PROJECT TITLE:

“Tour Guide Planner: Optimizing Travel Routes For Efficient Planning”

DESCRIPTION:

This project involves building a system to optimize travel routes for users based on time, distance, or cost. The system accepts multiple destination points and identifies the most efficient travel path from a starting location to all other locations. It aims to enhance travel planning by helping users save time and money while visiting various destinations.

SOLUTION METHODOLOGY:

The problem is approached using concepts from **graph theory**. Each location is treated as a node in a graph, and direct routes between locations are modeled as weighted edges. The weight of an edge represents the travel cost, distance, or time between two locations.

To solve the problem:

1. **Graph Construction:**
    - Each location is mapped to a node.
    - A list of connections (edges) is created using travel cost/time.
2. **Shortest Path Algorithm:**
    - **Dijkstra's Algorithm** is used to calculate the shortest path from the starting location to every other location.
    - A priority queue (min-heap) ensures efficient selection of the next minimum-distance node.
3. **Output Generation:**
    - Once the distances are calculated, the system displays the cost/time to reach every destination from the source.
    - This helps users select the best route plan from a given starting point.

APPROACH:

To find the most **efficient travel route** (with minimum cost/time/distance) between a starting location and multiple destinations using a graph-based algorithm.

Step-by-Step Approach:

1. Problem Modeling Using Graph Theory

- **Locations** are modeled as **nodes** (also called vertices).
- **Travel routes** between locations are modeled as **edges** with associated **weights**.
- The **weight** represents the cost, distance, or time to travel between two places.

This makes the problem a **Shortest Path Problem in a Weighted Graph**.

2. Data Structures Used

- `unordered_map<string, int>` – maps location names to indices.

- `vector<string>` – stores location names for reference.
- `vector<vector<Edge>>` – an **adjacency list** representing the travel graph.
- `priority_queue` – used in Dijkstra's algorithm for fast retrieval of the next closest location.
- `vector<int> dist` – stores the **minimum distance** from the source to each node.

3. Dijkstra's Algorithm: Finding the Shortest Path

We use **Dijkstra's algorithm**, a classical algorithm to compute the shortest distance from a single source node to all other nodes in a graph with **non-negative edge weights**.

How it works:

- Initialize all distances as infinity (`INF`).
- Set distance of source to 0.
- Use a **min-heap** (priority queue) to always select the node with the **smallest tentative distance**.
- For every neighbor of the current node, if the path through the current node is shorter, update the neighbor's distance.
- Repeat until all nodes are processed.

This ensures:

- The shortest path is found for every reachable location from the source.
- The algorithm runs efficiently in O((V + E) log V) time using a priority queue.

4. Route Optimization

Once shortest distances from the source are computed:

- Print the optimized travel cost from the starting point to every other destination.
- Any destination with `INF` distance is marked as **unreachable**.

This helps the user:

- Prioritize nearby or cheaper destinations.
- Avoid costly or unreachable routes.
- Build an efficient travel itinerary.

CODE :

```
#include <bits/stdc++.h>
using namespace std;

const int INF = 1e9;

struct Edge {
    int to, cost;
};

vector<int> dijkstra(int start, const vector<vector<Edge>>& graph) {
    int n = graph.size();
    vector<int> dist(n, INF);
    dist[start] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.push({0, start});

    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();

        if (d > dist[u]) continue;

        for (auto& edge : graph[u]) {
            int v = edge.to, cost = edge.cost;
            if (dist[u] + cost < dist[v]) {
                dist[v] = dist[u] + cost;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}

int main() {
    int n, m;
    cout << "Enter number of locations and connections: ";
    cin >> n >> m;

    vector<string> locationNames(n);
    unordered_map<string, int> nameToIndex;

    cout << "Enter location names:\n";
    for (int i = 0; i < n; ++i) {
        cin >> locationNames[i];
        nameToIndex[locationNames[i]] = i;
    }

    vector<vector<Edge>> graph(n);

    cout << "Enter connections (from to cost):\n";
    for (int i = 0; i < m; ++i) {
        string from, to;
        int cost;
        cin >> from >> to >> cost;
        int u = nameToIndex[from], v = nameToIndex[to];
        graph[u].push_back({v, cost});
        graph[v].push_back({u, cost});
    }

    cout << "Enter starting location: ";
    string startLoc;
    cin >> startLoc;
    int start = nameToIndex[startLoc];

    vector<int> dist = dijkstra(start, graph);

    cout << "\nOptimized Travel Route (from " << startLoc << "):\n";
    for (int i = 0; i < n; ++i) {
        if (i == start) continue;
        cout << "To " << locationNames[i] << ": " << (dist[i] == INF ?
"Unreachable" : to_string(dist[i])) << " units\n";
```

```
    }

    return 0;
}
```

INPUT:

```
5 6                          → 5 locations, 6 connections
Locations: A B C D E
Edges:
A B 4
A C 2
B C 1
C D 7
B E 5
D E 2
Starting location: A
```

Graph Representation:

We use a 0-indexed system:

```
0 - A
1 - B
2 - C
3 - D
4 - E
```

Adjacency List:

```
A (0): B(1, 4), C(2, 2)
B (1): A(0, 4), C(2, 1), E(4, 5)
C (2): A(0, 2), B(1, 1), D(3, 7)
D (3): C(2, 7), E(4, 2)
E (4): B(1, 5), D(3, 2)
```

Dijkstra's Algorithm Starting at Node A (0):

Initial State:

```
Distance[] = [0, INF, INF, INF, INF]
Priority Queue = [(0, A)]
```

Iteration 1: Node A (0)

- Neighbors:
  - B (1): cost = 0 + 4 = 4 → Update
  - C (2): cost = 0 + 2 = 2 → Update

```
Distance[] = [0, 4, 2, INF, INF]
Queue = [(2, C), (4, B)]
```

Iteration 2: Node C (2)

- Neighbors:
    - A already visited
    - B: 2 + 1 = 3 < 4 → Update B
    - D: 2 + 7 = 9 → Update

```
Distance[] = [0, 3, 2, 9, INF]
Queue = [(3, B), (4, B), (9, D)]
```

Iteration 3: Node B (1)

- Neighbors:
    - E: 3 + 5 = 8 → Update
    - A and C already optimal

```
Distance[] = [0, 3, 2, 9, 8]
Queue = [(4, B), (9, D), (8, E)]
```

Iteration 4: Node B again (already visited with shorter path) → skip

Iteration 5: Node E (4)

- Neighbor:
    - D: 8 + 2 = 10 > 9 → no update

```
Distance[] = [0, 3, 2, 9, 8]
Queue = [(9, D)]
```

Iteration 6: Node D (3) → already optimal

Queue empty → Done ✅

Final Shortest Distances from A:

Location	Distance
A	0
B	3
C	2
D	9
E	8

OUTPUT:

```
Optimized Travel Route (from A):
To B: 3 units
To C: 2 units
To D: 9 units
To E: 8 units
```

ENHANCEMENTS POSSIBLE:

This basic approach can be extended to:

- Handle **round trips** (TSP approximation).
- Include **multi-criteria optimization** (e.g., time + cost).
- Integrate **real GPS/maps APIs**.
- Show **complete paths**, not just cost (by tracking parent nodes).

CONCLUSION:

This project effectively demonstrates how graph algorithms like Dijkstra's can be applied to real-world travel planning. It provides a practical method for generating efficient travel routes and can be extended to incorporate additional constraints like rest stops, fuel limits, or round-trip planning.