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.

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 oracle circuit:

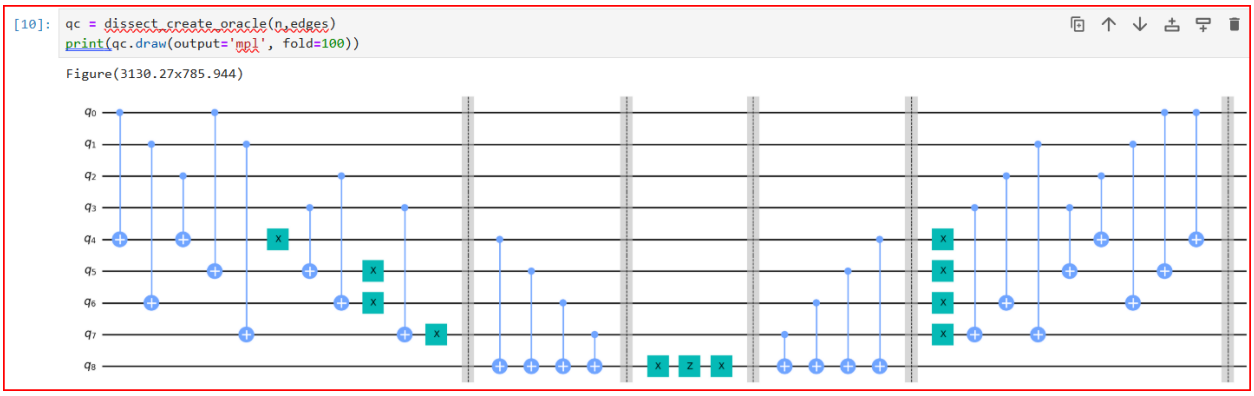


Figure 1. Grover's Oracle example for Bipartite Problem

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

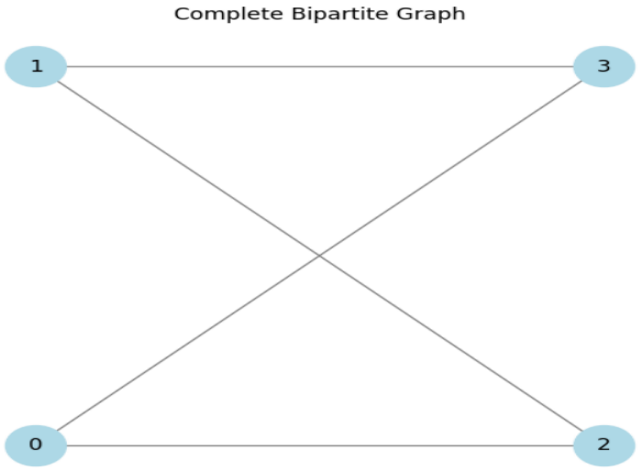


Figure 2. 4-nodes bipartite graph

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.*

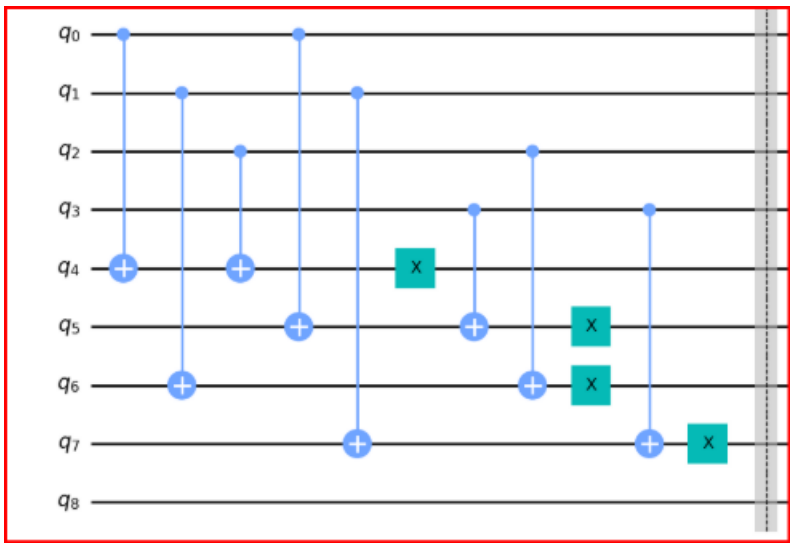
Imagine you have a main task to do, but you need an extra hand to help you complete it. The ancilla qubit is that extra hand. It's not the main focus of the computation, but it helps the main qubits (the ones doing the primary work) to perform their tasks more efficiently.

