`hsolve`: A Difficulty Metric and Puzzle Generator for Sudoku

MCM Team #2858: Christopher Chang, Zhou Fan, and Yi Sun

February 19, 2008

Abstract

Creating and rating the difficulty of Sudoku puzzles is currently a hard computational problem. In particular, most current approaches require the use of somewhat arbitrary choices for the relative difficulties of Sudoku strategies. We present here a novel solver-based solution to this problem by framing Sudoku as a search problem and using the expected search time to determine the difficulty of different strategies. Our method was chosen for its relative *independence* from external views on the relative difficulties of strategies.

Upon validation of our metric with a sample of 800 externally rated puzzles with 8 gradations of difficulty, we found a Goodman-Kruskal γ coefficient of 0.82, indicating significant correlation. An independent evaluation of 1000 typical puzzles produced a difficulty distribution similar to the distribution of solve times empirically created by millions of users at `www.websudoku.com`.

Based upon this difficulty metric, we created two separate puzzle generators. One generates mostly easy to medium puzzles; when run with 4 difficulty levels, it creates boards of levels 1, 2, 3, and 4 in 0.25, 3.1, 4.7, and 30 minutes, respectively. The other modifies difficult boards to create boards of similar difficulty; when tested on a board of difficulty 8122, it was able to create 20 boards with average difficulty 7111 in 3 minutes.

Contents

1 Introduction

Sudoku is a logic puzzle that has recently become extremely popular. In Sudoku, a player is presented with a 9×9 grid divided into nine 3×3 regions. Some of the 81 cells of the grid are initially filled with digits between 1 and 9 such that there is a unique way to complete the rest of the grid while satisfying the following rules:

1. Each cell contains a digit between 1 and 9
2. Each row, column, and 3×3 region contains exactly one copy of the digits $\{1, 2, \ldots, 9\}$.

A *Sudoku puzzle* consists of such a grid together with an initial collection of digits that guarantees a unique final configuration. Call this final configuration a *solution* to the puzzle. The goal of Sudoku is to find this unique solution from the initial board.

Below is an example of a Sudoku puzzle and solution from the February 16th, 2008 edition of the London Times [18]:

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

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

In this particular example, we cannot have the numbers 8, 3, or 7 appear anywhere else on the bottom row, since each number can only show up in the bottommost row once. Similarly, the number 8 cannot appear in any of the empty squares in the lower-left-hand region.

1.1 Notation

We first introduce some notation. Number the rows and columns from 1 to 9, beginning at the top and left, respectively, and number each 3×3 region of the board as follows:

1	2	3
4	5	6
7	8	9

We will refer to a cell by an ordered pair (i, j), where i is its row and j its column, and the term *group* will collectively denote a row, column, or region.

Given a Sudoku board B, define the *Sudoku Solution Graph* (SSG) $S(B)$ of B to be the structure that associates to each cell in B the set of the digits that are currently thought to be possible candidates for the cell. For example, in the first Sudoku puzzle presented, cell $(9, 9)$ cannot take the values $\{1, 3, 4, 7, 8, 9\}$ because it shares a group with cells with these values. Therefore, this cell has values $\{2, 5, 6\}$ in the corresponding SSG.

To solve a Sudoku board, a player generally applies a number of different *strategies*, or patterns of logical deduction. These range from fairly obvious to extremely complicated, and a list of Sudoku strategies used in this paper is provided in Appendix A. In this paper, we assume the SSG has been evaluated for every cell on the Sudoku board before applying any strategies.

1.2 Problem Background

Since its recent rise in popularity, Sudoku has been the focus of increased academic and recreational study. Most of the efforts directed at Sudoku have been directed at solving Sudoku puzzles or analyzing the computational complexity of resolving Sudoku as done in [13], [4], and [14]. Most notably, Sudoku can now be solved extremely quickly via a reduction to an exact cover problem and an application of Knuth's Algorithm X given in [11], and solving the $n^2 \times n^2$ generalization of Sudoku is known to be NP-complete as a consequence of results given by Yato in [22].

However, a perhaps deeper and more difficult issue involves rating the difficulty of and generating Sudoku puzzles. This problem encompasses the following two questions:

1. Given a specific Sudoku puzzle, how does one define and determine its difficulty?

2. Given a specified difficulty, how does one generate a Sudoku puzzle of this difficulty?

While generating a valid Sudoku puzzle is not too complex, the non-local and unclear process of deduction makes determining or specifying a difficulty much more complicated.

Traditional approaches to difficulty rating involve rating a puzzle by the strategies necessary to find the solution, while some other approaches have been proposed in [1] and [3]. In particular, a genetic algorithms approach taken by Mantere et. al. in [15] found some correlation with human-rated difficulties, and Simonis presents some interesting similar findings with a constraint-based rating [17]. However, in both cases, the correlation is not completely clear.

Puzzle generation seems to be an even more difficult issue. Most existing generators use complete search algorithms to systematically add numbers to cells in a grid until a unique solution is found. To generate a puzzle of a given difficulty, this process is repeated until the desired difficulty is achieved. This is the approach found in [15], while [17] posits both this and a similar method based upon removal of cells from a completed board. In [5], Felgenhauer et. al. has enumerated the total possible number of valid Sudoku puzzles.

In this paper, we present a new approach to rating and generating puzzles. We create `hsolve`, a program to simulate the way a human solver approaches a Sudoku puzzle in order to present a new solver-based difficulty metric based upon `hsolve`'s simulation of human solving behavior. We then propose two different methods again based on `hsolve` to generate Sudoku puzzles of different difficulties.

2 Problem Setup

In this paper, we approach the problem in two portions. First, we define a difficulty metric for a Sudoku puzzle and determine an algorithm to compute it. We then create an algorithm to generate Sudoku puzzles with desired levels of difficulty. Before beginning to describe our model, however, we first specify the problem a bit more closely in both cases.

2.1 Difficulty Metric

Our goal in this part of the paper is to create an algorithm that takes a Sudoku puzzle and returns a real number that represents its abstract "difficulty" according to some metric.

The central issue in determining such a metric is then what we mean by the "difficulty" of a Sudoku puzzle. We base our definition of difficulty on the following general assumptions:

1. The amount of time it takes for any given solver to solve a puzzle is monotonically increasing with difficulty.

2. Every solver tries to solve Sudoku puzzles by applying various strategies.

The difficulty of a puzzle is an inherently subjective quantity and may vary among different solvers who use different methods of solving Sudoku puzzles. However, this type of subjective consideration is not inherent to the puzzle, so we must restrict ourselves to purely objective reactions. For this purpose, it is a useful intellectual tool to consider some hypothetical "typical" solver of some postulated skill level.

