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

List of Figures

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 evil[1]).
- 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 $X \mapsto x$.

[1]This 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: $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$. We refer unambiguously to the row groupings $\mathbf{R}_i$, the column groupings $\mathbf{C}_j$ and the block groupings $\mathbf{B}_c$, following the indexing scheme in section 1.6. The set of all groupings will be denoted $\mathbb{G}$.

Metric We call a function $m : \mathbb{P} \to \mathbb{R}$ (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: $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{R}$. 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: $X \in \mathbf{P}$. 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 5$^{\text{th}}$ row and 5$^{\text{th}}$ column 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 $\mathbb{V} \backslash X?$.

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

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

Value A symbol that may be assigned to a cell. 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 indicies 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:

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

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

As an example of this arrangement is shown in figure 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 $\{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$. Then, by exhaustion, there is a one-to-one set of assignments from $\{X, Y, Z\}$ to $\{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$. Therefore, no other cell in $\mathbf{A}$ may map to a value in $\{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$.

An example 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.

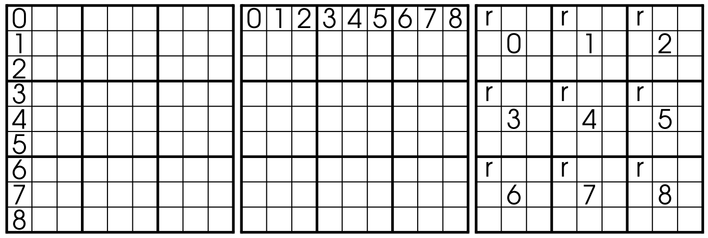


Figure 1: Demonstration of indexing schemes.

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

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

2.1.3 Hidden Pair

Informally, this rule is conjugate to the Naked Pair rule (section 2.1.1). Here, we also consider a single grouping $\mathbf{A}$ and two cells $X, Y \in \mathbf{A}$, but the condition is that there exist two values u and v such that at least one of $\{\mathsf{u}, \mathsf{v}\}$ is in each of $X?$ and $Y?$, but such that for any cell $Q \in (\mathbf{A} \setminus \{X, Y\})$, $\mathsf{u}, \mathsf{v} \notin Q?$. Thus, since $\mathbf{A}$ must contain a cell with each of the values, we can force $X?, Y? \subseteq \{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$.

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

2.1.4 Hidden Triplet

As with the Naked Pair rule (section 2.1.1), we can extend the Hidden Pair rule (section 2.1.3) so that it applies to three cells. In particular, let $\mathbf{A}$ be a grouping, and let $X, Y, Z \in \mathbf{A}$ be cells such that at least one of $\{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$ is in each of $X?$, $Y?$ and $Z?$ for some values t, u and v. Then, if for any other cell $Q \in (\mathbf{A} \setminus \{X, Y, Z\})$, $\mathsf{t}, \mathsf{u}, \mathsf{v} \notin Q?$, we claim that we can force $X?, Y?, Z? \subseteq \{\mathsf{t}, \mathsf{u}, \mathsf{v}\}$.

An example of this is shown in Figure 8, where in the grouping $\mathbf{R}_5$, only the cells marked X, Y and Z can take on the values of 1, 2 and 7. We would thus conclude that any candidate of X, Y or Z that is not either 1, 2 or 7 may be eliminated.

2.1.5 Multi-Line

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

An example of this rule is shown in Figure 9. In that figure, cells that we are interested in, and for which 5 is a candidate, are marked with an asterisk. We will be letting $a = 0$, $b = 6$, $c = 3$, $x = 0$, $y = 1$ and $z = 2$. Then, note that all of the asterisks for blocks 0 and 6 are located in the first two columns. Thus, in order to satisfy the constraint that a 5 appear in each of these blocks, block 0 must have a 5 in either column 0 or 1, while block 6 must have a 5 in the other column. This leaves only column 2 open for block 3, and so we can remove 5 from the candidate lists for all

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 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.

2.2.3 GNOME Sudoku

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 $c(X)$:

$$c(X) = |X?| \tag{1}$$

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 $\mathbf{P}$ which are empty:

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

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

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

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, $\mathrm{wef}(\mathbf{P})$, by convoluting the histogram with a weight function $w(n)$:

$$\mathrm{wef}(\mathbf{P}) = \sum_{n=1}^{9} w(n) \cdot c_n(\mathbf{P}) \tag{3}$$

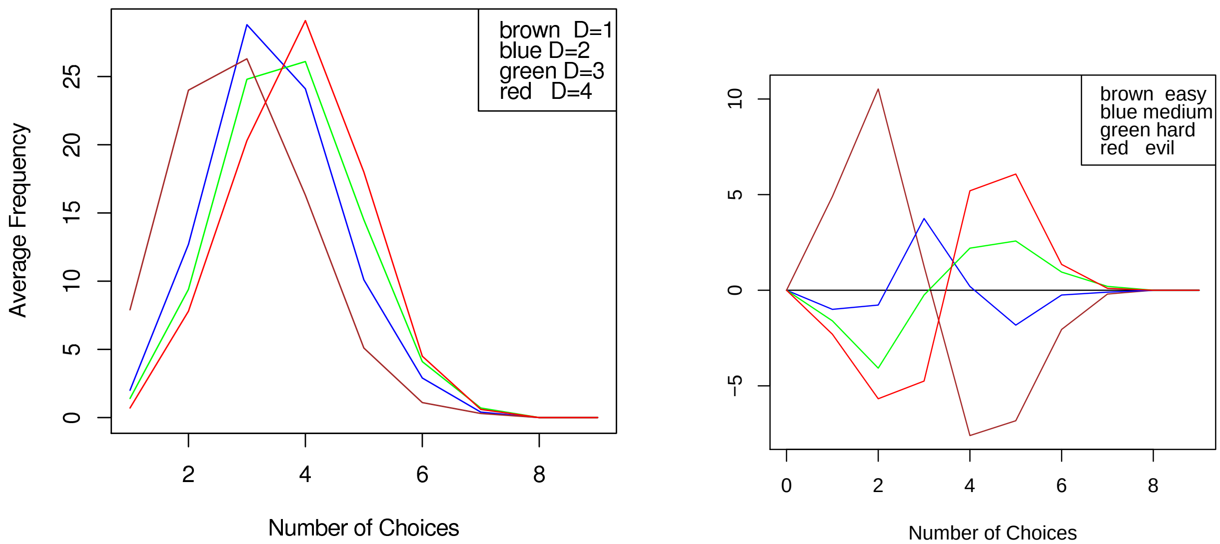
where $c_n(\mathbf{P})$ is the n^{th} value in the histogram $\vec{c}(\mathbf{P})$. This function, however, has the absurd trait that removing information from a puzzle results in more empty cell, which in turn causes the function to strictly increase. We therefore calculate the weighted and *normalized*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

$$\mathrm{wnef}(\mathbf{P}) = \frac{\mathrm{wef}(\mathbf{P})}{w(1) \cdot |E(\mathbf{P})|} \tag{4}$$

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 $\star$ puzzles.
  - 10 $\star\star$ puzzles.
  - 10 $\star\star\star$ puzzles.
  - 10 $\star\star\star\star$ puzzles.
- GNOME Sudoku [5]
  - 2000 Hard puzzles.
- “top2365” [2]
  - 2365 Evil puzzles.

[2]This 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 indicies 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 $\mathbf{P}_i$ included:

- $|E(\mathbf{P}_i)|$, the total number of empty cells in $\mathbf{P}_i$.
- $C(\mathbf{P}_i) = \sum_{X \in \mathbf{P}_i} X?$, the number of possible choices for all cells.
- 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 $\text{wnef}(\mathbf{P})$ for the control puzzles to try to further explore the relationship and found a clear negative correlation between the difficulty level of $\mathbf{P}$ and $\text{wnef}(\mathbf{P})$ for the control puzzles (Figure 11). This leads us to introduce $\overline{\text{wnef}}(d)$ 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 $\overline{\text{wnef}}(d) \neq \overline{\text{wnef}}(d+1)$ for $d \in \{1,2,3\}$, we conducted a hypothesis test for $d = 1,2,3$ with the following hypotheses:

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

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

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

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 the 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.

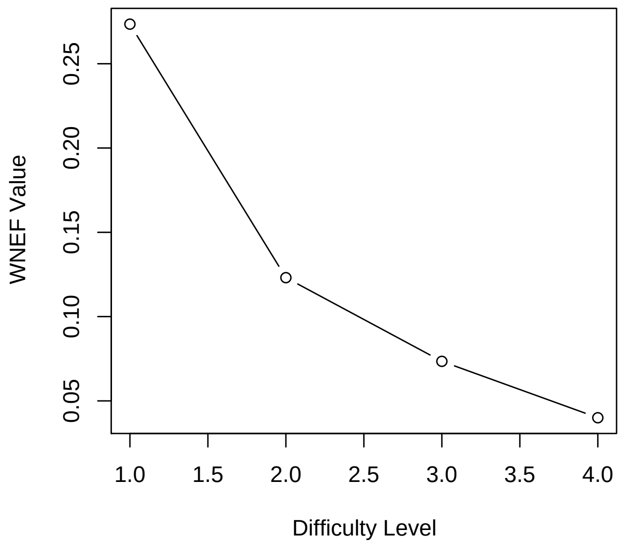


Figure 11: WNEF for control puzzles by difficulty.

