	Pimpri Chinchwad Education Trust's **Pimpri Chinchwad College of Engineering**
Assignment No: 4	

1. **Problem Statement:**

   In a modern smart city infrastructure, efficient emergency response is critical to saving lives. One major challenge is ensuring that ambulances reach hospitals in the shortest possible time, especially during peak traffic conditions. The city's road network can be modeled as a graph, where intersections are represented as nodes and roads as edges with weights indicating real-time travel time based on current traffic congestion.

   The goal is to design and implement an intelligent traffic management system that dynamically computes the fastest route from an ambulance's current location (source node S) to the nearest hospital (destination node D) in the network. Due to ever-changing traffic conditions, edge weights must be updated in real-time, requiring the system to adapt and re-compute optimal routes as necessary.

2. **Course Objective:**
   2.1 To know the basics of computational complexity of various algorithms.
   2.2 To select appropriate algorithm design strategies to solve real-world problems.

3. **Course Outcome:**
   3.1 Analyze the asymptotic performance of algorithms
   3.2 Solve computational problems by applying suitable paradigms of Divide and Conquer/Greedy Methodologies

4. **Theory:**

   Problem Overview:
   Given a weighted graph with non-negative edge weights, the objective is to find the shortest path from a given source node S to all other nodes in the graph. The edges in the graph represent costs or distances, and the aim is to minimize the total distance (or cost) from the source node to each other node.The graph can be represented as an adjacency matrix or an adjacency list, where:

   G=(V,E) is the graph, where:
   - V is the set of vertices (nodes).
   - E is the set of edges (connections between nodes), with each edge having a weight.

   Greedy Algorithm:
   Dijkstra's algorithm is a greedy algorithm, meaning that it makes the locally optimal choice at each step with the hope that these local solutions lead to a globally optimal solution.
   The algorithm works as follows:
   1. Choose the node with the smallest tentative distance that hasn't been processed yet.
   2. Update the distances to its neighboring nodes.
   3. Repeat the process until the shortest path to all nodes has been determined.

Steps of the Algorithm:

1. **Initialize**:
    - Set the distance of the source node S to 0 (since the distance to itself is 0).
    - Set the distance to all other nodes to infinity (indicating that they are initially unreachable).
    - Mark all nodes as unvisited.

2. **Set of unvisited nodes**:
    - Create a priority queue (or min-heap) that stores nodes with their current shortest distance from the source node.
    - The priority queue ensures that we always extract the node with the smallest known distance.

3. **Relaxation (update distances)**:
    - While there are still unvisited nodes:
        - Extract the node u with the smallest distance from the priority queue.
        - For each neighbor v of node u, calculate the distance through u.
        - If this new distance is shorter than the current distance to v, update the distance of v and push it back into the priority queue.

4. **Termination**:
    - Once all nodes have been visited, the algorithm terminates, and the shortest distances from the source node to all other nodes are known.

5. **Path Reconstruction**:
    - If the path from the source node to each node is required, a parent array can be maintained to track which node was visited from which (this is done during the relaxation step).
    - By backtracking using this parent array, the shortest path to any node can be reconstructed.

Why Greedy Works:

1. **Greedy-choice property**:
    - Dijkstra's algorithm works because at each step, we make the greedy choice of selecting the node with the smallest tentative distance. This ensures that we are always expanding the shortest known path.
    - Once a node's shortest path is determined (i.e., it has been extracted from the priority queue), its value will never change. This guarantees correctness.

2. **Optimal substructure**:
    - The problem exhibits optimal substructure, meaning that once the shortest path to a node is known, it remains optimal and no further updates are required. This is because the shortest path to any node depends only on the shortest paths to its predecessor nodes.
    - If a node's shortest distance is ddd, then all the shortest paths to the neighbors of that node will be computed using this ddd, ensuring that the algorithm does not miss any optimal solutions.

Time Complexity:

1. **Initialization**:
    - Setting the initial distances and priority queue takes O(V), where V is the number of nodes.

