#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

#include <map>

#include <set>

#include <queue>

#include <algorithm>

#include <chrono>

using namespace std;

struct Point {

int x, y;

};

struct Node {

Point pos;

int g, h, f;

Node\* parent;

};

// Euclidean distance

int heuristic(Point a, Point b) {

return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));

}

bool isValid(Point p, vector<vector<char>>&grid, int nrows, int ncols) {

if (p.x < 0 || p.x >= nrows || p.y < 0 || p.y >= ncols) {

return false;

}

if (grid[p.x][p.y] == 1) {

return false;

}

return true;

}

vector<Point> getNeighbors(Point p,vector<vector<char>>&grid, int nrows, int ncols) {

vector<Point> neighbors;

int dx[] = {-1, 0, 1, 0};

int dy[] = {0, 1, 0, -1};

for (int i = 0; i < 4; i++) {

Point neighbor = {p.x + dx[i], p.y + dy[i]};

if (isValid(neighbor, grid, nrows, ncols)) {

neighbors.push\_back(neighbor);

}

}

return neighbors;

}

struct CompareNodes {

bool operator()(Node\* a, Node\* b) const {

return a->f > b->f;

}

};

struct ComparePoints {

bool operator()(const Point& a, const Point& b) const {

if (a.x < b.x)

return true;

if (a.x == b.x && a.y < b.y)

return true;

return false;

}

};

vector<Point> aStar(vector<vector<char>>&grid, int nrows, int ncols, Point start, Point End,bool includeParent,int a) {

// Create the open and closed sets

priority\_queue<Node\*, vector<Node\*>, CompareNodes> openSet;

set<Point,ComparePoints> closedSet;

// Create the start node

Node\* startNode = new Node{start, 0, heuristic(start, End), a\*heuristic(start, End), NULL};

openSet.push(startNode);

// Loop until the open set is empty

while (!openSet.empty()) {

// Get the node with the lowest f value from the open set

Node\* current = openSet.top();

//cout<<current->pos.x<<" "<<current->pos.y<<endl;

if (grid[current->pos.x][current->pos.y]!='S'&&grid[current->pos.x][current->pos.y]!='E')

grid[current->pos.x][current->pos.y]='\*';

openSet.pop();

// If we have reached the end, construct the path and return it

if (current->pos.x == End.x && current->pos.y == End.y) {

vector<Point> path;

while (current != nullptr) {

path.push\_back(current->pos);

if (grid[current->pos.x][current->pos.y]!='S'&&grid[current->pos.x][current->pos.y]!='E')

grid[current->pos.x][current->pos.y]='P';

current = current->parent;

}

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

return path;

}

// Add the current node to the closed set

closedSet.insert(current->pos);

// Get the neighbors of the current node

vector<Point> neighbors = getNeighbors(current->pos, grid, nrows, ncols);

// Loop through the neighbors

for (auto neighbor : neighbors) {

// If the neighbor is already in the closed set, skip it

if (closedSet.find(neighbor) != closedSet.end()) {

continue;

}

// Calculate the tentative g value for the neighbor

int nextG = current->g + 1;

// Check if the neighbor is already in the open set

bool neighborInOpenSet = false;

// If the neighbor is not in the open set, create a new node for it

if (grid[neighbor.x][neighbor.y]!='#') {

Node\* neighborNode = new Node{neighbor, nextG, a\*(heuristic(neighbor, End)+includeParent\*heuristic(current->pos, End)), 0, current};

neighborNode->f = neighborNode->g + neighborNode->h;

openSet.push(neighborNode);

}

}

}

// there is no path from start to end

return vector<Point>();

}

int main() {

ifstream file("F:/Year 4/Semester2/Robotics/A\_star/25x50.txt");

if (!file.is\_open()) {

cout << "Failed to open the file." << endl;

return 0;

}

int height, width;

file >> height >> width;

vector<vector<char>>grid(height,vector<char>(width));

Point start,End;

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

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

file >> grid[i][j];

if (grid[i][j]=='S')start={i,j};

if (grid[i][j]=='E')End={i,j};

}

}

file.close();

auto startTime = chrono::high\_resolution\_clock::now();

vector<Point>path=aStar(grid,height,width,start,End,0,1);

auto endTime = chrono::high\_resolution\_clock::now();

if (path.empty()){

cout<<"NO PATH!"<<endl;

}

else{

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

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

cout << grid[i][j]<<" ";

}

cout<<endl;

}

}

auto duration = chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime);

cout << "Time taken: " << duration.count() << " milliseconds" << std::endl;

return 0;

}