAOA Parameters to tweak

In Qiskit, the Quantum Approximate Optimization Algorithm (QAOA) has several parameters that you can tweak to optimize its performance. Here are the key parameters:

1. p (Number of Layers): This parameter determines the depth of the QAOA circuit. It is an integer value that specifies the number of alternating operators applied in the algorithm.
2. β (Beta Angles): These are the angles associated with the mixing Hamiltonian. They are typically represented as a list of angles, one for each layer.
3. γ (Gamma Angles): These are the angles associated with the problem Hamiltonian. Similar to β, they are represented as a list of angles, one for each layer.
4. Initial Parameters: You can set initial values for β and γ to start the optimization process. These initial parameters can significantly impact the convergence of the algorithm.
5. Optimizer: Qiskit allows you to choose from various classical optimizers to optimize the β and γ angles. Common choices include COBYLA, SPSA, and NELDER-MEAD.
6. Measurement Shots: This parameter determines the number of times the quantum circuit is executed to obtain the probability distribution. More shots can lead to more accurate results but also increase computational cost.
7. Quantum Instance: This parameter specifies the backend on which the QAOA circuit will be executed. It can be a simulator or a real quantum device.

Hamiltonian for the spin glass

The Hamiltonian for the spin glass can be written in terms of Pauli-\(Z\) operators:

$$H = \sum_{(i,j)\in E} J_{ij} Z_i Z_j$$

where J_{ij} is the interaction strength between spins i and j and E is the set of the edges of respective randomly generated interaction graph.

*SIDE NOTE: Let's break down the concept of a **Hamiltonian** in simple terms.*

What is a Hamiltonian?

A Hamiltonian is a mathematical expression used in physics to describe the total energy of a system. Think of it as a formula that tells you how much energy a system has based on its current state.

Why is it Important?

The Hamiltonian is important because it helps us understand how a system behaves over time. By knowing the energy, we can predict how the system will evolve and what states it will prefer.

Components of a Hamiltonian

A Hamiltonian can include different types of energy, such as:

- ***Kinetic Energy**: The energy due to motion.*
- ***Potential Energy**: The energy due to position or configuration.*

Example: A Simple Pendulum

Imagine a simple pendulum swinging back and forth. The Hamiltonian for this pendulum would include:

- *The kinetic energy of the pendulum as it moves.*
- *The potential energy due to its height above the ground.*

Hamiltonian in Quantum Mechanics

In quantum mechanics, the Hamiltonian is used to describe the energy of particles, like electrons, in a system. It includes terms that represent interactions between particles, such as:

- ***Pauli Z Operators**: These represent the spin state of particles (up or down).*
- ***Coupling Strengths**: These represent the interactions between pairs of particles.*

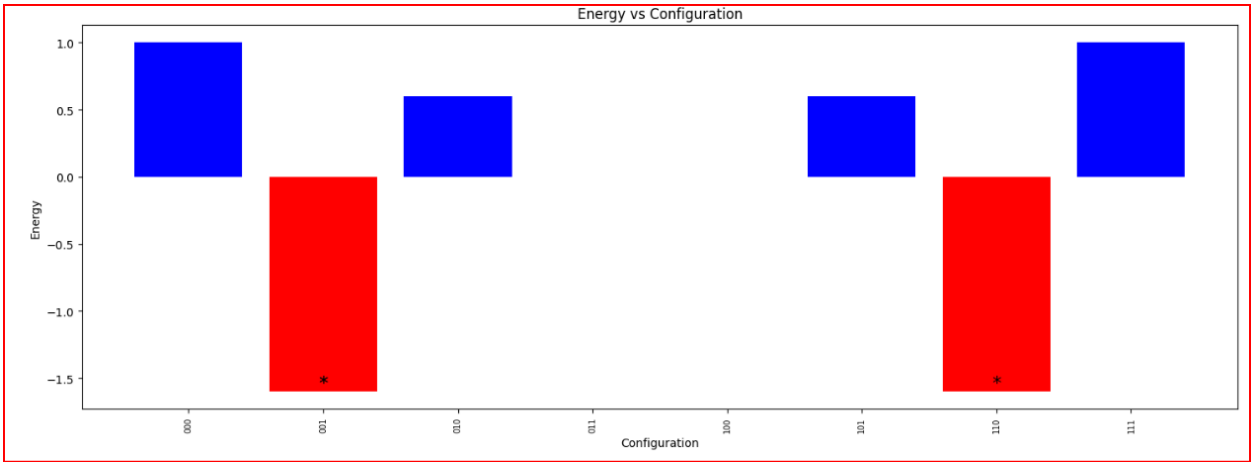
Example: Spin Glass

A spin glass is a type of magnetic system where the spins (magnetic moments) are randomly oriented and interact with each other. The Hamiltonian for a spin glass includes terms that represent these interactions. For example: $H = 0.5Z_1Z_2 - 0.3Z_2Z_3 + 0.8Z_1Z_3$. *Here,* Z_1, Z_2, *and* Z_3 *are Pauli Z operators representing the spin states, and the numbers (0.5, -0.3, 0.8) are the coupling strengths.*

Summary

In summary, a Hamiltonian is a formula that describes the total energy of a system. It helps us understand how the system behaves and evolves over time. In quantum mechanics, it includes terms that represent interactions between particles, allowing us to study complex systems like spin glasses.

Let's proceed with the Hamiltonian mentioned above for a preview. The lowest energy states are highlighted in red,



and both classical calculations and QAOA have correctly identified these states.

Optimal parameters: {ParameterVectorElement(t[0]): 5.82265002628899, ParameterVectorElement(t[1]): 3.3158158231975516, ParameterVectorElement(t[2]): -2.648525766316039, ParameterVectorElement(t[3]): -2.89095523942005, ParameterVectorElement(t[4]): -4.123850389928019, ParameterVectorElement(t[5]): 0.6025541865086546, ParameterVectorElement(t[6]): -0.5454488035910722, ParameterVectorElement(t[7]): -3.026686694140186}

Total prob: 1.0

Max prob: 0.4306640625

0.65625 110

QAOA ground state (bitstring): 110

energy: 1.0 config: 000

energy: -1.6 config: 001

energy: 0.6000000000000001 config: 010

energy: -5.551115123125783e-17 config: 011

energy: -5.551115123125783e-17 config: 100

energy: 0.6000000000000001 config: 101

energy: -1.6 config: 110

energy: 1.0 config: 111

Classical ground state (bitstring): 001

Classical ground state energy: -1.6

The energy distribution does appear to have some symmetry. Specifically, the energies for configurations 000 and 111 are the same, as are the energies for 001 and 110, and 010 and 101. This symmetry can occur in certain cases due to the specific structure of the interactions and the coupling strengths in the Hamiltonian.

In general, while spin glass systems are known for their complex and disordered energy landscapes, certain configurations and interaction patterns can lead to symmetrical energy distributions. This can happen when the interactions are balanced in such a way that **flipping all spins (changing 0 to 1 and vice versa) results in the same energy**.

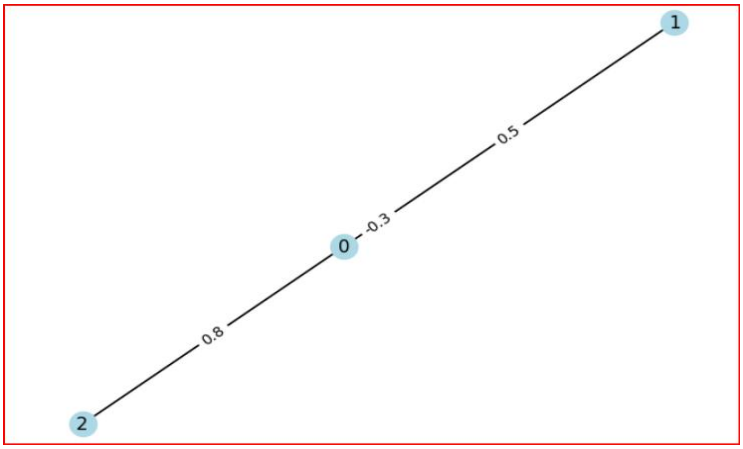
$$H = 0.5Z_1Z_2 - 0.3Z_2Z_3 + 0.8Z_1Z_3$$

has interactions that might lead to such symmetrical energy configurations. This is not always the case for all spin glass systems, but it can happen depending on the specific interactions and coupling strengths.

Quantum Code

Let's examine now the code that produced the energy distribution mentioned above, beginning with the graph generation. This code creates a graph with 3 nodes and edges with specified weights, arranges the nodes using a force-directed algorithm, labels the nodes and edges, and displays the graph with the coupling strengths indicated on the edges.

```
num_qubits = 3  # Small system for demonstration

graph = nx.Graph()

graph.add_nodes_from(range(num_qubits))

graph.add_edge(0, 1, weight=0.5)

graph.add_edge(1, 2, weight=-0.3)

graph.add_edge(0, 2, weight=0.8)


# Visualize the graph

pos = nx.spring_layout(graph)  # Position nodes using Fruchterman-Reingold force-directed algorithm

nx.draw(graph, pos, with_labels=True, node_color='lightblue')


# Draw edge labels (coupling strengths)

edge_labels = nx.get_edge_attributes(graph, 'weight')

nx.draw_networkx_edge_labels(graph, pos, edge_labels=edge_labels)


plt.show()
```



Next, the code defines a mathematical expression for the energy of a quantum system using Pauli operators and their coefficients, creates the Hamiltonian from these terms, and prints the resulting expression.

```
# Step 1: Define the Hamiltonian terms
hamiltonian_terms = [
   ('ZZI', 0.5),  # 0.5 * Z_1 * Z_2 * I_3
   ('IZZ', -0.3), # -0.3 * I_1 * Z_2 * Z_3
   ('ZIZ', 0.8)   # 0.8 * Z_1 * I_2 * Z_3
]

# Create the Hamiltonian
hamiltonian = PauliSumOp.from_list(hamiltonian_terms)

# Print the Hamiltonian
print("Hamiltonian:\n", hamiltonian)
```

Defining the Hamiltonian Terms

1. The variable hamiltonian_terms is a list of tuples. Each tuple represents a term in the Hamiltonian (a mathematical expression for the system's energy). The first element of each tuple is a string of Pauli operators ('ZZI', 'IZZ', 'ZIZ'), and the second element is a number (the coefficient or weight of the term).

- ('ZZI', 0.5): This term represents ($0.5Z_1Z_2I_3$), where (Z) is the Pauli Z operator and (I) is the identity operator.
- ('IZZ', -0.3): This term represents ($-0.3I_1Z_2Z_3$).
- ('ZIZ', 0.8): This term represents ($0.8Z_1I_2Z_3$).

Subscripts 1, 2, and 3 indicate the qubit on which the Z or I operators are applied.

Creating the Hamiltonian

2. The line `hamiltonian = PauliSumOp.from_list(hamiltonian_terms)` uses the PauliSumOp class to create the Hamiltonian from the list of terms. This combines all the terms into a single mathematical expression that represents the total energy of the system.

Printing the Hamiltonian

3. The line `print("Hamiltonian:\n", hamiltonian)` prints the Hamiltonian to the console, showing the combined expression.

Then, the code sets up and runs the QAOA algorithm to find the lowest energy state of a system, retrieves and prints the ground state configuration, and verifies the result using a classical brute-force calculation.

```
classical_config_list = []

classical_energy_list = []


# Step 2: Configure QAOA

optimizer = COBYLA(maxiter=100)

backend = Aer.get_backend('qasm_simulator')

quantum_instance = QuantumInstance(backend, shots=2048)

qaoa = QAOA(optimizer=optimizer, reps=4, quantum_instance=quantum_instance)  # reps = number of QAOA layers

backend = Aer.get_backend('qasm_simulator')



# Run QAOA

result = qaoa.compute_minimum_eigenvalue(hamiltonian)

print("\n\nresult:", result)

print("\n\nOptimal parameters:", result.optimal_parameters)


# Step 3: Get the ground state configuration from QAOA

counts = result.eigenstate

if counts is not None:
```

```
    tot_prob = np.sum([counts[key]*np.conj(counts[key]) for key in counts])
    max_prob = np.max([counts[key]*np.conj(counts[key]) for key in counts])
    print("Total prob:",tot_prob)
    print("Max prob:",max_prob)
    for key in counts:
      if counts[key]*np.conj(counts[key]) == max_prob:
        print(counts[key],key[::-1])
        ground_state=key[::-1]
    print("\nQAOA ground state (bitstring):", ground_state,"\n")
else:
    print("\nNo result from QAOA.\n")


# Step 4: Verify with classical brute-force calculation
classical_config, classical_energy = classical_ground_state(graph, num_qubits)
print("\nClassical ground state (bitstring):", classical_config)
print("\nClassical ground state energy:", classical_energy)
```

The code starts by creating two empty lists:

- `classical_config_list = []`: This list will store different configurations (arrangements) of the system.
- `classical_energy_list = []`: This list will store the energy values corresponding to each configuration.

Step 2: Configure QAOA

1. **Optimizer**: `optimizer = COBYLA(maxiter=100)` sets up an optimizer called COBYLA, which will try to find the best parameters for the quantum algorithm. It will run for a maximum of 100 iterations.
2. **Backend**: `backend = Aer.get_backend('qasm_simulator')` selects a quantum simulator called QASM, which will simulate the quantum computations.
3. **Quantum Instance**: `quantum_instance = QuantumInstance(backend, shots=2048)` sets up the quantum instance with the selected backend and specifies that each quantum circuit will be run 2048 times to get accurate results.
4. **QAOA Configuration**: `qaoa=QAOA(optimizer=optimizer,reps=4, quantum_instance=quantum_instance)` configures the Quantum Approximate Optimization Algorithm (QAOA) with the optimizer, 4 layers (reps), and the quantum instance.

Running QAOA

1. **Compute Minimum Eigenvalue**: `result = qaoa.compute_minimum_eigenvalue(hamiltonian)` runs the QAOA algorithm to find the minimum eigenvalue (lowest energy) of the Hamiltonian (a mathematical expression representing the system's energy).
2. **Print Results**: The code prints the result and the optimal parameters found by QAOA.

Step 3: Get the Ground State Configuration from QAOA

1. **Get Eigenstate**: `counts = result.eigenstate` retrieves the eigenstate (quantum state) from the QAOA result.
2. **Calculate Probabilities**: The code calculates the total and maximum probabilities of the eigenstate.
3. **Identify Ground State**: The code identifies the ground state configuration (bitstring) with the highest probability and prints it.

Step 4: Verify with Classical Brute-Force Calculation

1. **Classical Calculation**: `classical_config, classical_energy = classical_ground_state(graph, num_qubits)` runs a classical brute-force calculation to find the ground state configuration and its energy.
2. **Print Classical Results**: The code prints the ground state configuration and energy found by the classical calculation.

Kindly refer to the notebook for comprehensive details and the complete code. Let's demonstrate the approach with additional examples.

