Code:

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

#include <cmath>

#include <functional> // Include for 'function'

using namespace std;

// Structure to represent a node in the graph

struct Node {

int index; // Index of the node

int g\_cost; // Cost from start node to this node

int h\_cost; // Heuristic cost (estimated cost from this node to goal node)

int f\_cost; // f(n) = g(n) + h(n)

int parent; // Parent node index

Node(int i, int g, int h, int p) : index(i), g\_cost(g), h\_cost(h), f\_cost(g + h), parent(p) {}

};

// A\* algorithm function

int AStar(vector<vector<int>>& graph, vector<int>& heuristic, int start, int goal, vector<int>& path) {

int n = graph.size();

// Priority queue to store nodes to be explored, ordered by f\_cost

priority\_queue<Node, vector<Node>, function<bool(const Node&, const Node&)>> openList(

[](const Node& a, const Node& b) {

return a.f\_cost > b.f\_cost;

}

);

// Vector to keep track of visited nodes and their parent

vector<bool> visited(n, false);

vector<int> parent(n, -1);

// Add the start node to the open list

openList.push(Node(start, 0, heuristic[start], -1));

while (!openList.empty()) {

// Get the node with the lowest f\_cost from the open list

Node current = openList.top();

openList.pop();

// Check if the current node is the goal

if (current.index == goal) {

// Construct the shortest path

int node = current.index;

while (node != -1) {

path.push\_back(node);

node = parent[node];

}

reverse(path.begin(), path.end()); // Reverse the path to get it from start to goal

return current.g\_cost;

}

// Mark current node as visited

visited[current.index] = true;

// Expand current node

for (int neighbor = 0; neighbor < n; ++neighbor) {

// Check if there is a connection from current node to neighbor and neighbor is not visited

if (graph[current.index][neighbor] != 0 && !visited[neighbor]) {

int g\_cost = current.g\_cost + graph[current.index][neighbor];

int h\_cost = heuristic[neighbor];

int f\_cost = g\_cost + h\_cost;

// Add neighbor to open list

openList.push(Node(neighbor, g\_cost, h\_cost, current.index));

// Update parent information

parent[neighbor] = current.index;

}

}

}

// If goal node is not reachable

return -1;

}

int main() {

int n;

cout << "Enter the number of nodes: ";

cin >> n;

vector<vector<int>> graph(n, vector<int>(n, 0));

cout << "Enter the adjacency matrix:" << endl;

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

cin >> graph[i][j];

}

}

vector<int> heuristic(n, 0);

cout << "Enter the heuristic values for each node:" << endl;

for (int i = 0; i < n; ++i) {

cin >> heuristic[i];

}

int start, goal;

cout << "Enter the start node: ";

cin >> start;

cout << "Enter the goal node: ";

cin >> goal;

vector<int> path;

int shortestPathCost = AStar(graph, heuristic, start, goal, path);

if (shortestPathCost == -1) {

cout << "No path found from node " << start << " to node " << goal << endl;

} else {

cout << "Shortest path cost from node " << start << " to node " << goal << " is: " << shortestPathCost << endl;

cout << "Shortest path: ";

for (int node : path) {

cout << node << " ";

}

cout << endl;

}

return 0;

}

Output:

Enter the number of nodes: 5

Enter the adjacency matrix:

0 1 3 0 0

1 0 0 2 0

3 0 0 0 4

0 2 0 0 5

0 0 4 5 0

Enter the heuristic values for each node:

5 4 3 2 1

Enter the start node: 0

Enter the goal node: 4

Shortest path cost from node 0 to node 4 is: 8

Shortest path: 0 1 3 4