A Difficulty Metric and Puzzle Generator for Sudoku

Christopher Chang
Zhou Fan
Yi Sun

Harvard University
Cambridge, MA

Advisor: Clifford H. Taubes

Abstract

We present here a novel solution to creating and rating the difficulty of Sudoku puzzles. We frame Sudoku as a search problem and use the expected search time to determine the difficulty of various strategies. Our method is relatively independent from external views on the relative difficulties of strategies.

Validating our metric with a sample of 800 puzzles rated externally into eight gradations of difficulty, we found a Goodman-Kruskal γ coefficient of 0.82, indicating significant correlation [Goodman and Kruskal 1954]. An independent evaluation of 1,000 typical puzzles produced a difficulty distribution similar to the distribution of solve times empirically created by millions of users at `http://www.websudoku.com`.

Based upon this difficulty metric, we created two separate puzzle generators. One generates mostly easy to medium puzzles; when run with four difficulty levels, it creates puzzles (or *boards*) of those levels in 0.25, 3.1, 4.7, and 30 min. The other puzzle generator modifies difficult boards to create boards of similar difficulty; when tested on a board of difficulty 8,122, it created 20 boards with average difficulty 7,111 in 3 min.

Introduction

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

The UMAP Journal 29 (3) (2008) 305–326. ©Copyright 2008 by COMAP, Inc. All rights reserved. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice. Abstracting with credit is permitted, but copyrights for components of this work owned by others than COMAP must be honored. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior permission from COMAP.

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.

Figure 1 shows a Sudoku puzzle and its solution.

			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

Figure 1. Sudoku puzzle and solution from the London *Times* (16 February 2008) [Sudoku n.d.].

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

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 in **Figure 2**.

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

Given a Sudoku board B, define the *Sudoku Solution Graph* (SSG) $S(B)$ to be the structure that associates to each cell in B the set of digits currently thought to be candidates for the cell. For example, in **Figure 1**, 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.

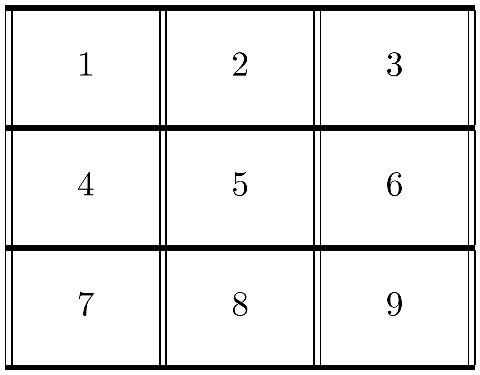


Figure 2. Numbering of 3 × 3 regions of a Sudoku board.

To solve a Sudoku board, a player applies strategies, patterns of logical deduction (see the **Appendix**). We assume the SSG has been evaluated for every cell on the board before any strategies are applied.

Problem Background

Most efforts on Sudoku have been directed at solving puzzles or analyzing the computational complexity of solving Sudoku [Lewis 2007, Eppstein 2005, and Lynce and Ouaknine 2006]. Sudoku can be solved extremely quickly via reduction to an exact cover problem and an application of Knuth's Algorithm X [2000]. However, solving the $n^2 \times n^2$ generalization of Sudoku is known to be NP-complete [Yato 2003].

We investigate:

1. Given a puzzle, how does one define and determine its difficulty?
2. Given a difficulty, how does one generate a 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 involve rating a puzzle by the strategies necessary to find the solution, while other approaches have been proposed by Caine and Cohen [2006] and Emery [2007]. A genetic algorithms approach found some correlation with human-rated difficulties [Mantere and Koljonen 2006], and Simonis presents similar findings with a constraint-based rating [2005]. However, in both cases, the correlation is not clear.

Puzzle generation seems to be more difficult. Most existing generators use complete search algorithms to add numbers systematically 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 Mantere and Koljonen [2006], while Simonis [2005] posits both this and a similar method based on removal of cells from a completed board. Felgenhauer and Jarvis [2005] calculate the number of valid Sudoku puzzles.

