JSS MAHAVIDYAPEETHA

JSS SCIENCE AND TECHNOLOGY UNIVERSITY

SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING

JSS Technical Institutions Campus, Mysuru – 570 006



A Project report on

# “Detect Cycle in an Undirected Graph using Queue”

PROGRAMMING WITH JAVA

20IS540

Bachelor of Engineering

in

Department of Information Science and Engineering

by,

1. Ramesh B S – 01JST20IS058
2. Gagan gowda L – 01JST21IS400
3. Suhas M S – 01JST20IS045

Submitted to, PROF. Prof. Shyla Raj

Assistant professor

Department of Information Science and Engineering

## CONTENTS

TABLE OF CONTENTS PAGE NO.

1. Introduction…………………………………………………………………………………………3
2. Examples……………………………………………………………………………………………3 – 4
3. Algorithm……………………………………………………………………………………………4 – 6
4. Code snippet…………………………………………………………………………………………6 – 9
5. Output……………………………………………..………………………………………………….9
6. Conclusion…………………………………………………………………………………………...10

# Detect Cycle in an Undirected Graph

### **INTRODUCTION**

A cycle in a graph is a path that starts and ends at the same vertex. In an undirected graph, a cycle is a path that consists of at least one edge and two vertices. Detecting cycles in a graph is an important problem in graph theory and is used in various applications such as in network design and optimization, scheduling, and testing graph planarity. There are several algorithms available for detecting cycles in an undirected graph, including depth-first search, breadth-first search, and union-find algorithms. In this report, we will discuss the basics of detecting cycles in an undirected graph and provide an overview of some of the most commonly used algorithms for solving this problem.

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, we will be learning about the BFS, and in the next, we will be learning how to use DFS to check.

Collection – **Queue**

In Java, a queue is a collection that is used to hold multiple elements prior to processing. It follows the First-In-First-Out (FIFO) principle, meaning that the first element added to the queue will be the first one to be removed.

The java.util.Queue interface provides the basic operations for a queue, including adding and removing elements, as well as checking the size of the queue. Some commonly used implementations of this interface include the LinkedList and PriorityQueue classes.

To add an element to a queue, the add() or offer() method can be used. To remove the first element of the queue, the remove() or poll() method can be used. To retrieve the first element without removing it, the element() or peek() method can be used.

**Detect cycle using BFS**

[**Breadth First Search**](https://takeuforward.org/graph/breadth-first-search-bfs-level-order-traversal/), 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.

The BFS algorithm for cycle detection in an undirected graph can be implemented using a queue to keep track of the vertices being visited. The algorithm starts by marking the starting vertex as visited and enqueuing it to the queue. Then, while the queue is not empty, the algorithm dequeues the front vertex, marks it as visited and enqueues all its unvisited neighbors. If a neighbor vertex is already visited and it is not the parent vertex of the current vertex, then a cycle is detected.

One of the advantages of using BFS for cycle detection is that it can also be used to find the shortest cycle in a graph, as the cycle is guaranteed to be found when the first vertex of the cycle is encountered.

**The algorithm steps are as follows:**

1. For BFS traversal, we need a queue data structure and a visited array.
2. 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.
3. Start the BFS traversal, pop out an element from the queue every time and travel to all its unvisited neighbors using an adjacency list.
4. 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.
5. If the queue becomes empty and no such node is found then there is no cycle in the graph.

### **JAVA code to detect cycle in undirected graph (Using BFS)**

import java.util.\*;

class Solution

{

static boolean checkForCycle(ArrayList<ArrayList<Integer>> adj, int s,

boolean vis[], int parent[])

{

Queue<Node> q = new LinkedList<>(); //BFS

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 is not its own parent node

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;

}

public static void main(String[] args)

{

ArrayList<ArrayList<Integer>> adj = new ArrayList<>();

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("Cycle detected");

else

System.out.println("No cycle");

}

}

class Node {

int first;

int second;

public Node(int first, int second) {

this.first = first;

this.second = second;

}

}

**Complexity**

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.

**Detect cycle using DFS**

The cycle in a graph starts from a node and ends at the same node. DFS is a traversal technique that involves the idea of recursion and backtracking. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes. The intuition is that we start from a source and go in-depth, and reach any node that has been previously visited in the past; it means there’s a cycle.

The DFS algorithm for cycle detection in an undirected graph can be implemented using a stack to keep track of the vertices being visited. The algorithm starts by marking the starting vertex as visited and pushing it to the stack. Then, while the stack is not empty, the algorithm pops the top vertex, marks it as visited and pushes all its unvisited neighbors. If a neighbor vertex is already visited and it is not the parent vertex of the current vertex, then a cycle is detected.