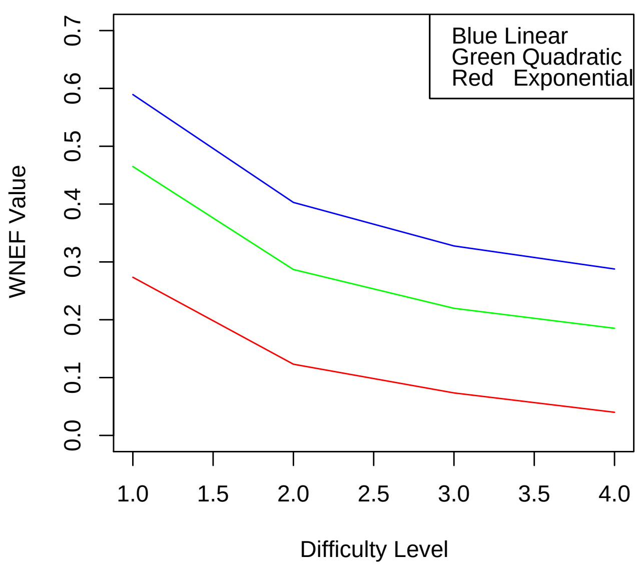


Figure 12: WNEF correlations for various weighting functions.

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 then 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 my 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 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 filed in with that value. These two logic processes are performed on a board until either the board is solved indicatng 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 then one value produces a unique result then the board has more then one solution. This solution calls itself recursively to determine the uniqueness of the board with the added values. The advantage of this 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 then 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. (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, $(9 \times 9 \times 9)^9 = \left(9^3\right)^9 = 9^{27} \approx 5.8 \times 10^{25}$. 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 as assume we only do this once per line. Here we get 9*9*9*9 or 6561. The actual value may be less then 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, it's 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:

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

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:

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

where $\overline{y}_d$ 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

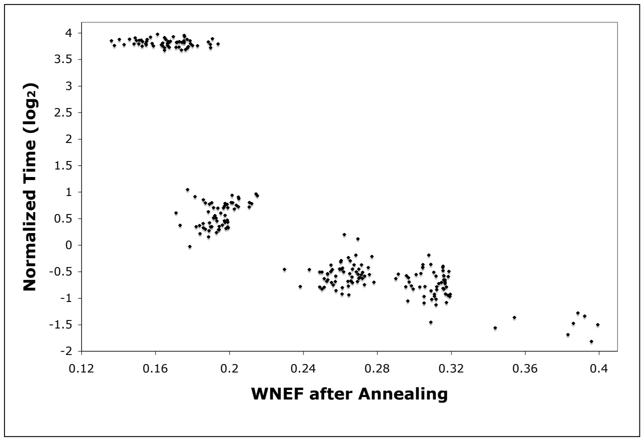


Figure 13: Running time as a function of the obtained WNEF.

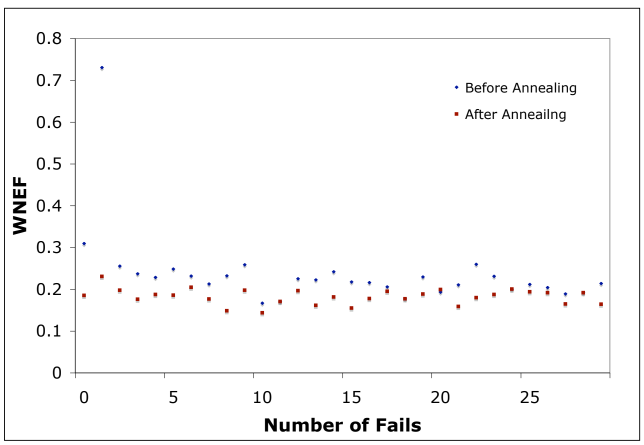


Figure 14: WNEF as a function of allowed failures.

Difficulty	1	2	3	4
		Pre-annealing		
Mean	0.523999895	0.327451814	0.271656591	0.27661661
Variance	0.017110796	0.005454866	0.002581053	0.004039649
		Post-annealing		
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 then 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.

```java
/*
 * Puzzle.java: Encapsulates most details about a puzzle.
 */

package sudokumetricizer;

import java.io.BufferedReader;
import java.io.Reader;
import java.util.Scanner;

public class Puzzle {

    /**
     * All values are calculated from the exponential weighting function.
     * See Section 4.2 for how these values were calculated,
     * and Table 1 for the actual values.
     */
    public static enum Difficulty {
        EASY        (1, 0.2680756,   0.00096963 ),
        MEDIUM      (2, 0.1108268,   0.000502135),
        HARD        (3, 0.09244832,  0.000255063),
        EVIL        (4, 0.04078146,  0.000125557);

        // For all of these fields, please see Section 4.2.
        public final double
                /**
                 * Estimate of the variance in the WNEF for puzzles of this
                 * difficulty.
                 */
                EST_VAR_WNEF,
                /**
                 * Estimate of the mean WNEF for puzzles of this difficulty.
                 */
                EST_MEAN_WNEF,
                /**
                 * Estimmate of the standard deviation for puzzles of this
                 * difficulty.
                 */
                EST_STDDEV_WNEF;

        /**
         * Numeric value that may be used in interprolation.
         */
        public final int DIFFICULTY_INDEX;

        Difficulty(int difficulty_index,
                   double est_mean_wnef,
                   double est_var_wnef) {
            DIFFICULTY_INDEX = difficulty_index;
            EST_VAR_WNEF = est_var_wnef;
            EST_MEAN_WNEF = est_mean_wnef;
            EST_STDDEV_WNEF = Math.sqrt(EST_VAR_WNEF);
        }

        /**
         * A useful numerical constant equal to 1/√2π.
         */
        public final static double
```

```
                ROOT_1OVER_2PI = Math.sqrt(1.0/(2.0*Math.PI));



        /*f (wnef = w | D = d) = 1/(2πσ̂²) exp{ −½σ̂² (w − μ̂)}*/









        public double pdf(double given_wnef) {
            double p = (1.0/EST_STDDEV_WNEF) * ROOT_1OVER_2PI *
                    Math.exp(
                        (-0.5 / EST_VAR_WNEF) *
                        Math.pow(given_wnef - EST_MEAN_WNEF, 2.0)
                    );
            return p;
        }

    }

    private final static int[] EXP_EASE_WEIGHTS =
        {256,128,64,32,16,8,4,2,1};

    private final static int[] LINEAR_EASE_WEIGHTS =
        {9,8,7,6,5,4,3,2,1};

    private final static int[] SQUARE_EASE_WEIGHTS =
        {81,64,49,36,25,16,9,4,1};

    private int[][] cells;

    /**
     * Builds a puzzle given its cells.
     */
    public Puzzle(int[][] cells) {
        this.cells = cells.clone();
    }

    /**
     * Builds a puzzle given its cells expressed in a one-dimensional array.
     */
    public Puzzle(int[] linear_cells) {
        this.cells = new int[9][9];
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                cells[i][j] = linear_cells[i*9+j];
            }
        }
    }

    /**
     * Builds up a puzzle by reading integers from a Reader object.
     */
    public Puzzle(Reader r) {

        int idx = 0;
        final int max = 81;