Example 1 (5 qubits)

Hamiltonian:

0.4639878836228102 * ZIZII

- 0.687962719115127 * ZIIZI

+ 0.2022300234864176 * IZIZI

+ 0.6648852816008435 * IIZZI

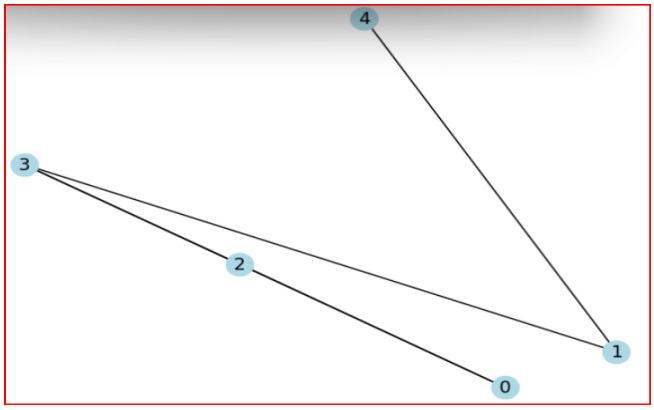
- 0.9588310114083951 * IZIIZ

QAOA ground state (bitstring): 01001

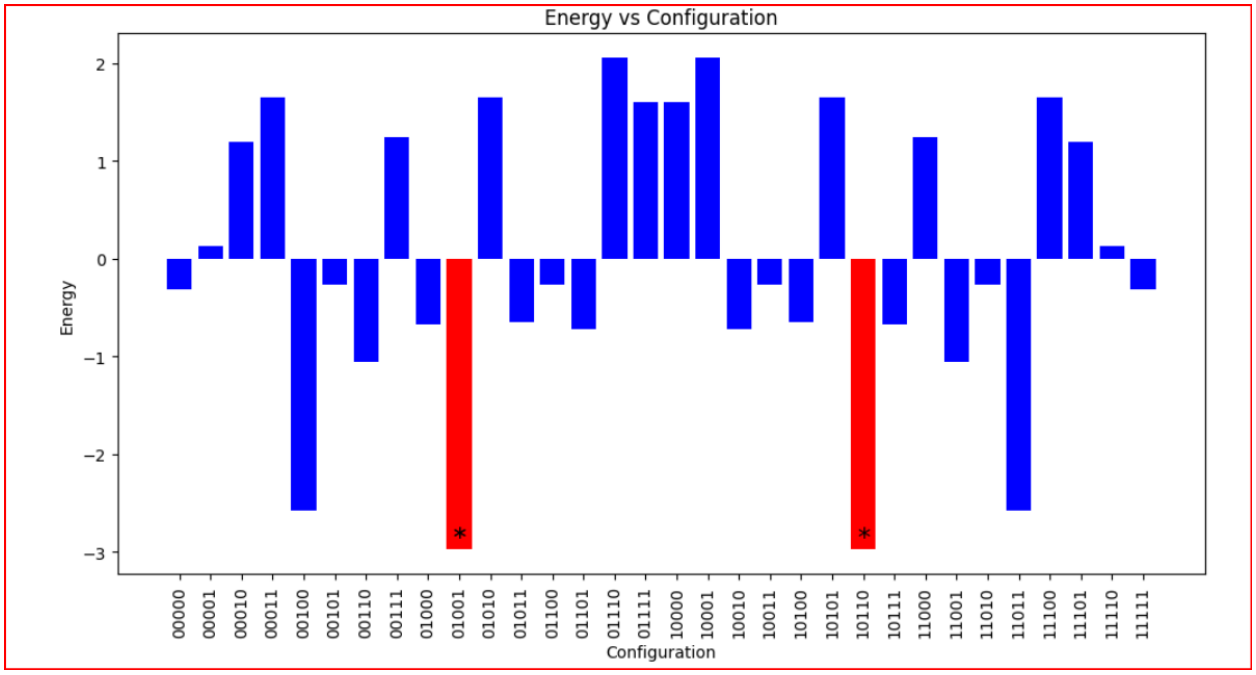
Classical ground state (bitstring): 01001

Classical ground state energy: -2.977896919233593

Respective random graph:



Lowest energy states highlighted in red



Example 2 (6 qubits)

Hamiltonian:

0.4639878836228102 * ZIZIII

+ 0.2022300234864176 * IZZIII

- 0.687962719115127 * ZIIZII

- 0.9588310114083951 * IZIZII

+ 0.6648852816008435 * IZIIZI

- 0.13610996271576847 * IIIZZI

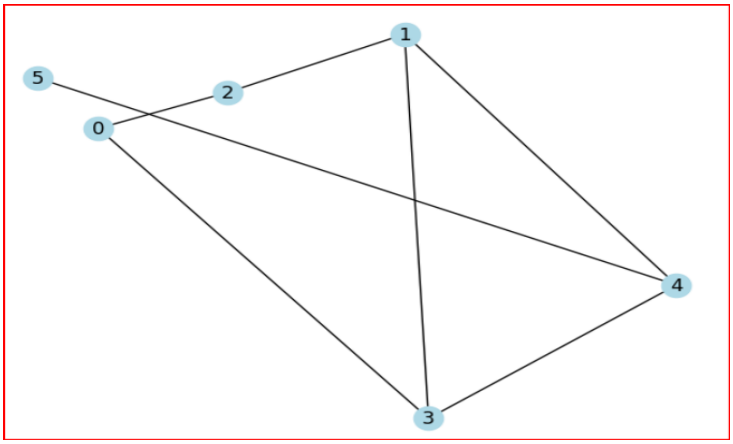
- 0.7210122786959163 * IIIIZZ

QAOA ground state (bitstring): 110100

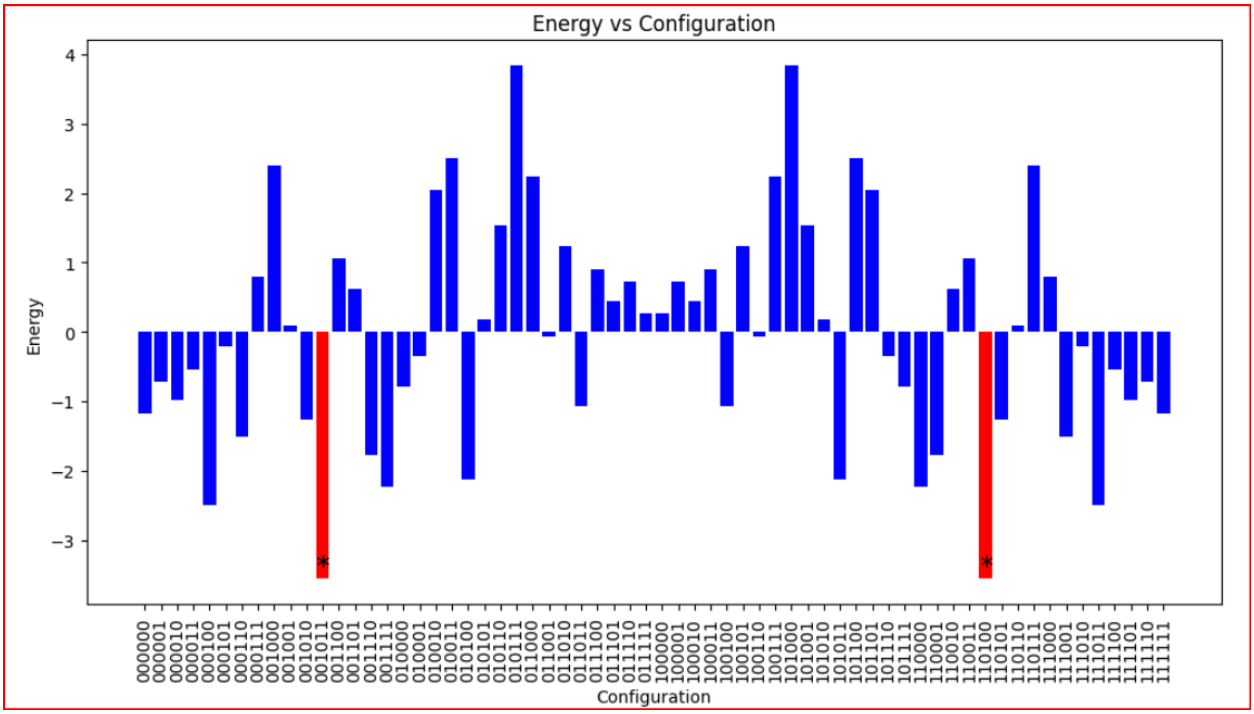
Classical ground state (bitstring): 001011

Classical ground state energy: -3.562799235213741

Random graph



Lowest energy states highlighted in red



Notice the interactions are balanced in such a way that flipping all spins (changing 0 to 1 and vice versa) results in the same energy. For example,

QAOA ground state (bitstring): 110100

Classical ground state (bitstring): 001011

have the same lowest energy.

Example 3 (7 qubits)

Hamiltonian:

0.4639878836228102 * ZIZIIII

- 0.9588310114083951 * IZZIIII

- 0.687962719115127 * ZIIZIII

+ 0.6648852816008435 * IZIZIII

- 0.13610996271576847 * IIZIZII

- 0.6006524356832805 * IIIIZZI

+ 0.2022300234864176 * ZIIIIIZ

- 0.7210122786959163 * IIZIIIZ

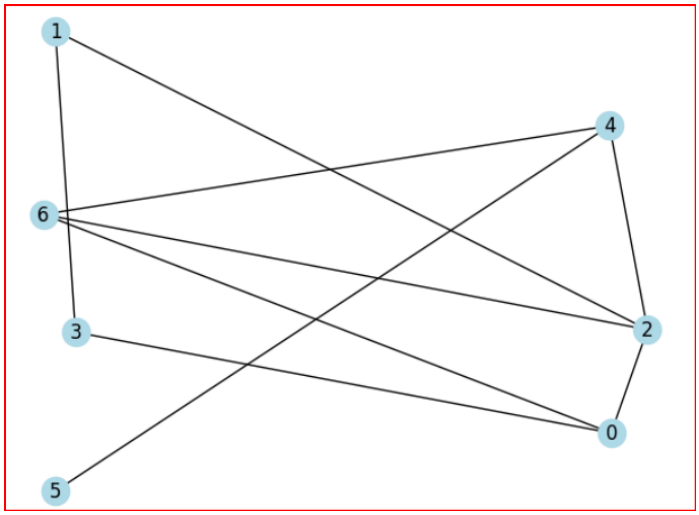
+ 0.18482913772408494 * IIIIZIZ

QAOA ground state (bitstring): 0111001

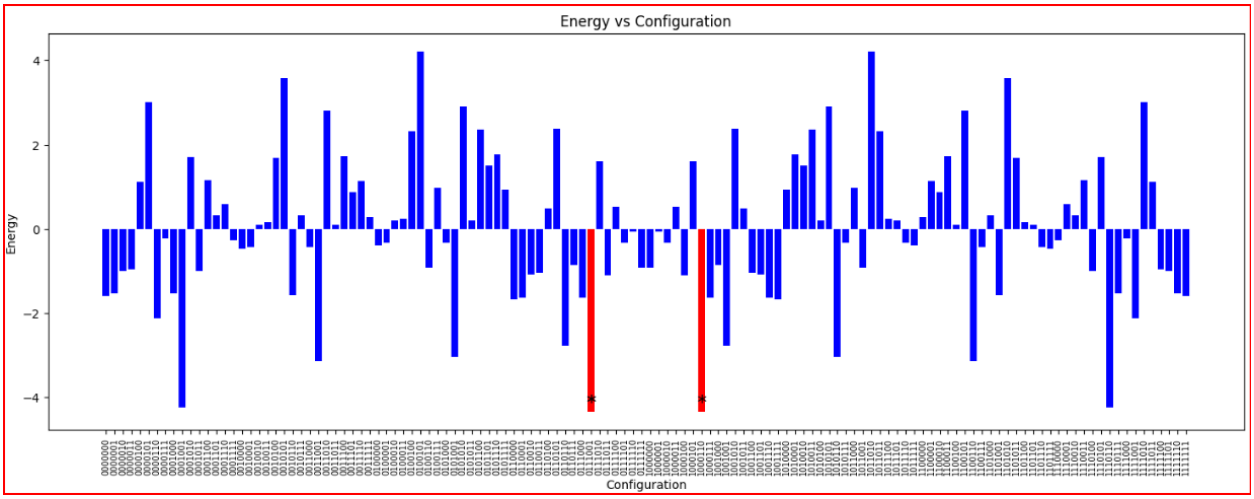
Classical ground state (bitstring): 0111001

Classical ground state energy: -4.348280808621107

Random Graph



Lowest energy states highlighted in red



Summary:

Here are the benefits and deficiencies of the provided code:

Benefits:

1. **Comprehensive Steps**: The code covers all necessary steps for solving the spin glass model using QAOA, from defining the interaction graph to verifying the results with classical brute-force calculations.
2. **Random Interaction Graph**: The use of a randomly generated interaction graph (create_random_graph(n)) ensures variability and robustness in testing the algorithm.
3. **Hamiltonian Conversion**: The conversion of the graph to an Ising Hamiltonian (graph_to_hamiltonian(graph)) is essential for representing the problem in a quantum framework.
4. **QAOA Configuration**: The code configures QAOA with a specified optimizer (COBYLA) and backend (qasm_simulator), allowing for flexibility in optimization and simulation.
5. **Result Analysis**: The code includes detailed analysis of the QAOA results, including optimal parameters and ground state configuration.
6. **Classical Verification**: The classical brute-force calculation (classical_ground_state(graph, num_qubits)) provides a benchmark for comparing the quantum results, ensuring accuracy.

Deficiencies:

1. **Error Handling**: The code lacks error handling mechanisms, which could lead to issues if any step fails or returns unexpected results.
2. **Efficiency**: The classical brute-force calculation may become inefficient for larger graphs, as it scales poorly with the number of qubits.

3. **Parameter Tuning**: The optimizer's maximum iterations (maxiter=100) and QAOA layers (reps) are hardcoded, which may not be optimal for all cases. Allowing these parameters to be adjustable would improve flexibility.
4. **Result Interpretation**: The interpretation of the QAOA results (counts) could be more robust, as the current method may not handle edge cases or unexpected results effectively.

Overall, the code provides a solid foundation for implementing QAOA to solve the spin glass model, but it could be improved with better error handling, efficiency considerations, and more flexible parameter tuning.

Please feel free to download the notebook and further explore:

SpinGlass/SpinGlass.ipynb at main · samlip-blip/SpinGlass

The last chapter explains the Deutsch-Jozsa algorithm, a quantum algorithm used to solve specific problems faster than classical algorithms. It also discusses Eulerian and half-odd-half-even graphs, their properties, and their relevance to the Deutsch-Jozsa algorithm.

Deutsch-Jozsa algorithm and Eulerian/half-Odd-half-Even Graphs

Our exciting journey is nearing its conclusion. In this chapter, we will demonstrate the practical application of the Deutch-Josza algorithm to determine whether a given graph is Eulerian or a half-odd-half-even graph. But first, let's delve into a few useful explanations.

What is a Eulerian Graph?

A **Eulerian graph** is one where you can start at any vertex and travel along the edges to visit every edge exactly once and return to the starting vertex. For a graph to be Eulerian, it must satisfy two conditions:

1. All vertices must have an even degree (an even number of edges connected to them).
2. The graph must be connected, meaning there is a path between any two vertices.

