TABLE OF CONTENTS

Introduction

In the study of algorithms, graphs serve as a powerful abstraction for modelling connections and relationships in a vast array of real-world systems, from computer networks and road systems to social networks and biological pathways. A fundamental problem in graph analysis is finding the shortest path between two vertices, which translates to finding the most efficient route, the cheapest connection, or the fastest communication link.

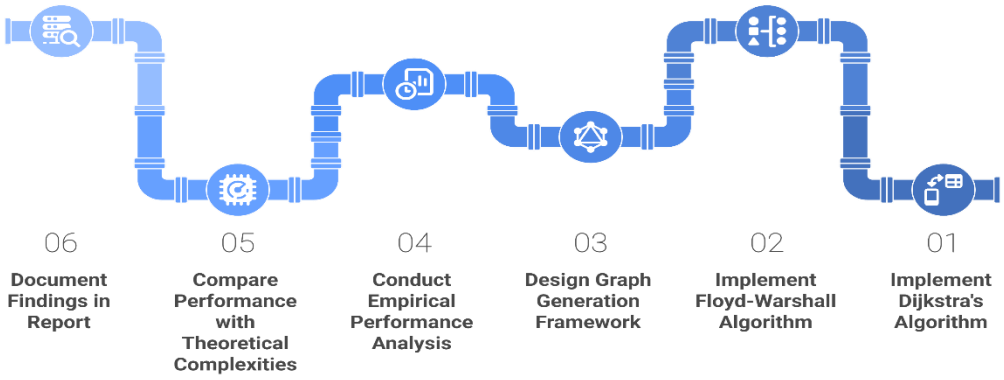
Shortest path problems are primarily categorized into two types:

1. ***Single-Source Shortest Path (SSSP)**: Finding the shortest paths from a single starting vertex to all other vertices in the graph.*
2. ***All-Pairs Shortest Path (APSP):** Finding the shortest paths between every pair of vertices in the graph.*

*This project focuses on two canonical algorithms that address these problems. **Dijkstra's algorithm**, developed by **Edsger W. Dijkstra** in 1956, is a **greedy algorithm** that efficiently solves the SSSP problem for graphs with **non-negative edge weights**. On the other hand, the **Floyd-Warshall** algorithm is a dynamic programming algorithm that solves the APSP problem, offering the flexibility to handle graphs with **negative edge weights** (provided there are no negative-weight cycles).*

*The primary goal of this study is to **move** beyond **theoretical understanding** and conduct an **empirical** comparison of these two algorithms. By implementing them and testing their **performance** on graphs of varying characteristics, we can gain **practical** insights into their **scalability, efficiency, and optimal** use cases.*

