CmpE 160 - Introduction to Object Oriented Programming

Project 3 - Puzzle Solver

Deadline: 27 May 2019, 23:55

1 Introduction

Note: Before diving into the project description, we would like to explain some concepts about the chess engines; concerning how they work, how they decide for the next move, etc. We would like to emphasize that the project is not about chess engines but the same approach applies when solving the puzzle in this project, as well. If you do not read this part, you will miss nothing but some cool brief information about chess. engines. The project description starts from Section 2: Description

Since you all learned the tree structure in this course and are probably familiar with the decision trees, we may use that knowledge to understand the main idea behind the chess engines. The idea is that we keep all possible moves in a tree, whose nodes has a field representing a table configuration. Here is the algorithm:

- Step 1 - Put initial configuration of the board to the root of your tree.
- Step 2 - Create new children nodes of the root for **each** possible configuration after the next possible move. Note that the tree is not a binary tree and a node can have many children nodes.
- Step 3 - For each leaf node perform the following: Create new children nodes for each possible configurations after the next possible move.
- Step 4 - Go to Step 3

Notice that after Step 3, "leaf nodes" change and every time Step 3 is performed with a different set of leaf nodes. Obviously, there is a problem with the algorithm above. It runs indefinitely and does not give any hint about which move to do. Hence, many engines stop looking further possibilities at some particular depth. Then, the path which yields to reach that ending leaf node with the maximum point up to that depth level is returned. So, what is the "maximum point" here? It's simply a metric of the situation of dominating or humiliating in the game. It can be calculated in different ways, which are up to the programmer. For instance, a very dummy engine may be programmed to think that: "If I have a queen and the opponent does not, then it's a better

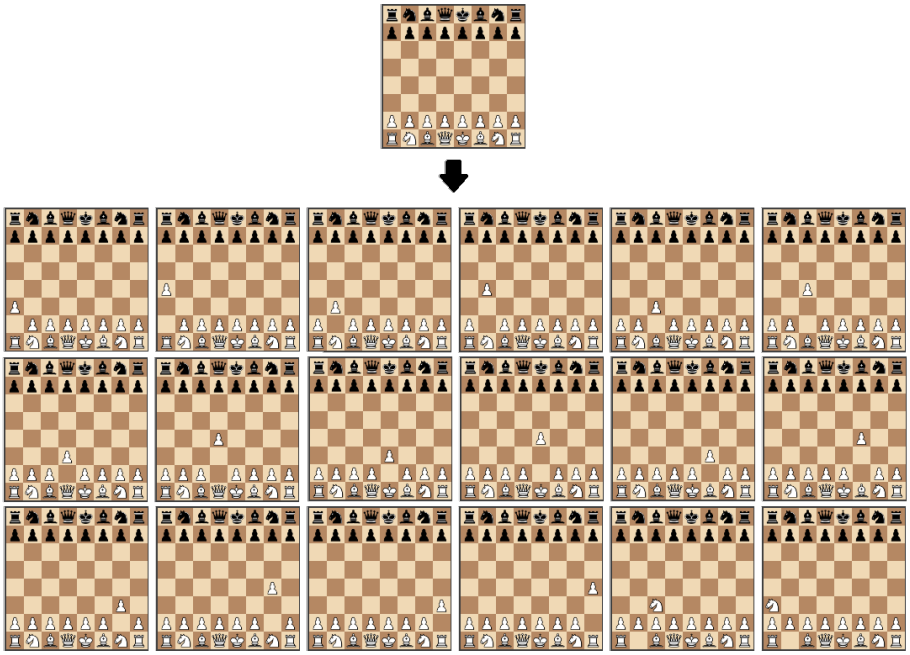

Figure 1: Chess Opening Decision Tree

configuration for me, and hence has +9 points". At the max depth level (limited by the programmer), after all points are calculated, the maximum one is returned with the path caused it to reach there. It simply accomplishes this by backtracking up to the root by using the parent fields of the nodes on the tree. Some of you may think that it's not that hard to create such an engine. Alright, check Shannon Number:

"Shannon showed a calculation for the lower bound of the game-tree complexity of chess, resulting in about 10^{120} possible games, to demonstrate the impracticality of solving chess by brute force, in his 1950 paper "Programming a Computer for Playing Chess". This influential paper introduced the field of computer chess."