```

```
        cells = new int[9][9];

        Scanner scan = null;
        scan = new Scanner(new BufferedReader(r));

        while (scan.hasNext() && idx < max) {
            int next = scan.nextInt();
            cells[idx / 9][idx % 9] = next;
            idx++;
        }

    }

    /**
     * Counts the number of empty cells in the puzzle.
     */
    public int numEmptyCells() {

        int count = 0;

        for (int[] row: cells) {

            for (int c: row) {
                if (c == 0) {
                    count++;
                }
            }

        }

        return count;

    }

    /**
     * Returns the cluster number of the cell (i,j).
     */
    public int blockOf(int i, int j) {
        return (int) (Math.floor(j/3) + 3*Math.floor(i/3));
    }

    /**
     * Returns the row index of the block representative for the given block
     * index.
     */
    public int rowRepresentativeOf(int block) {
        return 3 * (int) Math.floor((double) block / 3.0);
    }

    /**
     * Returns the row index of the block representative for the cell with given
     * row and column indicies.
     */
    public int rowRepresentativeOf(int i, int j) {
        return rowRepresentativeOf(blockOf(i,j));
    }

    /**
     * Returns the column index of the block representative for the given block
     * index.
     */
    public int colRepresentativeOf(int cluster) {
```

```
        return 3 * (cluster % 3);
    }

    /**
     * Returns the column index of the block representative for the cell with
     * given row and column indicies.
     */
    public int colRepresentativeOf(int i, int j) {
        return colRepresentativeOf(blockOf(i,j));
    }

    /**
     * Finds constraints on a cell by examining other cells on the same row.
     *
     * @param constraints
     *     constraints[n] == true indicates that cell[i][j]
     *     cannot be (n + 1).
     */
    public void constrainCellByRow(int i, int j, boolean[] constraints) {

        for (int other_j = 0; other_j < cells[i].length; other_j++) {
            if (other_j != j && cells[i][other_j] != 0) {
                constraints[cells[i][other_j] - 1] = true;
            }
        }

    }

    /**
     * Finds constraints on a cell by examining other cells on the same column.
     *
     * @param constraints
     *     constraints[n] == true indicates that cell[i][j]
     *     cannot be (n + 1).
     */
    public void constrainCellByCol(int i, int j, boolean[] constraints) {

        for (int other_i = 0; other_i < cells.length; other_i++) {
            if (other_i != i && cells[other_i][j] != 0) {
                constraints[cells[other_i][j] - 1] = true;
            }
        }

    }

    /**
     * Finds constraints on a cell by examining other cells within the same
     * block.
     *
     * @param constraints
     *     constraints[n] == true indicates that cell[i][j]
     *     cannot be (n + 1).
     */
    public void constrainCellByCluster(int i, int j, boolean[] constraints) {

        int orig_i = rowRepresentativeOf(i,j),
            orig_j = colRepresentativeOf(i,j);

        final int lim_i = orig_i + 3, lim_j = orig_j + 3;

        for (int other_i = orig_i; other_i < lim_i; other_i++) {
            for (int other_j = orig_j; other_j < lim_j; other_j++) {
```

```
                if (other_i != i && other_j != j && cells[other_i][other_j] != 0) {
                    constraints[cells[other_i][other_j] - 1] = true;
                }
            }
        }

    }

    /**
     * Returns a histogram of the choices avaiable to each cell, as determined
     * by simple elimination.
     *
     * @returns
     *     An array c⃗ such that c_n is the number of cells with
     *     n+1 available choices.
     */
    public int[] histChoices() throws RuntimeException {

        int[] hist = new int[9];

        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                hist[numChoicesForCell(i, j) - 1]++;
            }
        }

        return hist;

    }

    /**
     * Counts the number of choices available for a given cell, as determined by
     * simple elimination.
     */
    public int numChoicesForCell(int i, int j) {

        int count = cells.length;

        boolean[] constraints = new boolean[cells.length];

        // Set everything to false.
        for (int idx = 0; idx < cells.length; idx++) {
            constraints[idx] = false;
        }

        constrainCellByRow(i, j, constraints);
        constrainCellByCol(i, j, constraints);
        constrainCellByCluster(i, j, constraints);

        // Count the number of restrictions.
        for (int idx = 0; idx < cells.length; idx++) {
            if (constraints[idx]) count--;
        }

        return count;

    }

    /**
     * Counts the total number of choices available to all empty cells on the
     * puzzle, as determined by simple elimination.
     */
```

```java
    public long totalChoices() {

        long count = 0;

        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (cells[i][j] == 0) {
                    count += numChoicesForCell(i, j);
                }
            }
        }

        return count;

    }

    /**
     * Evaluates the weighted normalized ease function for the puzzle, using the
     * exponential weight function.
     */
    public double wnef() {
        return wnef(EXP_EASE_WEIGHTS);
    }

    /**
     * Calculates the Weighted Normalized Ease Function.
     */
    public double wnef(int[] weights) {

        long count = 0;

        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (cells[i][j] != 0) {
                    count += weights[numChoicesForCell(i, j) - 1];
                }
            }
        }

        return (double) count / (double) (weights[0] * numEmptyCells());

    }

    /**
     * Estimates the difficulty class of the puzzle by finding which class gives
     * the highest value of the WNEF probability distribution function.
     *
     * This method effectively implements Equation ??.
     */
    public Difficulty estimatedDifficulty() {

        double w = wnef();
        double max_pdf = -1.0;
        Difficulty diff = null;

        for (Difficulty d: Difficulty.values()) {
            double last_pdf = d.pdf(w);
            if (last_pdf > max_pdf) {
                max_pdf = last_pdf;
                diff = d;
            }
        }
```

```java

        return diff;

    }

    /**
     * Returns a space-separated list of metrics. In order:
     *      - number of empty cells
     *      - total number of choices
     *      - the exponential wnef
     *      - the square wnef
     *      - the linear wnef
     *      - the estimated difficulty index
     *      - the value of the pdf used to find the estimated difficulty
     */
    public String metricsString() {

        String histStr = java.util.Arrays.toString(histChoices());
        histStr = histStr.substring(1,histStr.length() - 1);

        Difficulty d = estimatedDifficulty();

        double w = wnef(EXP_EASE_WEIGHTS);

        return Integer.toString(numEmptyCells()) + " " +
               Long.toString(totalChoices()) + " " +
               java.util.Arrays.toString(histChoices()) + " " +
               Double.toString(w) + " " +
               Double.toString(wnef(SQUARE_EASE_WEIGHTS)) + " " +
               Double.toString(wnef(LINEAR_EASE_WEIGHTS)) + " " +
               Integer.toString(d.DIFFICULTY_INDEX) + " " +
               Double.toString(d.pdf(w));
    }

    @Override
    public String toString() {

        StringBuffer sb = new StringBuffer();

        for (int[] row: cells) {

            for (int c: row) {
                sb.append(c);
                sb.append(" ");
            }

            sb.append("\n");

        }

        return sb.toString();

    }

}
```

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

```java
/*
 * Main.java: Provides data for Puzzle.java.
 */

```

```java
package sudokumetricizer;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Iterator;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Main {

    public static void main(String[] args) throws IOException {

        if (args.length == 0) {
            System.out.println(
                    "Order␣of␣metrics:\n" +
                    "\tNumber␣of␣blanks.\n" +
                    "\tTotal␣number␣of␣choices.\n" +
                    "\tExponential␣weighted␣NEF.\n" +
                    "\tSquared␣weighted␣NEF.\n" +
                    "\tLinear␣weighted␣NEF.\n" +
                    "\tEstimated␣difficulty␣index.\n" +
                    "\tPDF␣used␣to␣estimate␣difficulty.\n");
            System.exit(0);
        }

        if (args[0].trim().equals("--")) {

            int[] linear_cells = new int[args.length-1];

            for (int i = 1; i < args.length; i++) {
                linear_cells[i - 1] = Integer.parseInt(args[i]);
            }

            printPuzzle(new Puzzle(linear_cells));

            System.exit(0);

        } else if (args[0].trim().equals("--qqwing")) {

            for (int i = 1; i < args.length; i++) {

                String filename = args[i];
                Iterator<int[]> linearFile = readLinearCells(filename);
                int j = 0;

                while (linearFile.hasNext()) {
                    int[] linear_cells = linearFile.next();
                    System.out.print(truncateFilename(filename) + ":" + j + "␣");
                    printPuzzle(new Puzzle(linear_cells));
                    j++;
                }


            }

            System.exit(0);

        }

        for (String filename: args) {
```

```java

            if (filename.trim().equals("-")) {
                System.out.print("stdin ");
                printPuzzle(new Puzzle(new InputStreamReader(System.in)));
            } else {
                System.out.print(truncateFilename(filename) + " ");
                printPuzzle(new Puzzle(new FileReader(filename)));
            }

        }

    }

    private static String truncateFilename(String str) {

        // Find the position of the second-to-last slash.
        int pos_from = str.lastIndexOf("/", str.lastIndexOf("/") - 1);

        return str.substring(pos_from + 1);

    }

    private static void printPuzzle(Puzzle p) {
        try {
            System.out.println(p.metricsString());
        } catch (RuntimeException rex) {
            System.out.println();
            System.err.println("Failed.");
        }
    }

    private static Iterator<int[]> readLinearCells(String filename)
        throws IOException
    {

        final BufferedReader br = new BufferedReader(new FileReader(filename));

        // Throw away the first line.
        br.readLine();

        return new Iterator<int[]>() {

            public boolean hasNext() {
                try {
                    return br.ready();
                } catch (IOException ex) {
                    Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
                    return false;
                }
            }

            public int[] next() {
                try {
                    int[] linear_cells = new int[81];
                    String line = br.readLine();
                    for (int i = 0; i < 81; i++) {
                        try {
                            linear_cells[i] = Integer.parseInt(line.substring(i, i+1));
                        } catch (NumberFormatException ex) {
                            linear_cells[i] = 0;
                        }
                    }
```

```
                    return linear_cells;
                } catch (IOException ex) {
                    Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
                    return null;
                }
            }

            public void remove() {
                throw new UnsupportedOperationException("Read-only iterator.");
            }

        };

    }

}
```

