Develop the Smart City Road Navigation System
Theme: Trees, Graphs, and Advanced Algorithms

GitHub:

https://github.com/mayank24000/University_Assignments/tree/main/Ds_%20Assignments/Theory_Assignments/Theory_Assignment_4

Code:

```
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <stack>
#include <algorithm>
#include <climits>
#include <iomanip>
#include <map>
#include <set>

using namespace std;

class RoadNode {
public:
```

```
    int nodeID;
    string nodeName;
    string zoneDetails;

    RoadNode() {
        nodeID = -1;
        nodeName = "";
        zoneDetails = "";
    }

    RoadNode(int id, string name, string zone) {
        nodeID = id;
        nodeName = name;
        zoneDetails = zone;
    }

    void display() {
        cout << "Node " << nodeID << ": " << nodeName
             << " [" << zoneDetails << "]" << endl;
    }
};

// ==================== GRAPH REPRESENTATION ====================

class RoadNetwork {
private:
    int numNodes;
    vector<vector<int>> adjacencyMatrix;
```

```
    vector<vector<pair<int, int>>> adjacencyList; //
pair<destination, weight>
    map<int, RoadNode> nodes;

public:
    RoadNetwork() {
        numNodes = 0;
    }

    void initializeNetwork(int n) {
        numNodes = n;
        adjacencyMatrix.resize(n, vector<int>(n, INT_MAX));
        adjacencyList.resize(n);

        // Initialize diagonal elements to 0
        for(int i = 0; i < n; i++) {
            adjacencyMatrix[i][i] = 0;
        }
    }

    void addNode(RoadNode node) {
        nodes[node.nodeID] = node;
    }

    void addRoad(int from, int to, int distance) {
        if(from < numNodes && to < numNodes) {
            // Add to adjacency matrix
            adjacencyMatrix[from][to] = distance;
            adjacencyMatrix[to][from] = distance; // Undirected road
```

```
            // Add to adjacency list
            adjacencyList[from].push_back({to, distance});
            adjacencyList[to].push_back({from, distance}); // 
Undirected road
        }
    }

    void addDirectedRoad(int from, int to, int distance) {
        if(from < numNodes && to < numNodes) {
            // Add to adjacency matrix
            adjacencyMatrix[from][to] = distance;

            // Add to adjacency list
            adjacencyList[from].push_back({to, distance});
        }
    }

    void displayAdjacencyMatrix() {
        cout << "\n--- Adjacency Matrix Representation ---" << endl;
        cout << "    ";
        for(int i = 0; i < numNodes; i++) {
            cout << setw(8) << i;
        }
        cout << endl;

        for(int i = 0; i < numNodes; i++) {
            cout << setw(4) << i;
            for(int j = 0; j < numNodes; j++) {
```

```
                if(adjacencyMatrix[i][j] == INT_MAX) {
                    cout << setw(8) << "INF";
                } else {
                    cout << setw(8) << adjacencyMatrix[i][j];
                }
            }
            cout << endl;
        }
    }

    void displayAdjacencyList() {
        cout << "\n--- Adjacency List Representation ---" << endl;
        for(int i = 0; i < numNodes; i++) {
            cout << "Node " << i;
            if(nodes.count(i)) {
                cout << " (" << nodes[i].nodeName << ")";
            }
            cout << ": ";

            for(auto& edge : adjacencyList[i]) {
                cout << "[" << edge.first << ", " << edge.second <<
"km] ";
            }
            cout << endl;
        }
    }

    // ========== DIJKSTRA'S ALGORITHM ==========
```

```
    void dijkstraShortestPath(int start, int end) {
        if(start >= numNodes || end >= numNodes) {
            cout << "Invalid node IDs!" << endl;
            return;
        }

        vector<int> distance(numNodes, INT_MAX);
        vector<int> parent(numNodes, -1);
        vector<bool> visited(numNodes, false);

        distance[start] = 0;

        // Priority queue: pair<distance, node>
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
        pq.push({0, start});

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

            if(visited[u]) continue;
            visited[u] = true;

            // Check all adjacent nodes
            for(auto& edge : adjacencyList[u]) {
                int v = edge.first;
                int weight = edge.second;
```

```
                if(distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
                    distance[v] = distance[u] + weight;
                    parent[v] = u;
                    pq.push({distance[v], v});
                }
            }
        }

        // Display result
        cout << "\n--- Dijkstra's Shortest Path ---" << endl;
        cout << "From: " << start;
        if(nodes.count(start)) cout << " (" << nodes[start].nodeName << ")";
        cout << "\nTo: " << end;
        if(nodes.count(end)) cout << " (" << nodes[end].nodeName << ")";
        cout << endl;

        if(distance[end] == INT_MAX) {
            cout << "No path exists!" << endl;
            return;
        }

        cout << "Shortest Distance: " << distance[end] << " km" << endl;

        // Reconstruct path
        vector<int> path;
        int current = end;
```

```
        while(current != -1) {
            path.push_back(current);
            current = parent[current];
        }

        reverse(path.begin(), path.end());

        cout << "Path: ";
        for(int i = 0; i < path.size(); i++) {
            cout << path[i];
            if(nodes.count(path[i])) {
                cout << " (" << nodes[path[i]].nodeName << ")";
            }
            if(i < path.size() - 1) cout << " -> ";
        }
        cout << endl;
    }

    // ========== FLOYD-WARSHALL ALGORITHM ==========

    void floydWarshallAllPairs() {
        cout << "\n--- Floyd-Warshall All Pairs Shortest Path ---"
<< endl;

        // Create distance matrix
        vector<vector<int>> dist(numNodes, vector<int>(numNodes));

        // Initialize with adjacency matrix
        for(int i = 0; i < numNodes; i++) {
```

```
            for(int j = 0; j < numNodes; j++) {
                dist[i][j] = adjacencyMatrix[i][j];
            }
        }

        // Floyd-Warshall algorithm
        for(int k = 0; k < numNodes; k++) {
            for(int i = 0; i < numNodes; i++) {
                for(int j = 0; j < numNodes; j++) {
                    if(dist[i][k] != INT_MAX && dist[k][j] !=
INT_MAX &&
                       dist[i][k] + dist[k][j] < dist[i][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j];
                    }
                }
            }
        }

        // Display distance matrix
        cout << "\nShortest Distances Between All Pairs:" << endl;
        cout << "     ";
        for(int i = 0; i < numNodes; i++) {
            cout << setw(6) << i;
        }
        cout << endl;

        for(int i = 0; i < numNodes; i++) {
            cout << setw(5) << i;
            for(int j = 0; j < numNodes; j++) {
```

```
                if(dist[i][j] == INT_MAX) {
                    cout << setw(6) << "INF";
                } else {
                    cout << setw(6) << dist[i][j];
                }
            }
            cout << endl;
        }
    }

    // ========== PRIM'S ALGORITHM FOR MST ==========

    void primMST() {
        cout << "\n--- Prim's Algorithm for Minimum Spanning Tree ---" << endl;

        vector<bool> inMST(numNodes, false);
        vector<int> key(numNodes, INT_MAX);
        vector<int> parent(numNodes, -1);

        // Start from node 0
        key[0] = 0;

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

        int totalCost = 0;
```

