Ease and Toil: Analyzing Sudoku

Seth B. Chadwick
Rachel M. Krieg
Christopher E. Granade

University of Alaska Fairbanks
Fairbanks, AK

Advisor: Orion S. Lawlor

Abstract

Sudoku is a logic puzzle in which the numbers 1 through 9 are arranged in a 9×9 matrix, subject to the constraint that there are no repeated numbers in any row, column, or designated 3×3 square.

In addition to being entertaining, Sudoku promises insight into computer science and mathematical modeling. Since Sudoku-solving is an NP-complete problem, algorithms to generate and solve puzzles may offer new approaches to a whole class of computational problems. Moreover, Sudoku construction is essentially an optimization problem.

We propose an algorithm to construct unique Sudoku puzzles with four levels of difficulty. We attempt to minimize the complexity of the algorithm while still maintaining separate difficulty levels and guaranteeing unique solutions.

To accomplish our objectives, we develop metrics to analyze the difficulty of a puzzle. By applying our metrics to published control puzzles with specified difficulty levels, we develop classification functions. We use the functions to ensure that our algorithm generates puzzles with difficulty levels analogous to those published. We also seek to measure and reduce the computational complexity of the generation and metric measurement algorithms.

Finally, we analyze and reduce the complexity involved in generating puzzles while maintaining the ability to choose the difficulty level of the puzzles generated. To do so, we implement a profiler and perform statistical hypothesis-testing to streamline the algorithm.

The UMAP Journal 29 (3) (2008) 363–379. ©Copyright 2008 by COMAP, Inc. All rights reserved. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice. Abstracting with credit is permitted, but copyrights for components of this work owned by others than COMAP must be honored. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior permission from COMAP.

Introduction

Goals

Our goal is to create an algorithm to produce Sudoku puzzles that:

- creates only valid puzzle instances (no contradictions, unique solution);
- can generate puzzles at any of four different difficulty levels;
- produces puzzles in a reasonable amount of time.

We explicitly do not aim to:

- "force" a particular solving method upon players,
- be the best available algorithm for the making exceedingly difficult puzzles, or
- impose symmetry requirements.

Rules of Sudoku

Sudoku is played on a 3×3 grid of blocks, each of which is a 3×3 grid of *cells*. Each cell contains a *value* from 1 through 9 or is empty. Given a partially-filled grid called a *puzzle*, the object is to place values in all empty cells so that the constraints (see below) are upheld. We impose the additional requirement that a puzzle admit exactly one solution.

The constraints are that in a solution, no row, column, or block may have two cells with the same value.

Terminology and Notation

Assignment A tuple (x, X) of a value and a cell. We say that X has the value x, X maps to x, or $X \mapsto x$.

Candidates Values that can be assigned to a square. The set of candidates for a cell X is denoted $X?$.

Cell A single square, which may contain a value between 1 and 9. We denote cells by uppercase italic serif letters: X, Y, Z.

Block One of the nine 3×3 squares in the puzzle. The boundaries of blocks are denoted by thicker lines on the puzzle's grid. No two blocks overlap (share common cells).

Grouping A set of cells in the same row, column or block. We represent groupings by uppercase boldface serif letters: $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$. We refer unambiguously to the row groupings $\mathbf{R}_i$, the column groupings $\mathbf{C}_j$ and the block groupings $\mathbf{B}_c$. The set of all groupings is $\mathbb{G}$.

Metric A function m from the set of valid puzzles to the reals.

Puzzle A 9×9 matrix of cells with at least one empty and at least one filled cell. We impose the additional requirement that a puzzle have exactly one solution. We denote puzzles by boldface capital serif letters: $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{R}$. We refer to cells belonging to a puzzle: $X \in \mathbf{P}$.

Representative of a block The upper-left cell in the block.

Restrictions In some cases, it is more straightforward to discuss which values a cell cannot have than to discuss the candidates. The restrictions set $X!$ for a cell X is $\mathbb{V} \backslash X?$.

Rule An algorithm that accepts a puzzle $\mathbf{P}$ and produces either a puzzle $\mathbf{P}'$ representing strictly more information (more restrictions have been added via logical inference or cells have been filled in) or some value that indicates that the rule failed to advance the puzzle towards a solution.

Solution A set of assignments to all cells in a puzzle such that all groupings have exactly one cell assigned to each value.

Value A symbol that may be assigned to a cell. All puzzles here use the traditional numeric value set $\mathbb{V} = \{1, \ldots, 9\}$. A value is denoted by a lowercase sans serif letter: x, y, z.

