August 23rd, 2018
A01630086
Luis Eduardo Vargas Victoria
TC2011 Intelligent systems
Assignment 1: Informed search - A*

This program gives the solution for a Maze with obstacles, it shows the result after the movement in a matrix and the movement of what it made that could be RIGHT, LEFT, UP and DOWN
All of this to get this result:

```
Start     W C W W W                                    End  W N W W W
          W P W W G                                         W N W W C
          P P P W P                                         P N N W N
          P W P W P                                         P W N W N
          P W P P P                                         P W N N N
          P P P W W                                         P P P W W
```

Where the program search for the Goal in the array to make the movements
The program is run with an example

How to run the code:

1. Save the code as **`State.java`**
2. Save the code as **`Maze.java`**
3. Go to terminal and type **`ls`**
4. Navigate to where the file is
5. Type **`javac Maze.java`**
6. Type **`java Maze`**
7. Shows the test

```java
public class State {
   public int row;
   public int column;
   public int heuristic;
   public String symbol;

   public State(int row, int column, String symbol) {
      this.row = row;
      this.column = column;
      this.symbol = symbol;
   }

   public int getRow() {
      return row;
   }
   public void setRow(int row) {
      this.row = row;
   }
   public int getColumn() {
      return column;
   }
   public void setColumn(int column) {
      this.column = column;
   }
   public int getHeuristic() {
      return heuristic;
   }
   public void setHeuristic(int heuristic) {
      this.heuristic = heuristic;
   }
   public int heuristics(State start, State end) {
      return Math.abs(this.row - start.row) + Math.abs(this.column -
start.column) + Math.abs(this.row - end.row) + Math.abs(this.column -
end.column);
   }
}

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.TimeUnit;


public class Maze {
   public List<Maze> children = new ArrayList<Maze>();
   public static List<String> movements = new ArrayList<String>();
   public Maze parent;
   public String move;
   public static State[] states = new State[30];
   public int positionX = 0;
   public int positionY = 0;
   public int index = 0;
   public int[] types = new int[4];

   public Maze(State[] pieces, String move) {
      setMaze(pieces);
      movements.add(move);
      this.move = move;
   }

   public boolean goal() {
      if (Maze.states[9].symbol == "C") {
         return true;
      }
      return false;

   }

   public void setMaze(State[] states) {
      for (int i = 0; i < Maze.states.length; i++) {
         Maze.states[i] = states[i];
      }
   }

   public void getMaze(State[]a, State[] b) {
      for (int i = 0; i < b.length; i++) {
         a[i] = b[i];
      }
   }

   public void makeMovements(State start, State end) {
      for (int index = 0; index < Maze.states.length; index++) {
         positionX++;
         positionY++;
         if (Maze.states[index].symbol == "C")
            this.index = index;
            if(positionY < 5) {
               positionY = 0;
            } else if(positionY < 10) {
               positionY = 1;
            } else if(positionY < 15) {
               positionY = 2;
            } else if(positionY < 20) {
               positionY = 3;
            } else if(positionY < 25) {
               positionY = 4;
            } else if(positionY < 30) {
               positionY = 5;
            }

            if(positionX % 5 == 0) {
               positionX = 0;
            } else if(positionX % 5 == 1) {
               positionX = 1;
            } else if(positionX % 5 == 2) {
               positionX = 2;
            } else if(positionX % 5 == 3) {
               positionX = 3;
            } else if(positionX % 5 == 4) {
               positionX = 4;
            }
      }
      for (int i = 0; i < Maze.states.length; i++) {
         if (Maze.states[i].symbol == "C") {
            Maze.states[i].symbol = "N";
         }
      }
      // Left
      if (positionX < 5) {
         if (index - 1 > 0) {
            if (states[index - 1].symbol == "P" || states[index - 1].symbol ==
"G") {
               State[] newPieces = new State[30];
               getMaze(newPieces, states);
               int move = newPieces[index - 1].heuristics(start, end);
               types[0] = move;
            }
         }
      }
      // Right
      if (positionX < 5) {
         if (index + 1 < 30) {
            if (states[index + 1].symbol == "P" || states[index + 1].symbol ==
"G") {
               State[] newPieces = new State[30];
               getMaze(newPieces, states);
               int move = newPieces[index + 1].heuristics(start, end);
               types[1] = move;
            }
         }
      }
      // Up
      if (positionY < 6) {
         if (index - 5 > 0) {
```

```
            if (states[index - 5].symbol == "P" || states[index - 5].symbol ==
"G") {
               State[] newPieces = new State[30];
               getMaze(newPieces, states);
               int move = newPieces[index - 5].heuristics(start, end);
               types[2] = move;
            }
         }
      }
      // Down
      if (positionY < 6) {
         if (index + 5 < 30) {
            if (states[index + 5].symbol == "P" || states[index + 5].symbol ==
"P") {
               State[] newPieces = new State[30];
               getMaze(newPieces, states);
               int move = newPieces[index + 5].heuristics(start, end);
               types[3] = move;
            }
         }
      }
      int heuristic = 0;
      int min = types[0];
      for (int i = 0; i < types.length; i++) {
         System.out.print(i + " ");
         System.out.println(types[i]);
         if (types[i] >= min){
            min = types[i];
            heuristic = i;
         }
      }
      movements(start, end, heuristic, types);

   }

   public void makeLeft(String symbol, int heuristic) {
      if (positionX < 5) {
         if (states[index - 1].symbol != "W") {
            State[] newPieces = new State[30];
            getMaze(newPieces, states);
            newPieces[index - 1].symbol = symbol;
            newPieces[index - 1].heuristic = heuristic;
            //Create child with newPieces
            String movement = "Left";
            Maze child = new Maze(newPieces, movement);
            children.add(child);
            child.parent = this;
            index = index - 1;
         }
      }
   }

   public void makeRight(String symbol, int heuristic) {
      if (positionX < 5) {
         if (states[index + 1].symbol != "W") {
            State[] newPieces = new State[30];
            getMaze(newPieces, states);
            newPieces[index + 1].symbol = symbol;
            newPieces[index + 1].heuristic = heuristic;
            //Create child with newPieces
            String movement = "Right";
            Maze child = new Maze(newPieces, movement);
            children.add(child);