Now, Assumption 1 suggests that we should measure difficulty by the amount of time spent solving the puzzle. Because difficulty is increasing in time spent, any definition of difficulty must give the same ordering as time spent. Since this gives an objectively precise definition, we adopt it; therefore, it only remains to determine more precisely what type of solver our metric should consider.

The difficulties of puzzles can vary for solvers of different skill levels, and it may be argued that one should measure each of these difficulties separately. However, we are assuming that all solvers' time spent will be consistent with difficulty, and assigning absolute difficulty values has much practical value. Therefore, in order to avoid the dependence of our results on the novice's ignorance of various techniques and to extend the range of measurable puzzles, we take our hypothetical solver to be an expert.

Hence, we define the *difficulty* of a Sudoku puzzle to be **the average amount of time a hypothetical "typical" Sudoku expert would spend solving it**. In the subsequent parts of this paper, our aim will be to create and evaluate a metric to determine this time.

2.2 Puzzle Generation

A second objective of this paper is to develop a method of puzzle generation. Our main goal in puzzle generation is to produce a valid puzzle of a given desired difficulty level that has a unique solution. As there is no objective scale in difficulty and no fixed criteria that indicate the difference between puzzles of varying difficulties, we will interpret the given number of difficulty levels and the desired difficulty level as the division of the set of all Sudoku boards into the given number of difficulty percentile intervals of equal size and the choice of a particular desired percentile interval. We will take a sample of 1000 Sudoku boards and assume that the difficulty distribution of these boards, as measured by our human solver, is representative of the difficulty distribution of all Sudoku boards.

Our second goal in puzzle generation is to minimize the complexity of the generation algorithm. We note that since the size of the 9×9 Sudoku board is fixed, the order of growth of our generation algorithm is irrelevant. Therefore, in this paper, we will interpret the complexity of a generation algorithm to be the expected amount of execution time required to find a Sudoku puzzle of the desired difficulty level.

3 A Difficulty Metric

In this section, we describe and evaluate a difficulty metric for Sudoku puzzles based upon a computer solver `hsolve` that simulates the actions of a human Sudoku expert.

3.1 Assumptions and Metric Development

As stated in Section 2, we define the difficulty of a puzzle as the average amount of time required for an expert Sudoku solver to solve it. In order to measure this time, there are essentially two possibilities:

1. Model the process of solving the puzzle
2. Find some heuristic for board configurations that predicts the solve time.

There are some known heuristics for finding the difficulty of a puzzle; most notably, puzzles with a small number of initial givens are somewhat harder than most. However, according to [9], the overall correlation is weak. Other methods of this type have similar problems, and, more importantly, they cannot be used to determine the difficulty of any specific puzzle.

Therefore, the second possibility can be used at most as a guide for the first, and we must model the actual process of solving the puzzle. We postulate that the following assumptions hold for the solver:

1. The possible strategies can be ranked in order of difficulty, and the solver always applies them from least to most difficult.

   This is consistent with more or less all references on the subject of solving sudoku puzzles. We use a widely accepted ranking of strategies described in Appendix A.

2. During the search for a strategy application, each ordering of possible strategy applications occurs with equal probability.

   There are two components of a human search for a possible location to apply a strategy: complete search and intuitive pattern recognition. While human pattern recognition is extremely powerful (see, for example [2]), it is extremely difficult to determine its precise consequences, especially due to possible differences between solvers. Therefore, we do not consider any intuitive component to pattern recognition and restrict our model to a complete search for strategy applications. Such a search will proceed between possible applications in the random ordering that we postulate.

We define a *possible application* of a strategy to be a configuration on the board that is checked by a human to determine if the given strategy can be applied; a list of exactly which configurations are checked varies by strategy and is given in Appendix A. We can now model our solver as following the algorithm *HumanSolve* defined as follows:

Definition. Given a sudoku puzzle, the algorithm HumanSolve repeats the following steps until there are no remaining empty squares:

1. Choose the least difficult strategy that has not yet been searched for in the current board configuration.
2. Search through possible applications of any of these strategies for a valid application of a strategy.
3. Apply the first valid application found.

We take the difficulty of a single run of HumanSolve to be **the total number of possible applications that the solver must check**; here we assume that each check takes roughly the same amount of time. Note that multiple runs of this method on the same puzzle may have different difficulties due to different valid applications being recognized first. We are now ready to define the difficulty metric of a sudoku board B.

Definition. For a sudoku board B, take its difficulty metric $m(B)$ to be **the average total number of possible applications checked by the solver** while using the HumanSolve algorithm.

3.2 `hsolve` and Metric Calculation

In order to actually calculate the difficulty $m(B)$ of a board B, we use `hsolve`, a program which simulates the running of HumanSolve and calculates the resulting difficulty. `hsolve` is implemented in Java 1.6, and its source code is attached in Appendix B.

Given a Sudoku puzzle B, `hsolve` does the following:

1. Set the initial difficulty $d = 0$
2. Repeat the following actions in order until B is solved or the solver cannot progress:
   (a) Choose the tier of easiest strategies S that has not yet been searched for in the current board configuration.
   (b) Find the number p of possible applications of S.
   (c) Find the set V of all valid applications of S and compute the size v of V.

   (d) Compute $E(p, v)$, the expected number of possible applications that will be examined before a valid application is found.

   (e) Increment d by $E(p, v) \cdot t$, where t is the standard check time. Pick a random application in V and apply it to the board.

3. Return the value of d and the final solved board.

While `hsolve` is mostly a direct implementation of HumanSolve, it does not actually perform a random search through possible applications; instead it uses the expected search time $E(p, v)$ to simulate this search. Note that the following lemma gives an extremely convenient closed form expression for $E(p, v)$ that we use in `hsolve`.

Lemma 1. Assuming that all search paths through p possible approaches are equally likely, the expected number $E(p, a)$ of checks required before finding one of v valid approaches is given by

$$E(p, v) = \frac{p+1}{v+1}.$$

Proof. For our purposes, to specify a search path it is enough to specify the v indices of the valid approaches out of p choices, so there are $\binom{p}{v}$ possible search paths. Let I be the random variable equal to the smallest index of a valid approach. Then, we have that

$$E(p, v) = \sum_{i=1}^{p-v+1} iP(I = i) = \sum_{i=1}^{p-v+1} \sum_{j=i}^{p-v+1} P(I = j) = \sum_{i=1}^{p-v+1} P(I \geq i)$$
$$= \frac{1}{\binom{p}{v}} \sum_{i=1}^{p-v+1} \binom{p+1-i}{v} = \frac{1}{\binom{p}{v}} \sum_{j=0}^{p-v} \binom{v+j}{v} = \frac{\binom{p+1}{v+1}}{\binom{p}{v}} = \frac{p+1}{v+1},$$

where we've used the Hockeystick identity. □

Given a Sudoku puzzle B, then, we calculate $m(B)$ by running `hsolve` several times and taking the average of the returned difficulties. To increase the level of accuracy, the number of runs can be increased; 20 runs per puzzle was found to give a ratio of standard deviation to mean of about $\frac{\sigma}{\mu} \approx \frac{1}{10}$, so we use this value throughout the rest of the paper.

