

更多免费学习资料，请关注微博：学神资料站

更多学习资料，请关注淘宝店铺：学神资料站 <https://shop156050543.taobao.com/>

Ease and Toil: Analyzing Sudoku

February 18, 2008

Look at any current magazine, newspaper, computer game package or handheld gaming device and you likely find sudoku, the latest puzzle game sweeping the nation. Sudoku is a number-based 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 valuable insight into computer science and mathematical modeling. In particular, since sudoku solving is an NP-Complete problem, algorithms to generate and solve sudoku puzzles may offer new approaches to a whole class of computational problems. Moreover, we can further explore mathematical modeling techniques through generating puzzles since sudoku construction is essentially an optimization problem.

The purpose of this paper is to propose an algorithm that may be used to construct unique sudoku puzzles with four different levels of difficulty. We attempted to minimize the complexity of the algorithm while still maintaining separate difficulty levels and guaranteeing unique solutions.

In order to accomplish our objectives, we developed metrics with which to analyze the difficulty of a given puzzle. By applying our metrics to published control puzzles with specific difficulty levels we were able to develop classification functions for specific difficulty ratings. We then used the functions we developed to ensure that our algorithm generated puzzles with difficulty levels analogous to those currently published. We also sought out to measure and reduce the computational complexity of the generation and metric measurement algorithms.

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

Contents

Contents		
1	Introduction	3
1.1	Statement of Problem	3
1.2	Relevance of Sudoku	3
1.3	Goals	3
1.4	Rules of Sudoku	3
1.5	Terminology and Notation	3
1.6	Indexing	4
1.7	Formal Rules of Sudoku	5
1.8	Example Puzzles	5
2	Background	5
2.1	Common Solving Techniques	5
2.1.1	Naked Pair	5
2.1.2	Naked Triplet	5
2.1.3	Hidden Pair	6
2.1.4	Hidden Triplet	6
2.1.5	Multi-Line	6
2.2	Previous Works	7
2.2.1	SudokuExplainer	7
2.2.2	QQWing	7
2.2.3	GNOME Sudoku	7
3	Metric Design	10
3.1	Overview	10
3.2	Assumptions	10
3.3	Mathematical Basis for WNEF	10
3.3.1	Complexity	10
4	Metric Calibration and Testing	11
4.1	Control Puzzle Sources	11
4.2	Testing Method	12
4.2.1	Defining Difficulty Ranges	12
4.2.2	Information Collection	12
4.2.3	Statistical Analysis of Control Puzzles	12
4.3	Choice of Weight Function.	12
5	Generator Algorithm	12
5.1	Overview	12
5.2	Detailed Description	14
5.2.1	Completed Puzzle Generation	14
5.2.2	Cell Removal	14
5.2.3	Uniqueness Testing	15
5.3	Pseudocode	15
5.3.1	Completed Board Generation	15
5.3.2	Random Masking	16
5.3.3	Tuned Masking	17
5.3.4	Uniqueness Testing	17
5.4	Complexity Analysis	18
5.4.1	Parameterization	18
5.4.2	Complexity of Completed Puzzle Generation	18
5.4.3	Complexity of Uniqueness Testing and Random Filling	18
5.4.4	Profiling Method	18
5.4.5	WNEF vs Running Time	19
5.5	Testing	19
5.5.1	WNEF as a Function of Design Choices	19
5.5.2	Hypothesis Testing	19
6	Strengths and Weaknesses	19
7	Conclusions	21
References		21
1	Source Code	23
2	Screenshots of Puzzle Generator	62
<hr/>		
List of Figures		
1	Demonstration of indexing schemes.	6
2	Puzzle generated by WebSudoku (ranked as “Easy”).	6
3	Top 1465 Number 77.	7
4	An example of a hand-made Nikoli puzzle.	7
5	Example of the Naked Pair rule.	8
6	Example of the Naked Triplet rule.	8
7	Example of the Hidden Pair rule.	8
8	Example of the Hidden Triplet rule.	9
9	Example of the Multi-Line rule.	9
10	Examples of choice histograms.	11
11	WNEF for control puzzles by difficulty.	13
12	WNEF correlations for various weighting functions.	13
13	Running time as a function of the obtained WNEF.	20
14	WNEF as a function of allowed failures.	20
15	Screenshots of puzzle generator.	62

List of Figures

1	Demonstration of indexing schemes.	6
2	Puzzle generated by WebSudoku (ranked as “Easy”).	6
3	Top 1465 Number 77.	7
4	An example of a hand-made Nikoli puzzle.	7
5	Example of the Naked Pair rule. . .	8
6	Example of the Naked Triplet rule. .	8
7	Example of the Hidden Pair rule. . .	8
8	Example of the Hidden Triplet rule.	9
9	Example of the Multi-Line rule. . . .	9
10	Examples of choice histograms. . . .	11
11	WNEF for control puzzles by diffi- culty.	13
12	WNEF correlations for various weighting functions.	13
13	Running time as a function of the obtained WNEF.	20
14	WNEF as a function of allowed fail- ures.	20
15	Screenshots of puzzle generator. . . .	62

1 Introduction

1.1 Statement of Problem

We set out to design an algorithm that would construct unique sudoku puzzles of various difficulties as well as to develop metrics by which to measure the difficulty of a given puzzle. In particular, our algorithm must admit at least four levels of difficulty while minimizing its level of complexity.

1.2 Relevance of Sudoku

We feel that this problem is relevant and of interest, since the game of sudoku is inherently mathematical, and offers rich opportunities to explore mathematical techniques. Indeed, the problem is NP-Complete [3], and yet manages to be somewhat accessible to casual analysis. Moreover, by developing techniques for use with a problem over which we have such complete control, we may expand into other and more practical problems. In fact, sudoku is essentially an exercise in compression, and so techniques for generating difficult puzzle instances lead directly to realizations about information content and entropy. We, however, shall restrict our focus directly to the problem at hand, and be content to leave these reasons, along with sudoku's entertainment value, as our motivation for exploring the game.

1.3 Goals

Our goal is to create an algorithm that will produce sudoku puzzles. In doing so, and to meet the conditions of the proposed problem (section 1.1), we aim to create an algorithm with the following properties:

- Will only create valid puzzle instances (no contradictions, and admitting a unique solution).
- Can generate puzzles at any of four different difficulty levels (easy, medium, hard and evil¹).
- Produces puzzles in a reasonable amount of time, regardless of the chosen difficulty.

Such a set of goals could easily lead to a project of an unmanageable scope. Thus, we explicitly do not aim for any of the following properties:

- Attempt to “force” a particular solving method upon players.
- To be the best available algorithm for the task of making exceedingly difficult puzzles.
- Impose symmetry requirements .

1.4 Rules of Sudoku

The game of sudoku is played upon a 3×3 grid of blocks, each of which is a 3×3 grid of *cells*. Each cell can have a *value* of 1 through 9, subject to a simple constraint, or may be empty. The object of the game is to, given a partially-filled out grid called a puzzle, use logical inference to place values in all of the empty cells such that the constraints are upheld. It is fully possible to create a puzzle which has no solution (it contradicts itself, forcing the player to violate a constraint), or which has multiple solutions. We shall impose the additional requirement upon puzzles that they admit exactly one solution each.

When properly filled out, no row, column or block may have two cells with the same value. This simple constraint is what allows all of the inference to work. Some examples of puzzles and their solutions may be found in Section 1.8. For more details and a complete tutorial, please see [1].

1.5 Terminology and Notation

It is difficult to discuss our solution to the proposed problem without understanding some common terminology. Moreover, since we will apply more mathematical formalism here than in most documents dealing with sudoku, it will be helpful to introduce notational conventions.

Assignment A tuple (x, X) of a value and a cell. If a puzzle contains an assignment (x, X) , we say that X has the value x , that X maps to x , or that $X \mapsto x$.

¹This term was chosen for traditional reasons, as many sources prefer to use references to immorality to measure difficulty.

Candidates A set of those values which may be assigned to a square. As more information is taken into account, the set is reduced until only one candidate remains, at which point it becomes the value of the cell. We denote the set of candidates for some cell X by $X?$.

Cell A single square within a sudoku puzzle, which may have one of the integer values from 1 to 9. We denote cells using uppercase italic serif letters: X, Y, Z .

Block One of the nine 3×3 squares within the puzzle. The boundaries of these blocks are denoted by thicker lines on the puzzle's grid. It is important to note that in sudoku, no two blocks overlap (share common cells). There are variants of sudoku, such as hypersudoku in which this occurs, but we will focus our attention on the traditional rules.

Grouping A set of cells in the same row, column or block. We represent groupings with uppercase boldface serif letters: X, Y, Z . We refer unambiguously to the row groupings R_i , the column groupings C_j and the block groupings B_c , following the indexing scheme in section 1.6. The set of all groupings will be denoted \mathbb{G} .

Metric We call a function $m : \mathbb{P} \rightarrow \mathbb{R}$ (assigning a real number to each valid puzzle) a metric if it provides information about the relative difficulty of the puzzle.

Puzzle A 9×9 matrix of cells, with at least one empty and at least one filled cell. For our purposes, we impose the additional requirement that all puzzles have exactly one solution. We denote puzzles by boldface capital serif letters: P, Q, R . Since this notation conflicts with that for groupings, we will always denote that a variable is a puzzle. Moreover, we refer to cells belonging to a puzzle: $X \in P$. Finally, in the rare instance that we wish to denote the set of all valid puzzles, we shall do so with a double-struck P : \mathbb{P} .

Representative The upper-left cell in each block is that block's representative. For example, the cell in the 5th row and 5th col-

umn has as its representative the cell at the fourth row and column.

Restrictions In some cases, it is more straightforward to discuss which values a cell cannot be assigned to than to discuss the set of candidates. Thus, the restrictions set $X!$ for a cell X is defined as $\mathbb{V} \setminus X?$.

Rule An algorithm which accepts a puzzle P and produces either a puzzle 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. For our purposes, all sudoku puzzles use the traditional numeric value set $\mathbb{V} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. This can be confusing at times, since we will be discussing other numbers, but we choose to do so for the sake of convention. A value is denoted by a lower case sans serif letter: x, y, z .

1.6 Indexing

Define the following indicies using the terminology above (section 1.5). As a convention, all indicies will start with zero for the first cell or block.

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

These indices are related by the following functions:

$$\begin{aligned}
 c(i, j) &= \frac{j}{3} + \left\lfloor \frac{i}{3} \right\rfloor \cdot 3 \\
 i(c, k) &= 3 \left\lfloor \frac{c}{3} \right\rfloor + \left\lfloor \frac{k}{3} \right\rfloor \\
 j(c, k) &= (c \bmod 3) \cdot 3 + (k \bmod 3) \\
 i'(c) &= 3 \left\lfloor \frac{c}{3} \right\rfloor \\
 j'(c) &= (c \bmod 3) \cdot 3 \\
 i'(i) &= 3 \left\lfloor \frac{i}{3} \right\rfloor \\
 j'(j) &= 3 \left\lfloor \frac{j}{3} \right\rfloor
 \end{aligned}$$

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”.

1.7 Formal Rules of Sudoku

We may now formally state the rules of sudoku that restrict allowable assignments using the notation developed thus far:

$$(\forall \mathbf{G} \in \mathbb{G} \forall X \in G) \quad X \mapsto v \Rightarrow \nexists Y \in \mathbf{G} : Y \mapsto v$$

Applying this sort of formalism to the rules of sudoku will allow us to make strong claims about solving techniques later, and so it is useful introduce this notation for the rules.

1.8 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 6 shows a puzzle ranked as “Easy” by WebSudoku [4].

By contrast, Figures 7 and 7 show the results of two different approaches to generating difficult puzzles: the first one was computer generated as part of an experiment in minimal sudoku puzzles, whereas the second was hand-made by the authors at Nikoli, the company most famously associated with sudoku. It is interesting that two such completely different approaches result in very similar looking puzzles.

2 Background

2.1 Common Solving Techniques

As with any activity, several sets of techniques have emerged to help solve sudoku puzzles. We collect some here so that we may refer to them in our own development. In all of the techniques below, we assume that the puzzle being solved has a single unique solution. These techniques and examples are adapted from [10] and [2].

2.1.1 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 may 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 $Z \mapsto p$, then $X \not\mapsto p$, which implies that $X \mapsto q$. This in turn means that $Y \not\mapsto q$, but we have from $Z \mapsto p$ that $Y \not\mapsto p$, leaving $Y? = \emptyset$. Since the puzzle has a solution, this is a contradiction, and $Z \not\mapsto p$.

As an example of this arrangement is shown in figure 5. The cells marked X and Y satisfy $X? = Y? = \{2, 8\}$, and so we can remove both 2 and 8 from all other cells in \mathbf{R}_8 . That is, $\forall Z \in (\mathbf{R}_8 \setminus \{X, Y\}) : 2, 8 \notin Z?$.

2.1.2 Naked Triplet

This rule is analogous to the Naked Pair rule (section 2.1.1), but instead it 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 $\{t, u, v\}$. Then, by exhaustion, there is a one-to-one set of assignments from $\{X, Y, Z\}$ to $\{t, u, v\}$. Therefore, no other cell in \mathbf{A} may map to a value in $\{t, u, v\}$.

An example of this is given in Figure 6. Here, 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 may remove them from the candidates for those cells marked with an asterisk.

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

Figure 3: Top 1465 Number 77.

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

Figure 4: An example of a hand-made Nikoli puzzle.

of the cells in column 0 and block 3.

2.2.3 GNOME Sudoku

2.2 Previous Works

2.2.1 SudokuExplainer

The SudokuExplainer application [6] 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.

2.2.2 QQWing

The QQWing application [8] 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.

Included with the GNOME Desktop Environment, GNOME Sudoku is a desktop application for playing the game. It is written in Python, and distributed in source form, and so one may directly call the generator routines that it uses.

The application assigns a difficulty value on the range from zero to one to each puzzle, and rather than tuning the generator to requests, simply regenerates any puzzle outside of a requested difficulty range. It was thus not useful as a model of how to write a tunable generator, but was very helpful for quickly generating large amounts of control puzzles. We used a small Python script, shown on page 61, to extract the puzzles.

			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 5: Example of the Naked Pair rule.

		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 6: 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 7: Example of the Hidden Pair rule.

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 8: 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 9: Example of the Multi-Line rule.

3 Metric Design

3.1 Overview

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

As the first step in we first step in calculating this metric, we count the number of choices for each empty cell's value. Next, we 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. The implementations of this calculation process are shown on pages 28 and 42.

3.2 Assumptions

The design of the WNEF metric was predicated on two basic and important assumptions:

- We assumed that difficulty of a puzzle exists; that is, that there exists some objective standard by which we may rank puzzles in order of difficulty.
- We assumed that the difficulty of a puzzle is roughly proportional to the number of choices that a solver may make without directly contradicting any of the basic constraints outlined in Sections 1.4 and 1.7.

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

- We assume that the difficulty of the individual puzzles are independently and identically distributed over each source.