Listing 3: Implementation of generation algorithm.

```
<?php
    include( "tuning.php" );

    set_time_limit( 45 );
    /*
    * This header file contains all operations associated with the
    * generation and ranking of Sudoku puzzles
    */

    // This class keeps track of the times spent in each function
    $profile_data = array();
    class Profiler
    {
        var $time;
        var $_line;
        var $_function;
        function __construct($f, $l)
        {
            $this->_function    = $f;
            $this->_line        = $l;
            $this->time = microtime(true);
        }

        function __destruct()
        {
            global $profile_data;

            $end_time = microtime(true);
            $dtime = ($end_time-$this->time);
            $str = "$this->_line: $this->_function";
            if( !isset( $profile_data[ $str ] ) )   $profile_data[ $str ] =  $dtime;
            else                                    $profile_data[ $str ] += $dtime;
            $str .= " #called";
            if( !isset( $profile_data[ $str ] ) )   $profile_data[ $str ] =  1;
            else                                    $profile_data[ $str ] ++;
        }
    }

    // This function normalizes php array keys, such that {1=>x, 2+>y..} shall become {0=>x,
        1=>y,...}
    function NormalizeKeys( $array )
    {
        return array_values( $array );
```

```
    }

    // This function converts a wnef to a string difficulty
    function MakeDifficulty( $wnef )
    {
        if( $wnef > .28 ) return "Easy";
        if( $wnef > .2250 ) return "Medium";
        if( $wnef > .18 ) return "hard";
        return "Evil";
    }

    // Shuffles an array withour messing with key value pair association
    // from: http://us2.php.net/shuffle
    // user: "rich"
    function shuffle_assoc(&$array)
    {
        if (count($array)>1)    //$keys needs to be an array, no need to shuffle 1 item anyway
        {
            $keys = array_rand($array, count($array));

            foreach($keys as $key)  $new[$key] = $array[$key];

            $array = $new;
        }
        return true; //because it's a wannabe shuffle(), which returns true
    }

    // This class contains all the algorithms and information regarding a Sudoku puzzle
    class Sudoku
    {


        // ****************************************************************************************

        // vars

        // this is a list of all valid numbers a Sudoku cell may be set to
        var $numbers = array( 1, 2, 3, 4, 5, 6, 7, 8, 9 );

        // this is a two dimensional array storing a solved Sudoku puzzle
        var $board = array();

        // this is a two dimensional array indicating which board spots are given at the start of a game
        var $mask = array();

        // array of choices available for each cell
        var $choices = array();





        // ****************************************************************************************

        // Utility functions

        function A( $c, $i ) { return floor($c/3)*3+floor($i/3); }
        function B( $c, $i ) { return (intval($c)%3)*3 + intval($i)%3; }
```

```
    function C( $a, $b ) { return floor($b/3)+floor($a/3)*3; }
    function I( $a, $b ) { return intval($b)%3+(intval($a)%3)*3; }

    // this function returns indices of all cells in a given cluster
    function ClusterCanidates( $c )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        static $clusters = array(
            array(
                array(0,0), array(0,1), array(0,2),
                array(1,0), array(1,1), array(1,2),
                array(2,0), array(2,1), array(2,2)
                ),
            array(
                array(0,3), array(0,4), array(0,5),
                array(1,3), array(1,4), array(1,5),
                array(2,3), array(2,4), array(2,5)
                ),
            array(
                array(0,6), array(0,7), array(0,8),
                array(1,6), array(1,7), array(1,8),
                array(2,6), array(2,7), array(2,8)
                ),

            array(
                array(3,0), array(3,1), array(3,2),
                array(4,0), array(4,1), array(4,2),
                array(5,0), array(5,1), array(5,2)
                ),
            array(
                array(3,3), array(3,4), array(3,5),
                array(4,3), array(4,4), array(4,5),
                array(5,3), array(5,4), array(5,5)
                ),
            array(
                array(3,6), array(3,7), array(3,8),
                array(4,6), array(4,7), array(4,8),
                array(5,6), array(5,7), array(5,8)
                ),

            array(
                array(6,0), array(6,1), array(6,2),
                array(7,0), array(7,1), array(7,2),
                array(8,0), array(8,1), array(8,2)
                ),
            array(
                array(6,3), array(6,4), array(6,5),
                array(7,3), array(7,4), array(7,5),
                array(8,3), array(8,4), array(8,5)
                ),
            array(
                array(6,6), array(6,7), array(6,8),
                array(7,6), array(7,7), array(7,8),
                array(8,6), array(8,7), array(8,8)
                )
            );

        return $clusters[$c];
    }

    // this function returns indices of all cells in a given row
```

```
        function RowCanidates( $a )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            // remember our cluster
            $row = array();
            for( $b=0; $b<9; $b++ )
            {
                $row[] = array( $a, $b );
            }
            return $row;
        }

        // this function returns indices of all columns in a given row
        function ColCanidates( $b )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            // remember our cluster
            $col = array();
            for( $a=0; $a<9; $a++ )
            {
                $col[] = array( $a, $b );
            }
            return $col;
        }

        // returns the number of values not hidden by the given mask
        function NumValues( $our_mask )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            $num = 0;
            foreach( $our_mask as $g2 )
            {
                foreach( $g2 as $g )
                {
                    if( $g == 1 ) $num++;
                }
            }
            return $num;
        }







        //
            ************************************************************************************
        // Loading and Storing functions


        // creates a string representation of the board given a mask
        // this representation shall replace any hidden value with a 0
        function GetPuzzleString( $our_mask )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            $puzzle_string = "";
            foreach( $this->board as $k1=>$a )
```

```
        {
            foreach( $a as $k2=>$b )
            {
                // only add to puzzle file if this is a given cell, else write 0
                if( $our_mask[$k1][$k2] )     $puzzle_string .= "$b␣";
                else                          $puzzle_string .= "0␣";
            }
        }
        return $puzzle_string;
    }


    // Writes this puzzle to a file given an integer id
    // "samples/s$number.txt" is the solved puzzle
    // "samples/b$number.txt" is the initial puzzle
    function ToFile( $number )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        // contents of solution file
        $file_string_s = "";

        // contents of puzzle file
        $file_string_b = "";

        // convert board to string
        foreach( $this->board as $k1=>$a )
        {
            foreach( $a as $k2=>$b )
            {
                $file_string_s .= $b . "␣";

                // only add to puzzle file if this is a given cell, else write 0
                if( $this->mask[$k1][$k2] )     $file_string_b .= "$b␣";
                else                            $file_string_b .= "0␣";
            }

            $file_string_s .= "\r\n";
            $file_string_b .= "\r\n";
        }

        // output files
        file_put_contents( "samples/s$number.txt", $file_string_s );
        file_put_contents( "samples/b$number.txt", $file_string_b );
    }


    // Reads this puzzle from a file given an integer id
    // "samples/s$number.txt" is the solved puzzle
    // "samples/b$number.txt" is the initial puzzle
    function FromFile( $number )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        $file_strings_s = file( "samples/s$number.txt" );
        $file_strings_b = file( "samples/b$number.txt" );

        foreach( $file_strings_s as $key => $val )
        {
            $this->board[$key] = explode( "␣", $val );
        }