Which means that there could be millions of nodes to be dealt with. Those who want to get more information about those engines may search for Gull, Fire, Komodo and Stockfish engines. Also Figure 1 is a visualization of the decision tree at depth 1, except for two missing Knight moves.

2 Description

In this project, the main objective is developing a program that solves the puzzle game, whose rules are as follows:

- Game table is an NxN table whose cells are filled with numbers from 1 to $(N^2 - 1)$, and the remaining one is blank.
- Only the blank box can be moved on the table and the moving direction can be either horizontal or vertical (diagonal moves are not possible).
- When the blank box is moved, the blank one and the box in the destination position of the blank box are swapped.

Here is an example of these rules: Assume that we're given with a particular 3x3 initial configuration illustrated in Figure 2. Then, the three possible moves on this configuration are given in the same figure. Also, the final configuration that your program is supposed to find out the way of reaching that pattern is given in Figure 3.

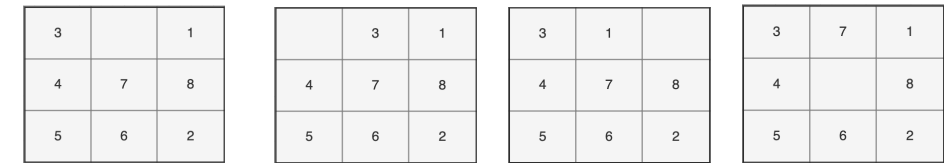

Figure 2: Possible moves from initial configuration

Figure 3: Goal Configuration

Your program should provide the necessary sequential moves in order to achieve the goal configuration from the initial configuration, which is given as input. When solving the problem, you are expected to construct a tree structure. The root of this tree is the initial configuration of the puzzle given as input to your program. Then, for every possible move on the configuration that belongs to a node, you have to create new children nodes and check whether the goal configuration is reached or not. Your program should continue processing until the node with the goal configuration. **But note that, some initial configurations do not have any solution!** You are expected to handle such conditions

for 3x3 case only (i.e. we will not test your program with 4x4, 5x5 or bigger tables which do not have any solution).

See the I/O section for a detailed information about the program responses for unsolvable cases. Also, you should be aware of the repetitions in your tree. If you do not handle such cases and create new nodes for the configurations which have been already generated on the upper levels of the tree, your program probably takes much longer to compute, as there would be millions of nodes.
A basic example with a 3x3 table is given in Figure 4, where the root configuration is given as input:

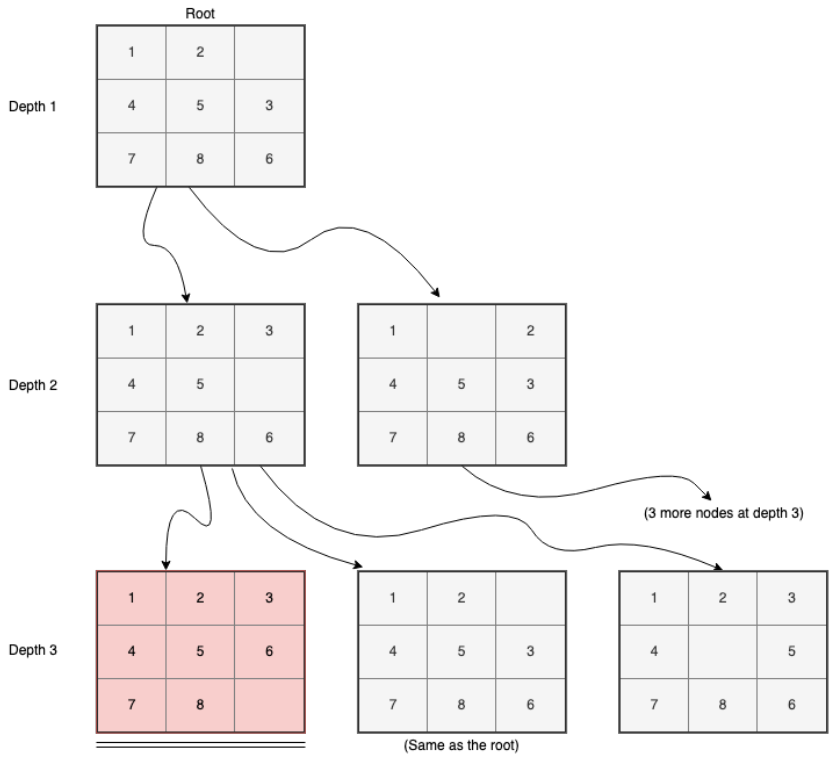

Figure 4: Example Tree Structure

For representing each of the possible two moves for the root configuration, two more nodes are created. Then, for each of these new configurations, new nodes are created again. Once the goal configuration is reached at depth 3, the program should terminate and report the obtained path.
Also notice that, at the 3rd level of the tree, there exist a configuration which is exactly the same as the root. Unless we discard such configurations that we have encountered before, we construct the same tree again and again within the same tree, which causes thousands of unnecessary nodes to be created. You should