2. **Priority Queue Operations**:
    - Extracting the minimum element from the priority queue and updating the distances to neighbors each take O(log V).
    - Each edge is processed once, and each update is done in logarithmic time due to the priority queue.
    - Hence, processing all the nodes and edges takes O((V+E)log V, where V is the number of vertices and E is the number of edges.

    Total Time Complexity:
    O((V+E)log V)
    This is efficient enough for large graphs, especially when using an adjacency list representation.

Space Complexity:

1. **Storage for Graph**:
    - The graph is stored using an adjacency list, which takes O(V+E) space.

2. **Priority Queue**:
    - The priority queue (min-heap) stores V nodes, and thus it requires O(V) space.

3. **Distance and Parent Arrays**:
    - The distance array (to store the shortest distance to each node) takes O(V) space.
    - The parent array (for path reconstruction) also takes O(V) space.

    Total Space Complexity:
    O(V+E)

5. **Implementation:**

```
BEGIN PROGRAM

  // ----------- INPUT GRAPH ------------
  INPUT V  ← number of intersections (vertices)
  CREATE TrafficNetwork city with V vertices

  INPUT E  ← number of roads (edges)
  PRINT "Enter edges (source destination weight):"
  FOR i = 1 TO E DO
     INPUT u, v, w
     city.addEdge(u, v, w)   // add bidirectional road
  END FOR

  // ----------- HOSPITAL LOCATIONS ------------
  INPUT numHospitals
  PRINT "Enter hospital locations:"
  FOR i = 1 TO numHospitals DO
     INPUT hospitalNode
     city.addHospital(hospitalNode)
```

```
    END FOR

    // ----------- AMBULANCE START ------------
    INPUT ambulanceLocation

    // ----------- INITIAL SHORTEST PATH ------------
    CALL city.shortestPathToNearestHospital(ambulanceLocation) → (distance, path)

    IF distance ≠ -1 THEN
        PRINT "Initial shortest travel time:" distance
        PRINT "Path:" path
    ELSE
        PRINT "No hospital reachable"
    END IF

    // ----------- REAL-TIME UPDATE ------------
    PRINT "Enter edge to update (u v newWeight):"
    INPUT u, v, newWeight
    city.updateEdgeWeight(u, v, newWeight)

    // ----------- RECOMPUTE PATH ------------
    CALL city.shortestPathToNearestHospital(ambulanceLocation) → (newDistance, newPath)

    IF newDistance ≠ -1 THEN
        PRINT "After update shortest travel time:" newDistance
        PRINT "Path:" newPath
    ELSE
        PRINT "No hospital reachable after update"
    END IF

END PROGRAM

// FUNCTIONS INSIDE TrafficNetwork CLASS

FUNCTION addEdge(u, v, w):
    ADD (v, w) to adjacency list of u
    ADD (u, w) to adjacency list of v    // because graph is undirected
END FUNCTION


FUNCTION updateEdgeWeight(u, v, newWeight):
    FOR each edge in adjacency list of u DO
        IF edge.to == v THEN
            edge.weight ← newWeight
        END IF
    END FOR
    FOR each edge in adjacency list of v DO
        IF edge.to == u THEN
            edge.weight ← newWeight
        END IF
    END FOR
END FUNCTION
```

```
FUNCTION addHospital(node):
   ADD node to hospital set
END FUNCTION


FUNCTION shortestPathToNearestHospital(source):

   CREATE array dist[V], initialized with ∞
   CREATE array parent[V], initialized with -1
   SET dist[source] = 0

   CREATE priority queue pq
   INSERT (0, source) into pq

   WHILE pq is not empty DO
      (curDist, u) ← REMOVE node with smallest distance from pq

      IF curDist > dist[u] THEN
         CONTINUE
      END IF

      IF u is in hospital set THEN
         path ← empty list
         v ← u
         WHILE v ≠ -1 DO
            ADD v to path
            v ← parent[v]
         END WHILE
         REVERSE path
         RETURN (dist[u], path)
      END IF

      FOR each neighbor v of u with weight w DO
         IF dist[u] + w < dist[v] THEN
            dist[v] ← dist[u] + w
            parent[v] ← u
            INSERT (dist[v], v) into pq
         END IF
      END FOR
   END WHILE

   RETURN (-1, empty path)  // No hospital reachable
END FUNCTION
```

