Combinitorial Generation of Minimal Well Formed Sudoku Puzzles

Taras Mychaskiw
COSC 4P03 Project

April 22nd, 2014

Abstract

The purpose of this project was to generate minimal well formed sudoku puzzles. A well formed sudoku puzzle is one that has exactly one solution, and a minimal sudoku is one where you cannot remove any clue without making the puzzle not well formed. Several sudoku generating strategies are employed, as well as several solving strategies as the generation of sudoku puzzles inherently requires solving them and counting the number of solutions to them.

Contents

1 Introduction

1.1 Terms

Here are the terms and names that will be used in this paper.

p	the width of each box in the sudoku
q	the height of each box in the sudoku
n	the size of the entire sudoku grid, equal to pq
cell	each point which contains a value in the sudoku
candidates	all the values that can lay in a cell
unit	a set of cells where no two cells may share a value (eg a row)
peers	the set all of cells that a cell may not share a value with
formity	describes if a puzzle has none, one or many solutions

1.2 Sudoku

Sudoku is one of the most popular puzzle games of all time. A sudoku puzzle involves placing the numbers 1 through n in a nxn grid so that each row, each column and each pxq box in the grid have each number exactly once. At the beginning of a puzzle, several cells have values filled in, and the player's job is to fill in the rest of the grid. It is always understood that every sudoku puzzle has exactly one solution to it. Figure 1 depicts a sample sudoku puzzle.

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

Figure 1: A standard 9x9 sudoku puzzle.

With the growing popularity of sudoku, many 9x9 puzzles are required to feed the demand. Additionally, there is some demand for larger or smaller sudoku puzzles (rather than 9x9) for a greater or simpler challenge.

2 Sudoku Solving Strategies

Generating sudoku puzzles inherently requires being able to solve them. More specifically, the solver must be able to check the formity of the sudoku puzzle and hopefully do it very quickly. Several solving strategies are used in the generation in hopes that one will outperform the others and speed the generation process. Note that each of the algorithms described below detail how to solve a puzzle, not how to get the formity of it. However, the algorithm to get the formity is only a slight modification. For example, one could increment a counter keeping track of how many solutions there have been so far once a solution is found instead of returning the solution right away.

2.1 Backtracking

Backtracking is the easiest solving strategy to understand. As the name suggests, it uses backtracking to solve the sudoku puzzle. Algorithm 2.1 describes the basic idea below. Essentially, the algorithm goes through each cell in the sudoku and tries assigning each possible value in a depth first search. If it finds a solution, the solved sudoku is returned right away. If some candidate selection leads to an impossible to solve sudoku, the cell is cleared of it's value and the backtracking tries again from the previous cell. To solve a sudoku puzzle using the backtracking algorithm, one would call $backtrack(sudoku, 0)$.

Algorithm 2.1: BACKTRACK$(sudoku, cell)$

comment: check if the sudoku is solved

if $cell >=$ GETTOTALCELLS$(sudoku)$
  then return $(sudoku)$
comment: if there is already a value here, go to the next cell

if HASVALUE$(sudoku, cell)$
  then BACKTRACK$(sudoku, cell + 1)$
comment: otherwise, try all candidates in depth first search

for each $cand \in$ GETCANDIDATES$(sudoku, cell)$
  do
- $sudoku[cell] \leftarrow cand$
- $board \leftarrow$ BACKTRACK$(sudoku, cell + 1)$
- **if** ISSOLVED$(board)$
  - **then return** $(board)$

$sudoku[cell] \leftarrow nothing$
return $(sudoku)$ **comment:** the sudoku is not solved

2.2 Constraint Propagation

Constraint propagation expands on the basic backtracking algorithm. It's more complex than backtracking, but solves sudoku puzzles much faster on average. Constraint propagation tries to solve a sudoku puzzle in the same manner that a human would. Each cell on the sudoku grid will contain a list of all the candidates to sit in that cell - similar to backtracking but they are used differently. Two rules are employed in this solving method:

(1) If a cell only has one possible value, *eliminate* the value from the cell's peers
(2) If a unit has only one possible cell for a value, *assign* the value there

These two rules continue to propagate, for example exercising rule 1 may trigger rule 2, which may trigger rule 1 and so on. Eventually, either each cell in the entire puzzle will only have one candidate (in which case the puzzle is solved), or the propagation stops and the puzzle is not yet solved. At that point, a version of backtracking will take place. A cell is selected, and one of it's candidate values is simply assigned to the cell. If the assumption was incorrect, a backtrack would occur and a different value would be selected for the cell. Which cell is selected differs from backtracking in that we choose the most constrained cell. That is, the cell with the fewest number of candidates, but more than one candidate, is found and a value is assigned there. This helps to limit potential backtracking. The fewer number of possibilities a cell has, the better chance that a random selection will end up being correct. To solve a particular sudoku puzzle, each of the cells that already contain values must be *assigned*, then call *search*. Algorithms 2.2 through 2.4 describe the flow of each function.