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.

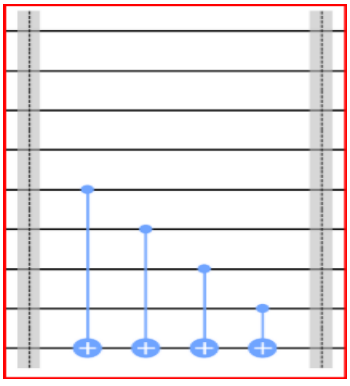
Next, the function checks each edge for same-color violation. For each edge (u,v) it marks edge ancilla if u and v have the same color (violation):

```
# 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)
```



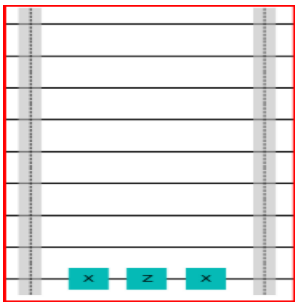
The flag qubit is set to 1 if any edge violates bipartition.

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



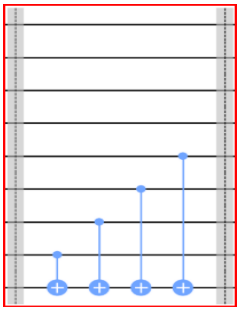
A phase flip (-1) is applied to the overall flag qubit only when there are no violations, meaning that the overall flag qubit is 0:

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



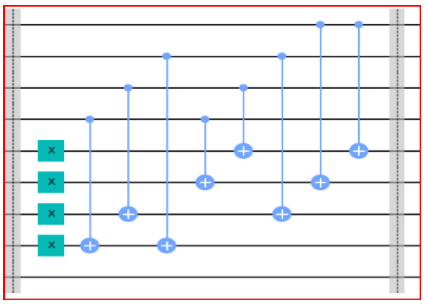
Finally, the function performs uncompute operations and returns the corresponding oracle circuit (qc).

```
# 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
```

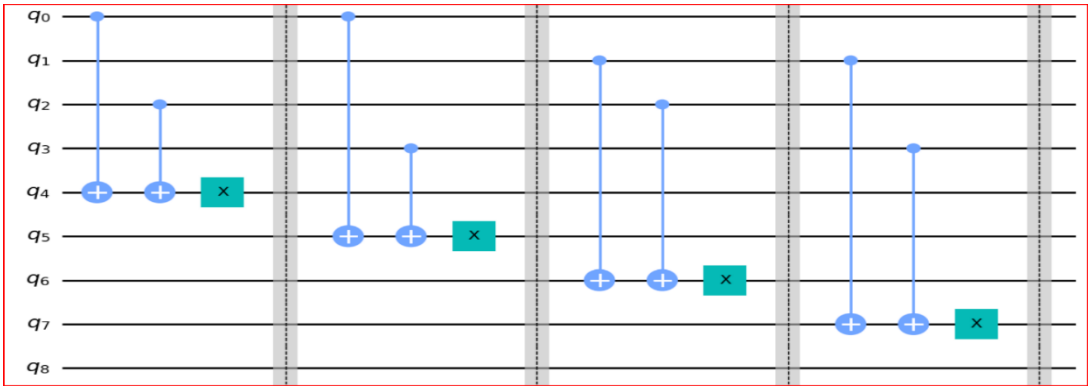


When we use ancilla qubits in a quantum circuit, **uncompute operations** play a crucial role in ensuring that the ancilla qubits are returned to their original state after they have assisted in the computation. This is important for several reasons:

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.