We present a new approach. We create `hsolve`, a program to simulate how a human solver approaches a puzzle, and present a new difficulty metric based upon `hsolve`'s simulation of human solving behavior. We propose two methods based on `hsolve` to generate puzzles of varying difficulties.

Problem Setup

Difficulty Metric

We create an algorithm that takes a puzzle and returns a real number that represents its abstract "difficulty" according to some metric. We base our definition of difficulty on the following general assumptions:

1. The amount of time for a human to solve a puzzle increases monotonically with difficulty.
2. Every solver tries various strategies. To avoid the dependence of our results on a novice's ignorance of strategies 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 that a hypothetical Sudoku expert would spend solving it*.

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. We take a sample of 1,000 Sudoku puzzles and assume that they are representative of the difficulty distribution of all puzzles. We also endeavor to minimize the complexity of the generation algorithm, measured as the expected execution time to find a puzzle of the desired difficulty level.

A Difficulty Metric

Assumptions and Metric Development

To measure the time for an expert Sudoku solver to solve a puzzle, there are two possibilities:

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

There are known heuristics for difficulty of a puzzle—for example, puzzles with a small number of initial givens are somewhat harder than most. However, according to Hayes [2006], the overall correlation is weak.

Therefore, we must model the process of solving. We postulate the following assumptions for the solver:

1. Strategies can be ranked in order of difficulty, and the solver always applies them from least to most difficult. This assumption is consistent with the literature. We use a widely accepted ranking of strategies described in the **Appendix**.
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, Cox et al. [1997]), 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 among 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 the **Appendix**. We model our solver as following the algorithm `HumanSolve` defined as follows:

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*; we assume that each check takes the same amount of time. Multiple runs of this method on the same puzzle may have different difficulties, due to different valid applications being recognized first.

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

`hsolve` and Metric Calculation

To calculate $m(B)$, we use `hsolve`, a program in Java 1.6 that simulates `HumanSolve` and calculates the resulting difficulty:

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 $E(p, v) \times t$, 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. The following lemma gives an extremely convenient closed-form expression for $E(p, v)$ that we use in `hsolve`.

Lemma. *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

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

where we've used the "hockeystick identity" [AoPS Inc. 2007]. □

Given a puzzle B, we calculate $m(B)$ by running `hsolve` several times and take the average of the returned difficulties. Doing 20 runs per puzzle gives a ratio of standard deviation to mean of $\frac{\sigma}{\mu} \approx \frac{1}{10}$, so we use 20 runs per puzzle.

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.

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 Hanssen [n.d.]. No other large datasets with varying difficulty ratings were available.

We ran `hsolve` on each puzzle 20 times and recorded the average difficulty for each board. We classified boards by difficulty on a ranking scale, with 8 groups of 100 puzzles. **Table 1** shows the results.

Table 1.

Results: $\chi^2 = 6350$ (df $= 49$), $\gamma = 0.82$.

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

A χ^2-test for independence gives $\chi^2 = 6350$ ($p < 0.0001$). Thus, there is a statistically significant deviation from independence.

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

Validation of Difficulty Distribution

When run 20 times on each of 1,000 typical puzzles from Lenz [n.d.], `hsolve` generates the distribution for measured difficulty shown in **Figure 3**. The distribution is sharply peaked near 500 and has a long tail

