October 14, 2022 ▪ Data Structure / Graph

Find Eventual Safe States – DFS: G-20

Problem Statement: A directed graph of V vertices and E edges is given in the form of an adjacency list adj. Each node of the graph is labeled with a distinct integer in the range 0 to V – 1. A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node. You have to return an array containing all the safe nodes of the graph. The answer should be sorted in ascending order.

Example 1:

```
Input Format: N = 7, E = 7
```

Search

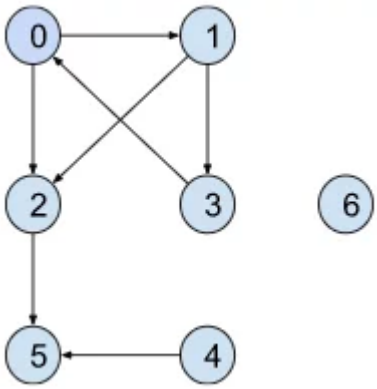
Search

Recent Posts

2023 – Striver's SDE Sheet Challenge

Top Array Interview Questions – Structured Path with Video Solutions

Longest Subarray with sum K | [Postives and Negatives]



Result: {2 4 5 6}

Explanation: Here terminal nodes ar

From node 0: two paths are there 0-
The second path does not end at a t
a safe node.
From node 1: two paths exist: 1->3-
But the first one does not end at a

From node 2: only one path: 2->5 an
So it is a ***safe node***.

From node 3: two paths: 3->0->1->3
but the first path does not end at
So it is not a ***safe node***.

From node 4: Only one path: 4->5 an
So it is also a ***safe node***.

From node 5: It is a terminal node.
So it is a ***safe node*** as well.

From node 6: It is a terminal node.
So it is a ***safe node*** as well.

Count Subarray sum Equals K

Binary Tree Representation in Java

Accolite Digital Amazon Arcesium arrays Bank of America Barclays BFS Binary Search Binary Search Tree Commvault CPP DE Shaw DFS DSA Self Paced google HackerEarth Hashing infosys inorder Interview Experience Java Juspay Kreeti Technologies Morgan Stanley Newfold Digital Oracle post order recursion Samsung SDE Core Sheet SDE Sheet Searching set-bits sorting Strivers A2ZDSA Course sub-array subarray Swiggy takeuforward TCS TCS CODEVITA TCS Ninja

Example 2:

```
Input Format: N = 4, E = 4
```

```
Result: {3}

Explanation: Node 3 itself is a ter
is a safe node as well. But all the
from other nodes do not lead to a t
excluded from the answer.
```

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Problem Link.

Solution:

A ***terminal node*** is a node without any outgoing edges(i.e outdegree = 0). Now a node is considered to be a ***safe node*** if all possible paths starting from it lead to a terminal node. Here we need to find out all safe nodes and return them sorted in ascending order.

If we closely observe, all possible paths starting from a node are going to end at some terminal node unless there exists a cycle and the paths return back to themselves. Let's understand it considering the below graph:

- In the above graph, there exists a cycle i.e 0->1->3->0, and node 7 is connected to the cycle with an incoming edge towards the cycle.
- Some paths starting from these nodes are definitely going to end somewhere in the

cycle and not at any terminal node. So, these nodes are not safe nodes.

- Though node 2 is connected to the cycle, the edge is directed outwards the cycle and all the paths starting from it lead to the terminal node 5. So, it is a safe node and the rest of the nodes (4, 5, 6) are safe nodes as well.

So, the intuition is to figure out the nodes which are either a part of a cycle or incoming to the cycle. We can do this easily using the cycle detection technique that was used previously to ***detect cycles in a directed graph (using DFS)***.

Note: *Points to remember that any node which is a part of a cycle or leads to the cycle through an incoming edge towards the cycle, cannot be a safe node. Apart from these types of nodes, every node is a safe node.*

Approach:

We will be solving it using DFS traversal. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes.

The algorithm steps are as follows:

1. We must traverse all components of the graph.
2. Make sure to carry two visited arrays in the DFS call. One is a visited array(vis) and the other is a path-visited(pathVis) array. The visited array keeps a track of visited nodes, and the path-visited keeps a track of visited nodes in the current traversal only.
3. Along with that, we will be carrying ***an extra array(check) to mark the safe nodes***.
4. While making a DFS call, at first we will mark the node as visited in both the arrays and then will traverse through its adjacent nodes. Now, there may be either of the three cases:

    1. **Case 1:** If the adjacent node is not visited, we will ***make a new DFS call recursively*** with that particular node.
    2. **Case 2:** If the adjacent node is visited and also on the same path(i.e marked visited in the pathVis array), we will ***return true,*** because it means it has a cycle, thereby the pathVis was true. Returning true will mean the end of the function call, as once we have got a cycle, there is no need to check for further adjacent nodes.
    3. **Case 3**: If the adjacent node is visited but not on the same path(i.e

not marked in the pathVis array), we will continue to the next adjacent node, as it would have been marked as visited in some other path, and not on the current one.

5. Finally, if there are no further nodes to visit, we will mark the node as safe in the check array and unmark the current node in the pathVis array and just return false. Then we will backtrack to the previous node with the returned value.
   The Point to remember is, while we enter we mark both the pathVis and vis as true, but at the end of traversal to all adjacent nodes, we just make sure to mark the current node safe and unmark the pathVis and still keep the vis marked as true, as it will avoid future extra traversal calls.

