**Business Case: City Management**

**Sub-task: Managing Traffic in City**

**SDG Goal: 11**

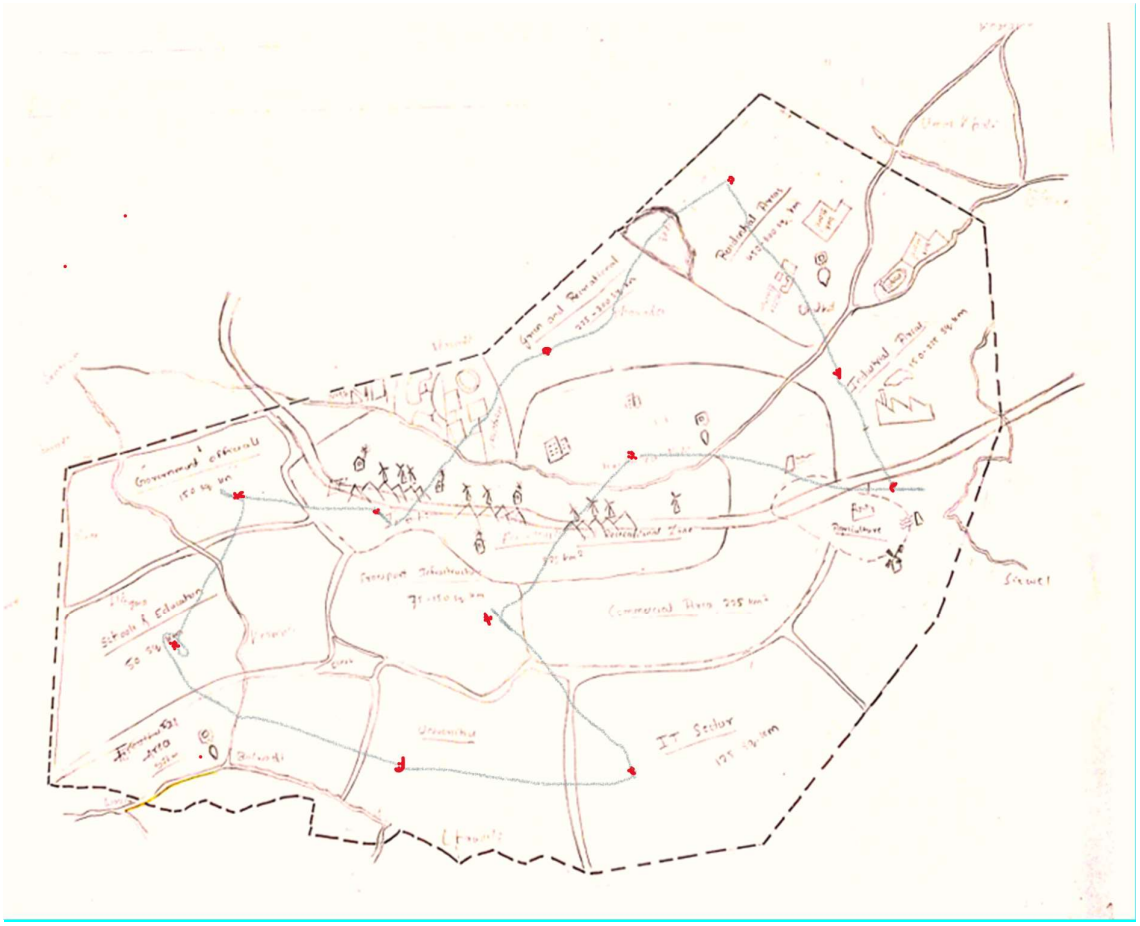
**Target: 11.2**

**Indicator: 11.2**

**Traffic Management Plan for Vatsalya Nagar:**

To address traffic management issues in Vatsalya Nagar, we will analyse and optimize the transportation network using **Dijkstra's Algorithm**, **Kruskal's Algorithm**, and **BFS/DFS** for critical traversal tasks.

| **Zone** | **Node ID** | **Connection** | **Distance (km)** | **Congestion Level (Weight)** |
|---|---|---|---|---|
| Resendetial Zone 1 | A | B,C | 3,5 | 2,3 |
| Commercial Hub | B | A,D | 3,7 | 2,5 |
| Industrial Area | C | A,D,E | 5,4,6 | 3,4,6 |
| Transport Hub | D | B,C,F | 7,4,8 | 5,4,7 |
| Recreational Park | E | C,F | 6,3 | 6,2 |
| Metro Station | F | D,E | 8,3 | 7,2 |



**Algorithm Details:**

**A. Dijkstra's Algorithm (Shortest Path Calculation)**

- Input: Graph with nodes, edges, and weights (distance or congestion).
- Output: Shortest path from the source node to all other nodes.

**Algorithm Steps:**

1. Initialize distances to all nodes as infinity (∞) except the source node (set to 0).
2. Use a priority queue to select the node with the smallest distance.
3. For the current node, update the distances to its neighbours if a shorter path is found.
4. Repeat until all nodes are processed.

**Diagrams for Traffic and Metro Design:**

**Traffic Management:**

- Nodes represent intersections (e.g., A, B, C).
- Edges represent roads with weights for congestion.
- Shortest paths marked using Dijkstra's algorithm.

**Metro Network:**

- Nodes as major hubs.
- Edges for rail lines with cost weights.
- Spanning tree derived from Kruskal's algorithm.