Example of a Eulerian Graph:

Imagine you have a graph with five vertices labeled A, B, C, D, and E, and edges connecting them as follows:

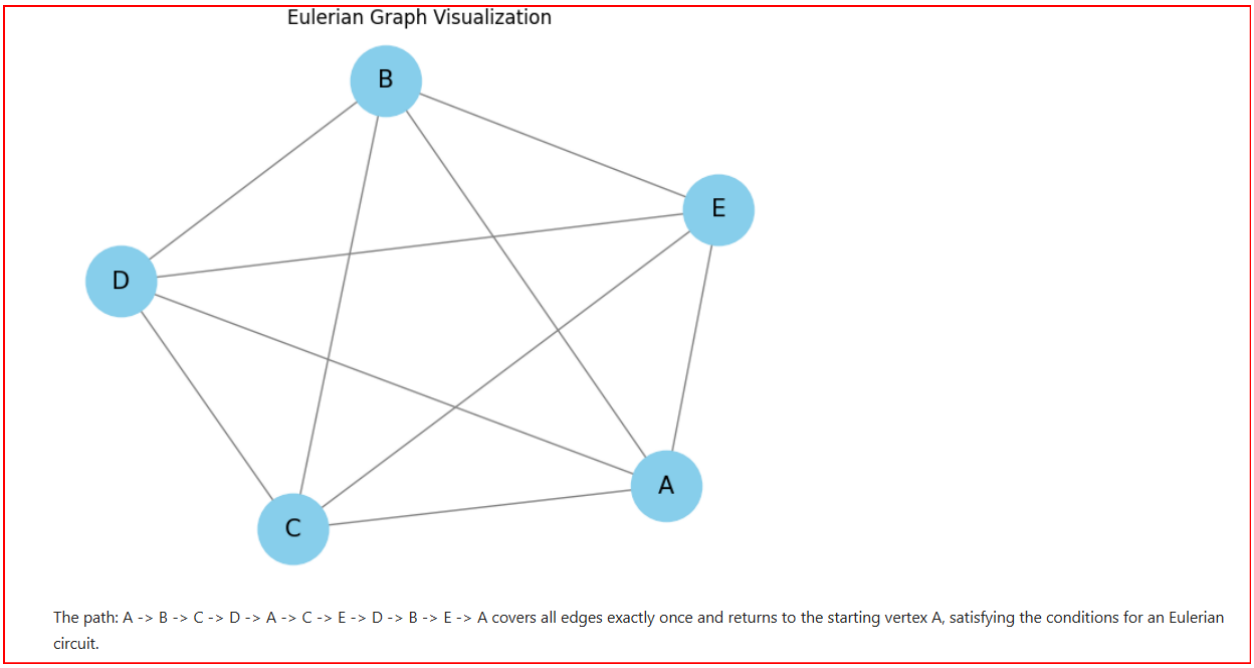
- A to B
- A to C
- A to D
- A to E
- B to C
- B to D
- B to E
- C to D
- C to E
- D to E

Degree of each vertex:

- Vertex A: 4 edges (A-B, A-C, A-D, A-E)
- Vertex B: 4 edges (B-A, B-C, B-D, B-E)
- Vertex C: 4 edges (C-A, C-B, C-D, C-E)
- Vertex D: 4 edges (D-A, D-B, D-C, D-E)

- Vertex E: 4 edges (E-A, E-B, E-C, E-D)

Since each vertex has an even degree (4 edges), the graph meets the first condition for being Eulerian. Additionally, the graph is connected, meaning there is a path between any two vertices. Therefore, this graph is Eulerian.



Eulerian graphs are useful in various applications, such as solving problems related to routing, network design, and even puzzles like the famous "Seven Bridges of Königsberg.

Practical Example: Network Routing

In network routing, Eulerian graphs can be used to design efficient routes for delivering goods or services. For instance, consider a scenario where a delivery truck needs to visit multiple locations in a city and return to the starting point without retracing any route. By modeling the city's road network as a graph, where intersections are vertices and roads are edges, an Eulerian graph can help determine the optimal route that covers all roads exactly once.

Practical Example: Circuit Design

In circuit design, Eulerian graphs can be used to create efficient layouts for electronic circuits. When designing a circuit, engineers need to connect various components with wires in a way that minimizes the total length of the wires and avoids overlapping connections. By representing the circuit as a graph, where components are vertices and connections are edges, an Eulerian graph can help design a layout that covers all connections exactly once, ensuring an efficient and organized circuit design.

What is half-odd half-even graph?

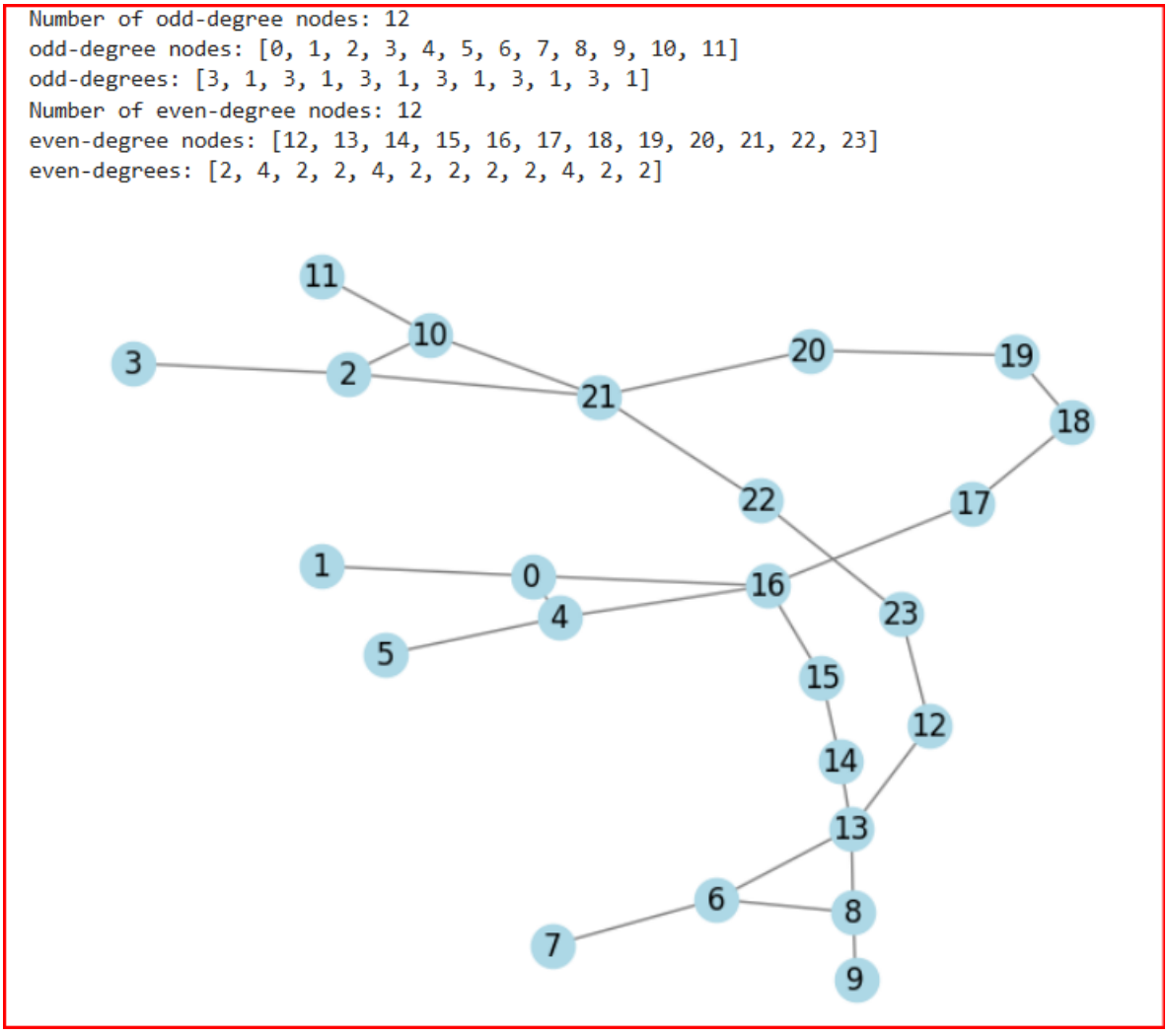
A graph where half the nodes have odd degrees and the other half have even degrees. Half-odd half-even graphs have practical applications in **quantum computing**, **cryptography**, and **network optimization**:

- **Quantum Computing**: These graphs help in designing quantum circuits that efficiently determine function properties, reducing computational complexity.
- **Cryptography**: They contribute to secure communication protocols by leveraging balanced functions in encryption schemes.
- **Network Optimization**: Used in parallel computing to distribute workloads efficiently, ensuring balanced processing.

Additionally, concepts related to **even and odd functions** play a role in **signal processing** and **Fourier analysis**, which are widely used in engineering and physics [1], [2], [3].

Example:

```
Number of odd-degree nodes: 12
odd-degree nodes: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
odd-degrees: [3, 1, 3, 1, 3, 1, 3, 1, 3, 1, 3, 1]
Number of even-degree nodes: 12
even-degree nodes: [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
even-degrees: [2, 4, 2, 2, 4, 2, 2, 2, 2, 4, 2, 2]
```



The Deutsch-Jozsa algorithm

The **Deutsch-Jozsa algorithm** is a quantum computing trick that helps us quickly figure out whether a function is **constant** (always gives the same answer) or **balanced** (gives two different answers equally).

Imagine you have a mysterious machine that takes numbers and spits out either a **0** or a **1**. You don't know how it works, but you do know one thing—it's either:

- **Constant** (always gives the same result no matter what number you put in), or
- **Balanced** (gives **0** for half the numbers and **1** for the other half).

If you were using a **regular computer**, you'd have to test multiple numbers to be sure. But with **quantum computing**, the Deutsch-Jozsa algorithm lets you figure it out in **just one step**!

How? Quantum computers use **superposition**, meaning they can test all possible numbers at the same time. Instead of checking each number one by one, the algorithm runs the function on all inputs simultaneously and instantly tells you whether it's constant or balanced.

It's one of the first examples of a quantum algorithm that is **exponentially faster** than classical methods. Pretty cool, right?

Imagine This:

You have a mystery box that takes in numbers and gives either a 0 or a 1. You don't know how it decides the output, but you do know:

- It's either constant, meaning it always gives the same answer (either always 0 or always 1).
- Or it's balanced, meaning it gives 0 for half the inputs and 1 for the other half.

Using a regular computer, you'd have to test at least half the inputs to be sure whether the box is balanced or constant. **But with quantum computing, the Deutsch-Jozsa algorithm lets you figure it out in just one step using something called superposition.**

How Does It Work?

1. Superposition Magic: Quantum computers can test all possible inputs at the same time instead of checking one by one.
2. Interference Trick: The quantum system processes the inputs and uses interference to eliminate the wrong possibilities.
3. Instant Answer: The result tells you immediately whether the function is balanced or constant without needing multiple tests.

Quantum Circuit Structure

Here's how the circuit works:

1. **Initialize Qubits:** The circuit begins with `n` qubits, where `n` is determined by the order of the graph being analyzed. The `diagonal` list encodes the parity for each vertex:

   In a **Eulerian graph**, all vertices have an even degree, so every parity value is `1`.

   In a **Half-Odd-Half-Even graph**, half the vertices have odd degrees while the others have even degrees, resulting in a mix of `1s` and `-1s`. The required number of qubits is determined by log2(len(diagonal)), ensuring the circuit can accommodate all possible inputs.

2. Apply Hadamard Gates: The Hadamard gate (H) is applied to all qubits, putting them into a superposition state. This allows the quantum computer to evaluate all possible inputs simultaneously.
3. Oracle Function (Black Box): The function you're testing is encoded into a quantum circuit (deutsch_jozsa_graph_check(n, diagonal)). This oracle applies a transformation based on whether the function is constant or balanced.
4. Measure the Output: The circuit is executed on a quantum simulator (Aer.get_backend('qasm_simulator')), and the results are analyzed:
   - If the output is all zeros, the function is constant (Eulerian graph).
   - If the output contains a mix of states, the function is balanced (half-odd half-even graph).

   Example Results

- Balanced Function (Half-Odd Half-Even Graph): The output {'10000': 1024} indicates a balanced function.
- Constant Function (Eulerian Graph): The output {'0000': 1024} confirms a constant function.
- Unknown Function: If multiple outputs appear, the function classification is unclear.

This approach allows quantum computers to determine function properties exponentially faster than classical methods.

The Deutsch-Jozsa algorithm is a fascinating example of how quantum computing can outperform classical methods. While it was originally designed as a theoretical demonstration rather than for practical applications, it has inspired techniques used in quantum machine learning, quantum cryptography, and quantum optimization.

Real-World Applications

1. Quantum Machine Learning: The principles behind the Deutsch-Jozsa algorithm—such as superposition and interference—are used in quantum machine learning to speed up classification tasks.
2. Quantum Cryptography: Some cryptographic protocols leverage similar quantum techniques to ensure secure communication.
3. Optimization Problems: The algorithm's ability to quickly determine function properties has inspired quantum approaches to solving complex optimization problems.

Although the algorithm itself is not widely used in industry, its concepts have influenced the development of more advanced quantum algorithms [4], [5], [6].

```
def deutsch_jozsa_graph_check(n, diagonal):
    """
    Constructs the Deutsch-Jozsa circuit for graph parity checking.
    Args:
        n (int): Number of qubits (log2 of vertices)
        diagonal (list): Diagonal elements defining the oracle
    Returns:
        QuantumCircuit: Configured quantum circuit
    """
    qc = QuantumCircuit(n)
    qc.h(range(n))  # Apply Hadamard to all qubits

    # Add the oracle (phase encoding of degree parity)
    qc.append(Diagonal(diagonal), range(n))

    qc.h(range(n))  # Apply Hadamard again
    qc.measure_all()
    return qc

# Example 1: Eulerian Graph (Constant function)
n = 2
constant_diagonal = [1,1,1,1]  # All degrees even
qc_constant = deutsch_jozsa_graph_check(n, constant_diagonal)
```

```
# Example 2: Half-Odd-half-Even Graph (Balanced function)
balanced_diagonal = [1,1,-1,-1]  # Degrees alternate even/odd
qc_balanced = deutsch_jozsa_graph_check(n, balanced_diagonal)

# Execute both circuits
simulator = Aer.get_backend('qasm_simulator')

# Run constant example
result_constant = execute(qc_constant, simulator, shots=1024).result()
print("Eulerian graph result (constant function):")
print(result_constant.get_counts())

# Run balanced example
result_balanced = execute(qc_balanced, simulator, shots=1024).result()
print("\nHalf-Odd-half-Even graph result (balanced function):")
print(result_balanced.get_counts())
```

Explanation:

1. **Eulerian Graph (Constant Function):**
    - All vertices have even degrees
    - Oracle applies no phase changes (diagonal = [1,1,1,1])
    - Measurement always results in 00, indicating a constant function
2. **Half-Odd-half-Even Graph (Balanced Function):**
    - Half the vertices have odd degrees
    - Oracle applies phase flips to states 10 and 11 (diagonal = [1,1,-1,-1])
    - Measurement results in 10, indicating a balanced function

Key Advantages:

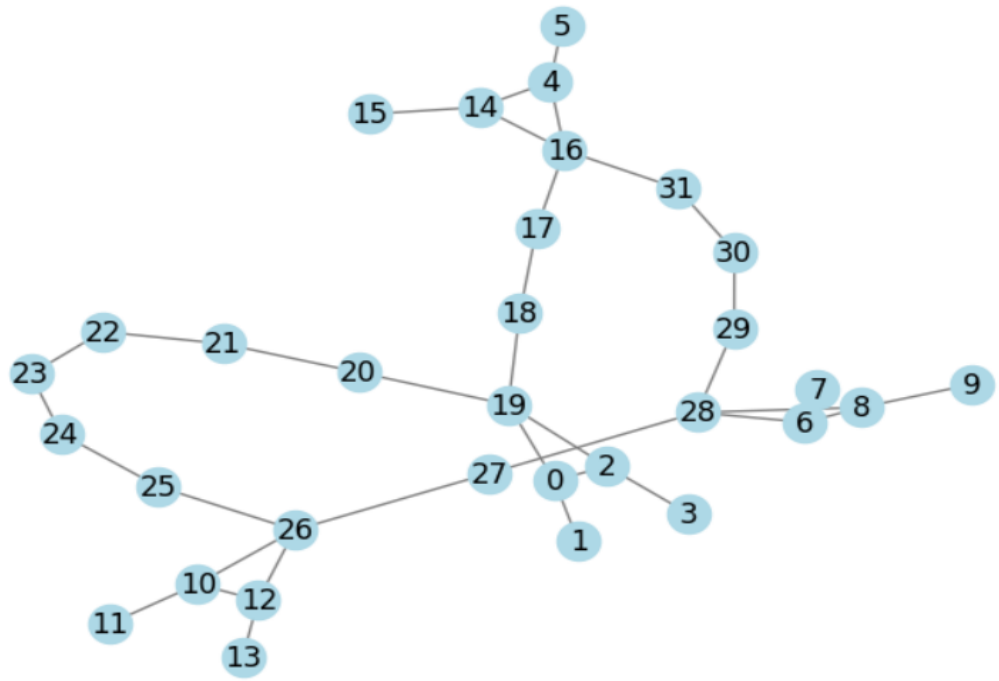
- Classical approach requires $\Theta(2^n)$ checks in worst case
- Quantum solution requires only 1 query
- Demonstrates exponential speedup for verifying graph properties

This implementation shows how quantum computing can efficiently solve graph theory problems that would be computationally expensive classically.