6. Output:

```
Enter number of vertices (intersections): 6
Enter number of edges (roads): 9
Enter edges (source destination weight):
0 1 10
0 2 5
1 2 2
1 3 1
2 3 9
```

2 4 2
3 4 4
3 5 7
4 5 6
Enter number of hospitals: 1
Enter hospital locations:
5
Enter ambulance starting location: 0
Initial shortest travel time: 14
Path: 0 2 4 5
Enter edge to update (u v newWeight): 2 4 20
After update shortest travel time: 15
Path: 0 2 4 5

7. CODE:

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

void dijkstra(int source, vector<vector<pair<int, int>>> &graph, vector<int>
&dist, vector<int> &parent) {
    int V = graph.size();
    dist.assign(V, INT_MAX);
    parent.assign(V, -1);
    dist[source] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> 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;
                parent[v] = u;
                pq.push({dist[v], v});
            }
        }
    }
}
void printPath(int node, const vector<int> &parent) {
    vector<int> path;
    while (node != -1) {
        path.push_back(node);
        node = parent[node];
```

```
    }
    reverse(path.begin(), path.end());
    cout << "Optimal Path: ";
    for (int i = 0; i < path.size(); ++i) {
        cout << path[i];
        if (i < path.size() - 1) cout << " -> ";
    }
    cout << endl;
}

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(V);
    cout << "Enter roads (u v travel_time):\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});
    }
    char update;
    cout << "Do you want to update travel times due to traffic? (y/n): ";
    cin >> update;
    while (update == 'y' || update == 'Y') {
        int u, v, new_w;
        cout << "Enter road to update (u v new_travel_time): ";
        cin >> u >> v >> new_w;
        for (auto &p : graph[u]) {
            if (p.first == v) {
                p.second = new_w;
                break;
            }
        }
        for (auto &p : graph[v]) {
            if (p.first == u) {
                p.second = new_w;
                break;
            }
        }
        cout << "Update another road? (y/n): ";
        cin >> update;
    }
    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 node indices: ";
    for (int i = 0; i < H; i++) {
        cin >> hospitals[i];
    }
    vector<int> dist, parent;
    dijkstra(source, graph, dist, parent);

    int nearestHospital = -1, minTime = INT_MAX;
    for (int h : hospitals) {
        if (dist[h] < minTime) {
            minTime = dist[h];
            nearestHospital = h;
        }
    }
    if (nearestHospital == -1 || dist[nearestHospital] == INT_MAX) {
        cout << "No hospital reachable.\n";
    } else {
        cout << "\nNearest hospital is at node " << nearestHospital
             << " with estimated time " << minTime << " minutes.\n";
        printPath(nearestHospital, parent);
    }
    return 0;
}
```

8. OUTPUT:

Enter number of intersections (vertices): 6
Enter number of roads (edges): 7
Enter roads (u v travel_time):
0 1 4
0 2 2
1 2 5
1 3 10
2 4 3
4 3 4
3 5 11
Do you want to update travel times due to traffic? (y/n): y
Enter road to update (u v new_travel_time): 1 3 6
Update another road? (y/n): n
Enter ambulance start location (source): 0
Enter number of hospitals: 2
Enter hospital node indices: 3 5

Nearest hospital is at node 3 with estimated time 9 minutes.
Optimal Path: 0 -> 2 -> 4 -> 3

9. Conclusion:

The smart traffic management system uses Dijkstra’s algorithm to provide ambulances with the fastest route to hospitals, dynamically updating paths based on real-time traffic conditions. Scalable for large city networks and supported with visual navigation, it ensures minimal response time, improved emergency handling, and contributes to building safer, smarter cities.p