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <climits> // For INT_MAX

using namespace std;

// Define a type for graph representation (adjacency list)
typedef pair<int, char> Edge; // (weight, neighbor)

// Dijkstra's Algorithm
unordered_map<char, int> dijkstra(unordered_map<char, vector<Edge>> &graph, char start_node) {
    // Initialize distances to infinity
    unordered_map<char, int> distances;
    for (auto &node : graph) {
        distances[node.first] = INT_MAX;
    }
```

```
distances[start_node] = 0;

// Min-heap to store (distance, node)
priority_queue<Edge, vector<Edge>, greater<Edge>> pq;
pq.push({0, start_node});

// Set to track visited nodes
unordered_map<char, bool> visited;

while (!pq.empty()) {
    int current_distance = pq.top().first;
    char current_node = pq.top().second;
    pq.pop();

    // If the node is already visited, skip it
    if (visited[current_node]) continue;
    visited[current_node] = true;

    // Update distances for neighbors
    for (auto &neighbor : graph[current_node]) {
        char next_node = neighbor.second;
        int weight = neighbor.first;

        int new_distance = current_distance + weight;
        if (new_distance < distances[next_node]) {
            distances[next_node] = new_distance;
            pq.push({new_distance, next_node});
        }
    }
}
```

```
    return distances;
}

int main() {
    // Graph representation using adjacency list
    unordered_map<char, vector<Edge>> graph = {
        {'A', {{2, 'B'}, {3, 'C'}}},
        {'B', {{2, 'A'}, {5, 'D'}}},
        {'C', {{3, 'A'}, {4, 'D'}, {6, 'E'}}},
        {'D', {{5, 'B'}, {4, 'C'}, {7, 'F'}}},
        {'E', {{6, 'C'}, {2, 'F'}}},
        {'F', {{7, 'D'}, {2, 'E'}}}
    };

    // Starting node
    char start = 'A';

    // Compute shortest paths
    unordered_map<char, int> shortest_paths = dijkstra(graph, start);

    // Print the results
    cout << "Shortest paths from " << start << ":\n";
    for (auto &entry : shortest_paths) {
        cout << entry.first << ": " << (entry.second == INT_MAX ? -1 : entry.second) << "\n";
    }

    return 0;
}
```

## Output:

```
'A': {{2, 'B'}, {3, 'C'}},
```

'B': {{2, 'A'}, {5, 'D'}},

'C': {{3, 'A'}, {4, 'D'}, {6, 'E'}},

'D': {{5, 'B'}, {4, 'C'}, {7, 'F'}},

'E': {{6, 'C'}, {2, 'F'}},

'F': {{7, 'D'}, {2, 'E'}}

Shortest paths from A:

A: 0

B: 2

C: 3

D: 7

E: 9

F: 11

### B. Kruskal's Algorithm (Designing Metro Network)

- **Input**: Weighted graph (edges with costs).
- **Output**: Minimal spanning tree (MST) connecting all nodes.

### Algorithm Steps:

1. Sort all edges by weight.
2. Initialize subsets for each node.
3. Iterate through edges, adding the edge to the MST if it doesn't form a cycle.
4. Stop when all nodes are connected.

### Kruskal's Algorithm Implementation:

```
#include <iostream>

#include <vector>

#include <algorithm>
```

```
using namespace std;

struct Edge {
    char src, dest;
    int weight;
};

// Find function for union-find
char findParent(unordered_map<char, char> &parent, char node) {
    if (parent[node] == node) return node;
    return parent[node] = findParent(parent, parent[node]);
}

// Union function for union-find
void unionNodes(unordered_map<char, char> &parent, unordered_map<char, int> &rank, char u,
char v) {
    char rootU = findParent(parent, u);
    char rootV = findParent(parent, v);
    if (rootU != rootV) {
        if (rank[rootU] > rank[rootV]) {
            parent[rootV] = rootU;
        } else if (rank[rootU] < rank[rootV]) {
            parent[rootU] = rootV;
        } else {
            parent[rootV] = rootU;
            rank[rootU]++;
        }
    }
}
```

```
void kruskal(vector<Edge> &edges, vector<char> &nodes) {
    // Sort edges by weight
    sort(edges.begin(), edges.end(), [](Edge a, Edge b) {
        return a.weight < b.weight;
    });

    unordered_map<char, char> parent;
    unordered_map<char, int> rank;
    for (char node : nodes) {
        parent[node] = node;
        rank[node] = 0;
    }

    vector<Edge> mst;
    for (Edge &edge : edges) {
        if (findParent(parent, edge.src) != findParent(parent, edge.dest)) {
            mst.push_back(edge);
            unionNodes(parent, rank, edge.src, edge.dest);
        }
    }

    // Print MST
    cout << "Minimal Spanning Tree (MST):\n";
    for (Edge &edge : mst) {
        cout << edge.src << " - " << edge.dest << " (Weight: " << edge.weight << ")\n";
    }
}

int main() {
    vector<Edge> edges = {
        {'A', 'B', 3}, {'A', 'C', 5}, {'B', 'D', 7},
```

```
        {'C', 'D', 4}, {'C', 'E', 6}, {'D', 'F', 8},
        {'E', 'F', 3}
    };
    vector<char> nodes = {'A', 'B', 'C', 'D', 'E', 'F'};

    kruskal(edges, nodes);

    return 0;
}
```

## C. DFS and BFS: Identifying Bottlenecks and Traversing Networks:

**DFS**

- Use case: Identify bottlenecks by traversing all paths in the network.
- Helps locate congested intersections by recursively visiting nodes.

**BFS**

- Use case: Identify shortest path in terms of edges (not weights).
- Ideal for analysing reachability or simple traversal.

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <unordered_set>

using namespace std;

// Graph representation
```

```
typedef unordered_map<char, vector<char>> Graph;

// DFS to identify bottlenecks
void dfs(char node, Graph &graph, unordered_set<char> &visited) {
    visited.insert(node);
    cout << node << " ";
    for (char neighbor : graph[node]) {
        if (visited.find(neighbor) == visited.end()) {
            dfs(neighbor, graph, visited);
        }
    }
}

// BFS for simple shortest path
void bfs(char start, Graph &graph) {
    queue<char> q;
    unordered_set<char> visited;
    q.push(start);
    visited.insert(start);
    cout << "BFS Traversal from " << start << ": ";

    while (!q.empty()) {
        char node = q.front();
        q.pop();
        cout << node << " ";

        for (char neighbor : graph[node]) {
            if (visited.find(neighbor) == visited.end()) {
                visited.insert(neighbor);
                q.push(neighbor);
            }
```

```
        }
    }
    cout << endl;
}

int main() {
    Graph graph = {
        {'A', {'B', 'C'}},
        {'B', {'A', 'D'}},
        {'C', {'A', 'D', 'E'}},
        {'D', {'B', 'C', 'F'}},
        {'E', {'C', 'F'}},
        {'F', {'D', 'E'}}
    };

    // Perform DFS and BFS
    unordered_set<char> visited;
    cout << "DFS Traversal from A: ";
    dfs('A', graph, visited);
    cout << endl;

    bfs('A', graph);

    return 0;
}
```

**Output:**

DFS Traversal from A: A B D C E F

BFS Traversal from A: A B C D E F

### D. Floyd-Warshall Algorithm: All-Pairs Shortest Paths:

- Use case: Optimize overall traffic network by finding shortest paths between all intersections.
- Useful for dense graphs or multi-source shortest path computation.