3.3 Analysis

Our evaluation of `hsolve` consists of three major components:

1. Checking that `hsolve`'s conception of difficulty is correlated with existing conceptions of difficulty.

2. Comparing the distribution of difficulties generated by `hsolve` to established distributions for solve time.

3. Finding the runtime of the algorithm.

3.3.1 Validation Against Existing Difficulty Ratings

For each of the difficulty ratings in `{supereasy,veryeasy,easy,medium,hard,harder, veryhard,superhard}`, we downloaded a set of 100 puzzles from [8] to obtain a different difficulty rating to compare with. While this dataset is not a standard difficulty benchmark, no other large datasets with varying difficulty ratings were available, and we are looking only for correlation, since there is no objective definition of difficulty.

We ran `hsolve` on each puzzle 20 times and recorded the average difficulty for each board. We then classified the boards by difficulty into 8 groups of 100 boards based on our difficulty metric. The table of results is shown below:

Difficulty	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
`supereasy`	81	19	0	0	0	0	0	0
`veryeasy`	19	68	12	1	0	0	0	0
`easy`	0	8	38	33	18	2	1	0
`medium`	0	2	26	29	22	17	4	0
`hard`	0	2	10	19	20	30	11	8
`harder`	0	0	5	7	22	26	36	4
`veryhard`	0	1	9	7	16	13	27	27
`superhard`	0	0	0	4	2	12	21	61

$\chi^2 = 6350\,(df = 49)$

$\gamma = 0.82$

Performing a χ^2-test for independence and computing the Goodman-Kruskal γ coefficient [6], we obtain that $\chi^2 = 6350$ and $\gamma = 0.82$. Note that this corresponds to a p value of less than 0.0001 for the χ^2 test, meaning that there is a statistically significant deviation from independence between these two measures of difficulty [7].

Furthermore, the Goodman-Kruskal coefficient $\gamma = 0.82$ is relatively close to 1, indicating a somewhat strong correlation between our measure of difficulty and the existing metric. This provides some support for the validity of our metric; more precise error analysis seems unnecessary here because we wish only to check that our values are close to those provided by others.

3.3.2 Validation of Difficulty Distribution

When run 20 times on each of 1000 typical Sudoku puzzles obtained from [12], `hsolve` generates the following distribution for measured difficulty. As can be seen in Figure 1, the distribution is sharply peaked near 500 and has a long tail towards higher difficulty.

We can compare this difficulty distribution plot with the distribution of times required for visitors to `www.websudoku.com` to solve the puzzles available there [21]. This distribution is generated by the solution times of millions of users and is shown in the plot in Figure 2.

The two distribution graphs both share a peak near 0 and have fat tails in the positive direction. While the tail of our measured difficulties is somewhat fatter, it exhibits the same qualitative behavior as a distribution of solve times generated by millions of real users, again providing validation for our difficulty metric.

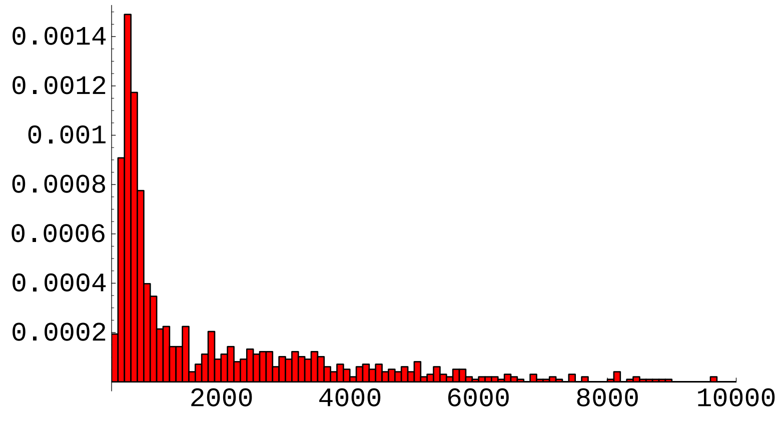

Figure 1: A histogram of the overall measured difficulty for 1000 typical puzzles. (drawn by Mathematica)

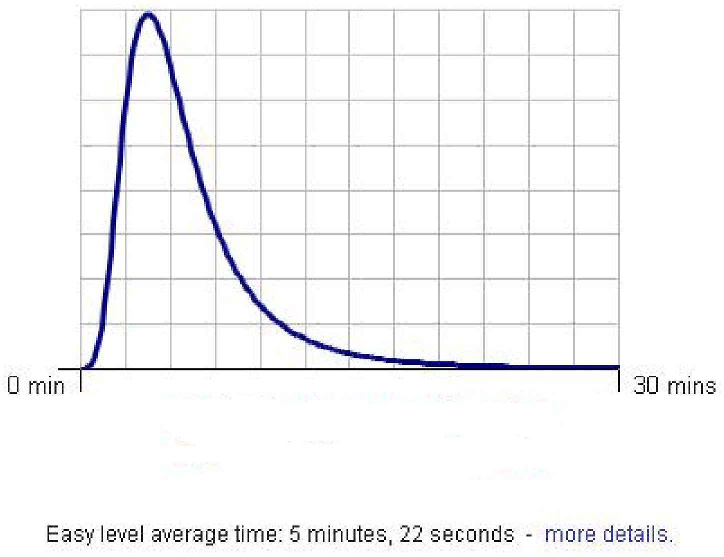

Figure 2: A distribution plot of the amount of time require to solve Easy level puzzles on `www.websudoku.com`.

3.3.3 Runtime

When evaluating the runtime of our metric, we do not consider the order of growth of our algorithm, since we are dealing with a fixed 9×9 board. Therefore, it suffices to consider the absolute time required to solve a single puzzle. When running 20 iterations of `hsolve` per puzzle, rating 100 puzzles required about 13 minutes, for a runtime of about $\frac{13 \cdot 60}{100} = 7.8$ seconds per puzzle on a 2 Ghz Centrino Duo processor with 256 Mb of Java heap space. While this runtime is a bit slower than existing difficulty raters, we feel that `hsolve` provides a much more detailed evaluation of the difficulty that justifies this extra time.

