Muhammed Salih DEDE

042101123

Introduction

This report handles the solution for 8 puzzle games based on the A* algorithm. We have some base code for preparing our solution. Which are Board.java, Eight Puzzle.java and Tail.java. In this paper first, we try to understand given base codes. After that, we discuss updating our base codes for our implementation. Finally, we add some new classes methods and solutions for our problem. We will talk about some gaps and sum up the paper.

Base Codes and Updates

Board Class

This class are responsible for setting the frame of the puzzle, and creating our randomly generated puzzle configuration in the 1D array format. Create tails between puzzle slices. Contains movement functions that give a chance to move cells. In our solution, we use Manhattan distance which is the heuristic for calculating all puzzle cell distances between initial and goal state. In this perspective, we need a 2D array instead of 1D. When we try to calculate distance 2D array will be a good choice because the 2D array same as our game board which has x and y coordinates.

The added method follows for converting a 1D array to a 2D array.

```
//----------------ADDED PART--------------------------
//methode for convert array 1D --> 2D
int [][] twoDimArray;
public int[][] twoDimArrayConverter (int [] oneDimArray){
   int x = 0;
   for(int i = 0;i<3;i++)
   {
      for(int j = 0;j<3;j++)
      {
         twoDimArray[i][j] = oneDimArray[x];
         x+=1;
      }
   }
   return twoDimArray;
};
//-----------------------------------------------------
```

(Updated Part of Board Class)

Tile Class

This class is responsible for creating our puzzle slices on the board. Define the colour, size and font of the tiles. Based on the tail coordinates which are defined in the Board class. Use in the Tile class to find the position of the correct puzzle slice. There isn't any update in this class.

Eight Puzzle Class

This class is our main class. In this class, we set frame, canvas etc. On the other hand, the important part of this class is movements. This class use move functions which are inside the board class. In the base code, the functions are defined by KeyEvent libraries which give the opportunity to use the keyboard to control the empty cell. We add GUI (Graphical User Interface) for the user.

Added start screen for take intension of the user. We add a guideline to guide the user.

```
//create Start screen for GUI
public static void drawStartScreen() {
    StdDraw.clear(Color.WHITE);
    StdDraw.setPenColor(Color.BLACK);
    Font font = new Font(name:"Arial", Font.BOLD, size:20);
    StdDraw.setFont(font);
    StdDraw.text(x:2, y:2.5, text:"START");
    StdDraw.text(x:2, y:2, text:"Press START to begin the puzzle");
    StdDraw.show();
}
```

(Start Screen Implementation)

Added the isMousePressed function to understand the click of the mouse by a user we set the limitation for understanding the click as the right click to start the program.

```
//create mousepressed for understand user click on the board.
public static void waitForStart() {
    while (true) {
        //give limitation for clicked area to understand its click start
        if (StdDraw.isMousePressed()) {
            double x = StdDraw.mouseX();
            double y = StdDraw.mouseY();
            if (x >= 1.5 && x <= 2.5 && y >= 2 && y <= 3) {
                break;
            }
        }
    }
}
```

(Mouse Click Listener Implementation)

We added the solveAndDraw function to solve the initial puzzle and show the movement iteration on the board. In this case, we update the movement conditions we call an ArrayList and use all of the stored strings inside it one by one. While the for loop processing is according to this movement animations process one by one.

```
//based on solution animate the call abimate the puzzle and write the movement on the board func.
public static PuzzleNode solveAndDraw(Board board, int[][] startState, int[][] goalState) {
    PuzzleNode solution = EightPuzzleSolver.solvePuzzle(startState, goalState);
    if (solution != null) {
        List <String> movementList = EightPuzzleSolver.printSolutionPath(solution);
        for (int i = 0; i < movementList.size(); i++) {
            board.draw();
            StdDraw.show();
            StdDraw.pause(t:200);

            if (movementList.get(i).equals(anObject:"R")) {
                board.moveRight();
                printMoveOnBord(move:"Move: Right", x:3.5, y:0.2, board);
            }
            if (movementList.get(i).equals(anObject:"L")) {
                board.moveLeft();
                printMoveOnBord(move:"Move: Left", x:3.5, y:0.2, board);
            }
            if (movementList.get(i).equals(anObject:"U")) {
                board.moveUp();
                printMoveOnBord(move:"Move: Up", x:3.5, y:0.2, board);
            }
            if (movementList.get(i).equals(anObject:"D")) {
                board.moveDown();
                printMoveOnBord(move:"Move: Down", x:3.5, y:0.2, board);
            }
        }
    }
    return solution;
}
```

(Implementation of updated movement functions)

Added Classes

The first added class is CountInversions in this class we calculate the sum of the inversion for all tiles. The calculation of this is used in the decision-handle whether the puzzle is solvable or unsolvable when the user clicks the start.

```java
public class CountInversions {
    public static int countInversions(int[][] arr) {
        int count = 0;
        int n = arr.length;

        for (int i = 0; i < n * n; i++) {
            int row1 = i / n;
            int col1 = i % n;

            if (arr[row1][col1] == 0) // Skip counting inversions for 0
                continue;

            for (int j = i + 1; j < n * n; j++) {
                int row2 = j / n;
                int col2 = j % n;

                if (arr[row2][col2] == 0) // Skip counting inversions for 0
                    continue;

                if (arr[row1][col1] > arr[row2][col2])
                    count++;
            }
        }
        System.out.println(count);
        return count;
    }
}
```

In this class, the method calculates the number of tiles which contain fewer numbers in their cell on the right of us. When calculating the end we take the mode of 2. If it's equal to 0 it means solvable puzzle bu if it’s not board return red and the text appears on the frame “Puzzle Not Solvable”.