Algorithm 2.2: ASSIGN$(cell, value)$

global $candidates$
for $cand \in candidates[cell] - \{value\}$
  do ELIMINATE$(cell, cand)$

Algorithm 2.3: ELIMINATE$(cell, value)$

global $candidates, peers, units$
$candidates[cell] \leftarrow candidates[cell] - \{value\}$
if $|candidates[cell]| = 1$
  then $\begin{cases} v \leftarrow q \in candidates[cell] \\ \textbf{for } peer \in peers[cell] \\ \quad \textbf{do } \text{ELIMINATE}(peer, v) \end{cases}$
for $unit \in units[cell]$
  do $\begin{cases} otherCells \leftarrow \{\ s \mid s \in unit, value \in candidates[s]\ \} \\ \textbf{if } |otherCells| = 1 \\ \quad \textbf{then } \begin{cases} c \leftarrow q \in otherCells \\ \text{ASSIGN}(c, value) \end{cases} \end{cases}$

Algorithm 2.4: SEARCH()

global *candidates*
if $\exists\ set = \emptyset\ \forall set \in candidates$
  then return (**false**)
if $|set| = 1, \forall set \in candidates$
  then return (**true**)
$cell \leftarrow c$ such that $|candidates[c]| > 1$ **and** is minimal
for $cand \in candidates[cell]$
  do { ASSIGN($cell, cand$)
  if SEARCH()
    then return ($solved$)
    else REVERTASSIGN($cell, cand$)
return (**false**)

2.3 Sudoku as an Exact Cover Problem

2.3.1 Exact Cover

Before getting into how to solve sudoku puzzles by treating them as instances of the exact cover problem, we must know what the exact cover problem is. The exact cover problem is:

> given a collection S of subsets of a set X, an *exact cover* is a subcollection S^* of S such that each element in X is contained in exactly one subset in S^*

The exact cover problem is NP-Complete, proved by Karp in 1972[5]. One way to represent an exact cover problem is to use a matrix filled with boolean values. Each column of the matrix represent one element in X, and each row represents one of the subsets in S. Each of the boolean values in the matrix tell us if an element exists in the subset for that row. An example is detailed in the table below. Here, $X = \{1, 2, 3, 4, 5, 6\}$, and $S^* = \{A, D, E\}$.

	1	2	3	4	5	6
A	0	1	0	0	0	0
B	0	0	1	1	1	0
C	0	1	0	0	0	0
D	0	0	1	0	1	1
E	1	0	0	1	0	0

2.3.2 Algorithm X and Dancing Links

Donald Knuth introduced Algorithm X alongside Dancing Links in his 2000 paper[1]. Algorithm X is defined as a non-deterministic backtracking algorithm that solves the exact

cover problem when the problem is in the matrix form described above. The goal is to select a subset of rows of that matrix A so that the true value appears only one in each column. Algorithm 2.5 describes Algorithm X.

Algorithm 2.5: ALGORITHMX(A)

if $A = \emptyset$
  then return (**true**)
$col \leftarrow$ a column of A
$row \leftarrow$ a row of A such that $A[row][col] = 1$
include row in the partial solution
for $j \in \{ v \mid A[row][v] = 1 \}$
  do { **for** $i \in \{ w \mid A[w][j] = 1 \}$
    do remove row $i$from A
    remove column $j$from A
repeat using the reduced matrix A

Dancing Links is an implementation of Algorithm X. At first glance of Algorithm X, it seems that much time would be wasted searching for 1's in the matrix A. Dancing Links provides a clever way to eliminate that search while also providing a very easy means for backtracking to occur. Dancing Links uses a quadruply linked list to represent the matrix, where each node in the list only contains the 1's in the matrix and are connected to the next 1 in the matrix in direction. Additionally, there is a column header for each column, which each node in the column points to as well. Figure 2 shows an example of the gigantic list that makes up Dancing Links.

