	Pimpri Chinchwad Education Trust's **Pimpri Chinchwad College of Engineering (PCCoE)** (An Autonomous Institute) Affiliated to Savitribai Phule Pune University (SPPU) ISO 21001:2018 Certified by TUV	

Department: Information Technology **Academic Year**: 2025-26 **Semester:** V

DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY

Name : Rahul Dhanaji Hile PRN : 123B1F031

ASSIGNMENT NO. 4

Aim:

To design and implement an intelligent traffic management system that finds the shortest path for an ambulance from a source location to the nearest hospital in a dynamic urban environment.

Objective:

- To model the city's road network as a weighted graph.
- To implement Dijkstra's algorithm to calculate the shortest travel time from an ambulance's starting point to all available hospitals.
- To ensure the system can handle real-time updates to road travel times (edge weights) due to changing traffic conditions.
- To identify the optimal route to the nearest hospital and suggest it for navigation.

Problem Statement:

Develop a system to determine the fastest route for an ambulance through a city's road network. The network is represented as a graph where intersections are nodes and roads are weighted edges, with weights corresponding to real-time travel minutes. The system must use Dijkstra's algorithm to find the shortest path from the ambulance's current location (Source) to various hospital locations (Destinations). It must also be capable of dynamically adjusting the recommended path in response to live changes in traffic congestion, thereby minimizing emergency response times.

Outcomes:

1. Apply graph data structures to model and solve a real-world logistics and routing problem.
2. Successfully implement Dijkstra's algorithm to find the single-source shortest path in a weighted graph.
3. Develop a solution that can adapt to dynamic data changes, a key requirement for real-time systems.

Theory:

1. Graph Representation of a City Road Network - A Digital Map

Think of a graph as a simplified digital map, perfect for a computer to understand. It strips away all the complex details of a real map and focuses only on what's important for navigation: intersections and the roads connecting them.

- **Nodes (Vertices):** Where Roads Meet In our system, every important intersection or junction in the city becomes a 'node'. A node is just a point on our digital map. We also create special nodes for the ambulance's starting location and for every hospital. For example, if we consider the "Chafekar Chowk" in Chinchwad, that would be one node, and the "Aditya Birla Hospital" would be another.
- **Edges:** The Roads Themselves An 'edge' is the line that connects two nodes. In our case, an edge represents a road or a street. For instance, the road connecting Chafekar Chowk to Dange Chowk would be an edge. If a road is one-way, we use a directed edge (it has an arrow pointing in the direction of traffic). If it's a two-way street, it can be represented as an undirected edge or two separate directed edges going in opposite directions.
- **Weights:** How Long a Road Takes This is the most important part. Every road (edge) is given a number, called a 'weight'. For our system, this weight is the number of minutes it takes to drive down that road right now. A clear road like the Mumbai-Pune Expressway at 3 AM might have a low weight. However, the same road during peak evening traffic might have a very high weight. This number isn't fixed; it changes throughout the day based on live traffic data. This is what makes our system "smart" and dynamic.

2. Dijkstra's Algorithm - The Ultimate Route Planner

Dijkstra's Algorithm is the brain of our operation. It's a famous and reliable method for finding the absolute shortest path from a single starting point to every other point on a map. It works by systematically exploring the graph and is guaranteed to find the best route as long as the travel times (weights) are not negative, which is always true in our scenario.

Here's a simple way to understand how it works, step-by-step:

- **Initialization: Getting Ready**
    - **The Distances List:** The algorithm starts by making a list of every intersection (node) in the city. It assumes the travel time to every single one is infinity (∞). This sounds strange, but it just means "we haven't found a path there yet." The only exception is the ambulance's starting location, which has a distance of 0 (it takes 0 minutes to get where you already are).
    - **The "To-Do" List (Priority Queue):** The algorithm creates a special, smart "to-do" list. Its job is to always keep track of which intersection to check next. It's a *priority list*, which means it automatically puts the intersection that is currently closest to the start at the very top. This is the key to its efficiency—it always explores the most promising option first.
    - **The "Visited" List:** This is a simple checklist. Once the algorithm is 100% certain it has found the fastest possible route to an intersection, it gets checked off. We won't look at it again.
- **Iteration: The Main Loop** The algorithm now repeats the same process over and over until the "to-do" list is empty.
    - **Step A: Pick the Closest Node:** The algorithm looks at its "to-do" list and picks the intersection at the very top—the one that is currently the fastest to get to. Let's call this current_intersection.

