Represent a graph of city using adjacency matrix/adjacency list. Nodes should represent the various landmarks and links should represent the distance between them. Find the shortest path using Dijkstra's algorithm from single source to all destinations.

Code

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

using namespace std;

const int V = 7; // Maximum number of vertices
const int MAX_EDGES = 3; // Maximum edges per vertex

// A structure to represent a weighted edge in the graph
struct Edge {
   int to;     // Destination vertex
   int weight; // Weight of the edge
};

// Adjacency list representation using arrays
Edge graph[V][MAX_EDGES]; // 2D array of edges
int edgeCount[V] = {0};    // Array to keep track of the number of edges for each
vertex

// Function to add an edge
void addEdge(int from, int to, int weight) {
   if (edgeCount[from] < MAX_EDGES) {
      graph[from][edgeCount[from]].to = to;
      graph[from][edgeCount[from]].weight = weight;
      edgeCount[from]++;
   }
}

// Function to perform Dijkstra's algorithm
void dijkstra(int src, int dist[], int parent[]) {
   bool visited[V] = {false}; // Track visited vertices
```

```
    dist[src] = 0; // Distance to the source is 0

    // Priority queue to store vertices based on their distance
    priority_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > > pq;
    pq.push(make_pair(0, src)); // Push the source with distance 0

    while (!pq.empty()) {
        int u = pq.top().second; // Get vertex with minimum distance
        pq.pop();

        if (visited[u]) continue; // Skip if already visited
        visited[u] = true; // Mark the vertex as visited

        // Process all adjacent vertices of u
        for (int i = 0; i < edgeCount[u]; ++i) {
            Edge edge = graph[u][i];
            int v = edge.to;
            int weight = edge.weight;

            // If there's a shorter path to v through u
            if (!visited[v] && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight; // Update the distance
                parent[v] = u; // Store the parent of v
                pq.push(make_pair(dist[v], v)); // Push new distance to the priority queue
            }
        }
    }
}

// Function to print the shortest path from source to destination
void printPath(int dest, const int parent[]) {
    if (parent[dest] == -1) {
        cout << dest << " "; // Base case: if we reach the source
        return;
    }
    printPath(parent[dest], parent); // Recursively print the path
    cout << dest << " "; // Print current destination vertex
```

```
}

int main() {
    // Add undirected edges between vertices
    addEdge(0, 1, 10);  // Edge 0 -> 1 with weight 10
    addEdge(1, 0, 10);  // Edge 1 -> 0 with weight 10 (reverse of above)

    addEdge(0, 3, 10);  // Edge 0 -> 3 with weight 15
    addEdge(3, 0, 10);  // Edge 3 -> 0 with weight 15 (reverse of above)

    addEdge(1, 2, 20);  // Edge 1 -> 2 with weight 20
    addEdge(2, 1, 20);  // Edge 2 -> 1 with weight 20 (reverse of above)

    addEdge(3, 2, 30);  // Edge 3 -> 2 with weight 30
    addEdge(2, 3, 30);  // Edge 2 -> 3 with weight 30 (reverse of above)

    addEdge(1, 4, 15);  // Edge 1 -> 4 with weight 15
    addEdge(4, 1, 15);  // Edge 4 -> 1 with weight 15 (reverse of above)

    addEdge(4, 5, 20);  // Edge 4 -> 5 with weight 20
    addEdge(5, 4, 20);  // Edge 5 -> 4 with weight 20 (reverse of above)

    addEdge(3, 5, 30);  // Edge 3 -> 5 with weight 30
    addEdge(5, 3, 30);  // Edge 5 -> 3 with weight 30 (reverse of above)

    addEdge(6,0,0);

    // User input for source and destination
    int source, destination;
    cout << "Enter source vertex (0 to 6): ";
    cin >> source;
    cout << "Enter destination vertex (0 to 6): ";
    cin >> destination;

    // Validate user input
    if (source < 0 || source >= V || destination < 0 || destination >= V) {
```

```
        cout << "Invalid source or destination vertex. Please enter values between 0 and
5." << endl;
        return 0;
    }

    // Initialize distance and parent arrays
    int dist[V];
    int parent[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX; // Set all distances to infinity
        parent[i] = -1;    // Initialize parent array
    }

    // Perform Dijkstra's algorithm
    dijkstra(source, dist, parent); // Compute shortest paths from source

    // Check if the destination is reachable
    if (dist[destination] == INT_MAX) {
        cout << "Cannot reach destination " << destination << " from source " << source
<< endl;
    } else {
        // Print the minimum distance and path
        cout << "Minimum distance from " << source << " to " << destination << " is: " <<
dist[destination] << endl;
        cout << "Path: ";
        printPath(destination, parent); // Print the path from source to destination
        cout << endl;
    }

    return 0;
}
```

```
Enter source vertex (0 to 6): 0
Enter destination vertex (0 to 6): 5
Minimum distance from 0 to 5 is: 40
Path: 0 3 5

--------------------------------
Process exited after 6.136 seconds with return value 0
Press any key to continue . . .
```