November 9, 2022 ▪ Data Structure / Graph

Course Schedule I and II | Pre-requisite Tasks | Topological Sort: G-24

Problem Statement I (Course Schedule):

There are a total of n tasks you have to pick, labeled from 0 to n-1. Some tasks may have prerequisites tasks, for example, to pick task 0 you have to first finish tasks 1, which is expressed as a pair: [0, 1]

Given the total number of n tasks and a list of prerequisite pairs of size m. Find the order of tasks you should pick to finish all tasks.

Note: There may be multiple correct orders, you need to return one of them. If it is impossible to finish all tasks, return an empty array.

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]

Problem Statement II (Pre-requisite Tasks):
There are a total of N tasks, labeled from 0 to N-1. Some tasks may have prerequisites, for example, to do task 0 you have to first complete task 1, which is expressed as a pair: [0, 1]

Given the total number of tasks N and a list of prerequisite pairs P, find if it is possible to finish all tasks.

Note: *These two questions are linked. The second question asks if it is possible to finish all the tasks and the first question states to return the ordering of the tasks if it is possible to perform all the tasks, otherwise return an empty array.*

Examples:

```
Example 1:
Input: N = 4, P = 3,  array[] = {{1,0},{2,1},{3,2}}
Output: Yes
Explanation: It is possible to finish all the tasks in the order : 3 2 1 0.
First, we will finish task 3. Then we will finish task 2, task 1, and task 0.

Example 2:
Input: N = 4, P = 4,  array[] = {{1,2},{4,3},{2,4},{4,1}}
Output: No
Explanation: It is impossible to finish all the tasks. Let’s analyze the pairs:
```

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

```
For pair {1, 2} -> we need to
finish task 1 first and then
task 2. (order : 1 2).
For pair{4, 3} -> we need to
finish task 4 first and then
task 3. (order: 4 3).
For pair {2, 4} -> we need to
finish task 2 first and then
task 4. (order: 1 2 4 3).
But for pair {4, 1} -> we need
to finish task 4 first and then
task 1 but this pair contradicts
the previous pair. So, it is not
possible to finish all the
tasks.
```


Solution

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

Problem link 1.

Problem link 2.

Solution:

The solutions will be similar for both questions as we need to check for one, and in the other, we need to print the order. The questions state that the given pairs signify the dependencies of tasks. For example, *the pair* ***{u, v}*** signifies that to perform task v, first we need to finish task u. Now, if we closely observe, we can think of a directed edge between u and v(u -> v) where u and v are two nodes. Now, if we can think of each task as a node and every pair as a directed edge

between those two nodes, the whole problem becomes a ***graph problem***.

Now, let's analyze the examples from the graph point of view:

For example 1, the number of tasks(considered as nodes) is 4, and pairs(considered as edges) are 3. If we draw the graph accordingly, the following illustration is produced:

And For example 2, the number of tasks(considered as nodes) is 4, and pairs(considered as edges) are 4. If we draw the graph accordingly, the following illustration is produced:

Analyzing the graphs, we can conclude that performing all the tasks from the first set is possible because the first graph does not contain any cycle but as the second graph contains a cycle, performing all the tasks from the second set is impossible(there exists a cyclic dependency between the tasks). So, first, we need to identify a graph as a ***directed acyclic graph*** and if it is so we need to find out the linear ordering of the nodes as well(*second part for the question: Course schedule*).

Now, we have successfully reduced the problem to one of our known concepts: ***Detect cycle in a directed graph***. We will solve this problem using the ***Topological Sort Algorithm or Kahn's Algorithm***.

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.

For the second problem, we can also apply the algorithm used in the *detection of cycles in a directed graph (using DFS)* where we used to find out if the graph has a cycle or not. But to solve the first question we have to figure out the linear ordering of the task as well. So, we will use the topological sort algorithm for both questions.

Intuition:

For problem I, the intuition is to find the linear ordering in which the tasks will be performed if it is possible to perform all the tasks otherwise, to return an empty array.

