Bicoloring (BFS)

Bicoloring, or checking whether a graph is bipartite, can be done using a Breadth-First Search (BFS) approach. The idea is to try coloring the graph with two colors while traversing it. Since the graph is assumed to be connected, we can start the BFS from node 0. We maintain a color array where each node is initially uncolored, often represented by –1. We assign the starting node a color, say 0, and add it to a queue. During BFS, we repeatedly dequeue a node and inspect all its neighbors. If a neighbor is uncolored, we assign it the opposite color of the current node and enqueue it for further traversal. However, if a neighbor already has a color and it matches the current node's color, a conflict arises, indicating that two adjacent nodes would have the same color. This makes two-coloring impossible, meaning the graph is not bicolorable. If the BFS completes without encountering any such conflicts, then the graph can be successfully colored with two colors, confirming that it is bicolorable. This BFS-based technique is both straightforward and efficient, as it naturally detects odd-length cycles that prevent a graph from being two-colorable.

BFS Steps to Check Bicoloring

1. Initialize all nodes as uncolored

- Start by marking every node in the graph as uncolored (for example, using –1).
- This helps track which nodes have already been assigned a color during BFS.

2. Start BFS from the first node

- Choose any node as the starting point, usually node 0.
- Assign it a color, say color 0.
- Add this node to a queue, which keeps track of nodes to process.

3. Process nodes in the BFS queue

- While the queue is not empty, take the front node as the current node.
- Examine all neighbors (adjacent nodes) of this node.

4. Assign colors to neighbors

For each neighbor of the current node:

- **If the neighbor is uncolored:**

    - Assign it the opposite color of the current node.
    - Add it to the queue so its neighbors can be processed later.
- **If the neighbor is already colored:**
    - Compare its color with the current node.
    - If the neighbor has the same color, a conflict occurs.
        - This means two adjacent nodes would share the same color, which is invalid.
        - The graph is not bicolorable.
    - If the colors differ, continue BFS normally.

5. Continue until all reachable nodes are processed

- BFS explores nodes level by level, naturally alternating colors.
- Nodes at the same BFS level get the same color, and their neighbors get the opposite color.
- If no conflicts are found by the time the queue is empty, the graph is bicolorable.

6. Determine the result

- **No conflicts:** All nodes are successfully colored with two colors → the graph is bicolorable.
- **Conflict found:** Two adjacent nodes have the same color → the graph is not bicolorable.

Input:

3

3

0 1

1 2

2 0

Stepwise BFS Coloring

Initialization

- All nodes are uncolored:
    - Node 0 → uncolored
    - Node 1 → uncolored
    - 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 the 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**
    - Node 1 and Node 2 are connected and have the same color
    - Conflict means the graph cannot be bicolored

Step 4: Stop BFS

- BFS stops immediately due to conflict
- Graph contains an odd cycle (triangle) → impossible to 2-color

Output:

Not Bicolored

Pseudocode:

```
WHILE true:
  READ n
  IF n == 0:
    BREAK

  READ l  // number of edges

  // Initialize graph with n nodes
  graph = array of n empty lists

  // Read edges and build the graph
  FOR i FROM 1 TO l:
    READ a, b
```

```
    ADD b to graph[a]
    ADD a to graph[b]

// Initialize color array (-1 means uncolored)
color = array of size n filled with -1

// BFS initialization
CREATE empty queue q
color[0] = 0
ENQUEUE q with 0

isBipartite = true

// BFS traversal
WHILE q is not empty AND isBipartite is true:
    u = DEQUEUE q
    FOR each neighbor v of u:
        IF color[v] == -1:
            // Assign opposite color
            color[v] = 1 - color[u]
            ENQUEUE q with v
        ELSE IF color[v] == color[u]:
            // Conflict detected
            isBipartite = false
            BREAK
```

```
// Output result
IF isBipartite:
    PRINT "BICOLORABLE."
ELSE:
    PRINT "NOT BICOLORABLE."
```

Here is the Bicoloring solution code:

https://github.com/Hazra32/Algorithm-Problem/blob/main/BFS/Bicolor/bicolor.cpp