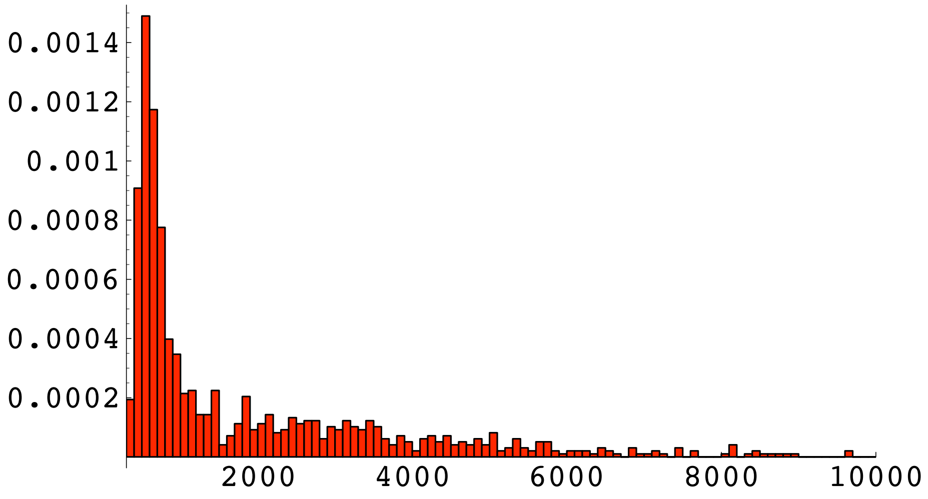


Figure 3. Histogram of measured difficulty for 1,000 typical puzzles.

towards higher difficulty.

We compare this difficulty distribution plot with the distributions of times required for visitors to `http://www.websudoku.com` to solve the puzzles available there [Web Sudoku n.d.]. This distribution, generated by the solution times of millions of users, is shown in **Figure 4**.

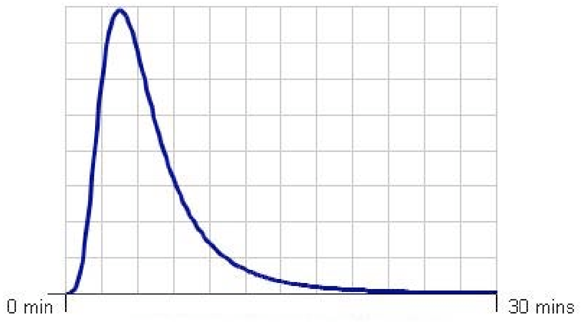


Figure 4. A distribution plot of the time to solve Easy-level puzzles on `www.websudoku.com`; the mean is 5 min 22 sec.

The two graphs share a peak near 0 and are skewed to the right.

Runtime

With running 20 iterations of `hsolve` per puzzle, rating 100 puzzles requires 13 min, or about 8 sec per puzzle, on a 2 Ghz Centrino Duo processor with 256 MB of Java heap space. While this runtime is slower than existing difficulty raters, we feel that `hsolve` provides a more detailed evaluation of difficulty that justifies the extra time.

Generator

Our choice of using a solver-based metric for difficulty has the following implications for puzzle generation:

- 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 that we take in both the generator and pseudo-generator described below.

- There is an inevitable trade-off between the ability to generate consistently difficult puzzles and the ability to generate truly random puzzles. A generator that creates puzzles with as randomized a process as possible is unlikely to create very difficult puzzles, since complex strategies would not be employed very often.

  Hence, for a procedure that consistently generates hard puzzles, we must either reduce the randomness in the puzzle-generating process or limit the types of puzzles that can result.

- The speed at which puzzles can be generated depends upon the speed of `hsolve`.

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

Standard Generator

Our standard puzzle generator follows this algorithm:

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. (That is, 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 a contradiction is reached or 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 a new cell is filled in manually at each iteration.

The final puzzle is the board with all of the numbers that were filled in manually at each iteration of the algorithm (i.e., the board without the numbers filled in by `hsolve`).

Guaranteeing a Unique Solution with Standard Generator

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.

Pseudo-Generator

Our pseudo-generator takes a completed Sudoku board and a set of cells to leave empty at beginning of a puzzle, called the *reserved cells*. The idea is to guarantee the use of a high-level strategy, such as Swordfish or Backtracking, by ensuring that a generated puzzle cannot be completed without such a strategy. Call the starting puzzle the *seed board* and the solution the *completed seed board*. To use the 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 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 the pseudo-generator the completed seed board and this list of reserved cells. The pseudo-generator iterates the 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 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.

Differences From Standard Generator

The main differences between the pseudo-generator and the standard generator are:

1. When filling in an empty cell, the standard 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, the pseudo-generator never selects one of the reserved cells.
3. `hsolve` is equipped with strategies only up to tier k.
4. The pseudo-generator terminates not when the board is completely filled in but rather when all of the unreserved cells are filled in.