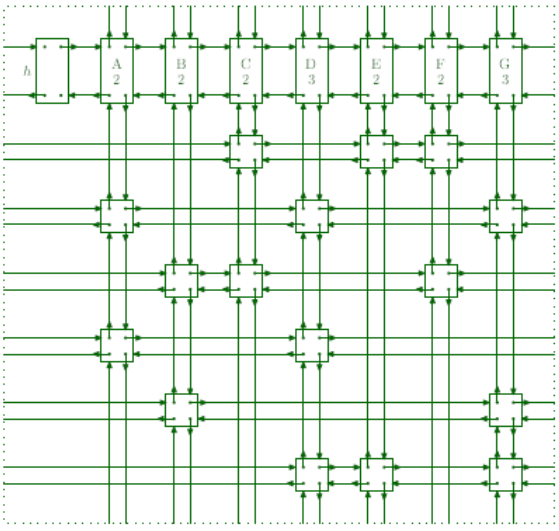

Figure 2: An example matrix representation using Dancing Links.

The list removes the task of searching for 1's in the matrix, but as mentioned it's also very simple to remove columns from and re-add columns for backtracking. To remove a node from a column, for example, it requires:

$$\begin{aligned} Node.left.right &= Node.right \\ Node.right.left &= Node.left \end{aligned}$$

And reinserting the node into the row requires:

$$\begin{aligned} Node.left.right &= Node \\ Node.right.left &= Node \end{aligned}$$

Using Dancing Links as the implementation for Algorithm X, solving the exact cover problem becomes very fast.

2.3.3 Solving Sudoku as an Exact Cover Problem

Sudoku can be represented by an exact cover problem. Using the rules of sudoku, we can come up with the set X and the collection of subsets S. If we think of each rule in sudoku as a column in the matrix A, then we can start to see how sudoku can be transformed into exact cover. Each of the rules must be satisfied exactly once, then the board is solved. If we get an exact cover of the matrix where each of the column is a sudoku rule, then we solve that sudoku. The rules state that the values 1 though n must appear exact once in each row, column and box in the sudoku. Another implicit rule, but a very important one, is that each cell can only contain one value. This sums up to $4n^2$ total constraints that must be satisfied. Each of the n rows, columns and boxes must contain n values ($3n^2$), and each cell must contain a value which adds the final n^2. Each of these makes up a single element in X - a single column of the matrix A.
The collection of subsets in n^3 in size. For each cell, any value can sit in it. We can represent these by row, column and value triples, such as (row 1, column 1, value 1), (row 1, column 1, value 1) ... (row n, column n, value n). Each triple satisfies four constraints, the row-value, column-value and box-values pairs, and that the cell is occupied. Using this information we can construct the matrix A then use Dancing Links to solve the puzzle.

3 Sudoku Generation

There are several ways of generating a sudoku puzzle. Three approaches are explored here. Each of them produces a random sudoku puzzle with a unique solution, where if you were to remove any clue, the puzzle would no longer have a unique solution. Each of the generator algorithms are fairly simple.

3.1 Top Down Generation

In top down generation, we start with a randomly solved sudoku board and randomly remove clues from it. With each clue removed, the puzzle so far is checked to ensure it still has exactly one solution. If it has more than one solution, then the value in the cell is replaced. This process is done for each cell in the sudoku.

Algorithm 3.1: TOPDOWN()

$sudoku \leftarrow$ a solved sudoku board
for $cell \in sudoku$
do
- remove the cell from sudoku
- $form \leftarrow$ formity of sudoku
- **if** $form \neq$ unique solution
  - **then** replace the value removed

return ($sudoku$)

3.2 Bottom Up Generation

Bottom up generation is the exact opposite of top down generation. Starting with an empty board, randomly add clues to random cells. Only those clues which are possible to sit in the cell are considered. Once a value is added, the formity of the sudoku is checked. If there are no solutions, the value added is removed, and the process is continued. If there are multiple solutions, the process is continued. The process stops once a sudoku with a unique solution is found. However this produces puzzles with approximately 30 clues in a 9x9 sudoku, so some post-processing is required. For each clue in the sudoku, try removing it. If the puzzle still no longer has a unique solution, then replace the value.

Algorithm 3.2: BOTTOMUP()

```
sudoku ← an empty sudoku board
while true
      do { cell ← a random empty cell ∈ sudoku
           sudoku[cell] ← a random possible value
           form ← formity of sudoku
           if form = unique solution
             then exit while
           if form = no solution
             then clear the cell placed
for clue ∈ sudoku
      do { remove the clue from sudoku
           if form ≠ unique solution
             then replace the value removed
return (sudoku)
```

3.3 Deduction Generation

Deduction generation was an attempt to produce sudoku puzzles with few clues. Starting with a solved sudoku puzzle, only add a clue to our returned puzzle if it is not deduced by all the other clues in our return puzzle already. That is, if a cell has more than 1 possible value, add the value from the solution to the cell. This also creates a 30-40 clue puzzle, so the same post-processing as bottom up generation is required.