3.3 Mathematical Basis for WNEF

For this metric, we started by defining the choice function of a cell $c(X)$:

$$c(X) = |X?| \quad (1)$$

That is, the choice function indicates the number of possible choices that, in the worst case, must be explored. This function is only useful for empty cells, and so it is convenient to introduce a way

of referencing all cells in a puzzle P which are empty:

$$E(P) = \{X \in P \mid \forall v \in \mathbb{V} : X \not\vdash v\}$$

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

$$c_n(P) = |\{X \in P \mid c(X) = n\}| = |\{X \in P \mid |X?| = n\}| \quad (2)$$

Examples of these histograms with and without the mean control histogram (obtained from the control puzzles described in Section 4.1) removed may be found in Figures 10 (a) and (b).

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

$$\text{wef}(P) = \sum_{n=1}^9 w(n) \cdot c_n(P) \quad (3)$$

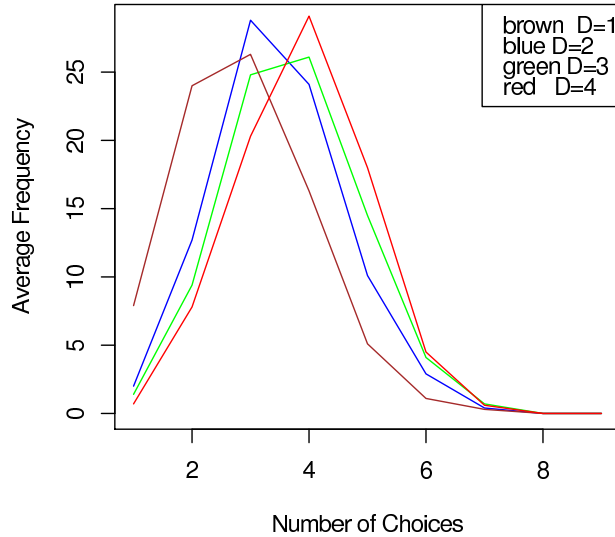
where $c_n(P)$ is the n^{th} value in the histogram $\vec{c}(P)$. This function, however, has the absurd trait that removing information from a puzzle results in more empty cell, which in turn causes the function to strictly increase. We therefore calculate the weighted and *normalized* ease function:

$$\text{wnef}(P) = \frac{\text{wef}(P)}{w(1) \cdot |E(P)|} \quad (4)$$

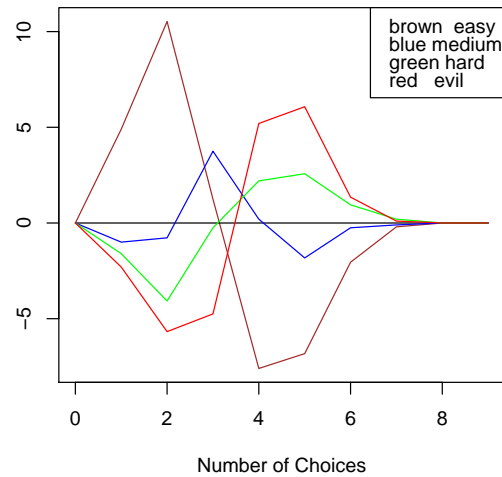
This calculates the ratio of the weighted ease function to the maximum value that it can have (all empty cells completely determined, but have not been filled in; that is, all cells may be assigned by elimination alone). We experimented with three different weight functions, before deciding upon the *exponential weight function*. This decision was made in response to tests performed during metric calibration, and thus a full discussion of why we chose that particular weight function will be deferred to Section 4.2. Whenever the choice of weighting function is ambiguous, we shall indicate the choice with a subscript *exp*, *sq* or *lin* corresponding to the exponential, squared and linear functions.

3.3.1 Complexity

Essentially, the level of complexity involved in finding the WNEF is the same as that of finding the choice histogram (normalized or not). To



(a) Original histograms.



(b) Histograms with mean removed.

Figure 10: Examples of choice histograms.

do that, we need to find the direct restrictions on each cell by examining the row, column and block that it is located in. 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 being made. Of course, we only check when the cell is empty, and so for any puzzle, the number of comparisons is strictly less than 1,944. That bound is constant for all puzzles, and so we conclude that finding the WNEF is a constant time operation with respect to the puzzle difficulty.

4 Metric Calibration and Testing

4.1 Control Puzzle Sources

In calibrating and testing the metrics, we used published puzzles from several sources and at several levels of difficulty, as labeled by their authors. The puzzles we obtained include the following:

- WebSudoku [4]
 - 10 Easy puzzles.
 - 10 Medium puzzles.
 - 10 Hard puzzles.
 - 10 Evil puzzles.
- Games World of Sudoku [7]
 - 10 * puzzles.
 - 10 ** puzzles.
 - 10 *** puzzles.
 - 10 **** puzzles.
- GNOME Sudoku [5]
 - 2000 Hard puzzles.
- “top2365”²
 - 2365 Evil puzzles.

²This list of puzzles was obtained from [9] and named by regulars of the Sudoku Player’s Forum. By forum tradition, lists of test puzzles tend to get short and minimal names. Other names for lists include “topn87” and “subig20.”

4.2 Testing Method

4.2.1 Defining Difficulty Ranges

In analogy with published puzzle collections, we separated our control puzzles into four broad ranges of difficulty: easy, medium, hard and evil. For the sake of brevity, we will often refer to these by the indices 1, 2, 3 and 4, respectively.

4.2.2 Information Collection

We used the control puzzles described in 4.1 to calibrate and the metrics by running programs designed to calculate the metrics on each puzzle. The information collected from the program for each puzzle P_i included:

- $|E(P_i)|$, the total number of empty cells in P_i .
- $C(P_i) = \sum_{X \in P_i} X?$, the number of possible choices for all cells.
- The choice histogram \vec{c} defined in Equation 2.

4.2.3 Statistical Analysis of Control Puzzles

When looking for a possible correlation between the data and the difficulty level, we found that the number of empty cells and number of total choices lacked any correlation. However, when we looked at the choice histograms for each puzzle, we noticed trends in the data. In easier puzzles, there seemed to be more cells with fewer choices than in the more difficult puzzles (Figure 10).

We then calculated the $wnef(P)$ for the control puzzles to try to further explore the relationship and found a clear negative correlation between the difficulty level of P and $wnef(P)$ for the control puzzles (Figure 11). This leads us to introduce $\overline{wnef}(d)$ as the mean WNEF of all control puzzles having difficulty d .

In order to conclude that the WNEF produces distinct difficulty levels, which is to say that $\overline{wnef}(d) \neq \overline{wnef}(d+1)$ for $d \in \{1, 2, 3\}$, we conducted a hypothesis test for $d = 1, 2, 3$ with the following hypotheses:

$$\begin{aligned} H_0 &: \overline{wnef}(d) = \overline{wnef}(d+1) \\ H_a &: \overline{wnef}(d) \neq \overline{wnef}(d+1) \end{aligned}$$

To test these hypotheses, we used the following test statistic:

$$t^* = \frac{(\overline{wnef}(d) - \overline{wnef}(d+1))}{\sqrt{\frac{s_d^2}{n_d} + \frac{s_{d+1}^2}{n_{d+1}}}}$$

where n_d is the number of control puzzles having difficulty d and where s_d^2 is the sample variance of the WNEF, over control puzzles at level d (this data is shown in Table 1). With a significance level of $\alpha = 0.0025$, we performed a hypothesis test using the Student's t distribution, and found that $t^* > t_\alpha$. Thus, we rejected the null hypothesis for each of $d = 1, 2$ and 3 , and concluded that the WNEF is able to distinguish different difficulty levels.

4.3 Choice of Weight Function.

As alluded to in Section 3.3, we tried three different weighting functions for finding 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. We discovered that regardless of the type of weighting function we used, the graph showing the weights of the puzzles vs. difficulty all looked very similar, in that they all produced a strong negative correlation (Figure 12).

We concluded that we could choose any of the three weighting functions, as long as we used the same function throughout. We arbitrarily chose w_{exp} .

5 Generator Algorithm

5.1 Overview

The generator algorithm works by creating first a valid solved sudoku board, and 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 the application of various *strategies*. A strategy is simply an algorithm which outputs cell locations to attempt to remove, based on some goal. After any cell is removed, the puzzle is

d	1	2	3	4
$\hat{\mu}_d = E(y)$	0.2680756	0.1108268	0.09244832	0.04078146
$\hat{\sigma}_d^2 = s^2$	0.00096963	0.000502135	0.000255063	0.000125557

Table 1: Estimated means and variances of control WNEF metrics.

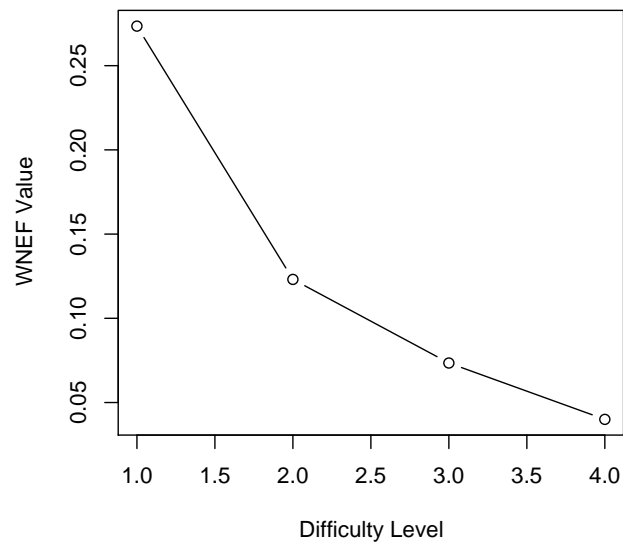


Figure 11: WNEF for control puzzles by difficulty.

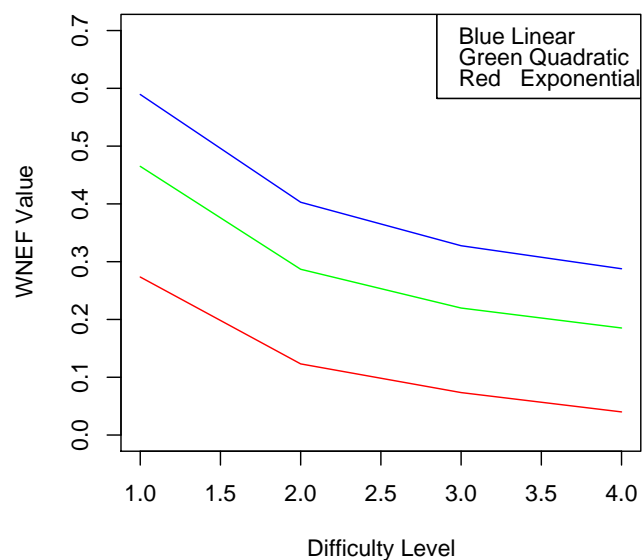


Figure 12: WNEF correlations for various weighting functions.

checked to ensure that it still admits 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 where additional cells are removed systematically, then return the completed puzzle. For harder difficulties, we introduce *annealing*.

5.2 Detailed Description

As mentioned, our algorithm for generating a deterministic Sudoku board consists of two stages. We first generate a solution, and then remove cells until we reach the desired difficulty, as measured by the WNEF metric. Also important is the *uniqueness test* algorithm used heavily in the process of removing cells..

5.2.1 Completed Puzzle Generation

Completed puzzles are generated via a method called 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 should be attempted in a systematic manner. If a valid solution is found, then we are done.

Backtracking can be a slow process, and as such one must make care to do so in a smart and efficient manner. In order to gain better efficiency, we take the 2D sudoku board and view it as a 1D list of rows. The problem now reduced to that of filling rows with values, and if we cannot, then we backtrack to the previous row. We are finished if we complete the last row.

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

There exists, of course, implementation details that one would need to iron out. To see our implementation, see Section 5.3.

5.2.2 Cell Removal

Having a solved puzzle is nifty, yes, however it is not very useful. In order to change this into a puzzle that is actually entertaining to solve we perform a series of removals that we shall call *masking*.

The basic idea behind masking is that one or more cells are removed from the puzzle (or 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, then the masking action is undone (or the cells are added back into the puzzle).

Random masking is one of the simplest and fastest forms of masking. Every cell is masked in turn, but in random order. Every cell that can be removed is, resulting in a minimal puzzle. This 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 that which can be gained with Random Masking (though easier puzzles can be created if they are not minimal). The idea behind tuned masking is that we can increase the probability that a given type of puzzle is generated. This depends heavily on probability, and hence takes some tweaking to make accurate. It can be done, however, such that the desired type of puzzle will be generated the majority of the time. As such, it is possible to ensure the generation of the puzzle type in question by regenerating the given type is generated. This has a terrible worst case. however probabilistic analysis may be used to show that, assuming your tuning is configured well, the probability of not gaining the desired puzzle type on a second try is very small.

The issue here is something I like to call *bleeding*. A given tuning, when ran many times, will produce a probability curve. In all likelihood, the produced puzzle will be of the type that constitutes the mean of the curve. However, should the puzzle lie far from these mean, on a tail, then it could overlap with a different tuning's curve and hence give you a conflict (such that you attempt to generate a hard puzzle and result in an evil puzzle, for example). Spacing the tunings out and minimizing their curve's spread is crucial to creating accurate tunings.

Behind the tuning algorithm is a series of *strategies*. A strategy is simply a function that examines the board and returns the cell it would like to try to remove. This should be based on some rule, perhaps it is in a cluster that has a lot of other filled cells in it, or its value is one that is currently very common. A set of these strategies defines how a tuning attempts to reduce a board.

The second stage of tuning is performed right after a value is removed from the board. This is that the board is evaluated to see if it is of the type that the tuning is seeking, and then the tuning's strategy is adjusted accordingly. In our example, if a board is found to be too difficult, then we might add back in a cell that will decrease the overall difficulty.

For our tuning we are seeking a board with a given WNEF. As such we apply strategies that will reduce the WNEF until we have reduced it sufficiently. Strategies that should have a large effect on the WNEF should not be applied if a low WNEF is not being sought. In the case that we reach a minimum WNEF that is not low enough, we can use a method from mathematical optimization known as *simulated annealing*. Here we add some number of values back into the board and then optimize from there, in hopes that doing so will allow us to reach a lower minimum. State saving allows us to then, after a time, revert to the board with the lowest WNEF. Experimentally we observed that annealing allowed us to produce puzzles with lower WNEF values than we could without applying the technique. The details of this test are given in Section 19.

5.2.3 Uniqueness Testing

In order to ensure we generate boards with only one solution, we must be able to test if this condition is true. There is a fast and a slow way of doing this. The fast way will find the uniqueness of any board which can be solved using logic. Any board which does not confirm to the rules of logic, but may still have a single solution, will fail the fast test. The slow test can determine this for any board.