```

```
            foreach( $file_strings_b as $key => $val )
            {
                $gs = explode( " ", $val );
                foreach( $gs as $kg => $g )
                {
                    if( $g )      $this->mask[$key][$kg] = 1;
                    else          $this->mask[$key][$kg] = 0;
                }
            }
        }


        // Saves a loaded control puzzle back to the given file
        // This is usefull for file type conversion
        function StoreControlPuzzle( $fname )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            // contents of puzzle file
            $file_string = "";

            // convert board to string
            foreach( $this->board as $k1=>$a )
            {
                foreach( $a as $k2=>$b )
                {
                    $file_string .= $b . " ";
                }

                $file_string .= "\r\n";
            }

            // output files
            file_put_contents( $fname, $file_string );
        }


        // Loads a control puzzle so that we may examin it
        // Is flexible to support differing ways of storing Sudoku data
        function LoadControlPuzzle( $path )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            $file_strings = file( $path );

            foreach( $file_strings as $key => $val )
            {
                $line = str_split( $val );
                $i = 0;
                foreach( $line as $l )
                {
                    if( $l == "." || $l == "-" ) $l = 0;
                    if( !is_numeric( $l ) ) continue;
                    $this->board[$key][] = $l;
                    $i++;
                    if( $i >= 9 ) break;
                }
            }

            foreach( $this->board as $key1=>$val1 )
            {
                foreach( $val1 as $key2=>$val2 )
```

```
                {
                    if( !is_numeric( $val2 ) ) unset( $this->board[$key1][$key2] );
                    else
                    {
                        if( $val2 == 0 )      $this->mask[$key1][$key2] = 0;
                        else                  $this->mask[$key1][$key2] = 1;
                    }
                }
                $this->board[$key1] = NormalizeKeys( $this->board[$key1] );
                $this->mask[$key1] = NormalizeKeys( $this->mask[$key1] );
            }

            $this->RenderPuzzle( $this->board, $this->mask );
        }


        // Outputs the puzzle to the screen in a simple debug fassion
        function RenderPuzzle( $our_board, $our_mask )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            echo "<table␣border=\"1\"␣v-align=\"center\">";
            foreach( $our_board as $k1=>$val1 )
            {
                echo "<tr>";
                foreach( $val1 as $k2=>$val2 )
                {
                    echo "<td␣width=\"60px\"␣height=\"60px\"␣><center>";
                    if( $our_mask[$k1][$k2] == 1 ) echo "<b>$val2</b>";
                    else
                    {
                        echo "<small>-</small>";
                    }
                    echo "</center></td>";
                }
                echo "</tr>";
            }
            echo "</table>";
            if( $this->ValidateBoard( $our_board ) )      echo "valid<br␣/>";
            else                                          echo "I␣N␣V␣A␣L␣I␣D<br␣/>";
        }






        // ************************************************************************************
        // Complete board generation


        // This function performs a backtracking algorithm that fills in the given line and recursively all following lines
        // with valid numbers.
        // $line:  the current line number
        // $clusters: the values in the current three clusters so far
        // $cols:  the values in the 9 columns so far
        function FillLines( $line, $clusters, $cols )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );
```

```

        // save our current state
        $our_numbers = $this->numbers;
        $our_clusters = $clusters;
        $our_cols = $cols;

        // base condition
        if( $line >= 9 ) return true;

        // shuffle the valid numbers list
        shuffle( $our_numbers );

        // keep track of the numbers remaining
        $numbers_left = $our_numbers;

        // keep track of our current column
        $index = 0;

        // keep track of numbers that we triad but failed to place
        $slack = array();

        // keep track of how many times we shifted the numbers array to try a new
            sequence
        $num_shifts = 0;

        // now let's try to place the numbers 1..9 into this row
        while( true )
        {
            // grab the next number
            $number = array_pop( $numbers_left );

            // if this number is not in our current cluster and not in our current column
                then we are good to go
            if( !in_array( $number, $our_clusters[ floor($index/3) ] ) && !in_array(
                $number, $our_cols[$index] ) )
            {
                // place the number into the board
                $this->board[$line][$index] = $number;

                // keep track of the addition to this cluster
                $our_clusters[ floor($index/3) ][] = $number;

                // keep track of the addition to this column
                $our_cols[$index][] = $number;

                // move on to the next column
                $index++;

                // add any slack numbers to the numbers we have left
                foreach( $slack as $s )    $numbers_left[] = $s;

                // clear the slack numbers
                $slack = array();
            }
            else
            {
                // no good, add this number to slack, and move on to the next
                $slack[] = $number;
            }

            // if we have covered all columns
            if( $index >= 9 )
```

```
                {
                    // if we are moving to the next group of three lines, then clear the clusters, as we are now leaving them
                    if( intval( $line+1 )%3 == 0 )    $nclusters = array( array(), array(), array() );
                    // else keep the same clusters
                    else                              $nclusters = $our_clusters;

                    // recurse
                    if( $this->FillLines( $line+1, $nclusters, $our_cols ) )    return true;
                }

                // remember, numbers may be in slack, and so this can happen even when we are not done
                if( count( $numbers_left ) == 0 )
                {
                    // if we have shifted as far as we can, then just give up
                    if( $num_shifts == 9 )    return false;

                    // else let's try this line over again
                    unset( $this->board[$line] );

                    // recall our data
                    $our_cols = $cols;
                    $our_clusters = $clusters;

                    // cycle the numbers
                    $numbers_left = $our_numbers;
                    array_shift( $numbers_left );
                    $numbers_left[] = $our_numbers[0];
                    $our_numbers = $numbers_left;

                    // reset the column
                    $index = 0;

                    // reset the slack
                    $slack = array();

                    // keep track of the number of times we do this
                    $num_shifts++;
                }
            }
        }

        // Fills in the board with valid Sudoku numbers
        function FillBoard()
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            shuffle( $this->numbers );
            $this->board = array();

            // set the first line to random values
            $this->board[] = $this->numbers;

            // add these values to clusters and cols, these keep track of what numbers have been used
            $clusters = array( array(), array(), array() );
            for( $i=0; $i<3; $i++ )    $clusters[0][] = $this->board[0][$i];
            for( $i=3; $i<6; $i++ )    $clusters[1][] = $this->board[0][$i];
            for( $i=6; $i<9; $i++ )    $clusters[2][] = $this->board[0][$i];
            $cols = array(    array(), array(), array(),
```

```
                        array(), array(), array(),
                        array(), array(), array() );
        for( $i=0; $i<9; $i++ )    $cols[$i][] = $this->board[0][$i];

        // now fill in the other lines subject to this constraint
        return ( $this->FillLines( 1, $clusters, $cols ) && $this->ValidateBoard( $this->
            board ) );
    }







    //
        ****************************************************************************************

    // Board Validation


    //  Tests if a board confirms to all Sudoku rules
    function ValidateBoard( $board )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        for( $c=0; $c<9; $c++ )
        {
            $cell = array();
            for( $i=0; $i<9; $i++ )
            {
                $a = floor($c/3)*3+floor($i/3);
                $b = (intval($c)%3)*3 + intval($i)%3;

                if( in_array( $board[$a][$b], $cell ) )
                {
                    return false;
                }
                if( $board[$a][$b] != 0 ) $cell[] = $board[$a][$b];
            }
        }
        for( $a=0; $a<9; $a++ )
        {
            $row = array();
            for( $b=0; $b<9; $b++ )
            {

                if( in_array( $board[$a][$b], $row ) )
                {
                    return false;
                }
                if( $board[$a][$b] != 0 ) $row[] = $board[$a][$b];
            }
        }
        for( $b=0; $b<9; $b++ )
        {
            $col = array();
            for( $a=0; $a<9; $a++ )
            {

                if( in_array( $board[$a][$b], $col ) )
                {
```

```
                    return false;
                }
                if( $board[$a][$b] != 0 ) $col[] = $board[$a][$b];
            }
        }

        return true;
    }





    // ***********************************************************************************************
    // Solver

    // returns the local weighted normalized ease function of the entire board
    function WNEF( $our_board, $our_mask, $num=-1 )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        $weights = array( 256, 128, 64, 32, 16, 8, 4, 2, 1 );

        $this->FindChoices( $our_board, $our_mask );
        if( $num == -1 ) $num = $this->NumValues( $our_mask );
        $num = 81-$num;

        if( $num == 0 ) return 1.0;

        $total = 0;
        for( $a=0; $a<9; $a++ )
        {
            for( $b=0; $b<9; $b++ )
            {
                $count = count( $this->choices[$a][$b] );
                if( $our_mask[$a][$b] == 0 && $count > 0 ) $total += $weights[ $count - 1 ];
            }
        }

        return $total / ($weights[0]*$num);
    }

    // returns an array including all unique choices between the given canidates
    function FindUnique( $canidates )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        $unique_spots = array( -2, -2, -2, -2, -2, -2, -2, -2, -2, -2);
        $counts = array(0,0,0,0,0,0,0,0,0,0);
        foreach( $canidates as $k=>$cell )
        {
            foreach( $this->choices[ $cell[0] ][ $cell[1] ] as $choice )
            {
                $unique_spots[$choice] = $k;
                $counts[$choice] ++;
            }
        }
        $unique = array();
        $spot_counts = array();
```

```
            foreach( $unique_spots as $k=>$u )
            {
                if( $counts[$k] == 1 )
                {
                    $unique[$k] = $u;
                    if( isset( $spot_counts[$u] ) ) return false;
                    $spot_counts[$u] = 1;
                }
            }

            return $unique;
        }

        // Removes a choice from all the given canidates
        function RemoveChoice( $canidates, $val )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            foreach( $canidates as $cell )
            {
                foreach( $this->choices[ $cell[0] ][ $cell[1] ] as $key => $choice )
                {
                    if( $choice == $val )
                    {
                        unset( $this->choices[ $cell[0] ][ $cell[1] ][$key] );
                        break;
                    }
                }
                $this->choices[ $cell[0] ][ $cell[1] ] = NormalizeKeys( $this->choices[ $cell
                    [0] ][ $cell[1] ] );
            }
        }


        // Find all choices for all cells in the board.
        // $follow_mask: calculate choices even for unmasked cells
        // $dependents: calculate the dependence instead od the choices
        function FindChoices( $our_board, $our_mask, $follow_mask = true, $dependents = false
             )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            // clear the array
            $this->choices = array();
            for( $a=0; $a<9; $a++ )
            {
                $this->choices[ $a ] = array();
                for( $b=0; $b<9; $b++ )
                {
                    $this->choices[ $a ][ $b ] = array();
                }
            }

            // the values in this cluster that we know
            $cluster = array();

            // traverse clusters
            for( $c=0; $c<9; $c++ )
            {
               $cluster[$c] = array();
                // fill in the cluster values
                for( $i=0; $i<9; $i++ )
```

```
            {
                $a = floor($c/3)*3+floor($i/3);
                $b = (intval($c)%3)*3 + intval($i)%3;

                if( $our_mask[$a][$b] ) $cluster[$c][] = $our_board[$a][$b];
            }
        }

        // traverse cells
        for( $a=0; $a<9; $a++ )
        {
            for( $b=0; $b<9; $b++ )
            {
                $c = floor($b/3)+floor($a/3)*3;

                // if this place is not known
                if( !$follow_mask || !$our_mask[$a][$b] )
                {
                    // find values along horizontal and vertical lines
                    $lines = array();

                    for( $d=0; $d<9; $d++ )
                    {
                        if( $our_mask[$a][$d] ) $lines[] = $our_board[$a][$d];
                        if( $our_mask[$d][$b] ) $lines[] = $our_board[$d][$b];
                    }

                    // now go through and find all values not in the cluster or along the lines
                    if( !$dependents )
                    {
                        for( $d=1; $d<=9; $d++ )
                        {
                            if( !( in_array( $d, $cluster[$c] ) || in_array( $d, $lines ) ) )
                            {
                                $this->choices[$a][$b][] = $d;
                            }
                        }
                    }
                    else
                    {
                        $this->choices[$a][$b] = array_merge( $cluster[$c], $lines );
                    }
                }
            }
        }
    }


    // Set the given cell to the given value, fixing choices acordingly
    function SetCell( $a, $b, $val, &$our_board, &$our_mask )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        // so let's take the move
        $our_mask[$a][$b] = 1;
        $our_board[$a][$b] = $val;

        $c = $this->C( $a, $b );
        $this->choices[$a][$b] = array();
        $this->RemoveChoice( $this->ClusterCanidates($c), $val );
```

```
            $this->RemoveChoice( $this->RowCanidates($a), $val );
            $this->RemoveChoice( $this->ColCanidates($b), $val );
        }

        // Test if the given board is deterministic, aka has only one solution
        function Unique( $our_board, $our_mask, $num, $brute_force=1 )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            // if the board is solved, then it is uniquely solvable

            $this->FindChoices( $our_board, $our_mask );
            while( true )
            {
                if( $num >= 81 ) return true;

                //$this->RenderPuzzle( $our_board, $our_mask );


                // look for cells with just one choice
                $done = false;
                for( $a=0; $a<9 && !$done; $a++ )
                {
                    for( $b=0; $b<9; $b++ )
                    {
                        // if we only have one choice here
                        if( count( $this->choices[$a][$b] ) == 1 )
                        {
                            // then we have a move
                            $num++;
                            $this->SetCell( $a, $b, $this->choices[$a][$b][0], $our_board,
                                $our_mask );

                            // let's get out of this dang thing.... a wish for a goto to
                                implement a deep continue
                            $done = true;
                            $counter = 0;
                            break;
                        }
                    }
                }
                if( $done ) continue;

                // cluster
                $done = false;
                for( $c=0; $c<9; $c++ )
                {
                    $unique = $this->FindUnique( $this->ClusterCanidates( $c ) );
                    if( $unique === false ) return false;
                    foreach( $unique as $k=>$u )
                    {
                        $a = $this->A( $c, $u );
                        $b = $this->B( $c, $u );

                        // then we have a move
                        $num++;
                        $this->SetCell( $a, $b, $k, $our_board, $our_mask );

                        // let's get out of this dang thing.... a wish for a goto to
                            implement a deep continue
                        $done = true;
                        $counter = 0;
```

```
                    break;
                }
            }
            if( $done ) continue;

            // rows
            $done = false;
            for( $a=0; $a<9; $a++ )
            {
                $unique = $this->FindUnique( $this->RowCanidates( $a ) );
                if( $unique === false ) return false;
                foreach( $unique as $k=>$u )
                {
                    $b = $u;

                    // then we have a move
                    $num++;
                    $this->SetCell( $a, $b, $k, $our_board, $our_mask );

                    // let's get out of this dang thing.... a wish for a goto to
                        implement a deep continue
                    $done = true;
                    $counter = 0;
                    break;

                }
            }
            if( $done ) continue;

            // columns
            $done = false;
            for( $b=0; $b<9; $b++ )
            {
                $unique = $this->FindUnique( $this->ColCanidates( $b ) );
                if( $unique === false ) return false;
                foreach( $unique as $k=>$u )
                {
                    $a = $u;

                    // then we have a move
                    $num++;
                    $this->SetCell( $a, $b, $k, $our_board, $our_mask );

                    // let's get out of this dang thing.... a wish for a goto to
                        implement a deep continue
                    $done = true;
                    $counter = 0;
                    break;

                }
            }
            if( $done ) continue;

            // last resort
            $least = 100;
            $least_pos = array( -1, -1 );
            $least_choices = array();
            for( $a=0; $a<9; $a++ )
            {
                for( $b=0; $b<9; $b++ )
                {
                    $n = count( $this->choices[$a][$b] );
```

```
                    if( $n != 0 && $n < $least )
                    {
                        $least = $n;
                        $least_pos = array( $a, $b );
                        $least_choices = $this->choices[$a][$b];
                    }
                }
            }

            $result = false;
            if( $brute_force > 0 )
            {
                foreach( $least_choices as $c )
                {
                    $our_mask[ $least_pos[0] ][ $least_pos[1] ] = 0;
                    $our_board[ $least_pos[0] ][ $least_pos[1] ] = $c;
                    $r = $this->Unique( $our_board, $our_mask, $num+1, $brute_force-1 );
                    if( $r && $result )
                    {
                        $result = false;
                        break;
                    }
                    else if( $r ) $result = true;
                }
            }

            // and that is that
            return $result;
        }
    }

    // Returns a cell to attempt to remove using random selection
    // $anneal controlls anealing by indicating the value in the grid that is associated
        with a "free" cell
    function StrategyRandom( $our_board, $our_mask, $persistance, $counter, $anneal = 1 )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        static $prev_value;

        $spots = array();
        for( $a=0; $a<9; $a++ )
        {
            for( $b=0; $b<9; $b++ )
            {
                if( $our_mask[$a][$b] == $anneal ) $spots[] = array( $a, $b );
            }
        }
        shuffle( $spots );

        $our_place = $spots[0];
        if( isset( $spots[1] ) && $prev_value == $our_place ) $our_place = $spots[1];
        $prev_value = $our_place;
        return $our_place;
    }

    // Returns a cell attempting to remove cells without many choices
    // $anneal controlls anealing by indicating the value in the grid that is associated
        with a "free" cell
    function StrategyCullLow( $our_board, $our_mask, $persistance, $counter, $anneal = 1
        )
    {
```

```
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        static $prev_value;

        $this->FindChoices( $our_board, $our_mask, false );
        $choice_rank = array();
        for( $a=0; $a<9; $a++ )
        {
            for( $b=0; $b<9; $b++ )
            {
                if( $our_mask[$a][$b] == $anneal ) $choice_rank[$a*9+$b] = count( $this->
                    choices[$a][$b] )+$persistance[$a][$b]/$counter;
            }
        }
        shuffle_assoc( $choice_rank );
        asort( $choice_rank );
        $keys = array_keys( $choice_rank );

        $our_place = array( intval($keys[0]/9), intval($keys[0])%9 );
        if( isset( $keys[1] ) && $prev_value == $our_place ) $our_place = array( intval(
            $keys[1]/9), intval($keys[1])%9 );
        $prev_value = $our_place;
        return $our_place;
    }

    // Returns a cell attempting to remove cells WITH many choices
    // $anneal controlls anealing by indicating the value in the grid that is associated
        with a "free" cell
    function StrategyCullHigh( $our_board, $our_mask, $persistance, $counter, $anneal = 1
        )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        static $prev_value;

        $this->FindChoices( $our_board, $our_mask, false );
        $choice_rank = array();
        for( $a=0; $a<9; $a++ )
        {
            for( $b=0; $b<9; $b++ )
            {
                if( $our_mask[$a][$b] == $anneal ) $choice_rank[$a*9+$b] = count( $this->
                    choices[$a][$b] )+$counter/$persistance[$a][$b];
            }
        }
        shuffle_assoc( $choice_rank );
        arsort( $choice_rank );
        $keys = array_keys( $choice_rank );

        $our_place = array( intval($keys[0]/9), intval($keys[0])%9 );
        if( isset( $keys[1] ) && $prev_value == $our_place ) $our_place = array( intval(
            $keys[1]/9), intval($keys[1])%9 );
        $prev_value = $our_place;
        return $our_place;
    }

    // Returns a cell attempting to remove cells in mostly filled clusters
    // $anneal controlls anealing by indicating the value in the grid that is associated
        with a "free" cell
    function StrategyTrimCluster( $our_board, $our_mask, $persistance, $counter, $anneal
        = 1 )
    {
```

```
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        $amounts = array();
        for( $c=0; $c<9; $c++ )
        {
            $amounts[$c] = 0;
            for( $i=0; $i<9;  $i++ )
            {
                $a = $this->A( $c, $i );
                $b = $this->B( $c, $i );

                if( $our_mask[$a][$b] == 1 ) $amounts[$c] += 1 + $counter/$persistance[$a
                    ][$b];
            }
        }
        shuffle_assoc( $amounts );
        if( $anneal == 1 )  arsort( $amounts );
        else                asort( $amounts );
        $keys = array_keys( $amounts );

        $vals = array( 0, 1, 2, 3, 4, 5, 6, 7, 8 );
        shuffle( $vals );
        $c = $keys[0];
        foreach( $vals as $v )
        {
            $a = $this->A( $c, $v );
            $b = $this->B( $c, $v );

            if( $our_mask[$a][$b] == $anneal ) return array( $a, $b );
        }

        return array( -1, -1 );
    }

    // Returns a cell attempting to remove cells in mostly rows
    // $anneal controlls anealing by indicating the value in the grid that is associated
        with a "free" cell
    function StrategyTrimRow( $our_board, $our_mask, $persistance, $counter, $anneal = 1
        )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        $amounts = array();
        for( $a=0; $a<9; $a++ )
        {
            $amounts[$a] = 0;
            for( $b=0; $b<9;  $b++ )
            {
                if( $our_mask[$a][$b] == 1 ) $amounts[$a] += 1 + $counter/$persistance[$a
                    ][$b];;
            }
        }
        shuffle_assoc( $amounts );
        if( $anneal == 1 )  arsort( $amounts );
        else                asort( $amounts );
        $keys = array_keys( $amounts );

        $vals = array( 0, 1, 2, 3, 4, 5, 6, 7, 8 );
        shuffle( $vals );
        $a = $keys[0];
        foreach( $vals as $v )
        {
```

```
                $b = $v;

                if( $our_mask[$a][$b] == $anneal ) return array( $a, $b );
            }

            return array( -1, -1 );
        }

        // Returns a cell attempting to remove cells in mostly filled columns
        // $anneal controlls anealing by indicating the value in the grid that is associated with a "free" cell
        function StrategyTrimCol( $our_board, $our_mask, $persistance, $counter, $anneal = 1 )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            $amounts = array();
            for( $b=0; $b<9; $b++ )
            {
                $amounts[$b] = 0;
                for( $a=0; $a<9;  $a++ )
                {
                    if( $our_mask[$a][$b] == 1 ) $amounts[$b] += 1 + $counter/$persistance[$a][$b];;
                }
            }
            shuffle_assoc( $amounts );
            if( $anneal == 1 )  arsort( $amounts );
            else                asort( $amounts );
            $keys = array_keys( $amounts );

            $vals = array( 0, 1, 2, 3, 4, 5, 6, 7, 8 );
            shuffle( $vals );
            $b = $keys[0];
            foreach( $vals as $v )
            {
                $a = $v;

                if( $our_mask[$a][$b] == $anneal ) return array( $a, $b );
            }

            return array( -1, -1 );
        }

        // Returns a cell attempting to remove cells with many dependents
        // $anneal controlls anealing by indicating the value in the grid that is associated with a "free" cell
        function StrategyTrimDependents( $our_board, $our_mask, $persistance, $counter, $anneal = 1 )
        {
            $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

            static $prev_value;

            $this->FindChoices( $our_board, $our_mask, false, true );

            $amounts = array();
            for( $a=0; $a<9; $a++ )
            {
                for( $b=0; $b<9;  $b++ )
                {
                    if( $our_mask[$a][$b] == $anneal )  $amounts[$a*9+$b] = count( $this->
```

```
                    choices[$a][$b] ) + $counter/$persistance[$a][$b];
                else                                $amounts[$a*9+$b] = 0;
            }
        }
        shuffle_assoc( $amounts );
        arsort( $amounts );
        $keys = array_keys( $amounts );

        $our_place = array( intval($keys[0]/9), intval($keys[0])%9 );
        if( isset( $keys[1] ) && $prev_value == $our_place ) $our_place = array( intval(
            $keys[1]/9), intval($keys[1])%9 );
        $prev_value = $our_place;
        return $our_place;
    }

    // Returns a cell attempting to remove cells that have many other existing of the
        same value
    // $anneal controlls anealing by indicating the value in the grid that is associated
        with a "free" cell
    function StrategyTrimValues( $our_board, $our_mask, $persistance, $counter, $anneal =
         1 )
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );

        $amounts = array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 );
        $places = array( array(), array(), array(), array(), array(), array(), array(),
            array(), array(), array() );
        for( $a=0; $a<9; $a++ )
        {
            for( $b=0; $b<9;  $b++ )
            {
                if( $our_mask[$a][$b] == $anneal )
                {
                    $amounts[$our_board[$a][$b]] += $counter/$persistance[$a][$b];
                    $places[$our_board[$a][$b]][] = array( $a, $b );
                }
            }
        }
        shuffle_assoc( $amounts );
        arsort( $amounts );
        $vals = array_keys( $amounts );

        $places = $places[ $vals[0] ];
        shuffle( $places );

        return $places[0];
    }

    // Fill in the mask
    function FillMask( $difficulty )
    {
        global $difficulty_levels;
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );


        if( $difficulty == 0 ) return $this->FillMaskRandom();

        $this->mask = array(
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
```

```
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
            );
        $this->persistance = array(
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
            );

        // remove some
        $num = 81;

        $total = 0;
        $count = 0;

        // set tuning options
        $strategies             = $difficulty_levels[$difficulty]["strategies"];
        $delta_strategies       = $difficulty_levels[$difficulty]["delta_strategies"];
        $delta_strategies_rate  = $difficulty_levels[$difficulty]["delta_strategies_rate"];
        $num_anneal_attempts    = $difficulty_levels[$difficulty]["num_anneal_attempts"];
        $failed_max             = $difficulty_levels[$difficulty]["failed_max"];
        $wnef_min               = $difficulty_levels[$difficulty]["wnef_min"];
        $wnef_max               = $difficulty_levels[$difficulty]["wnef_max"];
        $run_cleanup            = $difficulty_levels[$difficulty]["run_cleanup"];
        $brute_force            = $difficulty_levels[$difficulty]["brute_force"];

        $annealings = array(
            array( "Sudoku", "StrategyRandom" ),
            array( "Sudoku", "StrategyCullLow" ),
            array( "Sudoku", "StrategyTrimCluster" ),
            array( "Sudoku", "StrategyTrimRow" ),
            array( "Sudoku", "StrategyTrimCol" ),
            array( "Sudoku", "StrategyTrimDependents" ),
            array( "Sudoku", "StrategyTrimValues" ),
            );

        $best_mask = $this->mask;
        $best_wnef = 1;
        $best_num  = 0;
        $wnef_first = 0;
        $persistance_timer = 1;
        $wnef = 1;
        for( $anneal_attempts=0; $anneal_attempts<$num_anneal_attempts; $anneal_attempts++ )
        {
            $failed_count = 0;
            while( true )
            {
                shuffle($strategies);
                $spot = call_user_func( $strategies[0], $this->board, $this->mask, $this->persistance, $persistance_timer );
                //$persistance_timer += 1;
```

```
            if( $failed_count%$delta_strategies_rate == 0 )
            {
                $strategies = array_merge( $strategies, $delta_strategies );
            }

            $a = $spot[0];
            $b = $spot[1];

            // Sentinal value for no spot left
            if( $a == -1 ) break;

            if( $this->mask[$a][$b] != 0 )
            {
                $this->mask[$a][$b] = 0;
                if( !$this->Unique( $this->board, $this->mask, $num-1, $brute_force ) )
                {
                    $this->mask[$a][$b] = 1;

                    $this->persistance[$a][$b]++;
                    $failed_count++;
                }
                else
                {

                    $this->persistance[$a][$b] = 1;
                    $num-=1;
                    $failed_count = 0;
                }
            }
            else
            {
                $failed_count++;
            }
            $wnef = $this->WNEF( $this->board, $this->mask, $num );
            if( $wnef <= $wnef_min || $failed_count >= $failed_max ) break;
        }
        if( $wnef_first == 0 ) $wnef_first = $wnef;

        if( $best_wnef > $wnef )
        {
            $best_mask = $this->mask;
            $best_wnef = $wnef;
            $best_num  = $num;
        }
        else
        {
            $this->mask = $best_mask;
            $wnef       = $best_wnef;
            $num        = $best_num;
        }

        if( $anneal_attempts >= $num_anneal_attempts && $wnef > $wnef_max ) $num_anneal_attempts+=2;

        if( $anneal_attempts < $num_anneal_attempts-1 )
        {
            $num_times = 1+rand()%3;
            for( $i=0; $i<$num_times; $i++ )
            {
                shuffle( $annealings );
                $spot = call_user_func( $annealings[0], $this->board, $this->mask,
```

```
                        $this->persistance, $persistance_timer, 0 );
                    $this->mask[$spot[0]][$spot[1]] = 1;
                    $num += 1;
                }
                echo "\n";
            }
        }


        // endgame
        if( $wnef > $run_cleanup )
        {
            $done = false;
            for( $a=0; $a<9 && !$done; $a++ )
            {
                for( $b=0; $b<9 && !$done; $b++ )
                {
                    if( $this->mask[$a][$b] != 0 )
                    {
                        $this->mask[$a][$b] = 0;
                        if( !$this->Unique( $this->board, $this->mask, $num-1, 1 ) )
                        {
                            $this->mask[$a][$b] = 1;
                        }
                        else
                        {
                            $num-=1;
                            $wnef = $this->WNEF( $this->board, $this->mask, $num );
                            if( $wnef < $wnef_min ) $done = true;
                        }
                    }
                }
            }
        }

        $wnef = $this->WNEF( $this->board, $this->mask, $num );
        return array( $wnef_first, $wnef );
    }

    // Fills in a mask by sucessive removal of cells
    function FillMaskRandom()
    {
        $_Profiler_ = new Profiler( __FUNCTION__, __LINE__ );


        $this->mask = array(
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                        array( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
                    );

        // remove some
        $positions = array();
        for( $a=0; $a<9; $a++ )
        {
            for( $b=0; $b<9; $b++ )
```

```
                {
                    $positions[] = array( $a, $b );
                }
            }
            shuffle( $positions );

            $pos = 0;
            $num = 81;

            $failed = count( $positions );

            foreach( $positions as $key=>$pos )
            {
                $a = $pos[0];
                $b = $pos[1];
                $this->mask[ $a ][ $b ] = 0;

                if( !$this->Unique( $this->board, $this->mask, $num-1 ) )
                {
                    $this->mask[ $a ][ $b ] = 1;
                }
                else $num--;
            }
            $wnef = $this->WNEF( $this->board, $this->mask, $num );
            return array( $wnef, $wnef );
        }
    }

?>
```

