case: City Management

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

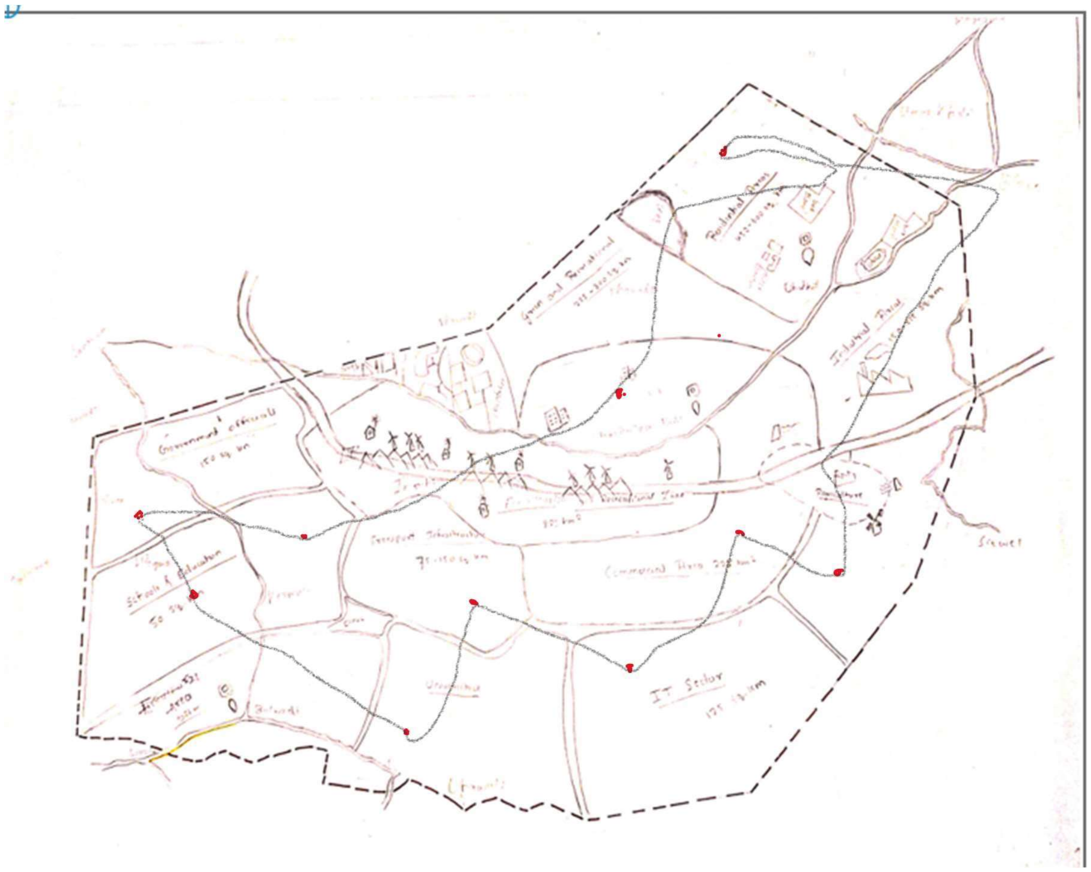
SDG Goal: 11 Business

Target: 11.7

Indicator: 11.7

Overview:
Tourism is a cornerstone of Vatsalya Nagar's city plan, emphasizing cultural heritage, recrea onal ac vi es, and sustainable development. The city's layout integrates parks, resorts, and a rac ons celebra ng local tradi ons. These efforts aim to:

- Drive economic growth.
- Encourage social interac on.
- Posi on Vatsalya Nagar as a cultural and recrea onal hub.



Alignment with SDG 11.3:

This ini a ve supports SDG 11.3 by promo ng inclusive and sustainable urbaniza on through par cipatory planning and management, ensuring a well-integrated and sustainable urban environment for both residents and visitors.

| Tourist Hub | Node in | Connec ons | Travel cost (Weight) |
|---|---|---|---|
| Central Park | A | B,C,D | 5,3,6 |
| Riverside Resort | B | A,C | 5,4 |
| Heritage Musuem | C | A,B,D | 3,4,2 |
| Green Retreat | D | A,C | 6,2 |

Explana on of Columns:

- Tourist Hub: Major tourist a rac ons in Vatsalya Nagar.
- Node ID: Represents the a rac on as a node in the graph.
- Connec ons: Indicates connected tourist hubs.
- Travel Cost (Weight): Cost of traveling between hubs (e.g., in terms of me or fuel consump on).

Algorithms for Tourism Op miza on

1. Dijkstra's Algorithm

Use Case:
Op mize tourist routes by calcula ng the shortest path between a rac ons to minimize travel me and fuel consump on.