Testing Examples

Half-odd half-even graph

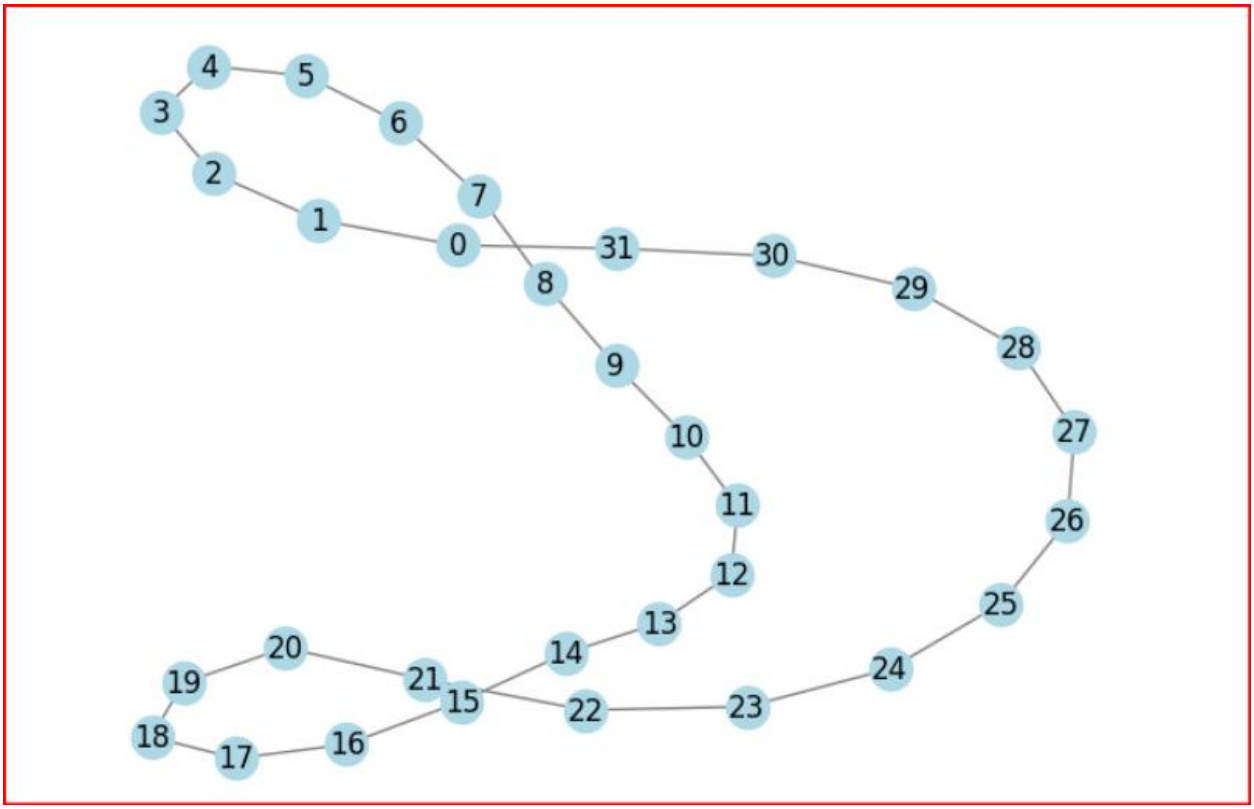
```
Number of odd-degree nodes: 16
odd-degree nodes: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
odd-degrees: [3, 1, 3, 1, 3, 1, 3, 1, 3, 1, 3, 1, 3, 1, 3, 1]
Number of even-degree nodes: 16
even-degree nodes: [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
even-degrees: [4, 2, 2, 4, 2, 2, 2, 2, 2, 2, 4, 2, 4, 2, 2, 2]
```



[118]:
```
import math
print(diagonal) # for the above graph
n = int(math.log2(len(diagonal)))
print("needed qubits:",n)
qc = deutsch_jozsa_graph_check(n, diagonal)

# Execute both circuits
simulator = Aer.get_backend('qasm_simulator')

# Run constant example
result = execute(qc, simulator, shots=1024).result()
print(result.get_counts())
k = [key for key in result.get_counts().keys()]
if len(result.get_counts().keys()) == 1 and k[0][0] == '1' and int(k[0][1:]) == 0:
    print("\nHalf-Odd-half-Even graph result (balanced function)")
elif len(result.get_counts().keys()) == 1 and int(k[0]) == 0:
    print("Eulerian graph result (constant function)")
else:
    print("Unknown function!")
```

```
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
needed qubits: 5
{'10000': 1024}

Half-Odd-half-Even graph result (balanced function)
```

Eulerian graph



[129]:
```
import math
print(diagonal) # for the above graph
n = int(math.log2(len(diagonal)))
print("needed qubits:",n)
qc = deutsch_jozsa_graph_check(n, diagonal)

# Execute both circuits
simulator = Aer.get_backend('qasm_simulator')

# Run constant example
result = execute(qc, simulator, shots=1024).result()
print(result.get_counts())
k = [key for key in result.get_counts().keys()]
if len(result.get_counts().keys()) == 1 and k[0][0] == '1' and int(k[0][1:]) == 0:
    print("\nHalf-Odd-half-Even graph result (balanced function)")
elif len(result.get_counts().keys()) == 1 and int(k[0]) == 0:
    print("Eulerian graph result (constant function)")
else:
    print("Unknown function!")
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
needed qubits: 5
{'00000': 1024}
Eulerian graph result (constant function)
```

Summary:

Let's break down the benefits and deficiencies of the code:

Benefits:

1. **Clear Structure**: The code is well-organized with clear comments and docstrings, making it easy to understand the purpose and functionality of each part.
2. **Modular Design**: The `deutsch_jozsa_graph_check` function is modular, allowing for easy reuse and adaptation for different graph parity checking scenarios.
3. **Quantum Circuit Construction**: The code effectively constructs the Deutsch-Jozsa quantum circuit, applying Hadamard gates and the oracle (Diagonal) for phase encoding.
4. **Execution and Simulation**: The code uses Qiskit's Aer backend to simulate the quantum circuits, providing a practical way to test and verify the results.
5. **Examples Provided**: The code includes examples for both Eulerian and half-odd-half-even graphs, demonstrating how to use the function for different types of graphs.

Deficiencies:

1. **Limited Scalability**: The code is designed for small-scale examples (e.g., n = 8). For larger graphs, the diagonal elements and the quantum circuit complexity would increase significantly, potentially leading to performance issues.
2. **Lack of Error Handling**: The code does not include error handling for invalid inputs or edge cases, such as non-square diagonal matrices or incorrect qubit numbers.

Overall, the code is a good starting point for implementing the Deutsch-Jozsa algorithm for graph parity checking, but it could be improved by addressing scalability, dynamic input generation, error handling, visualization, and documentation.

Please feel free to download the notebook and further explore:

DjAndEulerianGraphs/DjAndEulerianGraphs.ipynb at main · samlip-blip/DjAndEulerianGraphs

Could you suggest code modifications to accommodate a 5-node Eulerian graph?

Conclusion

In this book, we have explored the fascinating intersection of graph theory, quantum computing, and artificial intelligence through various applications and models. From the foundational concepts of bipartite graphs and shortest paths to the advanced applications involving the SYK model and black hole information scrambling, we have seen how these fields converge to solve complex problems and enhance our understanding of fundamental physics.

Graph theory provides a robust framework for analyzing relationships and interactions within networks, offering essential algorithms for optimizing data and designing efficient systems. Quantum computing, with its principles of superposition and entanglement, presents a revolutionary approach to performing calculations faster than classical computers, opening new avenues for solving intricate problems.

The DeepSeek AI chatbot exemplifies the power of artificial intelligence in interpreting complex data structures, generating code, and automating tasks. Its capabilities in enhancing natural language processing and supporting education highlight the potential of AI in various domains.

Through the exploration of models such as the SYK model, random graphs, and spin glass models, we have gained insights into quantum chaos, information scrambling, and the dynamics of black holes. The application of quantum algorithms, such as the Quantum Approximate Optimization Algorithm (QAOA) and the Deutsch-Jozsa algorithm, further demonstrates the potential of quantum computing in addressing challenging problems.

In conclusion, the integration of graph theory, quantum computing, and artificial intelligence offers a promising path towards solving complex problems and advancing our understanding of the universe. As these technologies continue to evolve, they hold the potential to revolutionize various fields and contribute to a sustainable, innovative future.

Bonus Chapter

This chapter is a bonus and not directly related to graphs. However, I included it to demonstrate the usefulness of Qiskit in simulating the core quantum mechanical behavior of the Type-I seesaw mechanism. It applies **quantum computing techniques** to study particle behavior, bridging quantum mechanics and fundamental physics! By modeling the mixing between light and heavy neutrino states, the code visualizes the oscillation between these states, showing how the light neutrino emerges from the mixing.

Feel free to skip this chapter dear reader if it doesn't interest you.

The Origin of Neutrino Mass: Theoretical Frameworks, Evidence, and Implications

Introduction

Neutrinos are elusive, electrically neutral elementary particles that were long assumed to be massless in the Standard Model (SM). However, late-20th-century discoveries of neutrino **oscillations** – the transformation of one neutrino flavor into another – provided compelling evidence that neutrinos possess a small but nonzero mass [1]. In the SM, neutrinos come only in left-handed form, with no right-handed partner, and thus no straightforward way to acquire a mass through the Higgs mechanism. The confirmation that neutrinos have mass therefore signaled new physics **beyond the Standard Model** [1]. This revelation has prompted a wide range of theoretical models for the origin of neutrino mass, each extending the SM in different ways [2]. In this report, we explore the leading theoretical frameworks – notably the **seesaw mechanism** and related models – that explain how neutrino mass might arise. We also review key experimental evidence supporting neutrino mass and discuss the far-reaching implications of neutrino mass for cosmology and particle physics.

Theoretical Frameworks for Neutrino Mass

Dirac vs Majorana Mass: In constructing neutrino mass terms, a crucial question is whether the neutrino is a **Dirac particle** (distinct from its antiparticle) or a **Majorana particle** (identical to its antiparticle). A Dirac mass term for neutrinos would require introducing right-handed neutrino fields (often called *sterile neutrinos* since they don't participate in weak interactions) and assigning a Yukawa coupling to the Higgs field. But to obtain a sub-eV neutrino mass with the Higgs vacuum expectation value, the Yukawa coupling would need to be on the order of 10^{-12} – an extraordinarily tiny number that many physicists find **unnatural**. Alternatively, neutrinos could be Majorana fermions, carrying no conserved lepton number. In that case, a neutrino mass term can violate lepton number by two units and does not require a right-handed neutrino in the usual sense, but it typically demands new physics to generate such a term. The Majorana possibility is especially appealing theoretically, as it allows mechanisms that naturally suppress neutrino masses relative to charged fermions. If neutrinos are Majorana particles, they would be their own antiparticles, an idea testable via neutrinoless double-beta decay (discussed later) [3].

Seesaw Mechanism: The most prominent framework for explaining tiny neutrino masses is the **seesaw mechanism**, so named because a heavy mass scale "leverages" the neutrino mass to very small values [3] [4]. The simplest **Type I seesaw** extends the SM by adding heavy right-handed neutrinos (one or more per generation) that are singlets under the electroweak force [4]. These heavy neutrinos have a large Majorana mass M (potentially approaching the grand unification scale, ~10^{14}–10^{15} GeV [4]) and also Yukawa couplings Y to the left-handed neutrinos and the Higgs field. Once the Higgs acquires a vacuum expectation value v, the neutrinos obtain a Dirac mass term $m_D = Yv$. In addition, the heavy right-handed neutrinos have their own large Majorana mass terms. The combined mass matrix for each neutrino generation takes the form (in a basis (ν_L, N_R)):

$$\begin{pmatrix} 0 & m_D \\ m_D & M \end{pmatrix},$$

which yields two eigenstates: one mostly corresponding to the heavy sterile neutrino (mass ~ M) and one mostly corresponding to the light neutrino. By diagonalizing this matrix, the light neutrino mass m_ν emerges approximately as $m_\nu \approx m_D^2/M$ [4]. This inverse relation is the essence of the seesaw: a very large M in the denominator **suppresses** the light neutrino's mass [3]. For example, if M is around 10^{14} GeV (a typical **GUT-scale** value) and m_D is of order 100 GeV (comparable to the Higgs scale), then m_ν comes out on the order of 0.1 eV – in the correct ballpark of observed neutrino masses. The heavy partner, being correspondingly massive, would have gone undetected so far (it would have existed in the hot early universe but decayed long ago) [2]. A key prediction of the seesaw mechanism is that light neutrinos are **Majorana fermions**. Lepton number is violated by the heavy Majorana mass term, and the light neutrino inherits this violation, being its own antiparticle. Indeed, the seesaw requires neutrinos to be Majorana particles for the mass suppression to work naturally [3]. If future experiments observe neutrinos annihilating themselves (for instance, via neutrinoless double-beta decay), it would strongly support the seesaw paradigm and the Majorana nature of neutrinos [3].

Type II and III Seesaws: Variants of the seesaw mechanism invoke different new particles. The **Type II seesaw** introduces a heavy Higgs **triplet scalar** (often denoted Δ) that carries two units of hypercharge and forms a triplet under SU(2)_L [2]. This Δ can directly couple to two left-handed lepton doublets at the Lagrangian level, effectively generating a Majorana mass term for neutrinos when Δ acquires a small vacuum expectation value. In Type II seesaw, the small neutrino mass arises because the induced triplet VEV is tiny compared to the SM Higgs VEV. The triplet's mass M_Δ is typically very large, and the induced VEV v_Δ is suppressed (often by the ratio of the electroweak scale squared to M_Δ) [2]. As a result, the neutrino mass is proportional to v_Δ and inversely related to M_Δ, again realizing a seesaw-like inverse proportionality [2]. Type II seesaw thus offers an alternate path: instead of heavy sterile neutrinos, a heavy scalar boson gives neutrinos their mass. The **Type III seesaw** is conceptually similar to Type I, but the heavy new fermions are in an **SU(2)_L triplet** representation (with zero hypercharge) rather than being singlets. These triplet fermions mix with ordinary neutrinos and, like in Type I, produce one heavy and one light state per generation. The light neutrino masses are suppressed by the heavy triplet mass in the same way as the Type I case. All three seesaw types share the common theme of a **heavy state** (whether a fermion or scalar) that "pulls down" the mass of the neutrino to small values when diagonalizing the mass matrix [2] [3].

