UI – Zadanie 1(d)

Hredzhev Oleksandr

ID: 123560

Problem

The **8 Puzzle** is a sliding puzzle which has 8 square tiles numbered 1 to 8 in a frame that is 3 tile positions high and 3 positions wide, with one unoccupied position. Tiles in the same row or column of the open position can be moved by sliding them horizontally or vertically, respectively. The goal of the puzzle is to place the tiles in numerical order (from left to right, top to bottom).

Named after the number of tiles in the frame, the 8 Puzzle may also be called a *"9 Puzzle"*, alluding to its total tile capacity.

The essence of the task is to find a more efficient algorithm using **two heuristics:**

* **The first heuristic(#1)** calculates the effectiveness of the subsequent move based on the number of tiles that are out of place (the fewer tiles that are out of place, the more profitable the combination).

* **The second heuristic(#2)** calculates the efficiency of the subsequent move based on the distance each tile needs to travel to reach its desired position. This heuristic is also called **Manhattan geometry**.

Solution

```
5 usages
Node(int[][] state, int cost, int heuristicValue, Node parent) {
    this.state = state;
    this.cost = cost;
    this.heuristicValue = heuristicValue;
    this.parent = parent;
}
```

General overview each 8-puzzle configuration

```
2 usages
int[][] goalState = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}}; // Target state
```

Target Puzzle Configuration

```
void greedySearch(int[][] initialState) {
    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> node.heuristicValue));
    Set<String> visited = new HashSet<>();

    Node initialNode = new Node(initialState,  cost: 0, heuristic(initialState),  parent: null);
    queue.add(initialNode);
```

This is the greedySearch method that performs the search. It uses a PriorityQueue to select the next state based on a heuristic evaluation. visited is a set that keeps track of visited states. We start with the initial state by creating an initialNode and adding it to the queue.

```
while (!queue.isEmpty()) {
    Node currentNode = queue.poll();

    if (Arrays.deepEquals(currentNode.state, goalState)) {
        // We have reached the target state
        // currentNode contains the path to the target
        printSolution(currentNode);
        return;
    }

    visited.add(Arrays.deepToString(currentNode.state));

    // Generating possible actions
    int[][] currentState = currentNode.state;
```

This is the main search loop. We remove the node from the queue with the lowest heuristic estimate, check whether we have reached the target state and also add the current state to the set of visited states

```
// Let's find an empty cell
int row = -1, col = -1;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (currentState[i][j] == 0) {
            row = i;
            col = j;
            break;
        }
    }
}
```

Loop to find an empty cell

```
// left(if possible)
if (col > 0) { // Checking lateral boundaries
    int[][] newState = cloneState(currentState);
    newState[row][col] = newState[row][col - 1];
    newState[row][col - 1] = 0;
    int hValue = heuristic(newState);
    Node newNode = new Node(newState, cost: currentNode.cost + 1, hValue, currentNode);
    if (!visited.contains(Arrays.deepToString(newNode.state))) {
        queue.add(newNode);
    }
}
```

This function creates a new puzzle configuration by moving the tile (left/right/down/up) if possible and, after heuristic evaluation, adds it to the queue if the given configuration has not already been used

```
    }

    System.out.println("Solution not found");
}
```

If the queue is empty and the target configuration is not found, then the loop ends and displays a message

