M.Kumarasamy
College of Engineering
NAAC Accredited Autonomous Institution
Approved by AICTE & Affiliated to Anna University
ISO 9001:2015 Certified Institution
Thalavapalayam, Karur - 639 113, TAMILNADU.

A Project Report

on

DETECTING CYCLE IN A DIRECTED GRAPH

Submitted in partial fulfilment of requirements for the award of the course

of

CGB1122 – DATA STRUCTURES

Under the guidance of

Mrs. S.RAMYA.M.E., (Ph.D.,),

Assistant Professor/AI

Submitted By

KAVIYA R **(927624BAM025)**

DEPARTMENT OF FRESHMAN ENGINEERING

M.KUMARASAMY COLLEGE OF ENGINEERING
(Autonomous)

KARUR – 639 113

MAY 2025

M.Kumarasamy College of Engineering
NAAC Accredited Autonomous Institution
Approved by AICTE & Affiliated to Anna University
ISO 9001:2015 Certified Institution
Thalavapalayam, Karur - 639 113, TAMILNADU.

M. KUMARASAMY COLLEGE OF ENGINEERING

(Autonomous Institution affiliated to Anna University, Chennai)

KARUR – 639 113

BONAFIDE CERTIFICATE

Certified that this project report on **"DETECTING CYCLE IN A DIRECTED GRAPH"** is the bonafide work of **KAVIYA R (927624BAM025)** who carried out the project work during the academic year 2024- 2025 under my supervision.

Signature

Mrs. S.RAMYA,M.E.,(Ph.D.,),

ASSISTANT PROFESSOR,

Department of Artificial Intelligence,

M. Kumarasamy College of Engineering,

Thalavapalayam, Karur -639 113.

Signature

Dr. K.CHITIRAKALA, M.Sc., M.Phil.,Ph.D.,

HEAD OF THE DEPARTMENT,

Department of Freshman Engineering,

M. Kumarasamy College of Engineering,

Thalavapalayam, Karur -639 113.

Submitted for the End Semester Review held on _____

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

VISION OF THE INSTITUTION

To emerge as a leader among the top institutions in the field of technical education

MISSION OF THE INSTITUTION

- Produce smart technocrats with empirical knowledge who can surmount the global challenges
- Create a diverse, fully-engaged, learner-centric campus environment to provide quality education to the students
- Maintain mutually beneficial partnerships with our alumni, industry, and Professional associations

VISION OF THE DEPARTMENT

To create highly qualified competitive professionals in Artificial Intelligence and Machine Learning by designing intelligent solutions to solve problems in variety of business domains, applications such as natural language processing, text mining, robotics, reasoning and problemsolving that serves society with greater cause.

MISSION OF THE DEPARTMENT

- Impart practical and technical knowledge along with applications of various integrated technologies.
- Design and develop various intelligent engineering projects to solve societal issues.
- Use of advanced engineering tools and equipment to enable research based learning to promote ethical values, lifelong learning and entrepreneurial skills.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO 1: Develop intelligent software solutions demonstrating reasoning, learning and decision support while handling uncertainty using domain knowledge.

PEO 2: Create significant research towards social benefits and engineering improvement with a wide breadth knowledge of AI & ML technologies and their applications.

PEO 3: Participate in life-long learning for effective professional growth and demonstrate leadership qualities in disruptive technologies along with a capacity to critically analyse and evaluate design proposals.

PROGRAM OUTCOMES (POs)

Engineering students will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

1. **PSO1:** Utilize multidisciplinary knowledge along with Artificial intelligence and Machine Learning Principles to create innovative solutions for the development of society.
2. **PSO2:** Graduates will use Information and Communication Technology (ICT) tools and techniques to attain advance knowledge to exhibit state of the art technologies to overcome the demand of sustainable development to meet future business and society needs.

