Bài tập Brute-force

Nhóm 13:

21520012-Lê Chí Cường
21520239-Đoàn Nguyễn Trần Hoàn

BT. A graph is said to be bipartite if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. (one can also say that a graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors; such graphs are also called 2-colorable.)

1. Design a DFS-based algorithm for checking whether a graph is bipartite

2. Design a BFS-based algorithm for checking whether a graph is bipartite

<u>Solution.</u>

1. Algorithm:

- Use a color[] array which stores 0 or 1 for every node which denotes opposite colors.
- Call the function DFS from any node.
- If the node u has not been visited previously, then assign !color[v] to color[u] and call DFS again to visit nodes connected to u.
- If at any point, color[u] is qual to color[v], then the node is not bipartite.
- Modify the DFS function such that it returns a Boolean value at the end.

```cpp
// graph is bipartite or not using DFS
#include <bits/stdc++.h>
using namespace std;
// function to store the connected nodes
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}
// function to check whether a graph is bipartite or not
bool isBipartite(vector<int> adj[], int v,
                 vector<bool>& visited, vector<int>& color)
{

    for (int u : adj[v]) {

        // if vertex u is not explored before
        if (visited[u] == false) {

            // mark present vertices as visited
            visited[u] = true;

            // mark its color opposite to its parent
            color[u] = !color[v];

            // if the subtree rooted at vertex v is not bipartite
            if (!isBipartite(adj, u, visited, color))
                return false;
        }

        // if two adjacent are colored with same color then
        // the graph is not bipartite
        else if (color[u] == color[v])
            return false;
    }
    return true;
}
```

```cpp
int main()
{
    // no of nodes
    int N = 6;
    // to maintain the adjacency list of graph
    vector<int> adj[N + 1];
    // to keep a check on whether
    // a node is discovered or not
    vector<bool> visited(N + 1);
    // to color the vertices
    // of graph with 2 color
    vector<int> color(N + 1);
    // adding edges to the graph
    addEdge(adj, 1, 2);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    addEdge(adj, 4, 5);
    addEdge(adj, 5, 6);
    addEdge(adj, 6, 1);
    // marking the source node as visited
    visited[1] = true;

    // marking the source node with a color
    color[1] = 0;

    // Function to check if the graph
    // is Bipartite or not
    if (isBipartite(adj, 1, visited, color)) {
        cout << "Graph is Bipartite";
    }
    else {
        cout << "Graph is not Bipartite";
    }

    return 0;
}
```

- Time complexity: O(N)

2. Algorithm:

- Assign RED color to the source vertex (putting into set U).
- Color all the neighbors with BLUE color (putting into set V).
- Color all neighbor’s neighbor with RED color (putting into set U).
- This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.
- While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

```
#include <bits/stdc++.h>
using namespace std;

bool isBipartite(int V, vector<int> adj[])
{
    // vector to store colour of vertex
    // assigning all to -1 i.e. uncoloured
    // colours are either 0 or 1
      // for understanding take 0 as red and 1 as blue
    vector<int> col(V, -1);

    // queue for BFS storing {vertex , colour}
    queue<pair<int, int> > q;

      //loop incase graph is not connected
    for (int i = 0; i < V; i++) {

      //if not coloured
        if (col[i] == -1) {

          //colouring with 0 i.e. red
            q.push({ i, 0 });
            col[i] = 0;
            while (!q.empty()) {
                pair<int, int> p = q.front();
                q.pop();

                  //current vertex
                int v = p.first;
                  //colour of current vertex
                int c = p.second;

                  //traversing vertexes connected to current vertex
                for (int j : adj[v]) {

                      //if already coloured with parent vertex color
                      //then bipartite graph is not possible
                    if (col[j] == c)
```

```cpp
        if (col[i] == -1) {

          //colouring with 0 i.e. red
            q.push({ i, 0 });
            col[i] = 0;
            while (!q.empty()) {
                pair<int, int> p = q.front();
                q.pop();

                  //current vertex
                int v = p.first;
                  //colour of current vertex
                int c = p.second;

                  //traversing vertexes connected to current vertex
                for (int j : adj[v]) {

                      //if already coloured with parent vertex color
                      //then bipartite graph is not possible
                    if (col[j] == c)
                        return 0;

                      //if uncoloured
                    if (col[j] == -1) {
                      //colouring with opposite color to that of parent
                        col[j] = (c) ? 0 : 1;
                        q.push({ j, col[j] });
                    }
                }
            }
        }
    }
    //if all vertexes are coloured such that
      //no two connected vertex have same colours
    return 1;
}
```

```cpp
int main()
{

    int V, E;
    V = 4 , E = 8;
      //adjacency list for storing graph
    vector<int> adj[V];
      adj[0] = {1,3};
      adj[1] = {0,2};
      adj[2] = {1,3};
      adj[3] = {0,2};


    bool ans = isBipartite(V, adj);
    //returns 1 if bipartite graph is possible
      if (ans)
        cout << "Yes\n";
    //returns 0 if bipartite graph is not possible
      else
        cout << "No\n";

    return 0;
}
```

- Time complexity: O(V+E)