KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

GRAPH PATH COUNTING USING MATRIX EXPONENTIATION AND DIVIDE AND CONQUER METHOD

FOR

DESIGN AND ANALYSIS OF ALGORITHMS (22ITT31)

SUBMITTED BY

VENMUGIL S

(23ITR169)

KONGU ENGINEERING COLLEGE
(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

GRAPH PATH COUNTING USING MATRIX EXPONENTIATION AND DIVIDE AND CONQUER METHOD

A MICRO PROJECT REPORT

FOR

DESIGN AND ANALYSIS OF ALGORITHMS (22ITT31)

SUBMITTED BY

VENMUGIL S

(23ITR169)

KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

BONAFIDE CERTIFICATE

Name	: VENMUGIL S
Course Code	: 22ITT31
Course Name	: DESIGN AND ANALYSIS OF ALGORITHMS
Semester	: IV

Certified that this is a bonafide record of work for application project done by the above student for 22ITT31-DESIGN AND ANALYSIS OF ALGORITHMS during the academic year 2024-2025.

Submitted for the Viva Voice Examination held on _____

Faculty Incharge Head of the Department

ABSTRACT

"Path Count Finder in Directed Graph using Matrix Exponentiation" is a web-based application designed to compute the number of distinct paths of a specified length k between two vertices in a directed graph. The system integrates an interactive frontend built with HTML and JavaScript and a Python (Flask) backend for executing the core logic.

The project is based on the Divide and Conquer approach, a fundamental concept in Design and Analysis of Algorithms (DAA). Specifically, it employs Exponentiation by Squaring to raise the adjacency matrix to the kth power in an optimized time complexity of $O(n^3 \times \log k)$, where n is the number of vertices. This method recursively reduces the problem size, minimizing unnecessary computations and ensuring faster results even for larger graphs.

The application accepts user-defined graph input, constructs the adjacency matrix, computes the powered matrix using matrix exponentiation, and displays the number of paths of length k between selected vertices. Additionally, it provides execution time and theoretical time complexity to help users understand algorithm performance.

This project showcases the practical use of efficient algorithmic strategies for solving real-world graph problems.

TABLE OF CONTENTS

CHAPTER NO	TITLE	PAGE NO
	ABSTRACT	**ix**
1.	**INTRODUCTION**	6
	1.1 Purpose	7
	1.2 Objective	7
	1.3 Methodology Overview	8
2.	**PROBLEM STATEMENT**	9
3.	**METHODOLGY**	10
	3.1 Input & Initialization	10
	3.2 Divide & Compare	10
	3.3 Brute Force	11
	3.4 Visualization & Output	12
4.	**IMPLEMENTATION**	13
	4.1 Input & Initialization	13
	4.2 Divide & Compare	13
	4.3 Brute Force	14
	4.4 Step By Step Implementation	14
5.	**RESULTS**	18

1. INTRODUCTION

Graphs are fundamental data structures used to model relationships and interactions in various domains such as computer networks, social networks, transportation systems, and biological systems. A common problem in graph theory is determining the number of distinct paths between two nodes for a given length. This information is crucial for analysing connectivity, communication paths, and data flow in directed networks.

This project, "Path Count Finder in Directed Graph using Matrix Exponentiation", aims to solve the problem of counting the number of distinct paths of a specified length k between two vertices in a directed graph. The application uses matrix exponentiation, an efficient algorithmic technique based on the divide and conquer paradigm, to compute the solution in reduced time complexity.

The system is implemented as a web-based application with an intuitive frontend (HTML and JavaScript) and a Python Flask backend that handles the core computation. Users can input a custom directed graph, specify the path length, and choose source and destination vertices to get real-time results. The matrix exponentiation technique employed significantly improves performance over naive recursive or brute-force methods, especially for large graphs.

This project showcases how theoretical algorithmic strategies can be practically applied to solve real-world graph problems efficiently, while also helping users understand the underlying computational complexity and optimization.

1.1 PURPOSE

The main purpose of this project is:

- To make advanced algorithmic techniques like matrix exponentiation more accessible and understandable to students and learners.
- To highlight the efficiency of the divide and conquer strategy in solving complex graph problems, such as path counting in directed graphs.
- To encourage logical thinking and algorithm design through an interactive and visual platform.
- To serve as a practical learning aid for understanding real-world applications of graph theory and algorithm optimization in computer science.

