# CSC384 Final Project

# Random Sudoku Puzzle Synthesis and Categorization

Abe Ratnofsky, Aaron Boswell

## *Motivation and background*

**Our project is a Sudoku puzzle generator, and it works it two parts. The first part generates a complete puzzle, with every cell filled in. The second part takes a complete puzzle, and unassigns variables until the puzzle reaches a threshold difficulty value based on a heuristic we define. By working from a completed board, we can ensure our generated puzzle is solvable. In this writeup, we will explain our scheme for generating complete puzzles, our difficulty heuristic, and our unassignment procedure. Although our puzzle generation scheme is specially tuned to work well for Sudoku, it can be extended to many problems that can be modelled similarly. We provide further explanation of these extensions in our Discussion section.**

**We chose CSP for this problem because Sudoku is a textbook example of a constraint-oriented problem, and we wanted our methods to be broadly applicable to more general CSPs.**

**Our CSP model for Sudoku is built on top of the code used in Assignment 2, especially cspbase.py, with our own modifications where needed. The model includes a variable for each cell on the board, for a total of 81 variables each with domain of {1..9}, as well as 9-ary ALL-DIFF constraints for each row, column, and cage. This comes to a total of 27 constraints, with 9! satisfying tuples for each constraint. Although this formulation is simple, it is also inefficient for sparse boards. With empirical testing, we found that the time required to generate the model is reasonable if at least two variables are given for each constraint. This brings number of satisfying tuples for a given constraint down from 9! to at most 7!. Randomly generating two non-contradictory assignments for each constraint was not trivial, and we elaborate on our attempts later on[1].**

**This formalization is practically identical to the rules of Sudoku, where each row, column, and cage must contain all of {1..9}. For this to be possible, no pair of cells *c1* and *c2* sharing a row, column, or cell can be the same value. This holds for all pairs of cells in the board. To state this formally:**

**Given a Sudoku board *S*, and distinct cells *c1*, *c2* in *S*:**
**row(*c1*) = row(*c2*) → c1 != c2**
**column(*c1*) = column(*c2*) → c1 != c2**
**cage(*c1*) = cage(*c2*) → c1 != c2**

**Our 9-ary ALL-DIFF constraints prevent pairwise duplicates, but in less memory than the required binary ALL-DIFF constraints would.**

## *Evaluation*

**Rather than creating a CSP model to represent a complex puzzle, and using this model to solve the puzzle, we are working backwards from a simple CSP model to create puzzles. Because of this inverted approach, more traditional means of evaluation are not applicable for this project. Instead, we will evaluate our project by its ability to create puzzles in a large range of difficulties, and its speed in generating these puzzles. More information on the speed of Synthesis, and Solving will follow.**

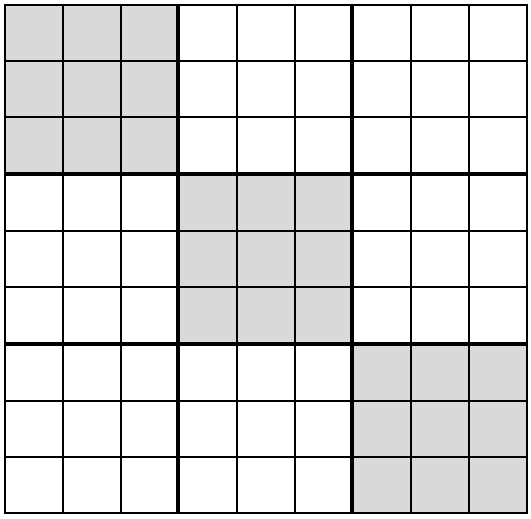
## *Method*

**As mentioned above, our puzzle generator works in two parts. The first part of the generator takes an additive approach, and creates a random completed board with a non-contradictory assignment for every cell. The second part of the generator takes a subtractive approach, and removes values strategically until a desired threshold metric is met. This section describes several additive strategies, an optimal additive strategy, the subtractive strategy, and the threshold metric.**

