****

**Green University of Bangladesh**

**Department of Computer Science and Engineering (CSE)**

**Faculty of Sciences and Engineering**

**Semester: (Spring, Year:2025), B.Sc. in CSE (Day)**

**Lab Report NO: 01**

**Course Title: Algorithm lab**

**Course Code: CSE 206 Section: D9**

**Lab Experiment Name:** Detecting Cycles in a Graph using BFS and Performing Topological Sorting using DFS

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
| **1.** | Ashab Uddin | 232002274 |

**Lab Date :**

**Submission Date : 23-04-2025**

**Course Teacher’s Name : Farjana Akter Jui**

|  |
| --- |
| **Lab Report Status**  **Marks: ………………………………… Signature:.....................**  **Comments:.............................................. Date:..............................** |

**1. TITLE OF THE LAB REPORT EXPERIMENT**

This lab experiment involves implementing two graph traversal techniques:

1. Detecting Cycles in a Graph using BFS.

2. Performing Topological Sorting using DFS.

Both tasks focus on understanding the applications of graph algorithms and their respective implementations in Java.

**2. OBJECTIVES**

* Explore how graphs can be explored using BFS and DFS methods.
* Use BFS to check if there are any cycles in an undirected graph.
* Apply DFS to sort the nodes topologically in a directed acyclic graph.
* Get hands-on practice working with graphs using adjacency lists.
* Compare how BFS and DFS behave and understand when to use each one.

**3. PROCEDURE**

**Cycle Detection using BFS**

1. Model the graph with an adjacency list structure.
2. Perform a breadth-first traversal using a queue while marking visited nodes.
3. If you encounter a node that has already been visited and isn't the immediate parent, a cycle exists.
4. Show the cycle if one is identified.

**Topological Sort using DFS**

1. Construct the graph as an adjacency list.
2. Use a stack to maintain the order of nodes.
3. Carry out a recursive DFS, visiting each node and pushing it onto the stack once fully explored.
4. Display the contents of the stack as the final topological sequence.

**4. IMPLEMENTATION**

1. **Cycle Detection in a Graph using BFS**

import java.util.\*;

public class CycleDetectionBFS {

static class Edge {

int source, destination;

public Edge(int source, int destination) {

this.source = source;

this.destination = destination;

}

}

static void buildGraph(ArrayList<Edge>[] graph) {

for (int i = 0; i < graph.length; i++) {

graph[i] = new ArrayList<>();

}

graph[0].add(new Edge(0, 1));

graph[1].add(new Edge(1, 0));

graph[1].add(new Edge(1, 2));

graph[2].add(new Edge(2, 1));

graph[2].add(new Edge(2, 3));

graph[3].add(new Edge(3, 2));

graph[3].add(new Edge(3, 4));

graph[4].add(new Edge(4, 3));

graph[4].add(new Edge(4, 1));

graph[1].add(new Edge(1, 4));

}

static boolean hasCycleUsingBFS(ArrayList<Edge>[] graph, int totalVertices) {

boolean[] isVisited = new boolean[totalVertices];

int[] parentNode = new int[totalVertices];

Arrays.fill(parentNode, -1);

for (int currentVertex = 0; currentVertex < totalVertices; currentVertex++) {

if (!isVisited[currentVertex]) {

if (checkCycleBFS(graph, isVisited, parentNode, currentVertex)) {

return true;

}

}

}

return false;

}

static boolean checkCycleBFS(ArrayList<Edge>[] graph, boolean[] isVisited, int[] parentNode, int startingNode) {

Queue<Integer> bfsQueue = new LinkedList<>();

bfsQueue.add(startingNode);

isVisited[startingNode] = true;

while (!bfsQueue.isEmpty()) {

int currentNode = bfsQueue.poll();

for (Edge connection : graph[currentNode]) {

int neighborNode = connection.destination;

if (!isVisited[neighborNode]) {

isVisited[neighborNode] = true;

parentNode[neighborNode] = currentNode;

bfsQueue.add(neighborNode);

} else if (neighborNode != parentNode[currentNode]) {

System.out.print("Cycle detected: ");

showCyclePath(parentNode, currentNode, neighborNode);

return true;

}

}

}

return false;

}

static void showCyclePath(int[] parentNode, int currentNode, int repeatedNode) {

List<Integer> path = new ArrayList<>();

int tempNode = currentNode;

while (tempNode != -1) {

path.add(tempNode);

if (tempNode == repeatedNode) break;

tempNode = parentNode[tempNode];

}

Collections.reverse(path);

path.add(repeatedNode);

for (int i = 0; i < path.size(); i++) {

System.out.print(path.get(i));

if (i < path.size() - 1) System.out.print(" -> ");

}

System.out.println();

}

public static void main(String[] args) {

int totalVertices = 5;

ArrayList<Edge>[] graph = new ArrayList[totalVertices];

buildGraph(graph);

if (!hasCycleUsingBFS(graph, totalVertices)) {

System.out.println("No cycle found in the graph.");

}

}

}

2. **Topological Sorting using DFS**

import java.util.\*;

public class TopoSortDFS {

static class Graph {

private int v;

private ArrayList<Integer>[] adj;

public Graph(int v) {

this.v = v;

adj = new ArrayList[v];

for (int i = 0; i < v; i++) {

adj[i] = new ArrayList<>();

}

}

public void addEdge(int u, int v) {

adj[u].add(v);

}

private void dfs(int node, boolean[] vis, Stack<Integer> stk) {

vis[node] = true;

for (int nbr : adj[node]) {

if (!vis[nbr]) {

dfs(nbr, vis, stk);

}

}

stk.push(node);

}

public void topoSort() {

Stack<Integer> stk = new Stack<>();

boolean[] vis = new boolean[v];

for (int i = 0; i < v; i++) {

if (!vis[i]) {

dfs(i, vis, stk);

}

}

System.out.print("Topological Order: ");

while (!stk.isEmpty()) {

System.out.print(stk.pop() + " ");

}

System.out.println();

}

}

public static void main(String[] args) {

Graph g = new Graph(6);

g.addEdge(5, 2);

g.addEdge(5, 0);

g.addEdge(4, 0);

g.addEdge(4, 1);

g.addEdge(2, 3);

g.addEdge(3, 1);

System.out.println("Performing Topological Sort using DFS:");

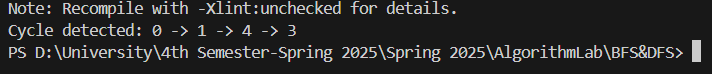
g.topoSort();

}

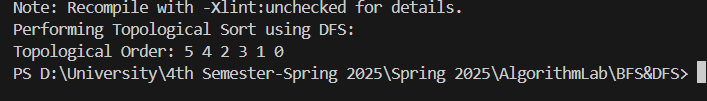
}

**5. TEST RESULT**

**Cycle Detection using BFS Output**



**Topological Sort using DFS Output**



**6. DISCUSSION**

**Cycle Detection using BFS:**

* This method helps find loops in undirected graphs.
* It checks each node and keeps track of which node it came from (the parent).
* If it visits a node that was already visited and it’s not the parent, a cycle is found.
* Time Complexity: O(V + E), where V = number of vertices and E = number of edges.

**Topological Sort using DFS:**

* This technique works only on Directed Acyclic Graphs (DAGs).
* It visits all connected nodes deeply before going back, then adds them to a stack.
* The final stack gives the topological order of the nodes.
* Time Complexity: O(V + E)

**Challenges Faced:**

* It was tricky to understand how to correctly follow and remember each node’s parent in BFS.
* Managing the stack correctly while using DFS to sort the nodes required careful attention.