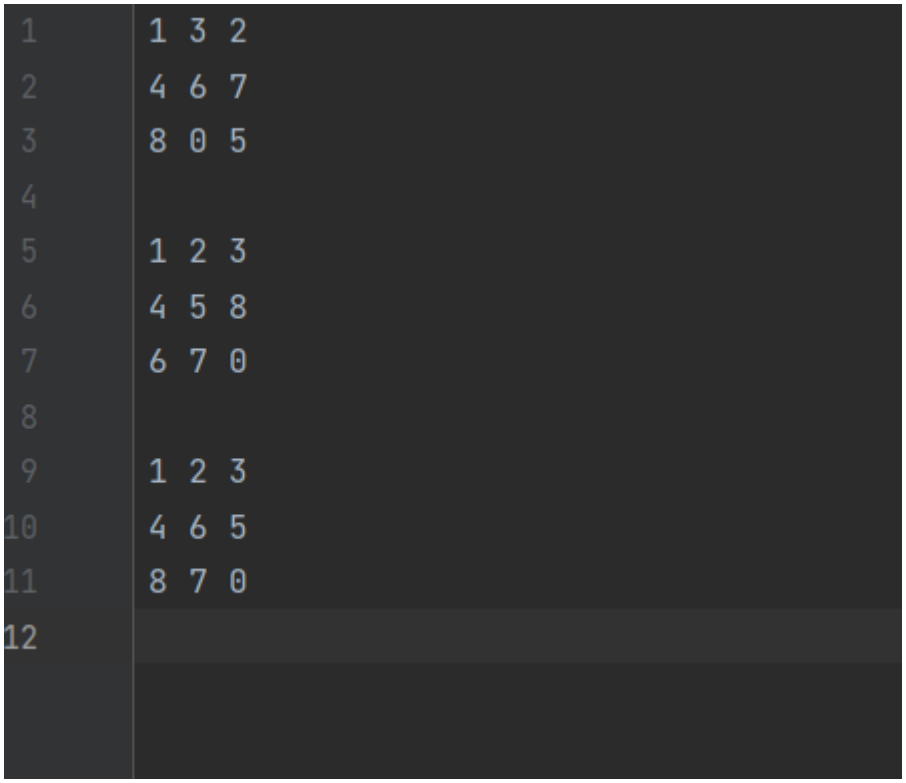
```java
void printSolution(Node node) {
    int count = 0;
    List<Node> path = new ArrayList<>();
    while (node != null) {
        path.add(node);
        node = node.parent;
    }
    Collections.reverse(path);
    for (Node n : path) {
        printState(n.state);
        count++;
    }
    System.out.println("Amount of steps:" + count);
}

// Display
1 usage
void printState(int[][] state) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            System.out.print(state[i][j] + " ");
        }
        System.out.println();
    }
    System.out.println();
}
```

This function restores the inheritance of configurations and displays each of them in the console

Each of the heuristics may behave differently depending on each specific configuration, in particular its depth. Therefore, I will test configurations of different depths for both configurations and provide the number of moves it took to solve each configuration.

Test

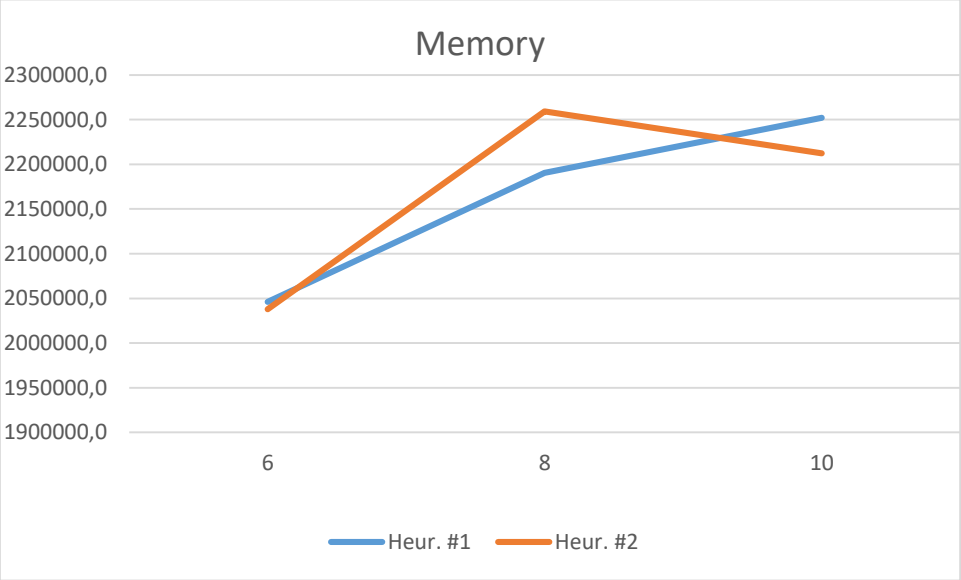
Test cases are in a separate text file. To add your example, enter it exactly as indicated in the screenshot, and leave one empty line after the last one (to avoid errors). To select a heuristic with which configuration data will be resolved after running the code, enter the number 0 or 1 (a message is displayed in the console which number is responsible for each heuristic). To solve the same examples using a different heuristic, simply run the code again and enter a different number.