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

            if(inMST[u]) continue;

            inMST[u] = true;
            totalCost += weight;

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

                if(!inMST[v] && w < key[v]) {
                    key[v] = w;
                    parent[v] = u;
                    pq.push({w, v});
                }
            }
        }

        cout << "\nMinimum Spanning Tree Edges (Prim's):" << endl;
        for(int i = 1; i < numNodes; i++) {
            if(parent[i] != -1) {
                cout << "Road: " << parent[i];
                if(nodes.count(parent[i])) cout << " (" <<
nodes[parent[i]].nodeName << ")";
                cout << " - " << i;
```

```
                if(nodes.count(i)) cout << " (" << nodes[i].nodeName
<< ")";
                cout << " | Cost: " << key[i] << " km" << endl;
            }
        }
        cout << "Total Cost of MST: " << totalCost << " km" << endl;
    }

    // ========== KRUSKAL'S ALGORITHM FOR MST ==========

    struct Edge {
        int src, dest, weight;
        bool operator<(const Edge& other) const {
            return weight < other.weight;
        }
    };

    int findParent(vector<int>& parent, int i) {
        if(parent[i] != i) {
            parent[i] = findParent(parent, parent[i]);
        }
        return parent[i];
    }

    void unionSets(vector<int>& parent, vector<int>& rank, int x,
int y) {
        int xroot = findParent(parent, x);
        int yroot = findParent(parent, y);
```