**The result of the additive component of the generator is a complete puzzle, in which each cell has an assigned value and no assignments contradict the constraints. Although there exists a trivial completed board (as shown in below), it is not immediately clear how to utilize this to create random completed boards.**

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

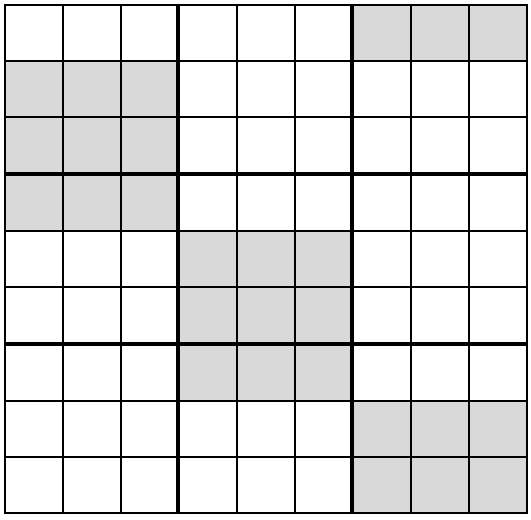
**Our goal was to write a random additive scheme without contradictions. At first, we tried using random assignments, checking that they had no immediate contradictions (backtracking if so), then solving once there were at least two assigned variables in every row, column, and cage. This process generated two solvable boards an hour, with a success rate of .763%. Our second strategy involved filling each diagonal cage (top-left, center-center, and bottom right) with a random permutation of {1..9}, and shifting some of the assignments to other cages, such that there were enough assignments for the model to be efficiently generated. This process is described in the following diagram. White cells represent unassigned variables, and gray cells represent assigned variables.**



***Step 1***

**Insert random permutations of {1..9} into each diagonal cage. These assignments are immediately consistent, but may yield eventual inconsistencies.**

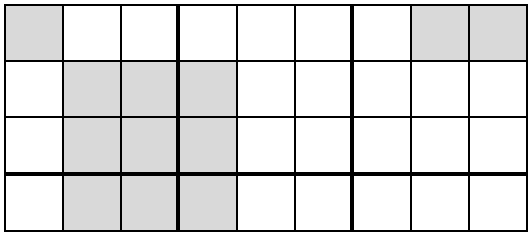
**From this board, the model cannot be generated efficiently, as the six unassigned cages do not have the minimum of two variable assignments required for efficient model building.**



***Step 2***

**Shift the top row of each cage down three rows, mod nine.**

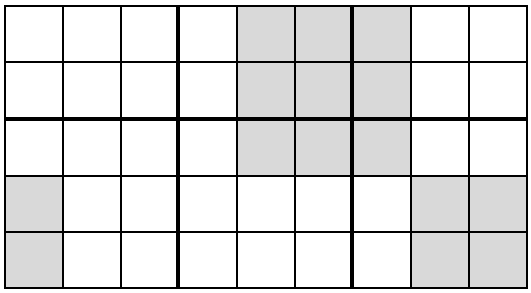
**Shifting rows in this manner preserves consistency.**



***Step 3***

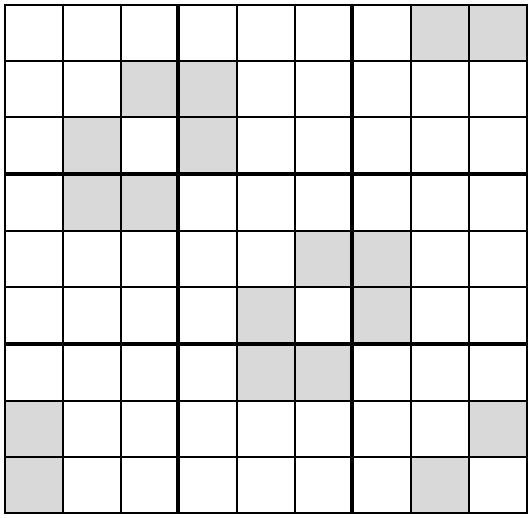
**Shift the left column of each cage right three rows, mod nine.**

