**PRACTICAL NO :2**

**NAME :MANSI KIRLOSKAR**

**ROLL NO:38**

**A\*ALGORITHM**

**CODE:**

**package** Asterist;

**import** java.util.PriorityQueue;

**import** java.util.Stack;

**public** **class** AAsterisk {

//Java Program to implement A\* Search Algorithm

//Here we're creating a shortcut for (int, int) pair

**public** **static** **class** Pair {

**int** first;

**int** second;

**public** Pair(**int** first, **int** second){

**this**.first = first;

**this**.second = second;

}

@Override

**public** **boolean** equals(Object obj) {

**return** obj **instanceof** Pair && **this**.first == ((Pair)obj).first && **this**.second == ((Pair)obj).second;

}

}

// Creating a shortcut for tuple<int, int, int> type

**public** **static** **class** Details {

**double** value;

**int** i;

**int** j;

**public** Details(**double** value, **int** i, **int** j) {

**this**.value = value;

**this**.i = i;

**this**.j = j;

}

}

// a Cell (node) structure

**public** **static** **class** Cell {

**public** Pair parent;

// f = g + h, where h is heuristic

**public** **double** f, g, h;

Cell()

{

parent = **new** Pair(-1, -1);

f = -1;

g = -1;

h = -1;

}

**public** Cell(Pair parent, **double** f, **double** g, **double** h) {

**this**.parent = parent;

**this**.f = f;

**this**.g = g;

**this**.h = h;

}

}

// method to check if our cell (row, col) is valid

**boolean** isValid(**int**[][] grid, **int** rows, **int** cols,

Pair point)

{

**if** (rows > 0 && cols > 0)

**return** (point.first >= 0) && (point.first < rows)

&& (point.second >= 0)

&& (point.second < cols);

**return** false;

}

//is the cell blocked?

**boolean** isUnBlocked(**int**[][] grid, **int** rows, **int** cols,

Pair point)

{

**return** isValid(grid, rows, cols, point)

&& grid[point.first][point.second] == 1;

}

//Method to check if destination cell has been already reached

**boolean** isDestination(Pair position, Pair dest)

{

**return** position == dest || position.equals(dest);

}

// Method to calculate heuristic function

**double** calculateHValue(Pair src, Pair dest)

{

**return** Math.sqrt(Math.pow((src.first - dest.first), 2.0) + Math.pow((src.second - dest.second), 2.0));

}

// Method for tracking the path from source to destination

**void** tracePath(

Cell[][] cellDetails,

**int** cols,

**int** rows,

Pair dest)

{ //A\* Search algorithm path

System.out.println("The Path: ");

Stack<Pair> path = **new** Stack<>();

**int** row = dest.first;

**int** col = dest.second;

Pair nextNode = cellDetails[row][col].parent;

**do** {

path.push(**new** Pair(row, col));

nextNode = cellDetails[row][col].parent;

row = nextNode.first;

col = nextNode.second;

} **while** (cellDetails[row][col].parent != nextNode); // until src

**while** (!path.empty()) {

Pair p = path.peek();

path.pop();

System.out.println("-> (" + p.first + "," + p.second + ") ");

}

}

// A main method, A\* Search algorithm to find the shortest path

**void** aStarSearch(**int**[][] grid,

**int** rows,

**int** cols,

Pair src,

Pair dest)

{

**if** (!isValid(grid, rows, cols, src)) {

System.out.println("Source is invalid...");

**return**;

}

**if** (!isValid(grid, rows, cols, dest)) {

System.out.println("Destination is invalid...");

**return**;

}

**if** (!isUnBlocked(grid, rows, cols, src)

|| !isUnBlocked(grid, rows, cols, dest)) {

System.out.println("Source or destination is blocked...");

**return**;

}

**if** (isDestination(src, dest)) {

System.out.println("We're already (t)here...");

**return**;

}

**boolean**[][] closedList = **new** **boolean**[rows][cols];//our closed list

Cell[][] cellDetails = **new** Cell[rows][cols];

**int** i, j;

// Initialising of the starting cell

i = src.first;

j = src.second;

cellDetails[i][j] = **new** Cell();

cellDetails[i][j].f = 0.0;

cellDetails[i][j].g = 0.0;

cellDetails[i][j].h = 0.0;

cellDetails[i][j].parent = **new** Pair( i, j );

// Creating an open list

PriorityQueue<Details> openList = **new** PriorityQueue<>((o1, o2) -> (**int**) Math.round(o1.value - o2.value));

// Put the starting cell on the open list, set f.startCell = 0

openList.add(**new** Details(0.0, i, j));

**while** (!openList.isEmpty()) {

Details p = openList.peek();

// Add to the closed list

i = p.i; // second element of tuple

j = p.j; // third element of tuple

// Remove from the open list

openList.poll();

closedList[i][j] = true;

// Generating all the 8 neighbors of the cell

**for** (**int** addX = -1; addX <= 1; addX++) {

**for** (**int** addY = -1; addY <= 1; addY++) {

Pair neighbour = **new** Pair(i + addX, j + addY);

**if** (isValid(grid, rows, cols, neighbour)) {

**if**(cellDetails[neighbour.first] == **null**){ cellDetails[neighbour.first] = **new** Cell[cols]; }

**if** (cellDetails[neighbour.first][neighbour.second] == **null**) {

cellDetails[neighbour.first][neighbour.second] = **new** Cell();

}

**if** (isDestination(neighbour, dest)) {

cellDetails[neighbour.first][neighbour.second].parent = **new** Pair ( i, j );

System.out.println("The destination cell is found");

tracePath(cellDetails, rows, cols, dest);

**return**;

}

**else** **if** (!closedList[neighbour.first][neighbour.second]

&& isUnBlocked(grid, rows, cols, neighbour)) {

**double** gNew, hNew, fNew;

gNew = cellDetails[i][j].g + 1.0;

hNew = calculateHValue(neighbour, dest);

fNew = gNew + hNew;

**if** (cellDetails[neighbour.first][neighbour.second].f == -1

|| cellDetails[neighbour.first][neighbour.second].f > fNew) {

openList.add(**new** Details(fNew, neighbour.first, neighbour.second));

// Update the details of this

// cell

cellDetails[neighbour.first][neighbour.second].g = gNew;

//heuristic function cellDetails[neighbour.first][neighbour.second].h = hNew;

cellDetails[neighbour.first][neighbour.second].f = fNew;

cellDetails[neighbour.first][neighbour.second].parent = **new** Pair( i, j );

}

}

}

}

}

}

System.out.println("Failed to find the Destination Cell");

}

// test

**public** **static** **void** main(String[] args) {

//0: The cell is blocked

// 1: The cell is not blocked

**int**[][] grid = {

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

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

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

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

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

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

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

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

};

// Start is the left-most upper-most corner

Pair src = **new** Pair(0,0);

//(8, 0);

// Destination is the right-most bottom-most corner

Pair dest = **new** Pair(6, 6);

AAsterisk app = **new** AAsterisk();

app.aStarSearch(grid, grid.length , grid[0].length, src, dest);

}

}