The following illustration will be useful in understanding the algorithm:

Let’s briefly understand the algorithm, using the above illustration.

- First, the DFS will start from node 0 and mark all the nodes visited and path-visited following the path: 0->1->2->3->5->6->7.
- As there are no further nodes after 7 to visit, it will backtrack to node 3 and meanwhile, it will unmark nodes 7, 6, and 4 from the pathVis array and also mark them as safe nodes.
- Now, from node 3, it again follows the path: 3->5->6->7. But after visiting node 5, it will find node 6 as visited but not path-visited and so it will not move further. And while returning it will unmark node 5 in the pathVis array and mark it a safe node as well.
- From node 3 it will backtrack to node 0 and unmark the nodes 3, 2, 1, and 0 in the

pathVis array and also mark them as safe nodes.

- Then, the DFS call will again start from node 8 following path 8->9->10. Now after reaching node 10, it finds node 8 previously visited as well as path-visited(i.e node 8 has been previously visited on the same path) and concludes that 8->9->10->8 is a cycle. So, nodes 8, 9, and 10 are not safe nodes.
- After that, the DFS call will start from node 11 which is incoming towards cycle 8->9->10->8. So it is not a safe node as well.
- Finally, the safe nodes will be 0, 1, 2, 3, 4, 5, 6, and 7 colored green in the illustration.

Code:

C++ Code

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

// User function Template for C++

class Solution {
private:
    bool dfsCheck(int node, vector<i
 int pathVis[],
        int check[]) {
        vis[node] = 1;
        pathVis[node] = 1;
        check[node] = 0;
        // traverse for adjacent nod
        for (auto it : adj[node]) {
```

```
                // when the node is not
                if (!vis[it]) {
                if (dfsCheck(it, adj, vi
                        check[node] = 0;
                        return true;
                    }

                }
                // if the node has been
                // but it has to be visi
                else if (pathVis[it]) {
                    check[node] = 0;
                    return true;
                }
            }
            check[node] = 1;
            pathVis[node] = 0;
            return false;
        }
    public:
        vector<int> eventualSafeNodes(in
            int vis[V] = {0};
            int pathVis[V] = {0};
            int check[V] = {0};
            vector<int> safeNodes;
            for (int i = 0; i < V; i++)
                if (!vis[i]) {
                    dfsCheck(i, adj, vis
                }
            }
            for (int i = 0; i < V; i++)
                if (check[i] == 1) safeN
            }
            return safeNodes;
        }
    };


    int main() {

        //V = 12;
        vector<int> adj[12] = {{1}, {2},
            {8},{9}};
        int V = 12;
        Solution obj;
        vector<int> safeNodes = obj.even
```

```
    for (auto node : safeNodes) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```

Output: 0 1 2 3 4 5 6 7

Time Complexity: O(V+E)+O(V), where V = no. of nodes and E = no. of edges. There can be at most V components. So, another O(V) time complexity.

Space Complexity: O(3N) + O(N) ~ O(3N): O(3N) for three arrays required during dfs calls and O(N) for recursive stack space.

Java Code

```
import java.util.*;

// User function Template for Java

class Solution {
    private boolean dfsCheck(int nod
    int pathVis[], int check[]) {
        vis[node] = 1;
        pathVis[node] = 1;
        check[node] = 0;
        // traverse for adjacent nod
        for (int it : adj.get(node))
            // when the node is not
            if (vis[it] == 0) {
                if (dfsCheck(it, adj
                    return true;
            }
            // if the node has been
            // but it has to be visi
            else if (pathVis[it] ==
```

```
                    return true;
                }
            }
            check[node] = 1;
            pathVis[node] = 0;
            return false;
        }

        List<Integer> eventualSafeNodes(
            int vis[] = new int[V];
            int pathVis[] = new int[V];
            int check[] = new int[V];
            for (int i = 0; i < V; i++)
                if (vis[i] == 0) {
                    dfsCheck(i, adj, vis
                }
            }
            List<Integer> safeNodes = ne
            for (int i = 0; i < V; i++)
                if (check[i] == 1)
                    safeNodes.add(i);
            }
            return safeNodes;
            // Your code here
        }
    }

    public class tUf {
        public static void main(String[]
            int V = 12;
            List<List<Integer>> adj = ne
            for (int i = 0; i < V; i++)
                adj.add(new ArrayList<>(
            }
            adj.get(0).add(1);
            adj.get(1).add(2);
            adj.get(2).add(3);
            adj.get(3).add(4);
            adj.get(3).add(5);
            adj.get(4).add(6);
            adj.get(5).add(6);
            adj.get(6).add(7);
            adj.get(8).add(1);
            adj.get(8).add(9);
            adj.get(9).add(10);
            adj.get(10).add(8);
```

```
        adj.get(11).add(9);

        Solution obj = new Solution(
        List<Integer> safeNodes = ob
        for (int node : safeNodes) {
            System.out.print(node +
        }
        System.out.println("");
    }
}
```

Output: 0 1 2 3 4 5 6 7

Time Complexity: O(V+E)+O(V), where V = no. of nodes and E = no. of edges. There can be at most V components. So, another O(V) time complexity.

Space Complexity: O(3N) + O(N) ~ O(3N): O(3N) for three arrays required during dfs calls and O(N) for recursive stack space.

> Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

« Previous Post
Detect cycle in a directed graph (using DFS) : G 19

Next Post »
Shortest Path in Directed Acyclic Graph Topological Sort: G-27

Load Comments