**a project on**

**OPTIMIZED TRAVEL PLANNER USING DIJKSTRA’S ALGORITM AND GRAPHS**

**Submitted by**

Sameer Syed Mohammad (AP23110010829)

Bhavana Paturi (AP23110010823)

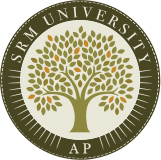
Mokshitha Sanka (AP23110010836)

**Bachelor of Technology**

**In**

**Computer Science and Engineering**

**School of Engineering and Sciences**



Under the guidance of

Mr. PAMULA UDAY RAJU

Assistant Professor-Ad Hoc

Department of CSE

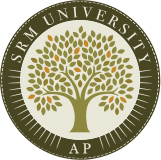
**[November, 2024]**

SRM University, AP, Neerukonda, Mangalagiri, Guntur.

Andhra Pradesh – 522 240

Department of Computer Science and Engineering

SRM University, AP



**CERTIFICATE**

*This is to certify that the Project report entitled “***Optimized Travel Planner Using Dijkstra’s and Graphs***” is being submitted by* ***SAMEER SYED MOHAMMAD*** *(AP23110010829),* ***BHAVANA PATURI*** *(AP23110010823) and* ***MOKSHITHA SANKA*** *(AP23110010836) students of Department of Computer Science and Engineering, SRM University, AP, in partial fulfilment of the requirement for Design and Analysis of Algorithms Lab for II-B. Tech (CSE), Semester III. carried out by him during the academic year 2024-2025.*

Signature of the Lab In charge

Mr. PAMULA UDAY RAJU

Assistant Professor-Ad Hoc

Department of CSE

TABLE OF CONTENTS

**Name of the content Page. No**

List of figures

List of tables

Abstract

1. Introduction
2. Module organization
   1. Module overview
   2. Module description
   3. Module implementation
3. Performance Evaluation
4. Conclusion

Appendix

* + - 1. Sample code
      2. Sample screenshots

**List of figures**

**Figure i-** Showshow depth first search works

**Figure ii-** Shows how breadth first traversal works

**Figure iii-**Shows how the **Dijkstra's** **algorithm** works

**Figure iv-** Shows the user prompt

**Figure v-** Shows **BFS and DFS Traversal**

**Figure vi** – Gives output according to **Dijkstra's Algorithm**

**List of tables**

**Table 1-** System requirements

**Table 2-**Module overview

# **Abstract:**

The **Travel Planner using Dijkstra's Algorithm** project aims to develop an efficient tool for route optimization, using Dijkstra's algorithm to find the shortest path between various locations in a travel network. The project simulates a real-world scenario where a traveler needs to plan the most time-efficient or cost-effective route across multiple destinations, considering various transport nodes (such as cities, landmarks, or airports) connected by weighted edges (representing distance, time, or cost).

In this project, a graph-based approach is used where each location is represented as a node, and the edges between nodes represent possible travel routes with associated weights. Dijkstra's algorithm, a well-known graph search algorithm, is employed to calculate the shortest path between a starting location and a destination, considering all possible routes and providing the traveller with an optimal route.

The system allows users to input a list of destinations, and the algorithm efficiently computes the most optimal path in terms of distance, time, or cost, depending on the user's preference. The project integrates a user-friendly interface that allows easy input of travel nodes and parameters, ensuring a smooth experience for users seeking to plan their trips effectively. This application could be expanded for use in various domains, such as urban planning, logistics, or tourism, where optimal route calculation is a critical component of efficient travel planning.

By utilizing Dijkstra's algorithm, the system ensures computational efficiency even with large networks of travel nodes, making it an invaluable tool for travellers, businesses, and planners seeking to optimize travel routes in real-time.

**1.Introduction**

In an era where time and cost efficiency are crucial for both travellers and service providers, route optimization plays a key role in effective travel planning. The task of finding the most efficient path between multiple destinations is complex, particularly when considering factors like distance, time, and cost. Traditional manual route planning can be time-consuming and prone to human error, especially when dealing with large networks of locations. To address these challenges, graph-based algorithms such as Dijkstra's algorithm provide a robust solution by systematically finding the shortest or most optimal path in a network.

This project focuses on leveraging Dijkstra's algorithm to create a Travel Planner application that can optimize travel routes based on user-defined parameters. By representing locations as nodes and travel paths as weighted edges, the algorithm can efficiently determine the shortest path between a source and a destination, even when multiple intermediate stops are involved. The proposed system not only calculates the most efficient route in terms of distance but also allows for flexibility in optimization, such as minimizing travel time or cost.

This Travel Planner aims to simplify travel decisions, provide quick route recommendations, and support a range of real-world applications, including tourism, logistics, and urban transport planning. Through this approach, travellers and businesses can save valuable resources and enhance the overall travel experience.