ABSTRACT

This project provides a comprehensive overview of cycle detection in directed graphs. Directed graphs are fundamental structures in computer science and numerous application domains such as software engineering, bioinformatics, and network analysis. One critical problem in working with directed graphs is detecting cycles situations where a path leads back to the origin vertex following the direction of edges. This application enables graph representation, initialization, DFS utility, cycle detection controller. Detecting cycle in a directed graph is used to identify if there is a loop where a path starts and ends at the same node, following the direction of edges. We begin with a fundamental introduction to directed graphs and cycles, followed by an in-depth examination of the Depth-First Search (DFS) algorithm.

ABSTRACT WITH POs AND PSOs MAPPING

ABSTRACT	POs MAPPED	PSOs MAPPED
This project provides a comprehensive overview of cycle detection in directed graphs. Directed graphs are fundamental structures in computer science and numerous application domains such as software engineering, bioinformatics, and network analysis. One critical problem in working with directed graphs is detecting cycles situations where a path leads back to the origin vertex following the direction of edges. This application enables graph representation, initialization, DFS utility, cycle detection controller. Detecting cycle in a directed graph is used to identify if there is a loop where a path starts and ends at the same node, following the direction of edges. We begin with a fundamental introduction to directed graphs and cycles, followed by an in-depth examination of the Depth-First Search (DFS) algorithm.	**PO1(2)** **PO2(3)** **PO3(2)** **PO4(2)** **PO5(3)** **PO6(1)** **PO7(3)** **PO8(2)** **PO9(3)** **PO10(3)** **PO11(2)** **PO12(2)**	**PSO1(3)** **PSO2(2)**

Note: 1- Low, 2-Medium, 3- High

SUPERVISOR **HEAD OF THE DEPARTMENT**

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

A cycle in a directed graph is a path that starts and ends at the same vertex, following the direction of edges consistently. Formally, a cycle exists if there is a sequence of edge leading from a vertex back to itself without repeating any edge or vertex, except for the start and end vertex. One effective way to detect a cycle in a directed graph is by using a Depth first search (DFS).

Depth-First Search (DFS) is a graph traversal technique used for exploring vertices and edges systematically. Starting from a vertex, DFS explores as deep as possible along each branch before backtracking, thereby visiting all reachable vertices. In the context of a Detecting cycle in a directed graph, it is critical to identify such cycles because they can cause infinite loops or logical inconsistencies in applications like task scheduling, deadlock detecting, and network routing. DFS can be adapted for both cycle detection and for extracting additional information like strongly connected components, making it a versatile and powerful tool in graph processing.

1.2 Objective

Our objective is to design and implement an algorithm that accurately detects whether a directed graph contains any cycles, ensuring the graph remains acyclic when necessary for correct execution of dependent operations.

1.3 Data Structure Choice

To detect cycles in a directed graph using Depth-First Search (DFS), appropriate data structures must be chosen to ensure efficient traversal and accurate cycle detection. The graph is typically represented using an adjacency list, which provides an efficient way to store and access the neighbors of each vertex. Visited[] is used

keep track of the nodes that have already been visited during the traversal. This helps avoid revisiting the same nodes and reduces unnecessary computations. Another important data structure is the recursion stack, maintained using a separate boolean array called recStack[]. This array keeps track of the nodes currently in the recursive call stack. If during DFS a node is encountered that is already present in the recursion stack, it indicates the presence of a cycle.