Listing 4: Script to render Sudoku puzzles.

```
<?php
    include( 'sudoku.php' );

    $puzzle = new Sudoku();

    $d = 0;
    if( isset( $_GET[ "d" ] ) ) $d = $_GET[ "d" ];

    if( !isset( $_COOKIE["sudoku_board"] ) )
    {
        /* Debug console
        echo "<center><textarea rows=10 cols=80>";
        */

        if( !$puzzle->FillBoard() ) echo "failed";
        $res = $puzzle->FillMask( $d );

        $wnef = $res[1];
        $difficulty = MakeDifficulty( $wnef );
        /*
        echo "\n\n\n";
        print_r( $profile_data );
        echo "</textarea></center>\n\n";
        */

    }
    else
    {

        $puzzle->mask = array(
```

```
                        array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
                        array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
                        array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
                        array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
                        array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
                        array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
                        array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
                        array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
                        array( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ),
                        );
        $puzzle->board = $puzzle->mask;

        $vals_a = explode( ":", $_COOKIE["sudoku_board"] );

        $difficulty = $vals_a[81];
        $wnef       = $vals_a[82];

        unset( $vals_a[81] );
        unset( $vals_a[82] );

        foreach( $vals_a as $key => $n )
        {
            if( $n != 0 )
            {
                $i = intval($key);
                $puzzle->mask[$i/9][$i%9] = 1;
                $puzzle->board[$i/9][$i%9] = $n;
            }
        }
    }
    // set cookie
    $cookie_vals = Array();
    for( $a=0; $a<9; $a++ )
    {
        for( $b=0; $b<9; $b++ )
        {
            if( $puzzle->mask[$a][$b] )
            {
                $cookie_vals[ 2+$a*9+$b ] = $puzzle->board[$a][$b];
            }
            else
            {
                $cookie_vals[ 2+$a*9+$b ] = 0;
            }
        }
    }

    $cookie_vals[] = $difficulty;
    $cookie_vals[] = $wnef;

    setcookie( "sudoku_board", implode( ":", $cookie_vals ), time()+32000000 );

?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
    <head>
        <title>Sudoku</title>

        <script language="javascript" src="js-include/mootools-release-1.11.js"><!-- --></
            script>
        <script language="javascript" src="script.js"><!-- --></script>
        <style>
            body
```

```
{
    padding:              0;
    margin:               0;
}

#difficulty
{
    width:            100%;
    text-align:       center;
    font-size:        300%;
    font-weight:      bold;
    color:            #668;
}

#wnef
{
    margin-top:       -1em;
    margin-bottom:    1em;
    width:            100%;
    text-align:       center;
    color:            #668;
    font-size:        80%;
}

#board,
#board2
{
    width:  1em;
    height: 1em;

    font-size:      20em;

    margin: auto;

    border-style:   solid;
    border-width:   1px;
    border-color:   blue;

    background-color:   black;
}

.large_square
{
    width:  32.4%;
    height: 32.4%;

    font-size:  32.4%;
    float:  left;

    margin: .4%;

    background-color: grey;
}

.small_square
{
    width:  31.3%;
    height: 31.3%;

    font-size:      20%;
    line-height:    160%;