Indexing

By convention, all indices start with zero for the first cell or block.

c	:	block number
k	:	cell number within a block
i, j	:	row number, column number
i', j'	:	representative row number, column number

These indicies are related by the following functions:

$$c(i,j) = \frac{j}{3} + 3\left\lfloor \frac{i}{3} \right\rfloor,$$

$$i(c,k) = 3\left\lfloor \frac{c}{3} \right\rfloor + \left\lfloor \frac{k}{3} \right\rfloor, \qquad j(c,k) = (c \bmod 3) \cdot 3 + (k \bmod 3),$$

$$i'(c) = 3\left\lfloor \frac{c}{3} \right\rfloor, \qquad j'(c) = (c \bmod 3) \cdot 3,$$

$$i'(i) = 3\left\lfloor \frac{i}{3} \right\rfloor, \qquad j'(j) = 3\left\lfloor \frac{j}{3} \right\rfloor.$$

Figure 1 demonstrates how the rows, columns and blocks are indexed, as well as the idea of a block representative. In the third Sudoku grid, the representatives for each block are denoted with an "r".

0								
1								
2								
3								
4								
5								
6								
7								
8								

0	1	2	3	4	5	6	7	8

r			r			r		
	0			1			2	
r			r			r		
	3			4			5	
r			r			r		
	6			7			8	

Figure 1. Demonstration of indexing schemes.

Rules of Sudoku

We state formally the rules of Sudoku that restrict allowable assignments, using the notation developed thus far:

$$(\forall \mathbf{G} \in \mathbb{G}\, \forall X \in G) \qquad X \mapsto \mathsf{v} \Rightarrow \nexists Y \in \mathbf{G} : Y \mapsto \mathsf{v}.$$

Applying this formalism will allow us to make strong claims about solving techniques.

Example Puzzles

The rules alone do not explain what a Sudoku puzzle looks like, and so we have included a few examples of well-crafted Sudoku puzzles. **Figure 2** shows a puzzle ranked as "Easy" by WebSudoku [Greenspan and Lee n.d.].

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

Figure 2. Puzzle generated by WebSudoku (ranked as "Easy").

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

Figure 3. Example of the Naked Pair rule.

Background

Common Solving Techniques

In the techniques below, we assume that the puzzle has a unique solution. These techniques and examples are adapted from Taylor [2008] and Astraware Limited [2005].

Naked Pair

If, in a single row, column or block grouping $\mathbf{A}$, there are two cells X and Y each having the same pair of candidates $X? = Y? = \{p, q\}$, then those candidates can be eliminated from all other cells in $\mathbf{A}$. To see that we can do this, assume for the sake of contradiction that there exists some cell $Z \in \mathbf{A}$ such that $Z \mapsto \mathsf{p}$, then $X \not\mapsto \mathsf{p}$, which implies that $X \mapsto \mathsf{q}$. This in turn means that $Y \not\mapsto \mathsf{q}$, but we have from $Z \mapsto \mathsf{p}$ that $Y \not\mapsto \mathsf{p}$, leaving $Y? = \varnothing$. Since the puzzle has a solution, this is a contradiction, and $Z \not\mapsto \mathsf{p}$.

As an example of this arrangement is shown in **Figure 3**. The cells marked X and Y satisfy $X? = Y? = \{2, 8\}$, so we can remove both 2 and 8 from all other cells in $\mathbf{R}_8$. That is, $\forall Z \in (\mathbf{R}_8 \setminus \{X, Y\}) : 2, 8 \notin Z?$.

Naked Triplet

This rule is analogous to the Naked Pair rule but involves three cells instead of two. Let $\mathbf{A}$ be some grouping (row, column or block), and let $X, Y, Z \in \mathbf{A}$ such that the candidates for X, Y and Z are drawn from $\{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$. Then, by exhaustion, there is a one-to-one set of assignments from $\{X, Y, Z\}$ to $\{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$. Therefore, no other cell in $\mathbf{A}$ may map to a value in $\{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$.

An example is in **Figure 4**. We have marked the cells $\{X, Y, Z\}$ accordingly and consider only block 8. In this puzzle, $X? = \{3, 7\}$, $Y? = \{1, 3, 7\}$ and $Z? = \{1, 3\}$. Therefore, we must assign 1, 3, and 7 to these cells, and can remove them as candidates for cells marked with an asterisk.

Hidden Pair

Informally, this rule is conjugate to the Naked Pair rule. We again consider a single grouping $\mathbf{A}$ and two cells $X, Y \in \mathbf{A}$, but the condition is that there exist two values u and v such that at least one of $\{\mathsf{u}, \mathsf{v}\}$ is in each of $X?$ and $Y?$, but such that for any cell $Q \in (\mathbf{A} \setminus \{X, Y\})$, $\mathsf{u}, \mathsf{v} \notin Q?$. Thus, since $\mathbf{A}$ must contain a cell with each of the values, we can force $X?, Y? \subseteq \{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$.

