Sudoku Solver

Emil Vikström

18th January 2013

Abstract

A backtracking solver for sudoku-like puzzles was implemented in Haskell, together with a web GUI and compiled to JavaScript with the Haste compiler.

1 License

Contact me at rsemil@gmail.com or http://www.emilvikstrom.se/ for source code requests.

2 Sudoku rules

Sudoku is a number game. Each row, column and region in figure 1[2] should be filled with the numbers 1 to 9, which means that each row/column/region may only contain each number one time. A well-designed sudoku does only have one solution.

3 How it Works

3.1 Algorithm

A backtracking algorithm is used. The general backtracking algorithm can be understood as building a tree of all possible solution to each subproblem, and then cut away those subtress that are not going to produce valid solutions to the main problem. This can be implemented in a functional language with a recursion where you optimistically try one of multiple possible solutions to each

[1] http://www.gnu.org/licenses/gpl-3.0.html

[2] Stellmach & King, Wikipedia 2009, http://en.wikipedia.org/wiki/File:Sudoku-by-L2G-20050714.svg

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 1: Sudoku

subproblem, until you either find a complete solution or find yourself stuck with a non-solvable problem. If you get to the "stuck" phase you go back (backtrack) and try a different solution to each subproblem. It is a depth-first approach where you optimistically pick a solution and go back only if it fails.

This is implemented in the sudoku solver as a recursion where at every step each possible value is tried for a cell and if a solution is found returns the complete solution.

The current algorithm is naïve and tries each field in order. Great speedups could probably be had by doing some smart picking of the subproblem (cell) to solve but that is not investigated at this time.

3.2 Definitions

These are the definitions I use in this project and in my solver.

Value An integer

Cell A single cell. The cell may have a single known value, or a set of possible values.

Area A set of cells, a "constraint area". Each known cell must have a unique value in the area. A cell may be included in multiple areas.

Row/Column Rows, columns and regions from the original sudoku definition are represented as areas.

Done The puzzle is solved when all cells have known values.

3.3 The General Solution

Each cell is member of a set of areas. Each area must have only unique values. This allows for a general solution where each row is an area, each column is an area and each region is an area.

Solving a subproblem (setting the value for a cell) means picking a value that is not already chosen by another cell in any of the areas the cell is part of. We know we are stuck (need backtracking) if there is no possible value to pick.

This general solution is easy to extend to similar puzzles. At this time the solver is able to solve 4x4, 9x9 and 16x16 sudoku puzzles with the included templates, but it is possible to add more templates without changing the solving engine (GUI code needs to be added as well if the web interface is to be used). Here are some other variants variants that is possible to solve by just adding more templates:

- Samurai (figure 2[3])
- Sudoku with irregular regions

[3] Nonenmac, Wikipedia 2008, http://en.wikipedia.org/wiki/File:A_nonomino_sudoku.svg

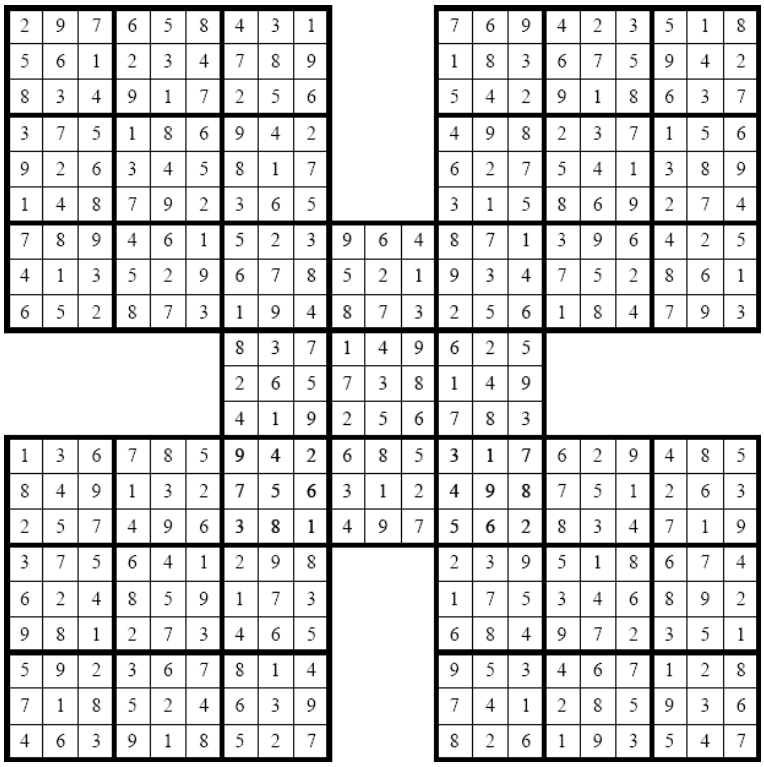

Figure 2: Samurai

- Non-symmetric puzzles (for example with 3x2 regions)
- A puzzle where some cells can contain only a subset of the possible numbers (3, 4, 5 for example)