```
        if(rank[xroot] < rank[yroot]) {
            parent[xroot] = yroot;
        } else if(rank[xroot] > rank[yroot]) {
            parent[yroot] = xroot;
        } else {
            parent[yroot] = xroot;
            rank[xroot]++;
        }
    }

    void kruskalMST() {
        cout << "\n--- Kruskal's Algorithm for Minimum Spanning Tree
---" << endl;

        vector<Edge> edges;

        // Collect all edges
        for(int i = 0; i < numNodes; i++) {
            for(auto& edge : adjacencyList[i]) {
                if(i < edge.first) {  // Avoid duplicates in
undirected graph
                    edges.push_back({i, edge.first, edge.second});
                }
            }
        }

        // Sort edges by weight
        sort(edges.begin(), edges.end());
```

```
        vector<int> parent(numNodes);
        vector<int> rank(numNodes, 0);
        for(int i = 0; i < numNodes; i++) {
            parent[i] = i;
        }

        cout << "\nMinimum Spanning Tree Edges (Kruskal's):" <<
endl;
        int totalCost = 0;
        int edgesAdded = 0;

        for(const Edge& e : edges) {
            int x = findParent(parent, e.src);
            int y = findParent(parent, e.dest);

            if(x != y) {
                cout << "Road: " << e.src;
                if(nodes.count(e.src)) cout << " (" <<
nodes[e.src].nodeName << ")";
                cout << " - " << e.dest;
                if(nodes.count(e.dest)) cout << " (" <<
nodes[e.dest].nodeName << ")";
                cout << " | Cost: " << e.weight << " km" << endl;

                totalCost += e.weight;
                unionSets(parent, rank, x, y);
                edgesAdded++;

                if(edgesAdded == numNodes - 1) break;
            }
```

```
        }

        cout << "Total Cost of MST: " << totalCost << " km" << endl;
    }

    // ========== TOPOLOGICAL SORT ==========

    void topologicalSort() {
        cout << "\n--- Topological Sort for Construction Priority --
-" << endl;

        // Calculate in-degree for each node
        vector<int> inDegree(numNodes, 0);
        for(int i = 0; i < numNodes; i++) {
            for(auto& edge : adjacencyList[i]) {
                inDegree[edge.first]++;
            }
        }

        queue<int> q;
        for(int i = 0; i < numNodes; i++) {
            if(inDegree[i] == 0) {
                q.push(i);
            }
        }

        vector<int> topOrder;

        while(!q.empty()) {
```

```
            int u = q.front();
            q.pop();
            topOrder.push_back(u);

            for(auto& edge : adjacencyList[u]) {
                inDegree[edge.first]--;
                if(inDegree[edge.first] == 0) {
                    q.push(edge.first);
                }
            }
        }

        if(topOrder.size() != numNodes) {
            cout << "Graph contains a cycle! Topological sort not
possible." << endl;
            return;
        }

        cout << "Construction Priority Order:" << endl;
        for(int i = 0; i < topOrder.size(); i++) {
            cout << "Priority " << (i+1) << ": Node " <<
topOrder[i];
            if(nodes.count(topOrder[i])) {
                cout << " (" << nodes[topOrder[i]].nodeName << ")";
            }
            cout << endl;
        }
    }
};
```

```
// ==================== BINARY SEARCH TREE FOR ZONES
====================

class ZoneNode {
public:
    int zoneID;
    string zoneName;
    string zoneType;
    ZoneNode* left;
    ZoneNode* right;

    ZoneNode(int id, string name, string type) {
        zoneID = id;
        zoneName = name;
        zoneType = type;
        left = nullptr;
        right = nullptr;
    }
};

class ZoneBST {
private:
    ZoneNode* root;

    ZoneNode* insertHelper(ZoneNode* node, int id, string name,
string type) {
        if(node == nullptr) {
            return new ZoneNode(id, name, type);
```