Steps:

1. Set the star ng point (e.g., a tourist’s hotel or current loca on).
2. Assign ini al distances to all nodes (infinity for unvisited nodes, 0 for the star ng node).
3. Update distances for all neighbors of the current node.
4. Mark the node as visited and repeat for the next nearest unvisited node.
5. Con nue un l the des na on (tourist site) is reached.

Applica on in Vatsalya Nagar:

- Nodes: Tourist a rac ons, parks, resorts.
- Edges: Roads with weights represen ng travel me or fuel cost.
- Output: Op mal tourist route.

2. Breadth-First Search (BFS)

Use Case:
Map connec vity between tourist des na ons and recommend guided tour sequences.

Steps:

1. Start from a node (e.g., a key a rac on).
2. Traverse all neighbouring nodes.
3. Mark visited nodes to avoid redundancy.
4. Generate a sequence of connected a rac ons.

Applica on in Vatsalya Nagar:

- Nodes: Tourist a rac ons.
- Edges: Connec ons or paths between a rac ons.
- Output: Guided tour sequences op mizing visita on order.

3. Prim's Algorithm (for Minimum Spanning Tree)

Use Case:
Op mize the development of infrastructure by connec ng all tourist hubs with the minimum total cost.

Steps:

1. Start with any node as the root of the tree.
2. Add the smallest edge that connects a node in the tree to a node outside it.
3. Repeat un l all nodes are connected.

Applica on in Vatsalya Nagar:

- Nodes: Tourist a rac ons.
- Edges: Travel paths with weights ( me, cost, or distance).
- Output: Minimum infrastructure cost to connect all hubs efficiently

4. Depth-First Search (DFS)

- Use Case:
  Explore possible paths between tourist hubs for crea ng exploratory or adventurous tour routes.
- Steps:
- Start at a node and mark it as visited.
- Recursively visit all unvisited neighbors.
- Backtrack when no unvisited neighbors remain.
- Applica on in Vatsalya Nagar:
- Nodes: Tourist a rac ons.

- Edges: Connec ons between a rac ons.
- Output: All possible paths, useful for planning exploratory tours.

5.Floyd-Warshall Algorithm

- Use Case:
  Iden fy shortest paths between all pairs of tourist hubs for comprehensive travel route planning.
- Steps:
- Ini alize a matrix where dist[i][j] represents the direct distance between nodes i and j.
- For each node k, update the matrix as dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]).
- Repeat for all nodes to compute shortest paths.
- Applica on in Vatsalya Nagar:
- Nodes: Tourist hubs.
- Edges: Travel paths with weights.
- Output: Matrix of shortest distances between all tourist hubs.

Code Implementa ons:

Dijkstra's Algorithm:

```
#include <iostream>

#include <vector>

#include <queue>

#include <climits>


using namespace std;


void dijkstra(int src, vector<vector<pair<int, int>>>& graph, vector<int>&

dist) {     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;

pq.push({0, src});     dist[src] = 0;


    while (!pq.empty()) {

        int d = pq.top().first;
        int u = pq.top().second;
pq.pop();

        if (d > dist[u]) con nue;

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

int main() {     int V = 5;
vector<vector<pair<int, int>>> graph(V);

    // Example edges: Add connec ons with weights (travel mes)
graph[0].push_back({1, 10});     graph[1].push_back({2, 5});
graph[2].push_back({3, 8});     graph[3].push_back({4, 3});
graph[0].push_back({4, 15});

    vector<int> dist(V, INT_MAX);
dijkstra(0, graph, dist);

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

2. BFS for Connec vity Mapping:

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

using namespace std;

void bfs(int start, vector<vector<int>>& adj, vector<int>& visited) {
queue<int> q;
  q.push(start);
visited[start] = 1;

  cout << "Tourist sequence: ";
while (!q.empty()) {        int
node = q.front();
    q.pop();      cout
<< node << " ";

    for (int neighbor : adj[node]) {
if (!visited[neighbor]) {
visited[neighbor] = 1;
        q.push(neighbor);
      }
    }
  }
  cout << endl;
}
```

```
int main() {     int V = 5;
vector<vector<int>> adj(V);

    // Example connec ons between a rac ons
    adj[0].push_back(1);
adj[1].push_back(2);
adj[2].push_back(3);
adj[3].push_back(4);

    vector<int> visited(V, 0);
bfs(0, adj, visited);

    return 0;
}
```

3.Prim's Algorithm:

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

void prim(int V, vector<vector<pair<int, int>>>& graph) {
vector<int> key(V, INT_MAX);     vector<bool> inMST(V, false);
vector<int> parent(V, -1);     priority_queue<pair<int, int>,
vector<pair<int, int>>, greater<>> pq;

    pq.push({0, 0});
key[0] = 0;     while
(!pq.empty()) {         int
u = pq.top().second;
pq.pop();
inMST[u] = true;
```