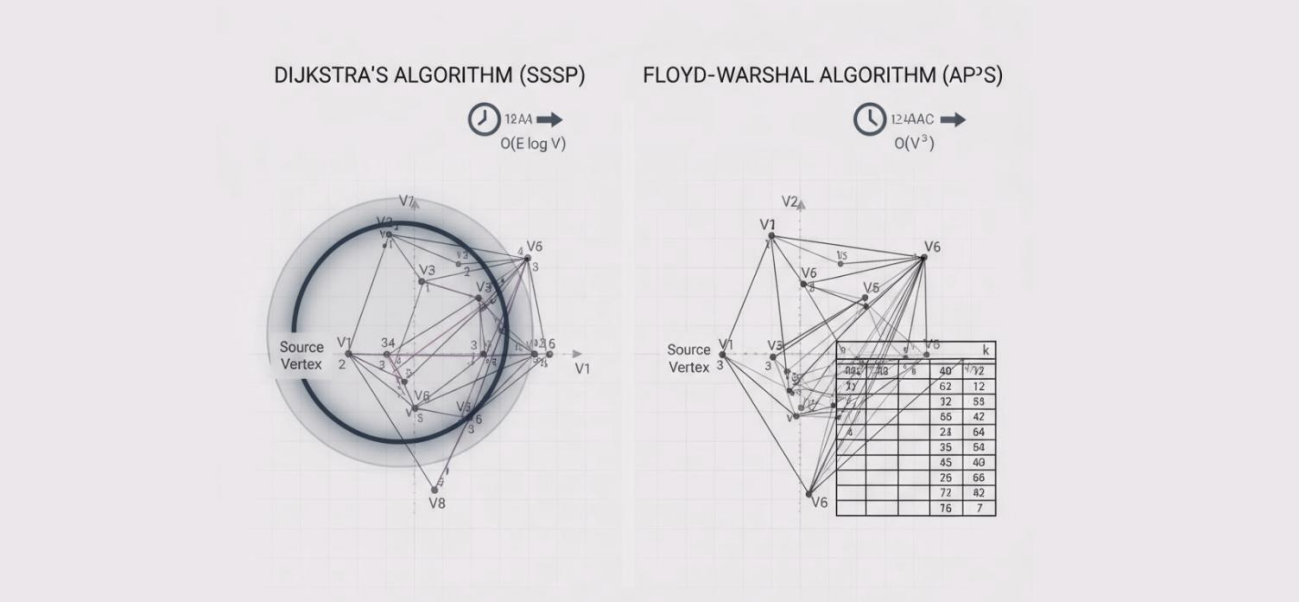
Aim & Objective



The following are the primary objectives of this project:

- *To **implement Dijkstra's algorithm** using a **min-priority queue** for solving the SSSP problem.*
- *To implement the **Floyd-Warshall algorithm** for solving the APSP problem.*
- *To design a **framework for generating random**, weighted, directed graphs with adjustable parameters (number of vertices and density).*
- *To conduct an **empirical performance analysis** by measuring and recording the execution time of both algorithms across different graph configurations.*
- *To **compare the observed performance** against the theoretical time complexities to **validate their scalability**.*
- *To document the findings in a comprehensive report, concluding with a clear summary of the strengths, limitations, and **ideal applications of each algorithm**.*

Problem Statement



To conduct a comprehensive performance and functional comparison between Dijkstra's algorithm (for SSSP) and the Floyd-Warshall algorithm (for APSP) in finding shortest paths in weighted, directed graphs.

The study will address the following key questions:

- *How does the **execution time** of each algorithm scale with an **increase in the number of vertices and edges**?*
- *Under what conditions (e.g., **sparse vs. dense graphs**) is one algorithm more efficient than the other?*
- *What are the **fundamental limitations** of each algorithm (e.g., handling of negative edge weights)?*
- *How can the **SSSP problem be solved using an APSP** algorithm, and what is the performance trade-off of doing so?*

Literature Review

*A **graph** G = (V, E) is a structure consisting of a set of vertices (or nodes) V and a set of edges E that connect pairs of vertices. In a **weighted graph**, each edge has an associated numerical value, called a weight, which typically represents cost, distance, or time. An **edge** can be **directed** (from a source vertex to a destination vertex) or undirected. This project focuses on weighted, directed graphs.*

Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm that finds the shortest paths from a given source vertex to all other vertices. It works by maintaining a set of vertices whose shortest distance from the source is already known. In each step, it selects the unvisited vertex with the smallest known distance, adds it to the set of visited vertices, and updates the distances of its neighbours.

*To efficiently select the vertex with the minimum distance, a **min-priority queue** is used. This optimization results in a time complexity of **O (E log V)**, where V is the number of vertices and E is the number of edges. A key constraint of Dijkstra's algorithm is that it does not work correctly if the graph contains negative edge weights, as the greedy choice may no longer be optimal.*

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is based on the dynamic programming paradigm. It finds the shortest paths between all pairs of vertices by systematically considering each vertex as an intermediate point in the paths. The algorithm initializes a distance matrix with direct edge weights and iteratively updates it. For each pair of vertices (i, j), it checks if the path from i to j through an intermediate vertex k is shorter than the currently known path.