The fast solution utilizes the two basic logic rules of Sudoku solving: Hidden Single and Naked Single. That is that any cell with only one possible value can be filled in with that value,

and any cell who is the only cell in some reference frame (such as its cluster, row, or column) with the potential of some value may be filled 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. If this test succeeds, then we know that the board always has a solution, as we generated the board from a solution. On the other hand, it may produce false negatives, and reject a board with 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 then the board has more than one solution. This solution calls itself recursively to determine the uniqueness of the board with the added values. The advantage of this solution is that it is completely accurate, and will not result in false negatives.

A hybrid method is to utilize the slow solution in the case that the fast one fails. A further optimization is to restrict the number of times the slow solution may be used. This is similar to saying "if we had to guess more than twice, then we reject the board." In the interest of expedience, it is the hybrid method that we adopt here. This allows us to prevent a large amount of false negatives while still offering quick solutions.

5.3 Pseudocode

5.3.1 Completed Board Generation

Given an empty 9×9 array that we shall call "board", do the following:

1. Fill the top row of the board with a random permutation of the sequence 1 through 9.
2. Initialize a 9 element array of lists. This shall hold all numbers placed so far in each column.
3. Initialize a 3 element array of lists. This shall hold all numbers placed in the three clusters that the current row (right now, this is the first row) spans.
4. Add the values of the first row to their respective column lists.

5. Add the values of the first row to their respective cluster lists.
6. Call a recursive function, and pass it the following:
 - A parameter directing it to fill the second row.
 - The columns array.
 - The clusters array.

The recursive function then performs the bulk of the algorithm:

1. Create an array containing a permutation of the sequence 1 through 9, which we shall call this “numbers.”
2. Create copies of the columns array, the clusters array, and of the numbers array, so that we may backtrack later.
3. If the requested line is the 10th line (off the end of the board), then we are done, and **return true**.
4. Initialize an empty “slack” array, which shall hold those values whose being placed caused a violation of constraints.
5. Move to the first column.
6. Repeat the following:
 - a) Pop a value off of the “numbers” array.
 - b) If this number is not in the clusters list for this column’s cluster, and is not in the columns list for this column, then:
 - i. Set this board location to this number.
 - ii. Add this number to the cluster and column lists that it applies to.
 - iii. Append all numbers in the “slack” array to the “numbers” array.
 - iv. Move to the next column.
 - c) Else we add the number to the slack array.
 - d) If we have passed the last column, then:

- i. If moving to the next line moves us passed our current three clusters (i. e. $(\text{line}+1)\%3$ is 0) then recurse with a reset clusters list and current columns list and incremented line number.
- ii. Else recurse with current clusters list and current columns list and incremented line number.
- iii. If recursion returned true, **return true**. Otherwise go on.

- e) If there are no numbers left (all numbers are slack, or recursion failed):
 - i. If we have shifted 9 or more times, **return false**.
 - ii. Recall all of our saved data.
 - iii. Delete all values from this row.
 - iv. Move to first column.
 - v. Erase the slack array.
 - vi. Cycle the numbers array, so the first item becomes last and all other items shift accordingly.
 - vii. Increment times shifted.

See also ?? and 40

5.3.2 Random Masking

Given a 9×9 array that we shall call “board”:

1. Initialize a 9×9 array of booleans to true, which we shall call the “mask”.
2. Initialize a list of 81 points with one point for every cell in the board.
3. Randomly permute the array of points.
4. For each element in this array:
 - a) Set the mask at that point to false. This will result in that value being considered not part of the board (or not given).
 - b) Test if this new puzzle is uniquely solvable.
 - c) If not, set the mask at that point back to true.

5.3.3 Tuned Masking

Given a 9×9 array that we shall call “board”:

1. Initialize a 9×9 array of booleans to true, call this the “mask”.
 - a) Repeat the following until we are done:
 - i. Apply some strategy in order to obtain the coordinates of a cell to remove.
 - ii. Set the mask at those coordinates to false. This will result in that value being considered not part of the board (or not given).
 - iii. Test if this new puzzle is uniquely solvable.
 - iv. If not, set the mask at those coordinates back to true and select a new strategy.
 - v. Calculate board statistics and test to see if we match them. In our case, this is the WNEF.
 - vi. If we are too high, continue from (a).
 - vii. If we are too low, repeat the following a small number of times:
 - A. Apply an annealing function to gain the location of a cell to add.
 - B. Set the mask at that location to true.
 - viii. If we are within the desired range, we are done.

5.3.4 Uniqueness Testing

Given a 9×9 array that we shall call “board”, a 9×9 array that we shall call “mask”, and a number of times to guess:

1. Fill in a 9×9 array with lists such that each lists represents the value choices available at that cell.
2. Repeat the following:
 - a) If mask contains no false values, return true.
 - b) If there exists any list in the choices array with only one value:

- i. Set the mask at that position to true.
 - ii. Continue from 2.
- c) Look for a value in the choices array that appears only once in a cluster, if found:
 - i. Set the mask at that position to true.
 - ii. Continue from 2.
- d) Look for a value in the choices array that appears only once in a row, if found:
 - i. Set the mask at that position to true.
 - ii. Continue from 2.
- e) Look for a value in the choices array that appears only once in a column, if found:
 - i. Set the mask at that position to true.
 - ii. Continue from 2.
- f) If the number of times we are allowed to guess is not 0:
 - i. Locate the blank cell with the least number of choices.
 - ii. Set a flag to false.
 - iii. For each choice:
 - A. Set that cell of the board to that choice and set that cell of the mask to true.
 - B. Recurse, decrementing the number of allowed guesses.
 - C. If the the result is true, and the flag is true, return false.
 - D. Else if the result was true, set the flag to true.
 - iv. If the flag is true, return true: we have found a unique solution.
- g) Return false: we know that the board is most likely not unique.

5.4 Complexity Analysis

5.4.1 Parameterization

Traditionally, when one analyzes the complexity of an algorithm, the complexity is considered as a function of some parameter representing the size of the problem. Thus, the first thing we must decide in analyzing the generator is what we will consider its complexity to be a function of. The most natural parameter would be the size of the sudoku grid, but since we only consider the traditional 9×9 grid (as opposed to “hex sudoku,” which is played on a 16×16 board, or the more pathological boards, such as those of size 36×36 and 100×100) this isn’t a parameter at all. Thus, instead, we resort to the only variable that we utilize when generating puzzles: the desired difficulty level d . Our complexity measure will thus be a function of the form $t(d) = f(d) \cdot t_0$, where t is the time complexity, f is some function that we will find through our analysis, and where t_0 is the time complexity for generating a puzzle randomly.

5.4.2 Complexity of Completed Puzzle Generation

The completed puzzle generation algorithm does a series of work for each line of the Sudoku, and potentially does this work over all possible different boards. As such, 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. That is, $(9 \times 9 \times 9)^9 = (9^3)^9 = 9^{27} \approx 5.8 \times 10^{25}$. While it is true that this is a constant, the size of the constant is prohibitively large.

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 (that is assume we have to try all 9 values, in all 9 cells, and perform all 9 shifts) but let us assume we only do this once per line. Here we get $9 \times 9 \times 9$ or 6561. The actual value may be less than that, or slightly more, but should hover about that area. The best case is of course 81, where all values work first try. We have a very high worst case, but very reasonable average and best cases. The worst case presented could likely be reduced with analysis

of how the rules of sudoku limit the number of invalid boards possible (worst case assumes that every board could be invalid). In practice this algorithm runs in negligible time in comparison to the masking algorithms.

5.4.3 Complexity of Uniqueness Testing and Random Filling

In the worst case, the “fast” uniqueness algorithm will examine each of the 81 cells, and compare it to each of the other 81 cells. Thus, without adding in any brute force functionality, the uniqueness test can be completed in a constant number of operations: $81 \times 81 = 6,561$. When we consider the hybrid algorithm, and include in our analysis the brute force searching, we find that 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 admits a linear complexity with respect to the number of allowed guesses.

This allows us to now consider the complexity of the random filling algorithm. Since it does not allow any guessing when it calls the uniqueness algorithm, and since it performs the uniqueness test exactly once per cell, it performs exactly $81^3 = 531,441$ comparisons. As such, it is a constant time operation, and can be used as a point of comparison for more complicated algorithms.

5.4.4 Profiling Method

In order to collect empirical data on the complexity of puzzle generation, we implemented a small code profiling utility class in PHP, as is shown on page 32. This class exploits that, in PHP 5.0 and later, when a function-scope class instance variable is created, its destructor is called immediately after the function returns. Thus, we create an instance of Profiler at the start of each interesting function, and pass the `__FUNCTION__` and `__LINE__` macros to its constructor. The class then compiles timing information into global variables that are queried after the puzzle is successfully generated.

In all uses of this profiling data, we will 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.

5.4.5 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 that is difficult to accommodate. Therefore, we rely on our profiler to gather empirical data about the complexity of generating puzzles. In particular, Figure 13 shows the normalized running time required to generate a puzzle as a function of the obtained WNEF after annealing is applied. In order to show detail, we plot the normalized time on a logarithmic scale (base 2).

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

5.5 Testing

5.5.1 WNEF as a Function of Design Choices

The generator algorithm, as written, is fairly generic. We thus need some way to empirically determine constant terms, such as how many times we will allow for cell removal to fail before we conclude that the puzzle is minimal. We thus plotted the number of failures that we permitted to the WNEF produced, shown in Figure 14. This plot shows us both that we only need to allow a very small number of failures to enjoy small WNEF values, and that annealing reduces the value still further, even in low-failure scenario

5.5.2 Hypothesis Testing

5.5.2.1 Effectiveness of Annealing To show that the process of annealing resulted in lower WNEF values, and was thus a useful addition to the algorithm, we tested the hypothesis that it was effective versus the null hypothesis that it was not:

$$\begin{aligned} H_0 &: \mu = \mu' \\ H_a &: \mu \neq \mu' \end{aligned}$$

where μ is the mean WNEF for puzzles produced without the aid of annealing and where μ' is

the mean WNEF for those produced with annealing enabled. We considered a sample of puzzles of size n , whose means and variances were (\bar{y}, s^2) for non-annealed puzzles and (\bar{y}', s'^2) for annealed. Once again, we used the following t -statistic:

$$t^* = \frac{(\bar{y} - \bar{y}')}{\sqrt{\frac{s^2}{n} + \frac{s'^2}{n}}}$$

At a significance level of $\alpha = 0.0005$ and using the data shown in Table 2, we rejected the null hypothesis and concluded that annealing lowered the WNEF values.

5.5.2.2 Distinctness of Difficulty Levels To determine whether the difficulty levels of our puzzle generator were unique, we performed a Student's t -distribution hypothesis test using the following hypotheses:

$$\begin{aligned} H_0 &: \mu_d = \mu_{d+1} \\ H_a &: \mu_d \neq \mu_{d+1} \end{aligned}$$

where μ_d is the mean WNEF of puzzles produced by our generator algorithm when given d as the target difficulty. Using a significance level of $\alpha = 0.0005$ with the data shown in Table 2, we use the following as our test statistic:

$$t^* = \frac{(\bar{y}_d - \bar{y}_{d+1})}{\sqrt{\frac{s_d^2}{n_d} + \frac{s_{d+1}^2}{n_{d+1}}}}$$

where \bar{y}_d is the mean of n_d puzzles produced by the algorithm, having a sample variance s_d^2 . We found that for all d , $t^* > t_\alpha$, and thus we were able to reject H_0 for all difficulty levels. We concluded that all of the difficulty levels of our puzzle generator are indeed unique.

6 Strengths and Weaknesses

Our approach to measuring the difficulty of sudoku puzzles admits some real and important weaknesses. Primary among these is that 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, may be crafted such that their

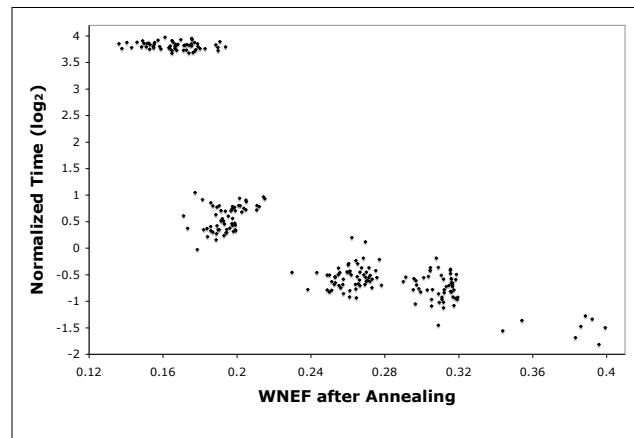


Figure 13: Running time as a function of the obtained WNEF.

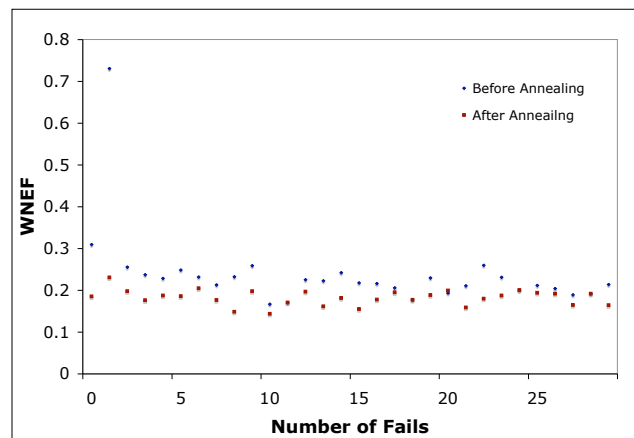


Figure 14: WNEF as a function of allowed failures.

Difficulty	1	2	3	4
<i>Pre-annealing</i>				
Mean	0.523999895	0.327451814	0.271656591	0.27661661
Variance	0.017110796	0.005454866	0.002581053	0.004039649
<i>Post-annealing</i>				
Mean	0.31876731	0.26157134	0.194262257	0.165920803
Variance	0.000696284	9.32606×10^{-5}	8.7219×10^{-5}	0.000185543

Table 2: Pre- and post-annealing WNEF mean and variances ($n = 60$).

WNEF is higher than easier puzzles. In acknowledging this weakness, we recognize that there is a limited regime over which the WNEF metric is useful. In practice, this regime seemed to exclude only those puzzles made by computer-based generators designed to enforce the use of particular techniques. This was the case, for example, with both QQWing and SudokuExplainer.

On the other hand, the WNEF approach offered one very definite and notable advantage: it may be calculated very quickly. In the worst case, it looks at the 24 cells adjacent to each cell in the puzzle. Thus, even at its worst, the WNEF requires only 1,944 cell look-ups, leading us to conclude that calculating the WNEF is constant with respect to the puzzle difficulty. Moreover, the actual constant bound is relatively small, allowing us to make frequent evaluations of the WNEF while tuning puzzles.

Likewise, our generator algorithm admits some very real weaknesses. In particular, it 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 attempts to direct the outcome of probability, but that it is still inherently a random algorithm. As such, the fact that the probability of randomly creating a puzzle with a small WNEF value is very low (that is, a random generator will produce them very infrequently) implies that our algorithm will produce them infrequently as well. As such, even with tuning, there is still a very good chance that one will not generate such a hard puzzle. The option of continuing with the algorithm until you do can take an unreasonable amount of time.

