Taking the Mystery Out of Sudoku Difficulty: An Oracular Model

Sarah Fletcher
Frederick Johnson
David R. Morrison

Harvey Mudd College
Claremont, CA

Advisor: Jon Jacobsen

Summary

In the last few years, the 9-by-9 puzzle grid known as Sudoku has gone from being a popular Japanese puzzle to a global craze. As its popularity has grown, so has the demand for harder puzzles whose difficulty level has been rated accurately.

We devise a new metric for gauging the difficulty of a Sudoku puzzle. We use an oracle to model the growing variety of techniques prevalent in the Sudoku community. This approach allows our metric to reflect the difficulty of the puzzle itself rather than the difficulty with respect to some particular set of techniques or some perception of the hierarchy of the techniques. Our metric assigns a value in the range $[0, 1]$ to a puzzle.

We also develop an algorithm that generates puzzles with unique solutions across the full range of difficulty. While it does not produce puzzles of a specified difficulty on demand, it produces the various difficulty levels frequently enough that, as long as the desired score range is not too narrow, it is reasonable simply to generate puzzles until one of the desired difficulty is obtained. Our algorithm has exponential running time, necessitated by the fact that it solves the puzzle it is generating to check for uniqueness. However, we apply an algorithm known as Dancing Links to produce a reasonable runtime in all practical cases.

The UMAP Journal 29 (3) (2008) 327–341.

Introduction

The exact origins of the Sudoku puzzle are unclear, but the first modern "Sudoku" puzzle showed up under the name "Number Place" in a 1979 puzzle magazine put out by Dell Magazines. Nikoli Puzzles introduced the puzzle to Japan in 1984, giving it the name "Suuji wa dokushin ni kagiru," which was eventually shortened to the current "Sudoku." In 1986, Nikoli added two new constraints to the creation of the puzzle: There should be no more than 30 clues (or givens), and these clues must be arranged symmetrically. With a new name and a more esthetically-pleasing board, the game immediately took off in Japan. In late 2004, Sudoku was introduced to the London *Times*; and by the summer of 2005, it had infiltrated many major American newspapers and become the latest puzzle craze [Wikipedia 2008b].

Sudopedia is a Website that collects and organizes electronic information on Sudoku, including solving techniques, from how do deal with "Fishy Cycles" and "Squirmbags" to identifying "Skyscrapers" and what to do if you discover that you have a "Broken Wing." It even explains the possibilities for what has happened if you find yourself hopelessly buried in a "Bivalue Universal Grave." Some techniques are more logically complex than others, but many of similar complexity seem more natural to different players or are more powerful in certain situations. This situation makes it difficult to use specific advanced techniques in measuring the difficulty of a puzzle.

Our goal is a metric to rate Sudoku puzzles and an algorithm to generate them. A useful metric should reflect the difficulty as perceived by humans, so we analyze how humans approach the puzzle and use the conclusions as the basis for the metric. In particular, we introduce the concept of an "oracle" to model the plethora of complicated techniques. We also devise a normalized scoring technique, which allows our metric to be extended to a variety of difficulty levels.

We devise a generation algorithm to produce puzzles with unique solutions that span all difficulty levels, as measured by our metric. To ensure uniqueness, our generation algorithm must solve the puzzle (multiple times) to check for extra solutions. Since solving a Sudoku puzzle is an NP-complete problem [Wikipedia 2008b], our algorithm has exponential running time at best.

Terminology

- A **completed Sudoku board** is a 9×9 grid filled with $\{1,\ldots,9\}$ such that every row, column and 3×3 subgrid contains each number exactly once.
- A **Sudoku puzzle** or **Sudoku board** is a completed Sudoku board from

which some of the cell contents have been erased.

- A **cell** is one of the 81 squares of a 9×9 grid.
- The nine 3×3 subgrids that appear by dividing the board into thirds are called **blocks**.
- A **house** refers to any row, column or block of a 9×9 grid.
- A **hint** is a cell that has already been filled in a Sudoku puzzle.
- A **candidate** is a number that is allowed to go in a given cell. Initially, any empty cell has the candidate set $\{1, \ldots, 9\}$. Candidates can be eliminated when a number can already be found in a house containing the cell and by more complicated techniques.
- **Singles** is a solving technique in which a cell is determined by one of two basic methods:

  Naked Singles: If a cell has only one remaining candidate, then that cell can be filled with that candidate.

  Hidden Singles: If there is only one cell in a given house that has a certain candidate, then that cell can be filled with that candidate.