The pseudo-generator is only partially random. It provides enough clues so that the unreserved cells of the board can be solved with strategies up to tier k, and the choice of which of these cells to reveal as clues is determined randomly. However, 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 must be unique.

The pseudo-generator never provides clues for reserved cells; hence, when `hsolve` solves a puzzle, it uses strategies of tiers 0 through k to fill in the unreserved cells, and then is forced to use a strategy in tier $k + 1$ to solve the remaining portion of the board.

Pseudo-Generator Puzzle Variability

The benefit of the pseudo-generator over the standard generator is generating puzzles in which a strategy of tier $k + 1$ must be used, thus guaranteeing a high level of difficulty (if k is high). The drawback is that the pseudo-generator cannot be said to generate a puzzle at random, since it starts with a puzzle already generated in the past and constructs a new puzzle (using some random choices) out of its solution.

We implement the pseudo-generator by first randomly permuting the rows, columns, and numbers of the given completed puzzle, so as to create an illusion that it is a different puzzle. Ideally, we should have a large database of difficult puzzles to choose from (together with the highest tier strategy needed to solve each puzzle and its list of reserved cells that cannot be filled with strategies of lower tiers).

Difficulty Concerns

"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 previous correlation analysis in which we divided 800 puzzles into eight difficulty levels, we forced each difficulty level to contain 100 puzzles.

Generating Puzzles with a Specific Difficulty

Figure 5 shows the measured difficulty of 1,000 puzzles generated by the standard generator. We can divide the puzzles into intervals of difficulty, with equal numbers of puzzles in each interval. To create a puzzle of given difficulty level using the standard generator, we iterate the generator until a puzzle is generated whose difficulty value falls within the appropriate interval.

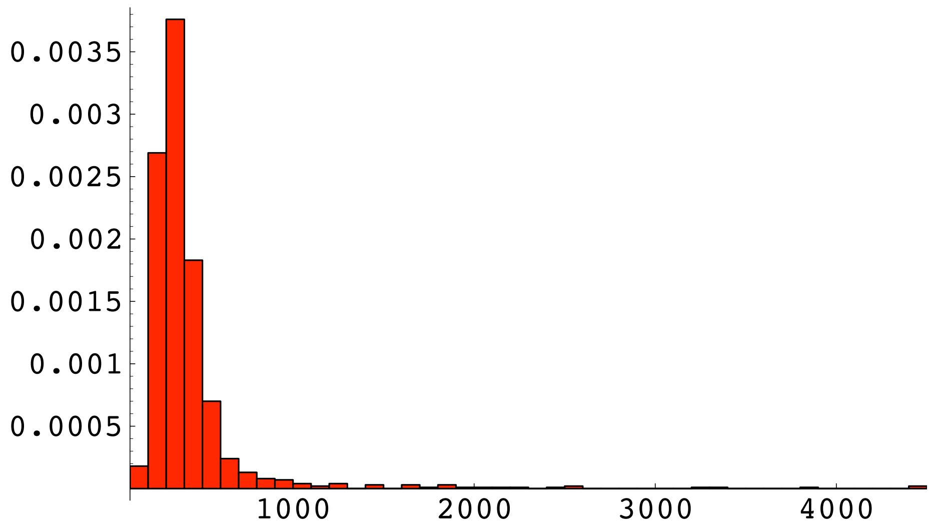


Figure 5. A histogram of the measured difficulty of 1,000 puzzles generated by the standard generator.

Standard Generator Runtime

It took 3 min to generate 100 valid boards (and 30 invalid boards) and 12 min to determine the difficulties of the 100 valid boards. Thus, 100 boards take a total 15 min to run, or an average of about 9 sec per valid board.

From the difficulty distribution in **Figure 5**, we can obtain an expected runtime estimate for each level of difficulty. For four levels, the expected number of boards that 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 $0.15 \times \frac{1000}{598} = 0.25$ min. Similarly, the expected runtimes to obtain boards of level 2, level 3, and level 4 are 3.1, 4.7, and 30 min.