Radiative Mass Generation Models: Another class of models forbids neutrino mass at tree-level (for example, by a symmetry) and generates it through quantum **loop corrections**. In these **radiative models**, new particles (additional scalars or fermions) are introduced such that neutrino masses arise only as higher-order effects, naturally small due to loop suppression. One famous example is the **Zee model**, which adds a charged scalar and a second Higgs doublet. In the Zee model, there is no direct mass term for neutrinos; instead, a one-loop diagram involving the new scalars produces a nonzero neutrino mass (violating lepton number softly via the scalar sector). Variants like the **Zee–Babu model** generate neutrino masses at the two-loop level by introducing doubly-charged scalar particles. The common feature is that each power of the coupling and loop factor (often $\frac{1}{16\pi^2}$ per loop) contributes to making the neutrino mass much smaller than the electroweak scale, even if the new particles are not astronomically heavy. Radiative models can operate at relatively low new-physics scales (e.g. electroweak to TeV scale), and many lead to distinctive signatures such as lepton-flavor-violating decays (due to the extended Higgs sector).

Inverse and Other Seesaw Variants: While the classical seesaw calls for very heavy new particles (potentially out of experimental reach), **inverse seesaw** models achieve small neutrino masses with new physics at lower scales, sometimes even ~TeV. The inverse seesaw introduces not only heavy right-handed neutrinos but also additional very light sterile neutrinos, along with a small Majorana mass term μ for these new light states. In this scenario, the smallness of neutrino mass is tied to the smallness of μ (which might be on the order of keV or less), rather than an extremely large M. By tuning μ to be tiny, the mass of the active neutrinos can be kept very small even if the heavy neutrinos have masses in the GeV–TeV range. This makes the model potentially **testable** in collider experiments: the heavy neutrinos could be produced at high-energy colliders and lead to observable lepton-number-violating signals. Other frameworks include models with spontaneous lepton number violation and a light Nambu–Goldstone boson (the **Majoron**), extra-dimensional models where neutrino Yukawa couplings are small due to geometry, and **low-scale leptogenesis** models that tie the generation of neutrino mass to the generation of the cosmic matter–antimatter asymmetry [2]. Each model comes with its own set of parameters and predictions, but all must in the end account for the observed pattern of neutrino masses and mixings.

Summary of Key Models: The table below summarizes some prominent neutrino mass models, highlighting the new particles involved and how they generate a small neutrino mass.

Model/Mechanism	New Particles/Features	Neutrino Mass Generation
Dirac (SM Extension)	Right-handed neutrinos (SM singlets) with tiny Yukawa couplings	Higgs gives Dirac mass like other fermions, but Yukawa Y_ν must be extremely small (~10^(-12)) to yield $m_\nu \sim$ eV. (Lepton number conserved; neutrinos distinct from antineutrinos.)
Type I Seesaw	Heavy right-handed Majorana neutrinos (sterile)	Heavy N_R fields with Majorana mass M mix with left-handed ν_L. Light mass $m_\nu \approx m_D^2/M$ is suppressed by large M [3]. Predicts Majorana neutrinos and lepton-number violation.
Type II Seesaw	Higgs triplet scalar Δ (triplet under SU(2)_L)	Triplet scalar acquires a small induced VEV, generating a direct $\nu\nu$ Majorana mass term. m_ν is proportional to this tiny VEV (and inversely to the heavy scalar mass) [2]. Often arises in Left-Right symmetric models.
Type III Seesaw	Heavy fermion triplet (SU(2)_L triplet like a heavy lepton)	Triplet fermions mix with neutrinos analogously to Type I. Light m_ν suppressed by heavy triplet mass (seesaw effect). Majorana nature with lepton-number violation similar to Type I.
Radiative Models	Extra scalars/fermions (e.g. charged scalars in loops)	No tree-level ν mass (forbidden by symmetry); loops induce m_ν. E.g. Zee model (1-loop) or Zee-Babu (2-loop) generate small neutrino masses with relatively light new particles due to loop suppression. Lepton number is typically violated in the loops.
Inverse Seesaw	Extra sterile neutrinos + small μ mass term	Adds light sterile ν_S and uses a tiny Majorana mass μ for ν_S to suppress m_ν. Active ν masses $\propto \mu$ (instead of $1/M$), allowing heavy neutrinos at TeV scale with testable signatures. Lepton number is broken by $\mu \ll M$.

Each of these frameworks provides a mechanism to explain why neutrino masses are so incredibly small compared to the masses of charged leptons and quarks. Notably, many scenarios (seesaws and radiative models) predict that neutrinos are Majorana particles and thus anticipate **lepton number violation** in certain processes. In contrast, a pure Dirac-mass scenario preserves lepton number but leaves the question of **why** the neutrino Yukawa couplings are so tiny. Ongoing and future experiments aim to distinguish these possibilities by probing for heavy neutrinos, lepton-number-violating signals, and other hallmarks of the various models.

Experimental Evidence for Neutrino Mass

Direct and indirect experiments have firmly established that neutrinos are massive (albeit very light) particles. The evidence comes from multiple avenues:

- **Neutrino Oscillations (Indirect Mass Evidence):** The phenomenon of neutrino oscillation – wherein a neutrino of one flavor (electron, muon, or tau) can transform into another flavor as it propagates – is possible only if neutrinos have mass and the different flavor states are quantum mixtures of different mass states. The first compelling evidence came in 1998 from the Super-Kamiokande detector's study of atmospheric neutrinos, which showed a deficit of upward-going muon neutrinos consistent with $\nu_\mu \to \nu_\tau$ oscillations [1]. A few years later, the Sudbury Neutrino Observatory (SNO) in Canada and Super-Kamiokande in Japan together demonstrated that the long-standing **solar neutrino problem** was also solved by oscillations – electron neutrinos produced in the Sun were oscillating into muon and tau neutrinos, which escaped detection in experiments sensitive only to electron flavor [1]. These discoveries, awarded the 2015 Nobel Prize in Physics, prove that neutrinos have finite mass differences and mixings [1]. Neutrino oscillation experiments have since measured the oscillation parameters with increasing precision: we know two distinct mass-squared differences (approximately 7.4×10^{-5} eV^2 and 2.5×10^{-3} eV^2) and three mixing angles of the neutrino mixing matrix (the PMNS matrix). However, oscillations are only sensitive to differences in mass, not the absolute scale. They tell us that at least two of the neutrino mass eigenstates are non-zero in mass, and that the heaviest neutrino is at least about 0.05 eV [5]. The fact that neutrino oscillations occur is irrefutable evidence that new physics (beyond the minimal SM) is at play [1], since the SM predicted neutrinos to be massless.

- **Direct Mass Measurements (Beta Decay Kinematics):** To pin down the absolute neutrino mass scale, physicists study the kinematics of weak decays, most sensitively **tritium beta decay**. In a tritium decay ($^3\text{H} \to {}^3\text{He} + e^- + \bar{\nu}_e$), the neutrino mass manifests as a minute distortion near the endpoint of the electron energy spectrum. The Karlsruhe Tritium Neutrino experiment (KATRIN) is the state-of-the-art in this method. In 2019, KATRIN set an upper limit of about 1.1 eV on the effective electron-antineutrino mass, improving on prior searches. As of 2022, KATRIN pushed the limit down to **0.8 eV**, and most recently (2023/2024 data) it has halved the upper bound to 0.45 **eV** at 90% confidence [6]. This new sub-eV sensitivity, achieved with 1-tonne of gaseous tritium and high-precision spectrometry, marks the first direct model-independent probe into the sub-eV neutrino mass range [6]. To put it in perspective, if $m_\nu \approx 0.45$ eV, a neutrino would be more than a million times lighter than an electron's 511 keV mass [6]. So far, no nonzero mass has been observed in the spectral shape – only upper limits. KATRIN's continuing data-taking aims to reach a sensitivity of around 0.2 eV in the coming years [6]. These direct measurements confirm that neutrino masses are extremely small in absolute terms (sub-eV), consistent with the oscillation findings.

- **Neutrinoless Double Beta Decay (Majorana Mass Evidence):** If neutrinos are Majorana particles, they can mediate a rare nuclear decay process known as **neutrinoless double beta decay** ($0\nu\beta\beta$). In a normal double beta decay (observed in certain isotopes like ^{136}Xe, ^{76}Ge, or ^{130}Te), two neutrons in a nucleus convert into two protons, emitting two electrons and two antineutrinos ($2\nu\beta\beta$). But if the neutrino has a Majorana mass, the antineutrinos emitted could effectively annihilate each other (as the neutrino is its own antiparticle), resulting in no neutrinos emerging – a lepton-number-violating process. Detecting $0\nu\beta\beta$ would be a clear sign that neutrinos are Majorana fermions and would provide a measure of the effective Majorana mass of the electron neutrino. Experiments like KamLAND-Zen, GERDA/LEGEND, CUORE, and EXO are searching for the telltale signature: a peak in the summed electron energy spectrum at the endpoint (the Q-value) of the decay. To date, no confirmed observation of $0\nu\beta\beta$ has been made. The **KamLAND-Zen** experiment, for example, has reported no signal in over a **1-ton-year** exposure of Xe-136, setting a lower limit on the half-life of this process on the order of 10^{26} years [[5]. This translates to an **upper limit on the effective Majorana neutrino mass of about** < 100 **meV (0.1 eV)**, depending on nuclear matrix element uncertainties [5]. Intriguingly, this sensitivity is approaching the range suggested by the inverted neutrino mass ordering (where the lightest neutrino mass could be tens of meV). We know from oscillation data that at least one neutrino mass eigenstate is around 50 meV or heavier [5], so the current generation of $0\nu\beta\beta$ experiments is beginning to probe this regime. If a positive signal is seen in the next decade, it would not only reveal the Majorana nature of neutrinos but also pin down the absolute mass scale. On the other hand, if no signal is seen even down to the few-meV level, that would imply neutrinos follow a normal mass hierarchy with the lightest neutrino extremely light, or that other mechanisms (like cancellations in the Majorana phases) make the $0\nu\beta\beta$ rate vanishingly small. Upcoming experiments plan to reach sensitivities corresponding to effective masses of 10–20 meV, which would cover the entire inverted hierarchy region and significantly probe the normal hierarchy as well [5].

Together, the oscillation and direct experiments paint a consistent picture: neutrinos have nonzero masses, likely in the tens of meV range for the heavier ones, and their mass generation involves new physics beyond the SM. The experimental focus now is to refine these measurements – to determine the **neutrino mass ordering** (whether the spectrum is normal or inverted), measure any possible **CP-violating phases** in the neutrino mixing (which could have profound implications, as discussed below), and to detect or further constrain the Majorana nature of neutrinos through $0\nu\beta\beta$ searches.

Implications for Cosmology

Neutrinos are not only important in particle physics but also play a significant role in cosmology. Being extremely numerous (second only to photons in abundance), relic neutrinos form a **cosmic neutrino background** (CνB) that decoupled from matter about one second after the Big Bang. If neutrinos were exactly massless, they would forever remain ultra-relativistic radiation. But with even small masses (meV to eV scale), cosmological neutrinos eventually become non-relativistic and contribute to the matter density of the Universe. This transition has subtle but measurable effects on cosmic evolution.

Early Universe and Structure Formation: In the early Universe, neutrinos were hot, fast-moving particles. Their high velocities meant they could escape out of small-scale density perturbations — a behavior known as **free streaming**. Unlike cold dark matter which clumps and seeds galaxy formation, these light neutrinos resisted clustering on small scales. In fact, an abundance of massive neutrinos tends to **smooth out** density fluctuations below a characteristic free-streaming length, inhibiting the formation of structure on those scales [7]. This effect has been detected indirectly: precision measurements of the Cosmic Microwave Background (CMB) anisotropies by the Planck satellite, combined with large-scale structure surveys, are sensitive to the presence of free-streaming relic neutrinos. The data show consistency with three active neutrinos with small masses. If the neutrino masses were larger, we would see greater suppression of small-scale power in the matter distribution than is observed. By comparing observations with theoretical models, cosmologists have placed stringent limits on the sum of the neutrino masses $\sum m_\nu$. The latest analyses (post-Planck 2018) constrain the total neutrino mass to $\sum m_\nu \lesssim 0.12$**–0.17 eV** (95% confidence), with slight variations depending on assumptions about the neutrino mass hierarchy and data sets used. In other words, neutrinos collectively contribute at most on the order of a fraction of an eV in mass per particle. This upper bound is remarkably close to the lower bound implied by oscillation experiments ($\sum m_\nu \approx 0.06$ eV minimum for normal ordering), indicating that we may be honing in on the true neutrino masses. The fact that cosmology and laboratory experiments like KATRIN are now exploring the same sub-eV range is an exciting convergence [6].

With these masses, neutrinos make up only a small fraction of the Universe's matter. For instance, if $\sum m_\nu \approx 0.1$ eV, the neutrino energy density today is on the order of $\Omega_\nu h^2 \sim 0.001$, which is less than about 1% of the total matter density (far less than dark matter or baryons). In the Universe's first few seconds, neutrinos contributed to the radiation energy density (affecting quantities like the effective number of neutrino species, N_{eff}, relevant for Big Bang Nucleosynthesis and CMB). But once the Universe cooled below temperatures of roughly the neutrino mass scale (around $T \sim m_\nu \sim$ tens of meV, which occurred long after the CMB formation, during the matter-dominated era), neutrinos started behaving like matter. The delayed transition from radiation-like behavior to matter-like behavior alters the timing of matter-radiation equality and leaves an imprint on the expansion history and large-scale structure. Upcoming cosmological surveys, by improving sensitivity to the subtle effects of neutrino mass (for example, via gravitational lensing of the CMB or galaxy clustering statistics), might eventually detect the signature of the minimum mass sum of 0.06 eV, providing an independent measurement of neutrino mass in the cosmological context.

Dark Matter and Neutrinos: Given their low mass and high thermal velocities, neutrinos are a form of **hot dark matter**. They stream out of small potential wells and cannot explain the existence of the small-scale structures we see (galaxies, clusters). A Universe where most dark matter was hot (e.g. neutrinos of a few eV mass) would produce a top-down formation scenario (large structures form first), in conflict with observations. The concordance ΛCDM model therefore assumes dark matter is cold (non-relativistic from early times). The role of neutrinos is then a subdominant one that slightly erases small structures. The small allowed neutrino masses today mean neutrinos contribute only a minor portion of the dark matter. For example, in a scenario with $\sum m_\nu = 0.1$ eV, neutrinos would make up well under 1% of the total dark matter mass in the Universe. This also implies that neutrinos cannot solve the **dark matter problem** – some other non-baryonic cold particles (WIMPs, axions, etc.) must compose the bulk of dark matter. Intriguingly, cosmological data in combination with particle limits have all but ruled out neutrinos being a major component of dark matter. In the 1980s, when only upper limits $m_\nu < 30$ eV were known, one could imagine neutrinos making up a fair fraction of dark matter; today we know that is not the case [6].