Results

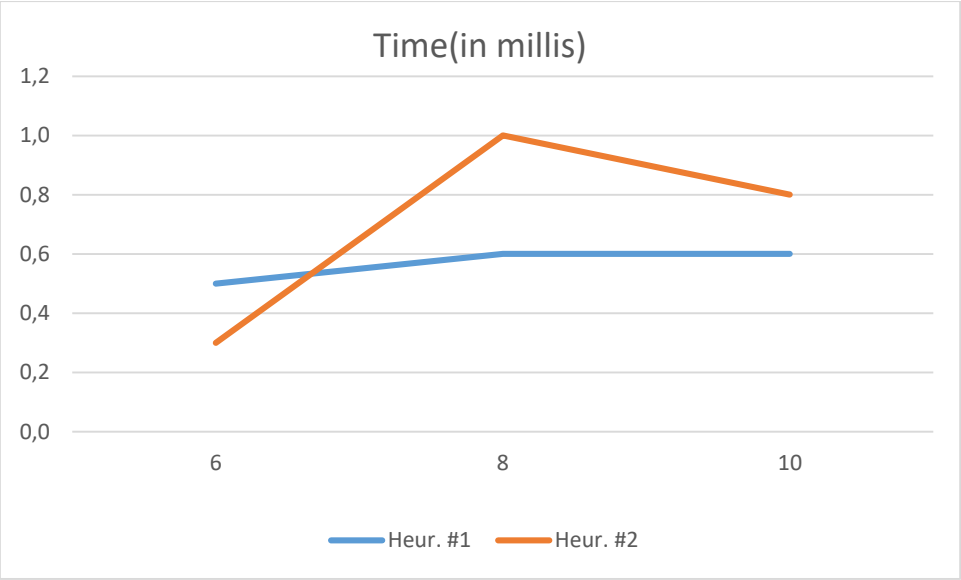
Memory:

Steps	Heur. #1	Heur. #2
6	2046016,8	2038003,5
8	2190657,7	2259197,3
10	2252242,6	2212232,0



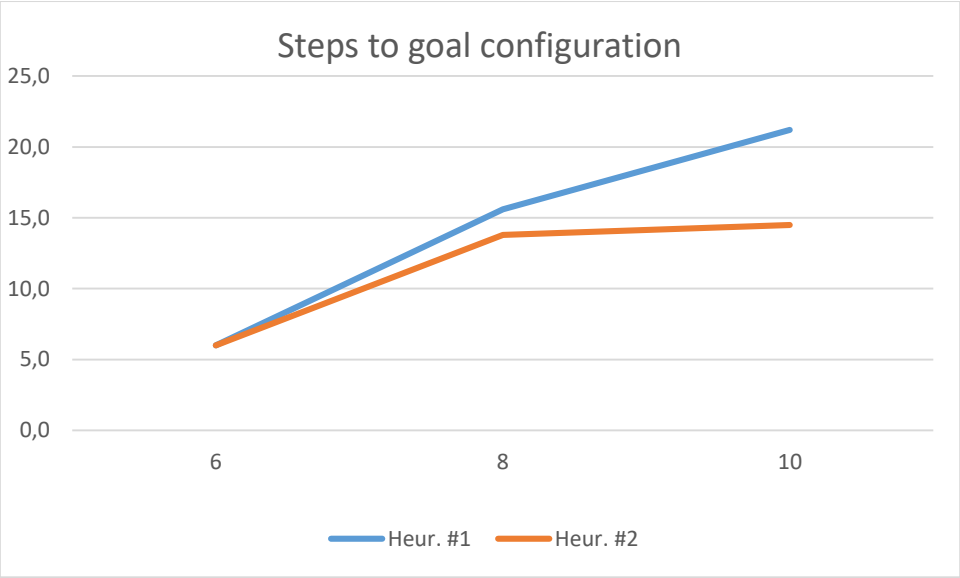
Time(in millis):

Steps	Heur. #1	Heur. #2
6	0,5	0,3
8	0,6	1,0
10	0,6	0,8



Steps to goal configuration:

Steps	Heur. #1	Heur. #2
6	6,0	6,0
8	15,6	13,8
10	21,2	14,5



***Steps** - the number of moves spent creating configurations that the program solves

Conclusion from the received information:

1) The amount of memory spent on the solution in both cases is approximately the same and does not have a clear trend.
2) Looking at the time spent searching for a solution, we can say that heuristic #1 (tiles out of place) is more effective because it is simpler and faster to process.
3) If we consider heuristics from the side of searching for the lowest number of moves, then heuristic #2 (Manhattan distance) will be more effective, since it has a more accurate estimate of the distance (not yes/no, but a deeper analysis)

Possible program improvements

One of the few (but the only one that I know of at the moment) improvements that could calculate the solution path to any given configuration, spending the least number of moves on it, would be the implementation of "Dijkstra's algorithm". If we built it on top of my representation of the Manhattan Path heuristic, it would be possible to go through all the possible solutions to a given puzzle configuration and select the shortest one. But such a solution would most likely take too much time, and in the end could be less effective.