### Floyd-Warshall Algorithm Implementation:

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

const int INF = INT_MAX;

void floydWarshall(vector<vector<int>> &graph) {
   int V = graph.size();
   vector<vector<int>> dist = graph;

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

   // Print shortest distances
```

```
    cout << "All-Pairs Shortest Paths:\n";
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            if (dist[i][j] == INF)
                cout << "INF ";
            else
                cout << dist[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    vector<vector<int>> graph = {
        {0, 2, 3, INF, INF, INF},
        {2, 0, INF, 5, INF, INF},
        {3, INF, 0, 4, 6, INF},
        {INF, 5, 4, 0, INF, 7},
        {INF, INF, 6, INF, 0, 2},
        {INF, INF, INF, 7, 2, 0}
    };

    floydWarshall(graph);

    return 0;
}
```

**Output:**

All-Pairs Shortest Paths:

0 2 3 7 9 11

2 0 5 5 11 13

3 5 0 4 6 8

7 5 4 0 8 7

9 11 6 8 0 2

11 13 8 7 2 0

**E. Bubble Sort and Quick Sort: Traffic Prioritization:**

**Bubble Sort:** Rank areas by congestion level (low to high).

**Quick Sort:** Rank intersections for real-time signal optimization.

**Bubble Sort and Quick Sort Implementation:**

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Bubble Sort
void bubbleSort(vector<pair<char, int>> &congestion) {
   int n = congestion.size();
   for (int i = 0; i < n - 1; ++i) {
      for (int j = 0; j < n - i - 1; ++j) {
         if (congestion[j].second > congestion[j + 1].second) {
            swap(congestion[j], congestion[j + 1]);
         }
      }
   }
}
```

```
// Quick Sort
int partition(vector<pair<char, int>> &data, int low, int high) {
    int pivot = data[high].second;
    int i = low - 1;

    for (int j = low; j < high; ++j) {
        if (data[j].second <= pivot) {
            i++;
            swap(data[i], data[j]);
        }
    }
    swap(data[i + 1], data[high]);
    return i + 1;
}

void quickSort(vector<pair<char, int>> &data, int low, int high) {
    if (low < high) {
        int pi = partition(data, low, high);
        quickSort(data, low, pi - 1);
        quickSort(data, pi + 1, high);
    }
}

int main() {
    vector<pair<char, int>> congestion = {
        {'A', 3}, {'B', 5}, {'C', 2}, {'D', 6}, {'E', 4}, {'F', 1}
    };

    // Bubble Sort
    bubbleSort(congestion);
```

```
    cout << "Bubble Sort (Low to High Congestion): ";
    for (auto &p : congestion) {
        cout << p.first << ":" << p.second << " ";
    }
    cout << endl;

    // Quick Sort
    quickSort(congestion, 0, congestion.size() - 1);
    cout << "Quick Sort (Low to High Congestion): ";
    for (auto &p : congestion) {
        cout << p.first << ":" << p.second << " ";
    }
    cout << endl;

    return 0;
}
```

**Output:**

Bubble Sort (Low to High Congestion): F:1 C:2 A:3 E:4 B:5 D:6

Quick Sort (Low to High Congestion): F:1 C:2 A:3 E:4 B:5 D:6

## Business Case: City Management

**Sub-task: Innovation and implementation in research institute**

**SDG Goal: 11**

**Target: 11.3**

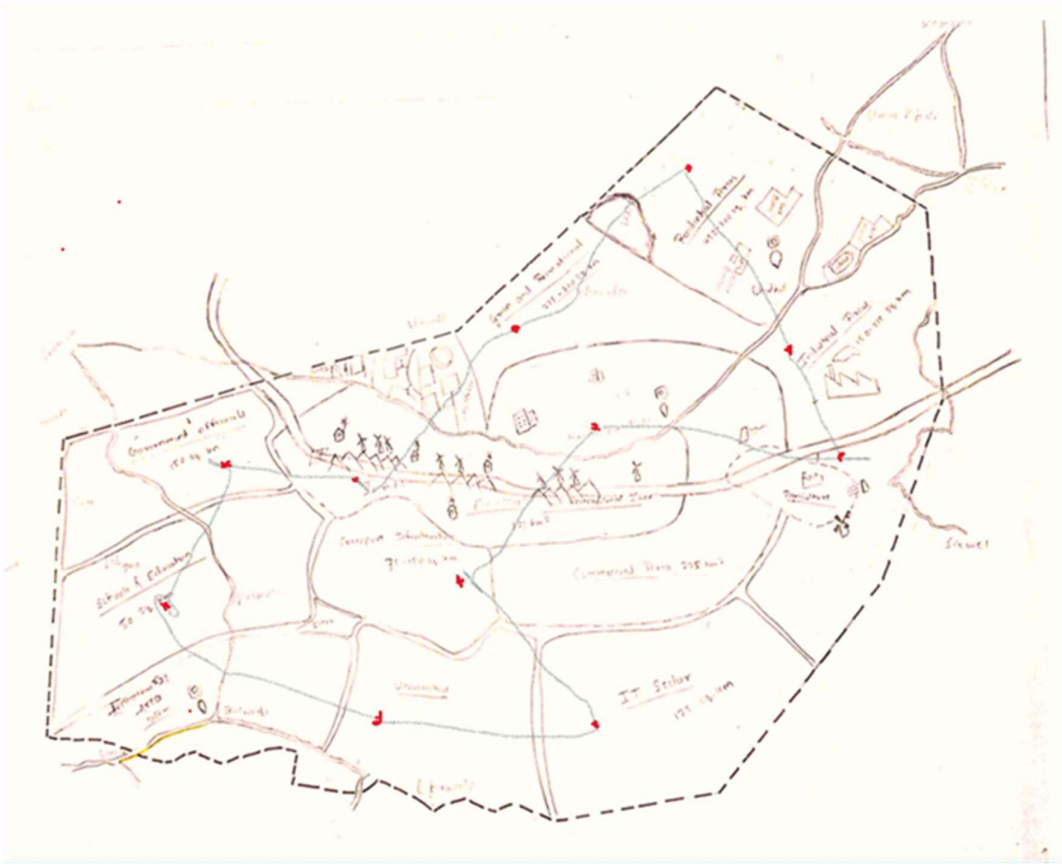
**Indicator: 11.3**

**Case 2: Innovation and Implementation in Research Institutes:**

Foster innovation through collaborations between research institutes in Vatsalya and other cities, aligned with SDG 11.3, which promotes inclusive and sustainable urbanization via partnerships and knowledge-sharing.