For problem II, the intuition is to find if it is possible to perform all the tasks(i.e. The graph contains a cycle or not).

Approach:

We will apply the BFS(Breadth First Search) traversal technique. Breadth First Search or BFS is a traversal technique where we visit the nodes level-wise, i.e., it visits the same

level nodes simultaneously, and then moves to the next level.

Initial Configuration:

Indegree Array: Initially all elements are set to 0. Then, We will count the incoming edges for a node and store it in this array. For example, if the indegree of node 3 is 2, indegree[3] = 2.

Queue: As we will use BFS, a queue is required. Initially, the node with indegree 0 will be pushed into the queue.

Answer array: Initially empty and is used to store the linear ordering.

The algorithm steps are as follows:

1. First, we will form the adjacency list of the graph using the pairs. For example, for the pair {u, v} we will add node v as an adjacent node of u in the list.
2. Then, we will calculate the in-degree of each node and store it in the indegree array. We can iterate through the given adj list, and simply for every node u->v, we can increase the indegree of v by 1 in the indegree array.
3. Initially, there will be always at least a single node whose indegree is 0. So, we will push the node(s) with indegree 0 into the queue.

4. Then, we will pop a node from the queue including the node in our answer array, and for all its adjacent nodes, we will decrease the indegree of that node by one. For example, if node u that has been popped out from the queue has an edge towards node v(u->v), we will decrease indegree[v] by 1.
5. After that, if for any node the indegree becomes 0, we will push that node again into the queue.
6. We will repeat steps 3 and 4 until the queue is completely empty. Now, completing the BFS we will get the linear ordering of the nodes in the answer array.
7. **For the first problem(Course Schedule):** We will return the answer array if the length of the ordering equals the number of tasks. Otherwise, we will return an empty array.
   For the Second problem(Prerequisite tasks): We will return true if the length of the ordering equals the number of tasks. otherwise, we will return false.

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

Code (Course Schedule):

C++ Code

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

```
class Solution
{
public:
    vector<int> findOrder(int V, int
    {
        vector<int> adj[V];
        for (auto it : prerequisites
            adj[it[1]].push_back(it[
        }



        int indegree[V] = {0};
        for (int i = 0; i < V; i++)
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++)
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            // node is in your topo
            // so please remove it f

            for (auto it : adj[node]
                indegree[it]--;
                if (indegree[it] ==
            }
        }

        if (topo.size() == V) return
        return {};
    }
};
```

```
int main() {

    int N = 4;
    int M = 3;

    vector<vector<int>> prerequisite
    prerequisites[0].push_back(0);
    prerequisites[0].push_back(1);

    prerequisites[1].push_back(1);
    prerequisites[1].push_back(2);

    prerequisites[2].push_back(2);
    prerequisites[2].push_back(3);

    Solution obj;
    vector<int> ans = obj.findOrder(

    for (auto task : ans) {
        cout << task << " ";
    }
    cout << endl;
    return 0;
}
```

Output: 3 2 1 0

Java Code

```
import java.util.*;

class Solution {
    static int[] findOrder(int n, in
        // Form a graph
        ArrayList<ArrayList<Integer>
        for (int i = 0; i < n; i++)
            adj.add(new ArrayList<>(
        }

        for (int i = 0; i < m; i++)
            adj.get(prerequisites.ge
        }
```

```
            int indegree[] = new int[n];
            for (int i = 0; i < n; i++)
                for (int it : adj.get(i)
                    indegree[it]++;
                }
            }


            Queue<Integer> q = new Linke
            for (int i = 0; i < n; i++)
                if (indegree[i] == 0) {
                    q.add(i);
                }
            }


            int topo[] = new int[n];
            int ind = 0;
            // o(v + e)
            while (!q.isEmpty()) {
                int node = q.peek();

                q.remove();
                topo[ind++] = node;
                // node is in your topo
                // so please remove it f

                for (int it : adj.get(no
                    indegree[it]--;
                    if (indegree[it] ==
                }
            }


            if (ind == n) return topo;
            int[] arr = {};
            return arr;
        }
    }

    public class tUf {
        public static void main(String[]
            int N = 4;
            int M = 3;
            ArrayList<ArrayList<Integer>
```

```
        for (int i = 0; i < N; i++)
            prerequisites.add(i, new
        }


        prerequisites.get(0).add(0);
        prerequisites.get(0).add(1);

        prerequisites.get(1).add(1);
        prerequisites.get(1).add(2);

        prerequisites.get(2).add(2);
        prerequisites.get(2).add(3);

        int[] ans = Solution.findOrd

        for (int task : ans) {
            System.out.print(task +
        }
        System.out.println("");
    }
}
```