handle this problem, otherwise your program becomes super inefficient. This leads to unnecessary utilization of the computer resources, which eventually results in minutes of runtime or even worse, you run out of memory and your program chrashes. (p.s. It is possible to solve any 3x3 puzzle in less than 1 second.)

3 Input & Output Format

Your program will run with two arguments. The first one will be the name of the input file (i.e. the initial configuration of the puzzle) and the other one will be the name of the output file. These files shall be under **src** directory.
The input file will contain a single line string of N^2 numbers separated by **'-'**, where 0 represents the empty cell. For instance, please check the following example of an input string that represents the initial configuration of the corresponding table:

4	6	7
3	2	1
5		8

Table representation of the input string: "4-6-7-3-2-1-5-0-8"

Your program must write the solution to the output file in the following format:

- Write all steps without blank spaces or newline characters.
- Starting from the initial configuration, for **LEFT** move, write letter **L**
- For **RIGHT** move, write letter **R**
- For **UP** move, write letter **U**
- For **DOWN** move, write letter **D**

At the termination of the program, the output for the puzzle given above should be the following:

ULURRDLULDRURDLDLUURDDR

which means, starting from the initial configuration, if we move the blank box UP, LEFT, UP, RIGHT, RIGHT ... then we end up with the goal configuration.

If an input configuration has no solution (for 3x3 tables only), then write a single letter **N** to the output.
Here are some inputs and corresponding outputs that might be useful during your implementation:

- *Input: "1-2-0-3-6-4-5-7-8", Output: DLLDRRULLURRDLULDRRD*
- *Input: "1-2-3-4-5-6-8-7-0", Output: N*
- *Input: "2-10-4-0-1-6-3-7-5-14-12-8-9-13-11-15",*
  Output: LDLULDDDRUURRDLDR
- *Input: "1-2-4-5-10-6-7-3-14-9-0-11-12-8-15-16-17-13-18-20-21-22-23-19-24",*
  Output: RRRURULLDDDRDR

4 Implementation Details

- **You must implement the project using tree structure explained above. Otherwise you will get no credit from the project even if it gives the correct output.**
- Note that, the output is not unique. You can reach the goal state through different ways as well. For example, a **U** action followed by a **D** action (and vice versa) does not affect the end configuration; you will end up with the same configuration.

  Of course, some solutions are optimal and some suboptimal. The outputs which are unreasonably long, **i.e. the outputs that are more than 20% longer than the length of the optimal solution** will be penalized. That is, the correct solutions will be graded but suboptimals will be penalized up to some points.

  Here is an example: Assume that there is a solution with 10 moves but your program gives a solution of length 13. Then, your output is unfortunately unreasonable, since the solutions up to length 12 are considered acceptable. As a result, you would not get full score for this case.

- If you want to create your own inputs, you better begin from the goal state of a table and then make moves on this configuration. By doing this you can guess the number of moves (a.k.a tree depth) needed to reach to the goal state. In such a case, starting with a few number of moves instead of many of them, which causes thousands of different configurations and huge tree width/depth, would be better.

- Minutes of runtime for this project is not reasonable. Of course you are not supposed to write a super efficient code but it shouldn't last more than $\approx$ 1 minute. **Think smart and do not hesitate to use structures you have learned in the course.**

- We will test inputs which are solvable at most 20 depth for 3x3 cases, and at most 15 depth for both 4x4 and 5x5 cases (possible in $\approx$ 1 minute).
- **You must place all code files (.java files) under the *src* directory, without any package.**

5 Bonus Points

When grading is done, we will run your codes with a special input, which is in the same format, but more difficult. The top 20 codes which solve this case relatively faster and in less than 30 seconds will get bonus points ranged from 1 to 20. That is, the first code will get 20 bonus points where the 20th code will get 1 bonus point.