August 27, 2022 ▪ Arrays / Data Structure / Graph

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
```

Search

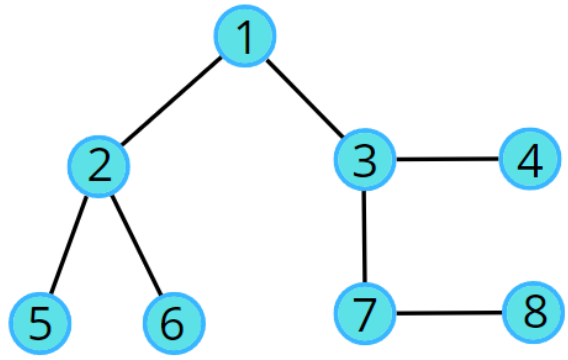
Search

Recent Posts

Top Array Interview Questions – Structured Path with Video Solutions

Longest Subarray with sum K | [Postives and Negatives]

Count Subarray sum Equals K



Output: 0

Explanation: No cycle in the given graph.

Example 2:
Input:
V = 8, E = 6

Output: 1

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

Solution

Binary Tree Representation in Java

Binary Tree Representation in C++

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

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

CODEVITA TCS Ninja

TCS NQT

VMware XOR

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:

- **Queue:** Define a queue and insert the source node along with parent data

(<source node, parent>). For example, (2, 1) means 2 is the source node and 1 is its parent node.

- **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:

Consider the following graph and its adjacency list.

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> a
      vis[src] = 1;
      // store <source node, parent
      queue<pair<int,int>> q;
      q.push({src, -1});
      // traverse until queue is not
      while(!q.empty()) {
          int node = q.front().first
          int parent = q.front().sec
          q.pop();

          // go to all adjacent node
          for(auto adjacentNode: adj
              // if adjacent node is
              if(!vis[adjacentNode])
                  vis[adjacentNode]
                  q.push({adjacentNo
              }
              // if adjacent node is
              else if(parent != adja
```

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

int main() {

    // V = 4, E = 2
    vector<int> adj[4] = {{}, {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) ~ O(N), Space for queue data structure and visited array.

Java Code

```
import java.util.*;

class Solution
{
   static boolean checkForCycle(Arra
            boolean vis[], int paren
    {
       Queue<Node> q =  new LinkedLi
       q.add(new Node(s, -1));
       vis[s] =true;

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

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

                // if adjacent node
               else if(par != it) re
           }
       }

       return false;
    }

    // function to detect cycle in a
    public boolean isCycle(int V, Ar
    {
        boolean vis[] = new boolean[
```

```
            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
                        return true;

            return false;
        }

        public static void main(String[]
        {
            ArrayList<ArrayList<Integer>
            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,
            if (ans)
                System.out.println("1");
            else
                System.out.println("0");
        }
    }

    class Node {
        int first;
        int second;
        public Node(int first, int secon
            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) ~ 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