```
        }

        if(id < node->zoneID) {
            node->left = insertHelper(node->left, id, name, type);
        } else if(id > node->zoneID) {
            node->right = insertHelper(node->right, id, name, type);
        }

        return node;
    }

    void inorderHelper(ZoneNode* node) {
        if(node != nullptr) {
            inorderHelper(node->left);
            cout << "Zone " << node->zoneID << ": " << node-
>zoneName
                 << " [" << node->zoneType << "]" << endl;
            inorderHelper(node->right);
        }
    }

    ZoneNode* searchHelper(ZoneNode* node, int id) {
        if(node == nullptr || node->zoneID == id) {
            return node;
        }

        if(id < node->zoneID) {
            return searchHelper(node->left, id);
        }
```

```
        return searchHelper(node->right, id);
    }

public:
    ZoneBST() {
        root = nullptr;
    }

    void insertZone(int id, string name, string type) {
        root = insertHelper(root, id, name, type);
    }

    void displayZones() {
        cout << "\n--- City Zones (BST Inorder Traversal) ---" <<
endl;
        inorderHelper(root);
    }

    ZoneNode* searchZone(int id) {
        return searchHelper(root, id);
    }
};

// ==================== AVL TREE FOR ZONES ====================

class AVLZoneNode {
public:
    int zoneID;
```

```
    string zoneName;
    string zoneType;
    AVLZoneNode* left;
    AVLZoneNode* right;
    int height;

    AVLZoneNode(int id, string name, string type) {
        zoneID = id;
        zoneName = name;
        zoneType = type;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

class ZoneAVL {
private:
    AVLZoneNode* root;

    int getHeight(AVLZoneNode* node) {
        if(node == nullptr) return 0;
        return node->height;
    }

    int getBalance(AVLZoneNode* node) {
        if(node == nullptr) return 0;
        return getHeight(node->left) - getHeight(node->right);
    }
```

```
    AVLZoneNode* rotateRight(AVLZoneNode* y) {
        AVLZoneNode* x = y->left;
        AVLZoneNode* T2 = x->right;

        x->right = y;
        y->left = T2;

        y->height = max(getHeight(y->left), getHeight(y->right)) +
1;
        x->height = max(getHeight(x->left), getHeight(x->right)) +
1;

        return x;
    }

    AVLZoneNode* rotateLeft(AVLZoneNode* x) {
        AVLZoneNode* y = x->right;
        AVLZoneNode* T2 = y->left;

        y->left = x;
        x->right = T2;

        x->height = max(getHeight(x->left), getHeight(x->right)) +
1;
        y->height = max(getHeight(y->left), getHeight(y->right)) +
1;

        return y;
    }
```

```
    AVLZoneNode* insertHelper(AVLZoneNode* node, int id, string
name, string type) {
        // Standard BST insertion
        if(node == nullptr) {
            return new AVLZoneNode(id, name, type);
        }

        if(id < node->zoneID) {
            node->left = insertHelper(node->left, id, name, type);
        } else if(id > node->zoneID) {
            node->right = insertHelper(node->right, id, name, type);
        } else {
            return node; // Duplicate IDs not allowed
        }

        // Update height
        node->height = 1 + max(getHeight(node->left),
getHeight(node->right));

        // Get balance factor
        int balance = getBalance(node);

        // Left Left Case
        if(balance > 1 && id < node->left->zoneID) {
            return rotateRight(node);
        }

        // Right Right Case
```

```
        if(balance < -1 && id > node->right->zoneID) {
            return rotateLeft(node);
        }

        // Left Right Case
        if(balance > 1 && id > node->left->zoneID) {
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }

        // Right Left Case
        if(balance < -1 && id < node->right->zoneID) {
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }

        return node;
    }

    void inorderHelper(AVLZoneNode* node) {
        if(node != nullptr) {
            inorderHelper(node->left);
            cout << "Zone " << node->zoneID << ": " << node-
>zoneName
                 << " [" << node->zoneType << "]" << endl;
            inorderHelper(node->right);
        }
    }
```

