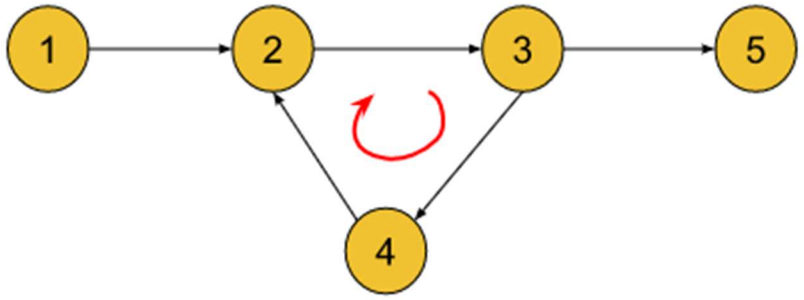
Detect a Cycle in Directed Graph | Topological Sort | Kahn's Algorithm | G-23

Problem Statement: Given a Directed Graph with V vertices and E edges, check whether it contains any cycle or not.

Example 1:

Input Format: V = 5, E = 5

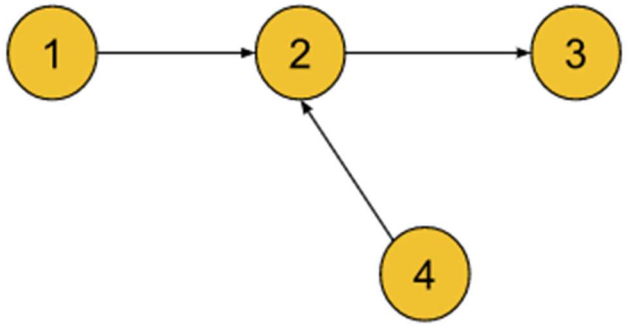


Result: True

Explanation: 2->3->4->2 is a cycle.

Example 2:

Input Format: V = 4, E = 3



Result: False

Explanation: There is no cycle in the graph.

This is a directed acyclic graph.

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

Problem link.

Solution:

Previously, we learned how to detect cycles in a directed graph using the DFS traversal technique. Now, we will use the BFS traversal for the same purpose.

While using DFS, the algorithm is designed with two visited arrays(vis and pathVis) in such a way that after making a DFS call the node is marked in both arrays, and while backtracking the node is unmarked from the path-visited (pathVis) array. And when we find a node marked in both arrays, we conclude that there exists a cycle. A point to note here is that we involve backtracking in the logic of DFS while finding the cycle, which is tough to replicate if we are trying to solve it iteratively.

Now, we intend to design such an algorithm using BFS which can check the cycle. To fulfill that purpose, we are going to use Kahn's Algorithm(Topological Sorting Using BFS).

Intuition:

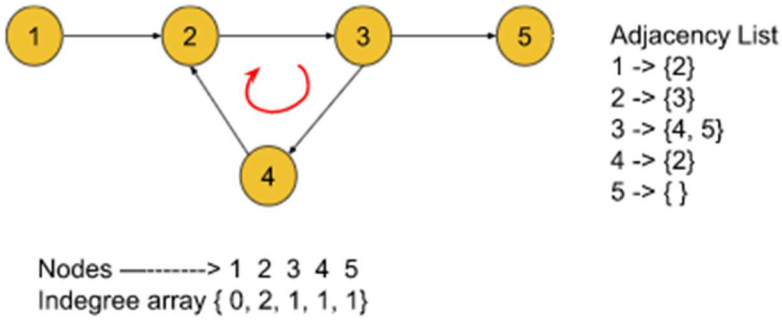
- Since we know topological sorting is only possible for ***directed acyclic graphs(DAGs)*** if we apply Kahn's algorithm in a directed cyclic graph(A directed graph that contains a cycle), it will fail to find the topological sorting(i.e. The final sorting will not contain all the nodes or vertices).
- So, finally, we will check the sorting to see if it contains all V vertices or not. If the result does not include all V vertices, we can conclude that there is a cycle.

Note: *The intuition is to check the size of the final topological sorting if it equals V(no. of vertices or nodes) or not.*

Let's quickly understand ***why Kahn's Algorithm will fail for a directed cyclic graph***.

If a directed graph contains a cycle, the indegree of the nodes that are parts of that cycle will never be 0 due to the cyclic dependency. But in that algorithm, we push a node into the queue only if its in-degree becomes 0. So, those nodes of the cycle will never be pushed into the queue as well as included in the topological sorting. And here it violates the rules of

topological sorting as topological sorting is a linear ordering of all V vertices (i.e. All V vertices must be present in that ordering). Consider the below graph:



Nodes ---------> 1 2 3 4 5
Indegree array { 0, 2, 1, 1, 1}

- First, we will put node 1 into the queue as its in-degree is 0.
- Now, after popping node 1 out of the queue, we will reduce the indegree[2] by 1. But in this step, there is no other node with in-degree 0. So, the algorithm is over. But the topological sorting only contains node 1.

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 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(Optional): Initially empty and is used to store the linear ordering.

The algorithm steps are as follows:

1. First, we will calculate the in-degree of each node and store it in the in-degree array. We can iterate through the given adj list, and simply for every node u->v, we can increase the in-degree of v by 1 in the in-degree array.

2. Initially, there will be always at least a single node whose indegree is 0. So, we will push the node(s) with in-degree 0 into the queue.
3. 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 in-degree 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.
4. After that, if for any node the in-degree becomes 0, we will push that node again into the queue.
5. 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.
6. Finally, we will check the length of the answer array. If it equals V(no. of nodes) then the algorithm will return false otherwise it will return true.

Space Optimization:

In this particular algorithm, we are only concerned about the length of the topological sorting and not the exact nodes it contains. So in step 3, after popping a node out of the queue, ***instead of putting it into an array we can carry a counter variable and increment it***. After completing the BFS this counter variable will give the length of the topological sorting. So, we need not use the extra answer.

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

Code:

- C++ Code
- Java Code

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


class Solution {
public:
        // Function to detect cycle in a directed graph.
        bool isCyclic(int V, vector<int> adj[]) {
                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);
                }
        }

        int cnt = 0;
        // o(v + e)
        while (!q.empty()) {
                int node = q.front();
                q.pop();
                cnt++;
                // node is in your topo sort
                // so please remove it from the indegree

                for (auto it : adj[node]) {
                        indegree[it]--;
                        if (indegree[it] == 0) q.push(it);
                }
        }

        if (cnt == V) return false;
        return true;
    }
};


int main() {

    //V = 6;
    vector<int> adj[6] = {{}, {2}, {3}, {4, 5}, {2}, {}};
    int V = 6;
    Solution obj;
    bool ans = obj.isCyclic(V, adj);
    if (ans) cout << "True";
    else cout << "Flase";
    cout << endl;
    return 0;
}
```

Output: True

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 in-degree array, and O(N) for the queue data structure used in BFS(where N = no.of nodes).