October 17, 2022 ▪ Data Structure / Graph

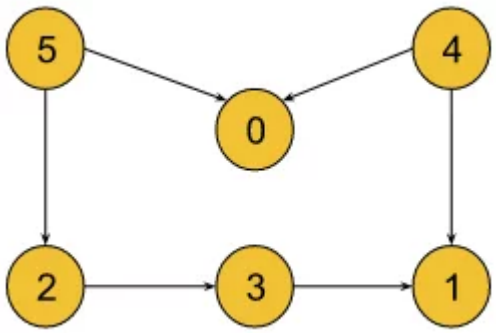
Topological Sort Algorithm | DFS: G-21

Problem Statement: Given a Directed Acyclic Graph (DAG) with V vertices and E edges, Find any Topological Sorting of that Graph.

Note: *In topological sorting, node u will always appear before node v if there is a directed edge from node u towards node v(u -> v).*

Example 1:

```
Input: V = 6, E = 6
```




```
Output: 5, 4, 2, 3, 1, 0
Explanation: A graph may have multi
```

```
The result is one of them. The neces
for the ordering are:

• According to edge 5 -> 0, node 5 
• According to edge 4 -> 0, node 4 
• According to edge 5 -> 2, node 5 
• According to edge 2 -> 3, node 2 
• According to edge 3 -> 1, node 3 
• According to edge 4 -> 1, node 4 


The above result satisfies all the 
[4, 5, 2, 3, 1, 0] is also one such
that satisfies all the conditions.
```

Example 2:

```
Input: V = 4, E = 3
```

```
Result: 3, 2, 1, 0
Explanation: The necessary conditio

• For edge 1 -> 0 node 1 must appea
• For edge 2 -> 0 node 1 must appea
• For edge 3 -> 0 node 1 must appea
```

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

```
[2, 3, 1, 0] is also another topolo
```


Solution

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

Problem link

Topological sorting only exists in Directed Acyclic Graph (DAG). If the nodes of a graph are connected through directed edges and the graph does not contain a cycle, it is called a **directed acyclic graph(DAG)**.

The ***topological sorting*** of a directed acyclic graph is nothing but the linear ordering of vertices such that if there is an edge between node u and v(u -> v), node u appears before v in that ordering.

Now, let's understand ***Why topological sort only exists in DAG:***

- **Case 1 (If the edges are undirected)**: If there is an undirected edge between node u and v, it signifies that there is an edge from node u to v(u -> v) as well as there is an edge from node v to u(v -> u). But according to the definition of topological sorting, it is practically impossible to write such ordering where ***u appears before v***

and ***v appears before u*** simultaneously. So, it is only possible for directed edges.

Case 2(If the directed graph contains a cycle): The following directed graph contains a cycle:

- If we try to get topological sorting of this cyclic graph, for edge 1->2, node 1 must appear before 2, for edge 2->3, node 2 must appear before 3, and for edge 3->1, node 3 must appear before 1 in the linear ordering. But such ordering is not possible as there exists a cyclic dependency in the graph. Thereby, topological sorting is only possible for a directed acyclic graph.

Approach:

We will be solving it using the DFS traversal technique. 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 a visited array(all elements are initialized to 0) and a stack data structure, where we are going to store the nodes after completing the DFS call.
3. In the DFS call, first, the current node is marked as visited. Then DFS call is made for all its adjacent nodes.
4. After visiting all its adjacent nodes, DFS will backtrack to the previous node and meanwhile, the current node is pushed into the stack.
5. Finally, we will get the stack containing one of the topological sortings of the graph.

