Ostrakov Oleksii

UI task 1

Variant d

My task is 8-puzzle that uses greedy algorithm I did this task in Python programming language. A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

For example, a greedy strategy for the travelling salesman problem (which is of high computational complexity) is the following heuristic: "At each step of the journey, visit the nearest unvisited city." This heuristic does not intend to find the best solution, but it terminates in a reasonable number of steps; finding an optimal solution to such a complex problem typically requires unreasonably many steps. In mathematical optimization, greedy algorithms optimally solve combinatorial problems having the properties of matroids and give constant-factor approximations to optimization problems with the submodular structure.

The evaluation of paths is done by comparing heuristics of possible nodes. There are 2 heuristics that my program uses:

1. **Heuristic 1**) The number of squares that are out of place
2. **Heuristic 2**) The sum of the distances of individual squares from their target position

Implementation

Class **Matrix** consists of list nodes that stores states of nodes. It has numbers of squares and 0 representing empty square. Also **Matrix** has **globalM globalN** and **globalLen**, representing amount of columns, rows and total amount of squares (with empty square) in puzzle. This variables are global, so they do not change after first allocation. Any other class can access it.

Class **globalVar** has functions to complete main algorithms. It stores **startMatrix**, **endMatrix**, **wayArray** that stores final list of nodes from start to target, **usedArray** that stores all occurrences of nodes so there are no duplicates. Function **uniqueMatrix** decides if given matrix is unique or was once used.

Function **swapLeft**/**swapRight** etc. creates a new node that fills empty square and correct square, one func per direction. **PrintSolution** prints all states that were occurred when targeted node. **PrintRoute** prints operators needed to be used to get target node. Function **checkWays** is the main recursive algorithm: firstly it check every possible operators that can be used on current node, sorts them by given heuristic and then recursively calls itself but sends nodes changed by operators.

In main function you can select size of nodes, input start node and target node, choose heuristic and start recursive function **checkWays**. After its completed, it will print all nodes in order from start to target, then prints all operators needed from start to target. It also prints time used on algorithm, amount of created nodes and amount of needed nodes.

Comparing heuristics

Heuristic 1 is very simple condition, but very ineffective. Not only that, its usually even worse than just trying all variants because usually to place node to a far location requires moving close nodes that may already on correct place. As it would increase amount of nodes on wrong places, this path would be chosen as the last, which greatly increases amount of nodes checked. Usually it creates depth more than 1000, which severely affects speed of process, amount of data used and risk of stack overflow.

On the other hand, **Heuristic 2** may be far more effective comparing to first heuristic. In this, far square would have higher value in deciding what square to move, as it may be more effective than placing closest squares to their places. Though it still is not perfect, it provides far better effectiveness then heuristic 1.

Examples

Example 1 : Start Target

3x3

Easy

Start

1	3	4
8	6	2
7		5

Target

1	2	3
8		4
7	6	5

Heuristic 1:

Search was completed in: 0.00044169998727738857 seconds.

Amount of unique nodes: 10

Amount of steps in correct route: 9

Heuristic 2:

Search was completed in: 0.00033809998421929777 seconds.

Amount of unique nodes: 6

Amount of steps in correct route: 5

Example 2: Start Target

3x3

Medium

Start

1	2	3
	7	6
5	4	8

Target

1	2	3
4	5	6
7	8	

Heuristic 1:

Search was completed in: 0.0025465000071562827 seconds.

Amount of unique nodes: 78

Amount of steps in correct route: 77

Heuristic 2:

Search was completed in: 0.00043139999615959823 seconds.

Amount of unique nodes: 8

Amount of steps in correct route: 7

Example 3:

3x3

Hard

Start

2	8	1
	4	3
7	6	5

Target

1	2	3
8		4
7	6	5

Heuristic 1:

RECURSION ERROR (recursion function reached 1000 depth)

Heuristic 2:

Search was completed in: 0.0013299999991431832 seconds.

Amount of unique nodes: 36

Amount of steps in correct route: 35

Example 4: Start Target

3x3

Very hard

2	8	1
4	6	3
	7	5

1	2	3
8		4
7	6	5

Heuristic 1:

RECURSION ERROR (recursion function reached 1000 depth)

Heuristic 2:

Search was completed in: 0.0018124000052921474 seconds.

Amount of unique nodes: 47

Amount of steps in correct route: 46

Example 5:

3x3

Worst-case scenario

Start

5	6	7
4		8
3	2	1

Target

1	2	3
8		4
7	6	5

Heuristic 1:

RECURSION ERROR (recursion function reached 1000 depth)

Heuristic 2:

RECURSION ERROR (recursion function reached 1000 depth)

Example 6:

4x4

Easy

Start

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

Target

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Heuristic 1:

Search was completed in: 0.00016159997903741896 seconds.

Amount of unique nodes: 3

Amount of steps in correct route: 2

Heuristic 2:

Search was completed in: 0.0001846000086516142 seconds.

Amount of unique nodes: 3

Amount of steps in correct route: 2