All this said, however, the algorithm has the advantage of creating puzzles quickly with little algorithmically induced similarities between puzzles. Our method here is very similar to the method of 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 this without generating more than one puzzle, and that we can generate difficult puzzles in less time than it takes to generate multiple puzzles and discard the easiest among them.

7 Conclusions

In this paper, we introduced and proposed a metric, the weighted normalized ease function (WNEF), with which to estimate the difficulty of a given sudoku puzzle. We based this metric upon the observation that the essential difficulty encountered in solving comes about as a result of the ambiguities which must be eliminated. Thus, the metric represented how this ambiguity was distributed across the puzzle.

Using data that we collected from the control puzzles, we found that the WNEF showed a strong negative correlation with the level of difficulty (the harder a puzzle was, the lower the WNEF value). We then conducted a hypothesis test to prove with a confidence level of 99.5% that the WNEF values of different difficulty levels were indeed distinct. We also found that the specific choice of weighting function did not change this correlation, and thus made an arbitrary choice to use as our weighting function.

We also designed an algorithm that employs these insights to create puzzles of selectable difficulty. This algorithm works by employing backtracking and annealing to optimize the WNEF metric towards some desired level. Statistical hypothesis tests showed with a 99.95% confidence level that the annealing led to more optimal results, and that the generator successfully produced puzzles falling into four distinct ranges of difficulty.

References

- [1] *Sudoku tutorial*. <http://www.nikoli.co.jp/en/puzzles/sudoku/>. Nikoli Puzzles.
- [2] *Techniques for solving sudoku*. <http://www.sudokuoftheday.com/pages/techniques-overview.php>. Astraware Limited.
- [3] L. AARONSON, *Sudoku science*. <http://spectrum.ieee.org/feb06/2809>, Feb. 2006.
- [4] G. GREENSPAN, *Websudoku*. <http://websudoku.com/>.

-
- [5] T. HINKLE, *Gnome sudoku application*. <http://http://gnome-sudoku.sourceforge.net/>.
- [6] N. JUILLERAT, *Sudokuexplainer application*. <http://diuf.unifr.ch/people/juillera/Sudoku/Sudoku.html>.
- [7] J. NANNI, *Games world of sudoku*, April 2008.
- [8] S. OSTERMILLER, *Qqwing application*. <http://ostermiller.org/qqwing/>.
- [9] STERTEN@AOL.COM, *Magictour hosted sudoku resources*. <http://magictour.free.fr/sudoku.htm>.
- [10] A. M. TAYLOR, *Dell sudoku challenge*, Spring 2008.
-

1 Source Code

Listing 1: Implementation of classification functions and WNEF metric.

```

1  /*
2   * Puzzle.java: Encapsulates most details about a puzzle.
3   */
4
5  package sudokumetricizer;
6
7  import java.io.BufferedReader;
8  import java.io.Reader;
9  import java.util.Scanner;
10
11 public class Puzzle {
12
13     /**
14      * All values are calculated from the exponential weighting function.
15      * See Section 4.2 for how these values were calculated,
16      * and Table 1 for the actual values.
17      */
18     public static enum Difficulty {
19         EASY      (1, 0.2680756, 0.00096963 ),
20         MEDIUM   (2, 0.1108268, 0.000502135),
21         HARD      (3, 0.09244832, 0.000255063),
22         EVIL      (4, 0.04078146, 0.000125557);
23
24         // For all of these fields, please see Section 4.2.
25         public final double
26             /**
27              * Estimate of the variance in the WNEF for puzzles of this
28              * difficulty.
29              */
28             EST_VAR_WNEF,
30             /**
31              * Estimate of the mean WNEF for puzzles of this difficulty.
32              */
32             EST_MEAN_WNEF,
33             /**
34              * Estimate of the standard deviation for puzzles of this
35              * difficulty.
36              */
36             EST_STDDEV_WNEF;
37
38         /**
39          * Numeric value that may be used in interpolation.
40          */
41         public final int DIFFICULTY_INDEX;
42
43         Difficulty(int difficulty_index,
44                   double est_mean_wnef,
45                   double est_var_wnef) {
46             DIFFICULTY_INDEX = difficulty_index;
47             EST_VAR_WNEF = est_var_wnef;
48             EST_MEAN_WNEF = est_mean_wnef;
49             EST_STDDEV_WNEF = Math.sqrt(EST_VAR_WNEF);
50         }
51
52         /**
53          * A useful numerical constant equal to  $1/\sqrt{2\pi}$ .
54          */
55         public final static double

```

```

59         ROOT_1OVER_2PI = Math.sqrt(1.0/(2.0*Math.PI));
60
61
62
63
64
65
66
67
68
69
70
71

```

$$/ * f(w_{nef} = w \mid D = d) = \frac{1}{2\pi\hat{\sigma}^2} \exp\left\{-\frac{1}{2}\hat{\sigma}^2(w - \hat{\mu})\right\} */$$

```

72     public double pdf(double given_wnef) {
73         double p = (1.0/EST_STDDEV_WNEF) * ROOT_1OVER_2PI *
74             Math.exp(
75                 (-0.5 / EST_VAR_WNEF) *
76                 Math.pow(given_wnef - EST_MEAN_WNEF, 2.0)
77             );
78         return p;
79     }
80
81 }
82
83 private final static int[] EXP_EASE_WEIGHTS =
84     {256,128,64,32,16,8,4,2,1};
85
86 private final static int[] LINEAR_EASE_WEIGHTS =
87     {9,8,7,6,5,4,3,2,1};
88
89 private final static int[] SQUARE_EASE_WEIGHTS =
90     {81,64,49,36,25,16,9,4,1};
91
92 private int[][] cells;
93
94 /**
95  * Builds a puzzle given its cells.
96  */
97 public Puzzle(int[][] cells) {
98     this.cells = cells.clone();
99 }
100
101 /**
102  * Builds a puzzle given its cells expressed in a one-dimensional array.
103  */
104 public Puzzle(int[] linear_cells) {
105     this.cells = new int[9][9];
106     for (int i = 0; i < 9; i++) {
107         for (int j = 0; j < 9; j++) {
108             cells[i][j] = linear_cells[i*9+j];
109         }
110     }
111 }
112
113 /**
114  * Builds up a puzzle by reading integers from a Reader object.
115  */
116 public Puzzle(Reader r) {
117
118     int idx = 0;
119     final int max = 81;
120

```



```
121     cells = new int[9][9];
122
123     Scanner scan = null;
124     scan = new Scanner(new BufferedReader(r));
125
126     while (scan.hasNext() && idx < max) {
127         int next = scan.nextInt();
128         cells[idx / 9][idx % 9] = next;
129         idx++;
130     }
131
132 }
133
134 /**
135  * Counts the number of empty cells in the puzzle.
136  */
137 public int numEmptyCells() {
138
139     int count = 0;
140
141     for (int[] row: cells) {
142
143         for (int c: row) {
144             if (c == 0) {
145                 count++;
146             }
147         }
148     }
149
150     return count;
151 }
152
153
154 /**
155  * Returns the cluster number of the cell (i,j).
156  */
157 public int blockOf(int i, int j) {
158     return (int) (Math.floor(j/3) + 3*Math.floor(i/3));
159 }
160
161
162 /**
163  * Returns the row index of the block representative for the given block
164  * index.
165  */
166 public int rowRepresentativeOf(int block) {
167     return 3 * (int) Math.floor((double) block / 3.0);
168 }
169
170 /**
171  * Returns the row index of the block representative for the cell with given
172  * row and column indicies.
173  */
174 public int rowRepresentativeOf(int i, int j) {
175     return rowRepresentativeOf(blockOf(i,j));
176 }
177
178 /**
179  * Returns the column index of the block representative for the given block
180  * index.
181  */
182 public int colRepresentativeOf(int cluster) {
```

```

183         return 3 * (cluster % 3);
184     }
185
186     /**
187      * Returns the column index of the block representative for the cell with
188      * given row and column indicies.
189      */
190     public int colRepresentativeOf(int i, int j) {
191         return colRepresentativeOf(blockOf(i,j));
192     }
193
194     /**
195      * Finds constraints on a cell by examining other cells on the same row.
196      *
197      * @param constraints
198      *      constraints[n] == true indicates that cell[i][j]
199      *      cannot be (n + 1).
200      */
201     public void constrainCellByRow(int i, int j, boolean[] constraints) {
202
203         for (int other_j = 0; other_j < cells[i].length; other_j++) {
204             if (other_j != j && cells[i][other_j] != 0) {
205                 constraints[cells[i][other_j] - 1] = true;
206             }
207         }
208     }
209
210
211     /**
212      * Finds constraints on a cell by examining other cells on the same column.
213      *
214      * @param constraints
215      *      constraints[n] == true indicates that cell[i][j]
216      *      cannot be (n + 1).
217      */
218     public void constrainCellByCol(int i, int j, boolean[] constraints) {
219
220         for (int other_i = 0; other_i < cells.length; other_i++) {
221             if (other_i != i && cells[other_i][j] != 0) {
222                 constraints[cells[other_i][j] - 1] = true;
223             }
224         }
225     }
226
227
228     /**
229      * Finds constraints on a cell by examining other cells within the same
230      * block.
231      *
232      * @param constraints
233      *      constraints[n] == true indicates that cell[i][j]
234      *      cannot be (n + 1).
235      */
236     public void constrainCellByCluster(int i, int j, boolean[] constraints) {
237
238         int orig_i = rowRepresentativeOf(i,j),
239         orig_j = colRepresentativeOf(i,j);
240
241         final int lim_i = orig_i + 3, lim_j = orig_j + 3;
242
243         for (int other_i = orig_i; other_i < lim_i; other_i++) {
244             for (int other_j = orig_j; other_j < lim_j; other_j++) {

```

```

245         if (other_i != i && other_j != j && cells[other_i][other_j] != 0) {
246             constraints[cells[other_i][other_j] - 1] = true;
247         }
248     }
249 }
250
251 }
252
253 /**
254  * Returns a histogram of the choices available to each cell, as determined
255  * by simple elimination.
256  *
257  * @returns
258  *     An array  $\vec{c}$  such that  $c_n$  is the number of cells with
259  *      $n+1$  available choices.
260  */
261 public int[] histChoices() throws RuntimeException {
262
263     int[] hist = new int[9];
264
265     for (int i = 0; i < 9; i++) {
266         for (int j = 0; j < 9; j++) {
267             hist[numChoicesForCell(i, j) - 1]++;
268         }
269     }
270
271     return hist;
272 }
273
274 /**
275  * Counts the number of choices available for a given cell, as determined by
276  * simple elimination.
277  */
278 public int numChoicesForCell(int i, int j) {
279
280     int count = cells.length;
281
282     boolean[] constraints = new boolean[cells.length];
283
284     // Set everything to false.
285     for (int idx = 0; idx < cells.length; idx++) {
286         constraints[idx] = false;
287     }
288
289     constrainCellByRow(i, j, constraints);
290     constrainCellByCol(i, j, constraints);
291     constrainCellByCluster(i, j, constraints);
292
293     // Count the number of restrictions.
294     for (int idx = 0; idx < cells.length; idx++) {
295         if (constraints[idx]) count--;
296     }
297
298     return count;
299 }
300
301 }
302
303 /**
304  * Counts the total number of choices available to all empty cells on the
305  * puzzle, as determined by simple elimination.
306  */

```

```

307     public long totalChoices() {
308
309         long count = 0;
310
311         for (int i = 0; i < 9; i++) {
312             for (int j = 0; j < 9; j++) {
313                 if (cells[i][j] == 0) {
314                     count += numChoicesForCell(i, j);
315                 }
316             }
317         }
318
319         return count;
320
321     }
322
323     /**
324      * Evaluates the weighted normalized ease function for the puzzle, using the
325      * exponential weight function.
326      */
327     public double wnef() {
328         return wnef(EXP_EASE_WEIGHTS);
329     }
330
331     /**
332      * Calculates the Weighted Normalized Ease Function.
333      */
334     public double wnef(int[] weights) {
335
336         long count = 0;
337
338         for (int i = 0; i < 9; i++) {
339             for (int j = 0; j < 9; j++) {
340                 if (cells[i][j] != 0) {
341                     count += weights[numChoicesForCell(i, j) - 1];
342                 }
343             }
344         }
345
346         return (double) count / (double) (weights[0] * numEmptyCells());
347
348     }
349
350     /**
351      * Estimates the difficulty class of the puzzle by finding which class gives
352      * the highest value of the WNEF probability distribution function.
353      *
354      * This method effectively implements Equation ??
355      */
356     public Difficulty estimatedDifficulty() {
357
358         double w = wnef();
359         double max_pdf = -1.0;
360         Difficulty diff = null;
361
362         for (Difficulty d: Difficulty.values()) {
363             double last_pdf = d.pdf(w);
364             if (last_pdf > max_pdf) {
365                 max_pdf = last_pdf;
366                 diff = d;
367             }
368         }

```

```

369         return diff;
370     }
371 }
372
373 /**
374  * Returns a space-separated list of metrics. In order:
375  *     - number of empty cells
376  *     - total number of choices
377  *     - the exponential wnef
378  *     - the square wnef
379  *     - the linear wnef
380  *     - the estimated difficulty index
381  *     - the value of the pdf used to find the estimated difficulty
382  */
383 public String metricsString() {
384     String histStr = java.util.Arrays.toString(histChoices());
385     histStr = histStr.substring(1, histStr.length() - 1);
386
387     Difficulty d = estimatedDifficulty();
388
389     double w = wnef(EXP_EASE_WEIGHTS);
390
391     return Integer.toString(numEmptyCells()) + "_" +
392            Long.toString(totalChoices()) + "_" +
393            java.util.Arrays.toString(histChoices()) + "_" +
394            Double.toString(w) + "_" +
395            Double.toString(wnef(SQUARE_EASE_WEIGHTS)) + "_" +
396            Double.toString(wnef(LINEAR_EASE_WEIGHTS)) + "_" +
397            Integer.toString(d.DIFFICULTY_INDEX) + "_" +
398            Double.toString(d.pdf(w));
399 }
400
401 @Override
402 public String toString() {
403     StringBuffer sb = new StringBuffer();
404
405     for (int[] row: cells) {
406         for (int c: row) {
407             sb.append(c);
408             sb.append("_");
409         }
410
411         sb.append("\n");
412     }
413
414     return sb.toString();
415 }
416
417 }
418
419 }
420
421 }
422
423 }

```

Listing 2: Command-line interface for `Puzzle` class.

```

1  /*
2  * Main.java: Provides data for Puzzle.java.
3  */
4

