Final Project

COMP 250 Winter 2021

posted:	Friday, Apr. 16, 2021
due:	Friday, Apr. 30, 2021 at 23:59

General Instructions

- **Submission instructions**
  - Please note that the submission deadline for the final project is very strict. **No submissions will be accepted after the deadline**.
  - As always you can submit your code multiple times but only the latest submission will be kept. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and codePost may be overloaded during rush hours).
  - This is the file you should be submitting on codePost:
    - `ChessSudoku.java`

    Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks
- Please note that the classes you submit should be part of a package called `finalproject`.
- Starter code is provided for this project. Do not change any of the class names, file names, method headers. You can add helper (i.e. `private`!) methods if you wish. Note also that for this project, you are NOT allowed to import any other class (all import statements other than the one provided in the starter code will be removed). **Any failure to comply with these rules will give you an automatic 0.**
- The project shall be graded automatically. Requests to evaluate the project manually shall not be entertained, so please make sure that you follow the instruction closely or your code may fail to pass the automatic tests.
- Whenever you submit your files to codepost, you will see the results of some exposed tests. These tests are a mini version of the tests we will be using to grade your work. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will test your code on a much more challenging set of examples. We highly encourage you to test your code thoroughly before submitting your final version.
- You will automatically get 0 if your code does not compile.
- Failure to comply with any of these rules will be penalized. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Piazza.

Learning Objectives

This project is meant for you to practice once again working with recursive algorithms. In particular, you will learn about backtracking algorithms which at the base use the idea depth-first search on search trees. The starter code provided for this project is not as extensive as in the past assignments. This will leave you more freedom on how to go about implementing the tasks your code is required to perform.

Introduction

A Sudoku is a Japanese logic puzzle which has been extremely popular for the past 20 years. In classic Sudoku, the player has to fill a 9 by 9 grid using digits from 1 to 9 making sure that no digit is repeated on any row, column or 3 by 3 box composing the grid. If you have never done so, I would suggest you to try solving some Sudoku puzzles (you can find many here). As you might know, many different variants of Sudoku exist. My personal favourite is Chess Sudoku which I discovered when last summer I found myself binge watching a YouTube channel called Cracking the Cryptic (by the way, I highly recommend the channel even if you think Sudoku is not for you. You'll change your mind!). A Chess Sudoku puzzle is a classic Sudoku puzzle with the addition of one or more of the following chess inspired rules:

- *Knight Rule*: a digit must not appear a chess knight's move away from itself. Knights in chess move forming an L shape: either two squares vertically and one horizontally, or two squares horizontally and one square vertically. See this figure for a graphical representation.
- *King Rule*: a digit must not be a King's move away from itself. Which to the classical rules of Sudoku only adds the fact that a digit cannot be a single diagonal move away from itself.
- *Queen Rule*: every 9 in the grid acts like a chess Queen and must not be in the same row/-column/3x3 box or diagonal of any other 9.

For this project you will write a program that will attempt to solve Classical and Chess Sudoku puzzles. The starter code contains a simple class representing a Sudoku puzzle. The class also contains several methods that allow you to load a Sudoku grid from a txt file and display it on the screen.

Your goal is to implement the method `solve()` which modifies the `ChessSudoku` object on which it is call, by filling in the grid to obtain a solution.

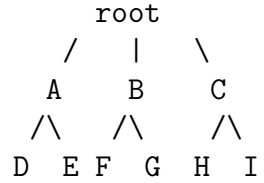
Backtracking

There are many different ways you can write a program that solves Sudoku's puzzles. A common way to approach the problem is using a backtracking algorithm. Backtracking is a brute force approach that consists in building a set of solutions incrementally. The idea is to use recursion to build a solution step by step. In the usual scenario, we have different options at each step and we simply make a decision. As soon as the algorithm realizes that the step taken led to a set of solutions that would be invalid (i.e. they won't satisfy the constraints of the problem), then it backtracks,

effectively eliminating all solutions that could be obtained by taking that last step. You can think of this as if the set of all possible solutions is represented by the leaves of a tree where each internal node represents a step taken toward a possible solution (i.e. a partial solve to the problem). Then a backtracking algorithm would perform a depth-first search on this tree and as soon as it figures out that the solutions obtained in a certain branch would be invalid, it simply discards that whole branch and moves to the next one.

