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**
1.1 Terms . . . 1
1.2 Sudoku . . . 1

2 Sudoku Solving Strategies **2**
2.1 Backtracking . . . 2
2.2 Constraint Propagation . . . 3
2.3 Sudoku as an Exact Cover Problem . . . 4
2.3.1 Exact Cover . . . 4
2.3.2 Algorithm X . . . 4
2.3.3 Dancing Links . . . 4

3 Sudoku Generation **4**
3.1 Top Down Generation . . . 4
3.2 Bottom Up Generation . . . 4
3.3 Deduction Generation . . . 4

4 Results **4**

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 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 $candidates$
$candidates[cell] \leftarrow candidates[cell] - \{value\}$
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 $\begin{cases} otherCells \leftarrow \{s | s \in unit, value \in candidates[s]\} \\ \textbf{if } |otherCells| = 1 \\ \quad \textbf{then } \begin{cases} c \leftarrow q \in otherCells \\ \text{ASSIGN}(c, value) \end{cases} \end{cases}$

Algorithm 2.4: SEARCH($\emptyset$)

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

2.3.2 Algorithm X

2.3.3 Dancing Links

3 Sudoku Generation

3.1 Top Down Generation

3.2 Bottom Up Generation

3.3 Deduction Generation

4 Results

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