CHAPTER 2

PROJECT METHODOLOGY

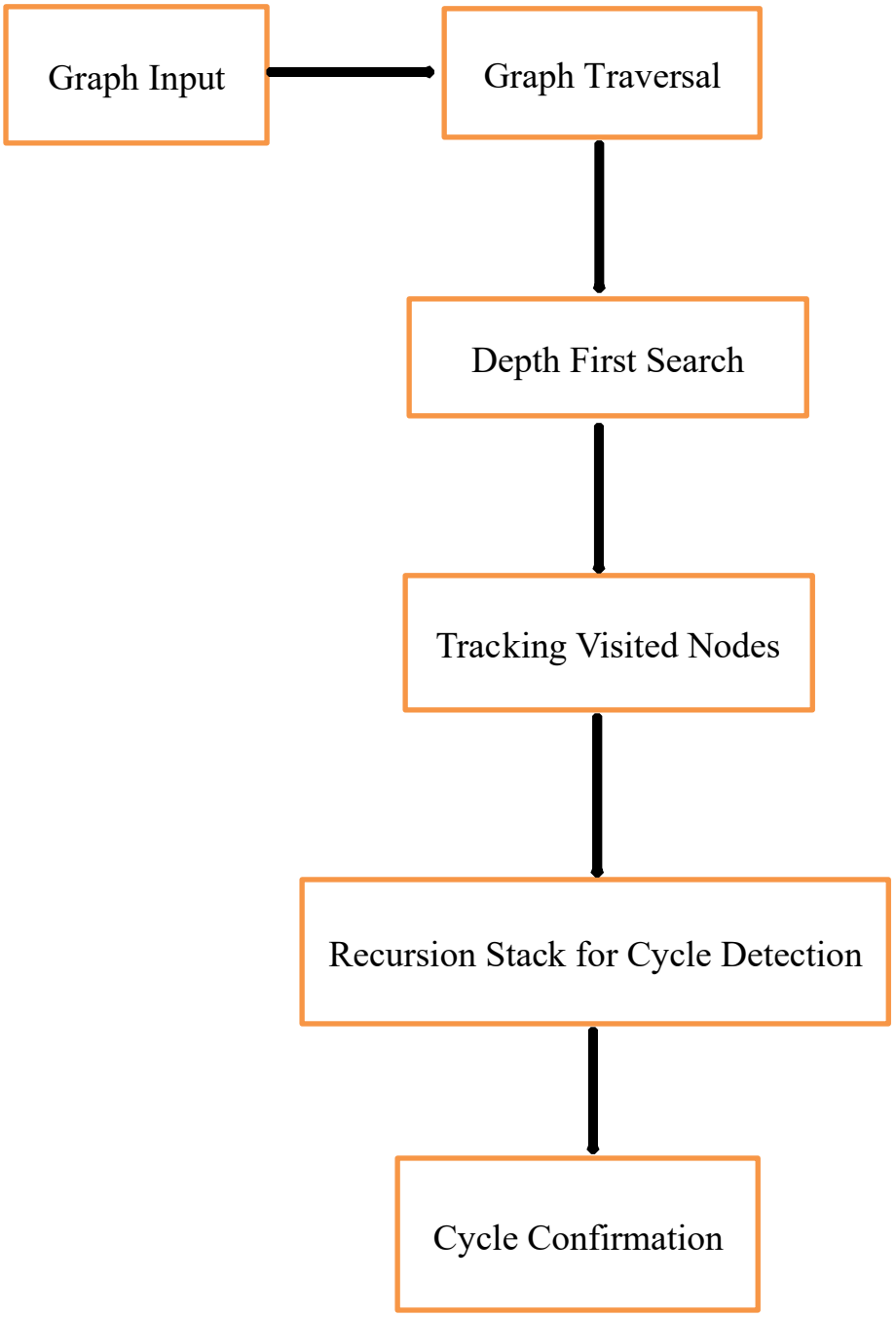
2.1 Depth First Search

Depth-First Search (DFS) is a graph traversal algorithm that explores nodes deeply along each path before backtracking. It works by visiting a starting node, exploring one of its neighbors recursively, and continuing until all reachable nodes are visited. If no unvisited neighbors remain, the algorithm backtracks to the previous node and explores other available paths. DFS can be implemented recursively or with an explicit stack.

Key features of a Depth First Search:

1. **Recursive or Stack-Based Execution**: DFS can be implemented using recursion or an explicit stack for iterative traversal.
2. **Backtracking Mechanism**: When a dead-end is reached, DFS returns to the previous node and explores alternative paths.
3. **Cycle Detection Capability**: DFS can track visited nodes and detect cycles, making it useful in dependency resolution and deadlock detection.
4. **Pathfinding and Connectivity**: Helps find connected components, check graph connectivity, and solve maze problems.
5. **Time Complexity Efficiency **: Runs in O(V + E) time, where V is the number of vertices and E is the number of edges.

2.2 Block Diagram



CHAPTER 3

MODULES

3.1 Graph Input

The process begins with taking a directed graph as input. This graph is structured using an adjacency list or adjacency matrix, allowing efficient representation of node relationships.