**2.Module organization**

**2.1. Module overview**

|  |  |  |
| --- | --- | --- |
| **Module** | **Data structure/Algorithm** | **Description** |
| **Taking source and destination** | **graphs** | **This module helps the user to give input (ie., location)** |
| **Adding weights** | **graphs** | **This module assigns the cost to travel b/w locations** |
| **Traversing routes** | **DFS/BFS** | **This module helps to know all the available routes b/w source and destination** |
| **Shortest path** | **Dijkstra’s** | **This module helps to find the optimal route for travelling** |

**2.2 Module description**

**1.Taking source and destination:** In this module the program prompts the user to give information about the locations they want to travel.

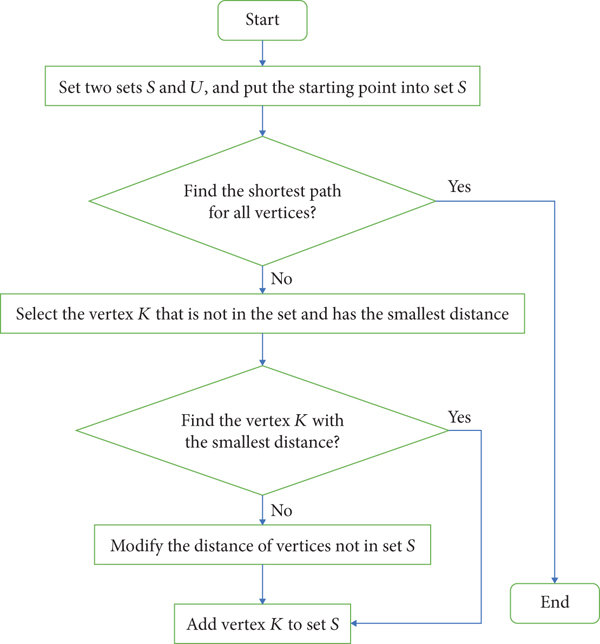
* The locations given by the user are converted and stored as nodes.
* By taking all the nodes(locations) these nodes are connected to form a graph data structure.

**2.adding weights:** This module helps the user to assign the distance between two places(nodes). This module does not accept negative weights or self loop. Which is essential in Dijkstra's algorithm.

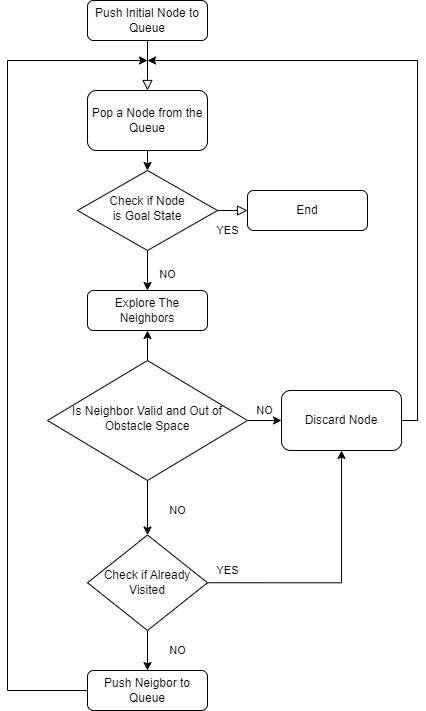
**3.Traversing routes:** This module facilitates the traversal of available routes between the source and destination using Depth-First Search (DFS) or Breadth-First Search (BFS) algorithms. It helps explore all possible paths within the graph from the starting point to the destination, allowing the user to see different ways to travel between locations. DFS explores routes deeply before backtracking, while BFS explores all adjacent routes level by level.

**Shortest Path (Dijkstra’s Algorithm):+**This module implements Dijkstra's algorithm to find the shortest path between the source and destination in a weighted graph. By considering the weights (costs) on the edges, the algorithm efficiently calculates the least costly route, ensuring optimal travel. It is particularly useful in scenarios where minimizing distance, time, or other costs is essential, and it helps users identify the most efficient route for travel.