```
public:
    ZoneAVL() {
        root = nullptr;
    }

    void insertZone(int id, string name, string type) {
        root = insertHelper(root, id, name, type);
    }

    void displayZones() {
        cout << "\n--- City Zones (AVL Tree Balanced) ---" << endl;
        inorderHelper(root);
    }
};

class SmartCityNavigationSystem {
private:
    RoadNetwork roadNetwork;
    ZoneBST zoneBST;
    ZoneAVL zoneAVL;
    bool networkInitialized;

public:
    SmartCityNavigationSystem() {
        networkInitialized = false;
    }

    void initializeSystem() {
        int numNodes;
```

```
        cout << "\n--- Initialize Road Network ---" << endl;
        cout << "Enter number of intersections/nodes: ";
        cin >> numNodes;

        roadNetwork.initializeNetwork(numNodes);

        // Add node details
        cin.ignore();
        for(int i = 0; i < numNodes; i++) {
            string name, zone;
            cout << "\nNode " << i << ":" << endl;
            cout << "  Name: ";
            getline(cin, name);
            cout << "  Zone Type
(Residential/Commercial/Industrial): ";
            getline(cin, zone);

            RoadNode node(i, name, zone);
            roadNetwork.addNode(node);

            // Add to trees
            zoneBST.insertZone(i, name, zone);
            zoneAVL.insertZone(i, name, zone);
        }

        networkInitialized = true;
        cout << "\nRoad network initialized successfully!" << endl;
    }
```

```
    void addRoads() {
        if(!networkInitialized) {
            cout << "Please initialize the network first!" << endl;
            return;
        }

        int numRoads;
        cout << "\n--- Add Roads ---" << endl;
        cout << "Enter number of roads to add: ";
        cin >> numRoads;

        cout << "\n1. Bidirectional Roads" << endl;
        cout << "2. Unidirectional Roads (for Topological Sort)" <<
endl;
        cout << "Choose road type: ";
        int choice;
        cin >> choice;

        for(int i = 0; i < numRoads; i++) {
            int from, to, distance;
            cout << "\nRoad " << (i+1) << ":" << endl;
            cout << "  From Node ID: ";
            cin >> from;
            cout << "  To Node ID: ";
            cin >> to;
            cout << "  Distance (km): ";
            cin >> distance;

            if(choice == 1) {
```

```
            roadNetwork.addRoad(from, to, distance);
        } else {
            roadNetwork.addDirectedRoad(from, to, distance);
        }
    }

    cout << "\nRoads added successfully!" << endl;
}

void displayNetwork() {
    if(!networkInitialized) {
        cout << "Please initialize the network first!" << endl;
        return;
    }

    cout << "\n--- Display Options ---" << endl;
    cout << "1. Adjacency Matrix" << endl;
    cout << "2. Adjacency List" << endl;
    cout << "3. City Zones (BST)" << endl;
    cout << "4. City Zones (AVL)" << endl;
    cout << "Enter choice: ";

    int choice;
    cin >> choice;

    switch(choice) {
        case 1:
            roadNetwork.displayAdjacencyMatrix();
            break;
```

```
            case 2:
                roadNetwork.displayAdjacencyList();
                break;
            case 3:
                zoneBST.displayZones();
                break;
            case 4:
                zoneAVL.displayZones();
                break;
            default:
                cout << "Invalid choice!" << endl;
        }
    }

    void findShortestPath() {
        if(!networkInitialized) {
            cout << "Please initialize the network first!" << endl;
            return;
        }

        cout << "\n--- Shortest Path Algorithms ---" << endl;
        cout << "1. Dijkstra's Algorithm (Single Source)" << endl;
        cout << "2. Floyd-Warshall (All Pairs)" << endl;
        cout << "Enter choice: ";

        int choice;
        cin >> choice;

        if(choice == 1) {
```

```
            int start, end;
            cout << "\nEnter start node ID: ";
            cin >> start;
            cout << "Enter end node ID: ";
            cin >> end;

            roadNetwork.dijkstraShortestPath(start, end);
        } else if(choice == 2) {
            roadNetwork.floydWarshallAllPairs();
        } else {
            cout << "Invalid choice!" << endl;
        }
    }

    void planInfrastructure() {
        if(!networkInitialized) {
            cout << "Please initialize the network first!" << endl;
            return;
        }

        cout << "\n--- Infrastructure Planning ---" << endl;
        cout << "1. Minimum Spanning Tree (Prim's Algorithm)" <<
endl;
        cout << "2. Minimum Spanning Tree (Kruskal's Algorithm)" <<
endl;
        cout << "3. Construction Priority (Topological Sort)" <<
endl;
        cout << "Enter choice: ";

        int choice;
```

```
        cin >> choice;

        switch(choice) {
            case 1:
                roadNetwork.primMST();
                break;
            case 2:
                roadNetwork.kruskalMST();
                break;
            case 3:
                roadNetwork.topologicalSort();
                break;
            default:
                cout << "Invalid choice!" << endl;
        }
    }

    void loadSampleData() {
        cout << "\n--- Loading Sample Smart City Data ---" << endl;