Assumptions

- *Every Sudoku puzzle has a unique solution.*
- *There are no restrictions on the locations of the hints in a Sudoku puzzle.* When the Japanese puzzle company Nikoli adapted the puzzle in 1986, it added the constraint that clues should be arranged symmetrically. We do not consider this esthetic touch to be important to the structure of the puzzle and hence ignore this constraint.
- *The singles solving techniques are sufficiently basic that the typical player uses them.* The logic for these techniques derives directly from the definition of the game.
- *The naked singles technique is "easier" than the hidden singles technique.* When we look for hidden singles first and move to naked singles only when hidden singles no longer produces new information, we can solve a puzzle in many fewer steps. On the other hand, if we first look for naked singles and then move to hidden singles, we could oscillate between the methods repeatedly. We do not claim that all human solvers find naked singles easier than hidden singles. However, hidden singles appears to be more powerful and thus in some sense "harder."
- *The difficulty of a puzzle cannot be based on any specific set of techniques.* There are many different techniques beyond the singles, and we cannot assume

that a player will use any particular one. A list of such techniques with explanations can be found at Sudopedia [2008]. Different puzzles will succumb more easily to different techniques and will thus seem easier (or harder) to different people, depending on what approaches they tend to use.

- *The difficulty of a puzzle does not scale linearly with the number of applications of higher-level techniques* There is an obvious jump in difficulty when a puzzle requires more than just the singles techniques, since then a player must use strategies that cannot be read directly from the rules. On the other hand, having to use the same or a similar higher technique repeatedly does not require any extra leap of logic.

Sudoku Difficulty Metric

Objectives

Our first task is to develop a metric, or scoring system, to determine the difficulty of an arbitrary Sudoku grid. However, the starting configuration of a puzzle is often quite deceptive about the level of difficulty; so we must analyze the difficulty by looking at both the starting configuration and the completed board.

Additionally, we want our metric to be extensible to varied difficulty levels and player abilities. That is, we would like those who disagree with our metric to be able to adjust it and produce a metric that they agree with. Finally, our metric should be representative of the *perceived* difficulty of a puzzle by a human solver, regardless of how "simple" it is for a computer to solve.

A Trip to the Oracle

We assume that a typical player starts solving a Sudoku puzzle begin by filling in cells that can be determined by the singles techniques. When the player can determine no more cells via those techniques, the player will begin to employ one or more higher-level methods and combine the new information with the singles techniques until solving the puzzle or getting stuck again.

We can exploit this observation to develop a metric that rates the difficulty of a puzzle simply by determining the number of different methods used to solve it. In particular, the more complicated the methods, the more challenging the puzzle is. However, due to the complicated nature of the more than 50 solving techniques [Sudopedia 2008], it is hard to say which are "more challenging" than others. Additionally, many humans approach a puzzle differently, applying different techniques at different stages of the

puzzle. To avoid becoming bogged down in a zoo of Fish and X-Wings, we introduce the concept of an oracle.

The *oracle* is a being that knows the solution to all puzzles and can communicate a solution to a player, as long as the player knows how to ask properly. We can think of the oracle as if it were another player who uses a more-advanced solving technique. When a player gets stuck, the player goes to the oracle for help, and the oracle reveals the value of a cell in the grid. The usefulness of the revealed cell depends on the manner in which the player phrases the question.

In slightly less mystical terms, we use an oracle to represent the fact that the player uses higher-level techniques to solve a puzzle. Doing so allows us to model the difficulty of a puzzle without knowing anything about specific difficulty levels of solving methods. The oracle can be any method beyond singles that the player uses to fill in additional cells, and we represent this in our metric by randomly filling in some cell in the matrix. The perceived difficultly level of a puzzle increases as more higher-level techniques are used, but this increase is not linear in the number of techniques.

A Sudoku Difficulty Metric

Based on our above assumptions about how a human being approaches a Sudoku puzzle, we developed **Algorithm 1** to rate a puzzle's difficulty.

Algorithm 1 SUDOKU METRIC

```
procedure SCORE(InitialGrid, Solution)
    for All trials do
        Board = InitialGrid
        while Board is unsolved do
            Find all singles                          ▷ Iterative process
            if stuck then
                Ask ORACLE for help
            end if
        end while
        Count singles and ORACLE visits
    end for
    Compute average counts
    SCOREFUNCTION(singlesCounts, oracleCounts)
    return score                                      ▷ Use tanh to scale
end procedure
```

First, we search for naked singles until there are no more to be filled in. Then, we perform one pass looking for hidden singles. We repeat this process until the board is solved or until we can get no further using singles. (The order in which we consider the singles techniques accords with our

assumption that naked singles are "easier" than hidden singles). When we can get no further, we consult the oracle. The algorithm keeps track of the the number of iterations, naked singles, hidden singles, and oracle visits, and presents this information to a scoring function, which combines the values and scales them to between 0 and 1, the "normalized" difficulty of the puzzle. The details of the scoring function are discussed below.