- **Step B: Look at Its Neighbors:** From the current_intersection, the algorithm looks at all the intersections that are directly connected to it by a road (its neighbors).
- **Step C: Ask a Key Question:** For each neighbor, it asks: "**Is it faster to get to you through the current_intersection?**"
  - For example: Imagine the fastest known time to current_intersection is 10 minutes. The road from there to its neighbor, Intersection_B, takes 3 minutes. This means we've just found a path to Intersection_B that takes a total of 13 minutes (10 + 3).
  - If the old record for reaching Intersection_B was 20 minutes, we've just found a much better route! The algorithm updates its main distances list: the new fastest time to Intersection_B is now 13 minutes. It then adds Intersection_B to the "to-do" list so it can be explored later. If the old time was 12 minutes, we would ignore this new path because it's slower.
- **Step D: Check it Off:** Once the algorithm has checked all the neighbors of current_intersection, it puts it on the "visited" list. The shortest path to it is now locked in and finalized.

- **Termination: The End Result** When the "to-do" list is finally empty, the process stops. The distances list now contains the final, shortest possible travel time from the ambulance's starting point to every single intersection in the city!

3. Handling Dynamic Edge Weights - Adapting to Traffic

The real world is messy. A road that was clear a minute ago might now have a traffic jam due to an accident or rush hour. This is where our system's dynamic nature is crucial.

When a traffic sensor or GPS data feed reports new information, the weight of the corresponding edge on our graph changes. A road that took 5 minutes might now take 20. A route that was optimal is now slow.

Our system continuously listens for these updates. If a road's travel time changes significantly, the system doesn't just stick to the old plan. It immediately re-runs Dijkstra's algorithm from the ambulance's *current* position, but using the new, updated map with the new weights. This might generate a completely different route, perhaps directing the ambulance down a side street it would have otherwise ignored. This ensures the ambulance is always following the best possible path based on what's happening on the road *right now*.

4. Algorithm for Finding the Shortest Path - Putting It All Together

Here is the entire process from start to finish in simple steps:

1. **Build the Digital Map:** The system first constructs the graph—defining all the intersections (nodes) and roads (edges), and assigning the latest travel times as weights.
2. **Run the Route Planner:** It executes Dijkstra's algorithm, with the ambulance's current location as the starting point.
3. **Find the Nearest Hospital:** After the algorithm is done, the system has a list of the shortest travel times to *every* node. It simply looks at the times for all the hospital nodes and picks the one with the lowest number.
4. **Generate Directions:** To create a navigable route, the algorithm works backward from the chosen hospital, using the path information it saved during the process, to build the step-by-step list of roads the ambulance driver needs to take.

5. **Watch and Repeat:** The system never stops listening for traffic updates. If a major change occurs, it instantly goes back to Step 2 to recalculate the route, ensuring constant optimization.

Code:

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

void dijkstra(int source, vector<vector<pair<int,int>>> &graph, vector<int> &dist) {
    int V = graph.size();
    dist.assign(V, INT_MAX);
    dist[source] = 0;

    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
    pq.push({0, source});

    while (!pq.empty()) {
        int u = pq.top().second;
        int d = pq.top().first;
        pq.pop();

        if (d > dist[u]) continue;

        for (auto &edge : graph[u]) {
            int v = edge.first;
            int w = edge.second;