**Shifting columns in this manner preserves consistency.**



**At this point, the model can be built, because each row, column, and cage has at least two assigned variables.**

**At this stage, the model can be built and the puzzle solved. Although the puzzle with these partial assignments has no contradictions, it not necessarily eventually consistent. There may be no solved puzzle with this set of partial assignments, so the solver must be run to verify that the partial assignments can be extended to form a consistent, complete board. We tested this for two hours, generating 619 boards, none of which were solvable. In this model, there are some constraints which have more than two assigned variables. Including these variables does not matter, as the model can still be built, but increases the likelihood that there will be an inconsistency. To resolve this, we unassign everything on the diagonal, leaving exactly two assigned variables in each constraint. We were astounded to find that this process generated ten solvable boards an hour, with a success rate of 19.23%.**



***Step 4***

**Unassign all variables on the diagonal, to reduce the likelihood of an inconsistency.**

**Each row, column, and cage still has two variables assigned, so the model can be built in a reasonable amount of time.**

**This strategy is effective, but inefficient. It requires the use of a backtracking solver, which is not ideal. The chief insight behind the optimal strategy is that shifting rows and columns, like in steps 2 and 3, does not impact consistency of a complete puzzle. As long as rows and columns are shifted in their respective groups of three, the resulting puzzle is still consistent. In fact, every complete puzzle can be reached by a series of transformations including row shuffling, column shuffling, and value remapping.**

**With this insight in mind, we can take any consistent, complete board and return a new, random one. Thus, this is an efficient solution to the first phase of puzzle generation. And we can always generate a solution puzzle, in less than a tenth of a second.**

**For the second phase of puzzle generation, we want to unassign variables in the complete board, until the board surpasses a threshold metric. This was done by randomly selecting a candidate assigned cell, unassigning it, and observing the resulting effect on the metric. An ideal metric is fast to check, and approximately reflects the difficulty of the puzzle. Although simple metrics, such as number of unassigned variables, may reflect one aspect of difficulty, they are shallow at best.**

**To efficiently and accurately describe the difficulty of a puzzle, we decided on a metric called *homogeneity*. Homogeneity is the pair of the least and most assignments in any single row, column or cage. The following puzzle is (1, 5)-homogenous, because there are at least 1 assignments (in the third column) and at most 5 assignments (in the top-left cage).**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

**With homogeneity, we can roughly characterize the number of places for a human to start solving the board, and the number of remaining assignments before the puzzle is solved. As a metric of difficulty, homogeneity does have its shortcomings, however. One example of such a shortcoming is lack of distributional knowledge. For example, all assignments could be of the same value, and homogeneity ignores this fact.**

**The following are example of (2,2), (2,9), (4,4), (4,9), (6,6) and (6,9) - homogenous puzzles we generated.**

**(2,2):**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 3 | | | | | | 9 |
| | 8 | | | 9 | | | | |
| | | | 4 | | | | 8 | |
| | 6 | | | | 5 | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | | | | | | 5 | | |
| | | | 9 | | | | 1 | |
| 8 | | 5 | | | | | | |
| | | | | | 9 | 1 | | |
| | | | | 6 | | | | 5 |

**(2,9):**

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

**(4,4):**

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

**(4,9):**

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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | | 4 | | 2 | | | | 3 |
| 7 | | 3 | | | | | 2 | 8 |
| 3 | 1 | | 9 | 7 | | 4 | | |
| | 4 | | | 3 | | | 7 | 2 |
| 2 | | | | | 4 | 3 | | 1 |