Using Pseudo-Generator to Generate Difficult Puzzles

To generate large numbers of difficult boards, it would be best to employ the pseudo-generator. We fed the pseudo-generator a puzzle ("Riddle of Sho") that can be solved only by using the tier-5 backtracking strategy [Sudoku Solver n.d.]. The difficulty of the puzzle was determined to be 8,122, while the average difficulty of 20 derived puzzles generated using this puzzle was 7,111. Since all puzzles derived from a puzzle fed into the pseudo-generator must share application of the most difficult strategy, the difficulties of the derived puzzles are approximately the same as that of the original puzzle.

With a 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, and generate a derived puzzle. If the difficulty of the derived puzzle fails to be within our bounds, we continue choosing an existing puzzle at random and creating a derived puzzle until the bound condition is met. The average generation time for a puzzle is 9 sec, the same as for the standard generator. For difficult boards, there is a huge difference between the two strategies in the expected number of boards that one needs to construct, and the pseudo-generator is much more efficient.

Conclusion

Strengths

Our human solver `hsolve` models how 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 preexisting 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.

Weaknesses

It was difficult to test the difficulty metric conclusively because of the dearth of available human-rated Sudoku puzzles. Hence, we could not conclusively establish what we believe to be a significant advantage of our difficulty metric over most existing ones.

While our puzzle generator generated puzzles of all difficulties according to our metric, it experienced difficulty creating very hard puzzles, as they occurred quite infrequently. Although we attempted to address this flaw by creating the pseudo-generator, it cannot 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 runtimes compared to other raters and generator.

Appendix: Sudoku Strategies

Most (but not all) Sudoku puzzles can be solved using a series of logical deductions [What is Sudoku? n.d.]. 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 Johnson [n.d.] and Sudoku Strategy [n.d.]).

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 **Figure A1**. We see that cell $(2, 9)$ is empty. Furthermore,

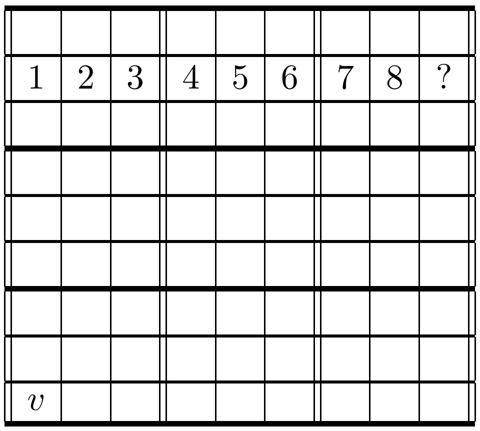


Figure A1. Example for Naked Single strategy.

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 in **Figure A2**.

  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 $(1, 1)$.

?								
	3	4	1					
	5	6				1		
	1							
		1						

Figure A2. Example for Hidden Single strategy.

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

AoPS Inc. 2007. Combinatorics and sequences. `http://www.artofproblemsolving.com/Forum/viewtopic.php?t=88383`.

Caine, Allan, and Robin Cohen. 2006. MITS: A Mixed-Initiative Intelligent Tutoring System for Sudoku. *Advances in Artificial Intelligence* 550–561.

Cox, Kenneth C., Stephen G. Eick, Graham J. Wills, and Ronald J. Brachman. 1997. Brief application description; visual data mining: Recognizing telephone calling fraud. *Data Mining and Knowledge Discovery* 225–331.

Emery, Michael Ray. 2007. Solving Sudoku puzzles with the COUGAAR agent architecture. Thesis. `http://www.cs.montana.edu/techreports/2007/MichaelEmery.pdf`.

Eppstein, David. 2005. Nonrepetitive paths and cycles in graphs with application to Sudoku. `http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0507053`.

Felgenhauer, Bertram, and Frazer Jarvis. 2005. Enumerating possible Sudoku grids. `http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf`.

Goodman, Leo A., and William H. Kruskal. 1954. Measures of association for cross classifications. *Journal of the American Statistical Association*, 49 (December 1954) 732–764.

GraphPad Software. n.d. QuickCalcs: Online calculators for scientists. `http://www.graphpad.com/quickcalcs/PValue1.cfm`.

Hanssen, Vegard. n.d. Sudoku puzzles. `http://www.menneske.no/sudoku/eng/`.

