Artificial intelligence

documentation for assignment 1

Contents

Assignment 3
Introduction 4
Functions 5
  Checker 5
  Building graph 7
  Node class 9
Testing 10
Conclusion 10

Assignment

Problem 2 task e.

Task is to find a solution to the 8-puzzle. The puzzle consists of 8 numbered tiles and one empty space arranged in a 3x3 grid. Tiles can be moved up, down, left, or right, but only if there is an empty space in that direction. There is always a given initial position and a target position, and the goal is to find a sequence of moves that will lead from the initial position to the target position

To solve the problem, I need to use the A* algorithm and compare the results of heuristics 1 and 2.

Assignment done in Java language

Introduction

The 8-puzzle is a classic and engaging puzzle game that you've probably encountered at some point in your life. It's a sliding tile puzzle that typically consists of a 3x3 grid with eight numbered tiles and one empty space. The goal is to arrange the tiles in ascending order, from 1 to 8, by sliding them into the empty space. The catch is that you can only move a tile into the empty space if it's adjacent to it, making it a bit of a brain-teaser.

People have been fascinated by the 8-puzzle for years because it's not just about sliding tiles around randomly; it requires strategy and problem-solving skills. It's often used as a fun way to challenge your mind and improve your spatial reasoning. In more advanced versions, like the one you mentioned, there are specific starting and ending configurations, and you need to figure out the best sequence of moves to get from the start to the finish. It's a great exercise in logical thinking and can provide hours of entertainment and mental stimulation.

My task is to write program which will solve 8-puzzle using A* algorithm and then compare to heuristics

Functions

Checker

```
public class checking  //class for checking if puzzle solvable
{
    2 usages
    static int inver(int[] arr) {...}

    1 usage
    static boolean can_solve(int[][] initial, int[][] finalState) {
        int new_p[] = new int[9];
        int new_p2[] = new int[9];
        int k = 0;

        // Converting 2-D puzzles to linear form
        for(int i = 0; i < 3; i++) {
            for(int j = 0; j < 3; j++) {
                new_p[k] = initial[i][j];
                new_p2[k] = finalState[i][j];
                k++;
            }
        }


        int inver1 = inver(new_p);
        int inver2 = inver(new_p2);


        return (inver1 % 2 == inver2 % 2);
    }
}
```

The checking class appears to be used for checking whether a given 8-puzzle is solvable. The solvability of an 8-puzzle is essential to determine if a valid solution exists or not. The class consists of two static methods: inver and can_solve

Inver method calculates the inversion count of an array. In the context of the 8-puzzle, an inversion occurs when a higher-numbered tile appears before a lower-numbered tile in the linear representation of the puzzle. The method takes an array of integers (arr) as input and iterates through pairs of elements to count inversions. It returns the inversion count.

This method checks whether a given initial state of an 8-puzzle and the desired final state are solvable.

It checks if the parity (odd or even) of the inversion counts is the same for both the initial and final states. If they have the same parity, it means the puzzle is solvable because the initial and final states share the same solvability characteristics. If they have different

parities, the puzzle is unsolvable, indicating that there is no valid sequence of moves to transform the initial state into the final state.

So, this puzzle is unsolvable

Example of checking unsolvable puzzle

```
easyPuzzles.add(new test(
        new int[][]{
                {1, 2, 3},
                {4, 5, 6},
                {8, 7, 0}
        },
        new int[][]{
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 0}
        }
));
```

Function

```
for (test testCase : easyPuzzles)
{
    checking check = new checking();
    boolean solve = checking.can_solve(testCase.getInitial(), testCase.getGoal());
    if (solve)
    {...} else
    {
        System.out.println("No solution found.");
    }
}
```

Checking if puzzle solvable

```
easyPuzzles.add(new test(
        new int[][]{
                {1, 3, 0},
                {4, 2, 5},
                {7, 8, 6}
        },
        new int[][]{
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 0}
        }
));
```

```
Easy
Operators: [Left, Down, Right, Down]
```

Building graph

This class is responsible for searching and finding the optimal solution for the puzzle while keeping track of the number of nodes generated during the search.

The primary function of this class is the solving method, which attempts to find a solution to the puzzle. It takes as input the initial puzzle state, the final (goal) puzzle state, and the type of heuristic to use. A heuristic is a way to estimate the distance from the current state to the goal state.

```
public List<node> solving(int[][] start,int[][] final_state, String type_heuristic)
{
    PriorityQueue<node> nodes = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
    Set<String> visited = new HashSet<>();

    node root = new node(start);
    if(Objects.equals(type_heuristic, b: "1"))
    {
    root.cost = factor(heuristic1(start, final_state), root.depth);}
    else if(Objects.equals(type_heuristic, b: "2"))
    {...}
    nodes.add(root);

    while (!nodes.isEmpty()) {
        nodeCount++;
        node current = nodes.poll();
        visited.add(Arrays.deepToString(current.puzzle));

        if (Arrays.deepEquals(current.puzzle, final_state)) {
            return reconstructPath(current);
        }

        List<node> children = children(current);
        for (node child : children) {
            if (!visited.contains(Arrays.deepToString(child.puzzle))) {
                child.cost = factor(heuristic1(child.puzzle, final_state), child.depth);
                nodes.add(child);
            }
        }
    }
```

To manage the puzzle-solving process, the code employs a priority queue named nodes, which ensures that nodes (possible states of the puzzle) are explored in order of their estimated cost. The set visited keeps track of the puzzle states that have already been explored to avoid revisiting them.