An example of this is given in **Figure 5**. We focus on the grouping $\mathbf{R}_8$, and label X and Y in the puzzle. Since X and Y are the only cells in $\mathbf{R}_8$ whose candidate lists contain 1 and 7, we can eliminate all other candidates for these cells.

		4				9	1	8
6	5	2	8				2	
8		9	1	3	2			5
5	1	2						4
	9		4	7	5	1	6	2
6	7	4	2	8	1	5	3	9
	4		6	2		X	5	Y
	3	5			8	2	*	6
2	6	7				*	*	Z

Figure 4. Example of the Naked Triplet rule.

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

Figure 5. Example of the Hidden Pair rule.

Hidden Triplet

As with the Naked Pair rule, we can extend the Hidden Pair rule to apply to three cells. Let $\mathbf{A}$ be a grouping, and let $X, Y, Z \in \mathbf{A}$ be cells such that at least one of $\{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$ is in each of $X?$, $Y?$ and $Z?$ for some values t, u and v. Then, if for any other cell $Q \in (\mathbf{A} \setminus \{X, Y, Z\})$, $\mathsf{t}, \mathsf{u}, \mathsf{v} \notin Q?$, we claim that we can force $X?, Y?, Z? \subseteq \{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$.

An example is shown in **Figure 6**, where in the grouping $\mathbf{R}_5$, only the cells marked X, Y and Z can take on the values of 1, 2 and 7. We would thus conclude that any candidate of X, Y or Z that is not either 1, 2, or 7 may be eliminated.

8	9	5		4	X	6	2	3
1	6	3	2			5	4	7
2	7	4		5		1	9	8
	8		4		Y			5
	5	2		3		4		1
4	3				5		6	2
9	1	7	5	6		2		4
3	2	8			4	7	5	6
5	4	6			Z		1	9

Figure 6. Example of the Hidden Triplet rule.

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

Figure 7. Example of the Multi-Line rule.

Multi-Line

We develop this technique for columns, but it works for rows with trivial modifications. Consider three blocks $\mathbf{B}_a$, $\mathbf{B}_b$, and $\mathbf{B}_c$ that all intersect the columns $\mathbf{C}_x$, $\mathbf{C}_y$, and $\mathbf{C}_z$. If for some value v, there exists at least one cell X in each of $\mathbf{C}_x$ and $\mathbf{C}_y$ such that $\mathsf{v} \in X?$ but that there exists no such $X \in \mathbf{C}_z$, then we know that the cell $V \in \mathbf{B}_c$ such that $V \mapsto \mathsf{v}$ satisfies $V \in \mathbf{C}_z$. Were this not the case, then we would not be able to satisfy the requirements for $\mathbf{B}_a$ and $\mathbf{B}_b$.

An example of this rule is shown in **Figure 7**. In that figure, cells that we are interested in, and for which 5 is a candidate, are marked with an asterisk. We will be letting $a = 0$, $b = 6$, $c = 3$, $x = 0$, $y = 1$ and $z = 2$. Then, note that all of the asterisks for blocks 0 and 6 are located in the first two columns. Thus, in order to satisfy the constraint that a 5 appear in each of these blocks, block 0 must have a 5 in either column 0 or 1, while block 6 must have a 5 in the other column. This leaves only column 2 open for block 3, and so we can remove 5 from the candidate lists for all of the cells in column 0 and block 3.

Previous Work

Sudoku Explainer

The Sudoku Explainer application [Juillerat 2007] generates difficulty values for a puzzle by trying each in a battery of solving rules until the puzzle is solved, then finding out which rule had the highest difficulty value. These values are assigned arbitrarily in the application.

QQWing

The QQWing application [Ostermiller 2007] is an efficient puzzle generator that makes no attempt to analyze the difficulty of generated puzzles beyond categorizing them into one of four categories. QQWing has the unique feature of being able to print out step-by-step guides for solving given puzzles.

GNOME Sudoku

Included with the GNOME Desktop Environment, GNOME Sudoku [Hinkle 2006] is a Python application for playing the game; since it is distributed in source form, one can directly call the generator routines.

The application assigns a difficulty value between 0 and 1 each puzzle. Rather than tuning the generator to requests, it simply regenerates any puzzle outside of a requested difficulty range. Hence, it is not a useful model of how to write a tunable generator, but it is very helpful for quickly generating large numbers of control puzzles. We used a small Python script to extract the puzzles.

Metric Design

Overview

The metric that we designed to test the difficulty of puzzles was the *weighted normalized ease function* (WNEF). It is was based on the calculation of a *normalized choice histogram*.

As the first step in calculating this metric, we count the number of choices for each empty cell's value and compile these values into a histogram with nine bins. Finally, we multiply these elements by empirically-determined weights and sum the result to obtain the WNEF.

