**Q4: Shortest Path in a City (Graph)**: Given a city represented as a **graph** with road networks (nodes for intersections, edges for roads), use **Dijkstra’s algorithm** to find the shortest path between any two intersections.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Shortest Path in a City</title>

<style>

body {

font-family: Arial, sans-serif;

text-align: center;

margin: 20px;

}

#graph {

margin-bottom: 20px;

}

table {

margin: 0 auto;

border-collapse: collapse;

}

table th, table td {

border: 1px solid #ddd;

padding: 8px;

}

table th {

background-color: #f4f4f4;

}

#result {

margin-top: 20px;

font-weight: bold;

}

</style>

</head>

<body>

<h1>Shortest Path in a City</h1>

<div id="graph">

<h3>Graph (City Intersections)</h3>

<table>

<thead>

<tr>

<th>From</th>

<th>To</th>

<th>Distance</th>

</tr>

</thead>

<tbody>

<tr><td>A</td><td>B</td><td>4</td></tr>

<tr><td>A</td><td>C</td><td>2</td></tr>

<tr><td>B</td><td>C</td><td>1</td></tr>

<tr><td>B</td><td>D</td><td>5</td></tr>

<tr><td>C</td><td>D</td><td>8</td></tr>

<tr><td>C</td><td>E</td><td>10</td></tr>

<tr><td>D</td><td>E</td><td>2</td></tr>

</tbody>

</table>

</div>

<div>

<label for="start">Start:</label>

<input type="text" id="start" placeholder="Enter Start Node">

<label for="end">End:</label>

<input type="text" id="end" placeholder="Enter End Node">

<button onclick="findShortestPath()">Find Shortest Path</button>

</div>

<div id="result"></div>

<script>

// City graph as an adjacency list

const graph = {

A: { B: 4, C: 2 },

B: { A: 4, C: 1, D: 5 },

C: { A: 2, B: 1, D: 8, E: 10 },

D: { B: 5, C: 8, E: 2 },

E: { C: 10, D: 2 },

};

// Dijkstra's algorithm

function dijkstra(graph, start, end) {

const distances = {};

const visited = new Set();

const previous = {};

const queue = [];

// Initialize distances

for (const node in graph) {

distances[node] = Infinity;

previous[node] = null;

}

distances[start] = 0;

queue.push({ node: start, distance: 0 });

while (queue.length) {

// Sort queue by distance and pick the node with the smallest distance

queue.sort((a, b) => a.distance - b.distance);

const { node } = queue.shift();

if (node === end) break;

if (visited.has(node)) continue;

visited.add(node);

// Update distances for neighbors

for (const neighbor in graph[node]) {

const newDist = distances[node] + graph[node][neighbor];

if (newDist < distances[neighbor]) {

distances[neighbor] = newDist;

previous[neighbor] = node;

queue.push({ node: neighbor, distance: newDist });

}

}

}

// Construct the shortest path

const path = [];

let current = end;

while (current) {

path.unshift(current);

current = previous[current];

}

return distances[end] === Infinity ? null : { distance: distances[end], path };

}

// Handler to find the shortest path

function findShortestPath() {

const start = document.getElementById('start').value.trim();

const end = document.getElementById('end').value.trim();

if (!graph[start] || !graph[end]) {

document.getElementById('result').innerText = 'Invalid start or end node.';

return;

}

const result = dijkstra(graph, start, end);

if (result) {

document.getElementById('result').innerText = `Shortest path from ${start} to ${end}: ${result.path.join(' -> ')} (Distance: ${result.distance})`;

} else {

document.getElementById('result').innerText = `No path found from ${start} to ${end}.`;

}

}

</script>

</body>

</html>