```
        for (auto& edge : graph[u]) {              int
v = edge.first, weight = edge.second;
if (!inMST[v] && weight < key[v]) {
key[v] = weight;                 pq.push({key[v],
v});                 parent[v] = u;
            }
        }
    }

    cout << "Edges in the MST:\n";
    for (int i = 1; i < V; i++) {          cout
<< parent[i] << " - " << i << "\n";
    }
}

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

    graph[0] = {{1, 5}, {2, 3}, {3,
6}};     graph[1] = {{0, 5}, {2, 4}};
graph[2] = {{0, 3}, {1, 4}, {3, 2}};
graph[3] = {{0, 6}, {2, 2}};
prim(V, graph);     return 0;
}
```

4.Depth First Search

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;
```

```
//      Perform      Depth-First      Search      void      dfs(int      start,
vector<vector<int>>& adj, vector<bool>& visited) {      stack<int>
s;
   s.push(start);

   cout << "Tourist sequence (DFS):
";    while (!s.empty()) {        int node
= s.top();
       s.pop();

       if (!visited[node]) {
cout << node << " ";
visited[node] = true;
       }

       // Add unvisited neighbors to the
stack         for (int neighbor : adj[node]) {
if (!visited[neighbor]) {
              s.push(neighbor);
          }
       }
   }
   cout << endl;
}

int main() {     int V = 4;  // Number
of tourist hubs
vector<vector<int>> adj(V);
```

```
    // Example connec ons between a rac ons (graph edges)     adj[0] = {1, 2};  //
Central Park connected to Riverside Resort and Heritage Museum     adj[1] = {0, 2};
// Riverside Resort connected to Central Park and Heritage Museum
    adj[2] = {0, 1, 3};  // Heritage Museum connected to Central Park, Riverside Resort, and
Green Retreat     adj[3] = {2};  // Green Retreat connected to Heritage Museum

    vector<bool> visited(V, false);

    // Perform DFS star ng from node 0 (Central Park)
    dfs(0, adj, visited);

    return 0;
}
```

5.Floyd-Warshall Algorithm

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

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

Conclusion:

- Dijkstra's Algorithm: Provides op mal tourist routes, reducing travel me and enhancing visitor experience.

- BFS: Ensures seamless connec vity and planned guided tours.

Prim’s Algorithm

- Purpose: Constructs a minimum spanning tree to connect all a rac ons with minimal total travel cost.
- Advantages: Ensures all hubs are connected op mally without cycles.
- Use Case in Vatsalya Nagar: Helps design efficient infrastructure connec ng all major hubs.

Depth-First Search (DFS)

- Purpose: Explores connected a rac ons by diving deeper into one path before backtracking.
  -  Advantages: Useful for finding paths and exploring connec vity.
- Use Case in Vatsalya Nagar: Provides an alterna ve guided tour sequence focusing on depthfirst explora on.
- Floyd-Warshall Algorithm
- Purpose: Finds the shortest paths between all pairs of a rac ons.
- Advantages: Comprehensive and ideal for smaller graphs.
- Use Case in Vatsalya Nagar: Analyzes overall accessibility between hubs for strategic planning.