Assumptions

The design of the WNEF metric is predicated on two assumptions:

- There exists some objective standard by which we can rank puzzles in order of difficulty.
- The difficulty of a puzzle is roughly proportional to the number of choices that a solver can make without directly contradicting any basic constraint.

In addition, in testing and analyzing this metric, we included a third assumption:

- The difficulties of individual puzzles are independently and identically distributed over each source.

Mathematical Basis for WNEF

We start by defining the choice function of a cell $c(X)$:

$$c(X) = |X?|,$$

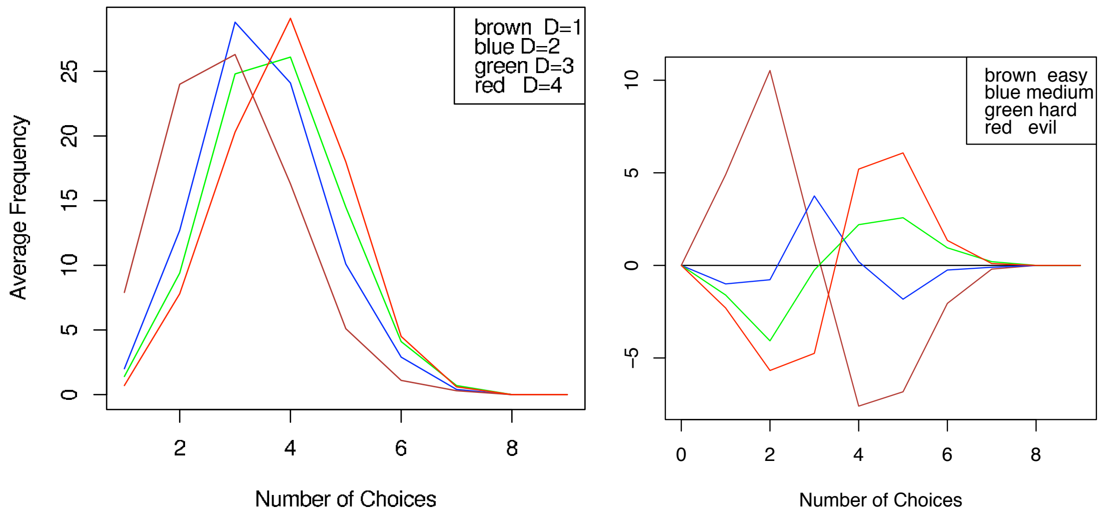
the number of choices available. This function is useful only for empty cells. We denote all empty cells in a puzzle $\mathbf{P}$ by $\mathbf{P}$

$$E(\mathbf{P}) = \{X \in P \mid \forall \mathsf{v} \in \mathbb{V} : X \not\mapsto \mathsf{v}\}.$$

By binning each empty cell based on the choice function, we obtain the choice histogram $\vec{c}(\mathbf{P})$ of a puzzle $\mathbf{P}$.

$$c_n(\mathbf{P}) = |\{X \in \mathbf{P} \mid c(X) = n\}| = |\{X \in \mathbf{P} \mid |X?| = n\}|. \qquad \textbf{(1)}$$

Examples of histograms with and without the mean control histogram (obtained from control puzzles) are in **Figures 8a** and **b**.



(a) Original histograms. (b) Histograms with mean removed.

Figure 8. Examples of choice histograms.

From the histogram, we obtain the value wef $(\mathbf{P})$ of the (unnormalized) weighted ease function by convoluting the histogram with a weight function $w(n)$:

$$\text{wef}(\mathbf{P}) = \sum_{n=1}^{9} w(n) \cdot c_n(\mathbf{P}),$$

where $c_n(\mathbf{P})$ is the nth value in the histogram $\vec{c}(\mathbf{P})$. This function, however, has the absurd trait that removing information from a puzzle results in more empty cells, which in turn causes the function to strictly increase. We therefore calculate the weighted and *normalized* ease function:

$$\text{wnef}(\mathbf{P}) = \frac{\text{wef}(\mathbf{P})}{w(1) \cdot |E(\mathbf{P})|}.$$

This calculates the ratio of the weighted ease function to the maximum value that it can have (which is when all empty cells are completely determined but have not been filled in). We experimented with three different weight functions before deciding upon the *exponential weight function*.

Complexity

The complexity of finding the WNEF is the same as for finding the choice histogram (normalized or not). To do that, we need to find the direct restrictions on each cell by examining the row, column, and block in which it is located. Doing so in the least efficient way that is still reasonable, we look at each of the 8 other cells in those three groupings, even though some are checked multiple times, resulting in 24 comparisons per cell. For a total of 81 cells, this results in 1,944 comparisons. Of course, we check only when the cell is empty; so for any puzzle, the number of comparisons is fewer. Hence, finding the WNEF is a constant-time operation.