4 Generator

Having determined a metric for evaluating the difficulty of a Sudoku board, the remainder of this paper addresses the issue of developing a Sudoku puzzle generator that can generate boards of a given difficulty level. Our choice of using a solver-based metric for difficulty has the following implications on the process of puzzle generation:

1. Due to the complex nature of this metric, it is impossible to make a very accurate prediction of the difficulty of the puzzle in the process of generating it, before all of the numbers on the puzzle have been determined. This is because adding or repositioning a number on the board can have a profound impact on which strategies are needed to solve the puzzle.

   Thus, given a difficulty, we create a puzzle generating procedure that generates a puzzle of approximately the desired difficulty and then runs `hsolve` on the generated puzzle to determine if the actual difficulty is the same as the desired difficulty. This is the approach we will take in both the generator and pseudo-generator described below.

2. There is an inevitable trade-off between the ability to generate consistently difficult puzzles and the ability to generate truly random puzzles. Given a generator that creates puzzles with as randomized a process as possible, its puzzles are unlikely to be very difficult, since complex strategies such as Naked Quad and Swordfish, which contribute to puzzle difficulty, would not be employed very often.

   Conversely, if we wanted a procedure that consistently generates hard puzzles, we would have to reduce the amount of randomness in the puzzle-generating process or limit the types of puzzles that can result to ensure difficulty.

3. With regards to computation time, because we check each generated puzzle's difficulty with `hsolve`, the speed at which puzzles can be generated depends upon the speed of `hsolve`.

We will proceed to describe two algorithms for generating puzzles: a standard generator and a pseudo-generator.

4.1 Standard Generator

Our standard puzzle generator follows the algorithm below:

1. Begin with an empty board and randomly choose one number to fill into one cell.
2. Apply `hsolve` to make all logical deductions possible. (i.e. After every step of generating a puzzle, keep track of the Sudoku Solution Graph for all cells of the board.)
3. Repeat the following steps until either (1) a contradiction is reached or (2) the board is completed:
    - Randomly fill an unoccupied cell on the board with a candidate for that cell's SSG.
    - Apply `hsolve` to make all logical deductions (which will fill in naked and hidden singles and adjust the SSG accordingly)
    - If a contradiction occurs on the board, abort the procedure and start the process again from an empty board.

If no contradiction is reached, then eventually the board must be completely filled since this algorithm manually fills in one new cell at each step. The final puzzle that we create is the solvable board with all of the numbers that were manually filled in at each iteration of our algorithm (i.e. the board without the numbers filled in by `hsolve`).

4.1.1 Guaranteeing a Unique Solution with Standard Generator

We note that in order for this algorithm to work, a small modification must be made in our backtracking strategy. If the backtracking strategy makes a guess that successfully completes the puzzle, we treat it as if this guess does not complete the puzzle but rather comes to a dead end. Thus, the backtracking strategy only makes a modification to the board if it makes a guess on some square that results in a contradiction, in which case it fills in that square with the other possibility. With this modification, we easily see that if our algorithm successfully generates a puzzle, then the puzzle must have a unique solution, because all of the cells of the puzzle that are not filled in are those that were determined at some point in the construction process by `hsolve`. With this updated backtracking strategy, `hsolve` makes a move only if the move follows logically and deterministically from the current state of the board, so if `hsolve` reaches a solution, it must be the unique one.

4.2 Pseudo-Generator

Our pseudo-generator takes a full, completed Sudoku board and a set of cells to leave empty at beginning of a puzzle, called the *reserved cells*. The idea behind the pseudo-generator is to guarantee the use of some high-level strategy, such as Swordfish or Backtracking, by ensuring that a generated puzzle cannot be completed without such strategies. We must start with some puzzle and its solution; call the puzzle the *seed board* and the solution the *completed seed board*. To use pseudo-generator, we must first prepare a list of reserved cells, found as follows:

1. Take a seed board that `hsolve` cannot solve using strategies only up to tier k, but `hsolve` can solve with strategies up to tier $k+1$ (see Appendix A for the different tiers of strategies we use).

2. Use `hsolve` to make all possible deductions (i.e. adjusting the SSG) using only strategies up to tier k.
3. Create a list of cells that are still empty.

We then pass to pseudo-generator the completed seed board and this list of reserved cells. Pseudo-generator the repeats the following algorithm below, starting with an empty board, until all the cells except the reserved cells are filled in:

1. Randomly fill an unoccupied, unreserved cell on the board that with the number in the corresponding cell of the completed seed board.
2. Apply `hsolve` to make logical deductions and to complete the board as much as possible.

4.2.1 Differences From Standard Generator

The main differences between Pseudo-Generator and Standard Generator are as follows:

1. When filling in an empty cell, the generator uses the number in the corresponding cell of the completed puzzle, instead of choosing this number at random from the cell's SSG.
2. When selecting which empty cell to fill in, pseudo-generator never selects one of the reserved cells.
3. `hsolve` is equipped only with strategies up to tier k.
4. The algorithm terminates not when the board is completely filled in, but rather when all of the unreserved cells are filled in.

This algorithm is only partially random. The pseudo-generator will provide enough clues so that the unreserved cells of the board can be solved with strategy up to tier k, and the choice of which of these cells to reveal as clues is determined randomly. We should note, however, that the solution of the generated puzzle is independent of these random choices and must be identical to the completed seed board. For the same reason as in the standard generator, the solution found in this way must be unique.

Pseudo-generator never provides clues for reserved cells; hence, when `hsolve` solves this puzzle, it will use strategies of tiers 0 through k to fill in the unreserved cells of the board, and then be forced to use some strategy in tier $k+1$, which then allows it to solve the remaining portion of the board.

4.2.2 Pseudo-Generator Puzzle Variability

The benefit of this algorithm over the standard algorithm is that it generates puzzles in which a strategy of tier $k+1$ must be used, thus guaranteeing a high level of difficulty for the puzzle if k is high. The drawback is that this algorithm cannot be said to randomly generate a puzzle, since it is really starting with a puzzle that has already been generated in the past and constructing a new puzzle, using some random choices, out of its solution.

We implement this algorithm by first randomly permuting the rows, columns, and numbers of the given completed puzzle to create an illusion that it is a different puzzle. Ideally, this algorithm should be used in conjunction with a large database of difficult puzzles (and necessary data for these puzzles such as the highest tier strategy needed to solve it and the list of reserved cells that cannot be filled with strategies of lower tiers), and it should choose a starting puzzle at random from this large database. This, in conjunction with random permutations of rows, columns, and numbers, as well as random choices of clues in the part of the board that is solvable with low-level strategies, can consistently generate a class of difficult puzzles that appear to be random–hence the "pseudo" in the algorithm name.