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Let's quickly understand the algorithm considering the following graph:

- DFS will start from **node 0** and mark it as visited. But it has no adjacent nodes. So the DFS will return putting node 0 into the stack(stack: {0}).
- Then DFS will again start from **node 1** and mark it as visited, but it also has no adjacent nodes. So node 1 is pushed into the stack(stack: {1, 0}) and the DFS call will be over.
- Then DFS will start from **node 2** and mark it as visited as well. It will again call DFS for its adjacent node 3 and mark 3 as visited. After visiting node 3, it will find out that only adjacent node 1 is previously visited.
- So it will backtrack to node 2, putting node 3 first and then node 2 into the stack (stack: {2, 3, 1, 0}).
- Again, DFS will start from **node 4** and mark it as visited. It will find all its adjacent nodes 0 and 1 have been previously visited. So, node 4 will be pushed into the stack(stack: {4, 2, 3, 1, 0}).
- Lastly, DFS will start from **node 5** and mark it as visited. Again, it will find all its adjacent nodes 0 and 2 are previously visited. So, it will return putting node 5 into the stack(stack: {5, 4, 2, 3, 1, 0}).
- Finally, the stack will contain {5, 4, 2, 3, 1, 0}, which is one of the topological sortings of the graph.

Let’s understand how the linear orderings are maintained considering the following simple graph:

The linear ordering for the above graph can be 1, 2, 3, 4, 5, 6 or 1, 2, 3, 4, 6, 5. If we closely observe this algorithm, it is designed in such a way that when the DFS call for a node is completed, the node is always kept in the stack. So for example, if the DFS call for 3 is over, we must have the nodes 3, 4, 5, and 6 linearly ordered in the stack. And this is true for every other node. Thus the linear ordering is always maintained for each node of the graph.

Intuition:

Since we are inserting the nodes into the stack after the completion of the traversal, we are making sure, there will be no one who appears afterward but may come before in

the ordering as everyone during the traversal would have been inserted into the stack.

Note: *Points to remember, that node will be marked as visited immediately after making the DFS call and before returning from the DFS call, the node will be pushed into the stack.*

Code:

C++ Code

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

class Solution {
private:
    void dfs(int node, int vis[], st
             vector<int> adj[]) {
        vis[node] = 1;
        for (auto it : adj[node]) {
            if (!vis[it]) dfs(it, vi
        }
        st.push(node);
    }
public:
    //Function to return list contai
    vector<int> topoSort(int V, vect
    {
        int vis[V] = {0};
        stack<int> st;
        for (int i = 0; i < V; i++)
            if (!vis[i]) {
                dfs(i, vis, st, adj)
            }
        }

        vector<int> ans;
        while (!st.empty()) {
            ans.push_back(st.top());
            st.pop();
        }
        return ans;
```

```
    }
};


int main() {

    //V = 6;
    vector<int> adj[6] = {{}, {}, {3
    int V = 6;
    Solution obj;
    vector<int> ans = obj.topoSort(V

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

    return 0;
}
```

Output: 5 4 2 3 1 0

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(2N) + O(N) ~ O(2N): O(2N) for the visited array and the stack carried during DFS calls and O(N) for recursive stack space, where N = no. of nodes.

Java Code

```
import java.util.*;

class Solution {
    private static void dfs(int node
            ArrayList<ArrayList<Inte
        vis[node] = 1;
```

```
            for (int it : adj.get(node))
                if (vis[it] == 0)
                    dfs(it, vis, st, adj
            }
            st.push(node);
        }

        // Function to return list conta
        static int[] topoSort(int V, Arr
            int vis[] = new int[V];
            Stack<Integer> st = new Stac
            for (int i = 0; i < V; i++)
                if (vis[i] == 0) {
                    dfs(i, vis, st, adj)
                }
            }

            int ans[] = new int[V];
            int i = 0;
            while (!st.isEmpty()) {
                ans[i++] = st.peek();
                st.pop();
            }
            return ans;
        }
    }

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

            int[] ans = Solution.topoSor
            for (int node : ans) {
                System.out.print(node +
            }
            System.out.println("");
```

```
        }
    }
```

Output: 5 4 2 3 1 0

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(2N) + O(N) ~ O(2N): O(2N) for the visited array and the stack carried during DFS calls and O(N) for recursive stack space, where N = no. of nodes.

> 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
Kahn's Algorithm | Topological Sort Algorithm | BFS: G-22

Next Post »
Shortest Path in Undirected Graph with unit distance: G-28

Load Comments