Building begins with the creation of the root node. The root node represents the initial puzzle state. Its cost is calculated based on the chosen heuristic type, and it's added to the priority queue.

For each node examined, the code checks if it matches the final state, indicating that a solution has been found. If this is the case, it returns the path from the goal state back to the root. This path represents the sequence of moves required to solve the puzzle.

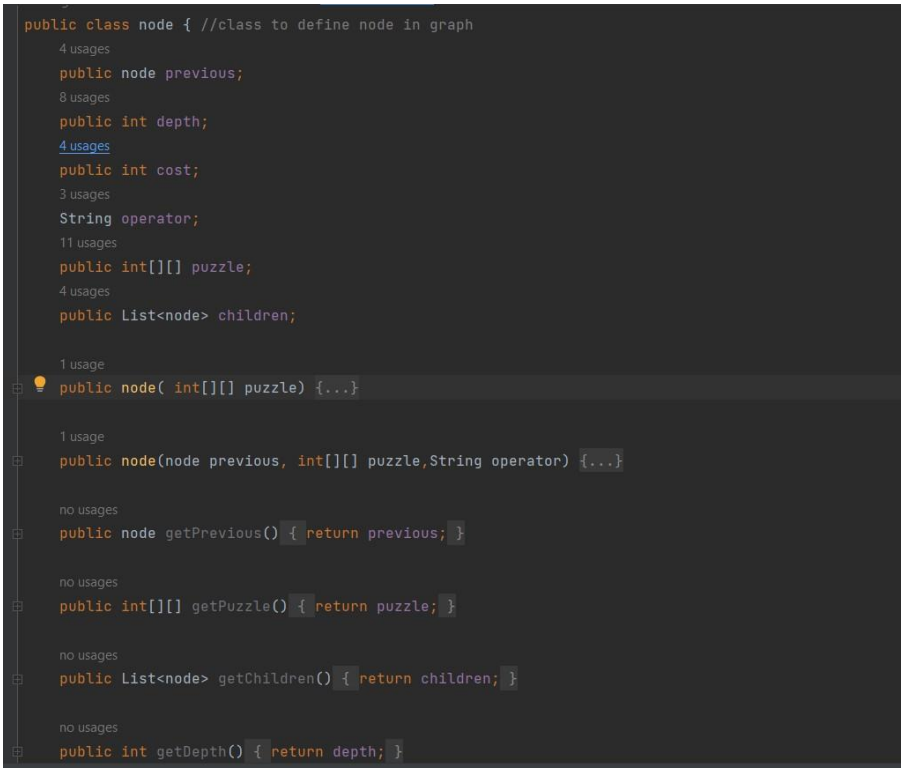
If the current node is not a solution, the code proceeds to generate child nodes. Child nodes are created by making valid moves from the current state. Each move is evaluated for its feasibility, and if it leads to an unvisited state, a child node is generated with the associated move and added to the priority queue .

The method reconstructPath is responsible for reconstructing the path from the goal node back to the root. This path represents the sequence of operators, or moves, used to solve the puzzle. It's worth noting that operators are actions such as "Up," "Down," "Left," and "Right."

Helper methods: finding the position of the zero tile (which is essential for determining valid moves), getting a list of possible moves, and swapping tiles within the puzzle state.

My code employs the A* search algorithm and heuristics to find the optimal solution while keeping track of the number of nodes generated and providing a record of the operators used in the solution path. It's a powerful tool for solving complex puzzles and analyzing the efficiency of puzzle-solving algorithms.

Node class



```
public class node { //class to define node in graph
    4 usages
    public node previous;
    8 usages
    public int depth;
    4 usages
    public int cost;
    3 usages
    String operator;
    11 usages
    public int[][] puzzle;
    4 usages
    public List<node> children;

    1 usage
    public node( int[][] puzzle) {...}

    1 usage
    public node(node previous, int[][] puzzle,String operator) {...}

    no usages
    public node getPrevious() { return previous; }

    no usages
    public int[][] getPuzzle() { return puzzle; }

    no usages
    public List<node> getChildren() { return children; }

    no usages
    public int getDepth() { return depth; }
```

The node class is for building block for the 8-puzzle solving algorithm

Node represents a state within the puzzle-solving process.

Each node has several attributes: previous state, depth of tree, a cost associated with the state, a record of the operator (move) that led to this state, the arrangement of the puzzle board, and a list of its child nodes.

For effective exploration of the puzzle space, each node maintains a list of its child nodes. Child nodes are states that can be reached by performing valid moves from the current state. These children represent the possible next steps in solving the puzzle.

