Day 6

Done	☑
Topic	Bridges in Graph, Flood Fill Algorithm, Tarjan's Algorithm
Languages	Java
Difficulty	⭐⭐⭐
Date Started	@March 6, 2023 11:30 PM
Date Completed	@March 7, 2023 2:00 AM
Related to Progress (Days)	Your Progress

What I Learned Today

Flood Fill Algorithm and Its Implementation

Tarjan's Algorithm

Bridges in Graph – Using Tarjan's Algorithm of time in and low time

Key Concepts

Flood Fill Algorithm

This can be solved using either Recursion or BFS.

Method 1 (Using Recursion):
The idea is simple, we first replace the color of the current pixel, then recur for 4 surrounding points. The following is a detailed algorithm.

```
// A recursive function to replace
// previous color 'prevC' at  '(x, y)'
// and all surrounding pixels of (x, y)
// with new color 'newC' and
floodFill(screen[M][N], x, y, prevC, newC)
1) If x or y is outside the screen, then return.
2) If color of screen[x][y] is not same as prevC, then return
3) Recur for north, south, east and west.
    floodFillUtil(screen, x+1, y, prevC, newC);
    floodFillUtil(screen, x-1, y, prevC, newC);
    floodFillUtil(screen, x, y+1, prevC, newC);
    floodFillUtil(screen, x, y-1, prevC, newC);
```

Method 2 (Using the BFS approach):

Algorithm for BFS based approach :

1. *Create a queue of pairs.*
2. *Insert an initial index given in the queue.*
3. *Mark initial index as visited in vis[][] matrix.*

Quick Links

Tutorial

Documentation

Tutorial

Documentation

4. *Until the queue is not empty repeat step 4.1 to 4.6*
    - *Take the front element from the queue*
    - *Pop from the queue*
    - *Store current value/color at coordinate taken out from queue (precolor)*
    - *Update the value/color of the current index which is taken out from the queue*
    - *Check for all 4 direction i.e (x+1,y),(x-1,y),(x,y+1),(x,y-1) is valid or not and if valid then check that value at that coordinate should be equal to precolor and value of that coordinate in vis[][] is 0.*
    - *If all the above condition is true push the corresponding coordinate in queue and mark as 1 in vis[][]*
5. *Print the matrix.*

Bridges in Graph – Using Tarjan's Algorithm of time in and low time

Problem Statement: There are n servers numbered from 0 to n – 1 connected by undirected server-to-server connections forming a network where connections[i] = [ai, bi] represents a connection between servers ai and bi. Any server can reach other servers directly or indirectly through the network.

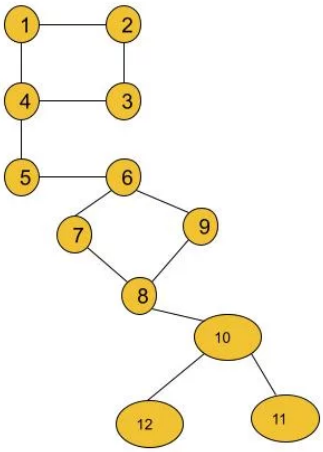
A critical connection is a connection that, if removed, will make some servers unable to reach some other servers.

Return all critical connections in the network in any order.

Note: Here servers mean the nodes of the graph.

Pre-requisite: DFS algorithm

Example :



```
Result: [[4, 5], [5, 6], [8, 10]]
Explanation: If we remove any of the three edges, the graph will be divided in
to 2 or more components.
```

If in this graph, we remove the edge (5,6), the component gets divided into 2 components. So, it is a bridge. But if we remove the edge (2,3) the component remains connected. So, this is not a bridge. In this graph, we have a total of 3 bridges i.e. (4,5), (5,6), and (10, 8).

In order to find all the bridges of a graph, we will implement some logic over the DFS algorithm. This is more of an algorithm-based approach. So, let's discuss the algorithm in detail. Before that, we will discuss two important concepts of the algorithm i.e. ***time of insertion and lowest time of insertion.***

- **Time of insertion:** Dring the DFS call, the time when a node is visited, is called its time of insertion. For example, if in the above graph, we start DFS from node 1 it will visit node 1 first then node 2, node 3, node 4, and so on. So, the time of insertion for node 1 will be 1, node 2 will be 2, node 3 will be 3 and it will continue like this. ***To store the time of insertion for each node, we will use a time array***.
- **Lowest time of insertion:** In this case, the current node refers to all its adjacent nodes ***except the parent*** and takes the minimum lowest time of insertion into account. To store this entity for each node, we will use another 'low' array.

The logical modification of the DFS algorithm is discussed below:

After the DFS for any adjacent node gets completed, we will just check if the edge, whose starting point is the current node and ending point is that adjacent node, is a bridge. For that, we will just check if any other path from the current node to the adjacent node exists if we remove that particular edge. If any other alternative path exists, this edge is not a bridge. Otherwise, it can be considered a valid bridge.

