Recruiting Assignment – Sudoku

Date: 2016-06-26  
Candidate: Jan Miderbäck <jan.miderback@live.com>

# Summary

The solution consists of the sudoeng.exe command line program written in C#. All sources and the exe itself are provided in the accompanying sudoeng.zip file. sudoeng.exe is capable of solving and generating Sudoku puzzles of standard size 9 x 9. The program is used in the following way

sudoeng -s <filepath>

Solves the Sudoku problem represented in the file <filepath>. The solution is printed to standard out, together with additional information such as if the solution is unique, calculation time, estimated difficulty level (VeryEasy, Easy, Medium, Hard, Samurai), and some other measures from the solver.

sudoeng -g <difficulty>

Generates a Sudoku problem with the difficulty level <difficulty>, which can be any of veryeasy, easy, medium, hard, or samurai. The generated problem is written to standard out. Following the generated problem come some measures from the generator, like targeted and graded difficulty levels which tells if the generator managed to generate a problem at the targeted difficulty level.

# Unsolved And Tricky Parts

The sudoeng.exe program provides solutions for Part 1 – 3. Some problematic areas exist and I’ll try to explain what limitations I think there are and ideas on how to improve.

Part 4 is not solved. I have not implemented any graphical user interface. My intention was to write a Forms application that would work on top of sudoeng.exe, passing information to it by invoking the program and then reading back results by reading standard out from the program. It would have been really nice to provide some part of this but the challenges to both solve, generate, and grade Sudoku problems took more of my attention, interest, and time.

## Difficulties in the Grading Algorithm

I realize that the grading algorithm implemented is not optimal. It has problems in determining the correct difficulty level. Looking at the problems provided with the assignment it makes the following grading

* The “easy” problem is graded as Medium
* The “medium” problem is graded as Hard
* The “hard” problem is graded as Samurai
* The “samurai” problem is graded as Samurai

This indicates that the solver in general grades problems too high. One could think that a simple calibration of the grader would fix this, but that is not the case. The problem is more likely to be with the grading measures I have implemented.

They are

* The number of given squares in the problem. Lower number of givens gives a harder problem.
* The lower bound of given squares in rows and columns in the problem. A lower “lower bound” gives a harder problem.
* The number of search calls made by the solver. A high number of calls (and backtracking) indicates a harder problem.

While all these measures are valid, there is one measure I would like to have implemented in addition to these and that is to determine which set of “human” logical reasoning patterns (CRME, Lone Rangers, Twins, Triplets, Swordfish etc. [3]) that is needed to solve a problem. Implementing that and weighing that in with the other measures would give the solver a much better chance to grade a problem correctly, based on what a human would need to do to solve it.

I estimated that the time I would need to spend implementing solvers using these patterns would take considerably more time and I would not be able to spend time on the generating problem. There could be simpler ways to get better grading, but I did not investigate it further.

## The Las Vegas Terminal Pattern Generator

The problem generation algorithm is based on generating a terminal pattern (solution) and then to do hole digging into that terminal pattern. I suspect that there is some problem in that the Las Vegas randomization of initial givens does not terminate in reasonable times. Sometimes it exits really quickly but I had to introduce a timer so that it runs for max five seconds before reinitializing and making a new attempt. While this is natural for a Las Vegas approach, the pattern of execution times tells me something is not fully correct. I have not spent time looking deeper into it. I have yet not seen that it has not terminated within a minute.

## Symmetry in Generated Problems

I tried to create symmetry in the generated patterns using the following strategies to dig holes

* Walk the grid in a straightforward pattern, based on the difficulty level. Resources [2] suggest that certain patterns are better than others at generating problems of certain difficulties.
* Once a square has been dug, try to dig diagonally opposite squares to maintain a level of symmetry among the remaining given squares.

While some symmetry can be spotted in the generated problems, I think there is some work needed to try to maintain even more symmetry. I have not investigated further what changes that would be needed.

# The Design

The design of the program should be pretty easy to understand and I have separated the solution into a number of different classes, each with well-defined responsibilities. Below follows a brief description of them

* Program – Contains the Main method and the entry point of the program. Parses the command line argument in a very simplistic way. Creates instances of classes needed to either solve or generate a Sudoku problem. Writes results back to standard out.
* Parser – Parses a Sudoku problem and provides a representation of the problem in the Grid class (see below)
* Grid – Represents the state of a Sudoku problem as values in each square of a Sudoku grid. Values in squares are represented by the SquareValue bit field enumeration.
* Reducer – Reduces a Sudoku grid eliminating values that are not possible at rows, columns, and mini-grids based on the game rules.
* Solver – Implements a backtracking, brute-force solver [1]. The solver is capable of determining if the solution is unique.
* SolverContext – Contains information from running the solver.
* Generator – Generates a Sudoku problem based on Las Vegas randomization of an initial set of givens and then the solver is used to produce a terminal pattern. Then, the problem is generated through hole digging.
* GeneratorContext – Contains information from running the generator.
* Grader – Calculates the difficulty of a problem by a weighted sum of scores, where the scores measure the difficulty of the problem using different metrics.
* Grading – Represents the final grading of a problem which include the difficulty level, which can be Easy, VeryEasy, Medium, Hard, or Samurai.

## Improvements

While the design and implementation is pretty straight forward, in order for it to qualify into production, some things would need to be improved

* Dependency injection should be introduced, but it is not. Factoring out into interfaces like ISolver and IGenerator etc. would provide for better patterns and would first of all enable better foundations for unit testing with mocking of implementations for true testing in isolation. I did not spend time on this.
* The Generator class is on the limit of being too big. To factor out the terminal pattern generator, the hole digger, and the propagator implementations into separate classes/interfaces would be better.
* There is some variance and inconsistency how object creation is performed, and whether data is passed in constructor or in arguments to calculation methods. Looking it over to get better consistency would be good.
* Naming and ordering. I have tried to be consistent but there are cases of inconsistency. I have not adopted any particular coding standard.
* There are likely places in the code where using some fancier C# language features would make the code more beautiful and more concise.
* Most likely, there are unnecessarily many copies of the Grid class created. In many places it is probably possible to reuse existing instances of the Grid class instead of copying values over into a new instance and then modify the copy. However, copying Grid instances should not be the performance bottleneck of the solution.

# References

1. Solving Every Sudoku Puzzle   
   *Peter Norvig*<http://norvig.com/sudoku.html>
2. Sudoku Puzzles Generating: From Easy to Evil  
   *XUE Yuan-hai, JIANG Biao-bin, LI Yong-zhuo Advisor: YAN Gui-feng, SUN Hua-fei  
   (Department of Mathematics, Beijing Institute of Technology, Beijing 100081, China)*<http://zhangroup.aporc.org/images/files/Paper_3485.pdf>
3. Sudoku Creation and Grading  
   *Andrew C. Stuart*<http://www.sudokuwiki.org/sudoku_creation_and_grading.pdf>