**Approach:**

1. **Graph Representation**
    - Nodes: Represent research institutes or collaborative hubs in Vatsalya and other cities.
    - Edges: Represent potential collaboration connections, with weights denoting the cost or effort for collaboration (e.g., funding, travel, or communication resources).
2. **Algorithms**
    - **Union-Find**: To manage and check connectivity between research hubs, ensuring efficient tracking of connected components (collaborative groups).
    - **Kruskal's Algorithm**: To identify the minimum-cost connections forming an optimal collaboration network.
    - **Prim's Algorithm**: Prim's Algorithm is another way to find a Minimum Spanning Tree (MST) but grows the MST one vertex at a time.
    - **Dijkstra's Algorithm:** If you need to calculate the shortest path from one research hub to all others for effective communication, use Dijkstra's Algorithm.
    - **Bellman-Ford Algorithm:** For graphs with negative weights (e.g., funding or resource savings), use Bellman-Ford to find shortest paths from a single source.
    - **Floyd-Warshall Algorithm:** For finding the shortest path between all pairs of nodes.



**Graph Representation :**

| Research Hub | Node ID | Connections | Collaboration Cost (Weight) |
|---|---|---|---|
| Vatsalya Central | A | B,C,D | 5,3,6 |
| Institute B | B | A,C | 5,4 |
| Institute C | C | A,B,D | 3,4,2 |
| Institute D | D | A,C | 6,2 |

**Solution:**

**1.Union-Find Algorithm:**

Use Case

Union-Find is used to track whether two research institutes are already in a collaborative group. It ensures that the network remains efficient and avoids redundant connections.

Steps:

1. Initialization: Each node is its own parent, and the rank is initialized to 0.
2. Find: Determine the root (leader) of a node's set.
3. Union: Merge two sets if they belong to different groups, using rank to keep the tree shallow.

**Kruskal's Algorithm:**

Use Case:

Find the minimum-cost network connecting all research hubs in Vatsalya and other cities.

Steps:

1. Sort all edges by weight (cost).
2. Use Union-Find to add edges without creating cycles.
3. Stop when all nodes are connected, forming a minimum spanning tree (MST).

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>

using namespace std;

// Edge structure
struct Edge {
    char src, dest;
    int weight;
};
```

```
// Find function for Union-Find
char findParent(unordered_map<char, char> &parent, char node) {
    if (parent[node] != node)
        parent[node] = findParent(parent, parent[node]);
    return parent[node];
}

// Union function for Union-Find
void unionNodes(unordered_map<char, char> &parent, unordered_map<char, int> &rank, char u,
char v) {
    char root_u = findParent(parent, u);
    char root_v = findParent(parent, v);

    if (root_u != root_v) {
        if (rank[root_u] > rank[root_v]) {
            parent[root_v] = root_u;
        } else if (rank[root_u] < rank[root_v]) {
            parent[root_u] = root_v;
        } else {
            parent[root_v] = root_u;
            rank[root_u]++;
        }
    }
}

// Kruskal's Algorithm
vector<Edge> kruskal(vector<Edge> &edges, vector<char> &nodes) {
    unordered_map<char, char> parent;
    unordered_map<char, int> rank;
    for (char node : nodes) {
        parent[node] = node;
```

```
        rank[node] = 0;
    }

    sort(edges.begin(), edges.end(), [](Edge a, Edge b) { return a.weight < b.weight; });

    vector<Edge> mst;
    for (Edge &edge : edges) {
        if (findParent(parent, edge.src) != findParent(parent, edge.dest)) {
            mst.push_back(edge);
            unionNodes(parent, rank, edge.src, edge.dest);
        }
    }

    return mst;
}

int main() {
    vector<Edge> edges = {
        {'A', 'B', 5}, {'A', 'C', 3}, {'A', 'D', 6},
        {'B', 'C', 4}, {'C', 'D', 2}
    };
    vector<char> nodes = {'A', 'B', 'C', 'D'};

    vector<Edge> mst = kruskal(edges, nodes);

    cout << "Minimum Spanning Tree:\n";
    for (Edge &edge : mst) {
        cout << edge.src << " - " << edge.dest << " (Weight: " << edge.weight << ")\n";
    }

    return 0;
```

}

**Output:**

Graph:

A --(5)-- B

A --(3)-- C

A --(6)-- D

B --(4)-- C

C --(2)—D

Minimum spanning Tree:

C - D (Weight: 2)

A - C (Weight: 3)

B - C (Weight: 4)

**2. Prim's Algorithm**

- **Use Case**: Find the Minimum Spanning Tree (MST), focusing on one node and growing the tree step by step.
- **How It Works**:
    - Starts from a specific node.
    - Adds the smallest edge connecting a visited node to an unvisited node, ensuring no cycles.
- **Relevance**:
    - Useful when collaboration begins from a primary hub (e.g., Vatsalya Central).
    - Ensures all research hubs are connected with minimum collaboration cost.

- **Efficiency**: O(Elog⁡E)O(E \log E)O(ElogE), where EEE is the number of edges.
- **Reason**: Efficient when the graph is sparse (few edges compared to nodes), as it focuses on edge-based operations.

- **Implementation:**

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

struct Edge {
  int dest, weight;
};

vector<pair<int, int>> prim(int nodes, vector<vector<Edge>> &graph) {
  vector<bool> visited(nodes, false);
  vector<pair<int, int>> mst; // Store MST edges
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;

  pq.push({0, 0}); // {weight, node}
  while (!pq.empty()) {
    auto [weight, node] = pq.top();
    pq.pop();

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

    for (auto &edge : graph[node]) {
      if (!visited[edge.dest]) {
        pq.push({edge.weight, edge.dest});
        mst.push_back({node, edge.dest});
      }
    }
  }
  return mst;
}

int main() {
  int nodes = 4;
  vector<vector<Edge>> graph(nodes);

  graph[0] = {{1, 5}, {2, 3}, {3, 6}}; // A: {B(5), C(3), D(6)}
  graph[1] = {{0, 5}, {2, 4}};        // B: {A(5), C(4)}
```

```
    graph[2] = {{0, 3}, {1, 4}, {3, 2}}; // C: {A(3), B(4), D(2)}
    graph[3] = {{0, 6}, {2, 2}};         // D: {A(6), C(2)}

    vector<pair<int, int>> mst = prim(nodes, graph);
    cout << "Minimum Spanning Tree:\n";
    for (auto &[u, v] : mst) {
        cout << u << " - " << v << "\n";
    }

    return 0;
}
```