The above description does not take into consideration that because we are using a random device to reveal information, separate runs may produce different difficulty values and hence the revealed cell may provide either more or less aid. To smooth out the impacts of this factor, we run the test many times and average the scores of the trials.

Scoring with tanh

A normalized score allows one puzzle to be compared with another. While on average we do not expect the unscaled score to be large, the number of oracle visits could be very high, thus sometimes producing large variability in the weighted sum of naked single, hidden single, and oracle visit counts. Simple scaling by an appropriate factor would give undue influence to outlier trials. Consequently, we pass the weighted sum through a sigmoid function that weights outliers on both the high and low ends similarly and gives the desired range of variability in the region that we expect most boards to fall into. We use a tanh function to accomplish this.

We would also like to model our assumption that each successive oracle visit is likely to provide less information than the previous one. We do this by passing the number of oracle visits through an inverse exponential function before scaling. Since we run a large number of trials to compute each score, we use the average number of oracle visits over all the trials in this exponential function, as doing so makes our scores fluctuate much less than if we average after applying the inverse exponential.

We arrive at the following equation for the unscaled score:

$$s = \alpha N + \beta H + \gamma \left(1 - e^{\delta \cdot (\overline{O} - \sigma)}\right), \tag{1}$$

where

- the Greek letters are user-tunable parameters (we discuss their significance shortly),
- N and H are the average number of naked singles and hidden singles found per scan through the board, and
- $\overline{O}$ is the average number of oracle visits per trial.

Note that N and H are averaged over a single trial, and together represent how many singles you can expect to find at a given stage in solving the

puzzle; $\overline{O}$ is averaged over all trials and represents how many times you can expect to use higher-order techniques to solve the puzzle.

Finally, we pass this unscaled score through an appropriately shifted sigmoid:

$$\text{ScaledScore} = \frac{1 + \tanh\big[A(s - B)\big]}{2},$$

where s is the unscaled score from **(1)**, and A and B are user-tunable parameters. We shift the function up by 1 and scale by $1/2$ to produce a range of values between 0 and 1 (**Figure 1**). We also smear the function out over a wide area to capture the differences in unscaled scores.

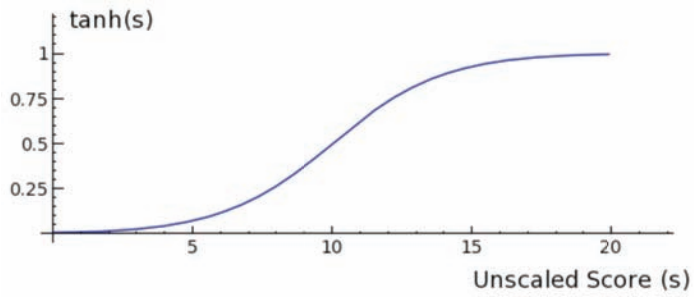

Figure 1. The scaled hyperbolic tangent that produces our final score. We shift the function up by 1 and scale by $1/2$ to produce a range of values between 0 and 1. We also smear the function out over a wide area to capture the differences in unscaled scores.

A Zoo of Parameters

Our parameters can be divided into two groups:

- those that represent some intrinsic notion of how challenging a Sudoku board is
  - α: Represents the difficulty of finding naked singles in a puzzle. It allows us to scale the observed number of naked singles in the puzzle based on how challenging we think they are to find.
  - β: Weights the difficulty of finding hidden singles in a board. To agree with our earlier assumptions, we assume that $\alpha < \beta$.
  - γ: Gives the weighting function for the number of oracle visits. This parameter will in general be quite high, as we believe that oracle visits should be the primary determination of difficulty level. In actuality, γ is a function of $\overline{O}$, since we don't want the exponential function in (1) to contribute negatively to the score. Thus, we have that

$$\gamma = \begin{cases} 0, & \text{for } \overline{O} < 1; \\ G, & \text{otherwise,} \end{cases}$$

    for some large constant G.

