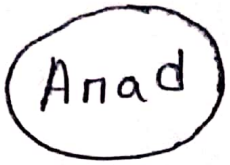
Given,

State	h(n)	State	h(n)
Arad	36	Mehadia	241
Buchanest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	pitesti	100
Efonie	161	Rimnicu vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	zerind	374

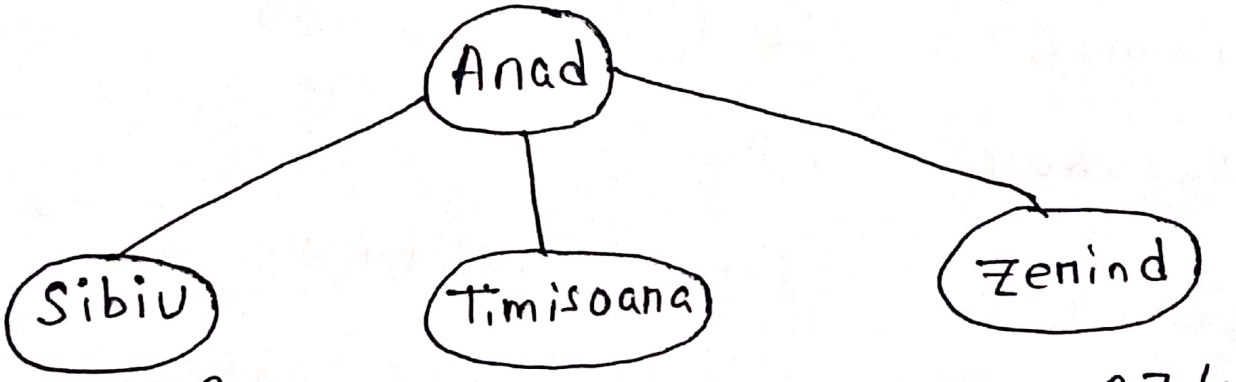
Steps to go Bucharest from Arad:

Step 1:



0+366 = 366

Step 2: Expanding Arad, we get:

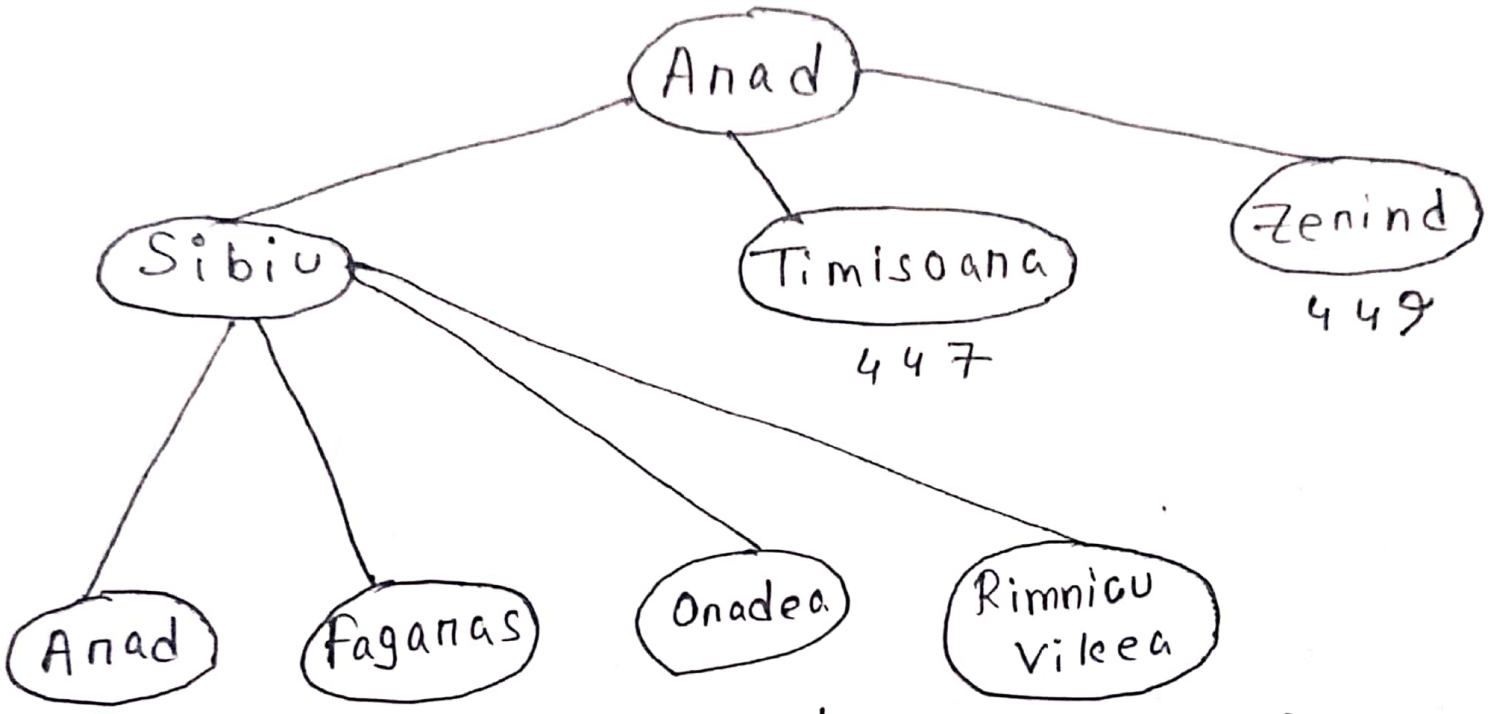


140+253
= 393
(min)

