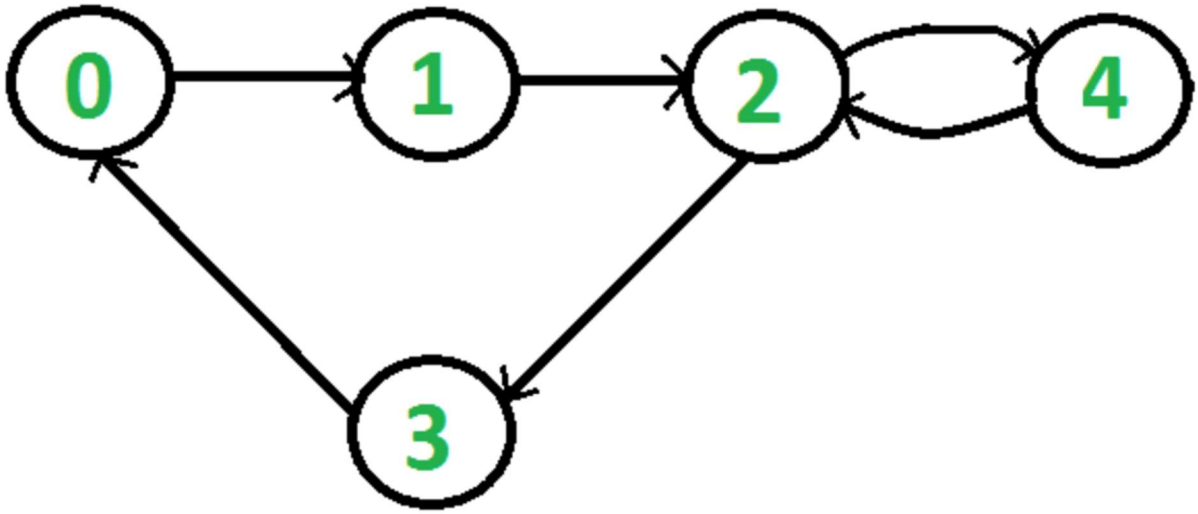
Detect Cycle in a Directed Graph using BFS

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)

C++

```cpp
// A C++ program to check if there is a cycle in
// directed graph using BFS.
#include <bits/stdc++.h>
using namespace std;


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

```
    // Pointer to an array containing adjacency list
    list<int>* adj;

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v);

    // Returns true if there is a cycle in the graph
    // else false.
    bool isCycle();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v);
}

// This function returns true if there is a cycle
```

```
// in directed graph, else returns false.
bool Graph::isCycle()
{
    // Create a vector to store indegrees of all
    // vertices. Initialize all indegrees as 0.
    vector<int> in_degree(V, 0);

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

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

    // Initialize count of visited vertices
    // 1 For src Node
    int cnt = 1;

    // Create a vector to store result (A topological
    // ordering of the vertices)
```

```
vector<int> top_order;

// One by one dequeue vertices from queue and enqueue
// adjacents if indegree of adjacent becomes 0
while (!q.empty()) {

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

    // Iterate through all its neighbouring nodes
    // of dequeued node u and decrease their in-degree
    // by 1
    list<int>::iterator itr;
    for (itr = adj[u].begin(); itr != adj[u].end(); itr++)

        // If in-degree becomes zero, add it to queue
        if (--in_degree[*itr] == 0)
        {
          q.push(*itr);
          //while we are pushing elements to the queue we will incrementing the cnt
          cnt++;
        }
```

```
    }

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

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(3, 4);
    g.addEdge(4, 5);

    if (g.isCycle())
        cout << "Yes";
    else
        cout << "No";

    return 0;
}
```

Output:

Yes

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