Metric Calibration and Testing

Control Puzzle Sources

In calibrating and testing the metrics, we used published puzzles from several sources with levels of difficulty as labeled by their authors, including:

- WebSudoku [Greenspan and Lee n.d.]: 10 each of Easy, Medium, Hard, and Evil puzzles
- Games World of Sudoku [Ganni 2008]: 10 each of puzzles with 1, 2, 3, and 4 stars
- GNOME Sudoku [Hinkle 2005]: 2000 Hard puzzles.
- "top2365" from Stertenbrink [2005]: 2365 Evil puzzles.

Testing Method

Defining Difficulty Ranges

We separated our control puzzles into four broad ranges of difficulty: easy, medium, hard, and evil, denoted by indices 1, 2, 3 and 4.

Information Collection

We calculated the metrics for each control puzzle. The information collected included:

- $|E(\mathbf{P}_i)|$, the total number of empty cells in $\mathbf{P}_i$;
- $C(\mathbf{P}_i) = \sum_{X \in \mathbf{P}_i} X?$, the number of possible choices for all cells; and
- the choice histogram $\vec{c}$ defined in **1**.

Statistical Analysis of Control Puzzles

The number of empty cells and the number of total choices lack any association with difficulty. In easier puzzles, there seem to be more cells with fewer choices than in more difficult puzzles (**Figure 8**).

We found a negative correlation between the difficulty level and WNEF for the control puzzles (lowest curve in **Figure 9**). This leads us to consider the mean WNEF for control puzzles of difficulty d, $d = 1, 2, 3, 4$. We tested the hypotheses that this mean is different for d and $d + 1$, for $d = 1, 2, 3$, using the mean WNEF, its standard deviation, and the t-test for difference of means. We concluded that the WNEF produces distinct difficulty levels, at significance level $\alpha = 0.0025$, for each of $d = 1, 2, 3$.

Choice of Weight Function.

We tried three different weighting functions for specifying WNEF values: exponential, quadratic and linear.

$$\begin{aligned} w_{\text{exp}}(n) &= 2^{9-n}, \\ w_{\text{sq}}(n) &= (10-n)^2, \\ w_{\text{lin}}(n) &= (10-n), \end{aligned}$$

where n is the number of choices for a cell. For all three, the graphs of WNEF vs. difficulty all looked very similar (**Figure 9**).

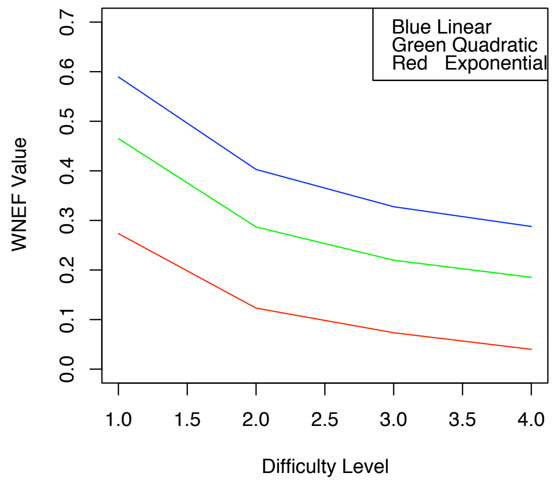


Figure 9. WNEF vs. difficulty level, for various weighting functions.

We concluded that we could choose any of the three weighting functions. We arbitrarily chose w_{exp}.

Generator Algorithm

Overview

The generator algorithm works by first creating a valid solved Sudoku board, then "punching holes" in the puzzle by applying a *mask*. The solved puzzle is created via an efficient backtracking algorithm, and the masking is performed via application of various *strategies*. A strategy is simply an algorithm that outputs cell locations to attempt to remove, based on some goal. After a cell entry is removed, the puzzle is checked to ensure that it still has a unique solution. If this test succeeds, another round is started. Otherwise, the board's mask is reverted, and a different strategy is consulted. Once all strategies have been exhausted, we do a final "cleanup" phase in which additional cells are removed systematically, then return the completed puzzle. For harder difficulties, we introduce *annealing*.

Completed Puzzle Generation

Completed puzzles are generated via backtracking. A solution is generated via some systematic method until a contradiction is found. At this point the algorithm reverts back to a previous state and attempts to solve the problem via a slightly different method. All methods are tried in a systematic manner. If a valid solution is found, then we are done.

Backtracking can be slow. To gain efficiency, we take the 2D Sudoku board and view it as a 1D list of rows. The problem reduces to filling rows with values; if we cannot, we backtrack to the previous row. We are finished if we complete the last row.

