Maze7x10.txt

|  | **Memory** | **Nodes Explored** | **Path Length** | **Execution Time (ticks)** | **Cost** |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Breadth First** | 4 | 29 | 13 | 21582 | 37 |  |
| **Depth First** | 9 | 20 | 19 | 14532 | 45 |  |
| **Uniform Cost** | 4 | 29 | 15 | 17706 | 39 |  |
| **Iterative Deepening** | 9 | 20 | 19 | 30184 | 45 |  |
| **A\*** | 11 | 20 | 14 | 13101 | 34 |  |

A\* output path:

0001000

0 3 0

0 24 0

00003 0

00 13 0

0 33 0

0 20000

003 0

0 14 0

0001000

Unfortunately I wasn’t able to get my Assignment 1 algorithms working properly.

However, if I compare my findings for A\* with my (inaccurate) findings from Assignment 1, it is clear that this algorithm uses the most memory but the least amount of time, and provides an optimal solution.

public class main {

static ArrayList<ArrayList<Position>> maze = new ArrayList<>();

static ArrayList<ArrayList<Position>> paths;

static ArrayList<Position> solPath;

static HashSet<Position> explored;

static Position goal;

static long startTime;

static long endTime;

static Solution sol = new Solution();

static int memory = 1;

public static void main(String[] args) {

Scanner console = new Scanner(System.in);

// Choose a maze

System.out.println("Please enter the file name:");

String fileName = console.nextLine();

readFromFile(fileName);

// Exectute Algorithm

boolean found = false;

found = UCS();

sol.memory = memory;

sol.nodesExplored = explored.size();

sol.pathLength = solPath.size() + 1;

sol.executionTime = endTime - startTime;

sol.cost = solPath.get(solPath.size()-1).totalCost + goal.cost;

//region Display stats

System.out.println("Memory: " + sol.memory + "\n" +

"Nodes Explored: " + sol.nodesExplored + "\n" +

"Path Length: " + sol.pathLength + "\n" +

"Execution Time: " + sol.executionTime + "\n" +

"Cost: " + sol.cost + "\n");

//endregion

// Display the final path

if (found) {

for (ArrayList<Position> list : maze) {

for (Position pos : list)

if (pos.cost > 0 && !in(pos, solPath) && !(pos.x == goal.x && pos.y == goal.y))

System.out.print(" ");

else

System.out.print(pos.cost);

System.out.println();

}

} else

System.out.println("ALGORITHM FAILED: A path could not be found.");

}

static void readFromFile(String fileName) {

try {

File file = new File(fileName);

Scanner reader = new Scanner(file);

int x = 0;

while (reader.hasNext()) {

String line = reader.nextLine();

int y = 0;

ArrayList<Position> list = new ArrayList<>();

for (char i : line.toCharArray()) {

Position newPos = new Position(x, y++, Integer.parseInt(Character.toString(i)));

list.add(newPos);

}

maze.add(list);

x++;

}

} catch (Exception e) {

e.printStackTrace();

}

}

static Boolean UCS() {

startTime = System.nanoTime();

PriorityQueue<Position> frontier = new PriorityQueue<>();

explored = new HashSet<>();

paths = new ArrayList<>();

//region Find Goal

for (Position pos : maze.get(maze.size() - 1))

if (pos.cost > 0) {

goal = pos;

break;

}

//endregion

findHeuristics();

//region Add initial State to frontier

for (Position pos : maze.get(0))

if (pos.cost > 0) {

pos.totalCost = pos.cost;

frontier.add(pos);

break;

}

//endregion

boolean found = false;

while (frontier.size() > 0) {

// explore a new position

Position curPos = frontier.remove();

// Have we found the goal?

if (curPos.x == goal.x && curPos.y == goal.y) {

found = true;

break;

}

explored.add(curPos);

addToPath(curPos);

//region Find new states for the frontier

ArrayList<Position> tempFront = new ArrayList<>();

if (curPos.y - 1 >= 0) {

Position newPos = maze.get(curPos.x).get(curPos.y - 1);

tempFront.add(newPos);

