SISTEMAS INTELIGENTES

A* LABORATORY EXERCISE

Group C :
Cergan Radu-Mihai
Micu Cristian-Mihai
Plaisanu Alexandru-Ciprian

PROBLEM SUMMARY

In this problem we have a battlefield with an agent which is generated in a initial position and his task is to parse the shortest path to a final position avoiding obstacles (The SittingDuck robots) .

The SittingDucks must be generated in a random position and they occupy 30% of the battlefield. The initial and final positions of the agent are also generated randomly, but the condition is that the tile must be unoccuppied by an obstacle.

The battlefield contains multiple tiles by 64x64 pixels. Those can also be viewed as states for A * algorithm. Each robot is 36x36 pixels in size.

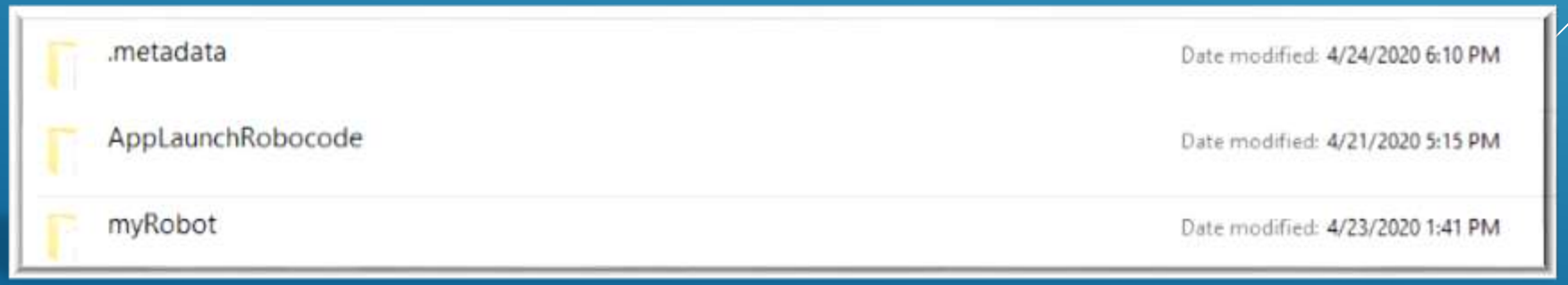
To reach the goal of this problem we use the A* algorithm to return the path the robot has to follow.

The whole process will be launched by a Java application that uses the robocode.control package to set up a battle according to the above specifications, so that we can see your robot moving through the battlefield according to the sequence of actions found by A*.

DESIGN AND IMPLEMENTATION

We have a workspace that is formed from two projects :

- Java application that launches the Robocode through Robocode engine
- The robot project



DESIGN AND IMPLEMENTATION

- JAVA APPLICATION
  - The java application has only one class which includes the main where is created one battlefield and some arrays for storing all types of information about the robots. All of these are necessary to launch the Robocode through robocode engine.
  - In order to create the battlefield(2D array that contains only Tile objects), we have to set the dimensions of the battlefield(length, width) and in our case those are 640x640.
  - For generating the obstacle map, we used a list that contains all integers between 0 and the product of the number of tiles per row and the number of tiles per column. All these integers represents every position on battlefield.
  - We've used that list to avoid generating more obstacles on the same tile.
  - For inserting the robots(SittingDuck) with their specifications in Robocode we have to convert the generated number of tile per row and the number of tile per column into double(pixels).