```

```
                float:  left;

                text-align:     center;
                vertical-align: center;

                margin: 1%;

                background-color: white;

                cursor: pointer;
            }

            div.small_square:hover
            {
                background-color:   #FAA;
            }

            .bad
            {
                color:  #A11;
            }

            .static
            {
                color:  #11A;

                cursor: default;
            }

            #menu
            {
                text-align:     center;
                margin: 0.4em;
            }

            #menu a, #menu select
            {
                font-weight:    bold;
                color:  #A48;


                border-style:   dotted;
                border-width:   1px;
                border-color:   #D8A;

                padding: 0.2em;

                cursor: pointer;
            }

            #menu a:hover, #menu select:hover
            {
                color:  #848;

                border-style:   solid;
            }
        </style>
    </head>
    <body>
        <div id="difficulty"> <?php echo $difficulty; ?> </div>
        <div id="wnef"> <?php echo number_format( $wnef, 3 ); ?> </div>
        <div id="board">
```

```
<?php
            // render
            // $puzzle->Unique( $puzzle->mask, 80, false );
            for( $c=0; $c<9; $c++ )
            {
                echo "<div␣class=\"large_square\">";

                for( $i=0; $i<9; $i++ )
                {
                    $a = floor($c/3)*3+floor($i/3);
                    $b = (intval($c)%3)*3 + intval($i)%3;

                    $keep = $puzzle->mask[$a][$b];
                    echo "<div␣class=\"small_square␣" . ($keep ? "static" : "" ) . "␣cell_$c␣
                        col_$b␣row_$a\"␣id=\"$c"."_$a"."_$b\">";

                    if( $keep )
                    {
                        echo $puzzle->board[$a][$b];
                    }
                    else
                    {
                    //  echo "<small>".$puzzle->board[$a][$b]."</small>";
                    }

                    echo "</div>";
                }
                echo "</div>";
            }
?>
        </div>
        <div id="menu">
            <select id="sel_difficulty">
                <option value ="0">Random</option>
                <option value ="1">Easy</option>
                <option value ="2">Medium</option>
                <option value ="3">Hard</option>
                <option value ="4">Evil</option>
            </select>
            <a onclick="NewBoard()">New Puzzle</a> <a onclick="Clear()">Clear Puzzle</a>
        </div>
    </body>
</html>
```