The second added class is the Eight Puzzle Solver Class which contains the A star algorithm, our node and data structure.

First, we declare our variables. For storing calculating or defining the needed parts and contains constructure. This contracture contains the calculateHeuristic method.

```java
class PuzzleNode {
    //current state
    int[][] state;
    int cost = 0;
    int heuristic;
    PuzzleNode parent;
    String moves; // to store movement in String format

    public PuzzleNode(int[][] state, int cost, PuzzleNode parent, String moves) {
        this.state = state;
        this.cost = cost;
        this.parent = parent;
        this.moves = moves;
        //calculate heuristic for node
        this.heuristic = calculateHeuristic();
    }
```

The calculate heuristic method is used to calculate the heuristic based on Manhattan distance. For this calculation, we sum up each tile distance between the initial and goal state.

```
    // Calculate menhatten distance for heuristic
    public int calculateHeuristic() {
        int heuristic = 0;
        for (int i = 0; i < state.length; i++) {
            for (int j = 0; j < state[0].length; j++) {
                int value = state[i][j];
                if (value != 0) {
                    //calculate target distance
                    int targetRow = (value - 1) / state[0].length;
                    int targetCol = (value - 1) % state[0].length;
                    //basic calc. for calculate distance
                    heuristic += Math.abs(targetRow - i) + Math.abs(targetCol - j);

                }
            }
        }
        return heuristic;
    }
}
```

(One of the functions used to define priority elements in the queue)

Then the A star algorithm needs one open and one closed list for store information. In this paper, we use a priority queue for an open list and an ArrayList for a closed list.

```
//openSet is using for store the possible configration to select the best one based on hueristic and coast
PriorityQueue<PuzzleNode> openSet = new PriorityQueue<>(Comparator.comparingInt(a -> a.cost + a.heuristic));
//visited for store the configrations which are selected in every initial state for find the solution way.
List<String> visited = new ArrayList<>();
```

(Implementation of data structure.)

The priority element deque from the priority queue and compares between goal states and if it's the same the method returns the current configuration which is the goal state.

```
PuzzleNode current = openSet.poll();
if (Arrays.deepEquals(current.state, goalState)) {  //verify ist it goal state or not
    return current;
}
```

If is not the goal state then the initial state passes through the closed list.

```
            visited.add(Arrays.deepToString(current.state));     //every
current which are selected by f(x) function added to the visited array
```

To update the open list with every possible solution configuration we apply defined movements like go up, go down, go left and go right. While the methods doing this behave based on borderlines and visited nodes. If the movement caused any common state in closed and open lists, this movement passed.

```
// Create possible movement of empty cell
int[] dr = {0, 0, 1, -1};
int[] dc = {1, -1, 0, 0};
String[] moves = {"R", "L", "D", "U"}; // Define movement types and store it
//Create all possible movements of empty cell
for (int i = 0; i < 4; i++) {
    int[][] nextState = makeMove(current.state, dr[i], dc[i]);
    //the next possible state is not not or not already visited add states in to the openSet for handling
    if (nextState != null && !visited.contains(Arrays.deepToString(nextState))) {
        // Yeni düğümü oluştururken geçiş hareketini de ekleyin
        String newMoves = moves[i];
        openSet.add(new PuzzleNode(nextState, current.cost + 1, current, newMoves));
    }

}
```

```
//Try to make movement
public static int[][] makeMove(int[][] state, int dr, int dc) {
    int emptyRow = -1, emptyCol = -1;
    //Define where is empty cell
    for (int i = 0; i < state.length; i++) {
        for (int j = 0; j < state[0].length; j++) {
            if (state[i][j] == 0) {
                emptyRow = i;
                emptyCol = j;
                break;
            }
        }
    }

    int newRow = emptyRow + dr;
    int newCol = emptyCol + dc;

    //Try to understand boarder lines of the board and understand iteration of empty cell with lines
    if (newRow < 0 || newRow >= state.length || newCol < 0 || newCol >= state[0].length) {
        return null;
    }

    //update state
    int[][] newState = new int[state.length][state[0].length];
    for (int i = 0; i < state.length; i++) {
        for (int j = 0; j < state[0].length; j++) {
            newState[i][j] = state[i][j];
        }
    }

    newState[emptyRow][emptyCol] = state[newRow][newCol];
    newState[newRow][newCol] = 0;

    return newState;
}
```

Finally, we create a method to see the path which takes us to the goal position.

```java
//print solution path
public static List<String> printSolutionPath(PuzzleNode solution) {
List<int[][]> path = new ArrayList<>();
List<String> movesList = new ArrayList<>();
while (solution != null) {
    if (!solution.moves.isEmpty()) { // Skip adding empty moves
        path.add(index:0, solution.state);    //store solution states
        movesList.add(index:0, solution.moves); // add movement based on solution moves
        System.out.println("Cost: " + solution.cost + ", Heuristic: " + solution.heuristic );   //for developer to see it's work correctly
    }
    solution = solution.parent;
}
return movesList;
```

We create a path for storing all states in the closed list and moveList for storing all moves while reaching the ordered states in the closed list. The return value of the method is novelist which is the path of movement to reach the goal state from the initial state. In the main method which is the Eight Puzzle, we use this return value in the for loop to control movements automatically.

References:

GfG. (2014, July 2). *What is Priority Queue Introduction to Priority Queue*. GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

Java ArrayList. (2024). W3schools.com. https://www.w3schools.com/java/java_arraylist.asp

risingsudhir. (2015). *GitHub - risingsudhir/8-Puzzle-A-Star-Algorithm: 8/15 puzzle C# implementation using A * (A Star) Algorithm.* GitHub. https://github.com/risingsudhir/8-Puzzle-A-Star-Algorithm?tab=readme-ov-file