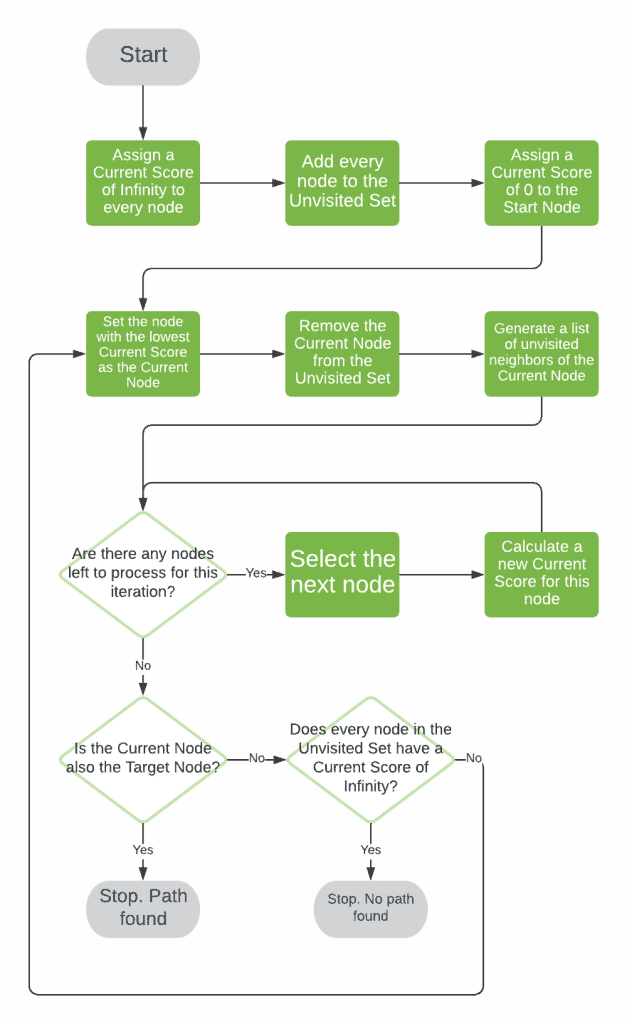
**Module implementation:**



Fig(i)



Fig(ii)



Fig(iii)

**Performance Evaluation**

**DFS (Depth-First Search)**:

**Time Complexity**:  
DFS explores all nodes and edges in the graph. In the worst case, it visits each vertex (V) and edge (E), leading to a time complexity of **O(V + E).**

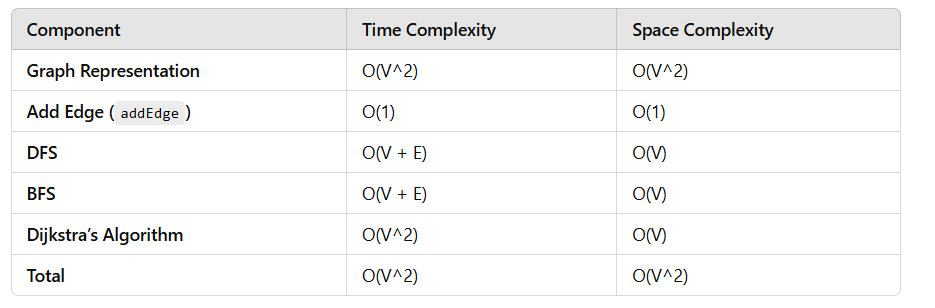
**BFS (Breadth-First Search)**:

**Time Complexity**:  
Similar to DFS, BFS processes each vertex and edge once, resulting in a time complexity of **O(V + E).**

**Dijkstra's Algorithm**:

**Time Complexity**:

* + **Naive Dijkstra's Algorithm (as implemented)**:  
    The algorithm repeatedly scans the list of vertices to find the one with the minimum tentative distance. This search takes **O(V)** time for each of the V vertices, leading to a total time complexity of **O(V^2).**



**Conclusion**

This project presents a basic implementation of graph-related algorithms, including Depth-First Search (DFS), Breadth-First Search (BFS), and Dijkstra’s algorithm, to solve pathfinding problems in a graph. The algorithms are implemented on an adjacency matrix, which provides an efficient way to represent and manipulate the graph for small to moderately-sized datasets.

In terms of performance, the project demonstrates the following:

Graph Representation: The adjacency matrix allows for efficient edge lookups and updates, but it comes with a space complexity of O(V^2), which may become inefficient for large or sparse graphs.

DFS and BFS: Both traversal algorithms efficiently explore the graph, with time complexity of O(V + E), making them suitable for most graph exploration tasks.

Dijkstra’s Algorithm: The naive implementation of Dijkstra’s algorithm, based on array-based minimum distance search, results in a time complexity of O(V^2). While functional, this approach is suboptimal for large graphs. The time complexity could be improved by using a priority queue (min-heap), which would reduce the complexity to

O((V + E) \*logV).

Overall, the project successfully implements fundamental graph algorithms, but the performance could be significantly improved, especially for larger graphs, by optimizing the Dijkstra’s algorithm. Future work could focus on adopting more efficient graph representations (e.g., adjacency lists for sparse graphs) and optimizing Dijkstra’s algorithm using data structures like heaps to handle larger-scale problems effectively.