            child.parent = this;
         }
      }
   }

   public void makeUp(String symbol, int heuristic) {
      if (positionY < 6) {
         if (states[index - 5].symbol != "W") {
            State[] newPieces = new State[30];
            getMaze(newPieces, states);
            newPieces[index - 5].symbol = symbol;
            newPieces[index - 5].heuristic = heuristic;
            //Create child with newPieces
            String movement = "Up";
            Maze child = new Maze(newPieces, movement);
            children.add(child);
            child.parent = this;
         }
      }
   }

   public void makeDown(String symbol, int heuristic) {
      if (positionY < 6) {
         if (states[index + 5].symbol != "W") {
            State[] newPieces = new State[30];
            getMaze(newPieces, states);
            newPieces[index + 5].symbol = symbol;
            newPieces[index + 5].heuristic = heuristic;
            //Create child with newPieces
            String movement = "Down";
            Maze child = new Maze(newPieces, movement);
            children.add(child);
            child.parent = this;
         }
      }
   }

   public void movements(State start, State end, int index, int[] heuristic) {
      System.out.println("indice elegido " + index);
      if (heuristic[0] == 0) {
         heuristic[0] = 100;
      }
      if (heuristic[1] == 0) {
         heuristic[1] = 100;
      }
      if (heuristic[2] == 0) {
         heuristic[2] = 100;
      }
      if (heuristic[3] == 0) {
         heuristic[3] = 100;
      }
      if (heuristic[0] < heuristic[1] && heuristic[0] < heuristic[2] &&
heuristic[0] < heuristic[3]) {
         makeLeft("C", heuristic[0]);
      } else if (heuristic[1] < heuristic[0] && heuristic[1] < heuristic[2] &&
heuristic[1] < heuristic[3]) {
         makeRight("C", heuristic[1]);
      } else if (heuristic[2] < heuristic[0] && heuristic[2] < heuristic[1] &&
heuristic[2] < heuristic[3]) {
         makeUp("C", heuristic[2]);
      } else if (heuristic[3] < heuristic[0] && heuristic[3] < heuristic[2] &&
heuristic[3] < heuristic[1]) {
         makeDown("C", heuristic[3]);
      }
   }

   public void printMaze() {
      System.out.println("");
      int m = 0;
      for (int i = 0; i < 6; i++) {
         for (int j = 0; j < 5; j++) {
            System.out.print(states[m].symbol + " ");
            m++;
         }
         System.out.println("");
      }
   }

   public void printMove() {
      System.out.print(move + " ");
   }
```

```
   public boolean samePuzzle(State[] states) {
      boolean samePuzzle = true;
      for (int i = 0; i < states.length; i++) {
         if(Maze.states[i] != states[i]) {
            samePuzzle = false;
         }
      }
      return samePuzzle;
   }

   public static List<Maze> aStarSearch(Maze root, State start, State end){
      List<Maze> path = new ArrayList<Maze>();
      List<Maze> frontier = new ArrayList<Maze>();
      List<Maze> explored = new ArrayList<Maze>();

      frontier.add(root);
      boolean goal = false;
      while (frontier.size() > 0 && !goal) {

         Maze currentMaze = frontier.get(0);
         explored.add(currentMaze);
         frontier.remove(0);

         currentMaze.makeMovements(start, end);
         for (int i = 0; i < currentMaze.children.size(); i++) {
            Maze currentChild = currentMaze.children.get(i);

            if (currentChild.goal()) {
               System.out.println("It has solution");
               goal = true;
               trace(path, currentChild);
            }

            // Checks if the currentChild exists in both if it doesn't add to
frontier
            if(!contains(frontier, currentChild) && !contains(explored,
currentChild)) {
               frontier.add(currentChild);
            } else if (!contains(frontier, currentChild)) {
               frontier.add(currentChild);
               try {
                  currentChild.printMaze();
                  TimeUnit.SECONDS.sleep(1);
               } catch (InterruptedException e) {
                  e.printStackTrace();
               }
            }
         }
      }
      System.out.println("Nodes visited: "+ (frontier.size() + explored.size()));
      return path;
   }

   public static boolean contains(List<Maze> list, Maze maze) {
      boolean contains = false;
      for (int i = 0; i < list.size(); i++) {
         if (list.get(i).samePuzzle(Maze.states)) {
            contains = true;
         }
      }
      return contains;
   }

   public static void trace(List<Maze> path, Maze puzzle) {
      Maze currentPuzzle = puzzle;
      path.add(currentPuzzle);

      while(currentPuzzle.parent != null) {
         currentPuzzle = currentPuzzle.parent;
         path.add(currentPuzzle);
      }
   }


   public static void main(String[] args) {
       State[] piecesInitial = new State[30];
       piecesInitial[0] = new State(0,0,"W");
       piecesInitial[1] = new State(0,1,"C");
       piecesInitial[2] = new State(0,2,"W");
       piecesInitial[3] = new State(0,3,"W");
       piecesInitial[4] = new State(0,4,"W");
       piecesInitial[5] = new State(1,0,"W");
       piecesInitial[6] = new State(1,1,"P");
       piecesInitial[7] = new State(1,2,"W");
       piecesInitial[8] = new State(1,3,"W");
       piecesInitial[9] = new State(1,4,"G");
       piecesInitial[10] = new State(2,0,"P");
       piecesInitial[11] = new State(2,1,"P");
       piecesInitial[12] = new State(2,2,"P");
       piecesInitial[13] = new State(2,3,"W");
       piecesInitial[14] = new State(2,4,"P");
       piecesInitial[15] = new State(3,0,"P");
       piecesInitial[16] = new State(3,1,"W");
       piecesInitial[17] = new State(3,2,"P");
       piecesInitial[18] = new State(3,3,"W");
       piecesInitial[19] = new State(3,4,"P");
       piecesInitial[20] = new State(4,0,"P");
       piecesInitial[21] = new State(4,1,"W");
       piecesInitial[22] = new State(4,2,"P");
       piecesInitial[23] = new State(4,3,"P");
       piecesInitial[24] = new State(4,4,"P");
       piecesInitial[25] = new State(5,0,"P");
       piecesInitial[26] = new State(5,1,"P");
       piecesInitial[27] = new State(5,2,"P");
       piecesInitial[28] = new State(5,3,"W");
       piecesInitial[29] = new State(5,4,"W");
       Maze initPuzzle = new Maze(piecesInitial, "root");
      long startTime = System.nanoTime();
      List<Maze> solution = aStarSearch(initPuzzle, piecesInitial[1],
piecesInitial[9]);
      Collections.reverse(solution);
      long endTime = System.nanoTime();
      double seconds = (endTime - startTime) / 1000000000.0;
      System.out.println("Cost of the path: "+ (solution.size()));
      System.out.println("Used memory: " + (72 * (solution.size())) + " bytes");
      System.out.println("Running time: "+ seconds + " s");
      if (solution.size() > 0) {

         System.out.print("Path to goal: [");
         for (int i = 0; i < solution.size(); i++) {
            solution.get(i).printMove();
         }
         System.out.print("]");
      } else {
         System.out.println("No solution");
      }
   }
}
```