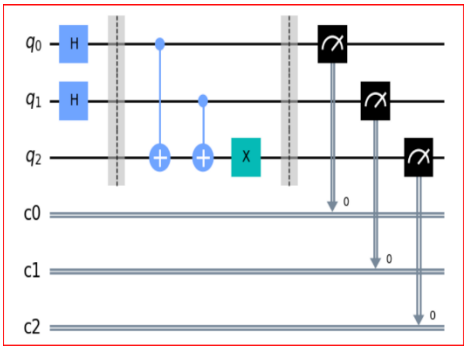
Using our graph from Figure 2 we see that

```
[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)
```

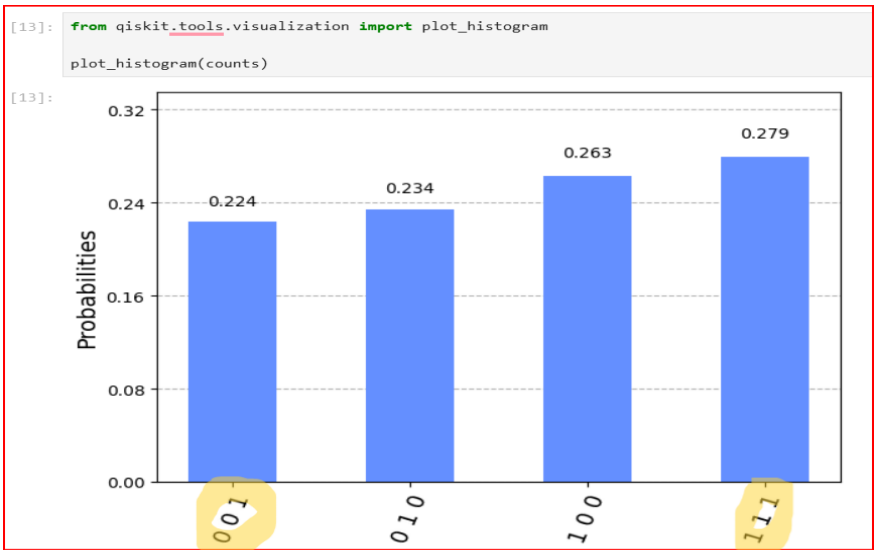
(0, 2) edge is encoded as CNOT(0,4) and CNOT(2,4). Then X gate as added to qubit 4. The same pattern is repeated for all other edges.

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

$$|q_0 q_1 q_2 \geq \frac{1}{\sqrt{2}}(|0> +|1>)\frac{1}{\sqrt{2}}(|0> +|1>)|0> =$$

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

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

$$|q_0 q_1 q_2>_{CNOT1} = \frac{1}{2}(|000> +|010> +|101> +|111>).$$

Then after the second CNOT we have

$$|q_0 q_1 q_2>_{CNOT1-CNOT2} = \frac{1}{2}(|000> +|011> +|101> +|110>).$$

The final state, after the X gate, is

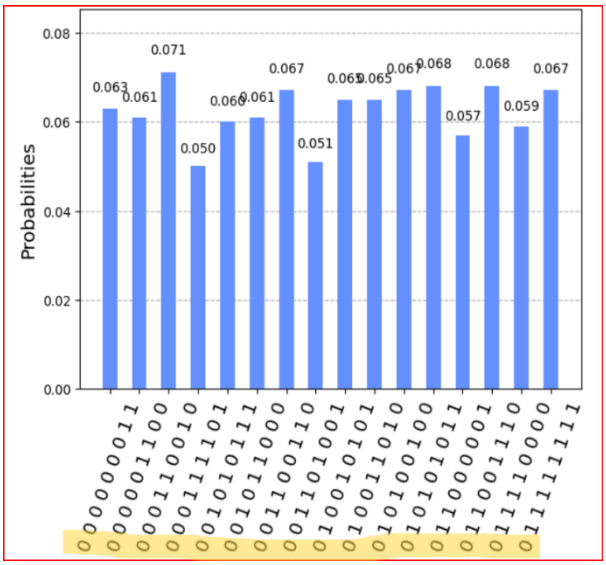
$$|q_0 q_1 q_2>_{CNOT1-CNOT2-X} = \frac{1}{2}(|001> +|010> +|100> +|111>).$$

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.