Baryon Asymmetry and Leptogenesis: One of the deep puzzles in cosmology is why the Universe contains much more matter than antimatter. An elegant theory called **leptogenesis** ties this asymmetry to neutrino mass generation. In a typical leptogenesis scenario (often realized in Type I seesaw models), the heavy right-handed neutrinos (the same ones that generate tiny neutrino masses) would have been produced in the hot early universe. As they decayed (in processes violating lepton number and CP symmetry), they could create a slight surplus of leptons over antileptons. This lepton asymmetry can then be converted to a baryon asymmetry via sphaleron processes in the SM (which violate $B+L$ but conserve $B-L$). The end result is an explanation for the cosmic matter–antimatter imbalance, rooted in the existence of heavy neutrinos and their Majorana masses. Remarkably, if neutrino masses are very small because of a seesaw at a very high scale (close to GUT scale), the out-of-equilibrium decays of those heavy neutrinos can naturally produce the observed baryon asymmetry of the Universe. Thus, the origin of neutrino mass might be directly linked to the origin of **baryonic matter** itself. Even if the heavy neutrinos are not accessible experimentally, this idea provides a strong motivation for why neutrinos should be Majorana particles. On the flip side, CP violation in the neutrino **oscillation** sector (the mixing matrix) could also contribute to leptogenesis. If the phases in the PMNS matrix are nonzero, leptonic processes could violate CP in the early universe. While the low-energy CP violation (e.g. as observable in long-baseline oscillation experiments) is not directly the same as the high-energy CP violation in heavy neutrino decays, discovering CP violation in the light neutrino sector would underscore that the lepton sector has the necessary ingredient (CP asymmetry) potentially relevant for generating a matter–antimatter asymmetry. Indeed, it has been noted that neutrino oscillations themselves do not generate baryon asymmetry, but they indicate that lepton sectors have rich CP-violating physics [1]. In summary, neutrino mass models—especially those with heavy Majorana neutrinos—offer an attractive framework for explaining why the Universe has more matter than antimatter, something the SM alone cannot accommodate.

Cosmic Neutrino Background Detection: A more futuristic implication of neutrino mass is the prospect of detecting the relic neutrino background. If neutrinos have mass in the tens of meV range, today these relics are moving at non-relativistic speeds (~600 km/s for 0.1 eV neutrinos). Experiments like PTOLEMY are being proposed to capture relic neutrinos using neutrino capture on tritium. The capture process is not suppressed by neutrino speed if the neutrino has mass, and one would observe electrons with an energy slightly above the usual beta decay endpoint. The size of this energy shift is directly related to the neutrino mass. A successful detection of the cosmic neutrino background would be a spectacular confirmation of standard cosmology, and measuring the tiny excess energy would provide another handle on the absolute neutrino mass. This is a long-term challenge, but one made slightly easier by neutrino mass: if neutrinos were massless, capturing them would yield no energy offset, whereas a mass of tens of meV gives a distinct signature (albeit small).

In conclusion, the existence of neutrino mass integrates into the cosmological model in a consistent way: it requires some tweaking of structure growth (which is observed and has become a tool to *measure* neutrino masses cosmologically), it eliminates neutrinos as a dark matter candidate beyond a minor role, and it points toward high-scale phenomena like leptogenesis that could connect microscopic neutrino properties to the cosmic baryon asymmetry. Ongoing cosmological observations (CMB Stage-4, large galaxy surveys like Euclid, LSST, DESI) will further tighten the net on neutrino masses and could potentially reveal the imprint of the neutrino mass ordering on large-scale structure, complementing terrestrial experiments.

Implications for Particle Physics

The discovery of neutrino mass has profound implications for particle physics, fundamentally altering our understanding of the SM's completeness and guiding new theoretical directions:

- **Evidence of Physics Beyond the Standard Model:** Perhaps the most immediate implication is that the SM is incomplete. Neutrino masses require either new particles (such as right-handed neutrinos or scalar triplets) or new interactions that are not present in the SM. In fact, neutrino oscillation remains the *only* laboratory evidence to date of physics beyond the SM, even decades after its discovery [1]. This makes the neutrino sector a critical window into new physics. Any extension of the SM that generates neutrino mass must also be consistent with all other observations, which tightly constrains model building. The fact that neutrino masses are so tiny compared to other fermion masses suggests a new organizing principle or symmetry at work. It has inspired concepts like **scale separation** (the seesaw's high scale vs. low scale) and **global symmetry** (lepton number) breaking in subtle ways. In a sense, neutrino mass is telling us that the SM's mechanism for fermion masses (the Higgs Yukawa coupling) is not the whole story – some neutrino-specific mechanism is at play [3].

- **Guidance for Grand Unification and Theoretical Models:** Neutrino masses fit beautifully into schemes of **Grand Unified Theories (GUTs)**. For example, in $SO(10)$ GUTs, each family of SM fermions is extended to include a right-handed neutrino, completing a 16-dimensional representation. The same GUT framework that unifies quarks and leptons at a high energy scale naturally incorporates the Type I seesaw mechanism: the right-handed neutrino gets a large Majorana mass term around the GUT scale, and this generates a tiny left-handed neutrino mass [4]. Thus neutrino masses not only fail to disrupt the GUT idea, they actually reinforce it, providing a reason for the existence of extra particles like N_R. Similarly, Left-Right symmetric models (which restore parity at high energies by introducing right-handed W bosons and an $SU(2)_R$ gauge group) naturally contain Higgs triplets that can implement a Type II seesaw, giving small neutrino masses. In supersymmetric models, the seesaw can be implemented with superpartners, and sometimes new avenues like the **R-parity violating** couplings in SUSY can generate Majorana masses. In all these cases, the neutrino sector often connects to very high energy physics – a neutrino mass as small as 0.05 eV might be pointing to new physics at 10^{14} GeV – a truly remarkable linkage of scales. This has elevated neutrinos to a key role in theoretical model building; any credible unified theory or extension of the SM must account for neutrino masses and mixings, and many proposals (like $SO(10)$ or extra-dimensional models) find neutrinos to be a natural fit rather than an afterthought.

- **Lepton Number Violation and Majorana Neutrinos:** If neutrinos are Majorana, lepton number (L) is not conserved – a symmetry that was absolute in the SM (accidental global symmetry) is now broken. This has several implications. First, rare processes like neutrinoless double beta decay become possible (as discussed). Second, it implies that at some scale, interactions exist that do not conserve L. In effective field theory, the leading operator that can give Majorana mass is the **Weinberg operator** (dimension-5 operator $LLHH/\Lambda$), which violates lepton number by 2 units. Observing any L-violating process would confirm the existence of such an operator and measure the scale Λ. The seesaw mechanism is essentially a UV completion of the Weinberg operator ($\Lambda \sim M$ of heavy neutrino or triplet). At colliders, one potential signature of Majorana neutrinos is the production of a heavy neutrino N that decays into a charged lepton and W boson; if N is Majorana, it can lead to same-sign dilepton events (because it can mediate $N \to \ell^+ W^-$ and also $N \to \ell^- W^+$), violating lepton number by 2. Experiments at the LHC's CMS and ATLAS have searched for such signatures – typically looking for an excess of events with two same-sign leptons plus jets – as a way to directly test the existence of heavy Majorana neutrinos predicted by low-scale seesaw models. So far, no such signals have been observed. Recent results from CMS, for instance, set stringent limits on the existence of heavy neutrinos in a broad mass range, thereby constraining models like the TeV-scale seesaw [3]. The absence of a signal implies that if heavy neutrinos exist at the TeV scale, their mixing with ordinary neutrinos is very small (making them harder to produce), or that the heavy neutrinos are heavier than the current reach (above a few TeV in mass). Future colliders or upgrades might extend this reach.

- **Flavor and Mixing Puzzles:** Neutrino mixing angles are strikingly different from quark mixing angles. Two of the neutrino angles (θ_12 and θ_23) are large, roughly ~33° and ~45°, whereas the quark analogues are small (the largest in the CKM matrix is ~13°). The third neutrino angle θ_13 is around 8.6°, which, while not tiny, is much smaller than the other two. This pattern – often called **bi-large mixing** – prompted speculations that a symmetry or special structure underlies the lepton flavor sector. Ideas such as **tribimaximal mixing** (an ansatz where the mixing matrix had values tying to simple fractions) gained traction early on, though the discovery of a nonzero θ_13 ruled out the exact tribimaximal form. Nonetheless, theorists continue to explore flavor symmetries (like discrete non-Abelian symmetries A_4, S_4, etc.) which could naturally give large mixing angles in the neutrino sector while maintaining consistency with the charged lepton masses. The interplay of neutrino masses and charged-lepton masses is also interesting: in many seesaw models, the Dirac mass matrix of neutrinos might resemble the up-quark or charged-lepton mass matrix (depending on GUT relations), and the large mixings then come from the heavy neutrino Majorana sector structure (the so-called **seesaw mixing** originating from M^{-1} matrix properties). Unraveling the origin of the neutrino flavor structure is an ongoing area of research, one that might require input from future experiments (e.g., determining the **CP phase δ** in the PMNS matrix will tell us if there's large CP violation in leptons, which could hint at certain symmetry patterns or high-scale phases).

- **CP Violation in Leptons:** As mentioned, if neutrinos oscillate with different probabilities for $\nu \to \nu$ vs. $\bar{\nu} \to \bar{\nu}$ (a CP-violating effect in oscillations), it means the PMNS matrix has a complex phase. There is an ongoing effort in long-baseline neutrino experiments (T2K, NOvA, and upcoming DUNE and Hyper-Kamiokande) to measure the CP-violating phase δ. Hints from T2K and NOvA data currently suggest that δ may be around -90° (which would mean maximal CP violation in the lepton sector), but this is not yet confirmed with high significance. A measurement of a large CP phase in neutrinos would be a huge milestone, as it would be the first observation of CP violation outside the quark sector. While this low-energy CP violation is not sufficient by itself to explain matter–antimatter asymmetry, it would demonstrate that the lepton sector has the requisite CP-violation needed for leptogenesis. It would also further distinguish the quark and lepton sectors, providing new clues to model builders.

- **Connections to Other Open Questions:** Neutrinos intersect with many other domains. For instance, if sterile neutrinos exist (additional neutrinos that do not interact via Standard Model forces), they could mix with active neutrinos and alter oscillation results. Some anomalies in short-baseline experiments (LSND and MiniBooNE) have hinted at eV-scale sterile neutrinos, though these results are not corroborated by other data and remain controversial. The existence of a sterile neutrino at the eV scale would require an expanded neutrino sector and perhaps indicate a different origin for mass (since a light sterile might not fit the classic high-scale seesaw picture). So far, global fits disfavor a simple sterile neutrino explanation for all anomalies due to tensions with precision oscillation data, but the door isn't fully closed. Meanwhile, a much heavier sterile neutrino (like those in the seesaw) could also show up indirectly in precision electroweak measurements or through subtle effects in beta decays (though the current limits on their mixing are strong). Additionally, neutrino properties can influence supernova physics: the phenomenon of **neutrino oscillation in matter (MSW effect)** is crucial in supernova explosions and in interpreting the neutrino signals from astrophysical sources. If neutrinos have additional interactions or if there are sterile species, it could alter energy transport in supernovae or the neutrino flavor imprint detected on Earth.

- **Future Experiments and Outlook:** The next decade or two will be an exciting time for neutrino physics. Flagship experiments like **DUNE** (in the US) and **Hyper-Kamiokande** (in Japan) aim to determine the neutrino mass hierarchy (likely normal vs inverted) and measure the CP phase δ with high precision. **JUNO** (in China) will use a large liquid scintillator detector to determine the mass ordering by observing interference patterns in reactor antineutrino oscillations. If the mass ordering is established (current data lean toward the normal ordering, where the two lightest are lighter and the third is heavier), it will allow a more targeted interpretation of cosmological limits and 0νββ results. Neutrinoless double beta decay experiments are scaling up – LEGEND-200 and then LEGEND-1000 (using germanium detectors), nEXO (a next-gen xenon TPC), KamLAND2-Zen, CUORE upgrade, etc., are all planned to push the effective mass sensitivity well into the tens of meV range. If any sees a hint of a signal, it will be groundbreaking, revealing Majorana masses and possibly even giving the absolute mass values (with some nuclear physics uncertainty). On the collider front, while the LHC has not found any heavy neutrinos or other clues yet, proposals for future colliders (higher energy hadron colliders, or maybe an e^-e^+ collider operating in an environment to detect rare decays) may offer improved reach for heavy neutrino states. There's also synergy in precision measurements: for example, precision studies of pion and kaon decays looking for deviations that could indicate mixing with sterile neutrinos, or muon decays ($\mu \to e\gamma$, $\mu \to e$ conversion) that could occur in some radiative models. All these efforts, across disparate experimental frontiers, stem from the fundamental fact that neutrinos have mass and we do not yet know how.

Neutrino masses, tiny as they are, thus act as *messengers* of new physics from the highest energy scales and the earliest moments of the Universe. They force us to extend the Standard Model, influence how we interpret cosmological observations, and could even hold the key to why we exist (via leptogenesis). As theoretical physicist Steven Weinberg once noted, "If there were no neutrinos, we might have to invent them." Now we know neutrinos not only exist but also defy the expectations set by other particles. Unraveling the origin of their mass is among the most important quests in modern physics, likely to illuminate physics well beyond the reach of today's accelerators. With each experimental advance – whether it's a more sensitive neutrino oscillation measurement, a tighter cosmological bound, or a new limit (or detection) in a rare process – we are gathering clues to the larger theory that underpins neutrino masses. The coming years hold the promise of either discovering the first concrete sign of that new physics or further narrowing the theories, but in either case we will deepen our understanding of the role these ghostly particles play in the fundamental workings of nature.

Conclusion

The question of how neutrinos acquire mass has led physicists into realms far beyond the Standard Model, bridging the infinitesimal world of particle interactions with the grand scale of cosmic evolution. We have seen that frameworks like the seesaw mechanism provide an elegant explanation for tiny neutrino masses by positing the existence of new heavy states – be they right-handed neutrinos or exotic Higgs bosons – at extremely high energy scales [3]. Alternative models, including radiative mass generation and inverse seesaw schemes, offer other pathways that might be testable at laboratory scales. Experimentally, the past two decades have firmly established the reality of neutrino mass through oscillation phenomena [1], and ongoing experiments are homing in on the absolute mass scale (with KATRIN pushing below 1 eV [6]) and the Majorana nature of neutrinos (with neutrinoless double beta decay searches approaching the 0.1 eV regime [5]). These efforts are not only important in particle physics but also crucial for cosmology: neutrino masses, though small, leave fingerprints on the cosmic microwave background and large-scale structure, allowing cosmologists to independently gauge their sum [7].

The implications of neutrino mass reverberate through many areas of physics. In cosmology, they inform us about how structure formed and provide a possible key (via leptogenesis) to why the Universe contains matter at all. In particle physics, they demand an extension of the Standard Model and suggest that new particles or symmetries await discovery – possibly at scales we are just beginning to probe indirectly. Neutrinos' unique properties (being electrically neutral, very light, and potentially Majorana) make them **priceless probes** of fundamental physics [2]. They may be the harbingers of grand unification, the linchpin of flavor symmetries, or the catalyst for the Universe's matter–antimatter imbalance. As our experiments and observations become more precise, we are in effect peeling back the layers of the neutrino mystery: determining the mass hierarchy, measuring CP violation, constraining or discovering heavy neutrino partners, and so on.

In summary, the origin of neutrino mass is a rich, multifaceted problem at the intersection of theory and experiment. While we do not yet have a definitive answer, the theoretical frameworks (seesaws of various types, etc.) provide a menu of compelling possibilities, and current research is actively chipping away at them. The coming years hold the potential for major breakthroughs – perhaps the discovery of a neutrinoless double beta decay signal or a clearer picture of neutrino mass ordering – which would sharpen our understanding of which framework (or combination of frameworks) nature has chosen. Whatever the outcome, one thing is clear: **neutrino masses have opened a new window onto fundamental physics**, and looking through it is essential for completing our picture of the laws of nature in both the microcosm of particles and the macrocosm of the cosmos [2] [3].