4.3 Results and Analysis

The primary goal of this section will be to establish, using the results of test runs of our puzzle generation procedures, some guidelines for how to use the two algorithms to achieve puzzles of particular difficulty levels, and to estimate the expected run time for each algorithm.

4.3.1 Difficulty Concerns

The first issue that we must address is that the concept of a particular difficulty level is not well-defined: In a system of three difficulty levels, how difficult is a medium puzzle, as compared to a hard or easy puzzle? In the correlation analysis previously performed in which we took 100 puzzles each from 8 difficulty levels, computed our difficulty metric on each puzzle, and divided the 800 puzzles into 8 difficulty levels ourselves, we assumed that each difficulty level should still contain 100 puzzles. Is this assumption reasonable? Without an objective guideline for how to classify puzzle difficulty, these questions are hard to answer.

4.3.2 Generating Puzzles with a Specific Difficulty

We take the following approach: Given the set of all Sudoku puzzles, of size n, that are distributed for people to solve, each one with its own difficulty value as measured by our metric, puzzles in the k^{th} highest difficulty level out of m levels should be chosen by dividing the set of puzzles into m evenly sized intervals of difficulty and selecting the $\frac{n}{m}$ puzzles that are in the k^{th} most difficult set. Unfortunately, due to the limited amount of resources that were available to us, we could only select a sample of 1000 Sudoku puzzles [8] and assume that the difficulty distribution of this sample reflects the difficulty distribution of all Sudoku puzzles. The distribution is shown in Figure 1.

Given any number of levels m and a particular level k, we may then find the bounds on the importance values for that level based on the distribution. For instance, for 4 levels, the bounds on importance on the levels are 317 to 598, 599 to 928, 929 to 2761, and 2762 to 20957. To create a puzzle of the given difficulty level using the standard generator, we may iterate the generator until a puzzle is generated whose difficulty value falls within the appropriate set of bounds; this functionality is implemented in the createBoard method (of the PuzzleConstructor class) of our code. In a trial run of 1000 generations of our standard generator, 598 of the puzzles were in the 317 to 598 range, 49 were in the 599 to 928 range, 32 were in the 929 to 2761 range, and 5 were in the 2762 to 20957 range.

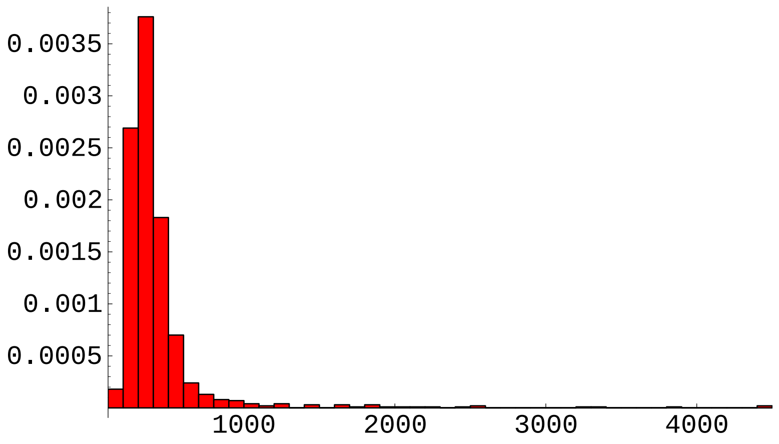

Figure 3: A histogram of the measured difficulty of 1000 generated puzzles. (drawn by Mathematica)

4.3.3 Standard Generator Runtime

For a rough runtime estimate, it took 3 minutes to generate 100 valid boards (and 30 invalid boards) and 12 minutes to determine the difficulties of the 100 valid boards. Thus, 100 boards take a total of about 15 minutes to run, so each board takes about 0.15 minutes on average. We can obtain an expected runtime estimate by taking the difficulty distribution of the 1000 generations of our standard generator (Figure 4), determining the proportion of the boards that fall in the given level, and then modeling the expected runtime as the expected value of a geometric random variable. For instance, for 4 levels as above, the expected number of boards one needs to construct to obtain a board of level 1 is a geometric random variable with parameter $p = \frac{598}{1000}$, so the expected runtime to obtain a board of level 1 is $0.15 \times \frac{1000}{598} = 0.25$ minutes. Similarly, the expected runtime to obtain a board of level 2, level 3, and level 4 are 3.1 minutes, 4.7 minutes, and 30 minutes, respectively. Expected runtimes for other numbers of levels can be computed similarly; we should note that for a division into a large number of levels, the expected runtime of obtaining a board at the highest level can be extremely long.

4.3.4 Using Pseudo-Generator to Generate Difficult Puzzles

If we wish to generate large numbers of difficult boards, it would be best to employ the pseudo-generator. In a trial run of the pseudo-generator, we fed the pseudo-generator a puzzle ("Riddle of Sho") that can only be solved using the tier 5 backtracking strategy [16]. The difficulty of the puzzle was determined to be 8122, while the average difficulty of 20 derived puzzles generated using this puzzle was 7111. By the constraint that all puzzles derived from a puzzle fed into the pseudo-generator must share the application of the most difficult strategy, the difficulty of the derived puzzles are approximately the same as the difficulty of the original puzzle; the exact difference and sign of the difference will depend on how difficult the original puzzle was up to the point of application of the difficult strategy.

Thus, provided that we have a large database of difficult puzzles, a method of employing the pseudo-generator is to find the midpoint of the difficulty bounds of the desired level, choose randomly a puzzle whose difficulty is close to this midpoint value, and generate a derived puzzle. If the difficulty of the derived puzzle fails to be within our bounds, we should continue choosing an existing puzzle at random and creating a derived puzzle until the bound condition is met. The average generation time for each puzzle was still 0.15 minutes, equal to that of the standard generator. Unfortunately, we cannot provide an analysis of the expected number of puzzles that need to be generated to fall within a particular difficulty interval due to the lack of a sufficiently large database of difficult puzzles to use as seed boards. It is clear, however, that for difficult boards, there is a huge difference in the expected number of boards that one needs to construct for the two strategies, and thus using the pseudo-generator will be much more efficient than using the standard generator in generating such puzzles of sufficiently high difficulty.