```

```
5 package sudokumetricizer;
6
7 import java.io.BufferedReader;
8 import java.io.FileReader;
9 import java.io.IOException;
10 import java.io.InputStreamReader;
11 import java.util.Iterator;
12 import java.util.logging.Level;
13 import java.util.logging.Logger;
14
15 public class Main {
16
17     public static void main(String[] args) throws IOException {
18
19         if (args.length == 0) {
20             System.out.println(
21                 "Order_of_metrics:\n" +
22                 "\tNumber_of_blanks.\n" +
23                 "\tTotal_number_of_choices.\n" +
24                 "\tExponential_weighted_NEF.\n" +
25                 "\tSquared_weighted_NEF.\n" +
26                 "\tLinear_weighted_NEF.\n" +
27                 "\tEstimated_difficulty_index.\n" +
28                 "\tPDF_used_to_estimate_difficulty.\n");
29             System.exit(0);
30         }
31
32         if (args[0].trim().equals("")) {
33
34             int[] linear_cells = new int[args.length-1];
35
36             for (int i = 1; i < args.length; i++) {
37                 linear_cells[i - 1] = Integer.parseInt(args[i]);
38             }
39
40             printPuzzle(new Puzzle(linear_cells));
41
42             System.exit(0);
43
44         } else if (args[0].trim().equals("--qqwing")) {
45
46             for (int i = 1; i < args.length; i++) {
47
48                 String filename = args[i];
49                 Iterator<int[]> linearFile = readLinearCells(filename);
50                 int j = 0;
51
52                 while (linearFile.hasNext()) {
53                     int[] linear_cells = linearFile.next();
54                     System.out.print(truncateFilename(filename) + ":" + j + "_");
55                     printPuzzle(new Puzzle(linear_cells));
56                     j++;
57                 }
58
59             }
60
61             System.exit(0);
62
63         }
64
65         for (String filename: args) {
```

```

67         if (filename.trim().equals("-")) {
68             System.out.print("stdin_");
69             printPuzzle(new Puzzle(new InputStreamReader(System.in)));
70         } else {
71             System.out.print(truncateFilename(filename) + "_");
72             printPuzzle(new Puzzle(new FileReader(filename)));
73         }
74     }
75
76 }
77
78 }
79
80 private static String truncateFilename(String str) {
81
82     // Find the position of the second-to-last slash.
83     int pos_from = str.lastIndexOf("/", str.lastIndexOf("/") - 1);
84
85     return str.substring(pos_from + 1);
86
87 }
88
89 private static void printPuzzle(Puzzle p) {
90     try {
91         System.out.println(p.metricsString());
92     } catch (RuntimeException rex) {
93         System.out.println();
94         System.err.println("Failed.");
95     }
96 }
97
98 private static Iterator<int[]> readLinearCells(String filename)
99     throws IOException
100 {
101
102     final BufferedReader br = new BufferedReader(new FileReader(filename));
103
104     // Throw away the first line.
105     br.readLine();
106
107     return new Iterator<int[]>() {
108
109         public boolean hasNext() {
110             try {
111                 return br.ready();
112             } catch (IOException ex) {
113                 Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
114                 return false;
115             }
116         }
117
118         public int[] next() {
119             try {
120                 int[] linear_cells = new int[81];
121                 String line = br.readLine();
122                 for (int i = 0; i < 81; i++) {
123                     try {
124                         linear_cells[i] = Integer.parseInt(line.substring(i, i+1));
125                     } catch (NumberFormatException ex) {
126                         linear_cells[i] = 0;
127                     }
128                 }

```

```

129         return linear_cells;
130     } catch (IOException ex) {
131         Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
132         return null;
133     }
134 }
135
136 public void remove() {
137     throw new UnsupportedOperationException("Read-only_iterator.");
138 }
139
140 };
141
142 }
143
144 }

```

Listing 3: Implementation of generation algorithm.

```

1 <?php
2 include( "tuning.php" );
3
4 set_time_limit( 45 );
5 /*
6  * This header file contains all operations associated with the
7  * generation and ranking of Sudoku puzzles
8  */
9
10 // This class keeps track of the times spent in each function
11 $profile_data = array();
12 class Profiler
13 {
14     var $time;
15     var $_line;
16     var $_function;
17     function __construct($f, $l)
18     {
19         $this->_function = $f;
20         $this->_line = $l;
21         $this->time = microtime(true);
22     }
23
24     function __destruct()
25     {
26         global $profile_data;
27
28         $end_time = microtime(true);
29         $dtime = ($end_time-$this->time);
30         $str = "$this->_line:_"$this->_function";
31         if( !isset( $profile_data[ $str ] ) ) $profile_data[ $str ] = $dtime;
32         else $profile_data[ $str ] += $dtime;
33         $str .= "_#called";
34         if( !isset( $profile_data[ $str ] ) ) $profile_data[ $str ] = 1;
35         else $profile_data[ $str ] ++;
36     }
37 }
38
39 // This function normalizes php array keys, such that {1=>x, 2=>y..} shall become {0=>x,
40 // 1=>y,...}
41 function NormalizeKeys( $array )
42 {
43     return array_values( $array );
44 }

```



```

43     }
44
45     // This function converts a wnef to a string difficulty
46     function MakeDifficulty( $wnef )
47     {
48         if( $wnef > .28 ) return "Easy";
49         if( $wnef > .2250 ) return "Medium";
50         if( $wnef > .18 ) return "hard";
51         return "Evil";
52     }
53
54     // Shuffles an array without messing with key value pair association
55     // from: http://us2.php.net/shuffle
56     // user: "rich"
57     function shuffle_assoc(&$array)
58     {
59         if (count($array)>1)    //$keys needs to be an array, no need to shuffle 1 item
60             anyway
61         {
62             $keys = array_rand($array, count($array));
63
64             foreach($keys as $key) $new[$key] = $array[$key];
65
66             $array = $new;
67         }
68         return true; //because it's a wannabe shuffle(), which returns true
69     }
70
71     // This class contains all the algorithms and information regarding a Sudoku puzzle
72     class Sudoku
73     {
74
75         //
76         *****
77
78         // vars
79
80         // this is a list of all valid numbers a Sudoku cell may be set to
81         var $numbers = array( 1, 2, 3, 4, 5, 6, 7, 8, 9 );
82
83         // this is a two dimensional array storing a solved Sudoku puzzle
84         var $board = array();
85
86         // this is a two dimensional array indicating which board spots are given at the
87         start of a game
88         var $mask = array();
89
90         // array of choices available for each cell
91         var $choices = array();
92
93
94         //
95         *****
96
97         // Utility functions
98
99         function A( $c, $i ) { return floor($c/3)*3+floor($i/3); }
100        function B( $c, $i ) { return (intval($c)%3)*3 + intval($i)%3; }

```

```

99     function C( $a, $b ) { return floor($b/3)+floor($a/3)*3; }
100     function I( $a, $b ) { return intval($b)%3+(intval($a)%3)*3; }
101
102     // this function returns indices of all cells in a given cluster
103     function ClusterCandidates( $c )
104     {
105         $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
106
107         static $clusters = array(
108             array(
109                 array(0,0), array(0,1), array(0,2),
110                 array(1,0), array(1,1), array(1,2),
111                 array(2,0), array(2,1), array(2,2)
112             ),
113             array(
114                 array(0,3), array(0,4), array(0,5),
115                 array(1,3), array(1,4), array(1,5),
116                 array(2,3), array(2,4), array(2,5)
117             ),
118             array(
119                 array(0,6), array(0,7), array(0,8),
120                 array(1,6), array(1,7), array(1,8),
121                 array(2,6), array(2,7), array(2,8)
122             ),
123
124             array(
125                 array(3,0), array(3,1), array(3,2),
126                 array(4,0), array(4,1), array(4,2),
127                 array(5,0), array(5,1), array(5,2)
128             ),
129             array(
130                 array(3,3), array(3,4), array(3,5),
131                 array(4,3), array(4,4), array(4,5),
132                 array(5,3), array(5,4), array(5,5)
133             ),
134             array(
135                 array(3,6), array(3,7), array(3,8),
136                 array(4,6), array(4,7), array(4,8),
137                 array(5,6), array(5,7), array(5,8)
138             ),
139
140             array(
141                 array(6,0), array(6,1), array(6,2),
142                 array(7,0), array(7,1), array(7,2),
143                 array(8,0), array(8,1), array(8,2)
144             ),
145             array(
146                 array(6,3), array(6,4), array(6,5),
147                 array(7,3), array(7,4), array(7,5),
148                 array(8,3), array(8,4), array(8,5)
149             ),
150             array(
151                 array(6,6), array(6,7), array(6,8),
152                 array(7,6), array(7,7), array(7,8),
153                 array(8,6), array(8,7), array(8,8)
154             )
155         );
156
157         return $clusters[$c];
158     }
159
160     // this function returns indices of all cells in a given row

```

```

161 function RowCandidates( $a )
162 {
163     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
164
165     // remember our cluster
166     $row = array();
167     for( $b=0; $b<9; $b++ )
168     {
169         $row[] = array( $a, $b );
170     }
171     return $row;
172 }
173
174 // this function returns indices of all columns in a given row
175 function ColCandidates( $b )
176 {
177     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
178
179     // remember our cluster
180     $col = array();
181     for( $a=0; $a<9; $a++ )
182     {
183         $col[] = array( $a, $b );
184     }
185     return $col;
186 }
187
188 // returns the number of values not hidden by the given mask
189 function NumValues( $our_mask )
190 {
191     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
192
193     $num = 0;
194     foreach( $our_mask as $g2 )
195     {
196         foreach( $g2 as $g )
197         {
198             if( $g == 1 ) $num++;
199         }
200     }
201     return $num;
202 }
203
204
205
206
207
208
209 //
210
211
212
213 // creates a string representation of the board given a mask
214 // this representation shall replace any hidden value with a 0
215 function GetPuzzleString( $our_mask )
216 {
217     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
218
219     $puzzle_string = "";
220     foreach( $this->board as $k1=>$a )

```

```

221     {
222         foreach( $a as $k2=>$b )
223         {
224             // only add to puzzle file if this is a given cell, else write 0
225             if( $our_mask[$k1][$k2] )      $puzzle_string .= "$b_";
226             else                            $puzzle_string .= "0_";
227         }
228     }
229     return $puzzle_string;
230 }
231
232
233 // Writes this puzzle to a file given an integer id
234 // "samples/s$number.txt" is the solved puzzle
235 // "samples/b$number.txt" is the initial puzzle
236 function ToFile( $number )
237 {
238     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
239
240     // contents of solution file
241     $file_string_s = "";
242
243     // contents of puzzle file
244     $file_string_b = "";
245
246     // convert board to string
247     foreach( $this->board as $k1=>$a )
248     {
249         foreach( $a as $k2=>$b )
250         {
251             $file_string_s .= $b . "_";
252
253             // only add to puzzle file if this is a given cell, else write 0
254             if( $this->mask[$k1][$k2] )      $file_string_b .= "$b_";
255             else                            $file_string_b .= "0_";
256         }
257
258         $file_string_s .= "\r\n";
259         $file_string_b .= "\r\n";
260     }
261
262     // output files
263     file_put_contents( "samples/s$number.txt", $file_string_s );
264     file_put_contents( "samples/b$number.txt", $file_string_b );
265 }
266
267
268 // Reads this puzzle from a file given an integer id
269 // "samples/s$number.txt" is the solved puzzle
270 // "samples/b$number.txt" is the initial puzzle
271 function FromFile( $number )
272 {
273     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
274
275     $file_strings_s = file( "samples/s$number.txt" );
276     $file_strings_b = file( "samples/b$number.txt" );
277
278     foreach( $file_strings_s as $key => $val )
279     {
280         $this->board[$key] = explode( "_", $val );
281     }
282

```

```

283     foreach( $file_strings_b as $key => $val )
284     {
285         $gs = explode( "_", $val );
286         foreach( $gs as $kg => $g )
287         {
288             if( $g )         $this->mask[$key][$kg] = 1;
289             else             $this->mask[$key][$kg] = 0;
290         }
291     }
292 }
293
294
295 // Saves a loaded control puzzle back to the given file
296 // This is usefull for file type conversion
297 function StoreControlPuzzle( $fname )
298 {
299     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
300
301     // contents of puzzle file
302     $file_string = "";
303
304     // convert board to string
305     foreach( $this->board as $k1=>$a )
306     {
307         foreach( $a as $k2=>$b )
308         {
309             $file_string .= $b . "_";
310         }
311
312         $file_string .= "\r\n";
313     }
314
315     // output files
316     file_put_contents( $fname, $file_string );
317 }
318
319
320 // Loads a control puzzle so that we may examin it
321 // Is flexible to support differing ways of storing Sudoku data
322 function LoadControlPuzzle( $path )
323 {
324     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
325
326     $file_strings = file( $path );
327
328     foreach( $file_strings as $key => $val )
329     {
330         $line = str_split( $val );
331         $i = 0;
332         foreach( $line as $l )
333         {
334             if( $l == "." || $l == "-" ) $l = 0;
335             if( !is_numeric( $l ) ) continue;
336             $this->board[$key][] = $l;
337             $i++;
338             if( $i >= 9 ) break;
339         }
340     }
341
342     foreach( $this->board as $key1=>$val1 )
343     {
344         foreach( $val1 as $key2=>$val2 )

```

```

345     {
346         if( !is_numeric( $val2 ) ) unset( $this->board[$key1][$key2] );
347         else
348         {
349             if( $val2 == 0 )      $this->mask[$key1][$key2] = 0;
350             else                  $this->mask[$key1][$key2] = 1;
351         }
352     }
353     $this->board[$key1] = NormalizeKeys( $this->board[$key1] );
354     $this->mask[$key1] = NormalizeKeys( $this->mask[$key1] );
355 }
356
357 $this->RenderPuzzle( $this->board, $this->mask );
358 }
359
360
361 // Outputs the puzzle to the screen in a simple debug fassion
362 function RenderPuzzle( $our_board, $our_mask )
363 {
364     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
365
366     echo "<table_border=\"1\"_v-align=\"center\">";
367     foreach( $our_board as $k1=>$val1 )
368     {
369         echo "<tr>";
370         foreach( $val1 as $k2=>$val2 )
371         {
372             echo "<td_width=\"60px\"_height=\"60px\"_><center>";
373             if( $our_mask[$k1][$k2] == 1 ) echo "<b>$val2</b>";
374             else
375             {
376                 echo "<small>--</small>";
377             }
378             echo "</center></td>";
379         }
380         echo "</tr>";
381     }
382     echo "</table>";
383     if( $this->ValidateBoard( $our_board ) )      echo "valid<br_/>";
384     else                                          echo "I_N_V_A_L_I_D<br_/>";
385 }
386
387
388
389
390
391
392 *****
393
394 // Complete board generation
395
396 // This function performs a backtracking algorithm that fills in the given line and
397 recursively all following lines
398 // with valid numbers.
399 // $line: the current line number
400 // $clusters: the values in the current three clusters so far
401 // $cols: the values in the 9 columns so far
402 function FillLines( $line, $clusters, $cols )
403 {
404     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