1.2 OBJECTIVE

The main objective of this project is to design and implement an efficient system that computes the number of distinct paths of a given length k between two vertices in a directed graph using matrix exponentiation. The specific goals include:

- To accept user-defined graph input and construct an adjacency matrix based on directed edges.
- To apply the divide and conquer technique through exponentiation by squaring for efficient matrix power computation.
- To provide accurate and fast calculation of the number of paths of length kkk using the powered matrix.
- To visualize the algorithm's performance by displaying execution time and time complexity, enhancing understanding of algorithmic efficiency.

1.3 METHODOLOGY OVERVIEW

The project follows a structured approach to calculate the number of distinct paths of a specific length k between two vertices in a directed graph using Matrix Exponentiation based on the Divide and Conquer strategy. The implementation steps are as follows:

1. **User Input:**
   The user inputs the number of vertices and edges in the directed graph, the edges themselves, the path length k, and the start and end vertices for path computation.
2. **Adjacency Matrix Construction:**
   A square adjacency matrix of size n×n \times n×n is constructed based on the user-defined edges, where each entry A[i][j] = 1 indicates a directed edge from vertex i to vertex j.
3. **Matrix Exponentiation (Divide and Conquer):**
   The adjacency matrix is raised to the power k using Exponentiation by Squaring, a divide and conquer algorithm that reduces the time complexity from O(n^k) to O(n^3·logk).
4. **Path Count Calculation:**
   After matrix exponentiation, the resulting matrix A^k holds in each cell A^k[i][j] the number of distinct paths of length k from vertex i to vertex j.
5. **Base Case Handling:**
   If k=0, the algorithm returns the identity matrix, as each node has one path of length 0 to itself and none to others.
6. **Visualization and Output:**
   The application displays the computed number of paths between the selected vertices, along with the generated adjacency and powered matrices. It also provides execution time and theoretical time complexity to help users understand algorithm performance.

2. PROBLEM STATEMENT

The Path Count Problem in directed graphs is a fundamental algorithmic challenge that involves computing the number of distinct paths of a specific length k between two given vertices in a graph with n vertices. These paths must follow the direction of edges, and the total number of such paths can grow rapidly with increasing k, making naive approaches inefficient for large graphs.

This project focuses on implementing a Matrix Exponentiation approach, a recursive divide and conquer algorithm that raises the graph's adjacency matrix to the power k. This method significantly reduces the computational time compared to traditional path enumeration techniques and provides accurate results even for large and complex graphs.

3. Path Count Finder Methodology

3.1Input & Initialization

- Accept the number of vertices n and edges e from the user.
- Collect the directed edges as pairs of source and destination vertices.
- Build an n×n \times n×n adjacency matrix to represent the graph.
- Accept additional input: the path length k, start vertex i, and end vertex j.

3.2 Matrix Exponentiation (Divide and Conquer)

- Apply the divide and conquer strategy using Exponentiation by Squaring.
- If k is even, recursively compute A^{k/2} and square it.
- If k is odd, compute A×A^{k-1} recursively.
- This reduces repeated multiplication and improves performance.

Algorithm:

i. Step 1: Define a function to multiply two square matrices A and B

matrix_mult(A, B):

- Create an empty result matrix of same size (n x n) initialized to 0
- For each cell [i][j] in result:
  - Loop over k from 0 to n-1
  - Multiply A[i][k] with B[k][j] and add to result[i][j]
  - Return the result matrix

ii. Step 2: Perform matrix exponentiation recursively

matrix_power(A, k):

- If k == 1:

- Return A (base case)

- If k is even:

  - Recursively compute half_power = matrix_power(A, k // 2)

  - Return matrix_mult(half_power, half_power)

- If k is odd:

  - Compute power_minus_one = matrix_power(A, k - 1)

  - Return matrix_mult(A, power_minus_one)

3.3 Recursive Power Calculation (Brute-Force)

- Recursively reduce the exponent until reaching the base case.
- For k=0, return the identity matrix.
- For each recursion, multiply intermediate matrices using standard matrix multiplication.
- The resulting matrix A^k stores the number of paths of length k between all vertex pairs.