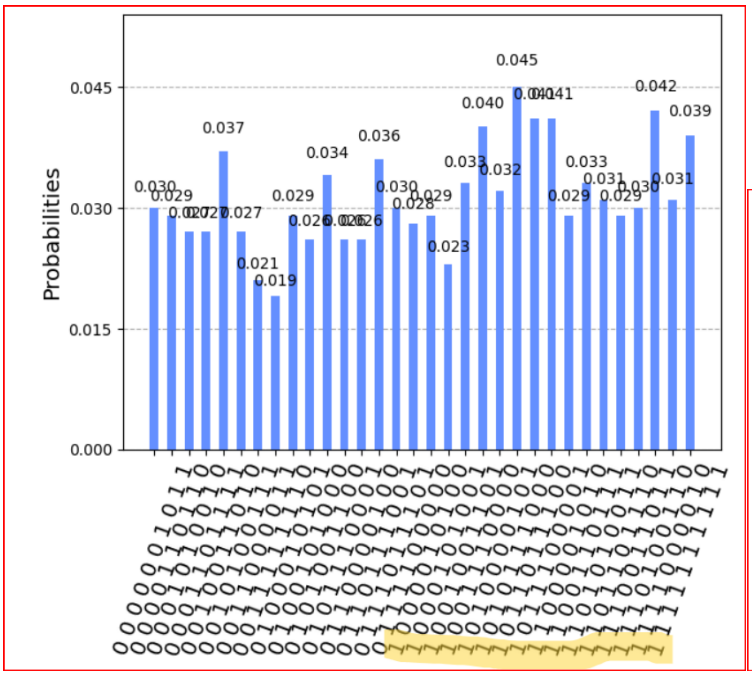
A	**B**	**A ⊙ B**
0	0	1
0	1	0
1	0	0
1	1	1

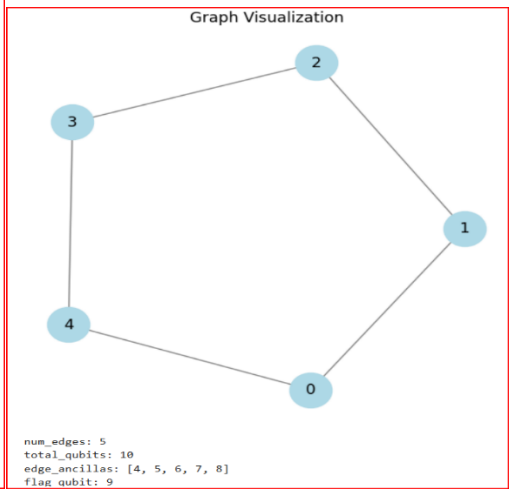
The XNOR gate outputs true (1) only when the inputs are the same. If the inputs are different, the output is false (0).

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.

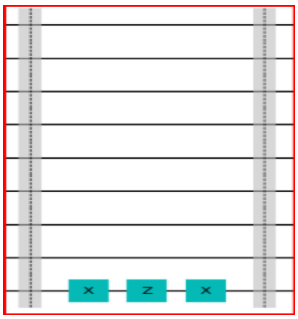


But, if a graph is not bipartite, the flag is set to 1. For example, this graph is not bipartite:





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.

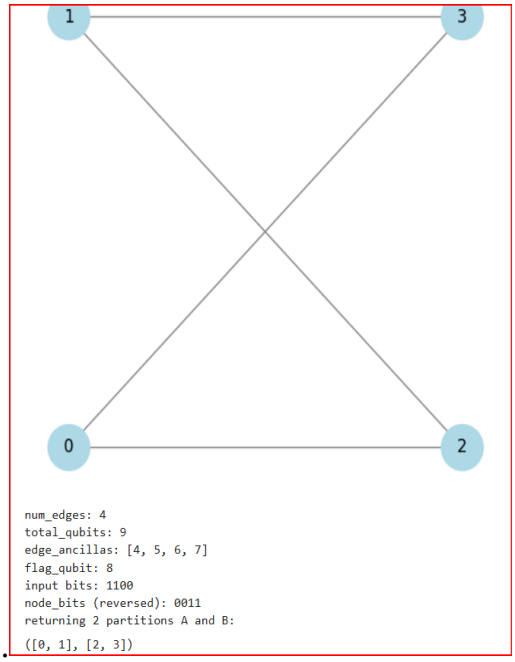
Let's now examine the remaining functions.