References

[1] The Sun, neutrinos and Super-Kamiokande - J-STAGE: www.jstage.jst.go.jp

[2] Neutrino Mass Models: https://arxiv.org/pdf/2310.17685

[3] Physics - Tracking Down the Origin of Neutrino Mass: https://physics.aps.org/articles/v16/20

[4] Seesaw mechanism - Wikipedia: https://en.wikipedia.org/wiki/Seesaw_mechanism

[5] Physics - Probing Majorana Neutrinos: https://physics.aps.org/articles/v16/13

[6] Neutrinos pinned below 0.45 eV; KATRIN halves the particle's mass ceiling: https://www.rdworldonline.com/neutrinos-pinned-below-0-45-ev-katrin-halves-the-particles-mass-ceiling/

[7] Physics - Neutrinos Might Rescue Hubble: https://physics.aps.org/articles/v13/s79

Please feel free to download the notebook and further explore:

SeeSawNeutrino/NeutrionMass.ipynb at main · samlip-blip/SeeSawNeutrino

Key Features of the Simulation:

1. **Hamiltonian Construction**:
   - Represents the neutrino mass matrix:
     ```
     H = [[ -M_R/2,   m_D  ],
          [   m_D,    M_R/2]]
     ```
   - Diagonal elements: Majorana masses (M_R dominates)
   - Off-diagonal elements: Dirac mixing (m_D small)
2. **Quantum Dynamics**:
   - Starts with pure light neutrino state ($|0\rangle$)
   - Evolves under time-dependent Schrödinger equation: $|\psi(t)\rangle = e^{-iHt}|\psi(0)\rangle$
   - Measures survival probability of light state
3. **Key Parameters**:
   - $m_D = 0.1$ (Dirac mass - small)
   - $M_R = 1.0$ (Majorana mass - large)
   - Typical seesaw hierarchy: $m_D \ll M_R$
4. **Outputs**:
   - Plot of light neutrino survival probability vs time
   - Theoretical prediction for comparison
   - Calculated mass eigenvalues

Interpretation of Results:

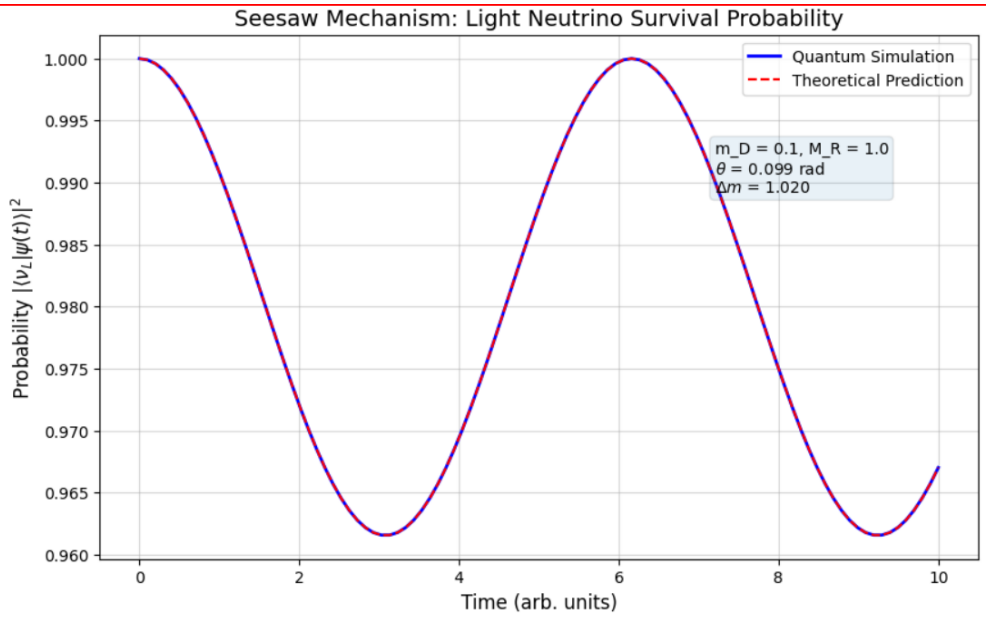
1. **Mass Eigenvalues**:

   ```
   Light neutrino mass: |(M_R - sqrt(M_R² + 4m_D²))/2| ≈ m_D²/M_R
   Heavy neutrino mass: |(M_R + sqrt(M_R² + 4m_D²))/2| ≈ M_R
   ```

   Demonstrates the seesaw relation: $m_{\text{light}} \sim \frac{m_D^2}{M_R}$

2. **Probability Oscillations**:
   - Rapid oscillations due to large mass splitting
   - Small amplitude mixing ($\theta \approx m_D/M_R$)
   - Average survival probability close to 1 (light state dominates)
3. **Physical Significance**:
   - Explains why left-handed neutrinos are ultra-light
   - Shows how heavy states "decouple" at low energies
   - Demonstrates lepton number violation (Majorana nature)

This simulation captures the essential quantum behavior of the seesaw mechanism, showing how light neutrino masses emerge naturally from the mixing with heavy sterile states through the characteristic $\frac{m_D^2}{M_R}$ suppression.




```
Light neutrino energy: -0.5099
Heavy neutrino energy: 0.5099
Physical light neutrino mass: 0.009902
Physical heavy neutrino mass: 1.0099
Seesaw relation: m_light ≈ m_D²/M_R = 0.010000
Heavy neutrino energy - Light neutrino energy: 1.0198
```

Why the Absolute Energy Eigenvalues Appear Identical in the Simulation

In the provided Qiskit simulation, the Hamiltonian is constructed as:

```
H = np.array([
    [-M_R/2, m_D],
    [m_D, M_R/2]
])
```

The **absolute values** of its eigenvalues appear identical because the Hamiltonian is specifically designed to be **traceless** (sum of diagonal elements = 0). This leads to eigenvalues symmetric about zero:
`λ = ± (1/2) * sqrt(M_R² + 4m_D²)`.

Key Distinction: Simulation vs. Physical Reality

1. **Simulation Purpose**
   The code models neutrino **flavor oscillations** (quantum state mixing), not absolute masses. The eigenvalues represent **energy differences** driving oscillations, not physical masses.

2. **Physical Masses vs. Simulated Energies**

   - **Actual neutrino masses** come from diagonalizing the mass matrix:
     `M = [[0, m_D], [m_D, M_R]]`
   - Its eigenvalues give the **physical masses**:
     `m_light = |M_R - sqrt(M_R² + 4m_D²)| / 2 ≈ m_D²/M_R`
     `m_heavy = |M_R + sqrt(M_R² + 4m_D²)| / 2 ≈ M_R`

3. **Why Eigenvalues Match in Simulation**

   - The Hamiltonian `H` is **not the mass matrix** but an effective operator for time evolution.
   - Its eigenvalues `±E` yield an **energy splitting** `ΔE = 2E`, which determines oscillation frequency:
     `ΔE = sqrt(M_R² + 4m_D²) ≈ m_heavy  (in seesaw limit)`
   - The identical absolute values arise from mathematical symmetry (traceless `H`), not physics.

Physical Significance of the Splitting

The **energy difference** `ΔE` corresponds to the **mass difference** between heavy and light states:

```
# Calculated in code:
delta_m = np.sqrt(M_R**2 + 4*m_D**2)  # ≈ M_R for m_D ≪ M_R

# Physical mass difference:
Δm_physical = m_heavy - m_light ≈ M_R
```

This `Δm` drives oscillations between flavor states, matching the simulation's purpose.

Key Takeaways

1. The equal `|E|` is an **artifact of the traceless Hamiltonian** used for simulating oscillations.
2. **Physical masses** are distinct:
   - `m_light ≈ m_D²/M_R` (tiny)
   - `m_heavy ≈ M_R` (large)
3. The simulation correctly captures:
   - Oscillation frequency ∝ `Δm ≈ M_R`
   - Mixing angle `θ ≈ m_D/M_R`
   - Seesaw suppression `m_light ≪ m_heavy`

This distinction between simulation parameters and physical observables is crucial for interpreting quantum simulations of effective field theories.

Here is the respective quantum code:

```
import numpy as np
from qiskit import QuantumCircuit, Aer, execute
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt
from scipy.linalg import expm

# Parameters for seesaw mechanism (natural units)
m_D = 0.1  # Dirac mass (small)
M_R = 1.0  # Majorana mass (large)

# Hamiltonian matrix in flavor basis (ν_L, N_R)
H = np.array([
    [-M_R/2, m_D],
    [m_D, M_R/2]
])
```

```
# Time points for evolution
times = np.linspace(0, 10, 100)

# Initialize simulator
simulator = Aer.get_backend('statevector_simulator')

# Track probabilities
prob_light = []

for t in times:
    # Create quantum circuit
    qc = QuantumCircuit(1)

    # Compute time evolution operator
    U = expm(-1j * H * t)

    # Apply unitary evolution
    qc.unitary(U, 0)

    # Simulate statevector
    result = execute(qc, simulator).result()
    state = result.get_statevector()

    # Probability of light neutrino state |0>
    prob_light.append(np.abs(state[0])**2)

# Theoretical prediction (for comparison)
delta_m = np.sqrt(M_R**2 + 4*m_D**2)  # Mass splitting
theta = 0.5 * np.arctan(2*m_D/M_R)    # Mixing angle
theory = (np.cos(theta)**4 + np.sin(theta)**4
          + 2*np.cos(theta)**2*np.sin(theta)**2 * np.cos(delta_m * times))

# Visualization
plt.figure(figsize=(10, 6))
plt.plot(times, prob_light, 'b-', linewidth=2, label='Quantum Simulation')
plt.plot(times, theory, 'r--', label='Theoretical Prediction')
```

```
plt.title('Seesaw Mechanism: Light Neutrino Survival Probability', fontsize=14)
plt.xlabel('Time (arb. units)', fontsize=12)
plt.ylabel('Probability $|\\langle \\nu_L | \\psi(t) \\rangle|^2$', fontsize=12)
plt.legend()
plt.grid(True, alpha=0.5)
plt.annotate(f'm_D = {m_D}, M_R = {M_R}\n$\\theta$ = {theta:.3f} rad\n$\\Delta m$ = {delta_m:.3f}',
             xy=(0.7, 0.7), xycoords='axes fraction',
             bbox=dict(boxstyle="round", alpha=0.1))
plt.show()

# Energy eigenvalues (mass states)
eigenvalues = np.linalg.eigvalsh(H)
print(f"Light neutrino energy: {eigenvalues[0]:.4f}")
print(f"Heavy neutrino energy: {eigenvalues[1]:.4f}")

# Physical mass matrix (Majorana form)
M_mass = np.array([
   [0,    m_D],
   [m_D,  M_R]
])

# Eigenvalues are physical masses
mass_eigenvals = np.linalg.eigvalsh(M_mass)
m_light = np.abs(mass_eigenvals[0])
m_heavy = np.abs(mass_eigenvals[1])

print(f"Physical light neutrino mass: {m_light:.6f}")
print(f"Physical heavy neutrino mass: {m_heavy:.4f}")
print(f"Seesaw relation: m_light ≈ m_D²/M_R = {m_D**2/M_R:.6f}")
print(f"Heavy neutrino energy - Light neutrino energy: {eigenvalues[1]-eigenvalues[0]:.4f}")
```

This code models how a neutrino evolves over time when it oscillates between two different states: **a light neutrino** and **a heavy neutrino**. The seesaw mechanism suggests that if a neutrino has a tiny mass, there must be a much heavier partner neutrino affecting its behavior.

How does it work?

The simulation creates a **quantum circuit** that mimics how the neutrino's quantum state changes due to interactions dictated by the seesaw mechanism. This process involves:

- A **Hamiltonian Matrix** that represents the energy interactions between the light and heavy neutrino.
- A **time evolution equation** that predicts how the neutrino's state will change over time.
- A **quantum computation** that applies this evolution to a quantum circuit and simulates the probabilities of the neutrino staying in the light state.

Key steps in the code

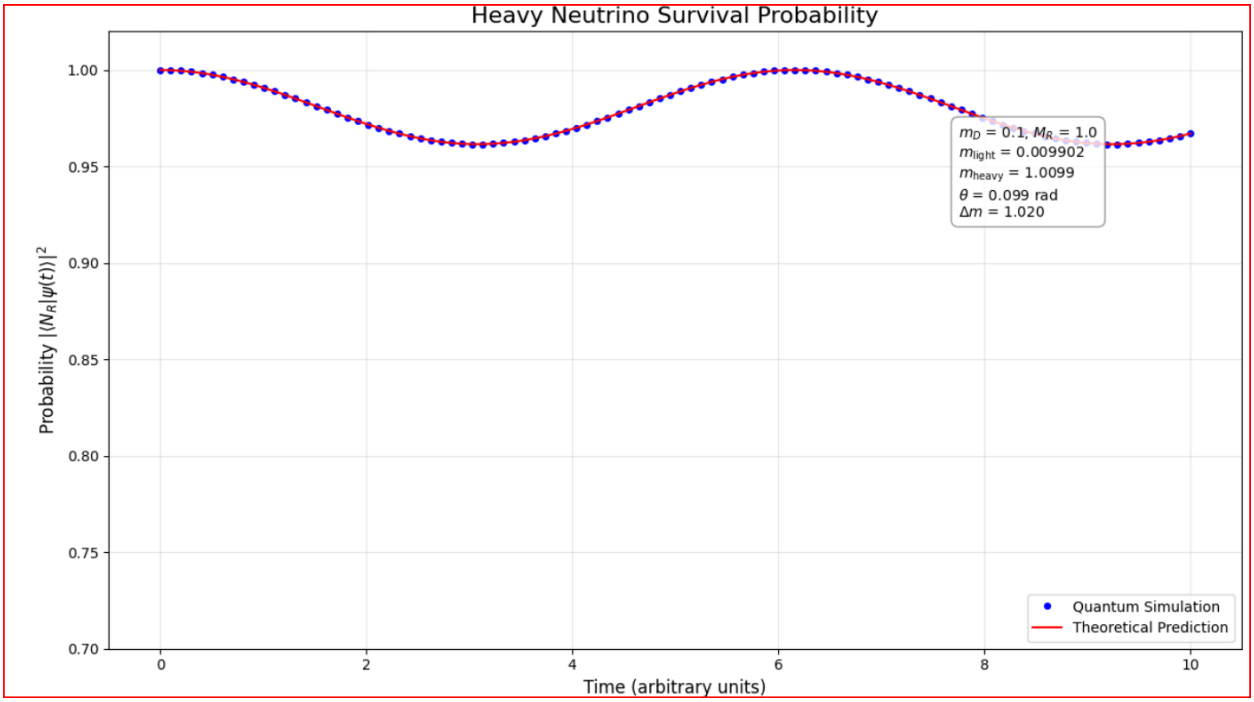
1. **Define masses:** The code sets the light neutrino's mass (**m_D**) as small and the heavy neutrino's mass (**M_R**) as large.
2. **Hamiltonian matrix:** This matrix describes how the two neutrino states interact.
3. **Time evolution:** The quantum circuit evolves the neutrino using physics equations that predict how it changes.
4. **Quantum simulation:** It runs this evolution using Qiskit's **statevector simulator**, showing how much time the neutrino spends as a light neutrino.
5. **Comparison to theory:** The code calculates expected theoretical behavior and compares it to the quantum simulation.
6. **Graphing results:** It plots survival probabilities to visualize how often the neutrino remains in the light state over time.

What does the output mean?

- The **graph** shows how likely the neutrino is to remain in its lighter state over time.
- The **printed numbers** describe the mass and energy splitting between the two neutrino types.