DESIGN AND IMPLEMENTATION

routeBot.java | RouteFinder.java

```
1  package searchpractice;
2  import java.util.List;
5  public class RouteFinder {
6      public static void main(String[] args) {
7          RobocodeEngine engine = new RobocodeEngine(new java.io.File("C:/robocode"));
8          engine.setVisible(true);
9
10         BattlefieldSpecification battlefield1 = new BattlefieldSpecification(640, 640);
11         int numberOfRounds = 5;
12         long inactivityTime = 10000000;
13         double gunCoolingRate = 1.0;
14         int sentryBorderSize = 50;
15         boolean hideEnemyNames = false;
16         int TileSize = 64;
17         int halfTile = (TileSize / 2);
18         int NumTileRows = (int)(640 / TileSize);
19         int NumTileCols = (int)(640 / TileSize);
20         int VerticalOffset = 640 % 64;
21
22         List<Integer> remainingPossiblePositions = routeBot.generateAllPositions(NumTileRows * NumTileCols);
23         double SittingDuckPercentage = 0.30;
24         int NumObstacles = (int)(NumTileRows * NumTileCols * SittingDuckPercentage);
25         RobotSpecification[] modelRobots = engine.getLocalRepository("sample.SittingDuck, searchpractice.routeBot*");
26         RobotSpecification[] existingRobots = new RobotSpecification[NumObstacles + 1];
27         RobotSetup[] robotSetups = new RobotSetup[NumObstacles + 1];
28         Random randomGenerator = new Random();
29         randomGenerator.setSeed(9);
30         for(int it = 0; it < NumObstacles; ++it) {
31             int idx1 = randomGenerator.nextInt(remainingPossiblePositions.size());
32             int position1 = remainingPossiblePositions.remove(idx1);
33
34             int InitialTileRow1 = (int)(position1 % NumTileRows);
35             int InitialTileCol1 = (int)(position1 / NumTileRows);
36             double InitialObstacleRow = (double)(InitialTileRow1 * TileSize + halfTile);
37             double InitialObstacleCol = (double)(InitialTileCol1 * TileSize + halfTile + VerticalOffset);
38
39             robotSetups[it] = new RobotSetup(InitialObstacleRow, InitialObstacleCol, 0.0);
40             existingRobots[it] = modelRobots[0];
41         }
```

DESIGN AND IMPLEMENTATION

- Also, we need to generate the initial and the final position of the agent. We do that by doing the same thing we did to the obstacle positions(Slide 4 last paragraph).
- Now, all we have to do is to set the rules of the battle(numberOfRounds, inactivityTime, gunCoolingRate, sentryBorderSize, hideEnemyNames) and run it.

routeBot.java | RouteFinder.java

```
37          double InitialObstacleCol = (double)(InitialTileCol1 * TileSize + halfTile + VerticalOffset);
38
39          robotSetups[it] = new RobotSetup(InitialObstacleRow, InitialObstacleCol, 0.0);
40          existingRobots[it] = modelRobots[0];
41      }
42      Random randomGenerator1 = new Random(7);
43      existingRobots[NumObstacles] = modelRobots[1];
44      // Generating the initial position of the agent
45      int idx = randomGenerator1.nextInt(remainingPossiblePositions.size());
46      int position = remainingPossiblePositions.remove(idx);
47
48      int InitialTileRow = position % NumTileRows;
49      int InitialTileCol = position / NumTileRows;
50
51      double InitialAgentRow = InitialTileRow * TileSize + halfTile;
52      double InitialAgentCol = InitialTileCol * TileSize + halfTile + VerticalOffset;
53
54      robotSetups[NumObstacles] = new RobotSetup(InitialAgentRow, InitialAgentCol, 0.0);
55
56      routeBot.startX = InitialTileRow;
57      routeBot.startY = InitialTileCol;
58
59      idx = randomGenerator1.nextInt(remainingPossiblePositions.size());
60      position = remainingPossiblePositions.remove(idx);
61
62      InitialTileRow = position % NumTileRows;
63      InitialTileCol = position / NumTileRows;
64
65      routeBot.endX = InitialTileRow;
66      routeBot.endY = InitialTileCol;
67      BattleSpecification battleSpec = new BattleSpecification(battlefield1, numberOfRounds, inactivityTime, gunCoolingRate, sentryBorderSize, hideEne
68
69      engine.runBattle(battleSpec, true);
70      engine.close();
71      System.exit(0);
72  }
73
74  }
75
```

DESIGN AND IMPLEMENTATION

- ROBOT PROJECT

  - This project contains only one class called routeBot which extends the Robot class making our class to behave as a robot.