Output: 3 2 1 0

Code (Pre-requisite Tasks):

C++ Code

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

class Solution {
public:
    bool isPossible(int V, vector<pa
        vector<int> adj[V];
        for (auto it : prerequisites
            adj[it.first].push_back(
        }



        int indegree[V] = {0};
        for (int i = 0; i < V; i++)
            for (auto it : adj[i]) {
```

```
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++)
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            // node is in your topo
            // so please remove it f

            for (auto it : adj[node]
                indegree[it]--;
                if (indegree[it] ==
            }
        }

        if (topo.size() == V) return
        return false;


    }
};

int main() {

    vector<pair<int, int>> prerequis
    int N = 4;
    prerequisites.push_back({1, 0});
    prerequisites.push_back({2, 1});
    prerequisites.push_back({3, 2});

    Solution obj;
    bool ans = obj.isPossible(N, pre

    if (ans) cout << "YES";
    else cout << "NO";
    cout << endl;
```

```
        return 0;
    }
```

Output: YES

Time Complexity: O(V+E), where V = no. of nodes and E = no. of edges. This is a simple BFS algorithm.

Space Complexity: O(N) + O(N) ~ O(2N), O(N) for the indegree array, and O(N) for the queue data structure used in BFS(where N = no.of nodes). Extra O(N) for storing the topological sorting. Total ~ O(3N).

Java Code

```
import java.util.*;

class Solution {
    public boolean isPossible(int V,
        // Form a graph
        ArrayList<ArrayList<Integer>
        for (int i = 0; i < V; i++)
            adj.add(new ArrayList<>(
        }
        int m = prerequisites.length
        for (int i = 0; i < m; i++)
            adj.get(prerequisites[i]
        }



        int indegree[] = new int[V];
        for (int i = 0; i < V; i++)
            for (int it : adj.get(i)
                indegree[it]++;
            }
        }
```

```
            Queue<Integer> q = new Linke
            for (int i = 0; i < V; i++)
                if (indegree[i] == 0) {
                    q.add(i);
                }
            }


            List<Integer> topo = new Arr
            // o(v + e)
            while (!q.isEmpty()) {
                int node = q.peek();

                q.remove();
                topo.add(node);
                // node is in your topo
                // so please remove it f

                for (int it : adj.get(no
                    indegree[it]--;
                    if (indegree[it] ==
                }
            }


            if (topo.size() == V) return
            return false;

        }

    }

    public class tUf {
        public static void main(String[]
            int N = 4;
            int[][] prerequisites = new
            prerequisites[0][0] = 1;
            prerequisites[0][1] = 0;

            prerequisites[1][0] = 2;
            prerequisites[1][1] = 1;

            prerequisites[2][0] = 3;
            prerequisites[2][1] = 2;

            Solution obj = new Solution(
```

```
            boolean ans = obj.isPossible
            if (ans)
                System.out.println("YES"
            else
                System.out.println("NO")
        }
    }
```

Output: YES

Time Complexity: O(V+E), where V = no. of nodes and E = no. of edges. This is a simple BFS algorithm.

Space Complexity: O(N) + O(N) **~** O(2N), O(N) for the indegree array, and O(N) for the queue data structure used in BFS(where N = no.of nodes). Extra O(N) for storing the topological sorting. Total ~ O(3N).

> 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
Minimum cost to cut the stick| (DP-50)

Next Post »
Find Eventual Safe States – BFS – Topological Sort: G-25

Load Comments