17.03.2024

COMP 204

PROGRAMMING STUDIO

A* ALGORITHM

NAME: SEMİH

SURNAME: YILMAZ

ID: 042101116

1. IMPLEMENTATION DETAILS OF THE JAVA PROGRAM

1. AstarPuzzleSolver Class:

The java util package was imported and the target state was specified as a 2-dimensional array. Movements (x and y coordinates) were defined for the movements of the pieces on the board. The movements of the space were determined to match the sequence of movements with the movements of the pieces on the board.

```
import java.util.*;

public class AStarPuzzleSolver {
    private static final int[][] GOAL_STATE = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};
    private static final int[][] MOVES = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    private static final String[] MOVE_NAMES = {"U", "D", "L", "R"};
```

Fig 1. Immutable Arrays

1.1. Node Class:

In order to see these nodes on the Puzzle board, the Node class takes the 2d board array, the empty square position, the movements made, the transition from one step to the next (g(n) increases by 1) and the heuristic (manhattan) value and creates a Node instance. This is used to create nodes

```
static class Node {
    int[][] board;
    int emptyX, emptyY;
    String moves;
    int cost;
    int heuristic;

    Node(int[][] board, int emptyX, int emptyY, String moves, int cost) {
        this.board = board;
        this.emptyX = emptyX;
        this.emptyY = emptyY;
        this.moves = moves;
        this.cost = cost;
        this.heuristic = manhattanDistance(board);
    }
}
```

Fig 2. Node Class Definition

1.2 AstarPuzzleSolver Class

The solvePuzzle method takes a two-dimensional array representing the starting state of a puzzle as

input and returns a string representing the solution path or a message indicating that no solution is found. It begins by checking if the puzzle is solvable using the isSolvable method. If the puzzle is not solvable, it returns a message stating that the puzzle is not solvable due to an odd inversion total. If the puzzle is solvable, the method initializes a priority queue to store nodes, with the priority determined by the sum of the cost and heuristic values of each node. It also creates a map to keep track of visited states. The initial state node is added to the queue. The method then enters a loop where it dequeues nodes from the priority queue until it is empty. For each dequeued node, it retrieves the current state, cost, heuristic, and path. It prints out information about the current state, including the state itself, the cost, and the heuristic. If the current state is the goal state, it prints out information about the solution and returns the solution path. Otherwise, it checks if the current state has been visited with a lower cost, and if not, marks it as visited and adds its neighboring states to the queue if they have not been visited or have a lower cost. If no solution is found after exploring all possible states, the method returns a message indicating that no solution is found.

Fig 3. SolvePuzzle Class

```
public String solvePuzzle(int[][] startState) {
    // Check if the puzzle is solvable
    if (!isSolvable(startState)) {
        return "No moves because this puzzle is not solvable. Inversion total is odd.";
    }

    // Priority queue to store nodes, ordered by cost + heuristic
    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt((Node node) -> node.cost + node.heuristic));

    // Map to keep track of visited states
    Map<String, Integer> visited = new HashMap<>();

    // Add initial state to the queue
    queue.add(new Node(startState, findEmptyX(startState), findEmptyY(startState), "", 0));

    // Iterate until the queue is empty
    while (!queue.isEmpty()) {
        // Retrieve the node with the lowest combined cost and heuristic
        Node currentNode = queue.poll();
        int[][] currentState = currentNode.board;
        int cost = currentNode.cost;
        int heuristic = manhattanDistance(currentState);
        String path = currentNode.moves;

        // Print current state, cost, and heuristic
        System.out.println("Current State:");
        printState(currentState);
        System.out.println("Cost: " + cost);
        System.out.println("Heuristic: " + heuristic);
        System.out.println();
```

```
        // Check if the current state is the goal state
        if (Arrays.deepEquals(currentState, GOAL_STATE)) {
            System.out.println("Solution Found:");
            System.out.println("Cost: " + cost);
            System.out.println("Heuristic: " + heuristic);
            System.out.println("Total Cost (Heuristic + Cost): " + (cost + heuristic));
            return path; // Solution Found
        }

        // Convert current state to string for checking visited states
        String currentStateString = Arrays.deepToString(currentState);
        // Skip if the state has been visited with a lower cost
        if (visited.containsKey(currentStateString) && visited.get(currentStateString) <= cost) {
            continue;
        }

        // Mark current state as visited
        visited.put(currentStateString, cost);

        // Get neighboring states
        List<Node> nextStates = getNeighbors(currentNode);
        // Iterate over neighbors
        for (Node neighbor : nextStates) {
            String neighborString = Arrays.deepToString(neighbor.board);
            // Add neighbors to the queue if not visited or have lower cost
            if (!visited.containsKey(neighborString) || neighbor.cost < visited.get(neighborString)) {
                queue.add(neighbor);
            }
        }
```