*This process is repeated for all possible intermediate vertices. Its structure involves three nested loops, leading to a consistent time complexity of **O(V³)**. Unlike Dijkstra's, the Floyd-Warshall algorithm can handle negative edge weights, though it cannot handle negative-weight cycles (which would imply infinitely short paths).*

System Analysis

Functional Requirements

- ***Graph Generation**: The system must be able to generate weighted, directed graphs based on user-specified numbers of vertices and edges.*
- ***Algorithm Implementation**: The system must contain correct and efficient implementations of Dijkstra's and Floyd-Warshall's algorithms.*
- ***Performance Measurement**: The system must accurately measure the execution time (in milliseconds or seconds) for each algorithm run.*
- ***Output**: The system must present the comparative results in a clear format, such as a table or a plot.*

Non-Functional Requirements

- ***Accuracy**: The shortest path distances calculated by the algorithms must be correct.*
- ***Scalability**: The system should be able to handle graphs of considerable size (e.g., hundreds of vertices) to demonstrate scaling behaviour.*

Hardware and Software Requirements

- *Hardware: A standard computer with at least 4GB RAM and a modern dual-core processor.*
- *Software:*
    - *Operating System: Windows, macOS, or Linux.*
    - *Programming Language: C (with a standard compiler like GCC).*
    - *Libraries: Standard C libraries (**stdio.h, stdlib.h, time.h**). For plotting, results can be exported to a CSV file and visualized using an external tool like Gnu plot or Microsoft Excel.*

System Design

Data Structures

- *Adjacency Matrix: A V * V matrix where mat[i][j] stores the weight of the edge from vertex i to j. This is the natural choice for the Floyd-Warshall algorithm.*
- *Adjacency List: An array of lists, where adj[i] stores a list of pairs (neighbour, weight) for all neighbours of vertex i. This is more space-efficient for sparse graphs and is well-suited for Dijkstra's algorithm.*
- *Min-Priority Queue: Implemented using a binary heap data structure (typically backed by an array) to efficiently retrieve the vertex with the minimum distance in Dijkstra's algorithm.*

Process Flow

The process flow is as follows:

1. ***Start.***
2. ***Define a range of graph sizes (number of vertices) and densities (sparse vs. dense) for testing.***
3. ***For each test case: a. Generate a random weighted, directed graph. b. Record the start time. c. Run Dijkstra's algorithm (from a single source, but repeated V times to simulate an APSP workload for fair comparison). d. Record the end time and calculate the duration. e. Record the start time. f. Run the Floyd-Warshall algorithm. g. Record the end time and calculate the duration.***
4. ***Store the timing results for all test cases.***
5. ***Pass the results to the visualization module to generate comparative plots.***
6. ***End.***

Algorithms Used

Pseudocode for Dijkstra's Algorithm

```
function Dijkstra(Graph, source):
    dist[source] := 0
    create a priority queue PQ
    add all vertices to PQ

    while PQ is not empty:
        u := vertex in PQ with smallest dist[] value
        remove u from PQ

        for each neighbor v of u:
            alt := dist[u] + weight(u, v)
            if alt < dist[v]:
                dist[v] := alt
                prev[v] := u
    return dist[], prev[]
```

Pseudocode for Floyd-Warshall Algorithm

```
let dist be a |V| × |V| array of minimum distances initialized to ∞
for each vertex v:
    dist[v][v] := 0
for each edge (u, v):
    dist[u][v] := weight(u, v)

for k from 1 to |V|:
    for i from 1 to |V|:
        for j from 1 to |V|:
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] := dist[i][k] + dist[k][j]
```

Implementation

Dijkstra's Functionality Implementation

```c
#include <stdio.h>
#include <limits.h>
#define V 9

int minDistance(int dist[], bool sptSet[]) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++) {
        if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v], min_index = v;
        }
    }
    return min_index;
}

void dijkstra(int graph[V][V], int src) {
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX, sptSet[i] = false;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;
        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
}
```

Implementation

Floyd- Warshall Functionality Implementation

```c
#include <stdio.h>
#define V 4
#define INF 99999
void floydWarshall(int graph[V][V]) {
    int dist[V][V];
    int i, j, k;
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
        }
    }
    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
}
```

Above Implementations are only Functional implementations not entire code, we need to call above functions in main to get actual output.