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.

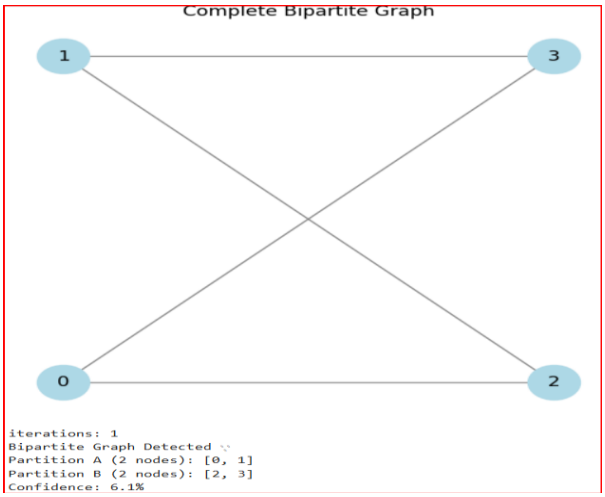
4. **Handle the case where the graph is not bipartite**:
   - `else:`: If there are no valid results, this conditional statement is executed.
   - `print("❌ Graph is not bipartite")`: This line prints a message indicating that the graph is not bipartite.

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.

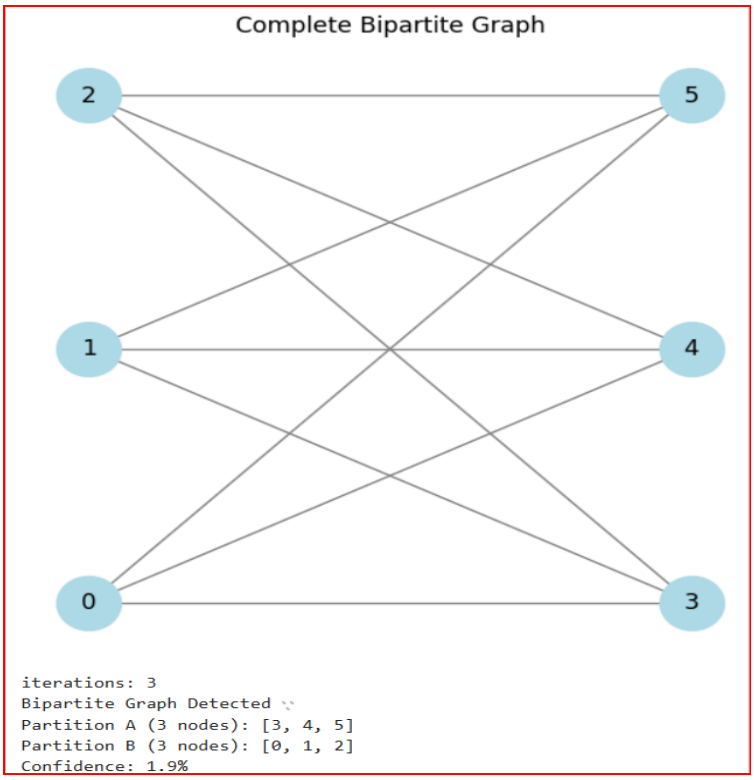
Testing

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