This recasting of the problem also simplifies the constraints; we need concern ourselves only with the values in each column and in the three clusters (or blocks) that the current row intersects. These constraints can be maintained by updating them each time a new value is added to a row.

Cell Removal

To change a puzzle to one that is entertaining to solve, we perform a series of removals that we call *masking*. One or more cells are removed from the puzzle (*masked out* of the puzzle), and then the puzzle is checked to ensure that it still has a unique solution. If this is not the case, the masking action is undone (or the cells are added back into the puzzle).

In *random masking*, every cell is masked in turn but in random order. Every cell that can be removed is, resulting in a minimal puzzle. This procedure is very fast and has potential to create any possible minimal puzzle, though with differing probability.

Tuned masking is slower and cannot create a puzzle any more difficult than random masking can. The idea is to tune the masking to increase the probability that a given type of puzzle is generated.

To create a board with a given WNEF, we apply strategies to reduce the WNEF. If we reach a minimum WNEF that is not low enough, we use a method from mathematical optimization, *simulated annealing*: We add some number of values back into the board and then optimize from there, in hope that doing so will result in a lower minimum. State-saving lets us to revert to the board with the lowest WNEF. Annealing allowed us to produce puzzles with lower WNEF values than we could have without it.

Uniqueness Testing

To ensure we generate boards with only one solution, we must test if this condition is met.

The fast solution uses Hidden Single and Naked Single: A cell with only one possible value can be filled in with that value, and any cell that is the only cell in some reference frame (such as its cluster, row, or column) with the potential of some value can be filed in with that value. These two logic processes are performed on a board until either the board is solved (indicating a unique solution) or no logic applies (which indicates the need to guess and hence a high probability that the board has multiple solutions). This test can produce false negatives, rejecting a board that has a unique solution.

The slow solution is to try every valid value in some cell and ask if the board is unique for each. If more than one value produces a unique result, the board has more then one solution. This solution calls itself recursively to determine the uniqueness of the board with the added values. The advantage of this approach is that it is completely accurate, and will not result in false negatives.

We used a hybrid method. It proceeds with the slow solution when the fast one fails. A further optimization restricts the number of times that the slow solution is applied to a board. This is similar to saying, “If we have to guess more then twice, we reject the board.”

Complexity Analysis

Parameterization

We measure the time complexity t for generating a puzzle of difficulty d $t\left(d\right) = f\left(d\right) \cdot t_0$, where f is a function that we will find through our analysis and t_0 is the time complexity for generating a puzzle randomly.

Complexity of Completed Puzzle Generation

The puzzle generation algorithm works on each line of the Sudoku and potentially does so over all possible boards. In the worst case, we have the 9 possible values times the 9 cells in a line times 9 shifts, all raised to the 9 lines power, or $\left(9 \times 9 \times 9\right)^9 \approx 5.8 \times 10^{25}$. While this is a constant, it is prohibitively large. The best case is 81, where all values work on the first try.

However, in the average case, we not only do not cover all possible values, or cover all possible shifts, but we also do not recurse all possible times. So let us keep the same value for the complexity of generating a line (we have to try all 9 values, in all 9 cells, and perform all 9 shifts) but let us assume that we only do this once per line, getting $9^4 = 6561$; the actual value may be less or slightly more. We have a very high worst case but a very reasonable average case. In practice, the algorithm runs in time that is negligible in comparison to the masking algorithms.

Complexity of Uniqueness Testing and Random Filling

In the worst case, the "fast" uniqueness algorithm examines each of the 81 cells and compares it to each of the others. Thus, the uniqueness test can be completed in $81 \times 81 = 6,561$ operations. For the hybrid algorithm with brute-force searching, in the worst case we perform the fast test for each allowed guess plus one more time before making a guess at all. Therefore, the hybrid uniqueness testing algorithm has complexity linear in the number of allowed guesses.

The random filling algorithm does not allow any guessing when it calls the uniqueness algorithm and it performs the uniqueness test exactly once per cell. So it performs exactly $81^3 = 531,441$ comparisons.

Profiling Method

To collect empirical data on the complexity of puzzle generation, we implemented a profiling utility class in PHP. We remove dependencies on our particular hardware by considering only the normalized time $\hat{t} = t/t_0$, where t_0 is the mean running time for the random fill generator.

WNEF vs. Running Time

For the full generator algorithm, we can no longer make deterministic arguments about complexity, since there is a dependency on random variables. Hence, we rely on our profiler to gather empirical data about the complexity of generating puzzles. In particular, **Figure 10** shows the normalized running time required to generate a puzzle as a function of the obtained WNEF after annealing is applied. To show detail, we plot the normalized time on a logarithmic scale (base 2).

This plot suggests that even for the most difficult puzzles that our algorithm generates, the running time is no worse than about 20 times that of the random case. Also, generating easy puzzles can actually be faster than generating via random filling.