5 Conclusion

5.1 Strengths

Our human solver `hsolve` models the way a human Sudoku expert would solve a sudoku puzzle by posing sudoku as a search problem. We judge the relative costs of each strategy by the number of verifications of possible strategy applications necessary to find it and thereby avoid assigning explicit numerical difficulty values to specific strategies. Instead, we allow the difficulty of a strategy to emerge from the difficulty of finding it, giving a more formal treatment of what seems to be an intuitive notion. This derivation of the difficulty provides a more objective metric than that used in most existing difficulty ratings.

The resulting metric has a Goodman-Kruskal γ-coefficient of 0.82 with an existing set of hand-rated puzzles, and it generates a difficulty distribution that corresponds to one empirically generated by millions of users. Thus, we have some confidence that this new metric gives an accurate and reasonably fast method of rating Sudoku puzzle difficulties.

We produced two puzzle generators, one able to generate original puzzles that are mostly relatively easy to solve, and one able to modify pre-existing hard puzzles to create ones of similar difficulty. Given a database of difficult puzzles, our pseudo-generator is able to reliably generate many more puzzles of these difficulties.

5.2 Weaknesses

It was difficult to test the difficulty metric conclusively because of the dearth of available human-rated sudoku puzzles. Because of a lack of data on puzzle solution time, we were unable to conclusively establish what we believe to be a significant advantage of our difficulty metric over most existing ones.

While our puzzle generator was able to generate puzzles of all difficulties according to our metric, it experienced difficulty creating very hard puzzles, as they occurred quite infrequently. While we attempted to address this flaw by creating the pseudo-generator, this pseudo-generator is not able to create puzzles with entirely different final configurations.

Because of the additional computations required to calculate the search space for human behavior, both the difficulty metric and the puzzle generator have relatively slow runtime compared to other pre-existing raters and generator.

5.3 Alternative Approaches and Future Work

There were several approaches to both the difficulty metric and the generator that we could not pursue due to the time constraint. We considered modifying `hsolve` to search for all possible strategy applications, irrespective of strategy tier. In this case, we would search for the strategy with the least number of possible applications; this might allow us to remove our dependence on the tier system of strategies. However, we felt that our algorithm would run too slowly, and that this method did not take into account the prevailing frequencies of the various actions across all Sudoku boards.

For the generator, we considered modifying our pseudo-generator to operate on all fully solved boards by randomly removing squares that would require advanced techniques to replace. Once we found an advanced technique, we could then continue the algorithm of the pseudo-generator to place the remainder of the squares on the board. Unfortunately, we did not have sufficient time to implement or test this idea.

A Sudoku Strategies

Most (but not all) Sudoku puzzles can be solved using a series of logical deductions [19]. These deductions have been organized into a number of common patterns, which we have organized by difficulty. The strategies have been classed into *tiers* between 0 and 5 based upon the general consensus of many sources on their level of complexity (for example, see [10] and [20]).

In this work, we have used what seem to be the most commonly occurring and accessible strategies together with some simple backtracking. There are, of course, many more advanced strategies, but since our existing strategies suffice to solve almost all puzzles that we consider, we choose to ignore the more advanced ones.

0. Tier 0 Strategies

   - **Naked Single:** A Naked Single exists in the cell (i, j) if cell (i, j) on the board has no entry, but the corresponding entry (i, j) on the Sudoku Solution Graph has one and only one possible value. For example, in the example below:

1	2	3	4	5	6	7	8	?

We see that cell $(2, 9)$ is empty. Furthermore, the corresponding Sudoku Solution Graph entry in $(2, 9)$ can only contain the number 9, since the numbers 1 through 8 are already assigned to cells in row 2. Therefore, since cell $(2, 9)$ in the corresponding Sudoku Solution Graph only has one (naked) value, we can assign that value to cell $(2, 9)$ on the sudoku board.

Application Enumeration: Since a Naked Single could occur in any empty cell, this is just the number of empty cells, since checking if any empty cell is a Naked Single requires constant time.

   - **Hidden Single:** A Hidden Single occurs in a given cell (i, j) when:

     (a) (i, j) has no entry on the Sudoku board

     (b) (i, j) contains the value k (among other values) on the Sudoku Solution Graph

     (c) No other cell in the same group as (i, j) has k as a value in its Sudoku Solution Graph

     Once we find a hidden single in (i, j) with value k, we assign k to (i, j) on the Sudoku board. The logic behind hidden singles is that given any group, all numbers

1 through 9 must appear exactly once. If we know cell (i, j) is the only cell that could contain the value k in a given row, then we know that it must hold value k on the actual Sudoku board. We can consider the example:

?								
	3	4	1					
	5	6				1		
	1							
		1						

We look at cell $(1, 1)$. First, $(1, 1)$ does not have an entry, and we can see that its corresponding entry in the Sudoku Solution Graph contains $\{1, 2, 7, 8, 9\}$. However, we see that the other cells in region 1 that don't have values assigned, i.e. cells $(1, 2), (1, 3), (2, 1)$ and $(3, 1)$, do not have the value 1 in their corresponding Sudoku Solution Graph cells; that is, none of the other four empty cells in the board besides $(1, 1)$ can hold the value 1, and so we can assign 1 to the cell $(1, 1)$.

Application Enumeration: Since a Hidden Single could occur in any empty cell, this is just the number of empty cells, since checking if any empty cell is a Hidden Single requires constant time (inspecting other cells in the same group).

1. Tier 1 Strategies

- **Naked Double:** A Naked Double occurs when two cells on the board in the same group g do not have values assigned, and both their corresponding cells in the Sudoku Solution Graph have only the same two values k_1 and k_2 assigned to them. A naked double in (i_1, j_1) and (i_2, j_2) does not immediately give us the values contained in either (i_1, j_1) or (i_2, j_2), but it does allows us to eliminate k_1 and k_2 from the Sudoku Solution Graph of all cells in g beside (i_1, j_1) and (i_2, j_2).

  Application Enumeration: For each row, column, and region, we sum up $\binom{n}{2}$ where n is the number of empty cells in each group, since a Naked Double requires two empty cells in the same group.

- **Hidden Double:** A Hidden Double occurs in two cells (i_1, j_1) and (i_2, j_2) in the same group g when:

  (a) (i_1, j_1) and (i_2, j_2) have no values assigned on the board

  (b) (i_1, j_1) and (i_2, j_2) share two entries k_1 and k_2 (and contain possibly more) in the Sudoku Solution Graph

  (c) k_1 and k_2 do not appear in any other cell in group g on the Sudoku Solution Graph

A hidden double does not allow us to immediately assign values to (i_1, j_1) or (i_2, j_2), but it does allow us to eliminate all entries other than k_1 and k_2 in the Sudoku Solution Graph for cells (i_1, j_1) and (i_2, j_2).