Test

```
0 0
1 0
2 0
3 4
indice elegido 3

W N W W W
W C W W G
P P P W P
P W P W P
P W P P P
P P P W W
0 0
1 0
2 0
3 6
indice elegido 3

W N W W W
W N W W G
P C P W P
P W P W P
P W P P P
P P P W W
0 8
1 6
2 0
3 0
indice elegido 0

W N W W W
W N W W G
P N C W P
P W P W P
P W P P P
P P P W W
0 0
1 0
2 0
3 8
indice elegido 3

W N W W W
W N W W G
P N N W P
P W C W P
P W P P P
P P P W W
0 0
1 0
2 0
3 10
indice elegido 3

W N W W W
W N W W G
P N N W P
P W N W P
P W C P P
P P P W W
0 0
1 10
2 0
3 12
indice elegido 3

W N W W W
W N W W G
P N N W P
P W N W P
P W N C P
P P P W W
0 0
1 10
2 0
3 0
```

```
indice elegido 1

W N W W W
W N W W G
P N N W P
P W N W P
P W N N C
P P P W W
0 0
1 14
2 8
3 0
indice elegido 1

W N W W W
W N W W G
P N N W P
P W N W C
P W N N N
P P P W W
0 0
1 12
2 6
3 0
indice elegido 1

W N W W W
W N W W G
P N N W C
P W N W N
P W N N N
P P P W W
0 0
1 10
2 4
3 0
indice elegido 1
It has solution

W N W W W
W N W W C
P N N W N
P W N W N
P W N N N
P P P W W
Nodes visited: 11
Cost of the path: 11
Used memory: 792 bytes
Running time: 10.048230816 s
Path to goal: [root Down Down Right Down Down
Right Right Up Up Up ]
```