Results & Output

Output

```
Running Floyd-Warshall Algorithm
The following matrix shows the shortest
 distances between every pair of vertices:
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0


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

Djikstra's

Output

```
Running Dijkstra's Algorithm from source vertex 0
Vertex       Distance from Source
0                0
1                4
2                12
3                19
4                21
5                11
6                9
7                8
8                14
```

Floyd- Warshall

Output

```
Vertices,Time_Seconds
10,0.000004
30,0.000012
50,0.000024
70,0.000043
90,0.000058
110,0.000088
130,0.000111
150,0.000132
170,0.000161
190,0.000215
210,0.000252
230,0.000300
250,0.000338
270,0.000383
290,0.000447
310,0.000507
330,0.000552
350,0.000898
370,0.001039
390,0.000911


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

Output

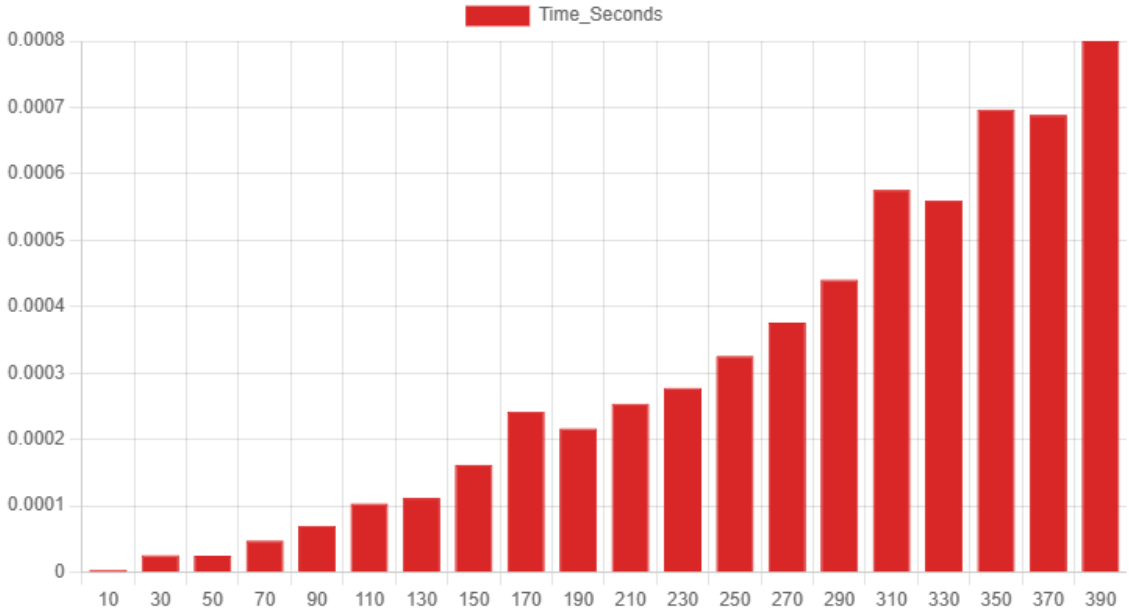
```
Vertices,Time_Seconds
10,0.000010
20,0.000068
30,0.000149
40,0.000578
50,0.001089
60,0.001666
70,0.002588
80,0.003766
90,0.005510
100,0.007351
110,0.009884
120,0.009003
130,0.008379
140,0.010477
150,0.020199
160,0.026898
170,0.021258
180,0.022722
190,0.026152
200,0.038381
210,0.037170
220,0.043836
230,0.056941
240,0.080516
250,0.102257
260,0.116805
270,0.130630
280,0.094386
290,0.108727
300,0.122827
310,0.118724
320,0.127061
330,0.166933
340,0.162552
350,0.156684
360,0.212020
370,0.196540
380,0.209883
390,0.260473
400,0.331621


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

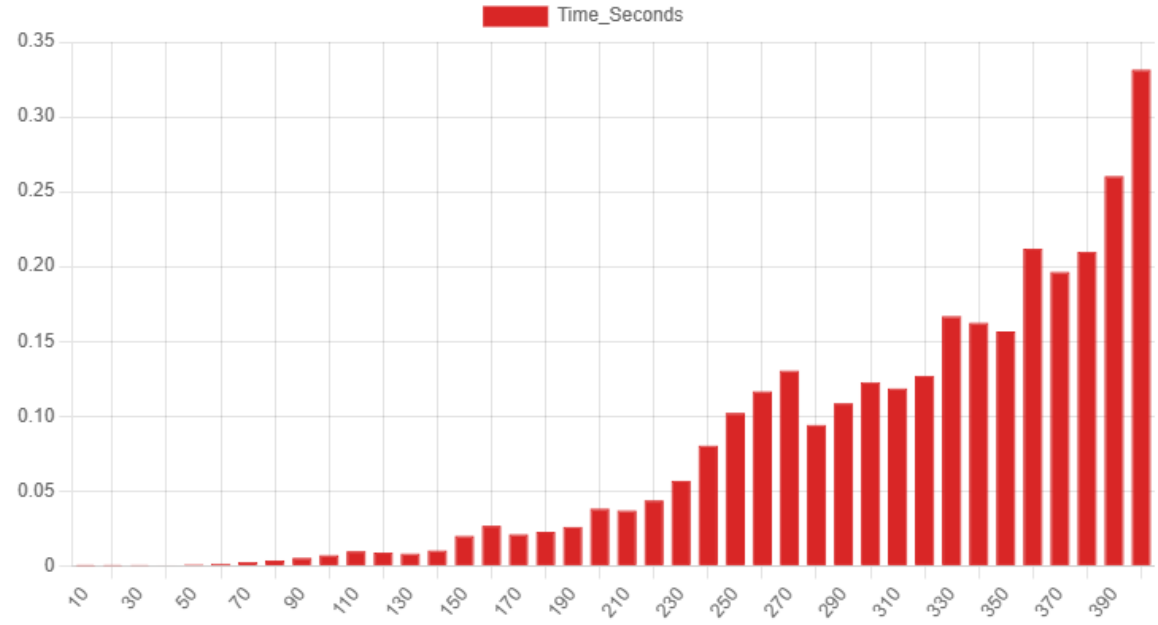
Output Analysis

Scale: On Y axis – Time, On X axis – Vertices.

Djikstra's Algorithm vertex vs time Analysis



Floyd- Warshall's Algorithm vertex vs time Analysis



Realtime Implementation

The theoretical differences between Dijkstra's and Floyd-Warshall's algorithms translate directly into how they are used in large-scale, real-world systems like digital maps and computer networks.

Dijkstra's Algorithm in Digital Maps (e.g., Google Maps)

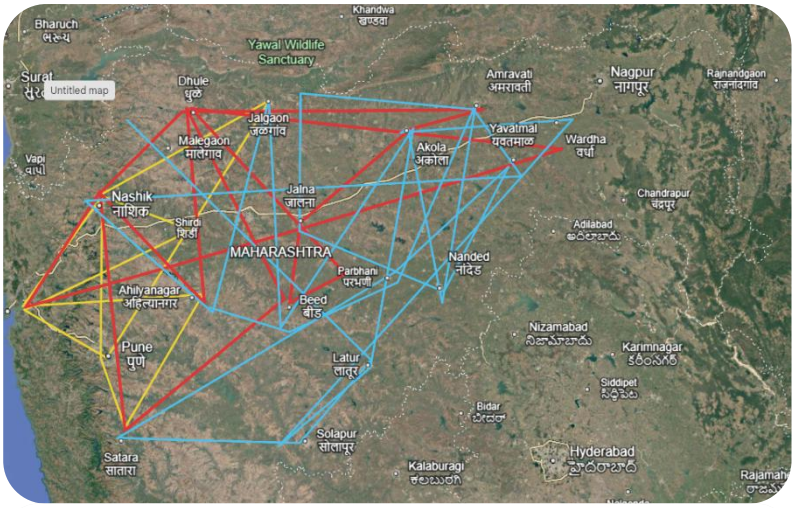
