

[Striver's DSA
Sheets](#)[Striver's DSA
Playlists](#)[System
Design](#)[CS
Subjects](#)[Interview Prep
Sheets](#)[Striver's CP
Sheet](#)

August 27, 2022 ■ Arrays / Data Structure / Graph

Search

Search

Detect Cycle in an Undirected Graph (using BFS)

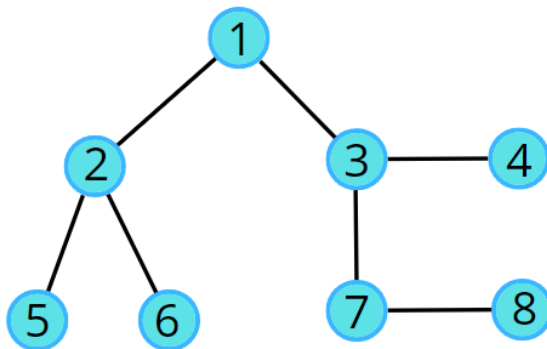
Problem Statement: Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

Examples:

Example 1:

Input:

$V = 8, E = 7$



Output: 0

Explanation: No cycle in the given graph.

Example 2:

Input:

$V = 8, E = 6$

Latest Video on
takeUforward



L2...



Latest Video on
Striver



I L...





Output: 1

Explanation: 4->5->6->4 is a cycle.

Solution

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

Intuition:

The cycle in a graph starts from a node and ends at the same node. So we can think of two algorithms to do this, in this article we will be reading about the BFS, and in the next, we will be learning how to use DFS to check.

Breadth First Search, 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.

The intuition is that we start from a node, and start doing BFS level-wise, if somewhere down the line, we visit a single node twice, it means we came via two paths to end up at the same node. It implies there is a cycle in the graph because we know that we start from different directions but can arrive at the same node only if the graph is connected or contains a cycle, otherwise we would never come to the same node again.

Approach:

Initial configuration:

Distance K in a Binary Tree

LCA in Binary Search Tree

Check if a tree is a Binary Search Tree or Binary Tree

Delete a Node in Binary Search Tree

Insert a Given Node in Binary Search Tree

from Rs5,280 from Rs17,832 fr

from Rs8,245 from Rs5,280 fr

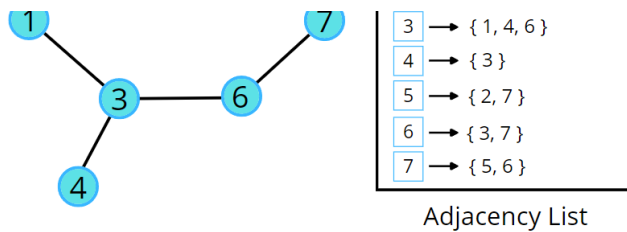
- **Visited array:** an array initialized to 0 indicating unvisited nodes.

The algorithm steps are as follows:

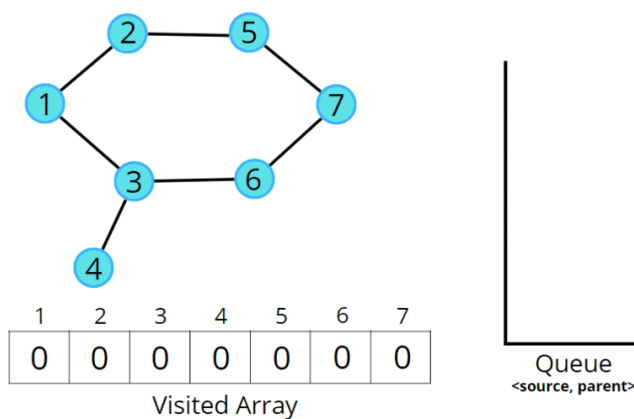
- For BFS traversal, we need a queue data structure and a visited array.
- Push the pair of the source node and its parent data (<source, parent>) in the queue, and mark the node as visited. The parent will be needed so that we don't do a backward traversal in the graph, we just move frontwards.
- Start the BFS traversal, pop out an element from the queue every time and travel to all its unvisited neighbors using an adjacency list.
- Repeat the steps either until the queue becomes empty, or a node appears to be already visited which is not the parent, even though we traveled in different directions during the traversal, indicating there is a cycle.
- If the queue becomes empty and no such node is found then there is no cycle in the graph.

A graph can have connected components as well. In such cases, if any component forms a cycle then the graph is said to have a cycle. We can follow the algorithm for the same:

```
// check for connected components in a graph
for ( i = 1; i <= n; i++)
{
    if(!vis[i])
    {
        if(detectCycle(i) == true)
            return true;
    }
}
return false;
```



Consider the following illustration to understand the process of detecting a cycle using BFS traversal.



