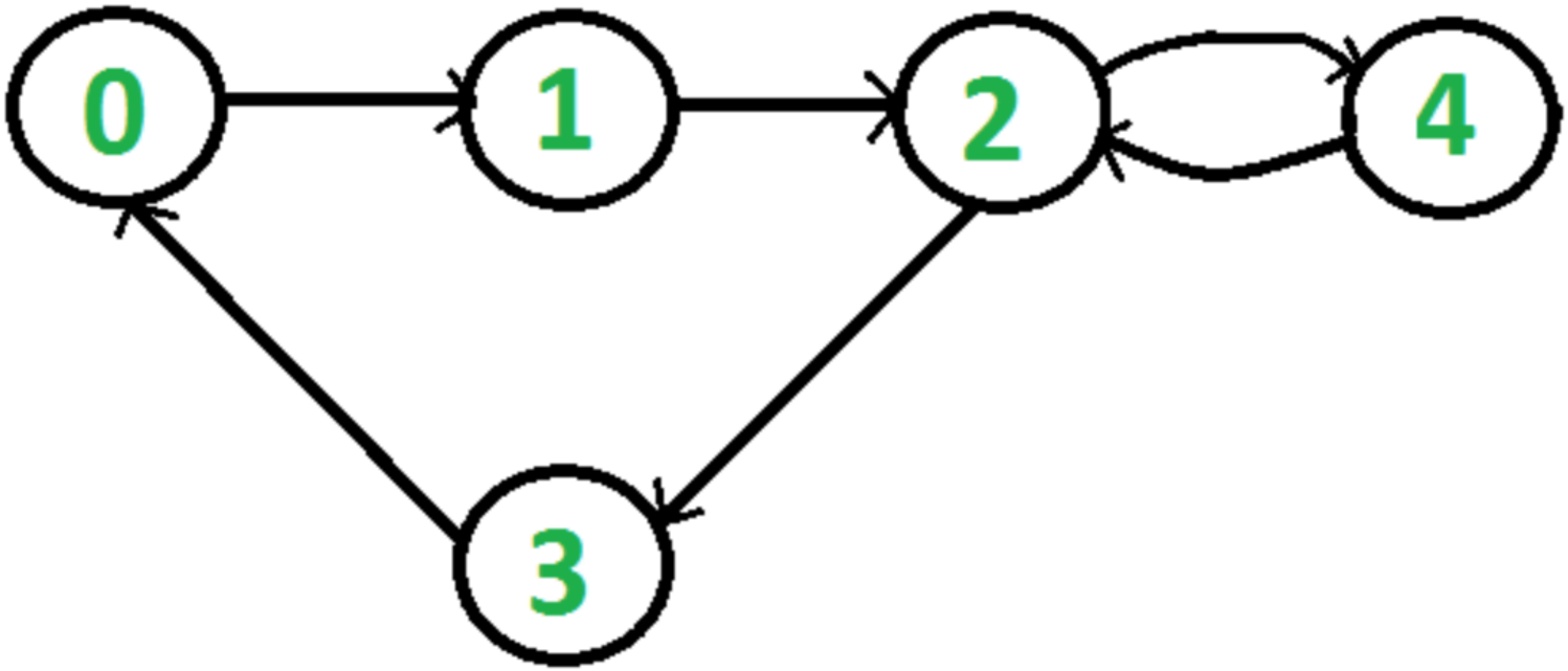
Dash
All
Articles
Videos
Problems
Quiz
Contest
Prev
Next

Detect Cycle in a Directed Graph (Part 2)

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. For example, the following graph contains two cycles 0->1->2->3->0 and 2->4->2, so your function must return true.



We have discussed a DFS based solution to detect cycle in a directed graph. In this post, BFS based solution is discussed.
The idea is to simply use Kahn's algorithm for Topological Sorting

Steps involved in detecting cycle in a directed graph using BFS.
Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the graph and initialize the count of visited nodes as 0.
Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)
Step-3: Remove a vertex from the queue (Dequeue operation) and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.

Step 4: Repeat Step 3 until the queue is empty.
Step 5: If count of visited nodes is **not** equal to the number of nodes in the graph has cycle, otherwise not.

How to find in-degree of each node?
There are 2 ways to calculate in-degree of every vertex:
Take an in-degree array which will keep track of
1) Traverse the array of edges and simply increase the counter of the destination node by 1.

```
for each node in Nodes
    indegree[node] = 0;
for each edge(src,dest) in Edges
    indegree[dest]++
```

Time Complexity: O(V+E)

2) Traverse the list for every node and then increment the in-degree of all the nodes connected to it by 1.

```
    for each node in Nodes
        If (list[node].size()!=0) then
```

```
for each dest in list
    indegree[dest]++;
```

Time Complexity: The outer for loop will be executed V number of times and the inner for loop will be executed E number of times, Thus overall time complexity is O(V+E).

The overall time complexity of the algorithm is O(V+E)


```java
// Java program to check if there is a cycle in
// directed graph using BFS.
import java.io.*;
import java.util.*;

class GFG
{

    // Class to represent a graph
    static class Graph
    {
        int V; // No. of vertices'

        // Pointer to an array containing adjacency list
        Vector<Integer>[] adj;

        @SuppressWarnings("unchecked")
        Graph(int V)
        {
            // Constructor
            this.V = V;
            this.adj = new Vector[V];
            for (int i = 0; i < V; i++)
                adj[i] = new Vector<>();
        }

        // function to add an edge to graph
```

```
void addEdge(int u, int v)
{
    adj[u].add(v);
}

// Returns true if there is a cycle in the graph
// else false.

// This function returns true if there is a cycle
// in directed graph, else returns false.
boolean isCycle()
{

    // Create a vector to store indegrees of all
    // vertices. Initialize all indegrees as 0.
    int[] in_degree = new int[this.V];
    Arrays.fill(in_degree, 0);

    // Traverse adjacency lists to fill indegrees of
    // vertices. This step takes O(V+E) time
    for (int u = 0; u < V; u++)
    {
        for (int v : adj[u])
            in_degree[v]++;
    }

    // Create an queue and enqueue all vertices with
    // indegree 0
    Queue<Integer> q = new LinkedList<Integer>();
    for (int i = 0; i < V; i++)
        if (in_degree[i] == 0)
            q.add(i);

    // Initialize count of visited vertices
    int cnt = 0;

    // Create a vector to store result (A topological
    // ordering of the vertices)
    Vector<Integer> top_order = new Vector<>();

    // One by one dequeue vertices from queue and enqueue
```

```
            // adjacents if indegree of adjacent becomes 0
            while (!q.isEmpty())
            {

                // Extract front of queue (or perform dequeue)
                // and add it to topological order
                int u = q.poll();
                top_order.add(u);

                // Iterate through all its neighbouring nodes
                // of dequeued node u and decrease their in-degree
                // by 1
                for (int itr : adj[u])
                    if (--in_degree[itr] == 0)
                        q.add(itr);
                cnt++;
            }

            // Check if there was a cycle
            if (cnt != this.V)
                return true;
            else
                return false;
        }
    }

    // Driver Code
    public static void main(String[] args)
    {

        // Create a graph given in the above diagram
        Graph g = new Graph(6);
        g.addEdge(0, 1);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(3, 4);
        g.addEdge(4, 5);

        if (g.isCycle())
            System.out.println("Yes");
        else
```

```
                System.out.println("No");
            }
        }

        // This code is contributed by
        // sanjeev2552
```


Output:

Yes

Time Complexity: O(V+E)
Auxiliary Space: O(V)