  - This class contains a static nested class(Tile) which represents the states for the A* algorithm.

  - One tile contains the estimated cost of the cheapest path from the state at node n to a goal state(heuristic h(n)), the real path cost from the initial state to the state at node n(g(n)) , the evaluation function(f(n)) which is the sum between h(n) and g(n), a boolean variable which checks if a tile was visited, the coordinates of the tile on the battlefield. It also has a previous tile, which is called parent and this helps us to reconstruct the path.

DESIGN AND IMPLEMENTATION

- Our robot also has different methods that help us to reach the goal of the problem. We will present those below :
  - *generateAllPositions(int n)*
    - it is a method that returns a list with all integers between 0 and n
  - *setBlocked(Tile[][] battlefield, int x, int y)*
    - this method is used to block the tile at x, y coordinates when an obstacle was generated on it
  - *ManhattanDistance(Tile tile)*
    - this method helps us to compute the estimated cost of the cheapest path from the state at node n to a goal state(heuristic h(n))
  - *updateCost(Tile currentTile, Tile nextTile)*
    - this method computes the evaluation function of the nextTile which represents the neighbor of currentTile and add it in the openSet

!! See the code of the above functions at the Slide 9 !!

DESIGN AND IMPLEMENTATION

```
182     public static List<Integer> generateAllPositions(int n){
183         List<Integer> list = new ArrayList<Integer>(n);
184         for(int x = 0; x < n; x++){
185             list.add(x);
186         }
187         return list;
188     }
189     public static void setBlocked(Tile[][] battlefield, int x, int y){
190         battlefield[x][y] = null;
191     }
192     public static int MovementCost = 1;
193     public static PriorityQueue<Tile> openSet;
194
195     static int startX, startY;
196     static int endX, endY;
197
198     public static int ManhattanDistance(Tile tile) {
199         return(Math.abs(tile.x - endX) + Math.abs(tile.y - endY));
200     }
201     public static void updateCost(Tile currentTile, Tile nextTile) {
202         if(nextTile == null || nextTile.closed)
203             return;
204         nextTile.h = ManhattanDistance(nextTile);
205         int nextTile_f = nextTile.h + currentTile.g + MovementCost;
206         boolean inOpenSet = openSet.contains(nextTile);
207         if(!inOpenSet || nextTile_f < nextTile.f){
208             nextTile.f = nextTile_f;
209             nextTile.g = currentTile.g + MovementCost;
210             nextTile.parent = currentTile;
211             if(!inOpenSet)
212                 openSet.add(nextTile);
213         }
214
215     }
```