**(6,6):**

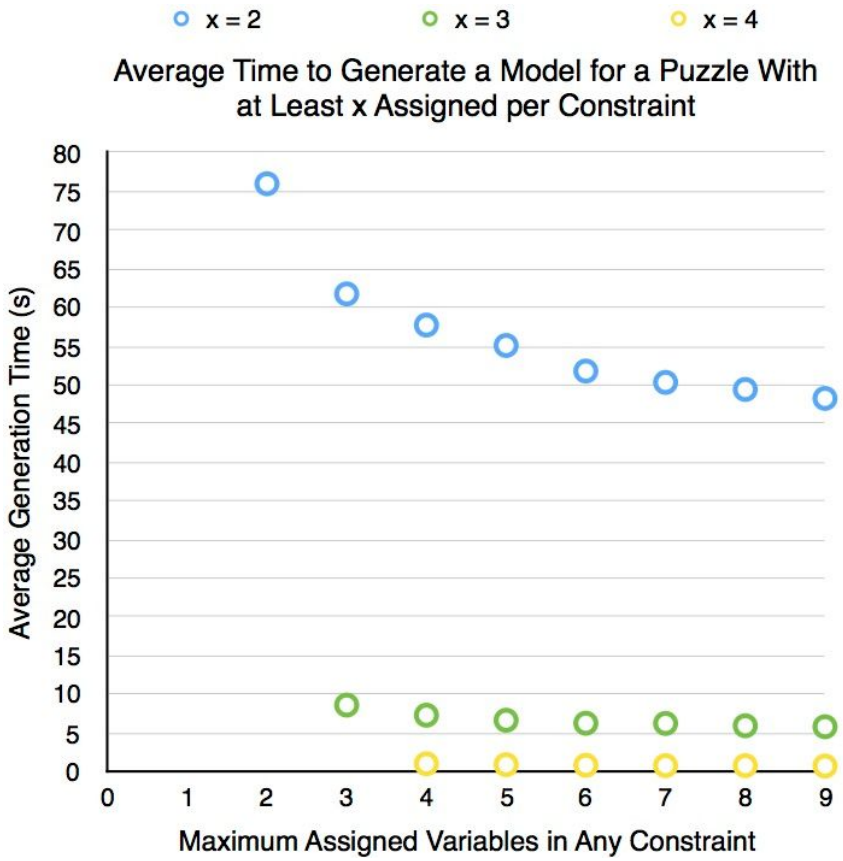
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 3 | 7 | 2 | 8 | | | 5 | 1 |
| 8 | | 5 | 3 | 1 | 4 | 7 | | |
| 1 | 5 | | | 7 | | 6 | 8 | 3 |
| 3 | | 1 | 4 | 6 | 8 | | 9 | |
| 5 | 7 | | 9 | | 2 | 1 | | 8 |
| | 8 | 9 | | 5 | | 3 | 6 | 2 |
| | | 4 | 8 | | 3 | 2 | 7 | 5 |
| 2 | 5 | 8 | 1 | 4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

**(6,9):**

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

**We collected timing-data by making 100 puzzles for each homogeneity and recording the time it took to generate a csp-model, and solve the model. This can be in source.zip/homogeneity-timings.xlsx**