Testing

WNEF as a Function of Design Choices

The generator algorithm is fairly generic. We thus need some empirical way to specify parameters, such as how many times to allow cell removal

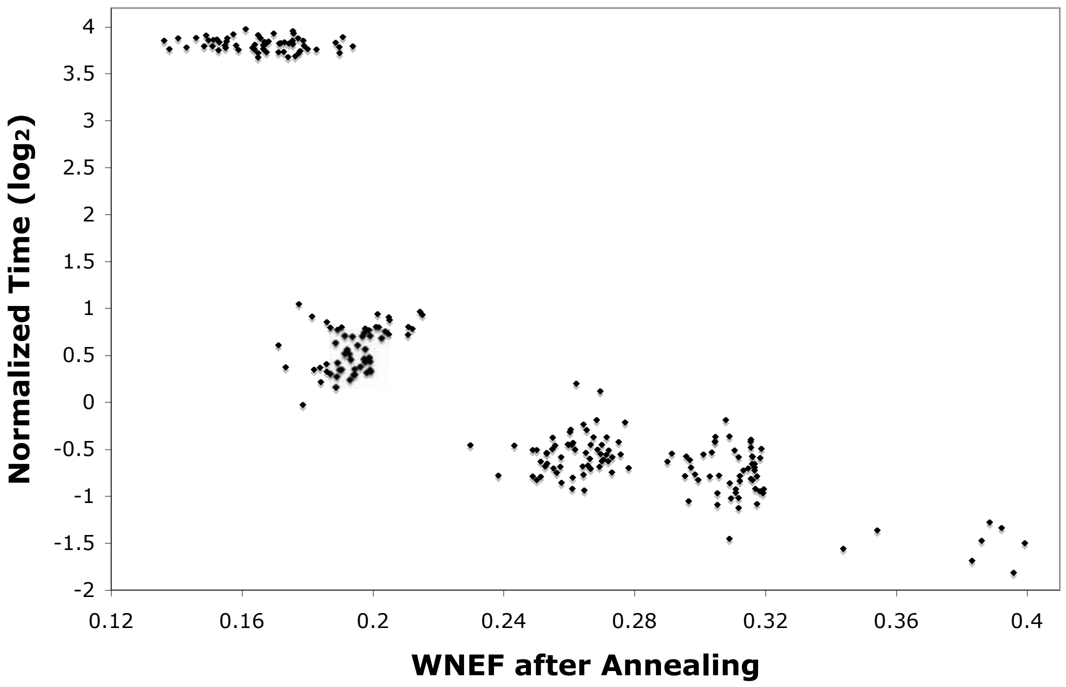


Figure 10. Log_2 of running time vs. WNEF.

to fail before concluding that the puzzle is minimal. We thus plotted the number of failures that we permitted vs. the WNEF produced, shown in **Figure 11**. This plot shows us both that we need to allow only a very small number of failures to enjoy small WNEF values, and that annealing reduces the value still further, even in the low-failure scenario.

Hypothesis Testing

Effectiveness of Annealing To show that the annealing resulted in lower WNEF values, and was thus useful, we tested the hypothesis that it was effective vs. the null hypothesis that it was not, using the mean WNEF for puzzles produced with annealing and the mean WNEF for those produced without it. A t-test at level of $\alpha = 0.0005$ concluded that annealing lowered the WNEF values.

Distinctness of Difficulty Levels To determine whether the difficulty levels of our puzzle generator are unique, we performed a t-test using the mean WNEF of puzzles produced by our generator algorithm with d as the target difficulty vs. those produced with target $d+1$. For $d = 1, 2, 3$, concluded that the difficulty levels are indeed different.