## Business case: City Management

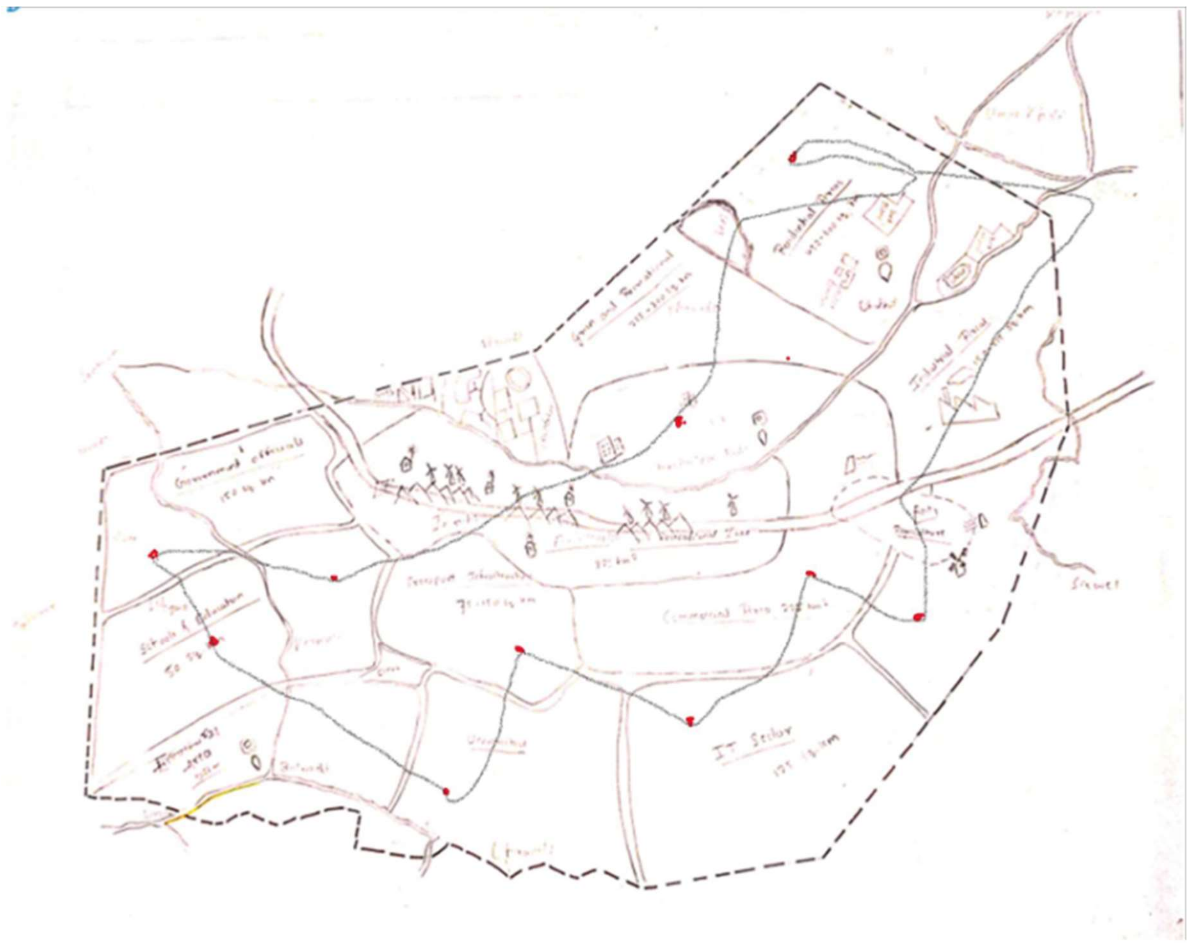
Sub-task: Land Leasing and Real Estate

SDG Goal: 11

Target: 11.3

Indicator: 11.3

Overview: Land leasing and real estate development form a significant part of Vatsalya Nagar's economic framework, providing a steady revenue stream and enabling sustainable urban expansion. By leasing land to private en es for residen al, commercial, and industrial purposes, the city aims to balance economic growth with inclusivity. Real estate ini a ves include affordable housing schemes, ensuring equitable access to homes for all income groups while fostering a vibrant community. This approach creates opportuni es for investment and employment, enhancing the city's appeal as a des na on for businesses and residents alike.



SDG Goal Alignment: SDG 11.3: Enhance inclusive and sustainable urbaniza on and capacity for par cipatory, integrated, and sustainable human se lement planning and management in all countries.

Algorithms and Their Use Cases:

1. Kruskal's Algorithm: Op mize U lity Networks
   - o Use Case: Op mize land parcel connec ons for u li es such as electricity, water, and internet by building a minimal cost-spanning network. This ensures cost-effec ve and efficient service delivery across residen al, commercial, and industrial zones. o Steps:
     1. Sort all edges (land parcel connec ons) by increasing weight (u lity costs).
     2. Add the smallest edge to the network, ensuring it doesn't form a cycle.
     3. Repeat un l all parcels are connected in a minimal spanning tree.
   - o Graph Representa on:
     -  Nodes: Land parcels.
     -  Edges: U lity connec ons with weights represen ng costs

2 Union-Find Data Structure: Manage Land Ownership

- Use Case: Efficiently manage land ownership and lease agreements by dynamically linking owner-user rela onships. This ensures real- me tracking of land u liza on and helps prevent disputes.
- Steps:
- Assign each land parcel to an ini al owner (union opera on).
- For lease agreements, dynamically connect the owner to the user (find opera on).
- Update changes in ownership or leasing status efficiently.
- Graph Representa on:
- Nodes: Land parcels, owners, users.
- Edges: Rela onships (ownership, leasing agreements).

3. Prim's Algorithm: Build Resilient Infrastructure Networks

- Use Case: Construct cost-effec ve and resilient infrastructure networks (e.g., roads, pipelines) by minimizing the total cost of connec ng various urban areas.
- Steps:
- Start from an ini al land parcel (node).
- Add the smallest weight edge to connect a new parcel to the network.
- Repeat un l all parcels are connected.
- Graph Representa on:
- Nodes: Urban areas.
- Edges: Infrastructure connec ons with weights represen ng costs.