- Adjacency List: Represents connections as a list of neighboring nodes for each vertex. This is memory-efficient for sparse graphs.
- Adjacency Matrix: Uses a 2D table where row-column intersections denote edges between nodes. Suitable for dense graphs

The input graph contains vertices and directed edges, which define connections between nodes. These edges indicate paths that can be traversed only in a specific direction.

3.2 Graph initialization

Before starting traversal, necessary tracking structures are initialized:

- Visited Array**:** Maintains records of nodes that have already been explored to avoid redundant checks.
- Recursion Stack**:** A temporary stack used during DFS traversal to track nodes currently being explored. This stack plays a crucial role in detecting cycles, as revisiting a node in this stack indicates a loop.

3.3 DFS Traversal

The Depth-First Search (DFS) algorithm is used to traverse the graph. DFS explores nodes in a depth-wise manner, diving deep along each path before backtracking.

Traversal Steps:

- Start from any unvisited node and mark it as visited.
- Push the node into the recursion stack, indicating it is actively being explored.
- Explore all neighboring nodes recursively by following directed edges.
- If an unvisited neighbor is found, recursively call DFS on it.
- If a neighbor is already in the recursion stack, a cycle is detected, as it means a node has been revisited within the same traversal path.

3.4 Cycle Detection Logic

This block determines whether a cycle exists within the graph based on traversal behavior.

- If DFS revisits a node that is still present in the recursion stack, this indicates a back edge, meaning the graph has a cycle.
- If DFS fully explores a path and removes nodes from the recursion stack without encountering repetitions, the path is cycle-free.
- The algorithm checks all nodes to confirm the presence or absence of cycles.

DFS ensures that cycle detection is efficient, running in O(V + E) time complexity, where V represents vertices and E represents edges.

3.5 Output Generation

Once DFS completes execution, the final result is determined:

- If a cycle is detected, the program outputs "Cycle Detected."
- If no cycles exist, the program outputs "No Cycle Found."

CHAPTER 4

RESULTS AND DISCUSSION

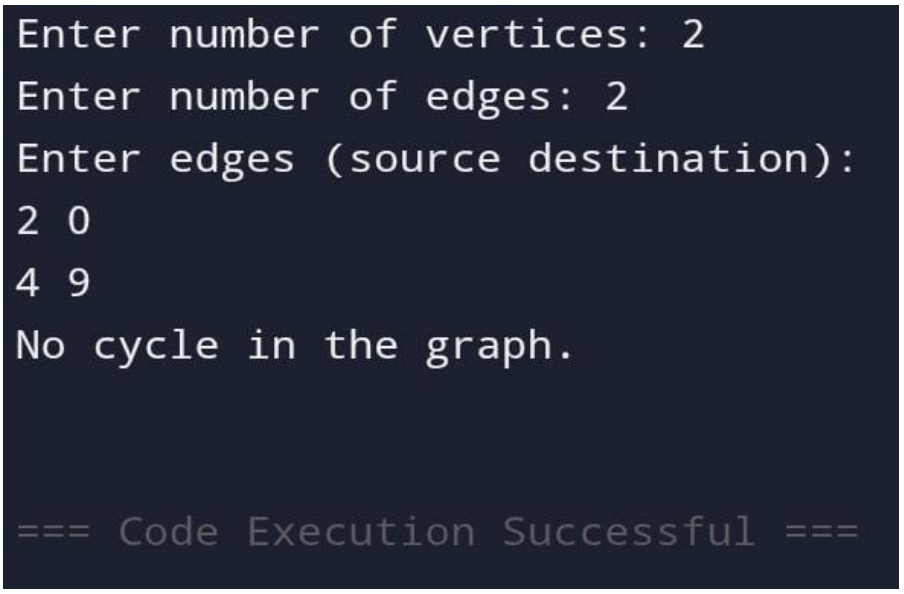
4.1 Results

4.1.1 Cycle Found

```
Enter number of vertices: 4
Enter number of edges: 4
Enter edges (source destination):
2 0
4 2
3 4
0 2
Cycle detected in the graph.


=== Code Execution Successful ===
```