}

if (curPos.x + 1 < maze.size()) {

Position newPos = maze.get(curPos.x + 1).get(curPos.y);

tempFront.add(newPos);

}

if (curPos.y + 1 < maze.get(0).size()) {

Position newPos = maze.get(curPos.x).get(curPos.y + 1);

tempFront.add(newPos);

}

if (curPos.x - 1 >= 0) {

Position newPos = maze.get(curPos.x - 1).get(curPos.y);

tempFront.add(newPos);

}

//endregion

// Add new states to the frontier

for (Position pos : tempFront)

if (pos.cost > 0 && !in(pos, explored)) {

if (!in(pos, frontier)) {

pos.totalCost = curPos.totalCost + pos.cost;

frontier.add(pos);

} else {

for (Position pos2 : frontier)

if (pos.x == pos2.x && pos.y == pos2.y)

if (pos.totalCost < pos2.totalCost)

pos2.totalCost = pos.totalCost;

}

}

if (frontier.size() > memory)

memory = frontier.size();

}

endTime = System.nanoTime();

// Found the goal

if (found) {

// Find the final path

solPath = new ArrayList<>();

for (ArrayList<Position> list : paths)

if (canReach(goal, list.get(list.size() - 1))) {

solPath = list;

break;

}

return true;

}

// Didn't find the goal

return false;

}

static void findHeuristics() {

for (ArrayList<Position> list : maze)

for (Position pos : list)

pos.heuristic = heuristic(pos, goal);

}

static int heuristic(Position pos, Position goal) {

// Change in x + change in y

return Math.abs(goal.x - pos.x) + Math.abs(goal.y - pos.y);

}

static void addToPath(Position newPos) {

boolean placed = false;

if (paths.size() == 0) {

ArrayList<Position> newList = new ArrayList<>();

newList.add(newPos);

paths.add(newList);

return;

}

// Check the paths and add

int i = 0;

while (!placed) {

ArrayList<Position> list = paths.get(i);

if (canReach(newPos, list.get(list.size() - 1))) {

paths.add((ArrayList<Position>)list.clone());

list.add(newPos);

placed = true;

}

i++;

}

}

static Boolean canReach(Position startPos, Position endPos) {

if (startPos.x == endPos.x) {

if (startPos.y - 1 == endPos.y || startPos.y + 1 == endPos.y)

return true;

} else if (startPos.y == endPos.y) {

if (startPos.x - 1 == endPos.x || startPos.x + 1 == endPos.x)

return true;

}

return false;

}

static Boolean in(Position position, ArrayList<Position> list) {

for (Position pos : list)

if (pos.x == position.x && pos.y == position.y)

return true;

return false;

}

static Boolean in(Position position, PriorityQueue<Position> queue) {

for (Position pos : queue)

if (pos.x == position.x && pos.y == position.y)

return true;

return false;

}

static Boolean in(Position position, HashSet<Position> set) {

for (Position pos : set)

if (pos.x == position.x && pos.y == position.y)

return true;

return false;

}

}

public class Position implements Comparable {

public int x;

public int y;

public int cost;

public int totalCost;

public int heuristic;

public Position(int x, int y) {

this.x = x;

this.y = y;

this.cost = 1;

}

public Position(int x, int y, int cost) {

this.x = x;

this.y = y;

this.cost = cost;

this.totalCost = 0;

this.heuristic = 0;

}

@Override

public int compareTo(Object o) {

return (this.cost + this.heuristic) - (((Position)o).cost + ((Position)o).heuristic);

}

@Override

public String toString() {

return "(" + x + ", " + y + ") " + this.totalCost;

}

}

public class Solution {

public int memory;

public int nodesExplored;

public long executionTime;

public int pathLength;

public int cost;

public Solution() {

}

public Solution(int memory, int nodesExplored, long executionTime, int pathLength, int cost) {

this.memory = memory;

this.nodesExplored = nodesExplored;

this.executionTime = executionTime;

this.pathLength = pathLength;

this.cost = cost;

}

}