CSC263H5 - Tutorial 9: Bipartite Graphs and Algorithms

1. Bipartite Graph Examples

- **Example of a bipartite graph with at least 7 vertices and 10 edges:** A graph with vertex set $V = \{1, 2, 3, 4, 5, 6, 7\}$ where edges form a bipartite structure such as:

$$E = \{(1, 4), (1, 5), (2, 6), (2, 7), (3, 5), (3, 6), (4, 7), (5, 7), (6, 4), (6, 7)\}$$

- **Example of a non-bipartite graph with at least 7 vertices and no triangle:** A graph that contains an odd cycle but avoids triangles, such as a 7-cycle:

$$E = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 1)\}$$

  This structure ensures there are no complete subgraphs of three vertices, but still contains an odd-length cycle, making it non-bipartite.

2. Modified Depth-First Search Algorithm for Bipartiteness

The following Depth-First Search (DFS) algorithm determines whether an undirected graph is bipartite and assigns each vertex to one of two partitions.

```
def is_bipartite(graph, n):
    side = [-1] * n  # -1 represents unvisited nodes

    def dfs(v, c):
        side[v] = c  # Assign side[v] as c (0 or 1)
        for neighbor in graph[v]:
            if side[neighbor] == -1:
                if not dfs(neighbor, 1 - c):  # Flip side assignment
                    return False
            elif side[neighbor] == side[v]:
                return False
        return True
```

```
13
14      for v in range(n):
15          if side[v] == -1:
16              if not dfs(v, 0):
17                  return False, side  # Graph is not bipartite
18      return True, side  # Graph is bipartite
```

Explanation:

- The algorithm initializes an array `side` where each vertex is marked as unvisited.
- DFS is used to traverse the graph, assigning each vertex a value of 0 or 1.
- If a neighboring vertex is already assigned and has the same value, the graph is not bipartite.

3. Proof and Complexity Analysis

Correctness: A graph is bipartite if and only if no two adjacent nodes have the same assigned value during DFS traversal. The algorithm ensures this by assigning each visited node the opposite value of its parent.

Time Complexity: DFS runs in $O(V + E)$ time because:

- Each vertex is visited once.
- Each edge is checked at most twice (once from each endpoint).

Thus, the worst-case complexity is $O(V + E)$, which is optimal for bipartiteness checking.

4. Proof: A Graph is Bipartite if and only if it has no Odd-Length Cycles

Proof Sketch:

- ($\Rightarrow$) If a graph is bipartite, it can be colored with two colors. Any cycle in the graph must alternate colors at each step, which implies the cycle length must be even.
- ($\Leftarrow$) If a graph contains an odd-length cycle, it cannot be two-colored without two adjacent vertices sharing the same color. Thus, it is not bipartite.

5. Bug Gender Problem Algorithm

Problem Interpretation: The problem reduces to checking whether the bug interaction graph is bipartite.

Algorithm:

```python
from collections import deque

def bug_gender_check(N, interactions):
    graph = [[] for _ in range(N)]
    for u, v in interactions:
        graph[u].append(v)
        graph[v].append(u)

    side = [-1] * N  # -1 means unvisited

    def bfs(start):
        queue = deque([start])
        side[start] = 0  # Assign first gender

        while queue:
            node = queue.popleft()
            for neighbor in graph[node]:
                if side[neighbor] == -1:
                    side[neighbor] = 1 - side[node]  # Assign
                        opposite gender
                    queue.append(neighbor)
                elif side[neighbor] == side[node]:
                    return False  # Conflict detected
        return True

    for i in range(N):
        if side[i] == -1:
            if not bfs(i):
                return "AMY IS WRONG"

```

```
30      return "AMY MIGHT BE RIGHT"
```

Complexity Analysis:

- The algorithm uses BFS to check for bipartiteness, similar to the previous approach.
- Each vertex is visited once, and each edge is checked at most twice.
- Time complexity: $O(N + K)$, where N is the number of bugs and K is the number of interactions.

Conclusion: The problem is effectively solved by checking for odd-length cycles in an undirected graph. If an odd-length cycle exists, Amy's assumption is incorrect; otherwise, it remains possible.