118+329
= 447

75+374
= 449

Step 3: Expanding Sibiu, we get,



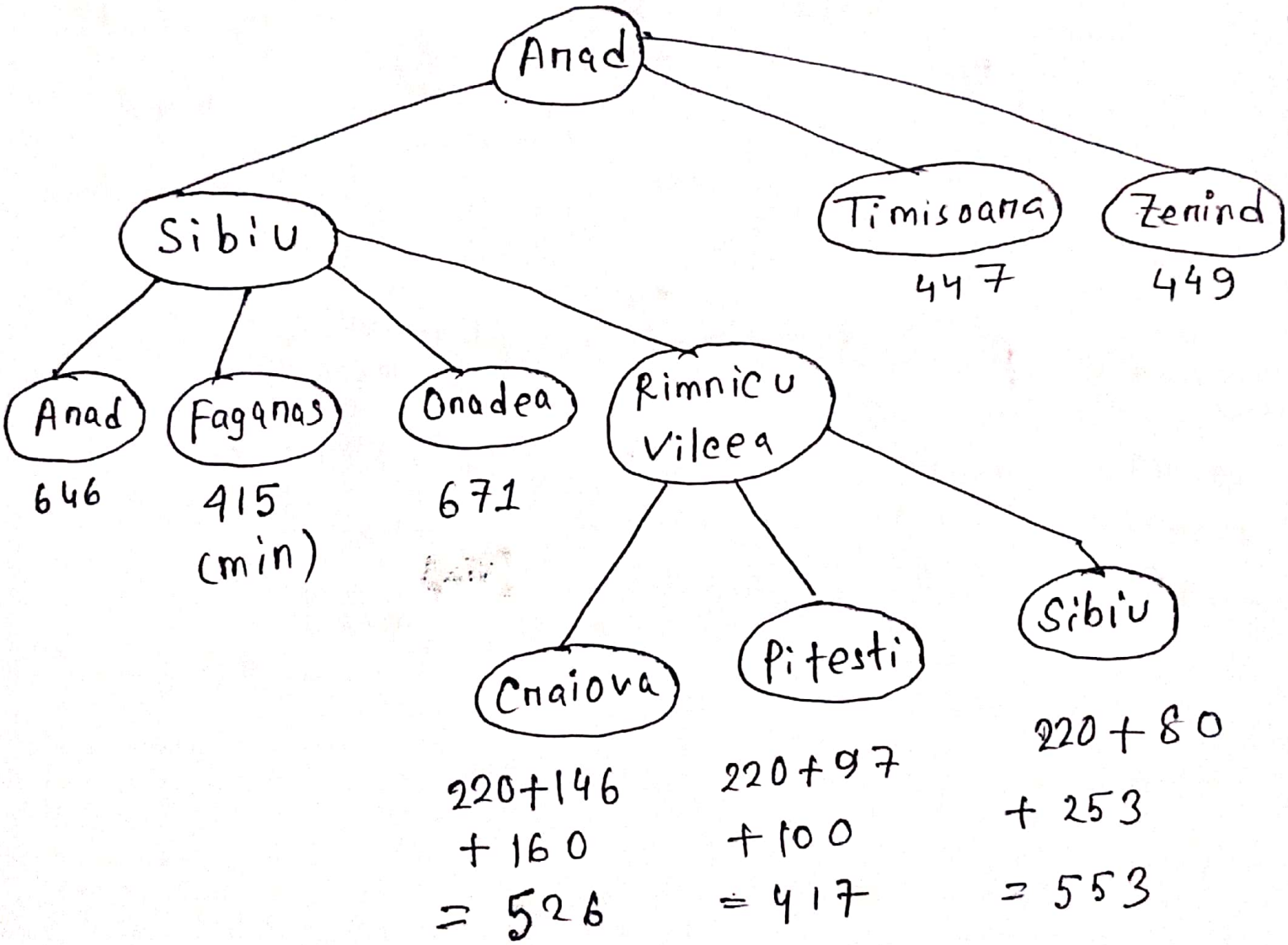
140+140