```

```

404
405 // save our current state
406 $our_numbers = $this->numbers;
407 $our_clusters = $clusters;
408 $our_cols = $cols;
409
410 // base condition
411 if( $line >= 9 ) return true;
412
413 // shuffle the valid numbers list
414 shuffle( $our_numbers );
415
416 // keep track of the numbers remaining
417 $numbers_left = $our_numbers;
418
419 // keep track of our current column
420 $index = 0;
421
422 // keep track of numbers that we triad but failed to place
423 $slack = array();
424
425 // keep track of how many times we shifted the numbers array to try a new
    sequence
426 $num_shifts = 0;
427
428 // now let's try to place the numbers 1..9 into this row
429 while( true )
430 {
431     // grab the next number
432     $number = array_pop( $numbers_left );
433
434     // if this number is not in our current cluster and not in our current column
        then we are good to go
435     if( !in_array( $number, $our_clusters[ floor($index/3) ] ) && !in_array(
        $number, $our_cols[$index] ) )
436     {
437         // place the number into the board
438         $this->board[$line][$index] = $number;
439
440         // keep track of the addition to this cluster
441         $our_clusters[ floor($index/3) ][] = $number;
442
443         // keep track of the addition to this column
444         $our_cols[$index][] = $number;
445
446         // move on to the next column
447         $index++;
448
449         // add any slack numbers to the numbers we have left
450         foreach( $slack as $s ) $numbers_left[] = $s;
451
452         // clear the slack numbers
453         $slack = array();
454     }
455     else
456     {
457         // no good, add this number to slack, and move on to the next
458         $slack[] = $number;
459     }
460
461     // if we have covered all columns
462     if( $index >= 9 )

```

```

463     {
464         // if we are moving to the next group of three lines, then clear the
465         // clusters, as we are now leaving them
466         if( intval( $line+1 )%3 == 0 )    $nclusters = array( array(), array(),
467             array() );
468         // else keep the same clusters
469         else                               $nclusters = $our_clusters;
470
471         // recurse
472         if( $this->FillLines( $line+1, $nclusters, $our_cols ) )    return true;
473     }
474
475     // remember, numbers may be in slack, and so this can happen even when we are
476     // not done
477     if( count( $numbers_left ) == 0 )
478     {
479         // if we have shifted as far as we can, then just give up
480         if( $num_shifts == 9 )    return false;
481
482         // else let's try this line over again
483         unset( $this->board[$line] );
484
485         // recall our data
486         $our_cols = $cols;
487         $our_clusters = $clusters;
488
489         // cycle the numbers
490         $numbers_left = $our_numbers;
491         array_shift( $numbers_left );
492         $numbers_left[] = $our_numbers[0];
493         $our_numbers = $numbers_left;
494
495         // reset the column
496         $index = 0;
497
498         // reset the slack
499         $slack = array();
500
501         // keep track of the number of times we do this
502         $num_shifts++;
503     }
504 }
505
506 // Fills in the board with valid Sudoku numbers
507 function FillBoard()
508 {
509     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
510
511     shuffle( $this->numbers );
512     $this->board = array();
513
514     // set the first line to random values
515     $this->board[] = $this->numbers;
516
517     // add these values to clusters and cols, these keep track of what numbers have
518     // been used
519     $clusters = array( array(), array(), array() );
520     for( $i=0; $i<3; $i++ )    $clusters[0][] = $this->board[0][$i];
521     for( $i=3; $i<6; $i++ )    $clusters[1][] = $this->board[0][$i];
522     for( $i=6; $i<9; $i++ )    $clusters[2][] = $this->board[0][$i];
523     $cols = array( array(), array(), array() );

```



```

521         array(), array(), array(),
522         array(), array(), array() );
523     for( $i=0; $i<9; $i++ )      $cols[$i][] = $this->board[0][$i];
524
525     // now fill in the other lines subject to this constraint
526     return ( $this->FillLines( 1, $clusters, $cols ) && $this->ValidateBoard( $this->
        board ) );
527 }
528
529
530
531
532
533
534
535 //
    *****

536 // Board Validation
537
538
539 // Tests if a board confirms to all Sudoku rules
540 function ValidateBoard( $board )
541 {
542     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
543
544     for( $c=0; $c<9; $c++ )
545     {
546         $cell = array();
547         for( $i=0; $i<9; $i++ )
548         {
549             $a = floor($c/3)*3+floor($i/3);
550             $b = (intval($c)%3)*3 + intval($i)%3;
551
552             if( in_array( $board[$a][$b], $cell ) )
553             {
554                 return false;
555             }
556             if( $board[$a][$b] != 0 ) $cell[] = $board[$a][$b];
557         }
558     }
559     for( $a=0; $a<9; $a++ )
560     {
561         $row = array();
562         for( $b=0; $b<9; $b++ )
563         {
564
565             if( in_array( $board[$a][$b], $row ) )
566             {
567                 return false;
568             }
569             if( $board[$a][$b] != 0 ) $row[] = $board[$a][$b];
570         }
571     }
572     for( $b=0; $b<9; $b++ )
573     {
574         $col = array();
575         for( $a=0; $a<9; $a++ )
576         {
577
578             if( in_array( $board[$a][$b], $col ) )
579             {

```

```

580         return false;
581     }
582     if( $board[$a][$b] != 0 ) $col[] = $board[$a][$b];
583 }
584 }
585
586 return true;
587 }
588
589
590
591
592
593 //
    *****
594 // Solver
595
596 // returns the local weighted normalized ease function of the entire board
597 function WNEF( $our_board, $our_mask, $num=-1 )
598 {
599     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
600
601     $weights = array( 256, 128, 64, 32, 16, 8, 4, 2, 1 );
602
603     $this->FindChoices( $our_board, $our_mask );
604     if( $num == -1 ) $num = $this->NumValues( $our_mask );
605     $num = 81-$num;
606
607     if( $num == 0 ) return 1.0;
608
609     $total = 0;
610     for( $a=0; $a<9; $a++ )
611     {
612         for( $b=0; $b<9; $b++ )
613         {
614             $count = count( $this->choices[$a][$b] );
615             if( $our_mask[$a][$b] == 0 && $count > 0 ) $total += $weights[ $count - 1
616                 ];
617         }
618     }
619     return $total / ($weights[0]*$num);
620 }
621
622 // returns an array including all unique choices between the given canidates
623 function FindUnique( $canidates )
624 {
625     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
626
627     $unique_spots = array( -2, -2, -2, -2, -2, -2, -2, -2, -2, -2);
628     $counts = array(0,0,0,0,0,0,0,0,0,0);
629     foreach( $canidates as $k=>$cell )
630     {
631         foreach( $this->choices[ $cell[0] ][ $cell[1] ] as $choice )
632         {
633             $unique_spots[$choice] = $k;
634             $counts[$choice] ++;
635         }
636     }
637     $unique = array();
638     $spot_counts = array();

```

```

639         foreach( $unique_spots as $k=>$u )
640         {
641             if( $counts[$k] == 1 )
642             {
643                 $unique[$k] = $u;
644                 if( isset( $spot_counts[$u] ) ) return false;
645                 $spot_counts[$u] = 1;
646             }
647         }
648
649         return $unique;
650     }
651
652     // Removes a choice from all the given candidates
653     function RemoveChoice( $candidates, $val )
654     {
655         $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
656
657         foreach( $candidates as $cell )
658         {
659             foreach( $this->choices[ $cell[0] ][ $cell[1] ] as $key => $choice )
660             {
661                 if( $choice == $val )
662                 {
663                     unset( $this->choices[ $cell[0] ][ $cell[1] ][$key] );
664                     break;
665                 }
666             }
667             $this->choices[ $cell[0] ][ $cell[1] ] = NormalizeKeys( $this->choices[ $cell
                [0] ][ $cell[1] ] );
668         }
669     }
670
671
672     // Find all choices for all cells in the board.
673     // $follow_mask: calculate choices even for unmasked cells
674     // $dependents: calculate the dependence instead of the choices
675     function FindChoices( $our_board, $our_mask, $follow_mask = true, $dependents = false
        )
676     {
677         $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
678
679         // clear the array
680         $this->choices = array();
681         for( $a=0; $a<9; $a++ )
682         {
683             $this->choices[ $a ] = array();
684             for( $b=0; $b<9; $b++ )
685             {
686                 $this->choices[ $a ][ $b ] = array();
687             }
688         }
689
690         // the values in this cluster that we know
691         $cluster = array();
692
693         // traverse clusters
694         for( $c=0; $c<9; $c++ )
695         {
696             $cluster[$c] = array();
697             // fill in the cluster values
698             for( $i=0; $i<9; $i++ )

```

```

699         {
700             $a = floor($c/3)*3+floor($i/3);
701             $b = (intval($c)%3)*3 + intval($i)%3;
702
703             if( $our_mask[$a][$b] ) $cluster[$c][] = $our_board[$a][$b];
704         }
705     }
706
707     // traverse cells
708     for( $a=0; $a<9; $a++ )
709     {
710         for( $b=0; $b<9; $b++ )
711         {
712             $c = floor($b/3)+floor($a/3)*3;
713
714             // if this place is not known
715             if( !$follow_mask || !$our_mask[$a][$b] )
716             {
717                 // find values along horizontal and vertical lines
718                 $lines = array();
719
720                 for( $d=0; $d<9; $d++ )
721                 {
722                     if( $our_mask[$a][$d] ) $lines[] = $our_board[$a][$d];
723                     if( $our_mask[$d][$b] ) $lines[] = $our_board[$d][$b];
724                 }
725
726                 // now go through and find all values not in the cluster or along the
727                 // lines
728                 if( !$dependents )
729                 {
730                     for( $d=1; $d<=9; $d++ )
731                     {
732                         if( !( in_array( $d, $cluster[$c] ) || in_array( $d, $lines ) ) )
733                         {
734                             $this->choices[$a][$b][] = $d;
735                         }
736                     }
737                 }
738                 else
739                 {
740                     $this->choices[$a][$b] = array_merge( $cluster[$c], $lines );
741                 }
742             }
743         }
744     }
745
746
747     // Set the given cell to the given value, fixing choices accordingly
748     function SetCell( $a, $b, $val, &$our_board, &$our_mask )
749     {
750         $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
751
752         // so let's take the move
753         $our_mask[$a][$b] = 1;
754         $our_board[$a][$b] = $val;
755
756         $c = $this->C( $a, $b );
757         $this->choices[$a][$b] = array();
758         $this->RemoveChoice( $this->ClusterCandidates($c), $val );

```

```

759     $this->RemoveChoice( $this->RowCanidates($a), $val );
760     $this->RemoveChoice( $this->ColCanidates($b), $val );
761 }
762
763 // Test if the given board is deterministic, aka has only one solution
764 function Unique( $our_board, $our_mask, $num, $brute_force=1 )
765 {
766     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
767
768     // if the board is solved, then it is uniquely solvable
769
770     $this->FindChoices( $our_board, $our_mask );
771     while( true )
772     {
773         if( $num >= 81 ) return true;
774
775         //$this->RenderPuzzle( $our_board, $our_mask );
776
777
778         // look for cells with just one choice
779         $done = false;
780         for( $a=0; $a<9 && !$done; $a++ )
781         {
782             for( $b=0; $b<9; $b++ )
783             {
784                 // if we only have one choice here
785                 if( count( $this->choices[$a][$b] ) == 1 )
786                 {
787                     // then we have a move
788                     $num++;
789                     $this->SetCell( $a, $b, $this->choices[$a][$b][0], $our_board,
790                                 $our_mask );
791
792                     // let's get out of this dang thing.... a wish for a goto to
793                     implement a deep continue
794                     $done = true;
795                     $counter = 0;
796                     break;
797                 }
798             }
799         }
800         if( $done ) continue;
801
802         // cluster
803         $done = false;
804         for( $c=0; $c<9; $c++ )
805         {
806             $unique = $this->FindUnique( $this->ClusterCanidates( $c ) );
807             if( $unique === false ) return false;
808             foreach( $unique as $k=>$u )
809             {
810                 $a = $this->A( $c, $u );
811                 $b = $this->B( $c, $u );
812
813                 // then we have a move
814                 $num++;
815                 $this->SetCell( $a, $b, $k, $our_board, $our_mask );
816
817                 // let's get out of this dang thing.... a wish for a goto to
818                 implement a deep continue
819                 $done = true;
820                 $counter = 0;

```

```

818         break;
819     }
820 }
821 if( $done ) continue;
822
823 // rows
824 $done = false;
825 for( $a=0; $a<9; $a++ )
826 {
827     $unique = $this->FindUnique( $this->RowCandidates( $a ) );
828     if( $unique === false ) return false;
829     foreach( $unique as $k=>$u )
830     {
831         $b = $u;
832
833         // then we have a move
834         $num++;
835         $this->SetCell( $a, $b, $k, $our_board, $our_mask );
836
837         // let's get out of this dang thing.... a wish for a goto to
            implement a deep continue
838         $done = true;
839         $counter = 0;
840         break;
841     }
842 }
843
844 if( $done ) continue;
845
846 // columns
847 $done = false;
848 for( $b=0; $b<9; $b++ )
849 {
850     $unique = $this->FindUnique( $this->ColCandidates( $b ) );
851     if( $unique === false ) return false;
852     foreach( $unique as $k=>$u )
853     {
854         $a = $u;
855
856         // then we have a move
857         $num++;
858         $this->SetCell( $a, $b, $k, $our_board, $our_mask );
859
860         // let's get out of this dang thing.... a wish for a goto to
            implement a deep continue
861         $done = true;
862         $counter = 0;
863         break;
864     }
865 }
866
867 if( $done ) continue;
868
869 // last resort
870 $least = 100;
871 $least_pos = array( -1, -1 );
872 $least_choices = array();
873 for( $a=0; $a<9; $a++ )
874 {
875     for( $b=0; $b<9; $b++ )
876     {
877         $n = count( $this->choices[$a][$b] );

```

```

878         if( $n != 0 && $n < $least )
879         {
880             $least = $n;
881             $least_pos = array( $a, $b );
882             $least_choices = $this->choices[$a][$b];
883         }
884     }
885 }
886
887 $result = false;
888 if( $brute_force > 0 )
889 {
890     foreach( $least_choices as $c )
891     {
892         $sour_mask[ $least_pos[0] ][ $least_pos[1] ] = 0;
893         $sour_board[ $least_pos[0] ][ $least_pos[1] ] = $c;
894         $r = $this->Unique( $sour_board, $sour_mask, $num+1, $brute_force-1 );
895         if( $r && $result )
896         {
897             $result = false;
898             break;
899         }
900         else if( $r ) $result = true;
901     }
902 }
903
904 // and that is that
905 return $result;
906 }
907
908
909 // Returns a cell to attempt to remove using random selection
910 // $anneal controls annealing by indicating the value in the grid that is associated
911 // with a "free" cell
912 function StrategyRandom( $sour_board, $sour_mask, $persistence, $counter, $anneal = 1 )
913 {
914     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
915
916     static $prev_value;
917
918     $spots = array();
919     for( $a=0; $a<9; $a++ )
920     {
921         for( $b=0; $b<9; $b++ )
922         {
923             if( $sour_mask[$a][$b] == $anneal ) $spots[] = array( $a, $b );
924         }
925     }
926     shuffle( $spots );
927
928     $sour_place = $spots[0];
929     if( isset( $spots[1] ) && $prev_value == $sour_place ) $sour_place = $spots[1];
930     $prev_value = $sour_place;
931     return $sour_place;
932 }
933
934 // Returns a cell attempting to remove cells without many choices
935 // $anneal controls annealing by indicating the value in the grid that is associated
936 // with a "free" cell
937 function StrategyCullLow( $sour_board, $sour_mask, $persistence, $counter, $anneal = 1
938 )
939 {