Hayes, Brian. 2006. Unwed numbers: The mathematics of Sudoku, a puzzle that boasts "No math required!" *American Scientist Online* `http://www.americanscientist.org/template/AssetDetail/assetid/48550?print=yes` .

Johnson, Angus. n.d. Solving Sudoku. `http://www.angusj.com/sudoku/hints.php` .

Knuth, Donald Ervin. 2000. Dancing links. In *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Professor Sir Antony Hoare*, edited by Jim Davies, Bill Roscoe, and Jim Woodcock, 187–214. Basingstoke, U.K.: Palgrave Macmillan. `http://www-cs-faculty.stanford.edu/~uno/preprints.html` .

Lenz, Moritz. n.d. Sudoku Garden. `http://sudokugarden.de/en` .

Lewis, Rhyd. 2007. Metaheuristics can solve sudoku puzzles. *Journal of Heuristics* 13 (4): 387–401.

Lynce, Inês, and Joël Ouaknine. 2006. Sudoku as a SAT Problem. `http://sat.inesc-id.pt/~ines/publications/aimath06.pdf` .

Mantere, Timo, and Janne Koljonen. 2006. Solving and rating Sudoku puzzles with genetic algorithms. In *Proceedings of the 12th Finnish Artificial Intelligence Conference STeP*.
`http://www.stes.fi/scai2006/proceedings/step2006-86-solving-and-rating-sudoku-puzzles.pdf` .

Simonis, Helmut. 2005. Sudoku as a constraint problem. In *Modelling and Reformulating Constraint Satisfaction*, edited by Brahim Hnich, Patrick Prosser, and Barbara Smith, 13–27. `http://homes.ieu.edu.tr/~bhnich/mod-proc.pdf#page=21` .

Sudoku. n.d. *Times Online*. `http://entertainment.timesonline.co.uk/tol/arts_and_entertainment/games_and_puzzles/sudoku/`.

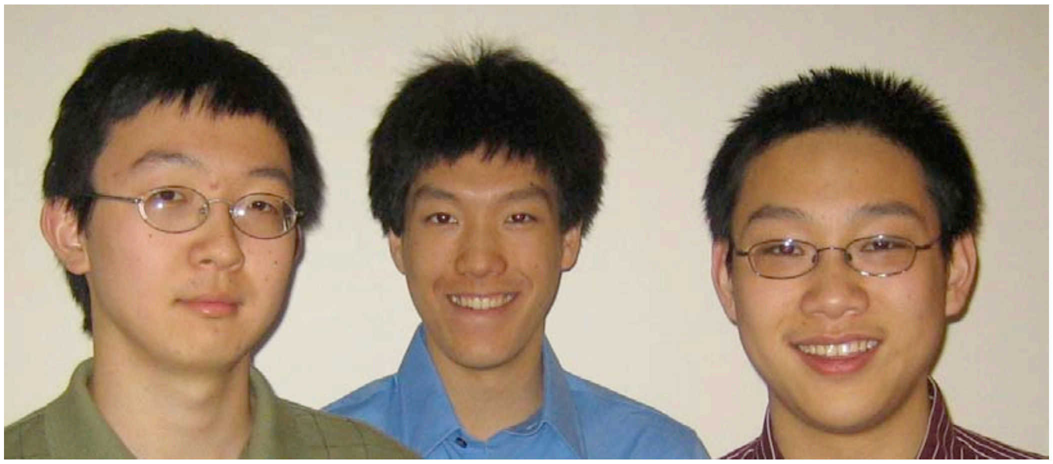
Sudoku solver. `http://www.scanraid.com/sudoku.htm`.

Sudoku strategy. n.d. *Sudoku Dragon*. `http://www.sudokudragon.com/sudokustrategy.htm` .

Web Sudoku. n.d. URL: `http://www.websudoku.com/`.

What is Sudoku? n.d. `http://www.sudokuaddict.com` .

Yato, Takayuki. 2003. Complexity and completeness of finding another solution and its application to puzzles. Thesis, January 2003. `http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf` .



Zhou Fan, Christopher Chang, and Yi Sun.