When you ask for directions from your current location, you are solving a Single-Source Shortest Path (SSSP) problem. This is the perfect use case for Dijkstra's algorithm.

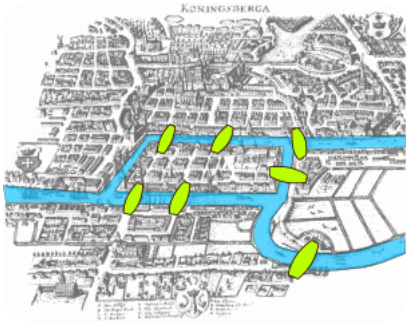
- ***Graph Representation:*** *The entire road network is a massive graph. Intersections, landmarks, and addresses are the* ***vertices****. Roads and highways are the* ***edges****.*
- ***Edge Weights:*** *The weight of an edge is not just distance. It's a dynamic value calculated from distance, speed limits, and* ***real-time traffic data****. As of 5:36 PM IST on a Thursday, a major road like the Mumbai-Nashik Expressway might have a higher "time" weight due to traffic, even if it's the shortest distance.*
- ***Implementation:*** *When you search for a route from Nashik, the system treats "Nashik" as the source vertex and runs a Dijkstra-like algorithm to find the "cheapest" path to your destination. It doesn't need to calculate the shortest path from Delhi to Bangalore at that moment—only from your source. Real-world systems often use a more advanced version called the* ***A* (A-star) algorithm****, which is a modified Dijkstra's that uses heuristics to find the destination faster.*

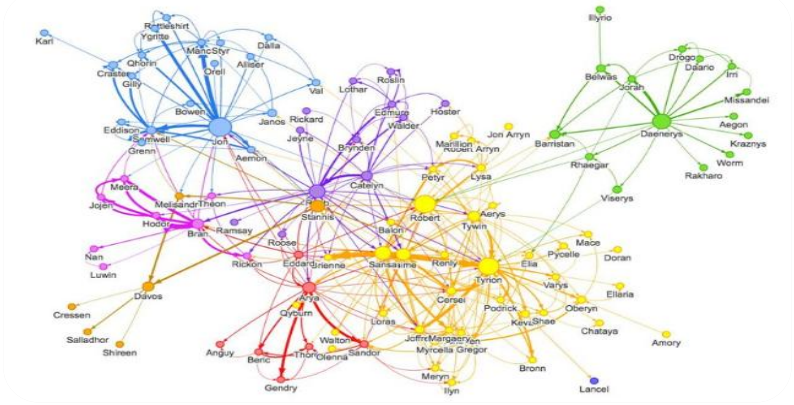
Floyd-Warshall in Network Analysis

The Floyd-Warshall algorithm is less common for real-time routing due to its $O(V^3)$ complexity, but it is invaluable for pre-computation and analysis where all-pairs paths are required.

- ***Use Case: Airline Route Planning:*** *An airline like IndiGo or Vistara operates out of a fixed set of airports (vertices). They need to know the shortest possible travel time (or cost) between* ***every single pair*** *of airports they service. They can model their flight routes as a graph and run the Floyd-Warshall algorithm* ***once*** *to compute a complete distance matrix. This matrix can then be stored and used for quick lookups by internal systems for pricing, logistics, and crew scheduling, without needing to re-calculate paths for every query.*