DESIGN AND IMPLEMENTATION

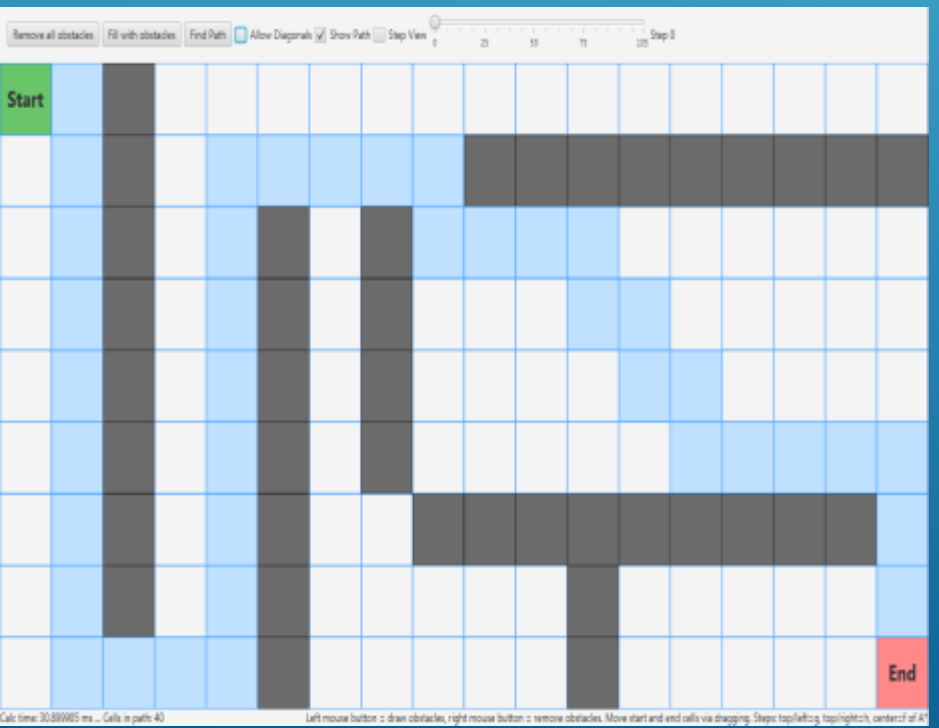
- *Astar(Tile[][] battlefield, int startX, int startY, int endX, int endY)*
  - represents the method that aims to find a path to the final position having the smallest cost starting from a specific initial position
  - for implementing this method, we used the pseudocode attached on the next slide (Slide 11), but we adjusted it in order to find a sequence of actions (moves) by adding a suitable heuristic function
  - to determine the path that the robot has to parse we have to begin from the coordinates of the final position and determine where is located his parent.
    - if the parent is below the current then we have to push **"0"** integer(that means the robot has to go **UP**) in the stack
    - if the parent is on the left side of the current then we have to push **"1"** integer(that means the robot has to go **RIGHT**) in the stack
    - if the parent is above of the current then we have to push **"2"** integer(that means the robot has to go **DOWN**) in the stack
    - if the parent is on the right side of the current then we have to push **"3"** integer(that means the robot has to go **LEFT**) in the stack
  - in the end of the method, if the final position is not reached it will return NULL(the stack is NULL)

For AStar() method check Slide 12

DESIGN AND IMPLEMENTATION

```
closedset := the empty set              // The set of nodes already evaluated
openset := {start}                      // The set of tentative nodes to be evaluated
parent := the empty map                 // The map of navigated nodes
g[start] := 0                           // Cost from start along best known path
f[start] := g[start] + h(start)         // Estimated total cost from start to nearest goal through y
while openset is not empty
        current := the node in openset having the lowest f[] value
        if is_goal(current)
                return reconstruct_path(parent, current)
        remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
                if neighbor in closedset continue
                tentative_g := g[current] + dist_between(current,neighbor)
                if neighbor not in openset or tentative_g < g[neighbor]
                                parent[neighbor] := current
                                g[neighbor] := tentative_g
                                f[neighbor] := g[neighbor] + h(neighbor)
                                if neighbor not in openset
                                                add neighbor to openset
return failure
```