1.3 isSolvable Class

The function isSolvable checks whether the initial state of the puzzle is solvable. First, it flattens the 2D initial state into a 1D array. Then, it counts the inversions between pairs of numbers in the flattened array. The empty cell (0) is disregarded in this counting process. It calculates the number of inversions and checks if this count is even. If the number of inversions is even, the puzzle is considered solvable, and it returns true. If the number of inversions is odd, the puzzle is considered unsolvable, and it returns false.

```
private static boolean isSolvable(int[][] initialState) {
    int inversions = 0;
    int[] arr = new int[initialState.length * initialState[0].length];
    int k = 0;

    // Flatten the 2D array
    for (int i = 0; i < initialState.length; i++) {
        for (int j = 0; j < initialState[i].length; j++) {
            arr[k++] = initialState[i][j];
        }
    }

    // Count inversions, ignoring the empty cell (0)
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[i] != 0 && arr[j] != 0 && arr[i] > arr[j]) {
                inversions++;
            }
        }
    }

    return inversions % 2 == 0;
}
```

Fig 4. isSolvable Class

1.4 manhattanDistance Class

This function iterates through each tile's position in the initial state using two loops. It calculates the Manhattan distance for each tile by determining its target position and computing the absolute differences between its current position and the target position in both row and column directions. This distance reflects the complexity of the initial state of the puzzle. It's useful for guiding search algorithms towards an optimal solution path and assessing the solvability of the puzzle.

Fig 5. manhattanDistance Class

```
private static int manhattanDistance(int[][] startState) {
    int distance = 0;
    for (int i = 0; i < startState.length; i++) {
        for (int j = 0; j < startState[i].length; j++) {
            int value = startState[i][j];
            if (value != 0) {
                // Find goal position
                int goalX = (value - 1) / startState.length;
                int goalY = (value - 1) % startState.length;
                // Calculate Manhattan distance and add
                distance += Math.abs(i - goalX) + Math.abs(j - goalY);
            }
        }
    }
    return distance;
```

1.5 getNeighbors Class

This method finds valid neighboring nodes for a given node, which represents the position of the empty square on the board, and returns them as a list. Firstly, we obtain the coordinates of the empty square. Then, we check for each move whether it is valid or not. The move is calculated by adding the current position of the empty square. We use the isValidMove method to check if the move is valid. If the move is valid, we compute the new board after making the move and increment the cost one by one to create a new node. We add this new node to our list of neighbors. Finally, we return the list of neighboring nodes.

```
private static List<Node> getNeighbors(Node node) {
    List<Node> neighbors = new ArrayList<>();
    int emptyX = node.emptyX;
    int emptyY = node.emptyY;

    for (int[] move : MOVES) {
        int newX = emptyX + move[0];
        int newY = emptyY + move[1];
        if (isValidMove(newX, newY, node.board)) {
            int[][] newBoard = performMove(emptyX, emptyY, newX, newY, node.board);
            int cost = node.cost + 1;
            neighbors.add(new Node(newBoard, newX, newY, node.moves + MOVE_NAMES[getMoveIndex(move)], cost));
        }
    }

    return neighbors;
}
```

Fig 6. getNeighbors Class

1.6 isValidMove Class

This function checks whether the given coordinates (x, y) represent a valid move within the board. It verifies if the specified coordinates stay within the boundaries of the board or extend outside of it.

```
private static boolean isValidMove(int x, int y, int[][] board) {
    return x >= 0 && x < board.length && y >= 0 && y < board[0].length;
}
```