        // Initialize with 7 nodes
        roadNetwork.initializeNetwork(7);

        // Add nodes (intersections)
        roadNetwork.addNode(RoadNode(0, "Central Station",
"Commercial"));
        roadNetwork.addNode(RoadNode(1, "Tech Park", "Industrial"));
        roadNetwork.addNode(RoadNode(2, "City Mall", "Commercial"));
```

```
        roadNetwork.addNode(RoadNode(3, "Residential Area",
"Residential"));
        roadNetwork.addNode(RoadNode(4, "Hospital", "Healthcare"));
        roadNetwork.addNode(RoadNode(5, "University",
"Educational"));
        roadNetwork.addNode(RoadNode(6, "Airport",
"Transportation"));

        // Add to BST and AVL trees
        zoneBST.insertZone(0, "Central Station", "Commercial");
        zoneBST.insertZone(1, "Tech Park", "Industrial");
        zoneBST.insertZone(2, "City Mall", "Commercial");
        zoneBST.insertZone(3, "Residential Area", "Residential");
        zoneBST.insertZone(4, "Hospital", "Healthcare");
        zoneBST.insertZone(5, "University", "Educational");
        zoneBST.insertZone(6, "Airport", "Transportation");

        zoneAVL.insertZone(0, "Central Station", "Commercial");
        zoneAVL.insertZone(1, "Tech Park", "Industrial");
        zoneAVL.insertZone(2, "City Mall", "Commercial");
        zoneAVL.insertZone(3, "Residential Area", "Residential");
        zoneAVL.insertZone(4, "Hospital", "Healthcare");
        zoneAVL.insertZone(5, "University", "Educational");
        zoneAVL.insertZone(6, "Airport", "Transportation");

        // Add roads (edges with distances in km)
        roadNetwork.addRoad(0, 1, 5);   // Central Station - Tech
Park
        roadNetwork.addRoad(0, 2, 3);   // Central Station - City
Mall
```

```
        roadNetwork.addRoad(1, 2, 4);   // Tech Park - City Mall
        roadNetwork.addRoad(1, 3, 6);   // Tech Park - Residential
Area
        roadNetwork.addRoad(2, 3, 7);   // City Mall - Residential
Area
        roadNetwork.addRoad(2, 4, 8);   // City Mall - Hospital
        roadNetwork.addRoad(3, 4, 2);   // Residential Area -
Hospital
        roadNetwork.addRoad(3, 5, 5);   // Residential Area -
University
        roadNetwork.addRoad(4, 5, 3);   // Hospital - University
        roadNetwork.addRoad(4, 6, 10);  // Hospital - Airport
        roadNetwork.addRoad(5, 6, 7);   // University - Airport

        networkInitialized = true;

        cout << "Sample data loaded successfully!" << endl;
        cout << "7 intersections and 11 roads added to the network."
<< endl;
    }

    void analyzeNetwork() {
        if(!networkInitialized) {
            cout << "Please initialize the network first!" << endl;
            return;
        }

        cout << "\n===== NETWORK ANALYSIS REPORT =====" << endl;

        // Display all representations
```

```
    cout << "\n1. GRAPH REPRESENTATIONS:" << endl;
    roadNetwork.displayAdjacencyMatrix();
    roadNetwork.displayAdjacencyList();

    // Display zones
    cout << "\n2. CITY ZONE INDEXING:" << endl;
    zoneBST.displayZones();

    // Show shortest paths from central node (0)
    cout << "\n3. SHORTEST PATHS FROM CENTRAL STATION:" << endl;
    for(int i = 1; i < 7; i++) {
        roadNetwork.dijkstraShortestPath(0, i);
        cout << endl;
    }