For example, when we begin to solve a problem we can think about starting at the root of a tree where its children denoted the possible choices we have available. By choosing one of its children (i.e. by making a choice) we take a step toward a certain sets of possible solutions (i.e. the leaves that are reachable from that child). Other choices will then be available to us and we'll keep taking steps forward until we either reach a leaf (i.e. a solution) or an internal node which we know will not lead to any valid solution. If we reach a valid solution, then our task might be over. On the other hand, if we reach an invalid solution or an internal node that will not lead to any valid solution, we take a step back and at the previous level of the tree we choose the next child.

Consider the following tree:



The leaves denote the set of possible solutions to the problem. Let's say that only `G`, `H`, and `I` are valid solutions. Now, a backtracking algorithm would start from the root and visit the first child (i.e. take a decision that represents a first step toward a solution). We might then already realize that `A` would not lead to any valid solution. If that's the case we would then backtrack, undoing what we had done in the last step and getting back to the root node where we'll then take the next available choice that will lead us to node `B`. From `B` the next step will lead us to the leaf `F` which does not represent a valid solution. We will then backtrack, undoing what we had done in the last step and getting back to `B` where we can then take the next available choice. This will lead us to `G` which happens to be a valid solution and hence we are done.

How can you apply this type of reasoning to solving a Sudoku? The idea would be the following:

```
as long as there are empty cells in the grid
    pick an empty cell and place a number in it
    if a conflict occurs (i.e., at least one of the Sudoku rules is broken),
        then remove that number and backtrack

    if there are no conflicts, attempt to fill in the rest of the grid

    if you were unable to fill the rest of the grid
        then remove that number and backtrack
```

It is up to you to figure out how to code this idea into your program. I would highly suggest you

to spend some time solving a couple of Sudokus before starting to write your program. This will allow you to better understand the problem at hand and think about possible steps you can take to optimize your program. This project is much more open-ended than your previous assignments and it leaves you the freedom to be more creative and try to bring together everything you have learned throughout the semester.

Your Task

In the `ChessSudoku` class implement the method `solve()`. You can find additional information on the method in the template code. Please feel free to add as many `private` helper methods/classes as your see fit.

Testing and Evaluation

Your program will be tested on several classic Sudoku puzzles, as well as different variants of the chess Sudoku puzzles.

Together with the template code, we provided you with a set of txt files representing different Sudoku puzzles with different difficulties which you can use to test your program. Following the same format you can create as many new puzzles as you like to further test your code. The puzzles we'll be using to grade your assignment will remain secret. Please note that the txt files should be stored in the **project folder** of your program, and not the package folder, nor the src folder.

This is how we will evaluate your code:

+ **75%** Your program finds a solution to 3x3 Sudoku puzzles in less than 10 seconds (assuming they have a solution). The points awarded will be proportional to the number of puzzles from the secret tests your code can solve.

+ **10%** Your program find a solution to square puzzles up to 5x5 (assuming they have a solution).

+ **10%** Your program has an additional feature that finds all possible solutions to a given puzzle. This will involve continuing the search after the first solution is found (i.e., pretending that solution doesn't exist) to see if another solution can be found. All the solved Sudokus should be correctly stored in the field named `solutions`.

last 5% The remaining 5% of the grade will be established based on your code's ranking in a tournament where each program will be given one minute to solve each Sudoku puzzle on our computer. The ranking is established based on the number of Sudokus it succeeds at solving, from a set of 10 problems ranging from very easy to very difficult.