3.4 Implementation

I choose to implement this using *Data.Array* for to keep track of all areas in a puzzle. The datatype was choosen for fast, $\Theta(1)$ lookup of values. Cells are then part in areas only by the area index (which is also the index in the array). *Data.Array* is immutable so it won't break referential transparency.

Each area is represented using only a list of known values in the area. That way not all cells must be traversed to check if a value is set in the area (a bit array may give even better performance).

A puzzle is a list of cells. Each cell contains information about (possible) value(s) and area indices. This list is compiled into the initial area array.

The backend (solver) and frontend (web UI) is separated into two modules, Solver and Main. Solver may be independently used in other projects for solving

sudoku-like puzzles.

Main makes heavy use of the Haste library, included with the Haste compiler. The library contains functionality to manipulate the HTML DOM tree (adding and removing elements) as well as listening for events from the GUI. Everything is contained in the IO monad.

4 Performance

The recursion step will theoretically try all possible values except when it encounters something that breaks the invariant for an area. This is in the end a brute-force approach which may take a very long time in the worst case. In the worst case a lot of values will be tried before reaching a backtracking point. Large subtress may then need to be thrown away in which case the calculations leading up to that point was wasted.

My non-scentific tests of some sudokus show that the general performance is very good, though, with solution times under a second for many puzzles designed to be hard, extreme or worse for humans. But some problem-puzzles exists. An extreme example can be seen in figure 3[4] which was specifically designed to be hard to solve by brute force. That puzzle took a few hours to solve on my machine, in Opera 12.

Performance may vary between different browsers, of course.

5 What I Learned

The backtracking algorithm was very easy to implement. It is a straightforward divide-and-conquer algorithm. I am pretty happy with the general approach, though. I will definitely work some more on this to at least get Samurai sudokus working.

The hard part was the GUI. Haste had some quirks to get going (as most smaller projects), but the overall quality of Haste was very good; it compiles to small and fast code. But the hardest part was using monads. I have read about monads and understands the theoretic foundations of it, but using it requires some practice. I ended up with a lot of type errors at first, but after I while I started to get a hang of it.

6 Usage

You need the Haste compiler[5] to compile this to JavaScript. An already-compiled version of *Main.js* is included in the distribution if you just want to try it out. Open *index.html* in a web browser with script support.

If you have Haste, run *make Main.js* to compile your JavaScript file.

[4] Lithiumflash, Wikipedia 2007, http://en.wikipedia.org/wiki/File:Sudoku_puzzle_-hard_for_brute_force.jpg

[5] Anton Ekblad 2012, https://github.com/valderman/haste-compiler

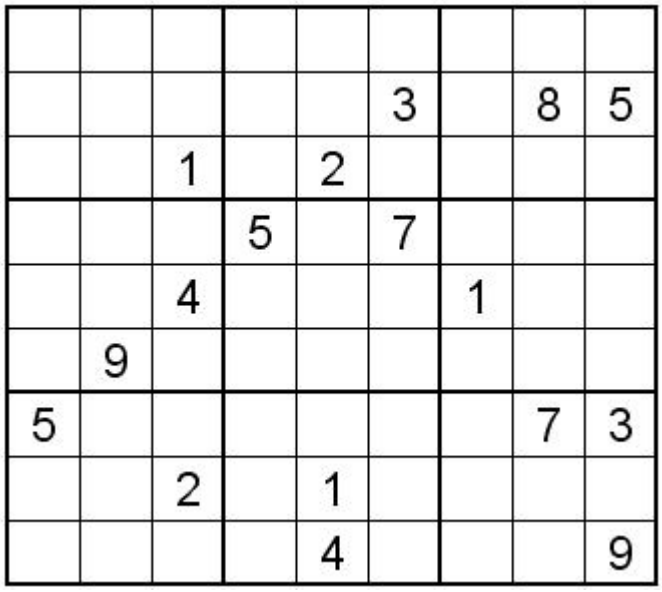

Figure 3: Sudoku puzzle hard to brute force

Solver.hs may be independently used as a module for your own projects. It is tried with the Glasgow Haskell Compiler. Haddock dokumentation can be generated by *make doc*