```

```

937     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
938
939     static $prev_value;
940
941     $this->FindChoices( $our_board, $our_mask, false );
942     $choice_rank = array();
943     for( $a=0; $a<9; $a++ )
944     {
945         for( $b=0; $b<9; $b++ )
946         {
947             if( $our_mask[$a][$b] == $anneal ) $choice_rank[$a*9+$b] = count( $this->
                choices[$a][$b] )+$persistance[$a][$b]/$counter;
948         }
949     }
950     shuffle_assoc( $choice_rank );
951     asort( $choice_rank );
952     $keys = array_keys( $choice_rank );
953
954     $our_place = array( intval($keys[0]/9), intval($keys[0])%9 );
955     if( isset( $keys[1] ) && $prev_value == $our_place ) $our_place = array( intval(
        $keys[1]/9), intval($keys[1])%9 );
956     $prev_value = $our_place;
957     return $our_place;
958 }
959
960 // Returns a cell attempting to remove cells WITH many choices
961 // $anneal controlls anealing by indicating the value in the grid that is associated
962 with a "free" cell
963 function StrategyCullHigh( $our_board, $our_mask, $persistance, $counter, $anneal = 1
    )
964 {
965     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
966
967     static $prev_value;
968
969     $this->FindChoices( $our_board, $our_mask, false );
970     $choice_rank = array();
971     for( $a=0; $a<9; $a++ )
972     {
973         for( $b=0; $b<9; $b++ )
974         {
975             if( $our_mask[$a][$b] == $anneal ) $choice_rank[$a*9+$b] = count( $this->
                choices[$a][$b] )+$counter/$persistance[$a][$b];
976         }
977     }
978     shuffle_assoc( $choice_rank );
979     arsort( $choice_rank );
980     $keys = array_keys( $choice_rank );
981
982     $our_place = array( intval($keys[0]/9), intval($keys[0])%9 );
983     if( isset( $keys[1] ) && $prev_value == $our_place ) $our_place = array( intval(
        $keys[1]/9), intval($keys[1])%9 );
984     $prev_value = $our_place;
985     return $our_place;
986 }
987
988 // Returns a cell attempting to remove cells in mostly filled clusters
989 // $anneal controlls anealing by indicating the value in the grid that is associated
990 with a "free" cell
991 function StrategyTrimCluster( $our_board, $our_mask, $persistance, $counter, $anneal
    = 1 )
992 {

```



```

991     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
992
993     $amounts = array();
994     for( $c=0; $c<9; $c++ )
995     {
996         $amounts[$c] = 0;
997         for( $i=0; $i<9; $i++ )
998         {
999             $a = $this->A( $c, $i );
1000             $b = $this->B( $c, $i );
1001
1002             if( $our_mask[$a][$b] == 1 ) $amounts[$c] += 1 + $counter/$persistence[$a]
                [$b];
1003         }
1004     }
1005     shuffle_assoc( $amounts );
1006     if( $anneal == 1 ) arsort( $amounts );
1007     else                asort( $amounts );
1008     $keys = array_keys( $amounts );
1009
1010     $vals = array( 0, 1, 2, 3, 4, 5, 6, 7, 8 );
1011     shuffle( $vals );
1012     $c = $keys[0];
1013     foreach( $vals as $v )
1014     {
1015         $a = $this->A( $c, $v );
1016         $b = $this->B( $c, $v );
1017
1018         if( $our_mask[$a][$b] == $anneal ) return array( $a, $b );
1019     }
1020
1021     return array( -1, -1 );
1022 }
1023
1024 // Returns a cell attempting to remove cells in mostly rows
1025 // $anneal controls annealing by indicating the value in the grid that is associated
1026 with a "free" cell
1027 function StrategyTrimRow( $our_board, $our_mask, $persistence, $counter, $anneal = 1
1028 )
1029 {
1030     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
1031
1032     $amounts = array();
1033     for( $a=0; $a<9; $a++ )
1034     {
1035         $amounts[$a] = 0;
1036         for( $b=0; $b<9; $b++ )
1037         {
1038             if( $our_mask[$a][$b] == 1 ) $amounts[$a] += 1 + $counter/$persistence[$a]
                [$b];;
1039         }
1040     }
1041     shuffle_assoc( $amounts );
1042     if( $anneal == 1 ) arsort( $amounts );
1043     else                asort( $amounts );
1044     $keys = array_keys( $amounts );
1045
1046     $vals = array( 0, 1, 2, 3, 4, 5, 6, 7, 8 );
1047     shuffle( $vals );
1048     $a = $keys[0];
1049     foreach( $vals as $v )
1050     {

```

```

1049         $b = $v;
1050
1051         if( $our_mask[$a][$b] == $anneal ) return array( $a, $b );
1052     }
1053
1054     return array( -1, -1 );
1055 }
1056
1057 // Returns a cell attempting to remove cells in mostly filled columns
1058 // $anneal controls annealing by indicating the value in the grid that is associated
1059 with a "free" cell
1060 function StrategyTrimCol( $our_board, $our_mask, $persistence, $counter, $anneal = 1
1061 )
1062 {
1063     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
1064
1065     $amounts = array();
1066     for( $b=0; $b<9; $b++ )
1067     {
1068         $amounts[$b] = 0;
1069         for( $a=0; $a<9; $a++ )
1070         {
1071             if( $our_mask[$a][$b] == 1 ) $amounts[$b] += 1 + $counter/$persistence[$a
1072                ][$b];;
1073         }
1074     }
1075     shuffle_assoc( $amounts );
1076     if( $anneal == 1 ) arsort( $amounts );
1077     else asort( $amounts );
1078     $keys = array_keys( $amounts );
1079
1080     $vals = array( 0, 1, 2, 3, 4, 5, 6, 7, 8 );
1081     shuffle( $vals );
1082     $b = $keys[0];
1083     foreach( $vals as $v )
1084     {
1085         $a = $v;
1086
1087         if( $our_mask[$a][$b] == $anneal ) return array( $a, $b );
1088     }
1089
1090     return array( -1, -1 );
1091 }
1092
1093 // Returns a cell attempting to remove cells with many dependents
1094 // $anneal controls annealing by indicating the value in the grid that is associated
1095 with a "free" cell
1096 function StrategyTrimDependents( $our_board, $our_mask, $persistence, $counter,
1097     $anneal = 1 )
1098 {
1099     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
1100
1101     static $prev_value;
1102
1103     $this->FindChoices( $our_board, $our_mask, false, true );
1104
1105     $amounts = array();
1106     for( $a=0; $a<9; $a++ )
1107     {
1108         for( $b=0; $b<9; $b++ )
1109         {
1110             if( $our_mask[$a][$b] == $anneal ) $amounts[$a*9+$b] = count( $this->

```

```

1106         choices[$a][$b] ) + $counter/$persistance[$a][$b];
1107     else
1108         $amounts[$a*9+$b] = 0;
1109     }
1110     shuffle_assoc( $amounts );
1111     arsort( $amounts );
1112     $keys = array_keys( $amounts );
1113     $sour_place = array( intval($keys[0]/9), intval($keys[0])%9 );
1114     if( isset( $keys[1] ) && $prev_value == $sour_place ) $sour_place = array( intval(
1115         $keys[1]/9), intval($keys[1])%9 );
1116     $prev_value = $sour_place;
1117     return $sour_place;
1118 }
1119 // Returns a cell attempting to remove cells that have many other existing of the
1120 // same value
1121 // $anneal controls annealing by indicating the value in the grid that is associated
1122 // with a "free" cell
1123 function StrategyTrimValues( $sour_board, $sour_mask, $persistance, $counter, $anneal =
1124     1 )
1125 {
1126     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
1127     $amounts = array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 );
1128     $places = array( array(), array(), array(), array(), array(), array(), array(),
1129         array(), array(), array() );
1130     for( $a=0; $a<9; $a++ )
1131     {
1132         for( $b=0; $b<9; $b++ )
1133         {
1134             if( $sour_mask[$a][$b] == $anneal )
1135             {
1136                 $amounts[$sour_board[$a][$b]] += $counter/$persistance[$a][$b];
1137                 $places[$sour_board[$a][$b]][] = array( $a, $b );
1138             }
1139         }
1140     }
1141     shuffle_assoc( $amounts );
1142     arsort( $amounts );
1143     $vals = array_keys( $amounts );
1144     $places = $places[ $vals[0] ];
1145     shuffle( $places );
1146     return $places[0];
1147 }
1148 // Fill in the mask
1149 function FillMask( $difficulty )
1150 {
1151     global $difficulty_levels;
1152     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
1153     if( $difficulty == 0 ) return $this->FillMaskRandom();
1154     $this->mask = array(
1155         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1156         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1157         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1158         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1159     );
1160 }

```

```

1162         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1163         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1164         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1165         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1166         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1167     );
1168     $this->persistence = array(
1169         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1170         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1171         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1172         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1173         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1174         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1175         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1176         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1177         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1178     );
1179
1180     // remove some
1181     $num = 81;
1182
1183     $total = 0;
1184     $count = 0;
1185
1186     // set tuning options
1187     $strategies      = $difficulty_levels[$difficulty]["strategies"];
1188     $delta_strategies = $difficulty_levels[$difficulty]["delta_strategies"];
1189     $delta_strategies_rate = $difficulty_levels[$difficulty]["delta_strategies_rate"];
1190
1191     $num_anneal_attempts = $difficulty_levels[$difficulty]["num_anneal_attempts"];
1192     $failed_max          = $difficulty_levels[$difficulty]["failed_max"];
1193     $wnef_min            = $difficulty_levels[$difficulty]["wnef_min"];
1194     $wnef_max            = $difficulty_levels[$difficulty]["wnef_max"];
1195     $run_cleanup         = $difficulty_levels[$difficulty]["run_cleanup"];
1196     $brute_force         = $difficulty_levels[$difficulty]["brute_force"];
1197
1198     $annealings = array(
1199         array( "Sudoku", "StrategyRandom" ),
1200         array( "Sudoku", "StrategyCullLow" ),
1201         array( "Sudoku", "StrategyTrimCluster" ),
1202         array( "Sudoku", "StrategyTrimRow" ),
1203         array( "Sudoku", "StrategyTrimCol" ),
1204         array( "Sudoku", "StrategyTrimDependents" ),
1205         array( "Sudoku", "StrategyTrimValues" ),
1206     );
1207
1208     $best_mask = $this->mask;
1209     $best_wnef = 1;
1210     $best_num  = 0;
1211     $wnef_first = 0;
1212     $persistence_timer = 1;
1213     $wnef = 1;
1214     for( $anneal_attempts=0; $anneal_attempts<$num_anneal_attempts; $anneal_attempts
1215         ++ )
1216     {
1217         $failed_count = 0;
1218         while( true )
1219         {
1220             shuffle($strategies);
1221             $spot = call_user_func( $strategies[0], $this->board, $this->mask, $this
1222                 ->persistence, $persistence_timer );
1223             // $persistence_timer += 1;

```

```

1221         if( $failed_count%$delta_strategies_rate == 0 )
1222         {
1223             $strategies = array_merge( $strategies , $delta_strategies );
1224         }
1225
1226         $a = $spot[0];
1227         $b = $spot[1];
1228
1229         // Sentinel value for no spot left
1230         if( $a == -1 ) break;
1231
1232         if( $this->mask[$a][$b] != 0 )
1233         {
1234             $this->mask[$a][$b] = 0;
1235             if( !$this->Unique( $this->board, $this->mask, $num-1, $brute_force )
1236                 )
1237             {
1238                 $this->mask[$a][$b] = 1;
1239
1240                 $this->persistence[$a][$b]++;
1241                 $failed_count++;
1242             }
1243             else
1244             {
1245                 $this->persistence[$a][$b] = 1;
1246                 $num-=1;
1247                 $failed_count = 0;
1248             }
1249         }
1250         else
1251         {
1252             $failed_count++;
1253         }
1254         $wnef = $this->WNEF( $this->board, $this->mask, $num );
1255         if( $wnef <= $wnef_min || $failed_count >= $failed_max ) break;
1256     }
1257     if( $wnef_first == 0 ) $wnef_first = $wnef;
1258
1259     if( $best_wnef > $wnef )
1260     {
1261         $best_mask = $this->mask;
1262         $best_wnef = $wnef;
1263         $best_num = $num;
1264     }
1265     else
1266     {
1267         $this->mask = $best_mask;
1268         $wnef = $best_wnef;
1269         $num = $best_num;
1270     }
1271
1272     if( $anneal_attempts >= $num_anneal_attempts && $wnef > $wnef_max )
1273         $num_anneal_attempts+=2;
1274
1275     if( $anneal_attempts < $num_anneal_attempts-1 )
1276     {
1277         $num_times = 1+rand()%3;
1278         for( $i=0; $i<$num_times; $i++ )
1279         {
1280             shuffle( $annealings );
1281             $spot = call_user_func( $annealings[0], $this->board, $this->mask,

```

```

1281         $this->persistence, $persistence_timer, 0 );
1282         $this->mask[$spot[0]][$spot[1]] = 1;
1283         $num += 1;
1284     }
1285     echo "\n";
1286 }
1287
1288
1289 // endgame
1290 if( $wnef > $run_cleanup )
1291 {
1292     $done = false;
1293     for( $a=0; $a<9 && !$done; $a++ )
1294     {
1295         for( $b=0; $b<9 && !$done; $b++ )
1296         {
1297             if( $this->mask[$a][$b] != 0 )
1298             {
1299                 $this->mask[$a][$b] = 0;
1300                 if( !$this->Unique( $this->board, $this->mask, $num-1, 1 ) )
1301                 {
1302                     $this->mask[$a][$b] = 1;
1303                 }
1304                 else
1305                 {
1306                     $num-=1;
1307                     $wnef = $this->WNEF( $this->board, $this->mask, $num );
1308                     if( $wnef < $wnef_min ) $done = true;
1309                 }
1310             }
1311         }
1312     }
1313 }
1314
1315 $wnef = $this->WNEF( $this->board, $this->mask, $num );
1316 return array( $wnef_first, $wnef );
1317 }
1318
1319 // Fills in a mask by successive removal of cells
1320 function FillMaskRandom()
1321 {
1322     $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
1323
1324
1325     $this->mask = array(
1326         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1327         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1328         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1329         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1330         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1331         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1332         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1333         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1334         array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
1335     );
1336
1337     // remove some
1338     $positions = array();
1339     for( $a=0; $a<9; $a++ )
1340     {
1341         for( $b=0; $b<9; $b++ )

