Tutorial 9: Bipartite Graphs and Bug Gender Problem

1. Examples of Bipartite and Non-Bipartite Graphs

Bipartite Graphs:

- A graph with vertices {1, 2, 3, 4, 5, 6, 7} and edges {(1,4), (2,5), (3,6), (4,7), (5,1), (6,2), (7,3), (2,6), (3,7), (5,7)}.
- A graph representing a tree (any acyclic graph is bipartite).

Non-Bipartite Graphs:

- A graph with vertices {1, 2, 3, 4, 5, 6, 7} and edges {(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,1)} (contains an odd-length cycle of 7).
- A graph with at least 7 vertices and no triangle but containing an odd cycle, such as a cycle of length 5.

2. Modified Depth-First Search for Bipartiteness

We modify DFS to check if a graph is bipartite by coloring nodes alternately:

```
def is_bipartite(graph, start):
    color = {}
    stack = [(start, 1)]  # (node, color)

    while stack:
        node, c = stack.pop()
        if node in color:
            if color[node] != c:
                return False
        else:
            color[node] = c
            for neighbor in graph[node]:
                stack.append((neighbor, 3 - c))  # Alternate color
```

```
    return True, color
```

3. Proof and Runtime Analysis

Correctness: If the graph is bipartite, then every node can be colored alternately. If an odd-length cycle exists, then some node will conflict with its expected color, making it non-bipartite.

Time Complexity: Since we traverse all edges and vertices once, the runtime is $O(V + E)$, which is the complexity of DFS.

4. Proof of Bipartiteness Criterion

We prove that a graph is bipartite if and only if it has no odd cycles.

Forward Direction: If a graph is bipartite, then every cycle must alternate between two colors, implying an even cycle length.

Backward Direction: If no odd cycles exist, we can perform a BFS or DFS-based two-coloring scheme, ensuring bipartiteness.

5. Bug Gender Problem: Bipartiteness Approach

We model the problem as checking whether the interaction graph is bipartite.

Algorithm:

```
def bug_interaction(N, interactions):
    graph = {i: [] for i in range(1, N+1)}
    for a, b in interactions:
        graph[a].append(b)
        graph[b].append(a)

    visited = {}

    def dfs(node, c):
        if node in visited:
            return visited[node] == c
```

```
        visited[node] = c
        return all(dfs(neighbor, 1 - c) for neighbor in graph[node])

    for bug in graph:
        if bug not in visited:
            if not dfs(bug, 0):
                return "AMY IS WRONG"
    return "AMY MIGHT BE RIGHT"
```

Runtime Analysis: Since the algorithm performs a DFS over all nodes and edges, it runs in $O(N + K)$, where N is the number of bugs and K is the number of interactions.