Application Enumeration: For each row, column, and region, we sum up $\binom{n}{2}$ where n is the number of empty cells in each group, since a Hidden Double requires two empty cells in the same group.

- **Locked Candidates:** A Locked Candidate occurs if we have cells (for simplicity, suppose we only have two: (i_1, j_1) and (i_2, j_2)) such that:

  (a) (i_1, j_1) and (i_2, j_2) have no entries on the board
  (b) (i_1, j_1) and (i_2, j_2) share two groups, g_1 and g_2 (i.e. both cells are in the same row and region, or the same column and region)
  (c) (i_1, j_1) and (i_2, j_2) share some value k in the Sudoku Solution Graph
  (d) $\exists g_3$, a group of the same type as g_1, $g_1 \neq g_3$, such that k occurs in cells of $g_2 \cap g_3$
  (e) k does not occur elsewhere in g_3 besides $g_3 \cap g_2$
  (f) k does not occur in g_2 aside from $(g_2 \cap g_1) \cup (g_2 \cap g_3)$

  Then, since k must occur at least once in g_3, we know k must occur in $g_2 \cap g_3$. However, since k can only occur once in g_2, then k cannot occur in $g_2 \cap g_1$, so we can eliminate k from the Sudoku Solution Graph cells corresponding to (i_1, j_1) and (i_2, j_2). A locked candidate can also occur with three cells.

  Application Enumeration: For every row i, we examine each three-cell subset rs_{ij} formed as the intersection with some region j; there are twenty-seven such subsets. Out of those twenty-seven, we denote the number of subsets that have two or three empty cell as r_l. We define c_l for columns analogously, so this is just the sum $r_l + c_l$.

2. Tier 2 Strategies

- **Naked Triple:** A Naked Triple occurs when three cells on the board, $(i_1, j_1), (i_2, j_2)$ and (i_3, j_3), in the same group g do not have values assigned, and all three of their corresponding cells in the Sudoku Solution Graph share only the same three possible values, k_1, k_2 and k_3. However, each cell of a Naked Triple does not have to have all three values, e.g. we can have (i_1, j_1) have values k_1, k_2 and k_3, (i_2, j_2) have k_2, k_3 and (i_3, j_3) have k_1 and k_3 on the Sudoku Solution Graph. We can remove k_1, k_2 and k_3 from all cells except for $(i_1, j_1), (i_2, j_2)$ and (i_3, j_3) in the Sudoku Solution Graph that are also in group g; the logic is similar to that of the Naked Double strategy.

  Application Enumeration: For each row, column, and region, we sum up $\binom{n}{3}$ where n is the number of empty cells in each group, since a Naked Triple requires three empty cells in the same group.

- **Hidden Triple:** A Hidden Triple is similar to a Naked Triple the way a Hidden Double is similar to a Naked Double, and occurs in cells $(i_1, j_1), (i_2, j_2)$ and (i_3, j_3) sharing the same group g when:

(a) $(i_1, j_1), (i_2, j_2)$ and (i_3, j_3) contain no values on the Sudoku Board
(b) Values k_1, k_2 and k_3 appear among $(i_1, j_1), (i_2, j_2)$ and (i_3, j_3) in their SSG
(c) k_1, k_2 and k_3 do not appear in any other cells of g in the SSG

Then, we can eliminate all values beside k_1, k_2 and k_3 in the SSG of cells $(i_1, j_1), (i_2, j_2)$ and (i_3, j_3). The reasoning is the same as for the Hidden Double strategy.

Application Enumeration: For each row, column, and region, we sum up $\binom{n}{3}$ where n is the number of empty cells in each group, since a Hidden Triple requires three empty cells in the same group.

- **X-Wing:** Given a value k, an X-Wing occurs if:

  (a) $\exists$ two rows, r_1 and r_2, such that the value k appears in the SSG for exactly two cells each of r_1 and r_2
  (b) $\exists$ distinct columns c_1 and c_2 such that k only appears in rows r_1 and r_2 the Sudoku Solution Graph in the set $(r_1 \cap c_1) \cup (r_1 \cap c_2) \cup (r_2 \cap c_1) \cup (r_2 \cap c_2)$

  Then, we can eliminate the value k as a possible value for all cells in c_1 and c_2 that are not also in r_1 and r_2, since k can only appear in each of the two possible cells of in each row r_1 and r_2 and k. Similarly, the X-Wing strategy can also be applied if we have a value k that is constrained in columns c_1 and c_2 in exactly the same two rows.

  Application Enumeration: For each value k, 1 through 9, we count the number of rows that contain k exactly twice in the SSG of its empty cells, r_k. Since we need two such rows to form an X-Wing for any one number we take $\binom{r_k}{2}$. We also count the number of columns that contain k exactly twice in the SSG of its cells, c_k, and similarly take $\binom{c_k}{2}$. We sum over all values k, so this value is $\sum_k \binom{r_k}{2} + \binom{c_k}{2}$.

3. Tier 3 Strategies

- **Naked Quad:** A Naked Quad is similar to a Naked Triple; it occurs when four unfilled cells in the same group g contain only elements of set K of at most four possible values in their SSG. In this case, we can remove all values in K from all other cells in group g, since the values in K must belong only to the four unfilled cells.

  Application Enumeration: For each row, column, and region, we sum up $\binom{n}{4}$ where n is the number of empty cells in each group, since a Naked Quad requires three four empty cells in the same group.

- **Hidden Quad:** A Hidden Quad is analogous to a Hidden Triple. It occurs when we have four cells $(i_1, j_1), (i_2, j_2), (i_3, j_3)$ and (i_4, j_4) in the same group g such that:

  (a) $(i_1, j_1), (i_2, j_2), (i_3, j_3)$ and (i_4, j_4) share (among other elements) elements of the set K of at most four possible values in their SSG
  (b) No values of K appear in the SSG of any other cell in g

Then we can eliminate all values that cells $(i_1, j_1), (i_2, j_2), (i_3, j_3)$ and (i_4, j_4) take on other than values in K from their corresponding cells in the Sudoku Solution Graph. The reasoning is analogous to the Hidden Triple strategy.

Application Enumeration: For each row, column, and region, we sum up $\binom{n}{4}$ where n is the number of empty cells in each group, since a Hidden Quad requires three four empty cells in the same group.

- **Swordfish:** The Swordfish Strategy is the three-row analogue to the X-Wing Strategy. Suppose we have three rows, r_1, r_2 and r_3, such that the value k has not been assigned to any cell in r_1, r_2 or r_3. If the cells of r_1, r_2 and r_3 that have k as a possibility in their corresponding SSG are all in the same three columns c_1, c_2 and c_3, then no other cells in c_1, c_2 and c_3 can take on the value k, so we may eliminate the value k from the corresponding cells in the SSG. (This strategy can also be applied if we have columns that restrict the occurrence of k to three rows).