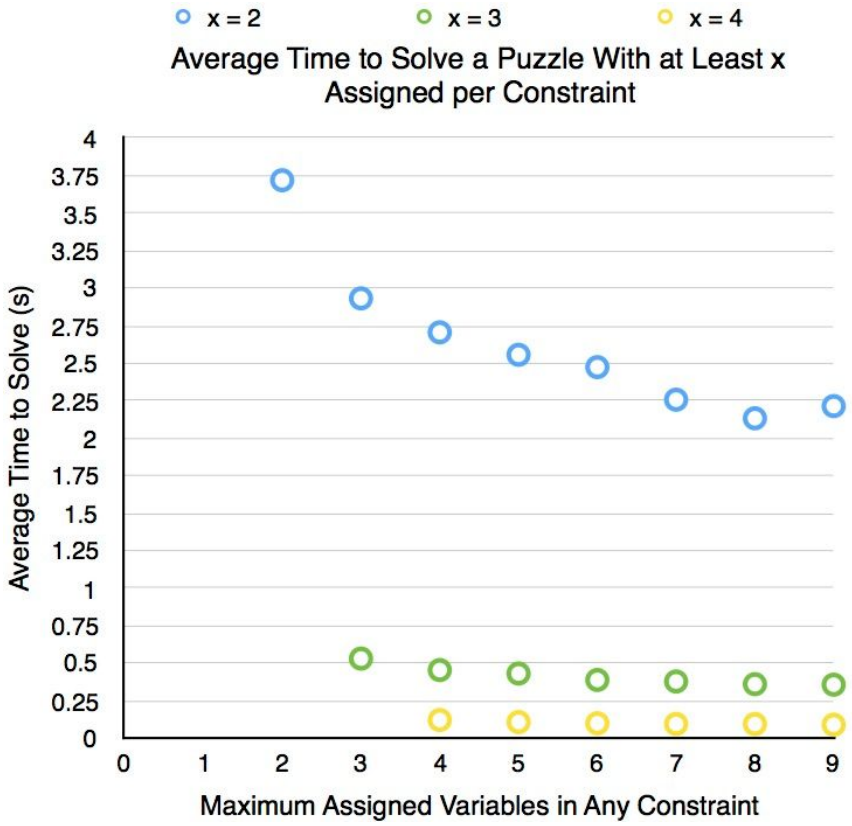
**The following are graphs of the average time it takes to generate models for (2-X), (3-X) and (4-X)-homogenous puzzles:**




**(2,X)-homogenous puzzles take 86 to 38 with an average of 56 seconds to generate a model. (3,X)-homogenous puzzles take 9.2 to 4.9 with an average of 6.6 seconds to generate a model. (4,X)-homogenous puzzles take 1.04 to 0.53 with an average of 0.86 seconds to generate a model.**

**This rapid decrease makes sense, because in (2,X)-homogenous puzzles, there will be at least one constraint with 7! satisfying tuples, and in (3,X)-homogenous puzzles, at least one constraint with 6!. In fact our data almost perfectly fits this drop off rate.**

**The following are graphs of the average time it takes to solve puzzles given a model for (2-X), (3-X) and (4-X)-homogenous puzzles:**




**(2,X)-homogenous puzzles take 4 to 2 with an average of 2.4 seconds to solve. (3,X) and (4,X)-homogenous puzzles always take less than one second to solve. This is consistent with our drop off rate of the number of satisfying tuples, for each model.**

**It was interesting to find that the minimum homogeneous number was what dictated the time it takes to generate, and solve a board. That is, (2,2)-homogeneous is more similar in ease to (2,9)-homogeneous, than (2,9)-homogeneous is to (3,3)-homogeneous**

## *Conclusion*

**In this project, we explored several strategies for generating Sudoku puzzles, and compared the merits of each. We found that all there exists a set of transformations with the property of consistency invariance, and that the set of all valid complete Sudoku puzzles is closed under these transformations. Shifting rows, shifting columns, and renumbering, can be used to transform any solved board into any other. From this point, there is a simple randomized backtracking algorithm, that when paired with a heuristic for difficulty, can be used to generated puzzles of high quality. Even simple heuristics, such as homogeneity, can be used to generate entertaining puzzles for humans.**

**One possible extension of the results of this project is application of this method to general CSPs. Although our additive approach was finely tuned for Sudoku puzzles in particular, there are heuristics for approximating partial random solutions for most CSPs, which can then be solved with backtracking algorithms. From a CSP with full assignments, puzzles can be generated with backtracking algorithms, and difficulty metrics and thresholds. Other puzzles that may work well for this approach include KenKen, Tenner, and others.**