Fig 7. isValidClass

1.7 getMoveIndex Class

This function is used to find the index corresponding to a given sequence of moves within a predefined array of moves, which contains possible moves of the game. It

takes a move sequence as input and searches for the index corresponding to this sequence within the predefined array of moves.

```
private static int getMoveIndex(int[] move) {
    for (int i = 0; i < MOVES.length; i++) {
        if (Arrays.equals(MOVES[i], move)) {
            return i;
        }
    }
    return -1;
}
```

Fig 8. getMoveIndex

piece and update the

nother calls this function and results in an updated version of the board.

```
private static int[][] performMove(int x, int y, int newX, int newY, int[][] board) {
    int[][] newBoard = new int[board.length][board[0].length];
    for (int i = 0; i < board.length; i++) {
        System.arraycopy(board[i], 0, newBoard[i], 0, board[0].length);
    }
    int temp = newBoard[x][y];
    newBoard[x][y] = newBoard[newX][newY];
    newBoard[newX][newY] = temp;
    return newBoard;
}
```

Fig 9. performMove

1.8, 1.9 findEmptyX and findEmptyY Classes

These two functions are used to find the position of the empty tile (i.e., the tile with the value 0) in a given board array. The first function finds the row (x-coordinate) position of the empty tile, while the second one finds the column (y-coordinate) position. These functions are crucial for determining the position of the empty tile on the board, which plays a significant role in managing the movements that drive the progression of the game.

```
private static int findEmptyX(int[][] board) {
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            if (board[i][j] == 0) {
                return i;
            }
        }
    }
    return -1;
}
```

```
private static int findEmptyY(int[][] board) {
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            if (board[i][j] == 0) {
                return j;
            }
        }
    }
    return -1;
}
```

Fig 10. findEmptyX and findEmptyY

2. Board Class

Board class is given to us as base code. The change made is to turn the 1d array into a 2d array because the numbers are a single string randomly generated array. The puzzle is 3x3, that is, 2D, so it is necessary. In this piece of code, an array containing the numbers 0 to 8 is first created. These numbers are then randomly shuffled using the randomShuffling method and the shuffled numbers are arranged in a 3x3 2D array called tilestemp. The Shuffle method is given to us as a base method.

```
public Board() {
    // create an array that contains each number from 0 to 8

    int[] numbers = new int[9];
    tiletemp = new int[3][3];
    for (int i = 0; i < 9; i++)
        numbers[i] = i;
    // randomly shuffle the numbers in the array by using the randomShuffling
    // method defined below
    randomShuffling(numbers);

    int x = 0;
    for(int i = 0; i<3;i++)
    {
        for(int j = 0; j<3;j++)
        {
            tiletemp[i][j]= numbers[x];
            x+=1;
        }
    }
```

Fig 11. Board Class

3. EightPuzzle Class

The EightPuzzle class is the main class of the program, and the main method resides within this class. Firstly, an object is created from the AStarPuzzleSolver class. Then, the drawStartButton method is called to draw the start button, and the screen is updated. User input is awaited until the start button is clicked. Upon clicking the start button, an object is created from the Board class to generate a random board, and the initial state of this board is assigned to the startState variable. The initial state is printed to the console. Subsequently, the puzzle is solved using the solvePuzzle method of the AStarPuzzleSolver object, and the solution steps are assigned to the solution variable. Another important section is used to apply the solution steps. In each step, the index `i`, which represents the next move in the solution, is obtained. As long as this index is less than the length of the solution steps, the loop continues. At each iteration of the loop, the character at index `i` in the `solution` is retrieved. This character represents a move made to the right (R), left (L), up (U), or down (D). Then, the board is moved accordingly based on this character.

```
// Apply the move
if (i < solution.length()) {
    char move = solution.charAt(i);
    if (move == 'R') {
        board.moveRight();
    } else if (move == 'L') {
        board.moveLeft();
    } else if (move == 'U') {
        board.moveUp();
    } else if (move == 'D') {
        board.moveDown();
```

```
AStarPuzzleSolver AS = new AStarPuzzleSolver();
// Draw the START button
drawStartButton();
StdDraw.show();


// Wait until the START button is clicked
while (true) {
    if (StdDraw.isMousePressed() && isMouseInStartButton()) {
        break;
    }
}

// Create a random board for the 8 puzzle
Board board = new Board();
int[][] startState = board.tiletemp;

System.out.println("Initial State:");
for (int[] row : startState) {
    System.out.println(Arrays.toString(row));
}

// Use the A* algorithm to solve the puzzle
String solution = AS.solvePuzzle(startState);
System.out.println("Solution Steps: " + solution);
```