### 3. Dijkstra's Algorithm

- **Use Case**: Determine the shortest path from a single research hub to all others.
- **How It Works**:
  - Starts from a source node and iteratively updates the shortest distance to its neighbors.
  - Prioritizes smaller weights using a priority queue.
- **Relevance**:
  - Helps analyze the communication or travel costs from a specific institute (e.g., A) to every other hub.
  - Ensures efficient planning for direct collaborations.
- **Efficiency**: O(E+Vlog⁡V)O(E + V \log V)O(E+VlogV).
- **Reason**: Effective for finding shortest paths from a single source node to others.
- **Implementation**:

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

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

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

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

```
        if (d > dist[u]) continue;

        for (auto &[v, weight] : graph[u]) {
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    cout << "Shortest distances from node " << src << ":\n";
    for (int i = 0; i < nodes; ++i) {
        cout << "Node " << i << ": " << dist[i] << "\n";
    }
}

int main() {
    int nodes = 4;
    vector<vector<pair<int, int>>> graph(nodes);

    graph[0] = {{1, 5}, {2, 3}, {3, 6}}; // A: {B(5), C(3), D(6)}
    graph[1] = {{0, 5}, {2, 4}};         // B: {A(5), C(4)}
    graph[2] = {{0, 3}, {1, 4}, {3, 2}}; // C: {A(3), B(4), D(2)}
    graph[3] = {{0, 6}, {2, 2}};         // D: {A(6), C(2)}

    dijkstra(0, graph);
    return 0;
}
```

### 4. Bellman-Ford Algorithm

- **Use Case**: Single-source shortest path, including scenarios with negative weights.
- **How It Works**:
    - Iterates through all edges multiple times to ensure shortest paths are updated.
    - Detects negative weight cycles.
- **Relevance**:
    - Can handle scenarios where certain collaborations provide funding or resource savings (negative weights).
    - Useful for evaluating all possible paths while considering incentives.
- **Efficiency**: O(V×E)O(V \times E)O(V×E).

- **Reason**: Handles negative weights but is slower due to iterative edge relaxation.
- **Implementation**:

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

void bellmanFord(int src, int nodes, vector<vector<int>> &edges) {
    vector<int> dist(nodes, INT_MAX);
    dist[src] = 0;

    for (int i = 0; i < nodes - 1; ++i) {
        for (auto &edge : edges) {
            int u = edge[0], v = edge[1], weight = edge[2];
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    cout << "Shortest distances from node " << src << ":\n";
    for (int i = 0; i < nodes; ++i) {
        cout << "Node " << i << ": " << dist[i] << "\n";
    }
}

int main() {
    int nodes = 4;
    vector<vector<int>> edges = {
        {0, 1, 5}, {0, 2, 3}, {0, 3, 6},
        {1, 2, 4}, {2, 3, 2}
    };

    bellmanFord(0, nodes, edges);
    return 0;
}
```

## 5. Floyd-Warshall Algorithm

- **Use Case**: Find the shortest paths between all pairs of nodes.
- **How It Works**:
    - Uses dynamic programming to evaluate paths through intermediate nodes.

    - Updates the shortest path iteratively for all node pairs.
- **Relevance**:
    - Maps all collaboration costs across research hubs, giving a global view of the network.
    - Helps optimize partnerships when hubs in different cities require frequent connections.
- **Efficiency**: O(V3)O(V^3)O(V3).
- **Reason**: Computes all-pairs shortest paths but is computationally expensive.
- **Implementation**:

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

void floydWarshall(vector<vector<int>> &graph) {
   int nodes = graph.size();
   vector<vector<int>> dist = graph;

   for (int k = 0; k < nodes; ++k) {
      for (int i = 0; i < nodes; ++i) {
         for (int j = 0; j < nodes; ++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];
            }
         }
      }
   }

   cout << "Shortest distances between all pairs:\n";
   for (int i = 0; i < nodes; ++i) {
      for (int j = 0; j < nodes; ++j) {
         if (dist[i][j] == INT_MAX)
            cout << "INF ";
         else
            cout << dist[i][j] << " ";
      }
      cout << "\n";
   }
}

int main() {
   int nodes = 4;
   vector<vector<int>> graph = {
```

```
    {0, 5, 3, 6},
    {5, 0, 4, INT_MAX},
    {3, 4, 0, 2},
    {6, INT_MAX, 2, 0}
  };

  floydWarshall(graph);
  return 0;
}
```

## Comparison of the cases:

| Algorithm | Key Feature | Best for this case |
|---|---|---|
| Prim's Algorithm | Builds MST starting from a single node. | Building a cost-efficient collaboration tree starting from a central hub like Vatsalya Central |
| Kruskal's Algorithm | Builds MST by considering edges in ascending order of weight. | Creating an MST when edges and their costs are pre-defined, focusing on minimizing total collaboration cost. |
| Dijkstra's Algorithm | Single-source shortest paths (no negatives). | Optimizing direct communication or travel costs from a single research hub. |
| Bellman-Ford Algorithm | Handles negative weights. | Evaluating collaborations with funding or savings incentives. |
| Floyd-Warshall Algorithm | All-pairs shortest paths. | Mapping the entire collaboration network for optimal resource allocation. |

## Most Efficient Algorithm for the Case:

## Recommendation: Kruskal's Algorithm

- **Why?**
  - The graph is small and sparse.
  - Collaboration costs are pre-defined (weights on edges).
  - It minimizes the total cost of connections efficiently.
  - Simple to implement and avoids unnecessary cycles.