DESIGN AND IMPLEMENTATION




```java
    public static Stack<Integer> AStar(Tile[][] battlefield, int startX, int startY, int endX, int endY) {
        openSet = new PriorityQueue<Tile>((Tile t1, Tile t2) -> { if(t1.f < t2.f) return -1;
        else if (t1.f > t2.f) return 1;
        else return 0;});

        openSet.add(new Tile(startX, startY));
        Tile current;
        while(true) {
            current = openSet.poll();
            if(current == null)
            {
                break;
            }
            current.closed = true;
            if(current.x == endX && current.y == endY) {
                break;
            }
            Tile next;
            if(current.x - 1 >= 0) {
                next = battlefield[current.x - 1][current.y];
                updateCost(current, next);
            }
            if(current.x + 1 < battlefield.length) {
                next = battlefield[current.x + 1][current.y];
                updateCost(current, next);
            }
            if(current.y + 1 < battlefield[0].length) {
                next = battlefield[current.x][current.y + 1];
                updateCost(current, next);
            }
            if(current.y - 1 >= 0){
                next = battlefield[current.x][current.y - 1];
                updateCost(current, next);
            }
        }
        Stack<Integer> path = new Stack<Integer>();
        if(battlefield[endX][endY].closed == true) {
            Tile current1 = battlefield[endX][endY];
            while(current1.parent != null) {
                if(current1.y > current1.parent.y)
                    path.push(0);
                else if(current1.x > current1.parent.x)
                    path.push(1);
                else if(current1.y < current1.parent.y)
                    path.push(2);
                else
                    path.push(3);
                current1 = current1.parent;
            }
            return path;
        }
        else
            return  null;
    }
}
```

DESIGN AND IMPLEMENTATION

- *run()*

  -the main method in every robot, we must override this to set up our robot's behavior.

  -in order to move our robot we have to store the path generated by A* algorithm in a stack

  -we iterate through the stack and interpret the integers between 0 and 3 as robot movement, like we said at the Slide 10

  -at every move of the robot we must test his heading, and if it is not oriented correctly we must turn it with the heading on the next tile of the path

  -in case of an empty stack we will print the message "No path found!" in the robot console

For run() method, check Slide 14 and Slide 15

```
public void run()
{
    int NumTileRows = 10;
    int NumTileCols = 10;
    Tile[][] battlefield = new Tile[NumTileRows][NumTileCols];
    for(int i = 0; i < NumTileRows; ++i)
    {
        for(int j = 0; j < NumTileCols; ++j)
        {
            battlefield[i][j] = new Tile(i, j);
        }
    }

    List<Integer> remainingPossiblePositions = generateAllPositions(NumTileRows * NumTileCols);
    double SittingDuckPercentage = 0.30;
    int NumObstacles = (int)(NumTileRows * NumTileCols * SittingDuckPercentage);

    Random randomGenerator = new Random();
    randomGenerator.setSeed(9);
    int idx1;
    int position1;
    int InitialTileRow1;
    int InitialTileCol1;
    for(int it = 0; it < NumObstacles; ++it) {
        idx1 = randomGenerator.nextInt(remainingPossiblePositions.size());
        position1 = remainingPossiblePositions.remove(idx1);

        InitialTileRow1 = (int)(position1 % NumTileRows);
        InitialTileCol1 = (int)(position1 / NumTileRows);
        setBlocked(battlefield, InitialTileRow1, InitialTileCol1);
    }
    Random randomGenerator1 = new Random(7);
    int idx = randomGenerator1.nextInt(remainingPossiblePositions.size());
    int position = remainingPossiblePositions.remove(idx);

    int InitialTileRow = (int)(position % NumTileRows);
    int InitialTileCol =(int)(position / NumTileRows);
    startX = InitialTileRow;
    startY = InitialTileCol;
    out.println("x : " + InitialTileRow + "y : " + InitialTileCol);
    idx = randomGenerator1.nextInt(remainingPossiblePositions.size());
    position = remainingPossiblePositions.remove(idx);

    int InitialTileRow2 = (int)(position % NumTileRows);
    int InitialTileCol2 = (int)(position / NumTileRows);
    endX = InitialTileRow2;
    endY = InitialTileCol2;
    out.println("x : " + InitialTileRow2 + "y : " + InitialTileCol2);
    Stack<Integer> aux = AStar(battlefield, startX, startY, endX, endY);
    if(!aux.empty())
    {

        while(!aux.empty())
        {
            Integer it = aux.pop();
            if(it == 0)
            {
                if(getHeading() == 90)
                {
                    turnLeft(90);
                    ahead(64);
                }
                else if(getHeading() == 0)
                {
                    ahead(64);
                }
                else if(getHeading() == 180)
                {
                    turnLeft(180);
                    ahead(64);
                }
                else if(getHeading() == 270)
                {
                    turnRight(90);
                    ahead(64);
                }
            }
            else if(it == 1)
```

```
100                 {
101                     if(getHeading() == 90)
102                     {
103                         ahead(64);
104                     }
105                     else if(getHeading() == 0)
106                     {
107                         turnRight(90);
108                         ahead(64);
109                     }
110                     else if(getHeading() == 180)
111                     {
112                         turnLeft(90);
113                         ahead(64);
114                     }
115                     else if(getHeading() == 270)
116                     {
117                         turnRight(180);
118                         ahead(64);
119                     }
120 
121                 }
122                 else if(it == 2)
123                 {
124                     if(getHeading() == 90)
125                     {
126                         turnRight(90);
127                         ahead(64);
128                     }
129                     else if(getHeading() == 0)
130                     {
131                         turnRight(180);
132                         ahead(64);
133                     }
134                     else if(getHeading() == 180)
135                     {
136                         ahead(64);
137                     }
138                     else if(getHeading() == 270)
139                     {
140                         turnLeft(90);
141                         ahead(64);
142                     }
143                 }
144                 else if(it == 3)
145                 {
146 
147                     if(getHeading() == 90)
148                     {
149                         turnRight(180);
150                         ahead(64);
151                     }
152                     else if(getHeading() == 0)
153                     {
154                         turnLeft(90);
155                         ahead(64);
156                     }
157                     else if(getHeading() == 180)
158                     {
159                         turnRight(90);
160                         ahead(64);
161                     }
162                     else if(getHeading() == 270)
163                     {
164                         ahead(64);
165                     }
166                 }
167             }
168         }
169         else
170         {
171             out.println("No path found!");
172 
173         }
174     }
```