Fig 12. Board Class's Important Parts

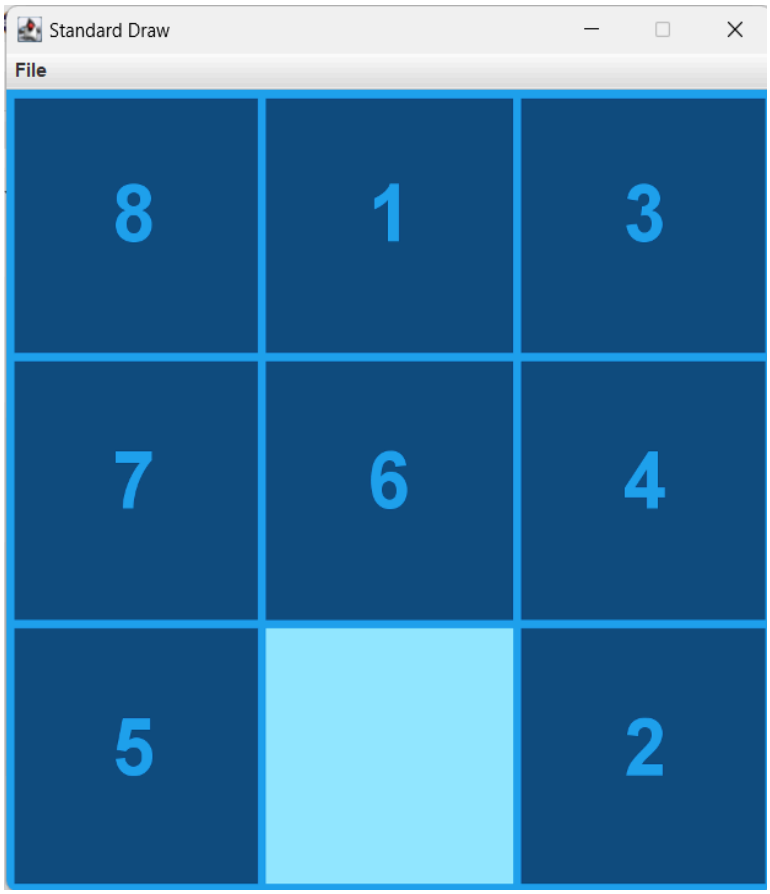
4. TileClass

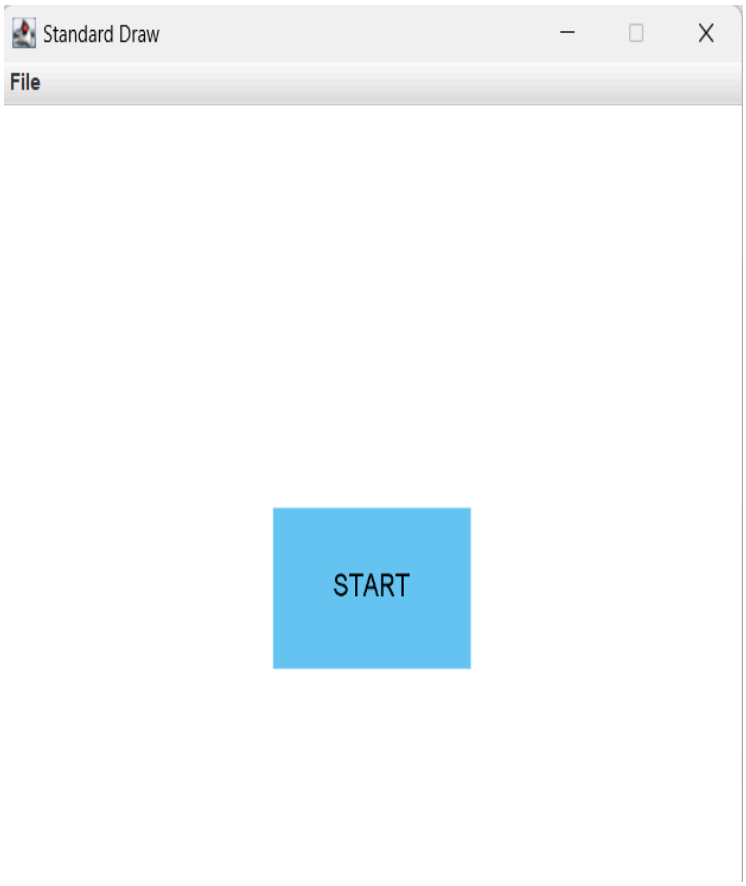
This class is used to represent each square tile of the 8-puzzle. Each tile is distinguished by its number and background color. This class provides a fundamental structure for creating the graphical user interface.