Algorithm:

i. Matrix Multiplication Procedure

```
Function MATRIX_MULTIPLY(A, B):
    Input: Two matrices A and B of size n x n
    Output: Matrix C = A × B
    1. Create a new matrix C of size n x n, initialized with 0s
    2. For i = 0 to n-1:
        For j = 0 to n-1:
            For k = 0 to n-1:
                C[i][j] = C[i][j] + A[i][k] * B[k][j]
    3. Return matrix C
```

ii. Identity Matrix Generator

```
Function IDENTITY_MATRIX(n):
```

Input: Integer n

Output: Identity matrix I of size n x n

1. Create a new matrix I of size n x n, initialized with 0s
2. For i = 0 to n-1:
   I[i][i] = 1
3. Return I

iii. Recursive Power Function

Function MATRIX_POWER(A, k):

Input: Matrix A (adjacency matrix), Integer k

Output: Matrix A^k

1. If k == 0:
   Return IDENTITY_MATRIX(size of A)
2. If k == 1:
   Return A
3. Else:
   Temp = MATRIX_POWER(A, k - 1)
   Return MATRIX_MULTIPLY(A, Temp)

3.4 Visualization & Output

- Display the initial adjacency matrix and the final powered matrix A^k.
- Show the number of paths of length k from the selected start to end vertex.
- Provide additional details such as execution time and theoretical time complexity O(n3·logk).

4. IMPLEMENTATION :

4.1 Input & Initialization

```
const vertices = parseInt(document.getElementById("vertices").value);
const edges = [];
const adjMatrix = Array.from({ length: vertices }, () => Array(vertices).fill(0));
edges.forEach(([u, v]) => {
if (u >= 0 && u < vertices && v >= 0 && v < vertices) {
adjMatrix[u][v] = 1;
}
});
```

4.2 Divide & Compare

```
function multiplyMatrices(A, B) {
const n = A.length;
const result = Array.from({ length: n }, () => Array(n).fill(0));
for (let i = 0; i < n; i++)
for (let j = 0; j < n; j++)
for (let k = 0; k < n; k++)
result[i][j] += A[i][k] * B[k][j];
return result;
}
function matrixExponentiation(matrix, power) {
const n = matrix.length;
let result = Array.from({ length: n }, (_, i) =>
Array.from({ length: n }, (_, j) => (i === j ? 1 : 0))
);
let base = matrix;
while (power > 0) {
if (power % 2 === 1) result = multiplyMatrices(result, base);
base = multiplyMatrices(base, base);
power = Math.floor(power / 2);
}
return result;
}
```

4.3 Path Counting And Output

```
const poweredMatrix = matrixExponentiation(adjMatrix, k);
const pathCount = poweredMatrix[start][end];
console.log(`Number of paths of length ${k} from vertex ${start} to ${end}:
${pathCount}`);
```

4.4 Step By Step Implementation

- Input:
  i. Vertices: 3
  ii. Edges: (0→1), (1→2), (2→0)
  iii. k = 2, Start = 0, End = 2
- Adjacency Matrix built

  A = [[0, 1, 0],
  [0, 0, 1],
  [1, 0, 0]]
- Raised matrix to power k using multiplication
- Final result taken from A^k[start][end]
- Output: 1 path from 0 to 2 of length 2

DIFFERENCE BETWEEN BRUTE FORCE AND DIVIDE AND CONQUER:

Brute Force:

Concept:

- Calculate all possible paths of length k by recursively exploring every combination.
- Traverse the graph from the start node and follow all valid paths of length k.

How it works:

- From the start vertex, visit every connected vertex recursively until

  the path length reaches kkk.

- Count paths only if the last vertex matches the destination vertex.
- Repeat this process for all possible paths of length kkk.
- Accumulate the total count of valid paths.

Time Complexity:

- Worst-case: O(nk)O(n^k)O(nk)
- Every recursive step may branch to multiple vertices, leading to exponential growth in time.

Divide and Conquer Approach (Matrix Exponentiation):

Concept:

- Represent the graph using an adjacency matrix.
- Use Exponentiation by Squaring, a divide and conquer method, to raise the matrix to the power kkk.
- The resulting matrix directly gives the number of paths of length kkk between all pairs of vertices.

How it works:

- Construct the adjacency matrix AAA for the given graph.
- Compute A^k using matrix exponentiation.
    1. If k is even, compute A^{k/2} and square it.
    2. If k is odd, compute A×A^k−1.
- The entry at A^k[i][j] gives the number of paths of length k from vertex i to vertex j.

