Informatics 1: Object Oriented Programming

Tutorial 06

Week 7: 04/03 - 08/03

Volker Seeker (volker.seeker@ed.ac.uk)
Naums Mogers (naums.mogers@ed.ac.uk)

1 Introduction

In this tutorial you will practice working with Java collection classes by extending the Sudoku solver and implementing a ranking algorithm for Sudoku grids. The Ranker will be able to give a score to a new Sudoku game which indicates the difficulty of the given grid based on existing solutions and available clues. The less clues and the fewer solutions a Sudoku game has, the more difficult it is. For this purpose you will have to extend your existing code to find all possible solutions to a Sudoku game instead of only a single one and you will have to implement the ranking algorithm.

2 Exercises

The tasks of this exercise are broken down into three parts. The first part extends your existing backtracking algorithm so that it does not stop after a single solution but rather remembers it and keeps going to find other ones. The second part uses this new functionality to implement a ranking algorithm for a new Sudoku grid and the third part will make use of the `Ranker` for an example case.

Finding all Solutions

To find all solutions for a given `GameGrid` requires you to extend the existing backtracking algorithm so it would continue trying numbers after it found a solution. After a solution was found, it should be remembered so that you can return a list of all possible solutions after the backtracking has finished.

Task 1 - Extending the `Solver`

◁ **Task**

The following description will assume that you create a new function `findAllSolutions` in your `Solver` class but you could also simply adapt the function `solve` you already have.

Create a new function `findAllSolutions` in you `Solver` class which gets a `GameGrid` parameter and returns an `ArrayList` of `GameGrids`. It will use the iterative backtracking algorithm you already have while saving all discovered solutions in an `ArrayList` which is then returned.

Based on the existing backtracking algorithm, you need to change the part where you stop searching once you reach the end of the grid after filling in valid numbers into every field. Unlike before, you now make a copy of your current `GameGrid` and save it into an `ArrayList`. Then have the backtracking algorithm continue with the next possible number in the field you have filled in last, starting in the bottom right corner. It will then continue as usual until it finds a new solution or exhausts the search. Once no further solution can be found, return the list of solutions you remembered.

Task 2 - Menu integration ◁ Task

Integrate the new `findAllSolutions` function into your main menu and print all solutions you could find for the loaded Sudoku game from the returned `ArrayList`.

When you test this, be aware, that some of the given example Sudoku files might take a long time to be exhaustively searched if a solution exists at all.

Ranking Sudoku Games

To give a difficulty rank to a new Sudoku game[1], you can make use of two rules:

- The fewer solutions a Sudoku game has the higher its rank, i.e. a game with $n+1$ solutions has a lower rank than a game with n solutions. Since a solvable Sudoku game must have at least one solution, $n = 1$ is the optimum for this rule.
- Should two Sudoku games have the same number of solutions, the one with less given numbers, i.e. clue fields, should be ranked higher than the one with more given numbers. In the (unrealistic) best scenario, all 81 fields can be free.

Task 3 - Ranking algorithm ◁ Task

Create a new class `Ranker` with a `static` function `rankSudoku` which gets a `GameGrid` parameter for the game to be ranked and returns a `float` value with the calculated rank.

In this function, make use of your new `findAllSolutions` function to calculate the first parameter, i.e. number of possible solutions. Then count all free fields in the given `GameGrid` to get the second parameter, i.e. number of free fields. To do that, you can create a new `public` method in `GameGrid`, called `getFreeFields`, or simply use `getField` for each field to calculate it in the `Ranker`.

Now make use of the two parameters to come up with a rank which is better the less solutions exist and the more free fields. How can you best handle the case where a Sudoku has no solutions?

HINT: Since the optimum for number of solutions is one, it is a possibility to make this your best rank while worse ranks are between zero and one or numerically higher than one.

Task 4 - Menu integration ◁ Task

Integrate the new `rankSudoku` function into your main menu and print the resulting rank for the loaded Sudoku game.

Ranking Example

To test the ranking algorithm (and practice a bit more with collection classes) you will now write a program which reads multiple Sudoku games from a given directory, assigns a rank to each of them and returns the one with the highest rank, i.e. the most difficult one.

[1] There are only clue fields and empty fields yet

Task 5 - Loading multiple files ◁ Task

To keep things where they belong, you should implement the loading of multiple files as an extension of the `IOUtils` class. In this class, write a new function `loadFromFolder` which gets a `String` parameter indicating a directory with Sudoku games and returns a `HashMap` which associates `String` file names of Sudoku games with their corresponding `GameGrid`.

In this function you can create a `java.io.File` object from the given directory `String` to check whether the directory `exsists` and is in fact a directory (`isDirectory`). Have a look at the Java API to find out more. Should any of those checks be `false`, return an empty `HashMap`.

Using the method `list` from the `java.io.File` API, you can produce an array of `Strings` which contains all file names in that directory. Create a `GameGrid` instance for each of those file names and put them into a `HashMap` where you associate them with their file name. You need to make sure you provide the complete path to a file rather than only its name when loading it. Also make sure you only create `GameGrid` instances for actual ".sd" Sudoku files.

Task 6 - Finding the most difficult Sudoku ◁ Task

Integrating the testing of multiple files into your existing menu does not make much sense since it is designed for a single game only. Therefore, create a new `main` function in the `Ranker` class where you provide the directory `String` as command line argument.

Use the new `loadFromFolder` function to create `GameGrid` instances for all Sudoku files in the given directory. Then calculate the rank for each of them and print the file name and rank of the highest ranking Sudoku in your collection.

To give you an idea, here are the number of available solutions in the provided Sudoku files:

```
sudoku0: 1 solution
sudoku1: 1 solution
sudoku2: 2 solutions
sudoku3: 1 solution
sudoku4: 1 solution
sudoku5: 1 solution
sudoku6: 1 solution
sudoku7: 8 solution
sudoku8: 8 solutions
```

3 Optional Extra Tasks

If you have finished all of the above tasks and look to do more, please consider the following suggestions or come up with your own ideas:

Sudoku Generator Using the ranking method implemented in the previous part as cost function, write a Sudoku game generator using simulated annealing. You can find a good explanation of this algorithm here:

http://katrinaeg.com/simulated-annealing.html

You can follow these guidelines to get started:

- The generator starts at a fully and correctly solved Sudoku grid.
- Every step of the algorithm going towards the solution can either remove a number or add one.
- Use the rank of your intermediate solution to check whether you reached the optimum yet or at least a sufficiently good rank.
- Intermediate solutions are only accepted if they are following Sudoku rules.

HINT: To make the backtracking algorithm work correctly when counting solutions for different steps in your generation, you might need to make the `initial` member of your `Field` class mutable.