Leetcode May Challenge DAY: 27

[Python] Simple dfs traversal O(E+V), detailed explanations

This is classical graph theory problem: you have graph, and you need to check if it can be bipartitioned: all nodes must be divided into two groups, such that there is not connections between nodes in each group.

Classical way to solve this problem is any graph traversal algorithm, here I chose dfs.

The variable self.adj_list is adjacency list of our graph, for example if we have connections

[[1,2],[2,3],[3,4],[4,5],[1,5]], then it is equal to 1:{2,5}; 2:{1,3}, 3:{4,2}, 4:{3,5}, 5:{1,4}

I also have self.found_loop variable to terminate early if we found loop with odd length, which is sufficient and necessary condition that graph cannot be bipartitioned, for example in our case we have 1->2->3->4->5->1 the loop with size 5.

When we traverse nodes, we evaluate their distances (in my code list self.dist) from the starting point, and if in some moment node is already visited and its parity is broken, than we found odd loop.

Note also, that original graph is not necessarily connected, and we need to start our dfs from all nodes to make sure that we traverse all graph.

Complexity We traverse every connection between nodes only once, so we have classical O(E+V) time complexity, where E is number of edges and V is number of vertices (V = N in our case and V = len(dislikes)). Space complexity is also O(E+V), because we keep adjacency list for all our nodes and colours.

1. Python

```
class Solution:
    def dfs(self, start):
        if self.found_loop == 1: return       #early stop if we found odd cycle

        for neib in self.adj_list[start]:
            if self.dist[neib] > 0 and (self.dist[neib] - self.dist[start]) %2 == 0:
                self.found_loop = 1
            elif self.dist[neib] < 0:  #not visited yet
                self.dist[neib] = self.dist[start] + 1
                self.dfs(neib)

    def possibleBipartition(self, N, dislikes):
        self.adj_list = defaultdict(list)
        self.found_loop, self.dist = 0, [-1] *(N+1)

        for i,j in dislikes:
            self.adj_list[i].append(j)
            self.adj_list[j].append(i)

        for i in range(N):
            if self.found_loop: return False    #early stop if we found odd cycle

            if self.dist[i] == -1:    #not visited yet
                self.dist[i] = 0
                self.dfs(i)
        return True
```

2. C++

```
#define WHITE 0

#define RED 1

#define BLUE 2


class Solution

{

public:

   bool possibleBipartition(int N, vector<vector<int>> &edges)

   {

      vector<vector<int>> adj(N + 1); // adjacency list for undirected graph

      vector<int> color(N + 1, WHITE); // color of each vertex in graph, initially WHITE

      vector<bool> explored(N + 1, false); // to check if each vertex has been explored
exactly once


      // create adjacency list from given edges

      for (auto &edge: edges)

      {

         int u = edge[0];

         int v = edge[1];

         adj[u].push_back(v);

         adj[v].push_back(u);

      }


      // print adjacency list (comment out before submitting)

      for (int i = 0; i <= N; ++i)

      {

         cout << "adj[" << i << "]: ";

         for (int j = 0; j < adj[i].size(); ++j)
```

```
        {
            cout << adj[i][j] << " ";
        }
        cout << "\n";
    }

    // queue to perform BFS over each connected component in the graph
    // while performing BFS, we check if we encounter any conflicts while
    // coloring the vertices of the graph
    // conflicts indicate that bi-partition is not possible
    queue<int> q;

    for (int i = 1; i <= N; ++i)
    {
        if (!explored[i])
        {
            // this component has not been colored yet
                          // we color the first vertex RED and push it into the queue
            color[i] = RED;
            q.push(i);

            // perform BFS from vertex i
            while (!q.empty())
            {
                int u = q.front();
                q.pop();

                // check if u is already explored
                if (explored[u])
                {
                    continue;
                }

                explored[u] = true;

                // for each neighbor of u, execute this loop
                for (auto v: adj[u])
                {
                    // v is u's neighboring vertex

                    // checking if there's any conflict in coloring
                    if (color[v] == color[u])
                    {
                                                            // conflict edge found, so we return false
                                                            // as bi-partition will not be possible
                        return false;
                    }

                    // we color v with the opposite color of u
                    if (color[u] == RED)
                    {
                        color[v] = BLUE;
                    }
                    else
                    {
                        color[v] = RED;
                    }
```

```
                    q.push(v);
                }
            }
        }
    }

    // if no conflicts encountered then graph must be bipartite
    // so we return true

    return true;
  }
};
```

3. JAVA

```
class Solution {

    private static enum Colour {
        RED, BLUE;
    }

    public boolean possibleBipartition(int N, int[][] dislikes) {
        if (dislikes == null || dislikes.length == 0) return true;

        MyGraph g = new MyGraph(N+1);
        for (int[] d : dislikes) {
            g.addEdge(d[0], d[1]);
        }
        Colour[] colours = new Colour[N+1];
        for (int u = 1; u <= N; u++) {
            if (colours[u] == null) {
                if (!isBipartite(g, u, colours, Colour.RED))
                    return false;
            }
        }
        return true;
    }


    private static class MyGraph {
        int V;
        List<Integer>[] adj;
        public MyGraph(int V) {
            this.V = V;
            this.adj = (List<Integer>[])new ArrayList[V];
            for (int u = 0; u < V; u++) {
                adj[u] = new ArrayList<>();
            }
        }
        public void addEdge(int u, int v) {
            this.adj[u].add(v);
            this.adj[v].add(u);
        }

    }

    public boolean isBipartite(MyGraph g, int u, Colour[] colours, Colour colour) {
        colours[u] = colour;
        Colour nextColour = (colour == Colour.RED) ? Colour.BLUE : Colour.RED;
        for (int v : g.adj[u]) {
            if (colours[v] == null) {
                if (!isBipartite(g, v, colours, nextColour))
                    return false;
            }
            else if (colours[v] == colour)
                return false;
        }
        return true;
    }


}
```