CHOICES DURING THE DESIGN AND IMPLEMENTATION PROCESS

- We choose the size of the battlefield to be 640x640 because we avoid getting half-tiles on the map
- To avoid generating 2 obstacles on the same tile we store all the possible positions from the battlefield in a list
- For design, we choose the heading of our robots to be 0.0
- We choose to create the openset of type PriorityQueue for an easier extract of the tile with the smallest evaluation function (f)
- We choose that the function Astar to return a stack because for generating the path that the robot has to follow we have to parse from final position to initial position and if we hadn't used a stack we would have to reverse the array(list, vector, set, etc)
- In order to verify if the obstacle maps coincides between them , we created one battlefield in main for printing the coordinates of all robots from our map

MISTAKES

- One of the mistakes is that we set different seeds for generating the obstacle map on both projects and that made us realize that our robot was hitting obstacles and walls, so we decided to put the same seed for both java application and robot project and after that, it generated the same obstacle map
- When we thought that our project was finished, we ran the project and we saw that the robots are not placed correctly(on the middle of the tile), so we've seen that this problem could be solved by installing a newer version of Robocode
- Another mistake we did was putting all the code in a single project and we received an error, after reading the post on the forum, we created a new workspace that contains two different projects, one project for the java application that launches the Robocode through robocode engine and the other one representing our robot
- We generated the stack containing the path given by A* algorithm outside the run() method, so there was no way to read the stack inside it. We decided that the only way to do this is to generate the stack inside the run() method.
- The last mistake was not setting the seeds for generating the positions of the agent, which caused that the positions were different, so setting the same seed solved our final problem and after that we got drunk ☺

CONCLUSIONS

- We learned robocode which was a new thing for us because we never worked with graphical engines.
- This project was a good introduction in Java for us as well, because we worked only in C, C++ and a little bit of Python. We learned how to manage classes in Java and how to work in Eclipse IDE because we only worked in Visual Studio before
- We improved our abilities of team-working, which made us finish the project faster
- Another conclusion is that every time you encounter a problem, you should search for the problem on the internet, forums and not blocking on it

Here we attached a video of our robot succeeding his task