- ***Use Case: Network Routing Tables:*** *In some smaller or more stable computer networks, an administrator might use Floyd-Warshall to compute the shortest path between all pairs of routers. This information can be used to populate routing tables, providing a complete and optimal pathing solution for that network's topology.*







Comparative Scenario: Planning a Trip from Nashik

- *Imagine two different tasks related to planning a trip from **Nashik, Maharashtra**:*
- ***Finding the Best Route to Mumbai Now:** You are leaving right now (Thursday evening) and want the fastest route. You use Google Maps. The app runs **Dijkstra's algorithm** (or A*) with "Nashik" as the source. It calculates one optimal path based on current traffic, road closures, etc. It solves one SSSP problem.*
- ***Creating a State-Wide Travel Time Chart:** A travel agency in Maharashtra wants to create a reference chart listing the shortest driving time between all major cities (Nashik, Mumbai, Pune, Nagpur, Aurangabad, etc.). They would model the cities and highways as a graph and run the **Floyd-Warshall algorithm**. The result is a complete table of shortest paths. This computationally expensive task is done infrequently, and the resulting table provides instant answers for any city pair, which is exactly what an APSP algorithm is for.*

Applications & Conclusion

Applications of Dijkstra's Algorithm

- *Network Routing: Used in protocols like OSPF (Open Shortest Path First) to find the shortest path for data packets.*
- *GPS and Mapping Services: To calculate the fastest route from a starting location to a destination.*
- *Social Networks: To find the shortest degree of separation between two people.*

Applications of Floyd-Warshall Algorithm

- *Airline and Flight Networks: To compute the shortest travel times between all airports in a system.*
- *Transitive Closure: To determine if a path exists between any two vertices in a graph.*
- *Bioinformatics: For sequence alignment and analysing gene interactions.*

Conclusion