- **Efficiency Comparison**:
    - Kruskal's O(ElogE)O(E \log E)O(ElogE) is faster than Floyd-Warshall's O(V3)O(V^3)O(V3) and Bellman-Ford's O(V×E)O(V \times E)O(V×E) for this small graph.
    - It is comparable to Prim's but better for edge-centric problems.

**Conclusion**

Use **Kruskal's Algorithm** for its simplicity, efficiency, and alignment with the problem's goal: minimizing collaboration costs while ensuring all research hubs are connected.

+

**Business Case: City Management**

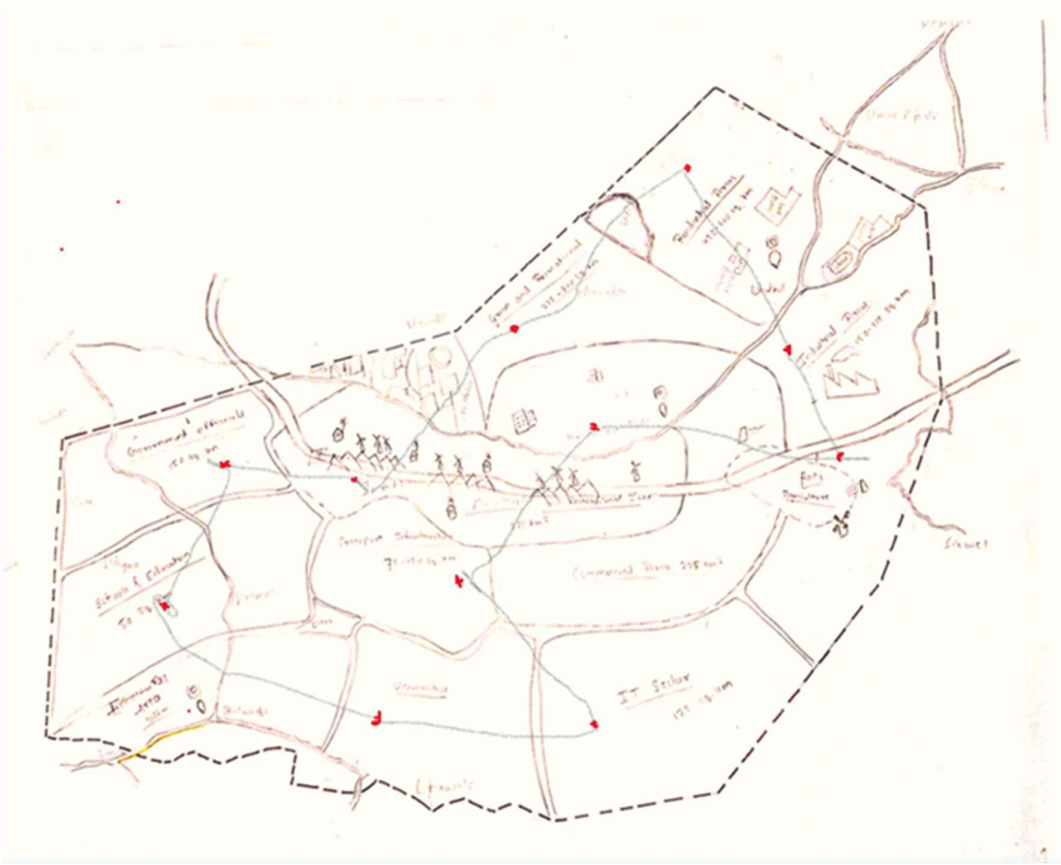
**Sub-task: Measures to be taken for safety of community**

**SDG Goal: 11**

**Target: 11.7**

**Indicator: 11.7**

Efforts to enhance community safety during disasters in Vatsalya align with the sub-targets under SDG 11, specifically SDG 11.7, which focuses on creating safer communities by ensuring access to safe and inclusive green spaces. These spaces are vital for disaster preparedness, as they serve as emergency shelters and areas for community activities, thus promoting both safety and well-being**.**



**Graph Representation:**

| Evacuation point | Node ID | Connections | Travel/Evacuation cost (weight) |
|---|---|---|---|
| Shelter A | A | B,C,D | 2,4,6 |
| Shelter B | B | A,C | 2,3 |
| Shelter C | C | A,B,D | 4,3,1 |
| Shelter D | D | A,C | 6,1 |

**Algorithms:**

**1. Prim's Algorithm:** Optimizing Resource Allocation for Disaster Relief

Use Case: This algorithm is useful in disaster relief scenarios to allocate resources (such as food, medical supplies, etc.) to different locations in an optimal way, minimizing the total cost of resource distribution.

Steps:

Initialize a tree with a starting node.

Add the smallest weight edge that connects a new node to the tree.

Repeat until all nodes are connected.

Graph Example: Nodes: Emergency shelters, hospitals, evacuation centers. Edges: Roads or paths between these locations with corresponding resource allocation costs.

- **Time Complexity**:
  - **Adjacency Matrix**: O(V2)O(V^2)O(V2)
  - **Adjacency List with Min-Heap**: O(Elog⁡V)O(E \log V)O(ElogV)
- **Efficiency**:
  - **Reason**:
    - In the case of sparse graphs (fewer edges compared to nodes), the adjacency list with a priority queue is more efficient with O(Elog⁡V)O(E \log V)O(ElogV).
    - **Prim's** is not ideal for very large graphs or highly dynamic scenarios, as it still requires updating and checking many edges. However, it is excellent for connecting nodes with minimal total cost, especially when the number of edges is large.