4.1.2 No Cycle Found



4.2 Discussion

The implemented algorithm successfully detects cycles in various test cases, including graphs with complex structures and multiple strongly connected components. Results demonstrate that the use of DFS with recursion stack tracking allows prompt identification without exhaustive enumeration of cycles. Performance analysis reveals that the time complexity remains O(V+E), where V is the number of vertices and E the edges, which is optimal for traversal-based solutions. The adjacency list's space efficiency further supports scalability to large graphs. Discussions focus on limitations such as handling extremely large graphs requiring iterative DFS or parallel processing for performance improvement. Extensions could include cycle path retrieval and algorithms tailored to specialized graph types.

CHAPTER 5

CONCLUSION

In conclusion, Cycle detection in directed graphs is essential for numerous applications to prevent logical errors and system failures. This project consolidates theoretical concepts with practical implementation using DFS and adjacency lists. The approach is efficient, reliable, and adaptable, making it suitable for educational purposes as well as real-world software systems. The modular design enhances maintainability and allows easy extension to incorporate enhanced features. Future work in cycle detection explores scaling algorithms for massive graphs, incorporating parallel and distributed processing. Advances in dynamic graph algorithms allow incremental cycle detection as graphs evolve in real- time. It could investigate alternative algorithms like Kahn's algorithm for topological sorting-based cycle detection, iterative implementations, and applying the solution to domain-specific problems like deadlock detection in operating systems.

REFERENCES

1. Detect Cycle in a Directed Graph – Provides a C implementation for cycle detection using DFS.
2. Geeks for Geeks Practice Problem – Offers a structured problem-solving approach for detecting cycles in directed graphs.
3. W3schools – Covers cycle detection using DFS and union-find with interactive Examples.

APPENDIX

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100

// Node structure for adjacency list
typedef struct Node {     int
vertex;     struct Node* next;
} Node;

// Graph structure typedef
struct Graph {     int
numVertices;
   Node* adjList[MAX_VERTICES];
} Graph;

// Create a graph
Graph* createGraph(int vertices) {
   Graph* graph = (Graph*)malloc(sizeof(Graph));
graph->numVertices = vertices;     for (int i = 0; i <
vertices; i++) {         graph->adjList[i] = NULL;
   }     return
graph;
}

// Add edge to the graph
```

```
void addEdge(Graph* graph, int src, int dest) {
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->vertex = dest;     newNode->next =
graph->adjList[src];     graph->adjList[src] =
newNode;
}

// DFS helper for cycle detection bool dfsCycle(Graph* graph, int v,
bool visited[], bool recStack[]) {     visited[v] = true;     recStack[v]
= true;

   Node* temp = graph->adjList[v];     while (temp !=
NULL) {         int neighbor = temp->vertex;         if
(!visited[neighbor]) {             if (dfsCycle(graph,
neighbor, visited, recStack))                 return true;
       } else if (recStack[neighbor]) {
return true;
       }
       temp = temp->next;
   }

   recStack[v] = false;
return false;
}

// Main function to check for cycles bool
isCyclic(Graph* graph) {     bool
visited[MAX_VERTICES] = { false };     bool
recStack[MAX_VERTICES] = { false };
```

```
   for (int i = 0; i < graph->numVertices; i++) {
if (!visited[i]) {             if (dfsCycle(graph, i,
visited, recStack))                 return true;
       }     }
return false; }

// Main int main() {     int vertices,
edges;     printf("Enter number of
vertices: ");     scanf("%d",
&vertices);

   printf("Enter number of edges: ");
scanf("%d", &edges);

   Graph* graph = createGraph(vertices);

   printf("Enter edges (source destination):\n");
for (int i = 0; i < edges; i++) {         int src,
dest;
       scanf("%d %d", &src, &dest);
addEdge(graph, src, dest);
   }
  if (isCyclic(graph))
printf("Cycle detected in the graph.\n");
 else
 printf("No cycle in the graph.\n");
```

```
return 0; }
```