  Application Enumeration: For each value k, 1 through 9, we count the number of rows that contain k exactly two or three times in the SSG of its empty cells, r_k. Since we need three such rows to form a Swordfish for any one number we take $\binom{r_k}{3}$. We also count the number of columns that contain k two or three times in the SSG of its cells, c_k, and similarly take $\binom{c_k}{3}$. We sum over all values k, so this value is $\sum_k \binom{r_k}{3} + \binom{c_k}{3}$.

4. Tier 4 Strategies

- **Jellyfish:** The Jellyfish Strategy is analogous to the Swordfish and X-Wing strategies. We apply similar reasoning to four rows r_1, r_2, r_3 and r_4 in which some value k is restricted to the same four columns c_1, c_2, c_3 and c_4. If the appearance of k in cells of r_1, r_2, r_3 and r_4 in the Sudoku Solution Graph is restricted to four specific columns, then we can eliminate k from any cells in c_1, c_2, c_3 and c_4 that are not in one of r_1, r_2, r_3 or r_4. Like the Swordfish strategy, the Jellyfish strategy may also be applied to columns instead of rows.

  Application Enumeration: For each value k, 1 through 9, we count the number of rows that contain k exactly two, three or four times in the SSG of its empty cells, r_k. Since we need four such rows to form a Jellyfish for any one number k, we take $\binom{r_k}{4}$. We also count the number of columns that contain k two, three or four times in the SSG of its cells, c_k, and similarly take $\binom{c_k}{4}$. We sum over all values k, so this value is $\sum_k \binom{r_k}{4} + \binom{c_k}{4}$.

5. Tier 5 Strategies

- **Backtracking:** Backtracking in the sense that we use is a limited version of complete search. When cell (i, j) has no assigned value, but exactly 2 possible values k_1, k_2 in its SSG, the solver will assign a test value (assume k_1) to cell (i, j) and continue solving the puzzle using only Tier 0 strategies.

  There are three possible results. If the solver arrives at a contradiction, he deduces that k_2 is in cell (i, j). If the solver completes the puzzle using the test value, this is the unique solution and the puzzle is solved. Otherwise, if the solver cannot proceed further but has not solved the puzzle completely, backtracking has failed and the solver must start a different strategy.

Application Enumeration: Since we only apply Backtracking to cells with exactly two values in its SSG, this is just the number of empty cells that have exactly two values in their SSG.

References

[1] Caine, Allan; Cohen, Robin. MITS: A Mixed-Initiative Intelligent Tutoring System for Sudoku. *Advances in Artificial Intelligence.* (2006), 550–561

[2] Cox, Kenneth C.; Eick, Stephen G.; Wills, Graham J.; Brachman, Ronald J. Brief Application Description; Visual Data Mining: Recognizing Telephone Calling Fraud. *Data Mining and Knowledge Discovery.* (1997) 225–331.

[3] Emery, Michael Ray. Solving Sudoku Puzzles with the COUGAAR Agent Architecture. Thesis, 2007. Available at `http://www.cs.montana.edu/techreports/2007/MichaelEmery.pdf`.

[4] Eppstein, David. Nonrepetitive Paths and Cycles in Graphs with Application to Sudoku. Preprint, 2005. Available at `http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0507053`.

[5] Felgenhauer, Bertram; Jarvis, Frazer. Enumerating possible Sudoku grids. Preprint, 2005. Available at `http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf`.

[6] Goodman, Leo A.; Kruskal, William H. "Measures of Association for Cross Classifications." *Journal of the American Statistical Association*, Vol. 49, No. 268. (Dec. 1954) pp. 732-764.

[7] GraphPad Software. "QuickCalcs: Online Calculators for Scientists." Available at `http://www.graphpad.com/quickcalcs/PValue1.cfm`.

[8] Hanssen, Vegard. "Sudoku Puzzles." *Sudoku Puzzles.* URL: `http://www.menneske.no/sudoku/eng/`.

[9] Hayes, Brian. Unwed Numbers: The mathematics of Sudoku, a puzzle that boasts 'No Math Required!' *American Scientist Online.* (2006) Available at `http://www.americanscientist.org/template/AssetDetail/assetid/48550?print=yes`.

[10] Johnson, Angus. "Solving Sudoku." *Simple Sudoku.* URL: `http://www.angusj.com/sudoku/hints.php`.

[11] Knuth, Donald Ervin. Dancing Links. in: *Millennial Perspectives in Computer Science.* (2000), 187–214. `http://arxiv.org/PS_cache/cs/pdf/0011/0011047v1.pdf`.

[12] Lenz, Moritz. "Sudoku Garden." URL: `http://sudokugarden.de/en`.

[13] Lewis, Rhyd. Metaheuristics can solve sudoku puzzles. *Journal of Heuristics.* **13**. No. 4 (2007), 387–401.

[14] Lynce, Inês; Ouaknine, Joël. Sudoku as a SAT Problem. Preprint, 2006. Available at `http://sat.inesc-id.pt/~ines/publications/aimath06.pdf`.

[15] Mantere, Timo; Koljonen, Janne. Solving and Rating Sudoku Puzzles with Genetic Algorithms, in *Proceedings of the 12th Finnish Artificial Intelligence Conference STeP* (2006). Available at `http://www.stes.fi/scai2006/proceedings/step2006-86-mantere-solving-and-rating-sudoku-puzzles.pdf`.

[16] "Sudoku Solver." URL: `http://www.scanraid.com/sudoku.htm`.

[17] Simonis, Helmut. "Sudoku as a Constraint Problem," in *Modelling and Reformulating Constraint Satisfaction Problems* (2005). Available at `http://homes.ieu.edu.tr/~bhnich/mod-proc.pdf#page=21`.

[18] "Sudoku." *Times Online*. URL: `http://entertainment.timesonline.co.uk/tol/arts_and_entertainment/games_and_puzzles/sudoku/`.

[19] "What is Sudoku?" *Sudoku Addict*. URL: `http://www.sudokuaddict.com`.

[20] "Sudoku Strategy." *Sudoku Dragon*. URL: `http://www.sudokudragon.com/sudokustrategy.htm`.

[21] Web Sudoku. URL: `http://www.websudoku.com/`.

[22] Yato, Takayuki. "Complexity and Completeness of Finding Another Solution and its Application to Puzzles." Thesis, January 2003. Available at `http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf`.