One of the advantages of using DFS for cycle detection is that it can be used to detect cycles in directed graphs as well as undirected graphs. Additionally, DFS can be used to detect cycles in disconnected graphs, whereas some other algorithms may require modifications to be used on disconnected graphs.

**The algorithm steps are as follows:**

1. For BFS traversal, we need a queue data structure and a visited array.
2. 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.
3. Start the BFS traversal, pop out an element from the queue every time and travel to all its unvisited neighbors using an adjacency list.
4. 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.
5. If the queue becomes empty and no such node is found then there is no cycle in the graph.

### JAVA code to detect cycle in undirected graph (Using DFS)

import java.util.\*;

class Solution

{

static boolean checkForCycle(ArrayList<ArrayList<Integer>> adj, int s,

boolean vis[], int parent[])

{

Queue<Node> q = new LinkedList<>(); //BFS

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 is not its own parent node

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;

}

public static void main(String[] args)

{

ArrayList<ArrayList<Integer>> adj = new ArrayList<>();

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("Cycle detected”);

else

System.out.println("No cycle");

}

}

class Node {

int first;

int second;

public Node(int first, int second) {

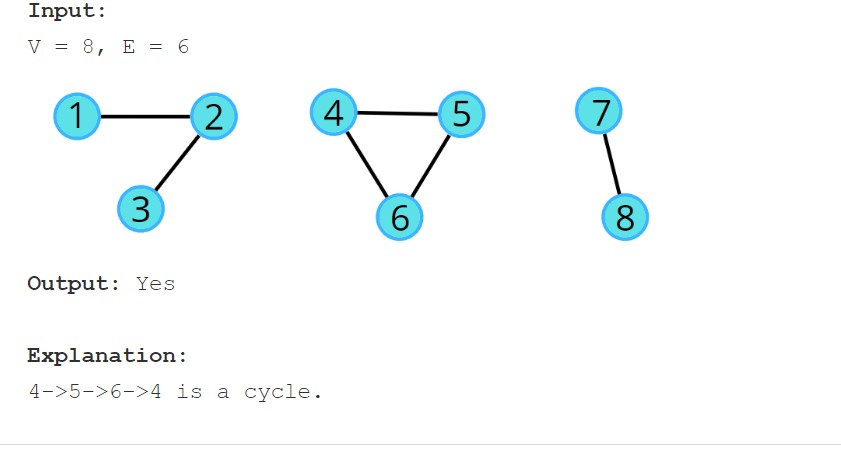
this.first = first;

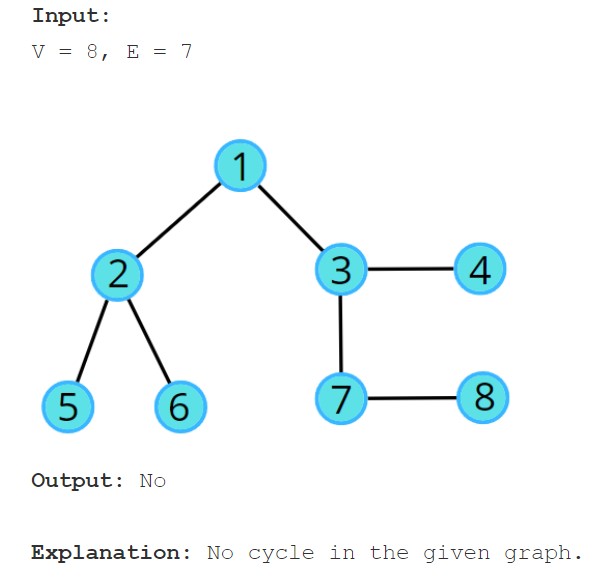
this.second = second;

}

}

**OUTPUT:**

****

****

**Complexity**

**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.

**CONCLUSION:**

To detect a cycle in an undirected graph using a queue, a breadth-first search (BFS) algorithm can be used. The basic idea is to start at a given vertex, and add all of its unvisited neighbors to a queue. Then, for each vertex in the queue, repeat the process of adding its unvisited neighbors to the queue. If a vertex is encountered that has already been visited, a cycle has been detected. To keep track of which vertices have been visited, a data structure such as a boolean array or a set can be used. This algorithm has a time complexity of O(V+E) where V is the number of vertices and E is the number of edges in the graph.

Additionally, during the BFS traversal, we can also keep track of the parent of each vertex. When a vertex is encountered that has already been visited and is not the parent of the current vertex, it means that a cycle has been detected. This is because if a vertex is the parent of the current vertex, it is not considered a cycle.

In summary, detecting a cycle in an undirected graph using a queue can be done by performing a breadth-first search (BFS) starting from a given vertex, adding all unvisited neighbors to a queue, and keeping track of the visited vertices using a boolean array or a set. If a vertex is encountered that has already been visited and is not the parent of the current vertex, it means that a cycle has been detected.