Bicoloring – Bipartite Checking

Bicoloring, also called **bipartite checking**, is the process of determining whether a graph can be colored using **two colors** such that no two adjacent vertices share the same color. A graph that satisfies this condition is called a **bipartite graph**. The most common method to check this is by using **BFS or DFS**. In BFS, we start from any node, assign it one color, and then assign the opposite color to all its neighbors. If we ever find a neighbor that has the same color as the current node, the graph is **not bipartite**. This method works for both connected and disconnected graphs.

Advantages

- Detects **bipartite graphs** efficiently.
- Can be implemented using **BFS or DFS** easily.
- Helps in solving problems like **matching, scheduling, and coloring**.
- Works for **both directed and undirected graphs**.

Limitations

- Only works for **two-color checking**; not general k-coloring.
- Graph must be **simple** (no self-loops).
- Does not directly give partitions; additional steps are needed to list two sets.

Steps for Bicoloring Using BFS

1. **Initialize colors**
   Assign all vertices a color of -1 (uncolored).
2. **Pick a starting vertex**
   Assign it color 0 and insert it into a queue.
3. **Perform BFS**
   While the queue is not empty:
    - Remove the current vertex.

    - For each neighbor:
        - If uncolored, assign the opposite color and add to the queue.
        - If already colored and the color is the same as current, the graph is **not bipartite**.

4. **Check disconnected graphs**
   Repeat the process for any unvisited vertex.

5. **Result**
   If no conflicts are found, the graph is bipartite.

Input:

3

3

0 1

1 2

2 0

Stepwise BFS Coloring

Initialization

• All nodes are uncolored: o Node 0 → uncolored o

Node 1 → uncolored o Node 2 → uncolored

• BFS queue is empty.

Step 1: Start BFS from node 0

• Assign color 0 to node 0.

• Add node 0 to the BFS queue.

Colors:

• Node 0 → 0

• Node 1 → uncolored

• Node 2 → uncolored

Queue: [0]

Step 2: Process node 0

• Neighbors of node 0: 1 and 2

• Both are uncolored → assign opposite color (1)

• Add neighbors to queue

Colors:

• Node 0 → 0 • Node 1 → 1

• Node 2 → 1

Queue: [1, 2]

Step 3: Process node 1

• Neighbors of node 1: 0 and 2

• Neighbor 0 → color 0 → fine

• Neighbor 2 → color 1 → conflict detected o Both node 1 and node 2 are connected and have the same color o Conflict means the graph cannot be bicolored

Step 4: Stop BFS

• BFS stops immediately because a conflict was found

• Graph contains an odd cycle (triangle) → impossible to 2-color

Output:

Not Bicolored.

Pseudocode for Bicoloring / Bipartite Check

```
BipartiteCheck(Graph):
   create color array, set all values to -1

   for each vertex v in Graph:
      if color[v] == -1:
         color[v] = 0
         create queue Q
         enqueue v

         while Q is not empty:
            u = dequeue Q
            for each neighbor n of u:
               if color[n] == -1:
                  color[n] = 1 - color[u]
                  enqueue n
               else if color[n] == color[u]:
                  return false   // Not bipartite

   return true   // Graph is bipartite
```

Time Complexity

- **O(V + E)**
  Where **V** = number of vertices and **E** = number of edges, because every vertex and edge is processed once.

Code:

https://github.com/shanto470/algorithm_plm/blob/main/BFS/Bicolor/bicolor.cpp