



Generating Sudoku

Eric Maki
eric@uc.org

Sudoku solvers are wonderful programming exercises. I've written a couple now, and both of them exercised my brain in interesting ways. The truth is that just sitting and doing Sudoku puzzles might have done the same, but I lack the patience. Why do the work myself when I have an x86 genius peering over my shoulder?

Generating Sudoku is in the realm of the NP-complete. It's not a decision problem, but generating a puzzle can be mapped to two NP-complete problems. I'll address these two problems in this article, and the result will be a Sudoku puzzle generator in Perl.

The first problem is to generate a solution grid: given nothing, return a 9 by 9 grid that satisfies all the constraints of Sudoku. This is essentially a standard Sudoku solver on steroids—I feed in a puzzle with no clues, and get a solution. The second problem is to decide which clues to remove to produce a puzzle. While researching my talk, I discovered that many of the most efficient generators and solvers (mostly written in C/C++) used a very interesting strategy. Rather than a data representation that dealt with a grid of possible values, they map the puzzle to a different puzzle, something called an 'exact cover' problem.

Exact Cover

The exact cover problem is: Given a boolean matrix, determine if there is a set of rows that when taken together, have a single 1 in each column.

```
1 0 1 0 1 0
0 1 0 0 1 0
0 1 0 0 0 0
0 0 0 1 0 1
```

In this trivial example, I can see by inspection that rows 0, 2 and 3 satisfy the conditions and exactly cover the columns. I wasn't able to find a Perl Sudoku generator using exact cover, so I decided to give it a try. I haven't determined if this is the most efficient

solution possible, but I think that it is a tremendously graceful one.

Once I wrote the exact cover solver, the rest was trivial. To implement a recursive exact cover solver is relatively straightforward. Here's the basic algorithm:

- pick a column **C**
- enumerate the rows that have a 1 in column **C**
- base-cases:
 - if there are no remaining columns, succeed
 - if **C** contains no rows, fail
- for each row **R** in the above set:
 - place **R** in the solution set
 - delete all the columns that **R** has a value of 1
 - delete all rows that had 1's in any of those columns
 - solve this derived puzzle

The algorithm uses the typical recursive approach of solving one component of a puzzle, then recursing to solve the remainder. If the recursive call to solve the sub-puzzle fails, I simply try the next row. In this manner I will eventually try all of the rows at each level until I hit them all or I find a solution. The column selection, though, is permanent. It might seem like a lot of work, but think of it this way: this problem maps all of the constraints of Sudoku into a single

Eric Maki designed the cover for this issue, using his program to generate the complete solutions in the background as well as the minimal puzzle left for you to solve. You can get the full code on our website.