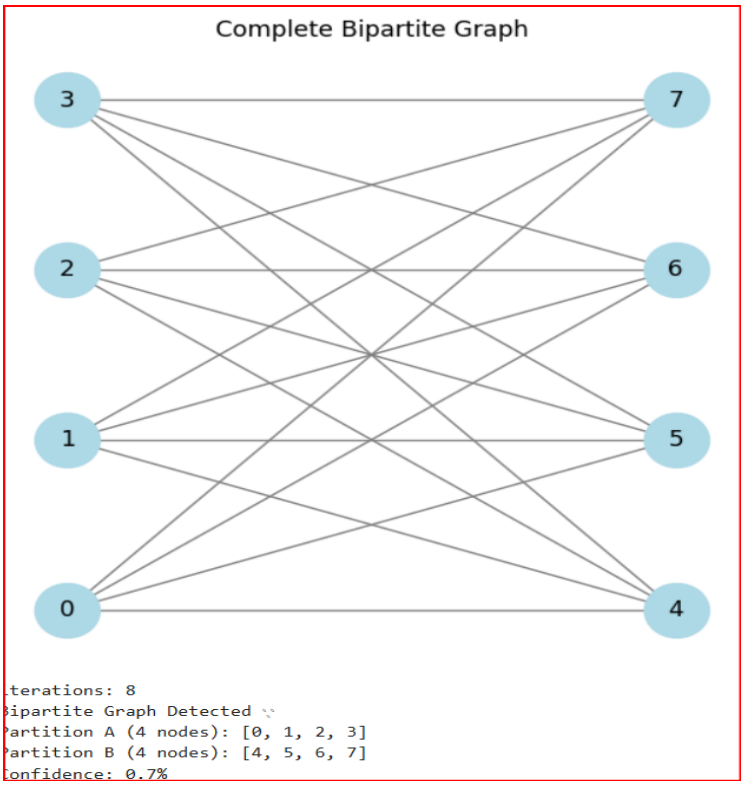
Test 1:


Complete Bipartite Graph
1
3
0
2
iterations: 1
Bipartite Graph Detected
Partition A (2 nodes): [0, 1]
Partition B (2 nodes): [2, 3]
Confidence: 6.1%

Test 2:

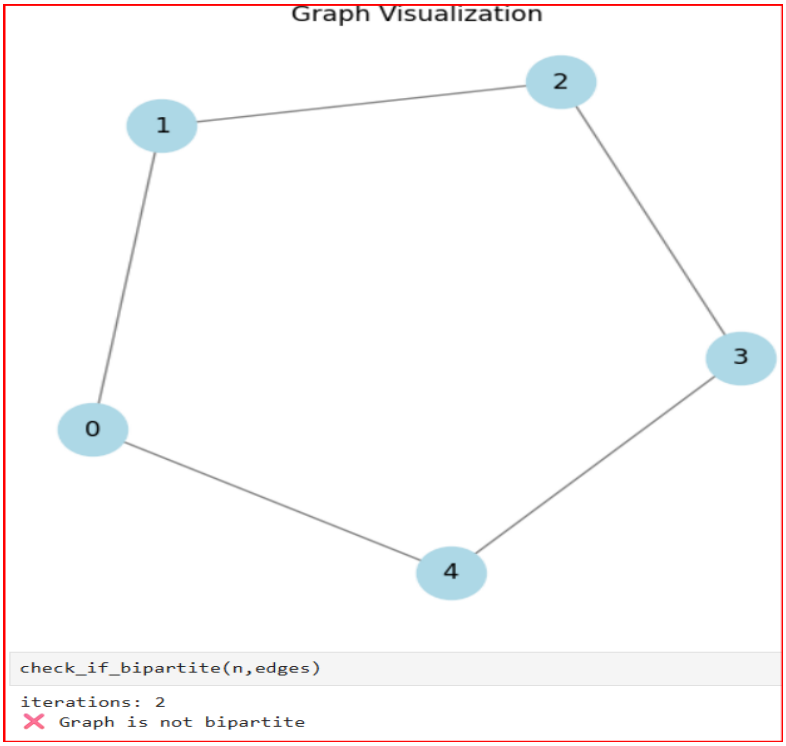

Complete Bipartite Graph
2
5
1
4
0
3
iterations: 3
Bipartite Graph Detected
Partition A (3 nodes): [3, 4, 5]
Partition B (3 nodes): [0, 1, 2]
Confidence: 1.9%

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.