- those that allow us to scale and shift a graph around:
  - δ: Controls the steepness of the exponential function.
  - σ: Controls the x-intercept of the exponential function.
  - A: Controls the spread of the $\tanh$ function.
  - B: Controls the shift of the $\tanh$ function.

The shift parameters are designed to allow for greater differentiation between puzzle difficulties, not to represent how difficult a puzzle actually is. Therefore, we believe that the first three parameters are the important ones. That is, those should be adjusted to reflect puzzle difficulty, and the last four should be set to whatever values allow for maximum differentiation among difficulty levels for a given set of puzzles. We discuss our choice of parameter values later. First, we turn to the problem of board generation.

Generation Algorithm

Objectives

It is natural to require that a Sudoku generator:

- always generate a puzzle with a unique solution (in keeping with our assumptions about valid Sudoku boards), and
- can generate any possible completed Sudoku board. (As it turns out, our algorithm does not actually generate all possible completed boards, but it should be able to if we expand the search space slightly.)

We also would like our generator be able to create boards across the spectrum of difficulty defined by our metric; but we do not demand that it be able to create a puzzle of a *specified* difficulty level, since small changes in a Sudoku board can have wide-reaching effects on its difficulty. However, our generator can be turned into an "on demand" generator by repeatedly generating boards until one of the desired difficulty level is produced.

Uniqueness and Complexity

It is easy to generate a Sudoku puzzle: Start with a completely filled-in grid, then remove numbers until you don't feel like removing any more. This method is quite fast but does not guarantee uniqueness. What is worse, the number of cells that you have to erase before multiple solutions can occur is alarmingly low! For example, if all of the 6s and 7s are removed from a Sudoku puzzle, there are now two possible solutions: the original solution, and one in which all positions of 6 and 7 are reversed. In fact, there is an even worse configuration known as a "deadly rectangle" [Sudopedia 2008] that can result in non-unique solutions if the wrong four cells of some

Sudoku boards are emptied. If we cannot guarantee uniqueness of solutions when only four numbers have been removed, how can we possibly guarantee it when more than 50 have?

The natural solution to this problem is to check the board for uniqueness after each cell is removed. If a removal causes the board to have a non-unique solution, replace it and try again. Unfortunately, there is no known fast algorithm for determining if the board has a unique solution. The only way is to enumerate all possible solutions, and this requires exponential time in the size of the board [Wikipedia 2008b].

The good news is that nevertheless there are fast algorithms, including Donald Knuth's Dancing Links algorithm [2000]. Dancing Links, also know as DLX, is an optimized brute-force enumeration algorithm for a problem known as *Exact Cover*. Exact Cover is an NP-complete problem dealing with membership in a certain collection of sets. By formulating the constraints on a Sudoku grid as sets, we can turn a Sudoku problem into an Exact Cover problem. While DLX is still an exponential algorithm, it outperforms most other such algorithms for similar problems. For our purposes, it is more than sufficient, since it solved the most challenging Sudoku problems we could find in 0.025 second.

DLX affords us an algorithm to generate puzzles with unique solutions: Simply remove cells from the completed board until no more cells can be removed while maintaining a unique solution, and use DLX at every stage to guarantee that the solution is still unique.

We now return to the issue of creating a completed Sudoku grid, since this is the one unfinished point in the algorithm. It again turns out to be quite difficult to generate a completed Sudoku grid, since doing so is akin to solving the puzzle. In theory, we could start with an empty grid and apply DLX to enumerate every possible solution—but there are 6.671×10^{21} completed boards.

Alternatively, many Websites suggest the following approach:

- Start with a completed Sudoku board.
- Permute rows, columns, and blocks (or other such operations that maintain the validity of the board).
- Output new Sudoku board.

This approach has two significant flaws:

- It assumes that we already have a valid Sudoku board (which is the very problem we're trying to solve); and
- it drastically limits the space of possible generated boards, since any single starting board can generate through these permutations only $3,359,232$ of the 6.617×10^{21} possible Sudoku boards [Wikipedia 2008a].

Thus, to perform our initial grid generation, we employ a combination of the two techniques that is quite fast and does not overly limit the size of

the search space. We generate three random permutations of $1, \ldots, 9$ and fill these in along the diagonal blocks of an empty grid. We then seed the DLX algorithm with this board and ask it to find the first N solutions to the board, for N a large number (in our case, 100). Finally, we randomly select one of these boards to be our final board.

In principle, if N is sufficiently large, this method can generate any valid Sudoku board, since the seed to DLX is random, even if DLX itself is deterministic. With $(9!)^3$ seeds and 100 boards per seed, we can generate 4.778×10^{18} Sudoku boards, assuming that each seed has at least 100 solutions.