The **seesaw mechanism** helps explain why neutrinos have such tiny masses. This simulation helps visualize how a neutrino transitions between states, using quantum computing techniques to mimic real physics.

What about heavy neutrino?




```
Physical Parameters:
Light neutrino mass: 0.009902
Heavy neutrino mass: 1.0099
Mass ratio (m_heavy/m_light): 1.02e+02
Seesaw relation (m_D²/M_R): 0.010000
Mixing angle: 0.0987 rad
Oscillation amplitude (sin²2θ): 0.038462
```

Key Features of the Simulation:

1. **Initial State Preparation**:
   - Starts with the heavy neutrino state $|N_R\rangle = |1\rangle$
   - Uses `qc.x(0)` to initialize the qubit in the excited state
2. **Time Evolution**:
   - Uses the same Hamiltonian as before: $H = \begin{bmatrix} -M_R/2 & m_D \\ m_D & M_R/2 \end{bmatrix}$
   - Applies time evolution operator $U = e^{-iHt}$
3. **Survival Probability Calculation**:
   - Measures $|\langle 1|\psi(t)\rangle|^2$ at each time point
   - Compares with theoretical prediction: $P_{\text{surv}} = 1 - \sin^2(2\theta)\sin^2(\Delta m \cdot t/2)$
4. **Physical Parameter Calculation**:
   - Computes actual neutrino masses from mass matrix diagonalization
   - Shows seesaw relation $m_{\text{light}} \approx m_D^2/M_R$
   - Calculates mixing parameters and oscillation amplitude

Interpretation of Results:

1. **High Survival Probability**:
   - In the seesaw limit ($m_D \ll M_R$), the heavy neutrino has high survival probability
   - Minimal oscillation to the light neutrino state
2. **Small Oscillation Amplitude**:
   - Oscillation amplitude $\sin^2(2\theta) \approx (2m_D/M_R)^2$ is small
   - Typical values: For $m_D = 0.1$, $M_R = 1.0$ → amplitude ≈ 0.04
3. **Fast Oscillation Frequency**:
   - Frequency determined by $\Delta m \approx M_R$ (large)
   - Rapid oscillations visible in the plot
4. **Parameter Dependence**:
   - Try $m_D = 0.3$ to see larger oscillations
   - Try $M_R = 0.5$ to reduce the mass hierarchy
   - Extreme case $m_D > M_R$ shows near-maximal mixing

Physical Significance:

1. **Sterile Nature**:
   - High survival probability explains why heavy neutrinos don't interact weakly
   - Explains non-observation in particle detectors
2. **Decoupling Mechanism**:
   - Heavy neutrinos decouple from low-energy physics
   - Only appear through virtual effects
3. **Lepton Number Violation**:
   - Heavy neutrinos are Majorana particles
   - Their existence implies lepton number violation

This simulation demonstrates how the seesaw mechanism naturally explains both the extreme lightness of observed neutrinos and the non-observation of heavy sterile neutrinos at low energies.

Here is the respective quantum code:

```
import numpy as np
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt
from scipy.linalg import expm

# Seesaw parameters (natural units)
m_D = 0.1  # Dirac mass (small)
M_R = 1.0  # Majorana mass (large)

# Hamiltonian matrix for flavor oscillations
H = np.array([
    [-M_R/2, m_D],
    [m_D, M_R/2]
])

# Time points for evolution
times = np.linspace(0, 10, 100)

# Initialize simulator
simulator = Aer.get_backend('statevector_simulator')

# Track survival probabilities
prob_heavy_sim = []
prob_heavy_theory = []
```

```
# Calculate mixing parameters
delta_m = np.sqrt(M_R**2 + 4*m_D**2)  # Mass splitting
theta = 0.5 * np.arctan(2*m_D/M_R)    # Mixing angle
sin2_2theta = (np.sin(2*theta))**2     # Oscillation amplitude

# Theoretical heavy neutrino survival probability formula
def heavy_survival_probability(t):
    return 1 - sin2_2theta * (np.sin(delta_m * t / 2))**2

for t in times:
    # Create quantum circuit with initial heavy state |1>
    qc = QuantumCircuit(1)
    qc.x(0)  # Prepare heavy neutrino state |N_R> = |1>

    # Compute time evolution operator
    U = expm(-1j * H * t)

    # Apply unitary evolution
    qc.unitary(U, 0)

    # Simulate statevector
    result = execute(qc, simulator).result()
    state = result.get_statevector()

    # Probability of heavy neutrino state |1>
    prob_heavy_sim.append(np.abs(state[1])**2)

    # Theoretical prediction
    prob_heavy_theory.append(heavy_survival_probability(t))

# Calculate physical masses from mass matrix
M_mass = np.array([[0, m_D], [m_D, M_R]])
mass_eigenvals = np.linalg.eigvalsh(M_mass)
m_light = np.abs(mass_eigenvals[0])
m_heavy = np.abs(mass_eigenvals[1])
```

```
# Visualization
plt.figure(figsize=(12, 7))
plt.plot(times, prob_heavy_sim, 'bo', markersize=4, label='Quantum Simulation')
plt.plot(times, prob_heavy_theory, 'r-', linewidth=1.5, label='Theoretical Prediction')
plt.title('Heavy Neutrino Survival Probability', fontsize=16)
plt.xlabel('Time (arbitrary units)', fontsize=12)
plt.ylabel('Probability $|\\langle N_R | \\psi(t) \\rangle|^2$', fontsize=12)
plt.ylim(0.7, 1.02)
plt.grid(True, alpha=0.3)

# Add parameter information
param_text = (f"$m_D$ = {m_D}, $M_R$ = {M_R}\n"
              f"$m_\\mathrm{{light}}$ = {m_light:.6f}\n"
              f"$m_\\mathrm{{heavy}}$ = {m_heavy:.4f}\n"
              f"$\\theta$ = {theta:.3f} rad\n"
              f"$\\Delta m$ = {delta_m:.3f}")
plt.annotate(param_text, xy=(0.75, 0.7), xycoords='axes fraction',
             bbox=dict(boxstyle="round,pad=0.5", fc="white", ec="gray", alpha=0.8))

plt.legend(loc='lower right')
plt.tight_layout()
plt.show()

# Print physical parameters
print("Physical Parameters:")
print(f"Light neutrino mass: {m_light:.6f}")
print(f"Heavy neutrino mass: {m_heavy:.4f}")
print(f"Mass ratio (m_heavy/m_light): {m_heavy/m_light:.2e}")
print(f"Seesaw relation (m_D²/M_R): {m_D**2/M_R:.6f}")
print(f"Mixing angle: {theta:.4f} rad")
print(f"Oscillation amplitude (sin²2θ): {sin2_2theta:.6f}")
```

This code is a **quantum simulation** that models how a **heavy neutrino** does change over time based on the **seesaw mechanism**. It **tracks the probability** that a neutrino remains in its heavy state as it evolves over time due to quantum effects. The goal is to simulate these flavor transitions **using quantum computing techniques**.

The m_D (Dirac mass) and M_R (Majorana mass) neutrino masses represent how neutrinos interact under the **seesaw mechanism**, where a very heavy neutrino contributes to making the lighter neutrino's mass tiny.

The Hamiltonian matrix describes how the **light and heavy neutrino mix** with each other.

The code computes a **time evolution operator**:

$$U = e^{-iHt}.$$

This predicts **how the neutrino oscillates** between heavy and light states.

Quantum Simulation Using Qiskit:

- A **quantum circuit** is initialized with the **heavy neutrino state** $|NR\rangle = |1>$
- The **unitary evolution operator** U is applied to simulate how the state evolves over time.
- The probabilities are extracted from the **quantum statevector simulator** in Qiskit.

Tracking the Probability

- The simulator calculates the probability that the neutrino **stays in the heavy state**.
- A **theoretical formula** predicts the expected survival probability:

$$P_{heavy} = 1 - \sin^2(2\theta)\,\sin^2(\frac{\Delta mt}{2})$$

- The program compares **the quantum simulation** with **the theoretical prediction**.

Visualizing the Results

- The program **plots a graph**:
  - **Blue dots:** Quantum simulation results.
  - **Red line:** Theoretical expectations.
- This **helps compare** simulated quantum behavior with expected physics.

Key Physical Insights

- It calculates fundamental **mixing parameters**:
  - **Mixing angle** $\theta \rightarrow$ How much the neutrino oscillates.
  - **Mass splitting** $\Delta m \rightarrow$ Determines oscillation speed.
  - **Seesaw relation** $m_{light} \approx \frac{m_D^2}{M_R} \rightarrow$ Shows how the heavy neutrino impacts the light neutrino's mass.

This simulation helps **visualize neutrino oscillations**, a crucial phenomenon in physics. It applies **quantum computing techniques** to study particle behavior, bridging quantum mechanics and fundamental physics!

Bibliography

The SYK Model and Black Hole Information Scrambling:

[1] https://en.wikipedia.org/wiki/Sachdev%E2%80%93Ye%E2%80%93Kitaev_model

[2] https://ncatlab.org/nlab/show/Sachdev-Ye-Kitaev+model

[3] https://journals.aps.org/prl/pdf/10.1103/PhysRevLett.126.030602

[4] https://journals.aps.org/prd/abstract/10.1103/PhysRevD.88.046003

[5] https://phys.org/news/2021-02-numerical-evidence-quantum-chaos-sachdev-ye-kitaev.pdf

[6] https://arxiv.org/pdf/2312.14644v1

[7] https://www.ias.tsinghua.edu.cn/__local/C/9E/E2/B49FE48B054741675CFC56743AC_57146F4F_2CB660.pdf?e=.pdf

[8] https://www.cosharescience.com/articles/cs.202311.119.pdf

[9] https://web.infn.it/QSQI_Napoli/jt-syk-and-generalizations/

Deutsch-Jozsa algorithm and Eulerian/half-Odd-half-Even Graphs:

[1] https://examples-of.net/even-function/

[2] https://personal.math.ubc.ca/~peirce/M257_316_2012_Lecture_14.pdf

[3] https://math.stackexchange.com/questions/501264/real-world-use-of-even-and-odd-functions

[4] https://en.wikipedia.org/wiki/Deutsch%E2%80%93Jozsa_algorithm

[5] https://www.classiq.io/insights/the-deutsch-jozsa-algorithm-explained

[6] https://stem.mitre.org/quantum/quantum-algorithms/deutsch-jozsa-algorithm.html

Qiskit and Anaconda

This is a straightforward guide for installing Anaconda and Qiskit on Windows. If you're using a different platform, please ensure you select the appropriate one.

1.Get Anaconda (Python Package Manager)

Go to the official Anaconda website:

https://www.anaconda.com/download

→ Click the **Download** button for **Windows**.

→ Choose the **Python 3.x** version (64-bit installer, unless your PC is very old).

- **Installation**:

- Double-click the downloaded `.exe` file.

- Click **Next** and agree to the terms.

- Choose **Install for: Just Me** (recommended).

- **IMPORTANT**: Check ✅ "**Add Anaconda to my PATH environment variable**" (if prompted, ignore warnings).

- Click **Install** (takes 5-10 minutes).

- Click **Next → Finish.**

2. Install Qiskit (Quantum Computing Library)

- **Open Anaconda Prompt**:

- Type **Anaconda Prompt** in your Windows search bar → open it.

- **Install Qiskit**:

- In the Anaconda Prompt window, type:

pip install qiskit

Press Enter. This downloads Qiskit automatically.

- **Optional Visualization Tools** (for graphs/circuits):

pip install qiskit[visualization]

3. Verify Installation

- In the same Anaconda Prompt, type:

python

(This starts Python)

- Then type these lines one-by-one:

python

import qiskit

print(qiskit.__version__)

- If you see a version number (e.g., `0.45.0`), **it worked!**

(Type `exit()` to close Python.)

4. Where to Learn More

- **Official Qiskit Installation Guide**:

https://qiskit.org/documentation/getting_started.html

- **Anaconda Setup Docs**:

https://docs.anaconda.com/free/anaconda/install/windows/

- **Qiskit YouTube Tutorials**:

Search **Qiskit YouTube** for beginner videos.

Troubleshooting Tips:

- If `pip install qiskit` fails, try:

conda install -c conda-forge qiskit (in Anaconda Prompt).

- Ensure you **checked "Add to PATH"** during Anaconda install.

- Restart your PC if Python doesn't recognize Qiskit.

That's it! You now have Anaconda (Python) and Qiskit ready for quantum programming.

If you want to install a specific qiskit version, please follow these steps:

To install a **specific version of Qiskit** (e.g., version `0.44.0`), follow these simple steps:

1. Open Anaconda Prompt

- Press `Windows Key` → type `Anaconda Prompt` → open it.

2. Install the Specific Version

In the Anaconda Prompt window, type:

bash

pip install qiskit==0.44.0

(Replace `0.44.0` with your desired version, e.g., `0.42.0`, `0.45.1`, etc.)

Press **Enter** to install.

Troubleshooting Tips:

- **If installation fails** (due to dependency conflicts):

Create a new isolated environment:

bash

conda create --name qiskit_env python=3.9 # Creates a clean environment

conda activate qiskit_env # Activates it

pip install qiskit==0.44.0 # Installs your version

- **Need older Python?**

Some Qiskit versions require specific Python versions (e.g., Qiskit 0.20 needs Python ≤3.7). Specify during env creation:

bash

conda create --name old_qiskit_env python=3.7

3. Verify the Version

After installing, check it worked:

bash

python -c "import qiskit; print(qiskit.__version__)"

You should see your installed version (e.g., `0.44.0`).

How to Find Available Versions?

- Browse Qiskit releases on PyPI:

https://pypi.org/project/qiskit/#history

Example Workflow:

bash

Create environment for Qiskit 0.44.0 (if needed)

conda create --name qiskit_044 python=3.9

conda activate qiskit_044

Install specific version

pip install qiskit==0.44.0

Verify

python -c "import qiskit; print(qiskit.__version__)"

Output: 0.44.0

Why This Works:

- `pip install qiskit==x.y.z` forces installation of **exactly version `x.y.z`**.

- Using a dedicated `conda` environment avoids conflicts with other Python packages.

Index

About the Author

Samir is a trained physicist with a deep passion for physics, computer programming, quantum computing, and data science. He thrives on intellectual challenges and enjoys dissecting complex code, exploring supersymmetry theory, and investigating the origins of neutrino mass. You can reach him at slipovaca@aol.com.

Here are links to all notebooks. Please feel free to download them and explore.

Bipartite Graphs:

deepseekAndBipartite/DeepSeekAndBipartiteGraphs.ipynb at main · samlip-blip/deepseekAndBipartite

The Shortest Path in a Graph:

deepseekandgraphs/DeepSeekAndShortestPath.ipynb at main · samlip-blip/deepseekandgraphs

Palindrome Binary Numbers and Symmetric Graphs:

palindromes/Palindrome.ipynb at main · samlip-blip/palindromes

The SYK Model and Black Hole Information Scrambling:

syk/SYK6.ipynb at main · samlip-blip/syk

Random Graphs and Information Spreading Inside a Black Hole:

randomGraphsAndblackHoles/RandomGraphsAndBlackHoles.ipynb at main · samlip-blip/randomGraphsAndblackHoles

Spin Glass Model With A Randomly Generated Interaction Graph:

SpinGlass/SpinGlass.ipynb at main · samlip-blip/SpinGlass

Deutsch-Jozsa algorithm and Eulerian/half-Odd-half-Even Graphs:

DjAndEulerianGraphs/DjAndEulerianGraphs.ipynb at main · samlip-blip/DjAndEulerianGraphs

Bonus Chapter:

SeeSawNeutrino/NeutrionMass.ipynb at main · samlip-blip/SeeSawNeutrino