The operator attribute is the move or action that was taken to transition from the previous state to the current one. It's like recording whether the move was "Up," "Down," "Left," or "Right."

The depth attribute provides context by specifying how many steps, or moves, have been taken to reach this state from the initial state.

All these attributes are instrumental in piecing together the puzzle-solving process. By creating and linking these nodes, the algorithm systematically explores the puzzle space, gradually moving closer to the solution.

So, the node class is like the unsung hero behind the scenes, diligently keeping track of puzzle states, their relationships, and the sequence of moves, all in the quest to unlock the mystery of the 8-puzzle. It's an essential character in this digital puzzle-solving adventure.

Testing

Testing my program for solving 8-Puzzle:

To make sure if my code (8-puzzle solver) works correctly I did tests. These tests helps to check if the solver is doing its job and also find out which way of solving puzzles is better, using heuristic1 or heuristic2.

Test Data:

I have different puzzles to test. Some are easy, some are medium, and some are hard.

Each puzzle has a starting state and a final state. The solver's must to figure out how to move from the start to the final state.

Time measurement:

Easy Puzzles:

First, I test the easy puzzles.I do this test 100 times to have correct time of solving.

In each test, I ask the solver to solve the puzzle using either heuristic1 or heuristic2.

I measure the time it takes to solve each puzzle. I measure it in unit - nanoseconds.

After I finish all 100 tests, I add up the time for each test and then find the average time.

Results: I find out how fast the solver is for easy puzzles and which way of solving (using heuristic1 or heuristic2) is better.

Medium Puzzles:

Next, I test puzzles that are a bit more challenging, called medium puzzles. I do 100 tests like before.

Again, I measure how long it takes for the solver to find the solution in nanoseconds.

Results: I see how well the solver does for medium puzzles using both heuristic1 and heuristic2.

Hard Puzzles:

Now, I take on the hardest puzzle.I do 100 tests for these, too.

I measure how long it takes for the solver to find the solution.

Results: I check how well the solver handles the toughest puzzles, and I compare which way of solving is better.

Test Execution:

My program groups puzzles into easy, medium, and hard categories.

The solver, which I call build_graph, is used to find solutions for each puzzle.

For easy ,medium puzzles and hard puzzles I measure time using System.currentTimeMillis().

I collect all the times and average them to get the final results.

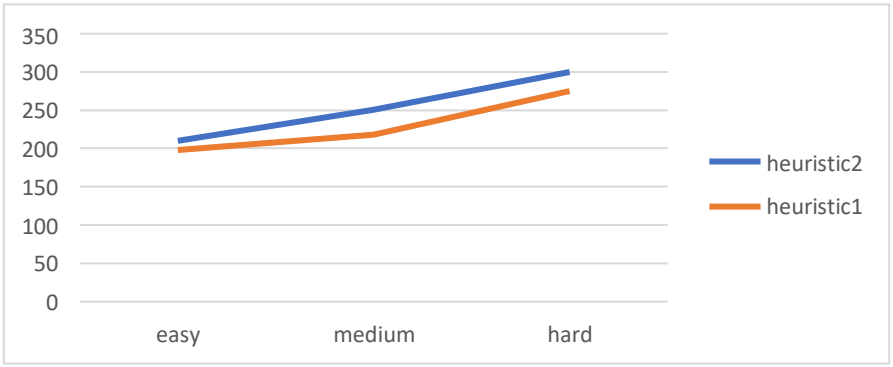
Test Class (test):

```
public class test { //testing class
    2 usages
    private int[][] start;
    2 usages
    private int[][] goal;

    9 usages
    public test(int[][] initial, int[][] goal) {
        this.start = initial;
        this.goal = goal;
    }

    12 usages
    public int[][] getInitial() { return start; }

    12 usages
    public int[][] getGoal() { return goal; }

}
```

I also made a special class called test to help me to manage test data. This class holds the starting setup and final goal for each puzzle, making things easier.

To sum it up, these tests help us make sure the solver works for puzzles of all levels, and they help us figure out which way of solving puzzles (heuristic1 or heuristic2) is faster. We measure time in very tiny units, like nanoseconds and milliseconds, to see how quick the solver is. These results are really important for improving the 8-puzzle solver and making it even better at solving puzzles.

Charts to show difference of solving using two heuristics



level	time	microsec
easy	210	µs
medium	251	µs
hard	300	µs

level	time	microsec
easy	198	µs
medium	218	µs
hard	275	µs

Conclusion

Testing and Evaluation:

In testing, I observed that for easy puzzles, heuristic1 often provided faster solutions.

For medium and hard puzzles, heuristic2 proved to be more efficient, taking into account the sum of distances to determine the next best move.

The choice of heuristic function should be adapted to the puzzle's difficulty level to achieve optimal results.

In conclusion, my program for 8-puzzle solving utilizing the A* algorithm with heuristic1 and heuristic2 represents a significant step in solving this classic problem. My tests have shown that using the right rules for different puzzle difficulties is really helpful. This progress paves the way for future improvements in algorithms and puzzle-solving.