4. Bellman-Ford Algorithm: Assess Accessibility Costs

Use Case:

Calculate the minimum cost to reach all urban zones from a central hub (e.g., city center) in the presence of varying costs or nega ve weights (e.g., subsidies)

5. Ford-Fulkerson Algorithm: Maximal U lity Alloca on

Use Case:

Op mize the alloca on of u li es (e.g., water, electricity) to urban areas by maximizing the flow through the network of supply points and urban zones.

Steps:

1. Represent the u lity network as a directed graph where edges represent capacity.
2. Use BFS or DFS to find augmen ng paths.
3. Increase the flow along augmen ng paths un l no more exist.
4. Output the maximum flow, ensuring op mal resource distribu on.

Graph Representa on:

- Nodes: Supply points and urban zones.
- Edges: Connec ons with capaci es represen ng the maximum u lity that can be supplied.

```
#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 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});
    }
    int find(vector<int>& parent, int i)
{         if (parent[i] == i)
return i;
        return parent[i] = find(parent, parent[i]);
    }

    void unionSets(vector<int>& parent, vector<int>& rank, int x, int y) {
int rootX = find(parent, x);

        int rootY = find(parent, y);

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

    void kruskal() {
sort(edges.begin(), edges.end());
vector<int> parent(V);
vector<int> rank(V, 0);

        for (int i = 0; i < V; i++)
parent[i] = i;
```

```
        cout << "Minimum Spanning Tree (MST) for U lity
Networks:\n";        for (const Edge& edge : edges) {            int u =
find(parent, edge.u);            int v = find(parent, edge.v);
            if (u !=
v) {
                cout << edge.u << " - " << edge.v << " (Cost: " << edge.weight << ")\n";
unionSets(parent, rank, u, v);
            }
        }
    }
};

int main() {
Graph g(4);
    g.addEdge(0, 1, 5);
    g.addEdge(1, 2, 4);
    g.addEdge(2, 3, 6);
    g.addEdge(0, 3, 3);

    g.kruskal();
return 0;
)
```

2. Union-Find for Land Ownership:

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

using namespace std;
```

```
class UnionFind {
vector<int> parent, rank;

public:
   UnionFind(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 unionSets(int x, int y)
{        int rootX = find(x);
int rootY = find(y);

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

int main() {
    UnionFind uf(5);

    uf.unionSets(0, 1);
uf.unionSets(1, 2);

    cout << "Land ownership tracking:\n";     for (int i = 0; i < 5; i++) {         cout << "Owner of parcel " <<
i << ": " << uf.find(i) << endl;
    }

    return 0;
}
```

3.Prim,s Algorithm

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

int findMinKey(vector<int>& key, vector<bool>& mstSet, int V)
{     int min = INT_MAX, minIndex;

    for (int v = 0; v < V; v++) {
if (!mstSet[v] && key[v] < min) {
min = key[v];             minIndex = v;
        }
    }

    return minIndex;
}
```

```
void printMST(vector<int>& parent, vector<vector<int>>& graph, int V) {
cout << "Edge \tWeight\n";
   for (int i = 1; i < V; i++) {         cout << parent[i] << " - " << i <<
"\t" << graph[i][parent[i]] << endl;
   }
}

void primMST(vector<vector<int>>& graph, int V) {     vector<int> parent(V);
// Array to store constructed MST     vector<int> key(V, INT_MAX); // Key
values used to pick minimum weight edge     vector<bool> mstSet(V, false); // To
represent the set of ver ces included in MST

   key[0] = 0;       // Make the first vertex as root
parent[0] = -1;   // First node is always the root of the MST

   for (int count = 0; count < V - 1; count++) {
int u = findMinKey(key, mstSet, V);

       mstSet[u] = true;

       for (int v = 0; v < V; v++) {             if (graph[u][v] &&
!mstSet[v] && graph[u][v] < key[v]) {
             parent[v] = u;
key[v] = graph[u][v];
          }
       }
   }

   printMST(parent, graph, V);
}
```

```
int main() {
vector<vector<int>> graph = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    int V = graph.size();     cout << "Minimum Spanning Tree
using Prim's Algorithm:\n";     primMST(graph, V);

    return 0;
}
```

4.Bellman-Ford Algorithm

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

using namespace std;

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

    for (int i = 0; i < V - 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 << "Minimum costs from source:\n";
    for (int i = 0; i < V; i++) {
        cout << "To node " << i << ": " << dist[i] << endl;
    }
}

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

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

5.Ford-Fulkerson Algorithm

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

using namespace std;

bool bfs(vector<vector<int>>& rGraph, int s, int t, vector<int>&
parent) {     int V = rGraph.size();     vector<bool> visited(V, false);
queue<int> q;
```

```
   q.push(s);
visited[s] = true;
parent[s] = -1;