**Implementation:**

**Prim's Algorithm:**

```
#include <iostream>

#include <vector>

#include <algorithm>

#include <climits>


using namespace std;
```

```
struct Edge {
    int u, v, weight;
};

class Graph {
    int V;
    vector<Edge> edges;

public:
    Graph(int V) : V(V) {}

    void addEdge(int u, int v, int weight) {
        edges.push_back({u, v, weight});
    }

    void prim() {
        vector<int> parent(V, -1);
        vector<int> key(V, INT_MAX);
        vector<bool> inMST(V, false);

        key[0] = 0;

        for (int count = 0; count < V - 1; count++) {
            int u = -1;
            for (int i = 0; i < V; i++) {
                if (!inMST[i] && (u == -1 || key[i] < key[u])) {
                    u = i;
                }
            }

            inMST[u] = true;
```

```
            for (auto edge : edges) {
                if (edge.u == u && !inMST[edge.v] && edge.weight < key[edge.v]) {
                    key[edge.v] = edge.weight;
                    parent[edge.v] = u;
                }
            }
        }

        cout << "Minimum Spanning Tree (MST) - Optimal Resource Allocation:\n";
        for (int i = 1; i < V; i++) {
            cout << parent[i] << " - " << i << " (Weight: " << key[i] << ")\n";
        }
    }
};

int main() {
    Graph g(5);
    g.addEdge(0, 1, 2);
    g.addEdge(1, 2, 3);
    g.addEdge(2, 3, 1);
    g.addEdge(3, 4, 5);
    g.addEdge(0, 4, 4);

    g.prim();
    return 0;
}
```

**Expected Output:**

Minimum Spanning Tree (MST) - Optimal Resource Allocation:

0 - 1 (Weight: 2)

1 - 2 (Weight: 3)

2 - 3 (Weight: 1)

3 - 4 (Weight: 5)

**2. BFS:** Identifying the Shortest Evacuation Routes

Use Case: BFS is particularly useful in finding the shortest path in an unweighted graph (where all edges have the same weight), such as when determining the quickest evacuation route from any point to a safe zone.

Steps:

Start from the source node (e.g., a building).

Explore all neighbouring nodes (locations).

Mark the visited nodes and continue exploring until the destination (safe zone) is reached.

Return the shortest path found.

Graph Example: Nodes: Buildings, roads, shelters. Edges: Paths between locations (unweighted for simplicity in BFS).

- **Time Complexity**:
    - O(V+E)O(V + E)O(V+E)
- **Efficiency**:
    - **Reason**:
        - **BFS** is the most efficient when the graph is unweighted (all edges have the same weight or cost) because it explores all neighbors level by level. In disaster evacuation scenarios where all paths have equal importance (such as in an emergency route map), **BFS** provides an optimal solution. However, it is inefficient for weighted graphs, as it doesn't account for varying costs or distances.

- **BFS**'s linear time complexity is excellent for small or sparse unweighted graphs but lacks the flexibility needed for more complex disaster response scenarios.

## Implementation:

### BFS:

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

void bfs(int start, vector<vector<int>>& adj, vector<int>& distance) {
    queue<int> q;
    q.push(start);
    distance[start] = 0;

    while (!q.empty()) {
        int node = q.front();
        q.pop();

        for (int neighbor : adj[node]) {
            if (distance[neighbor] == -1) {  // Not visited
                distance[neighbor] = distance[node] + 1;
                q.push(neighbor);
            }
        }
    }
}

int main() {
    int V = 5;
```

```
    vector<vector<int>> adj(V);

    adj[0].push_back(1);
    adj[1].push_back(0);
    adj[1].push_back(2);
    adj[2].push_back(1);
    adj[2].push_back(3);
    adj[3].push_back(2);
    adj[3].push_back(4);
    adj[4].push_back(3);

    vector<int> distance(V, -1);

    bfs(0, adj, distance);

    cout << "Shortest distances from node 0 to other nodes:\n";
    for (int i = 0; i < V; i++) {
        cout << "Distance to node " << i << ": " << distance[i] << endl;
    }

    return 0;
}
```

**Expected Output:**

Shortest distances from node 0 to other nodes:

Distance to node 0: 0

Distance to node 1: 1

Distance to node 2: 2

Distance to node 3: 3

Distance to node 4: 4

**3.Dijkstra's Algorithm**

- **Purpose**: Finds the shortest path from a source node to all other nodes in a weighted graph.
- **Application**:
    - Identifies the quickest evacuation routes when travel times or distances (weights) between locations vary.
    - Useful for navigating through the city during disasters to minimize evacuation time.
- **Example in Case**:
    - Nodes: Shelters and buildings.
    - Weights: Travel or evacuation costs between locations.
- **Time Complexity**:
  **Adjacency Matrix**: O(V2)O(V^2)O(V2)
  **Adjacency List with Min-Heap**: O(Elog V)O(E \log V)O(ElogV)
- **Efficiency**:

  **Reason**:

**Dijkstra's Algorithm** is the most efficient for **weighted graphs**, especially when edges have varying weights (e.g., evacuation times, road conditions, etc.). It can quickly calculate the shortest path from a source to all other nodes, which is ideal for evacuation or resource distribution where each path has a different cost. The algorithm's time complexity is reasonable even for large graphs when using an adjacency list and priority queue.

- **Implementation:**

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>

using namespace std;

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

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

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

        if (d > dist[node]) continue;

        for (auto& neighbor : graph[node]) {
            int nextNode = neighbor.first;
            int weight = neighbor.second;

            if (dist[node] + weight < dist[nextNode]) {
                dist[nextNode] = dist[node] + weight;
                pq.push({dist[nextNode], nextNode});
            }
        }
    }

    cout << "Shortest distances from node " << start << ":\n";
    for (int i = 0; i < V; i++) {
        cout << "Node " << i << ": " << dist[i] << endl;
    }
}

int main() {
    int V = 4;
    vector<vector<pair<int, int>>> graph(V);

    // Add edges (u, v, weight)
    graph[0].push_back({1, 2});
    graph[0].push_back({2, 4});
    graph[1].push_back({2, 1});
    graph[1].push_back({3, 7});
    graph[2].push_back({3, 3});

    dijkstra(0, graph);

    return 0;
}
```

## **Expected Output**:

Shortest distances from node 0:

Node 0: 0

Node 1: 2

Node 2: 3

Node 3: 6

**4. Kruskal's Algorithm**

- **Purpose**: Constructs a Minimum Spanning Tree (MST) to connect all nodes with the least total cost.
- **Application**:
    - Optimizes resource allocation by connecting shelters, hospitals, and other key locations with minimal cost.
    - Ensures all critical locations are linked, reducing redundancy in resource distribution.
- **Example in Case**:
    - Nodes: Critical infrastructure like shelters and hospitals.
    - Weights: Costs of transporting resources or building links.
- **Time Complexity**:

  O(ElogE)O(E \log E)O(ElogE) or O(ElogV)O(E \log V)O(ElogV) (depending on edge sorting)
- **Efficiency**:

  **Reason**:
    - **Kruskal's Algorithm** is efficient in **MST** construction, which is suitable for scenarios where we need to optimize connections between nodes (such as shelters, hospitals, or evacuation routes) with minimal overall cost. It's most efficient when the graph is sparse, as it works by sorting edges, which can be computationally expensive for dense graphs.
    - **Kruskal's** works well for resource distribution across locations but doesn't directly optimize the shortest evacuation route or real-time network changes.
- **Implementation:**

```
 #include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

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

class DisjointSet {
    vector<int> parent, rank;

public:
    DisjointSet(int n) : parent(n), rank(n, 0) {
        for (int i = 0; i < n; i++) parent[i] = i;
    }

    int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]);
        return parent[x];
    }

    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) parent[rootY] = rootX;
            else if (rank[rootX] < rank[rootY]) parent[rootX] = rootY;
            else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
```

```
        }
    }
};

void kruskal(int V, vector<Edge>& edges) {
    sort(edges.begin(), edges.end());
    DisjointSet ds(V);

    int mstCost = 0;
    cout << "Edges in the Minimum Spanning Tree:\n";
    for (Edge& edge : edges) {
        if (ds.find(edge.u) != ds.find(edge.v)) {
            ds.unite(edge.u, edge.v);
            mstCost += edge.weight;
            cout << edge.u << " - " << edge.v << " (Weight: " << edge.weight <<
")\n";
        }
    }
    cout << "Total cost of MST: " << mstCost << endl;
}

int main() {
    int V = 4;
    vector<Edge> edges = {
        {0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}
    };

    kruskal(V, edges);

    return 0;
}
```

**Expected Output:**

Edges in the Minimum Spanning Tree:

2 - 3 (Weight: 4)

0 - 3 (Weight: 5)

0 - 1 (Weight: 10)

Total cost of MST: 19

**5. Floyd-Warshall Algorithm:**

- **Purpose**: Computes the shortest paths between all pairs of nodes in a graph.
- **Application**:
    - Assists in comprehensive disaster response planning by evaluating all possible paths between locations.
    - Helps in identifying bottlenecks or critical links in the city's transportation network.
- **Example in Case**:
    - Nodes: Shelters, hospitals, evacuation centers.
    - Weights: Travel costs or evacuation times between every pair of locations.
- **Time Complexity**:

  O(V3)O(V^3)O(V3)
- **Efficiency**:

  **Reason**:
    - **Floyd-Warshall** computes the shortest paths between all pairs of nodes, which is highly efficient for small graphs. However, its **cubic time complexity** makes it impractical for larger graphs or networks, such as those in a city with many shelters, hospitals, and evacuation centers.
    - While it's useful for comprehensive planning (like calculating paths between all pairs of critical locations), it becomes inefficient when scaling to large cities due to its high computational overhead.
- **Implementation:**

```
#include <iostream>
#include <vector>
#include <climits>
```

```
using namespace std;

void floydWarshall(vector<vector<int>>& graph) {
   int V = graph.size();
   vector<vector<int>> dist = graph;

   for (int k = 0; k < V; k++) {
      for (int i = 0; i < V; i++) {
         for (int j = 0; j < V; j++) {
            if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX) {
               dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }
         }
      }
   }

   cout << "Shortest distances between every pair of nodes:\n";
   for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
         if (dist[i][j] == INT_MAX) cout << "INF ";
         else cout << dist[i][j] << " ";
      }
      cout << endl;
   }
}

int main() {
   vector<vector<int>> graph = {
      {0, 3, INT_MAX, 7},
      {8, 0, 2, INT_MAX},
      {5, INT_MAX, 0, 1},
      {2, INT_MAX, INT_MAX, 0}
   };

   floydWarshall(graph);

   return 0;
}
```

**Expected Output**:

Shortest distances between every pair of nodes:

0 3 5 7

8 0 2 3

5 8 0 1

2 5 7 0

**Comparison Table for above cases:**

| **Algorithm** | **Purpose** | **Use Case in Disaster Management** | **Graph Type** | **Time Complexity** | **Limitations** |
|---|---|---|---|---|---|
| Prim's Algorithm | Minimum Spanning Tree (MST) for resource optimization | Connects shelters and key locations with minimal resource allocation cost | Weighted, connected | O(V2) (adj. matrix) or O(ElogV)O(E \log V)O(ElogV) (adj. list with min-heap) | Requires connected graphs; cannot handle dynamic costs effectively. |
| BFS | Finds the shortest path in unweighted graphs | Identifies shortest evacuation routes in uniform-cost scenarios | Unweighted | O(V+E) | Not suitable for weighted graphs where travel costs vary. |
| Dijkstra's Algorithm | Shortest path from a single source to all nodes | Finds optimal evacuation routes when travel costs vary | Weighted, non-negative | O(V2) (adj. matrix) or O(ElogV)O(E \log V)O(ElogV) (adj. list with min-heap) | Cannot handle graphs with negative weights. |
| Kruskal's Algorithm | MST using edge sorting | Links all critical infrastructure with minimal cost | Weighted, connected | O(ElogE) | Does not prioritize resource distribution to specific nodes; assumes |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | all nodes are equally important. |
| Floyd-Warshall | Shortest paths between all pairs of nodes | Plans comprehensive disaster response by analyzing all routes | Weighted, any graph | O(V3) | High time complexity for large graphs; cannot handle negative cycles. |

**Conclusion:**

**Dijkstra's Algorithm** is the most efficient for disaster management scenarios where evacuation routes need to be quickly calculated in a weighted graph. It balances efficiency, scalability, and flexibility in real-world situations, making it the best choice for optimizing evacuation and resource distribution paths.