            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
}

int main() {
    int V, E;
    cout << "Enter number of intersections (vertices): ";
    cin >> V;
    cout << "Enter number of roads (edges): ";
    cin >> E;

    vector<vector<pair<int,int>>> graph;
    graph.resize(V);  // FIX: Allocate memory for V vertices

    cout << "Enter edges (u v w):\n";
    for (int i = 0; i < E; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u].push_back({v, w});
        graph[v].push_back({u, w}); // undirected road
    }

    int source;
    cout << "Enter ambulance start location (source): ";
    cin >> source;

    int H;
```

```
    cout << "Enter number of hospitals: ";
    cin >> H;
    vector<int> hospitals(H);
    cout << "Enter hospital nodes: ";
    for (int i = 0; i < H; i++) {


        cin >> hospitals[i];
    }

    vector<int> dist;
    dijkstra(source, graph, dist);

    int minTime = INT_MAX, nearestHospital = -1;
    for (int h : hospitals) {
        if (dist[h] < minTime) {
            minTime = dist[h];
            nearestHospital = h;
        }
    }

    if (nearestHospital == -1)
        cout << "No hospital reachable.\n";
    else
        cout << "Nearest hospital is at node " << nearestHospital
             << " with travel time " << minTime << " minutes.\n";

    return 0;
}
```

Output:

```
Enter number of intersections (vertices): 6
Enter number of roads (edges): 7
Enter edges (u v w):
0 1 4
0 2 2
1 2 1
1 3 5
2 3 8
2 4 10
3 5 2
Enter ambulance start location (source): 0
Enter number of hospitals: 2
Enter hospital nodes: 4 5

Nearest hospital is at node 5 with travel time 10 minutes.
```

Questions:

Q1. What is the role of the Priority Queue and why is it essential for the algorithm's performance in a large city?

The **Priority Queue** helps the algorithm quickly select the most important task or item, like the area that needs relief the most. It always keeps the highest-priority elements ready to process, without searching through everything.

In a **large city**, where there are many locations and requests, using a priority queue is essential because it:

1. **Saves time** by efficiently finding the next best option.
2. **Improves performance** by reducing unnecessary checks.
3. **Ensures effectiveness** by sending help to the most urgent areas first.

Example:
If dozens of areas need supplies, the priority queue lets the algorithm instantly find the area with the greatest need and send aid immediately, instead of scanning all areas every time.
This makes the algorithm faster and more reliable when handling large, complex problems.

Q2. Why is Dijkstra's algorithm more suitable for this problem than an uninformed search algorithm like Breadth-First Search (BFS)?

Dijkstra's algorithm is more suitable because it finds the shortest or fastest path by considering the actual distance or cost between locations. It always chooses the path with the lowest total cost so far.
In contrast, an **uninformed search algorithm like BFS** treats all paths equally and explores all nearby options without considering their distance or cost. This can lead to unnecessary exploration and slower results.
For problems in a **large city**, where routes have different distances or costs (like traffic or road conditions), Dijkstra's algorithm:

1. **Finds the best path** by calculating the shortest or cheapest route.
2. **Avoids unnecessary steps**, saving time and resources.
3. **Works efficiently** even when there are many possible routes.

Feature	Dijkstra's Algorithm	Breadth-First Search (BFS)
How it works	It chooses the next location with the lowest total distance or cost.	It explores all nearby paths level by level, without considering distance.
Performance in large cities	Fast and efficient because it finds the shortest path based on real distances or costs.	Slow and inefficient because it checks unnecessary paths.
Use of information	Uses distance or cost information to guide the search.	Does not use any extra information about distance or cost.
Best for	Problems where paths have different lengths or costs, like flood relief or transportation.	Problems where all paths are equal, or when only the number of steps matters.
Result	Always finds the optimal (shortest or cheapest) path.	May find a path, but not necessarily the best one.

Example:
If two areas need help, but one is closer or faster to reach, Dijkstra's algorithm will prioritize that route. BFS would explore all nearby paths first, even if they are longer, making it slower and less efficient.
Thus, Dijkstra's algorithm ensures optimal and faster solutions in complex, real-world problems like flood relief in a large city.

Conclusion:

Dijkstra's algorithm provides a highly effective and reliable foundation for an intelligent traffic management system designed to save lives. By modeling the urban road network as a weighted graph and dynamically updating edge weights to reflect real-time traffic, the system can consistently identify the genuinely fastest route for an ambulance. This approach is not only computationally efficient but also adaptable, ensuring that emergency responders are always equipped with the optimal path, thereby minimizing response times and improving patient outcomes. The implementation successfully demonstrates the power of graph theory in solving critical, real-world logistical challenges.