**Sample code**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX\_VERTICES 10 // Max number of vertices in the graph

// Graph structure (Adjacency matrix representation)

int graph[MAX\_VERTICES][MAX\_VERTICES];

// Function to initialize the graph with 0 (no edges initially)

void initGraph(int vertices) {

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

for (int j = 0; j < vertices; j++) {

graph[i][j] = 0;

}

}

}

// Function to add a weighted edge between two nodes

void addEdge(int u, int v, int weight) {

graph[u][v] = weight;

graph[v][u] = weight; // For undirected graph, assuming bidirectional edges

}

// Function to perform Depth-First Search (DFS) to find all routes from source to destination

void DFS(int graph[MAX\_VERTICES][MAX\_VERTICES], int visited[MAX\_VERTICES], int start, int destination, int vertices) {

printf("%d ", start);

visited[start] = 1;

if (start == destination) {

printf("\n");

return;

}

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

if (graph[start][i] != 0 && !visited[i]) {

DFS(graph, visited, i, destination, vertices);

}

}

visited[start] = 0; // Backtrack

}

// Function to perform Breadth-First Search (BFS) to find all routes from source to destination

void BFS(int graph[MAX\_VERTICES][MAX\_VERTICES], int start, int destination, int vertices) {

int queue[MAX\_VERTICES], front = 0, rear = 0;

int visited[MAX\_VERTICES] = {0};

queue[rear++] = start;

visited[start] = 1;

while (front != rear) {

int current = queue[front++];

printf("%d ", current);

if (current == destination) {

printf("\n");

return;

}

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

if (graph[current][i] != 0 && !visited[i]) {

queue[rear++] = i;

visited[i] = 1;

}

}

}

}

// Dijkstra's Algorithm to find the shortest path

void dijkstra(int graph[MAX\_VERTICES][MAX\_VERTICES], int source, int vertices) {

int dist[MAX\_VERTICES], prev[MAX\_VERTICES];

int visited[MAX\_VERTICES] = {0};

// Initialize distances as infinity and previous nodes as -1

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

dist[i] = INT\_MAX;

prev[i] = -1;

}

dist[source] = 0;

for (int count = 0; count < vertices - 1; count++) {

int minDist = INT\_MAX, u = -1;

// Find the vertex with the minimum distance

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

if (!visited[v] && dist[v] < minDist) {

minDist = dist[v];

u = v;

}

}

visited[u] = 1;

// Update distances for neighbors of u

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

if (graph[u][v] != 0 && !visited[v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

prev[v] = u;

}

}

}

// Print the shortest paths

printf("Shortest distances from node %d:\n", source);

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

if (dist[i] == INT\_MAX)

printf("No path to %d\n", i);

else

printf("Distance to %d: %d\n", i, dist[i]);

}

}

// Function to display the shortest path from source to destination

void printPath(int prev[MAX\_VERTICES], int destination) {

if (prev[destination] == -1) {

printf("No path to %d\n", destination);

return;

}

int path[MAX\_VERTICES], pathIndex = 0;

int current = destination;

while (current != -1) {

path[pathIndex++] = current;

current = prev[current];

}

printf("Shortest Path: ");

for (int i = pathIndex - 1; i >= 0; i--) {

printf("%d ", path[i]);

}

printf("\n");

}

int main() {

int vertices, edges, u, v, weight, source, destination;

// Take input for number of vertices and edges

printf("Enter number of vertices: ");

scanf("%d", &vertices);

initGraph(vertices);

// Take input for the edges and their weights

printf("Enter number of edges: ");

scanf("%d", &edges);

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

printf("Enter edge (u v weight): ");

scanf("%d %d %d", &u, &v, &weight);

addEdge(u, v, weight);

}

// Take source and destination for pathfinding

printf("Enter source and destination nodes: ");

scanf("%d %d", &source, &destination);

// DFS Traversal

printf("DFS Traversal from %d to %d:\n", source, destination);

int visited[MAX\_VERTICES] = {0};

DFS(graph, visited, source, destination, vertices);

// BFS Traversal

printf("BFS Traversal from %d to %d:\n", source, destination);

BFS(graph, source, destination, vertices);

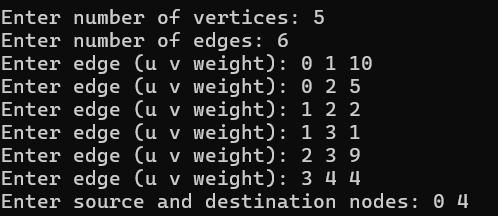
// Dijkstra's algorithm for shortest path

dijkstra(graph, source, vertices);

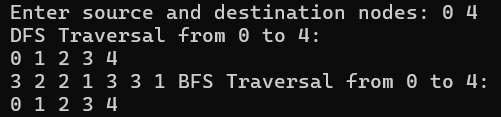
return 0;

}

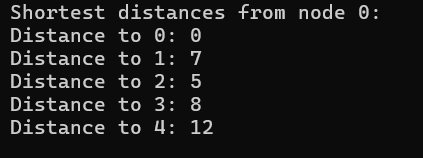
**Sample screenshots**



Fig(iv)

****

Fig(v)



Fig(vi)