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

       for (int v = 0; v < V; v++) {
if (!visited[v] && rGraph[u][v] > 0) {
parent[v] = u;                visited[v] = true;
if (v == t) return true;
               q.push(v);
           }
       }
   }
   return false;
}

int fordFulkerson(vector<vector<int>>& graph, int s, int
t) {     int V = graph.size();     vector<vector<int>> rGraph
= graph; // Residual graph     vector<int> parent(V);     int
maxFlow = 0;

   while (bfs(rGraph, s, t, parent)) {
int pathFlow = INT_MAX;

       for (int v = t; v != s; v = parent[v]) {
int u = parent[v];            pathFlow =
min(pathFlow, rGraph[u][v]);
```

```
        }

        for (int v = t; v != s; v =
parent[v]) {            int u = parent[v];
rGraph[u][v] -= pathFlow;
rGraph[v][u] += pathFlow;
        }

        maxFlow += pathFlow;
    }

    return maxFlow;
}

int main() {
vector<vector<int>> graph = {
        {0, 10, 5, 15},
        {0, 0, 4, 0},
        {0, 0, 0, 10},
        {0, 0, 0, 0}
    };

    cout << "Maximum u lity alloca on: " << fordFulkerson(graph, 0, 3) << endl;
    return 0;
}
```

Conclusion:

- Kruskal's Algorithm helps op mize u lity networks, minimizing the cost of providing basic services like water, electricity, and internet.
- Union-Find Data Structure efficiently tracks land ownership and lease agreements, ensuring transparency and reducing disputes.
- Ford-Fulkerson: Maximizes u lity alloca on to urban zones efficiently.

- . Bellman-Ford: Computes accessibility costs accurately, even with varying or nega ve weights.
- prim's Algorithm is a powerful tool for building cost-effec ve and resilient infrastructure networks, ensuring minimal total cost while connec ng all urban areas. It incrementally selects the least costly connec ons, making it ideal for designing transporta on routes, pipelines, and u lity systems. This approach supports sustainable urban planning, aligns with SDG Goal 11.3, and fosters inclusive development by enabling efficient and scalable infrastructure for ci es like Vatsalya Nagar.

Business case: City Management

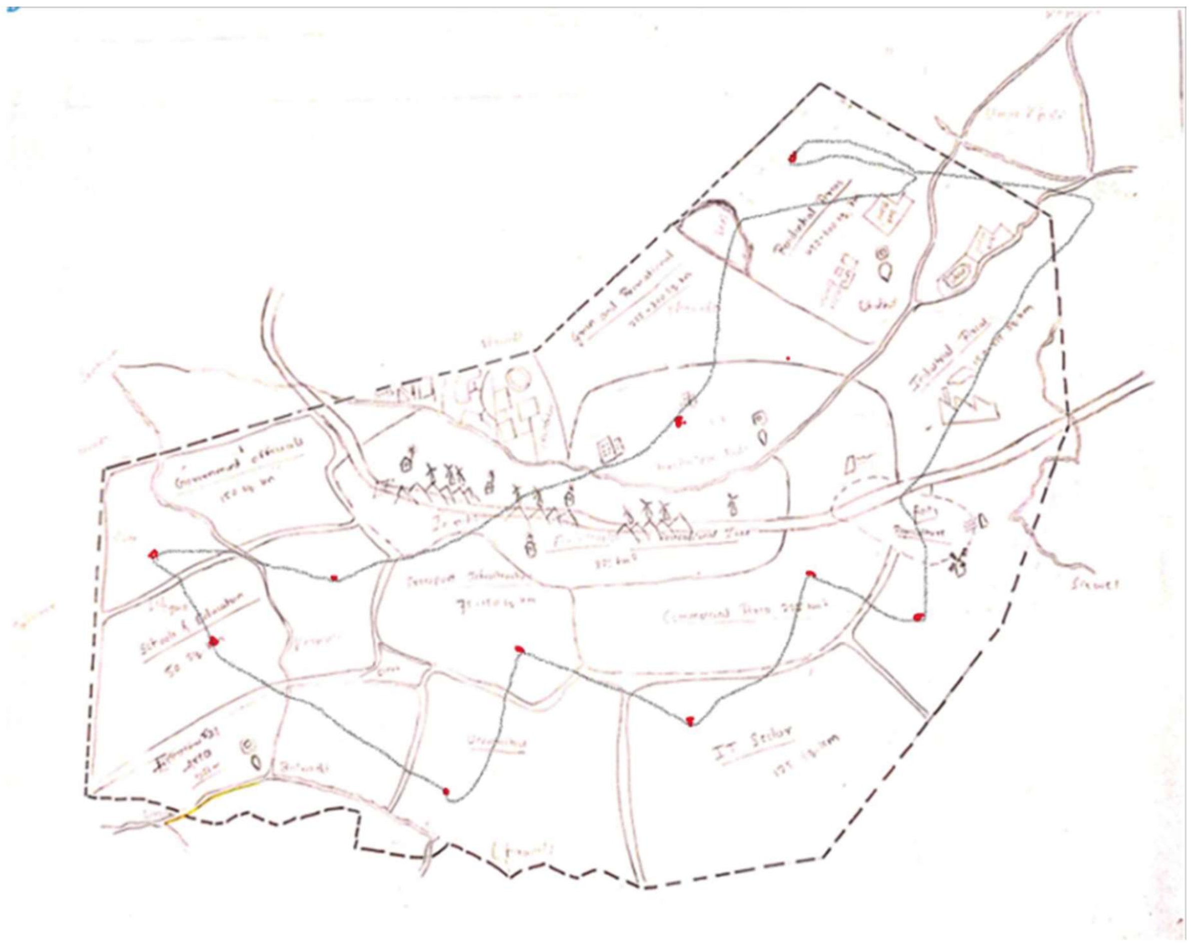
Sub-task: Smart City Services

SDG Goal: 11

Target: 11.2

Indicator: 11.2

Overview: Smart city services are integral to Vatsalya Nagar's vision of a technologically advanced and efficient urban ecosystem. By integra ng IoT, data management, and digital infrastructure, the city aims to enhance governance, improve service delivery, and promote transparency. These services include smart infrastructure management, e-governance pla orms, and real- me monitoring systems that op mize resource use. With an annual revenue poten al of ₹1,200 crore, smart city services also foster innova on and create opportuni es for businesses specializing in technology, making Vatsalya Nagar a model for modern urban living.



SDG Goal Alignment: SDG 11.2: Provide access to safe, affordable, accessible, and sustainable transport systems for all, improving road safety, notably by expanding public transport, with special a en on to the needs of those in vulnerable situa ons, women, children, persons with disabili es, and older persons.