Algorithm 3.3: DEDUCTION()

```
solved ← a solved sudoku board
sudoku ← an empty sudoku board
for cell ∈ sudoku
  do if |GETOPTIONS(sudoku, cell)| > 1
  then sudoku[cell] ← solved[cell]
for clue ∈ sudoku
      do { remove the clue from sudoku
           if form ≠ unique solution
             then replace the value removed
return (sudoku)
```

4 Results

A sample of $100,000$ (10 sets of $10,000$) sudoku puzzles were generated using each of the generation strategies discussed above. During each of these, all three sudoku solvers were

run concurrently to check formity of a puzzle. Once one solver got the answer, it would return the answer and the other two would stop calculating. The "winner" (the solver that took the least amount of time) of each of these was stored for an additional comparison of the solvers. All the results below (to do with this set) are are an average over the 10 runs, each of which calculated the average over generating $10,000$ sudoku puzzles.

4.1 Comparison of Solving Strategies

4.1.1 Solving Hard Puzzles

To compare the solving strategies, each of them were used to solve all of Peter Norvig's Top 95 sudoku puzzles[4]. These puzzles are considered some of the hardest ever found. Each of the solvers solved each puzzle, and checked the formity of each puzzle as well.

Solver	Total Time to Solve	Total Time to Check Formity
Backtracking	593475ms	1407571ms
Constraint	243ms	351ms
Exact Cover	166ms	99ms

Table 1: Results of each of the solvers solving and checking the formity of Peter Norvig's Top 95 sudoku puzzles.

Seems that exact cover and dancing links is far ahead of the competition. Constraint propagation didn't do too badly. Backtracking is just not as good as the other two. Something to note about the formity time for exact cover - it's less than the time to took to solve the same puzzles. Seems counter intuitive, but it has to do with how the solving was actually implemented. Solving a puzzle would return the solved version of the puzzle, while formity checking only returned an integer. In exact cover, post processing is required to convert the matrix back into a sudoku board, but this is not required in checking the formity. All that matters for formity is that a new solution was found (if it was), the actual solution is not overly important.

4.1.2 Use in Generation

Something surprising happened during generation - backtracking still won. It seems that, from the data above, backtracking would win very few times if at all. It was excepted that exact cover would be about 60% of the wins, and constraint propagation would be about 35%. However, this was not the case. Figure 3 should the results of the winners during the generation.

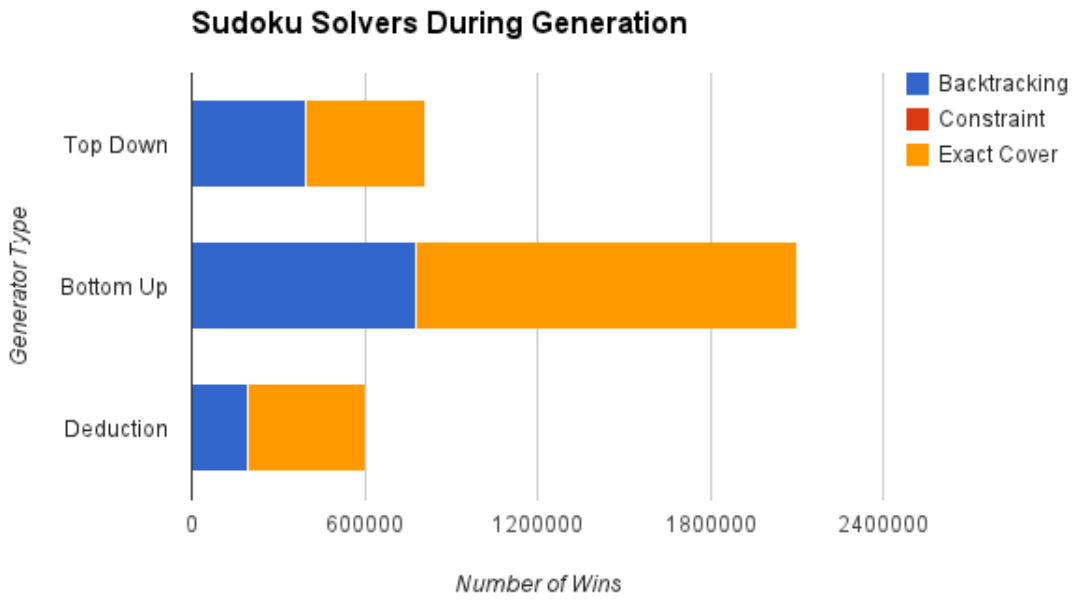

Figure 3: Results of how many times each solver won during each of the generation of sudoku puzzles.