Time Complexity:

- O(n^3·logk)

Pros:

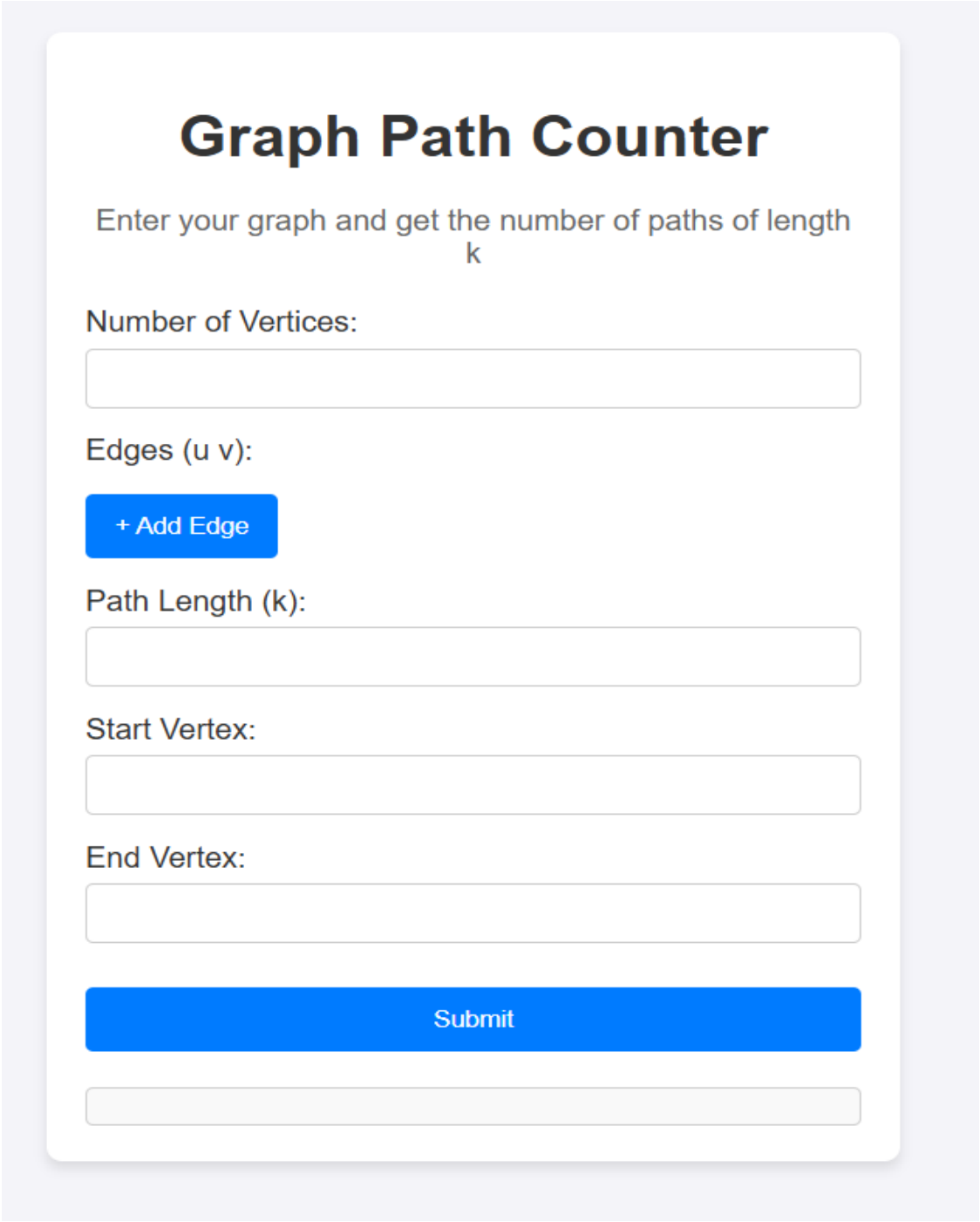
- Highly efficient for large graphs and higher path lengths.
- Significantly reduces computation using Exponentiation by Squaring.
- Scales well with increasing values of kkk due to logarithmic exponentiation steps.

Cons:

- Requires implementation of matrix operations, which can be complex for beginners.
- Involves higher space usage due to matrix storage and multiplication.
- May be less effective for extremely sparse graphs due to unnecessary matrix
- overhead.
- Difficult to interpret intermediate steps compared to step-by-step traversal methods.

Feature	**Brute Force**	**Divide and Conquer**
Strategy	Path-by-path enumeration	Recursive matrix exponentiation
Time Complexity	O(n^k)	O(n^3.log k)
Efficiency	Low	High
Ideal for	Very small graphs Or small k	Large graphs and large k
Computation Method	Recursive DFS or BFS traversal	Matrix-based exponentiation
Logic Complexity	Simple	Moderate

5. RESULTS


Graph Path Counter
Enter your graph and get the number of paths of length k
Number of Vertices:
Edges (u v):
+ Add Edge
Path Length (k):
Start Vertex:
End Vertex:
Submit

Github Link: https://github.com/venmugil182005/DAA