Algorithms and Their Use Cases:

1. Heap Sort: Priori ze Smart City Services
   - Use Case: Priori ze tasks such as garbage collec on, energy distribu on, and emergency responses based on urgency and resource availability.
   - Steps:
     1. Insert all tasks into a max heap, where the priority of a task is the key.
     2. Extract the highest-priority task from the heap.
     3. Repeat un l all tasks are completed in order of priority.
   - Implementa on Scenario: Ensures cri cal tasks (e.g., emergency services) are handled first, improving service efficiency.
2. Binary Search Tree (BST): Efficient Data Storage and Retrieval

- o Use Case: Store and retrieve smart city data such as sensor readings, service logs, or ci zen reports efficiently.
- o Steps:
  1. Insert each data point (e.g., mestamped sensor readings) into the BST.
  2. Search for specific data points or ranges of data quickly.
  3. Perform in-order traversal for sorted data analysis.
- o Implementa on Scenario: Enables fast data access for real- me decision-making and analy cs.

3. Bubble Sort: Low-Priority Task Management

- o Use Case: Handle low-priority tasks such as ranking ci zen feedback forms or sor ng non-cri cal no fica ons.
- o Steps:
  1. Iterate through the list of tasks, comparing adjacent items.
  2. Swap items if they are out of order.
  3. Repeat un l the en re list is sorted. o Implementa on

Scenario: Simple sor ng for tasks that do not require advanced algorithms or real- me performance.

- o 4. Dijkstra's Algorithm: Op mize Public Transport Routes
- o Use Case: Find the shortest and safest routes for public transport, ensuring reduced travel me and be er connec vity across the city. Steps:
- o Ini alize distances from the source to all nodes as infinity, except the source itself. o Use a priority queue to explore the nearest node and update distances for its neighbors.
- o Repeat un l all nodes are visited. o Output the shortest paths.

5. A Search Algorithm*: Op mize Emergency Services

Use Case: Efficiently navigate emergency vehicles through the city by considering realme traffic and heuris c distances. Steps:

1. Maintain open and closed sets for nodes.
2. Use a heuris c (e.g., straight-line distance) to es mate cost to the target.
3. Priori ze nodes in the open set by total es mated cost.
4. Expand nodes and update costs itera vely.

Code Implementa on:

Code Implementa on:

1. Heap Sort for Task Priori za on:

```
import heapq


def priori ze_tasks(tasks):
    # Convert task list to a max-heap by nega ng priori es
max_heap = [(-priority, task) for task, priority in tasks]
    heapq.heapify(max_heap)


    print("Task Priority Order:")
while max_heap:
        priority, task = heapq.heappop(max_heap)
print(f"Task: {task}, Priority: {-priority}")


# Example tasks with priori es tasks = [("Garbage Collec on", 3), ("Energy Distribu on",
5), ("Emergency Response", 10)] priori ze_tasks(tasks)
```

2. Binary Search Tree for Data Management:

```
class Node:
    def __init__(self, key):
        self.key = key
self.le = None
self.right = None


class BST:     def
__init__(self):
self.root = None
def insert(self, key):
if not self.root:
self.root =
Node(key)
else:
            self._insert(self.root, key)


    def _insert(self, node, key):
if key < node.key:             if
node.le is None:
node.le = Node(key)
            else:
self._insert(node.le , key)
else:
            if node.right is None:
node.right = Node(key)
            else:
                self._insert(node.right, key)


    def in_order_traversal(self, node):
        if node:
            self.in_order_traversal(node.le    )
print(node.key,              end="              ")
self.in_order_traversal(node.right)


# Example usage
bst = BST() data_points = [15, 10, 20,
8, 12, 16, 25] for data in data_points:
bst.insert(data)

print("In-Order Traversal (Sorted Data):")
bst.in_order_traversal(bst.root)
06330
066
```

3.Bubble-sort

```
Algorithm def
bubble_sort(arr):     n =
len(arr)     for i in
range(n - 1):
        # Last i elements are already sorted, no need to compare them
        for j in range(n - i - 1):
if arr[j] > arr[j + 1]:
                # Swap if the elements are in the wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]         print(f"Itera on
{i + 1}: {arr}")  # Debugging to show progress


# Example usage: Sor ng ci zen feedback scores
feedback_scores = [8, 5, 7, 2, 9, 3] print("Original
Feedback Scores:", feedback_scores)


bubble_sort(feedback_scores)


print("Sorted Feedback Scores:", feedback_scores)
```

4. Dijkstra's Algorithm

```
import heapq


def dijkstra(graph, start):     n = len(graph)
distances = [float('inf')] * n     distances[start] =
0     priority_queue = [(0, start)]  # (distance,
node)     while priority_queue:

        current_distance, current_node = heapq.heappop(priority_queue)


        if current_distance > distances[current_node]:
            con nue
        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight            if distance
< distances[neighbor]:                distances[neighbor] =
distance                heapq.heappush(priority_queue, (distance,
neighbor))

    return distances

# Example: Graph as an adjacency list
# Nodes represent city areas; edges represent public transport routes with weights as travel mes
graph = [
    [(1, 5), (2, 10)],  # Node 0
    [(0, 5), (2, 3)],   # Node 1
    [(0, 10), (1, 3)]   # Node 2
]

start_node = 0 print("Shortest distances from node 0:",
dijkstra(graph, start_node)
```

5. A Search Algorithm from heapq

```
import heappop, heappush

def a_star(graph, start, goal, h):     open_set = []
heappush(open_set, (0, start))     g_scores =
{node: float('inf') for node in graph}
g_scores[start] = 0     came_from = {}

    while open_set:
        current = heappop(open_set)[1]

        if current == goal:
```

```
      path = []           while current
in came_from:
path.append(current)               current
= came_from[current]           return
path[::-1]

    for neighbor, weight in graph[current]:
      tenta ve_g_score = g_scores[current] + weight
if tenta ve_g_score < g_scores[neighbor]:
came_from[neighbor] = current
g_scores[neighbor] = tenta ve_g_score
f_score = tenta ve_g_score + h[neighbor]
heappush(open_set, (f_score, neighbor))

  return []

# Example graph and heuris c graph
= {
  0: [(1, 2), (2, 4)],
  1: [(0, 2), (3, 7)],
  2: [(0, 4), (3, 1)],
  3: []
}
heuris c = {0: 5, 1: 3, 2: 2, 3: 0}
start_node = 0 goal_node
= 3

print("Op mal path:", a_star(graph, start_node, goal_node, heuris c))
```

Conclusion:

- Heap Sort ensures smart city services are priori zed based on urgency and resource availability, improving opera onal efficiency.

- Binary Search Tree (BST) provides fast and organized storage for smart city data, suppor ng real- me analy cs and decision-making.

Bubble Sort is intui ve and easy to implement. It works well for small datasets or cases where simplicity is preferred over performance. However, for larger datasets, more efficient algorithms like Quick Sort or Merge Sort are recommended.

Dijkstra's Algorithm:
This algorithm is highly effec ve for finding the shortest path in weighted graphs, making it ideal for op mizing public transporta on routes and traffic management in smart ci es. Its guaranteed accuracy ensures reliable pathfinding in various applica ons, including emergency response and infrastructure planning.

A Search Algorithm
A* combines the benefits of Dijkstra's algorithm and heuris c techniques, making it faster and more efficient for goal-oriented pathfinding. It is par cularly suitable for dynamic smart city environments, where real- me naviga on and adap ve planning are required, such as autonomous vehicles and delivery systems.