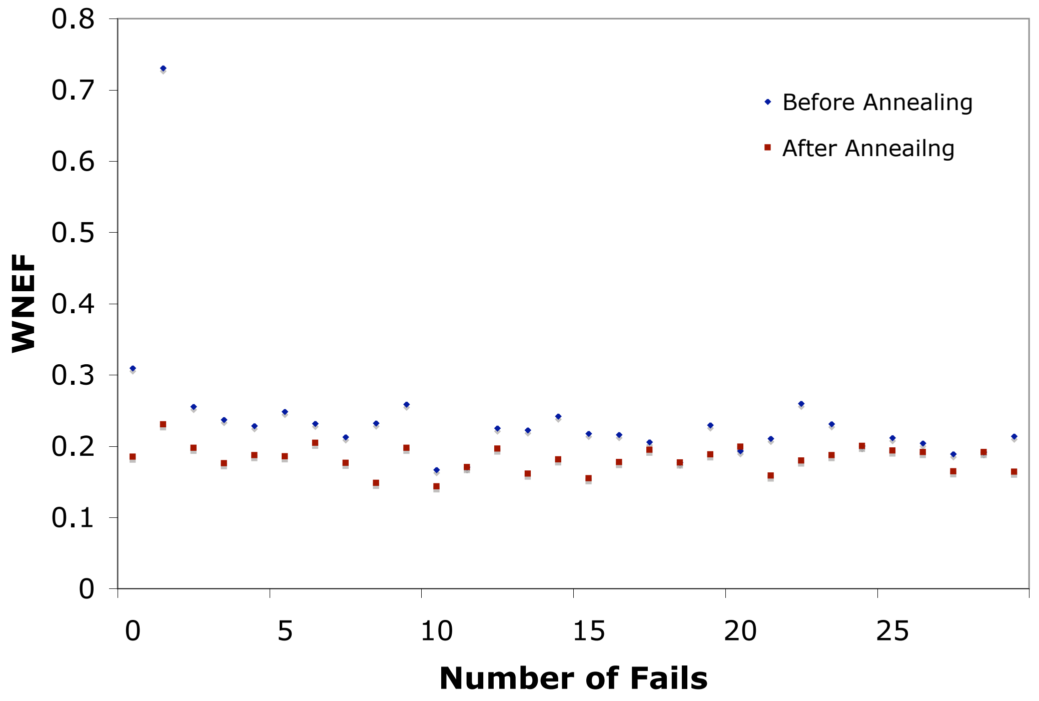


Figure 11. WNEF as a function of allowed failures.

Strengths and Weaknesses

It is possible to increase the difficulty of a puzzle without affecting its WNEF, by violating the assumption that all choices present similar difficulty to solvers. In particular, puzzles created with more-esoteric solving techniques, such as Swordfish and XY-Wing, can be crafted so that their WNEF is higher than easier puzzles. Thus, there is a limited regime over which the WNEF metric is useful.

On the other hand, the WNEF offers the notable advantage of being quick to calculate and constant for any puzzle difficulty, allowing us to make frequent evaluations of the WNEF while tuning puzzles.

Our generator algorithm seems to have difficulty generating puzzles with a WNEF lower than some floor, hence our decision to make our Evil difficulty level somewhat easier than published puzzles. The reason is that our tuning algorithm is still inherently a random algorithm and the probability of randomly creating a puzzle with a small WNEF value is very low.

The generator algorithm creates difficult puzzles quickly. Its method is similar to randomly generating puzzles until one of the desired difficulty is found (a method that is subject to the same disadvantage as ours), except that we can do so without generating more than one puzzle. We can generate a difficult puzzle in less time than it would take to generate multiple puzzles at random and discard the easy ones.

Conclusions

We introduce a metric, the weighted normalized ease function (WNEF), to estimate the difficulty of a Sudoku puzzle. We base this metric on the observation that the essential difficulty encountered in solving comes from ambiguities that must be resolved. The metric represents how this ambiguity is distributed across the puzzle.

Using data from control puzzles, we find that the WNEF shows a strong negative association with the level of difficulty (the harder the puzzle, the lower the WNEF). WNEF values of different difficulty levels are distinct. The choice of weighting function does not change this association.

We also designed an algorithm that employs these insights to create puzzles of selectable difficulty. It works by employing back-tracking and annealing to optimize the WNEF metric toward a desired level. Annealing leads to better results, and that the generator successfully produces puzzles falling into desired ranges of difficulty.

References

Aaronson, Lauren. 2006. Sudoku science. `http://spectrum.ieee.org/feb06/2809` .

Astraware Limited. 2005. Techniques for solving Sudoku. `http://www.sudokuoftheday.com/pages/techniques-overview.php` .

Ganni, J. 2008. *Games World of Sudoku* (April 2008). Blue Bell, PA: Kappa Publishing Group.

Greenspan, Gideon, and Rachel Lee. n.d. Web Sudoku. `http://www.websudoku.com` .

Hinkle, Tom. 2006. GNOME Sudoku. `http://gnome-sudoku.sourceforge.net/` .

Juillerat, N. 2007. Sudoku Explainer. `http://diuf.unifr.ch/people/juillera/Sudoku/Sudoku.html` .

Nikoli Puzzles. n.d. Sudoku tutorials. `http://www.nikoli.co.jp/en/puzzles/sudoku` .

Ostermiller, Stephen. 2007. QQwing—Sudoku generator and solver. `http://ostermiller.org/qqwing` .

Stertenbrink, Guenter. 2005. Sudoku. `magictour.free.fr/sudoku.htm` .

Taylor, A.M. 2008. *Dell Sudoku Challenge* (Spring 2008). New York: Dell Magazines.

Pp. 381–394 can be found on the *Tools for Teaching 2008* CD-ROM.