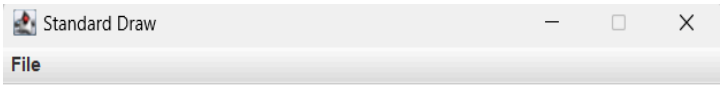
```
public class Tile {
   // The data fields
   // ------------------------------------------------------------------------
   // the color used for the tile (a static constant shared by all the instances)
   private static final Color tileColor = new Color(15, 76, 129);
   // the color used for the number on the tile (a static constant)
   private static final Color numberColor = new Color(31, 160, 239);
   // the color used for the box around the tile (a static constant)
   private static final Color boxColor = new Color(31, 160, 239);
   // the line thickness value for the box around the tile (a static constant)
   private static final double lineThickness = 0.01;
   // the font used for the number on the tile (a static constant)
   private static final Font numberFont = new Font("Arial", Font.BOLD, 50);
   private int number; // the number on the tile (an instance variable)
```

Fig 13. Tile Class

2. OUTPUTS







UNSOLVABLE PUZZLE

PUZZLE SOLVED

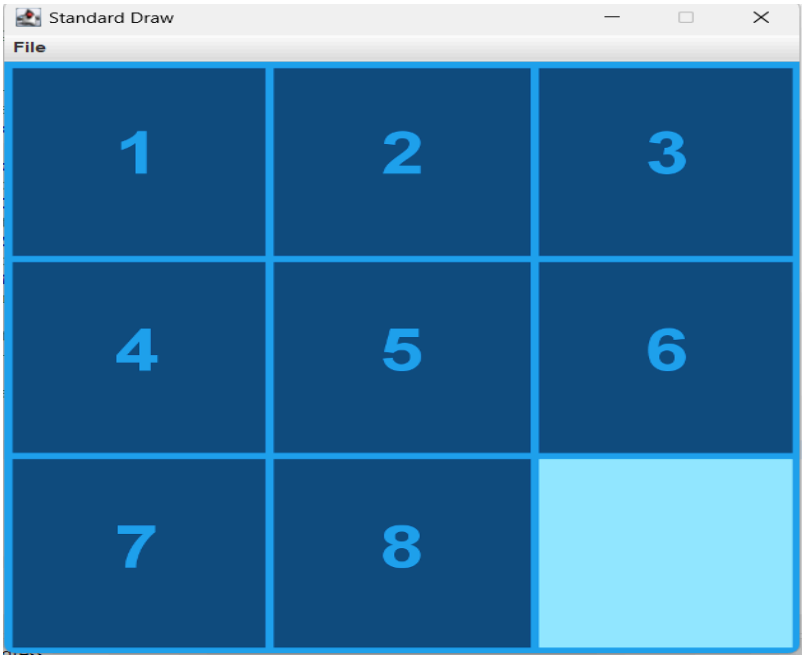


Fig 14. Five Outputs

3. Resources

Computer Science Department at Princeton University. (2019). Princeton.edu. https://www.cs.princeton.edu/

GitHub. (n.d.). *GitHub*. GitHub. https://github.com/

Stack Overflow. (n.d.). *Stack Overflow - Where Developers Learn, Share, & Build Careers*. Stack Overflow. https://stackoverflow.com/

YuChen, D. (2020, July 2). *Looking into k-puzzle Heuristics*. The Startup. https://medium.com/swlh/looking-into-k-puzzle-heuristics-6189318eaca2