```

```

1342         {
1343             $positions[] = array( $a, $b );
1344         }
1345     }
1346     shuffle( $positions );
1347
1348     $pos = 0;
1349     $num = 81;
1350
1351     $failed = count( $positions );
1352
1353     foreach( $positions as $key=>$pos )
1354     {
1355         $a = $pos[0];
1356         $b = $pos[1];
1357         $this->mask[ $a ][ $b ] = 0;
1358
1359         if( !$this->Unique( $this->board, $this->mask, $num-1 ) )
1360         {
1361             $this->mask[ $a ][ $b ] = 1;
1362         }
1363         else $num--;
1364     }
1365     $wnef = $this->WNEF( $this->board, $this->mask, $num );
1366     return array( $wnef, $wnef );
1367 }
1368 }
1369
1370 ?>

```

Listing 4: Script to render Sudoku puzzles.

```

1 <?php
2 include( 'sudoku.php' );
3
4 $puzzle = new Sudoku();
5
6 $d = 0;
7 if( isset( $_GET[ "d" ] ) ) $d = $_GET[ "d" ];
8
9 if( !isset( $_COOKIE[ "sudoku_board" ] ) )
10 {
11     /* Debug console
12     echo "<center><textarea rows=10 cols=80>";
13     */
14
15     if( !$puzzle->FillBoard() ) echo "failed";
16     $res = $puzzle->FillMask( $d );
17
18     $wnef = $res[1];
19     $difficulty = MakeDifficulty( $wnef );
20     /*
21     echo "\n\n\n";
22     print_r( $profile_data );
23     echo "</textarea></center>\n\n";
24     */
25
26 }
27 else
28 {
29
30     $puzzle->mask = array(

```

```

31         array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
32         array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
33         array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
34         array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
35         array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
36         array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
37         array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
38         array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
39         array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
40     );
41     $puzzle->board = $puzzle->mask;
42
43     $vals_a = explode( ":", $_COOKIE["sudoku_board"] );
44
45     $difficulty = $vals_a[81];
46     $wnef       = $vals_a[82];
47
48     unset( $vals_a[81] );
49     unset( $vals_a[82] );
50
51     foreach( $vals_a as $key => $n )
52     {
53         if( $n != 0 )
54         {
55             $i = intval($key);
56             $puzzle->mask[ $i/9 ][ $i%9 ] = 1;
57             $puzzle->board[ $i/9 ][ $i%9 ] = $n;
58         }
59     }
60 }
61 // set cookie
62 $cookie_vals = Array();
63 for( $a=0; $a<9; $a++ )
64 {
65     for( $b=0; $b<9; $b++ )
66     {
67         if( $puzzle->mask[ $a ][ $b ] )
68         {
69             $cookie_vals[ 2+$a*9+$b ] = $puzzle->board[ $a ][ $b ];
70         }
71         else
72         {
73             $cookie_vals[ 2+$a*9+$b ] = 0;
74         }
75     }
76 }
77
78 $cookie_vals[] = $difficulty;
79 $cookie_vals[] = $wnef;
80
81 setcookie( "sudoku_board", implode( ":", $cookie_vals ), time()+32000000 );
82
83 ?>
84 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
85     <head>
86         <title>Sudoku</title>
87
88         <script language="javascript" src="js-include/mootools-release-1.11.js"><!-- --></script>
89         <script language="javascript" src="script.js"><!-- --></script>
90     <style>
91         body

```



```
92     {
93         padding:          0;
94         margin:           0;
95     }
96
97     #difficulty
98     {
99         width:             100%;
100        text-align:        center;
101        font-size:         300%;
102        font-weight:       bold;
103        color:             #668;
104    }
105
106    #wnef
107    {
108        margin-top:         -1em;
109        margin-bottom:      1em;
110        width:             100%;
111        text-align:        center;
112        color:             #668;
113        font-size:         80%;
114    }
115
116    #board,
117    #board2
118    {
119        width: 1em;
120        height: 1em;
121
122        font-size:         20em;
123
124        margin: auto;
125
126        border-style:      solid;
127        border-width:      1px;
128        border-color:      blue;
129
130        background-color:  black;
131    }
132
133    .large_square
134    {
135        width: 32.4%;
136        height: 32.4%;
137
138        font-size: 32.4%;
139        float: left;
140
141        margin: .4%;
142
143        background-color: grey;
144    }
145
146    .small_square
147    {
148        width: 31.3%;
149        height: 31.3%;
150
151        font-size: 20%;
152        line-height: 160%;
153    }
```

```
154         float: left;
155
156         text-align: center;
157         vertical-align: center;
158
159         margin: 1%;
160
161         background-color: white;
162
163         cursor: pointer;
164     }
165
166     div.small_square:hover
167     {
168         background-color: #FAA;
169     }
170
171     .bad
172     {
173         color: #A11;
174     }
175
176     .static
177     {
178         color: #11A;
179
180         cursor: default;
181     }
182
183     #menu
184     {
185         text-align: center;
186         margin: 0.4em;
187     }
188
189     #menu a, #menu select
190     {
191         font-weight: bold;
192         color: #A48;
193
194
195         border-style: dotted;
196         border-width: 1px;
197         border-color: #D8A;
198
199         padding: 0.2em;
200
201         cursor: pointer;
202     }
203
204     #menu a:hover, #menu select:hover
205     {
206         color: #848;
207
208         border-style: solid;
209     }
210 </style>
211 </head>
212 <body>
213     <div id="difficulty"> <?php echo $difficulty; ?> </div>
214     <div id="wnef"> <?php echo number_format( $wnef, 3 ); ?> </div>
215     <div id="board">
```

```

216 <?php
217     // render
218     // $puzzle->Unique( $puzzle->mask, 80, false );
219     for( $c=0; $c<9; $c++ )
220     {
221         echo "<div_class=\"large_square\">";
222
223         for( $i=0; $i<9; $i++ )
224         {
225             $a = floor($c/3)*3+floor($i/3);
226             $b = (intval($c)%3)*3 + intval($i)%3;
227
228             $keep = $puzzle->mask[$a][$b];
229             echo "<div_class=\"small_square_\" . ($keep ? \"static\" : \"\") . \"_cell_\"$c_
                col_\"$b_row_\"$a\"_id=\"\"$c\".\"_\"$a\".\"_\"$b\">";
230
231             if( $keep )
232             {
233                 echo $puzzle->board[$a][$b];
234             }
235             else
236             {
237                 // echo "<small>\". $puzzle->board[$a][$b].\"</small>";
238             }
239
240             echo "</div>";
241         }
242         echo "</div>";
243     }
244 ?>
245 </div>
246 <div id="menu">
247     <select id="sel_difficulty">
248         <option value ="0">Random</option>
249         <option value ="1">Easy</option>
250         <option value ="2">Medium</option>
251         <option value ="3">Hard</option>
252         <option value ="4">Evil</option>
253     </select>
254     <a onclick="NewBoard()">New Puzzle</a> <a onclick="Clear()">Clear Puzzle</a>
255 </div>
256 </body>
257 </html>

```

Listing 5: Tuning parameters for generator algorithm.

```

1 <?php
2
3 $difficulty_levels = array(
4     1 => array(
5         "strategies" => array(
6             array( "Sudoku", "StrategyCullHigh" ),
7             ),
8         "delta_strategies" => array(),
9         "delta_strategies_rate" => 50,
10        "num_anneal_attempts" => 1,
11        "failed_max" => 5,
12        "wnef_min" => 0.32,
13        "wnef_max" => 0.35,
14        "run_cleanup" => 0.4,
15        "brute_force" => 0,
16    ),

```

```

17
18 2 => array(
19     "strategies"                => array(
20         array( "Sudoku", "StrategyRandom" )
21     ),
22     "delta_strategies"          => array(
23         array( "Sudoku", "StrategyRandom" )
24     ),
25     "delta_strategies_rate"     => 40,
26     "num_anneal_attempts"       => 5,
27     "failed_max"                => 2,
28     "wnef_min"                  => 0.28,
29     "wnef_max"                  => 0.28,
30     "run_cleanup"               => 0.28,
31     "brute_force"               => 0,
32 ),
33 3 => array(
34     "strategies"                => array(
35         array( "Sudoku", "StrategyTrimValues" ),
36         array( "Sudoku", "StrategyCullLow" ),
37         array( "Sudoku", "StrategyTrimCluster" ),
38         array( "Sudoku", "StrategyTrimRow" ),
39         array( "Sudoku", "StrategyTrimCol" ),
40         array( "Sudoku", "StrategyTrimDependents" ),
41     ),
42     "delta_strategies"          => array(
43         array( "Sudoku", "StrategyRandom" ),
44     ),
45     "delta_strategies_rate"     => 10,
46     "num_anneal_attempts"       => 10,
47     "failed_max"                => 3,
48     "wnef_min"                  => 0.2,
49     "wnef_max"                  => 0.2,
50     "run_cleanup"               => 0.2,
51     "brute_force"               => 0,
52 ),
53 4 => array(
54     "strategies"                => array(
55         array( "Sudoku", "StrategyTrimValues" ),
56         array( "Sudoku", "StrategyCullLow" ),
57         array( "Sudoku", "StrategyCullLow" ),
58         array( "Sudoku", "StrategyCullLow" ),
59         array( "Sudoku", "StrategyCullLow" ),
60         array( "Sudoku", "StrategyCullLow" ),
61         array( "Sudoku", "StrategyCullLow" ),
62         array( "Sudoku", "StrategyCullLow" ),
63         array( "Sudoku", "StrategyTrimCluster" ),
64         array( "Sudoku", "StrategyTrimRow" ),
65         array( "Sudoku", "StrategyTrimCol" ),
66         array( "Sudoku", "StrategyTrimDependents" ),
67     ),
68     "delta_strategies"          => array(
69         array( "Sudoku", "StrategyRandom" ),
70     ),
71     "delta_strategies_rate"     => 10,
72     "num_anneal_attempts"       => 100,
73     "failed_max"                => 3,
74     "wnef_min"                  => 0,
75     "wnef_max"                  => 0.10,
76     "run_cleanup"               => 0,
77     "brute_force"               => 2,
78 ),

```

```

79     );
80     ?>

```

Listing 6: Python script to extract GNOME Sudoku puzzles.

```

1  import sys
2  import getopt
3  from gnome_sudoku.sudoku_maker import SudokuMaker
4
5  def print_puzzles(sm, f, min_d, max_d):
6      puzzles = [sm.get_puzzle(d.calculate()) for d in sm.list_difficulties() if (d.calculate()
7          > min_d) and (d.calculate() < max_d) ]
8      for g,d in puzzles:
9          f.write(g.replace("_", "") + "\t" + d.value_string() + "\n")
10
11 shortargs = "e:m:h:v:w:"
12 longargs = ["easy=" "medium=" "hard=" "evil=" "writemode="]
13
14 def main(argv):
15     default = "controls/gnome-sudoku/gnome-sudoku-"
16     easypath = default + "easy"
17     medpath = default + "med"
18     hardpath = default + "hard"
19     evilpath = default + "evil"
20     writemode = "w"
21
22     opts, args = getopt.getopt(sys.argv[1:], shortargs, longargs)
23
24     for opt, arg in opts:
25         if opt in ("-e", "--easy"):
26             easypath = arg
27         if opt in ("-m", "--medium"):
28             medpath = arg
29         if opt in ("-h", "--hard"):
30             hardpath = arg
31         if opt in ("-v", "--evil"):
32             evilpath = arg
33         if opt in ("-w", "--writemode"):
34             writemode = arg
35
36     ef = open(easypath, writemode);
37     mf = open(medpath, writemode);
38     hf = open(hardpath, writemode);
39     vf = open(evilpath, writemode);
40
41     try:
42         sm = SudokuMaker(batch_size=10)
43     except exceptions.EOFError:
44         pass
45
46     sm.make_batch()
47
48     print_puzzles(sm, ef, 0.00, 0.25)
49     ef.close()
50
51     print_puzzles(sm, mf, 0.25, 0.50)
52     mf.close()
53
54     print_puzzles(sm, hf, 0.50, 0.75)
55     hf.close()
56
57     print_puzzles(sm, vf, 0.75, 1.00)

```

```

57     vf.close()
58
59
60 if __name__ == "__main__":
61     main(sys.argv)

```

2 Screenshots of Puzzle Generator

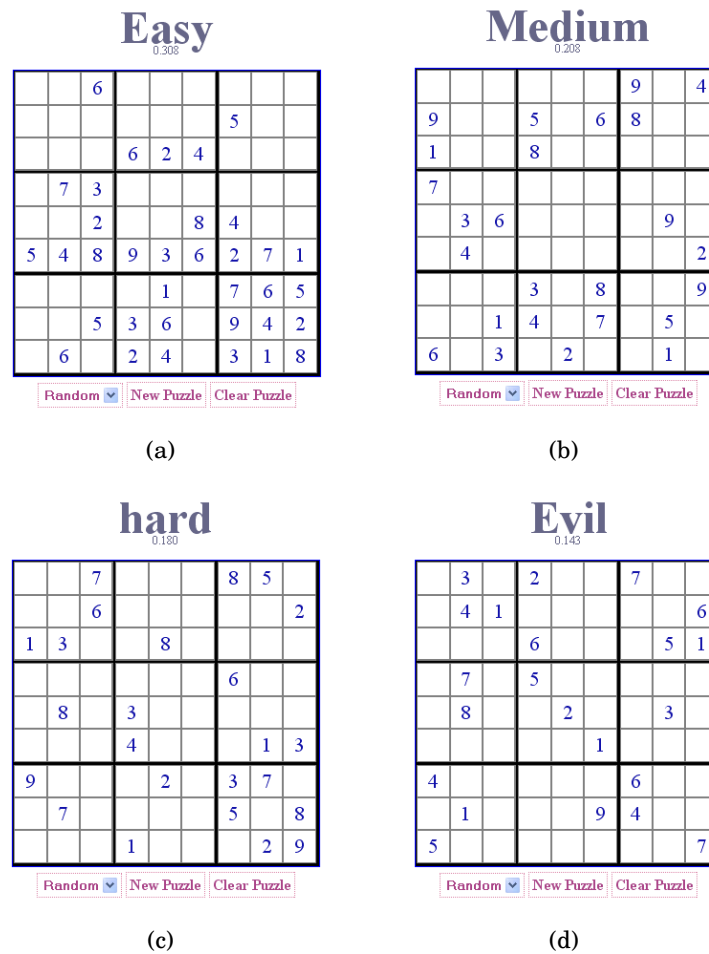


Figure 15: Screenshots of puzzle generator.