    // Show MST
    cout << "\n4. MINIMUM SPANNING TREE ANALYSIS:" << endl;
    roadNetwork.primMST();
    cout << endl;
    roadNetwork.kruskalMST();

    cout << "\n===== END OF ANALYSIS REPORT =====" << endl;
}

void compareAlgorithms() {
    if(!networkInitialized) {
        cout << "Please initialize the network first!" << endl;
        return;
    }
```

```
        cout << "\n--- Algorithm Comparison ---" << endl;

        cout << "\n1. SHORTEST PATH ALGORITHMS:" << endl;
        cout << "   Dijkstra's Algorithm:" << endl;
        cout << "   - Time Complexity: O((V + E) log V) with
priority queue" << endl;
        cout << "   - Best for: Single source shortest path" <<
endl;
        cout << "   - Use case: Navigation from current location" <<
endl;

        cout << "\n   Floyd-Warshall Algorithm:" << endl;
        cout << "   - Time Complexity: O(V³)" << endl;
        cout << "   - Best for: All pairs shortest paths" << endl;
        cout << "   - Use case: Precomputing all routes" << endl;

        cout << "\n2. MINIMUM SPANNING TREE ALGORITHMS:" << endl;
        cout << "   Prim's Algorithm:" << endl;
        cout << "   - Time Complexity: O((V + E) log V)" << endl;
        cout << "   - Best for: Dense graphs" << endl;

        cout << "\n   Kruskal's Algorithm:" << endl;
        cout << "   - Time Complexity: O(E log E)" << endl;
        cout << "   - Best for: Sparse graphs" << endl;

        cout << "\n3. TREE STRUCTURES:" << endl;
        cout << "   Binary Search Tree:" << endl;
        cout << "   - Average: O(log n), Worst: O(n)" << endl;
```

```
        cout << "\n   AVL Tree:" << endl;
        cout << "   - Guaranteed: O(log n) for all operations" <<
endl;
    }
};

int main() {
    SmartCityNavigationSystem system;
    int choice;

    cout << "============================================" << endl;
    cout << "    SMART CITY ROAD NAVIGATION SYSTEM      " << endl;
    cout << "============================================" << endl;

    while(true) {
        cout << "\n--- Main Menu ---" << endl;
        cout << "1.  Initialize Road Network" << endl;
        cout << "2.  Add Roads" << endl;
        cout << "3.  Display Network" << endl;
        cout << "4.  Find Shortest Path" << endl;
        cout << "5.  Plan Infrastructure (MST/Topological Sort)" <<
endl;
        cout << "6.  Analyze Network (Complete Analysis)" << endl;
        cout << "7.  Compare Algorithms" << endl;
        cout << "8.  Load Sample Data" << endl;
        cout << "9.  Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
```

```
        switch(choice) {
            case 1:
                system.initializeSystem();
                break;
            case 2:
                system.addRoads();
                break;
            case 3:
                system.displayNetwork();
                break;
            case 4:
                system.findShortestPath();
                break;
            case 5:
                system.planInfrastructure();
                break;
            case 6:
                system.analyzeNetwork();
                break;
            case 7:
                system.compareAlgorithms();
                break;
            case 8:
                system.loadSampleData();
                break;
            case 9:
                cout << "\nThank you for using Smart City Navigation
System!" << endl;
                cout << "Goodbye!" << endl;
```

```
                return 0;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    }

    return 0;
}
```

OUTPUT:

```
==========================================
    SMART CITY ROAD NAVIGATION SYSTEM
==========================================

--- Main Menu ---
1.  Initialize Road Network
2.  Add Roads
3.  Display Network
4.  Find Shortest Path
5.  Plan Infrastructure (MST/Topological Sort)
6.  Analyze Network (Complete Analysis)
7.  Compare Algorithms
8.  Load Sample Data
9.  Exit
Enter your choice: 1

--- Initialize Road Network ---
Enter number of intersections/nodes: 2

Node 0:
  Name: a
  Zone Type (Residential/Commercial/Industrial): ab

Node 1:
  Name: c
  Zone Type (Residential/Commercial/Industrial): cd

Road network initialized successfully!

--- Main Menu ---
```