+366
=646

140+99
+176
=415

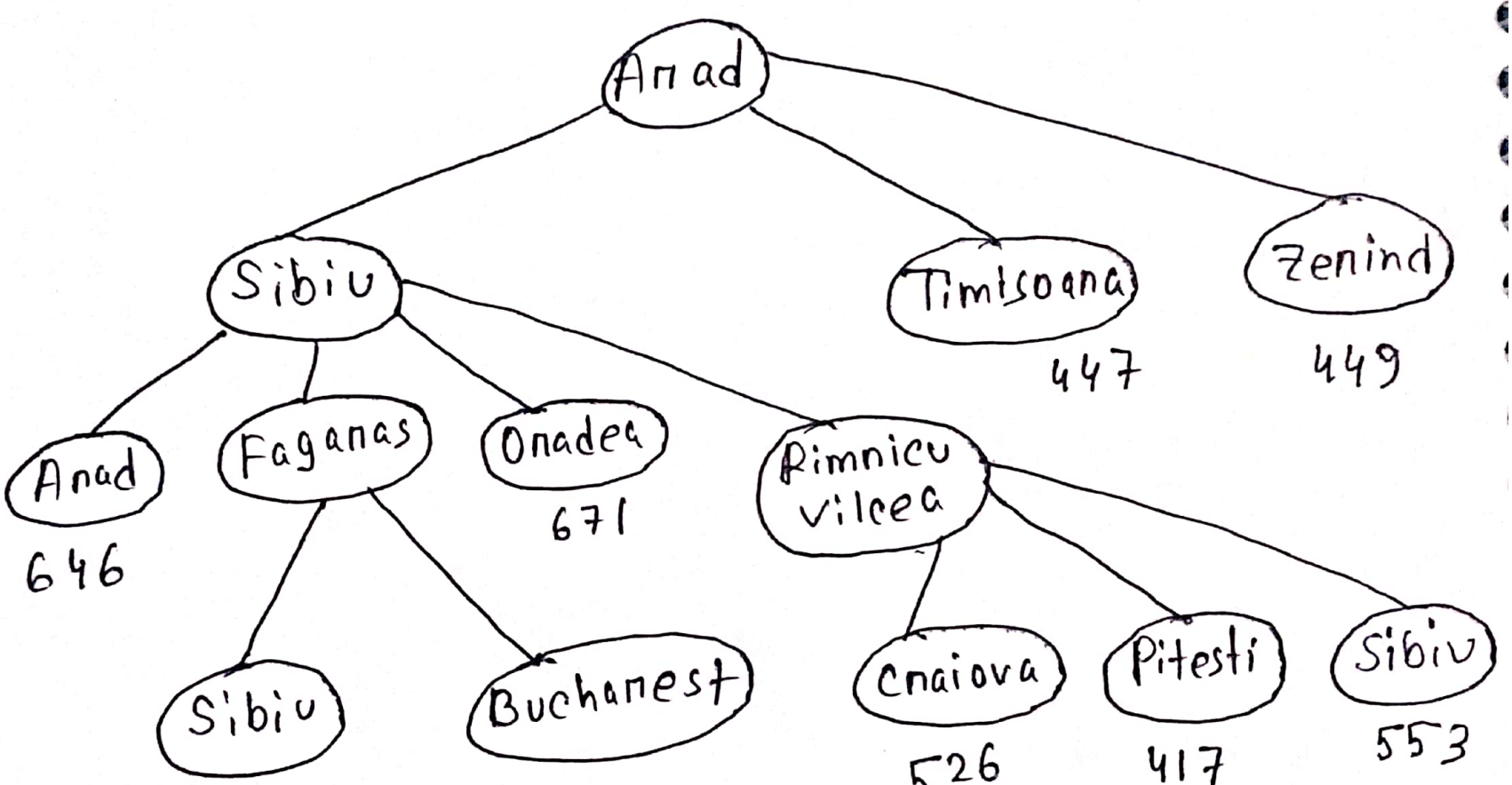
140+151
+380
=671

140+80+193
=413
(min)