As expected, exact cover does win about 60% of the time. However backtracking and constraint propagation are backwards (note this was *not* an implementation issue!). In fact, constraint propagation won so few times that it doesn't even appear in the graph. So why did backtracking do much better than expected? For top down generation, it actually makes sense. Backtracking requires absolutely no pre- or post-processing of the data. It just loops through the board, skipping clues already given in the puzzle. During top down generation, most of the puzzle is already given, so backtracking has very few calculations to actually do. Meanwhile, constraint propagation and exact cover are still pre-processing the board. Backtracking would win while the board is fairly full still, then later exact cover would outperform. And that's exactly the results we for the top down generation.
What about bottom up and deduction generation? Each of those start with an empty board that the solvers would run on. Backtracking is still a major contender. The difference between the results in the previous section and now is that the formity checks are on boards that are not well formed, they usually have more than one solution. Each of the solving strategies is a depth first search, so if there are many solutions, they would find two similar ones quickly and return that there are multiple solutions. The difference between backtracking and the others is again the lack of additional processing. On a near empty sudoku, basically any values you throw into a cell will lead to a solution with very little backtracking actually required. So, the same reasoning applies. While exact cover is still pre-processing, backtracking has gotten a head start, and may end up finding two very similar solutions very quickly and winning.

Constraint propagation didn't do anywhere near as well as expected. This is likely due to the unexpected backtracking results. Backtracking does exceptionally well in near full and near empty boards while checking for formity. In all other cases, exact cover would simply outperform constraint propagation. Given these facts, it makes sense that constraint propagation simply fails to show any results. If one were to redo the experiment without using exact cover, my personal estimate is that constraint propagation would take exact cover's position and win most of the time.

4.2 Comparison of Generation Strategies

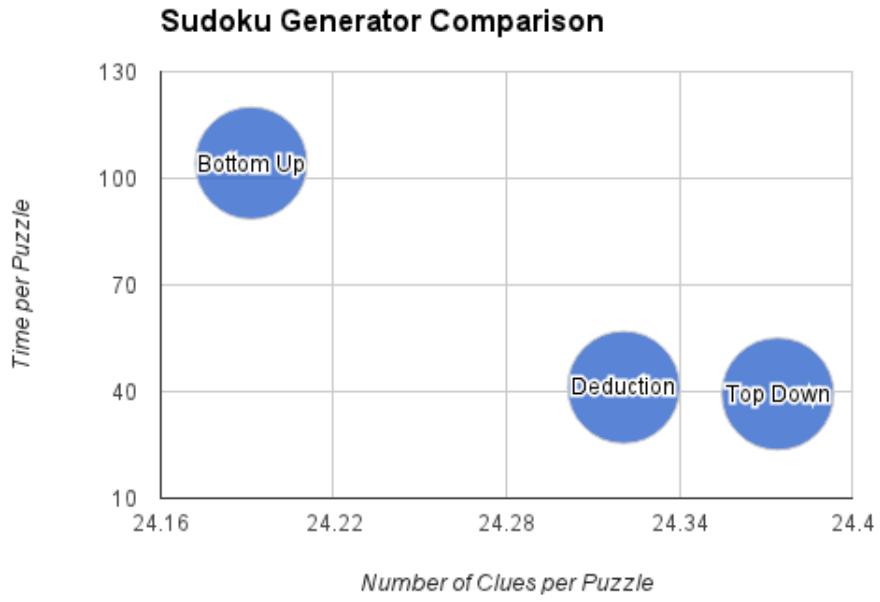

Figure 4: Results of generators.

References

[1] Donald Knuth, "Dancing Links" Stanford University, 2000

[2] Gary McGuire, Bastian Tugemanny, Gilles Civarioz, "There is no 16-Clue Sudoku", University College Dublin 2013

[3] Cornell University, "The Math Behind Sudoku", Retrieved April 2014 from http://www.math.cornell.edu/ mec/Summer2009/Mahmood/Home.html

[4] Peter Norvig, "Solving Every Sudoku Puzzle", Retrieved April 2014 from http://norvig.com/sudoku.html

[5] Richard Karp, "Reducibility Amoung Combinitorial Problems", University of California