**Name – Armaan Asif Shaikh**

**PRN- 22311412**

**Roll No - 382041**

**AI ASSIGNMENTS**

**Assignment -4**

#include <bits/stdc++.h>

using namespace std;

struct Edge {

string to;

int cost;

};

struct Node {

string name;

int g, f;

bool operator>(const Node& other) const {

return f > other.f;

}

};

class BidirectionalAStar {

unordered\_map<string, vector<Edge>> graph;

unordered\_map<string, int> heuristic;

public:

BidirectionalAStar() {

graph = {

{"Delhi", {{"Noida", 30}, {"Chandigarh", 250}}},

{"Noida", {{"Chandigarh", 220}}},

{"Chandigarh", {{"Ludhiana", 100}, {"Amritsar", 200}}},

{"Ludhiana", {{"Punjab", 80}}},

{"Amritsar", {{"Punjab", 50}}},

{"Punjab", {}}

};

heuristic = {

{"Delhi", 300}, {"Noida", 280}, {"Chandigarh", 150},

{"Ludhiana", 100}, {"Amritsar", 50}, {"Punjab", 0}

};

}

vector<string> findPath(string start, string goal) {

priority\_queue<Node, vector<Node>, greater<Node>> openF, openB;

unordered\_map<string, int> gF, gB;

unordered\_map<string, string> parentF, parentB;

unordered\_set<string> closedF, closedB;

gF[start] = 0;

gB[goal] = 0;

openF.push({start, 0, heuristic[start]});

openB.push({goal, 0, heuristic[goal]});

string meet = "";

while (!openF.empty() && !openB.empty()) {

// Forward step

if (!openF.empty()) {

Node current = openF.top(); openF.pop();

if (closedB.count(current.name)) {

meet = current.name;

break;

}

closedF.insert(current.name);

for (auto& edge : graph[current.name]) {

int tentative = gF[current.name] + edge.cost;

if (!gF.count(edge.to) || tentative < gF[edge.to]) {

gF[edge.to] = tentative;

parentF[edge.to] = current.name;

openF.push({edge.to, tentative, tentative + heuristic[edge.to]});

}

}

}

// Backward step

if (!openB.empty()) {

Node current = openB.top(); openB.pop();

if (closedF.count(current.name)) {

meet = current.name;

break;

}

closedB.insert(current.name);

// Reverse traversal — check all nodes that connect \*to\* current

for (auto& [city, edges] : graph) {

for (auto& edge : edges) {

if (edge.to == current.name) {

int tentative = gB[current.name] + edge.cost;

if (!gB.count(city) || tentative < gB[city]) {

gB[city] = tentative;

parentB[city] = current.name;

openB.push({city, tentative, tentative + heuristic[city]});

}

}

}

}

}

}

if (meet.empty()) {

cout << "No path found.\n";

return {};

}

// Build path from start -> meet and meet -> goal

vector<string> pathF, pathB;

for (string node = meet; !node.empty(); node = parentF[node])

pathF.push\_back(node);

reverse(pathF.begin(), pathF.end());

for (string node = parentB[meet]; !node.empty(); node = parentB[node])

pathB.push\_back(node);

vector<string> finalPath = pathF;

finalPath.insert(finalPath.end(), pathB.begin(), pathB.end());

int totalCost = gF[meet] + gB[meet];

cout << "Total cost = " << totalCost << " km\n";

return finalPath;

}

};

int main() {

BidirectionalAStar solver;

string start = "Delhi", goal = "Punjab";

vector<string> path = solver.findPath(start, goal);

if (!path.empty()) {

cout << "Optimal Path: ";

for (int i = 0; i < path.size(); i++) {

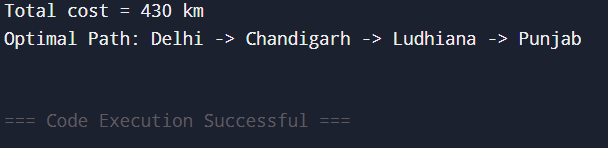
cout << path[i];

if (i != path.size() - 1) cout << " -> ";

}

cout << endl;

}

}****