Approach:

The algorithm steps are as follows:

1. First, we need to create the adjacency list for the given graph from the edge information(***If not already given***). And we will declare a variable timer(either globally or we can carry it while calling DFS), that will keep track of the time of insertion for each node.
2. Then we will start DFS from node 0(assuming the graph contains a single component otherwise, we will call DFS for every component) with parent -1.
    a. Inside DFS, we will first mark the node visited and then store the time of insertion and the lowest time of insertion properly. The timer may be initialized to 0 or 1.
    b. Now, it's time to visit the adjacent nodes.
        i. ***If the adjacent node is the parent itself***, we will just continue to the next node.

ii. ***If the adjacent node is not visited***, we will call DFS for the adjacent node with the current node as the parent.After the DFS gets completed, we will compare the lowest time of insertion of the current node and the adjacent node and take the minimum one.Now, we will check if the lowest time of insertion of the adjacent node is greater than the time of insertion of the current node.If it is, then we will store the adjacent node and the current node in our answer array as they are representing the bridge.

iii. ***If the adjacent node is already visited***, we will just compare the lowest time of insertion of the current node and the adjacent node and take the minimum one.

3. Finally, our answer array will store all the bridges.

Note: *We are not considering the parent's insertion time during calculating the lowest insertion time as we want to check if any other path from the node to the parent exists excluding the edge we intend to remove.*

Time Complexity: O(V+2E), where V = no. of vertices, E = no. of edges. It is because the algorithm is just a simple DFS traversal.

Space Complexity: O(V+2E) + O(3V), where V = no. of vertices, E = no. of edges. O(V+2E) to store the graph in an adjacency list and O(3V) for the three arrays i.e. tin, low, and vis, each of size V.

Code Snippets

```
{
    public static void helper(int[][] image, int sr, int sc, int color, boolean vis[][], int orgColor) {
        if (sr < 0 || sc < 0 || sr >= image.length || sc >= image[0].length || vis[sr][sc] || image[sr][sc] != orgColor) {
            return;
        }
        image[sr][sc] = color;
        //Left
        helper(image, sr, sc - 1, color, vis, orgColor);
        //Right
        helper(image, sr, sc + 1, color, vis, orgColor);
        //Up
        helper(image, sr - 1, sc, color, vis, orgColor);
        //Down
        helper(image, sr + 1, sc, color, vis, orgColor);

    }

    public static void floodFill(int[][] image, int sr, int sc, int color) {
        boolean[][] visited = new boolean[image.length][image[0].length];
        helper(image, sr, sc, color, visited, image[sr][sc]);
    }

    public static void main(String[] args) {
        int[][] image = {{1, 1, 1},
                {1, 1, 0},
                {1, 0, 1}};
        floodFill(image,1,1,2);
        for (int i=0;i<image.length;i++){
            for (int j=0;j<image[0].length;j++){
                System.out.print(image[i][j]+" ");
            }
            System.out.println();
        }
    }
}
```

```
import java.io.*;
import java.util.*;

class Solution {
    private int timer = 1;
    private void dfs(int node, int parent, int[] vis,
                     ArrayList<ArrayList<Integer>> adj, int tin[], int low[],
                     List<List<Integer>> bridges) {
        vis[node] = 1;
        tin[node] = low[node] = timer;
        timer++;
        for (Integer it : adj.get(node)) {
            if (it == parent) continue;
            if (vis[it] == 0) {
                dfs(it, node, vis, adj, tin, low, bridges);
                low[node] = Math.min(low[node], low[it]);
                // node --- it
                if (low[it] > tin[node]) {
                    bridges.add(Arrays.asList(it, node));
                }
            } else {
                low[node] = Math.min(low[node], low[it]);
            }
        }
    }
    public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {
        ArrayList<ArrayList<Integer>> adj =
            new ArrayList<ArrayList<Integer>>();
        for (int i = 0; i < n; i++) {
            adj.add(new ArrayList<Integer>());
        }
        for (List<Integer> it : connections) {
            int u = it.get(0); int v = it.get(1);
            adj.get(u).add(v);
            adj.get(v).add(u);
        }
        int[] vis = new int[n];
        int[] tin = new int[n];
        int[] low = new int[n];
        List<List<Integer>> bridges = new ArrayList<>();
        dfs(0, -1, vis, adj, tin, low, bridges);
        return bridges;
    }
}

class Main {
    public static void main (String[] args) {
        int n = 4;
        int[][] edges = {
            {0, 1}, {1, 2},
            {2, 0}, {1, 3}
        };
        List<List<Integer>> connections = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            connections.add(new ArrayList<Integer>());
        }
        for (int i = 0; i < n; i++) {
            connections.get(i).add(edges[i][0]);
            connections.get(i).add(edges[i][1]);
        }

        Solution obj = new Solution();
        List<List<Integer>> bridges = obj.criticalConnections(n, connections);

        int size = bridges.size();
        for (int i = 0; i < size; i++) {
            int u = bridges.get(i).get(0);
            int v = bridges.get(i).get(1);
            System.out.print("[" + u + ", " + v + "] ");
        }
        System.out.println("");
    }
}
```

Challenges Experienced

Finding Time of insertion and Lowest time of insertion

Resources Used

YouTube, TakeUForward, Geeksforgeeks, ChatGPT