Question:

Program to implement A\* Algorithm

Description:

A\* (pronounced "A-star") is a popular pathfinding and graph traversal algorithm. It's widely used in computer science and artificial intelligence, particularly in applications like route planning, video games, and robotics. A\* is known for its efficiency and accuracy in finding the shortest path between two points.

The key features of A\* are:

1. It uses a best-first search approach
2. It employs a heuristic function to estimate the cost from any point to the goal
3. It combines the actual cost of reaching a node with the estimated cost to the goal

Approach:

The approach used in the A\* algorithm:

1. Maintain two lists: an open list (nodes to be evaluated) and a closed list (nodes already evaluated)
2. Start with the initial node in the open list
3. For each iteration: a. Choose the node with the lowest f(n) = g(n) + h(n) from the open list where g(n) is the actual cost from start to n, and h(n) is the estimated cost from n to goal b. If this node is the goal, the path is found c. Otherwise, generate successor nodes d. For each successor:
   * Calculate its f(n) value
   * If it's not in the open or closed list, add it to the open list
   * If it's already in the open list, update its values if this path is better e. Move the current node to the closed list
4. Repeat until the goal is reached or the open list is empty (no path exists)

Code Implementation:

#include <iostream>

#include <vector>

#include <queue>

#include <cmath>

#include <algorithm>

using namespace std;

// Structure to represent a cell in the grid

struct Cell {

int x, y;

int f, g, h;

Cell\* parent;

Cell(int x, int y) : x(x), y(y), f(0), g(0), h(0), parent(nullptr) {}

};

// Function to calculate the heuristic (Manhattan distance)

int calculateHeuristic(int x1, int y1, int x2, int y2) {

return abs(x1 - x2) + abs(y1 - y2);

}

// A\* algorithm implementation

vector<Cell\*> aStar(vector<vector<int>>& grid, Cell\* start, Cell\* goal) {

int rows = grid.size();

int cols = grid[0].size();

// Define possible movements (up, right, down, left)

vector<pair<int, int>> directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};

// Create open and closed lists

auto compare = [](Cell\* a, Cell\* b) { return a->f > b->f; };

priority\_queue<Cell\*, vector<Cell\*>, decltype(compare)> openList(compare);

vector<vector<bool>> closedList(rows, vector<bool>(cols, false));

// Add the start cell to the open list

start->f = start->g + calculateHeuristic(start->x, start->y, goal->x, goal->y);

openList.push(start);

while (!openList.empty()) {

// Get the cell with the lowest f value

Cell\* current = openList.top();

openList.pop();

// If we've reached the goal, reconstruct and return the path

if (current->x == goal->x && current->y == goal->y) {

vector<Cell\*> path;

while (current != nullptr) {

path.push\_back(current);

current = current->parent;

}

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

return path;

}

// Add the current cell to the closed list

closedList[current->x][current->y] = true;

// Check all adjacent cells

for (const auto& dir : directions) {

int newX = current->x + dir.first;

int newY = current->y + dir.second;

// Check if the new position is within the grid and not an obstacle

if (newX >= 0 && newX < rows && newY >= 0 && newY < cols && grid[newX][newY] == 0 && !closedList[newX][newY]) {

Cell\* successor = new Cell(newX, newY);

successor->g = current->g + 1;

successor->h = calculateHeuristic(newX, newY, goal->x, goal->y);

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

successor->parent = current;

// Add the successor to the open list if it's not already there

if (find\_if(openList.top(), openList.top() + openList.size(),

[successor](Cell\* c) { return c->x == successor->x && c->y == successor->y; })

== openList.top() + openList.size()) {

openList.push(successor);

}

}

}

}

// If we get here, there's no path to the goal

return {};

}

void printGrid(const vector<vector<int>>& grid, const vector<Cell\*>& path) {

vector<vector<char>> display = grid;

for (auto& row : display) {

for (auto& cell : row) {

cell = cell == 1 ? '#' : '.';

}

}

for (const auto& cell : path) {

display[cell->x][cell->y] = 'o';

}

display[path.front()->x][path.front()->y] = 'S';

display[path.back()->x][path.back()->y] = 'G';

for (const auto& row : display) {

for (const auto& cell : row) {

cout << cell << ' ';

}

cout << endl;

}

}

int main() {

// Example usage

vector<vector<int>> grid = {

{0, 0, 0, 0, 0},

{0, 1, 1, 0, 0},

{0, 0, 0, 0, 0},

{0, 1, 1, 1, 0},

{0, 0, 0, 0, 0}

};

Cell\* start = new Cell(0, 0);

Cell\* goal = new Cell(4, 4);

vector<Cell\*> path = aStar(grid, start, goal);

if (!path.empty()) {

cout << "Path found:" << endl;

for (const auto& cell : path) {

cout << "(" << cell->x << ", " << cell->y << ") ";

}

cout << endl << endl;

cout << "Grid visualization:" << endl;

printGrid(grid, path);

} else {

cout << "No path found." << endl;

}

// Clean up memory

for (auto& cell : path) {

delete cell;

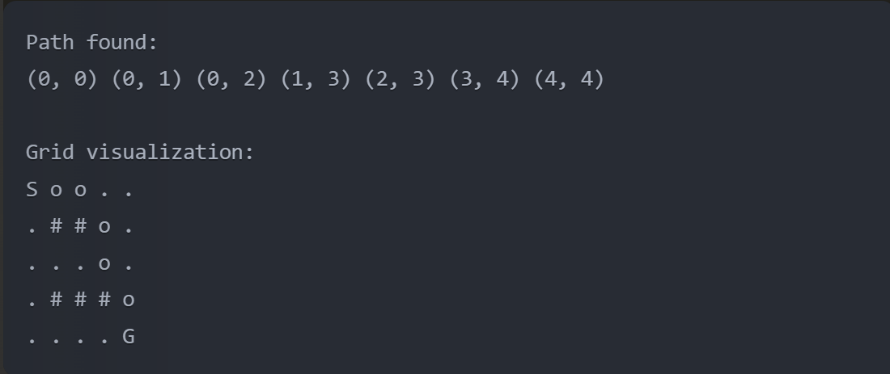
}

return 0;

}

Output:

In this output:

* The first line shows the coordinates of each cell in the path from start to goal.
* The grid visualization shows:
  + 'S' for the start point
  + 'G' for the goal point
  + 'o' for the path taken
  + '#' for obstacles
  + '.' for empty cells