Listing 5: Tuning parameters for generator algorithm.

```
<?php

    $difficulty_levels = array(
        1 => array(
            "strategies"             => array(
                                              array( "Sudoku", "StrategyCullHigh" ),
                                        ),
            "delta_strategies"       => array(),
            "delta_strategies_rate"  => 50,
            "num_anneal_attempts"    => 1,
            "failed_max"             => 5,
            "wnef_min"               => 0.32,
            "wnef_max"               => 0.35,
            "run_cleanup"            => 0.4,
            "brute_force"            => 0,
            ),
```

```

        2 => array(
            "strategies"            => array(
                                        array(  "Sudoku", "StrategyRandom" )
                                       ),
            "delta_strategies"      => array(
                                        array(  "Sudoku", "StrategyRandom" )
                                       ),
            "delta_strategies_rate" => 40,
            "num_anneal_attempts"   => 5,
            "failed_max"            => 2,
            "wnef_min"              => 0.28,
            "wnef_max"              => 0.28,
            "run_cleanup"           => 0.28,
            "brute_force"           => 0,
            ),
        3 => array(
            "strategies"            => array(
                                        array(  "Sudoku", "StrategyTrimValues" ),
                                        array(  "Sudoku", "StrategyCullLow" ),
                                        array(  "Sudoku", "StrategyTrimCluster" ),
                                        array(  "Sudoku", "StrategyTrimRow" ),
                                        array(  "Sudoku", "StrategyTrimCol" ),
                                        array(  "Sudoku", "StrategyTrimDependents" ),
                                       ),
            "delta_strategies"      => array(
                                        array(  "Sudoku", "StrategyRandom" ),
                                       ),
            "delta_strategies_rate" => 10,
            "num_anneal_attempts"   => 10,
            "failed_max"            => 3,
            "wnef_min"              => 0.2,
            "wnef_max"              => 0.2,
            "run_cleanup"           => 0.2,
            "brute_force"           => 0,
            ),
        4 => array(
            "strategies"            => array(
                                        array(  "Sudoku", "StrategyTrimValues" ),
                                        array(  "Sudoku", "StrategyCullLow" ),
                                        array(  "Sudoku", "StrategyCullLow" ),
                                        array(  "Sudoku", "StrategyCullLow" ),
                                        array(  "Sudoku", "StrategyCullLow" ),
                                        array(  "Sudoku", "StrategyCullLow" ),
                                        array(  "Sudoku", "StrategyCullLow" ),
                                        array(  "Sudoku", "StrategyCullLow" ),
                                        array(  "Sudoku", "StrategyTrimCluster" ),
                                        array(  "Sudoku", "StrategyTrimRow" ),
                                        array(  "Sudoku", "StrategyTrimCol" ),
                                        array(  "Sudoku", "StrategyTrimDependents" ),
                                       ),
            "delta_strategies"      => array(
                                        array(  "Sudoku", "StrategyRandom" ),
                                       ),
            "delta_strategies_rate" => 10,
            "num_anneal_attempts"   => 100,
            "failed_max"            => 3,
            "wnef_min"              => 0,
            "wnef_max"              => 0.10,
            "run_cleanup"           => 0,
            "brute_force"           => 2,
            ),
```

```
    );
?>
```

Listing 6: Python script to extract GNOME Sudoku puzzles.

```python
import sys
import getopt
from gnome_sudoku.sudoku_maker import SudokuMaker

def print_puzzles(sm, f, min_d, max_d):
    puzzles = [sm.get_puzzle(d.calculate()) for d in sm.list_difficulties() if (d.calculate() > min_d) and (d.calculate() < max_d) ]
    for g,d in puzzles:
            f.write(g.replace(" ", "") + "\t" + d.value_string() + "\n")

shortargs = "e:m:h:v:w:"
longargs = ["easy=" "medium=" "hard=" "evil=" "writemode="]

def main(argv):
    default = "controls/gnome-sudoku/gnome-sudoku-"
    easypath = default + "easy"
    medpath = default + "med"
    hardpath = default + "hard"
    evilpath = default + "evil"
    writemode = "w"

    opts, args = getopt.getopt(sys.argv[1:], shortargs, longargs)

    for opt, arg in opts:
        if opt in ("-e", "--easy"):
            easypath = arg
        if opt in ("-m", "--medium"):
            medpath = arg
        if opt in ("-h", "--hard"):
            hardpath = arg
        if opt in ("-v", "--evil"):
            evilpath = arg
        if opt in ("-w", "--writemode"):
            writemode = arg

    ef = open(easypath, writemode);
    mf = open(medpath, writemode);
    hf = open(hardpath, writemode);
    vf = open(evilpath, writemode);

    try:
        sm = SudokuMaker(batch_size=10)
    except exceptions.EOFError:
        pass

    sm.make_batch()

    print_puzzles(sm, ef, 0.00, 0.25)
    ef.close()

    print_puzzles(sm, mf, 0.25, 0.50)
    mf.close()

    print_puzzles(sm, hf, 0.50, 0.75)
    hf.close()

    print_puzzles(sm, vf, 0.75, 1.00)
```

```
    vf.close()


if __name__ == "__main__":
    main(sys.argv)
```

2 Screenshots of Puzzle Generator

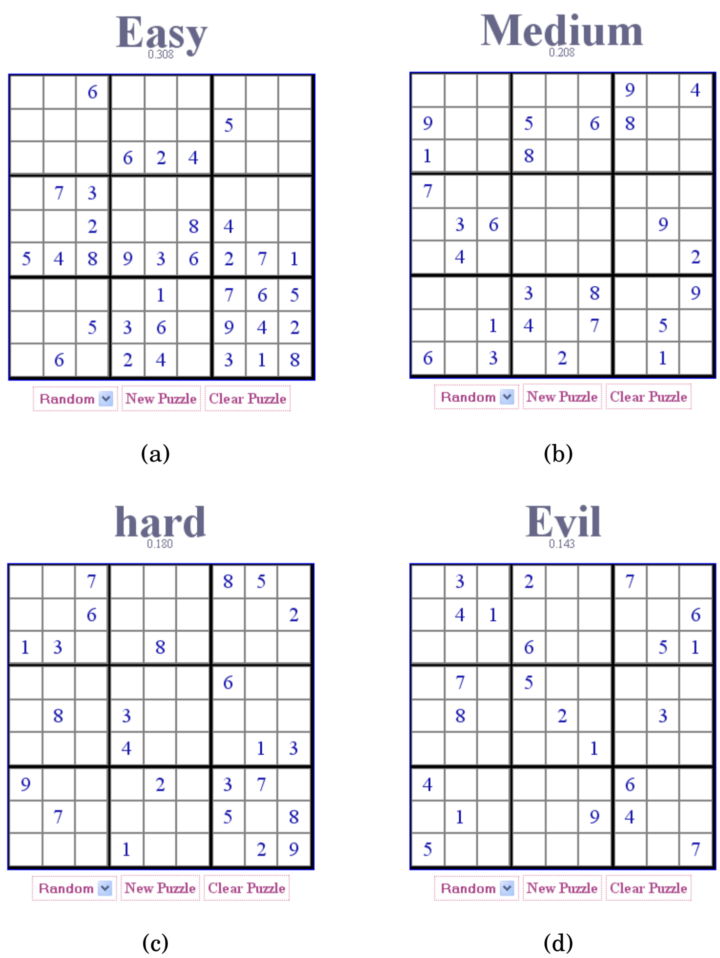


Figure 15: Screenshots of puzzle generator.