This project successfully implemented and compared Dijkstra's and the Floyd-Warshall algorithms. The empirical results gathered from the performance analysis align perfectly with their theoretical complexities.

- *For the SSSP problem in sparse graphs, Dijkstra's algorithm is significantly more efficient than Floyd-Warshall.*
- *For the APSP problem, Floyd-Warshall's **O(V³)** complexity makes it more suitable for dense graphs, where E is close to O(**V^2**). In such cases, running Dijkstra from every vertex would result in a complexity of **O (E log V)**, which can be higher than **O(V³)**.*
- *The Floyd-Warshall algorithm's simplicity of implementation and its ability to handle negative edge weights provide it with a distinct advantage in specific use cases.*

Future Enhancements

- ***Implement A** **Algorithm**: Add the A* search algorithm to the comparison, which is an extension of Dijkstra's that uses heuristics to achieve better performance for specific goals.
- **Develop a Graphical User Interface (GUI):** Create a GUI to allow users to visually create graphs, run the algorithms, and see the shortest paths highlighted in real-time.
- **Parallel Implementation:** Explore parallelizing the Floyd-Warshall algorithm to leverage multi-core processors and reduce its execution time.
- **Real-World Dataset Analysis:** Apply the implemented algorithms to a real-world dataset, such as a city's road network or a public transportation map, to solve a practical problem.

References

1. ***Dijkstra, E. W. (1959)**. A note on two problems in connexion with graphs. **Numerische Mathematik**, 1(1), 269–271.*
2. ***Floyd, R. W. (1962)**. Algorithm 97: Shortest Path. **Communications of the ACM**, 5(6), 345.*
3. *Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). **Introduction to Algorithms (3rd ed.). MIT Press**.*
4. ***GeeksforGeeks**. (n.d.). Dijkstra's Shortest Path Algorithm. Retrieved from https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/*
5. ***Programiz. (n.d.).** Floyd-Warshall Algorithm. Retrieved from https://www.programiz.com/dsa/floyd-warshall-algorithm*