Step 4: Expanding Rimnicu Vilcea, we get



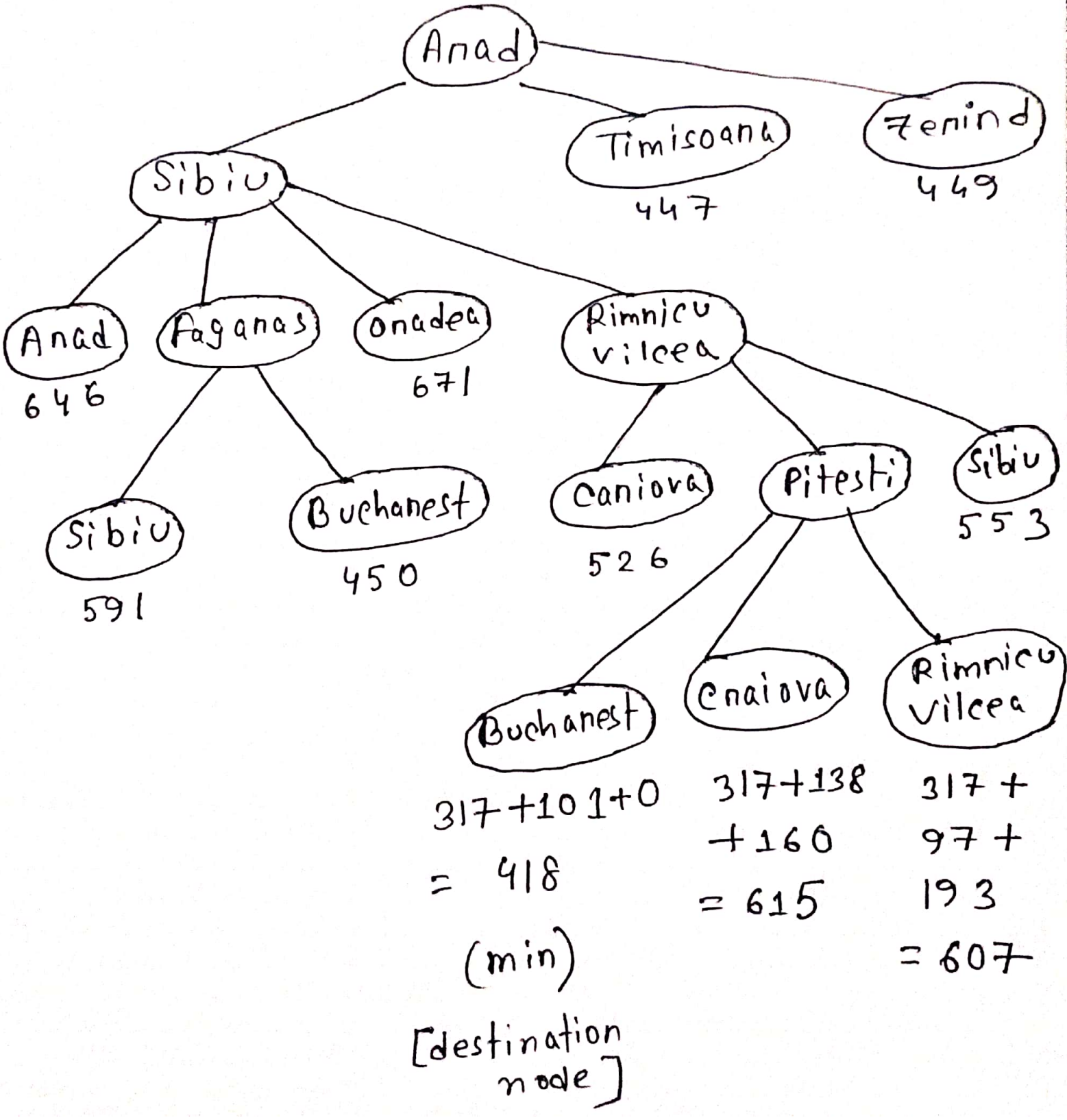
Step 5: Expanding fagaras, we get,



239 + 99
+ 253
= 591

239 + 211
+ 0
= 450

Step 6: Expanding Pitesti, we get.



A* Search Code in C++:

```
#include<bits/stdc++.h>
using namespace std;

const int MAXN = 100005;
const int INF = 1000000007;

int n, m; // n is the number of nodes and m is the number of edges
vector<pair<int,int>> adj[MAXN]; // stores the graph as an adjacency list

int dist[MAXN]; // stores the minimum distance from the start node to each node
int heuristic[MAXN]; // stores the heuristic value for each node

int start_node, end_node; // the start and end nodes for the search

void input_graph() {
   cin >> n >> m;
   int u, v, w;
   for (int i = 0; i < m; i++) {
      cin >> u >> v >> w;
      adj[u].push_back({v, w});
      adj[v].push_back({u, w});
   }
}

void input_start_end() {
   cin >> start_node >> end_node;
}

void dijkstra(int start) {
   priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq; // min-heap
```

```
    for (int i = 1; i <= n; i++) {
        dist[i] = INF;
    }
    dist[start] = 0;
    pq.push({0, start});
    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        for (auto v : adj[u]) {
            int new_dist = dist[u] + v.second;
            if (new_dist < dist[v.first]) {
                dist[v.first] = new_dist;
                pq.push({new_dist, v.first});
            }
        }
    }
}

void a_star(int start, int goal) {
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq; // min-heap
    map<int,int> parent; // stores the parent node for each node in the path
    for (int i = 1; i <= n; i++) {
        dist[i] = INF;
        heuristic[i] = INF;
    }
    dist[start] = 0;
    pq.push({0, start});
    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        if (u == goal) {
            break;
        }
```

```
        for (auto v : adj[u]) {
            int new_dist = dist[u] + v.second;
            if (new_dist < dist[v.first]) {
                dist[v.first] = new_dist;
                heuristic[v.first] = sqrt(pow(v.first - end_node, 2) + pow(v.second - end_node, 2)); // Euclidean distance as heuristic
                parent[v.first] = u;
                pq.push({dist[v.first] + heuristic[v.first], v.first});
            }
        }
    }
    // print the shortest path from start to goal
    vector<int> path;
    int u = goal;
    while (u != start) {
        path.push_back(u);
        u = parent[u];
    }
    path.push_back(start);
    reverse(path.begin(), path.end());
    cout << "Shortest path: ";
    for (auto u : path) {
        cout << u << " ";
    }
    cout << endl;
}

int main() {
    input_graph();
    input_start_end();
    dijkstra(start_node);
    a_star(start_node, end_node);
    return 0;
}
```