Code:

C++ Code

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

class Solution {
private:
    bool detect(int src, vector<int> adj[], int vis[]) {
        vis[src] = 1;
        // store <source node, parent node>
        queue<pair<int, int>> q;
        q.push({src, -1});
        // traverse until queue is not empty
        while(!q.empty()) {
            int node = q.front().first;
            int parent = q.front().second;
            q.pop();

            // go to all adjacent nodes
            for(auto adjacentNode: adj[node]) {
                // if adjacent node is unvisited
```

```

        else if (parent != adjacentNode)
            // yes it is a cycle
            return true;
        }
    }
}
// there's no cycle
return false;
}
public:
    // Function to detect cycle in an undirected graph
    bool isCycle(int V, vector<int> adj[]) {
        // initialise them as unvisited
        int vis[V] = {0};
        for(int i = 0; i < V; i++) {
            if(!vis[i]) {
                if(detect(i, adj, vis)) return true;
            }
        }
        return false;
    }
};

int main() {
    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}

```

Output: 0

Time Complexity: $O(N + 2E) + O(N)$, Where N = Nodes, $2E$ is for total degrees as we traverse all adjacent nodes. In the case of connected components of a graph, it will take another $O(N)$ time.

Space Complexity: $O(N) + O(N) \sim O(N)$, Space for queue data structure and visited array.

```

class Solution {
    static boolean checkForCycle(ArrayList<ArrayList<Integer>> adj,
        boolean vis[], int parent[])
    {
        Queue<Node> q = new LinkedList<>();
        q.add(new Node(s, -1));
        vis[s] = true;

        // until the queue is empty
        while(!q.isEmpty())
        {
            // source node and its parent node
            int node = q.peek().first;
            int par = q.peek().second;
            q.remove();

            // go to all the adjacent nodes
            for(Integer it: adj.get(node))
            {
                if(vis[it]==false)
                {
                    q.add(new Node(it, node));
                    vis[it] = true;
                }

                // if adjacent node is visited
                // and parent is not equal to it, then cycle exists
                else if(par != it) return true;
            }
        }

        return false;
    }

    // function to detect cycle in an undirected graph
    public boolean isCycle(int V, ArrayList<ArrayList<Integer>> adj)
    {
        boolean vis[] = new boolean[V];
        Arrays.fill(vis, false);
        int parent[] = new int[V];
        Arrays.fill(parent, -1);

        for(int i=0; i<V; i++)
            if(vis[i]==false)
                if(checkForCycle(adj, i, vis, parent))
                    return true;

        return false;
    }
}

```

```

    for (int i = 0, i < 4, i++) {
        adj.add(new ArrayList < > ());
    }
    adj.get(1).add(2);
    adj.get(2).add(1);
    adj.get(2).add(3);
    adj.get(3).add(2);

    Solution obj = new Solution();
    boolean ans = obj.isCycle(4, adj);
    if (ans)
        System.out.println("1");
    else
        System.out.println("0");
}
}

class Node {
    int first;
    int second;
    public Node(int first, int second) {
        this.first = first;
        this.second = second;
    }
}
}

```

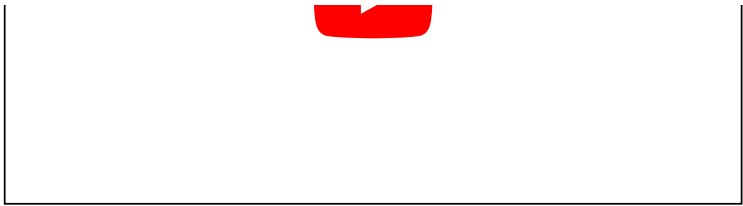
Output: 0

Time Complexity: $O(N + 2E) + O(N)$, Where N = Nodes, $2E$ is for total degrees as we traverse all adjacent nodes. In the case of connected components of a graph, it will take another $O(N)$ time.

Space Complexity: $O(N) + O(N) \sim O(N)$, Space for queue data structure and visited array.

Special thanks to [Vanshika Singh Gour](#) 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
Rotten Oranges

Next Post »
**Detect Cycle in an
Undirected Graph (using
DFS)**

Load Comments



The best place to learn data structures, algorithms, most asked coding interview questions, real interview experiences free of cost.

Follow Us



DSA Playlist

- Array Series
- Tree Series
- Graph Series
- DP Series

DSA Sheets

- Striver's SDE Sheet
- Striver's A2Z DSA Sheet
- SDE Core Sheet
- Striver's CP Sheet

Contribute

Write an Article

Copyright © 2023 takeufoward | All rights reserved