Use of the DLX algorithm makes this method fast enough for our purposes; we took advantage of Python code written by Antti Anjanki Ajanki [2006] that applies DLX to Sudoku puzzles. **Algorithm 2** gives the pseudocode for our generation algorithm.

Algorithm 2 SUDOKU GENERATOR

```
1:  procedure GENERATEBOARD
2:      for i = 1, ..., 3 do                          ▷ Seed DLX
3:          PERMUTE {1, ..., 9}
4:          Fill diagonal block i
5:      end for
6:      DLX(seed, numToGenerate)
7:      Select randomly generated board
8:      repeat
9:          Remove random cells
10:         Check uniqueness (DLX)
11:     until No more cells can be removed
12: end procedure
```

Results and Analysis

To test both the utility of our metric and the effectiveness of our generation algorithm, we generated and scored 1,000 boards, and as a baseline also scored some independently-generated grids from the Internet. Our results match up well with "accepted" levels of difficulty, though there are some exceptions (**Table 1**). Our generator is biased towards generating easy puzzles but can generate puzzles with quite high difficulty; we believe that this performance is a consequence of the fact that difficult Sudoku puzzles are hard to create.

According to our metric, the most important factor in the difficulty of a puzzle is the number of oracle visits. The easiest puzzles are ones that can be solved entirely by singles techniques and thus do not visit the oracle at

Table 1.
Results from our Sudoku generator.

Difficulty	Number generated	Average score	Average # oracle visits
Cinch	740	0.06	0.1
Confusing	273	0.32	0.6
Challenging	53	0.69	1.5
Crazy	34	0.88	2.3

all. In general, this type of puzzles has a score of less than 0.18, which we use as our first dividing region.

The next level of difficulty is produced by puzzles that visit the oracle once on average; these puzzles produce a scaled score of between 0.18 and 0.6, so we use this as our second dividing region. Puzzles that require two or three hints from the oracle are scored between 0.6 and 0.8, and the absolute hardest puzzles have scores ranging from 0.8 to 1.0, due to needing three or more oracle visits. We show the output function of our metric, and mark the four difficulty regions, in **Figure 2**.

A significant number of generated boards have very high scores but no oracle visits, perhaps because of a large number of hidden singles.

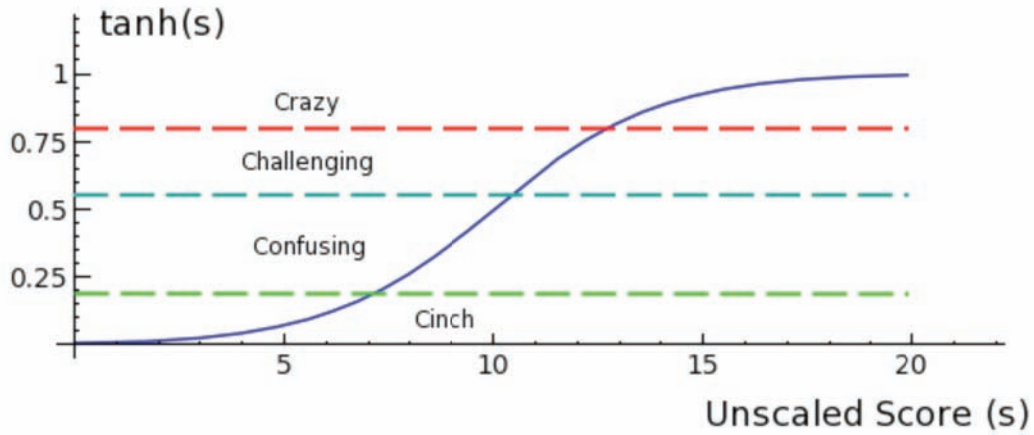

Figure 2. Our four difficulty regions plotted against the sigmoid scoring function.

For your solving pleasure, in **Figure 3** we included a bonus: four sample boards generated by our algorithm, ranging in difficulty level from "Cinch" (solvable by singles alone) to "Crazy" (requiring many advanced techniques). Try to solve them and see if you agree!

Our algorithm can generate puzzles across the spectrum but fewer of more difficult boards. This behavior appears to reflect the fact that Sudoku is very sensitive to seemingly minor modifications: Small changes to the layout of the hints or of the board can lead to vast changes in difficulty. In fact, a number of independently-generated boards that were rated by others as quite difficult turn out to be solvable by singles techniques alone,

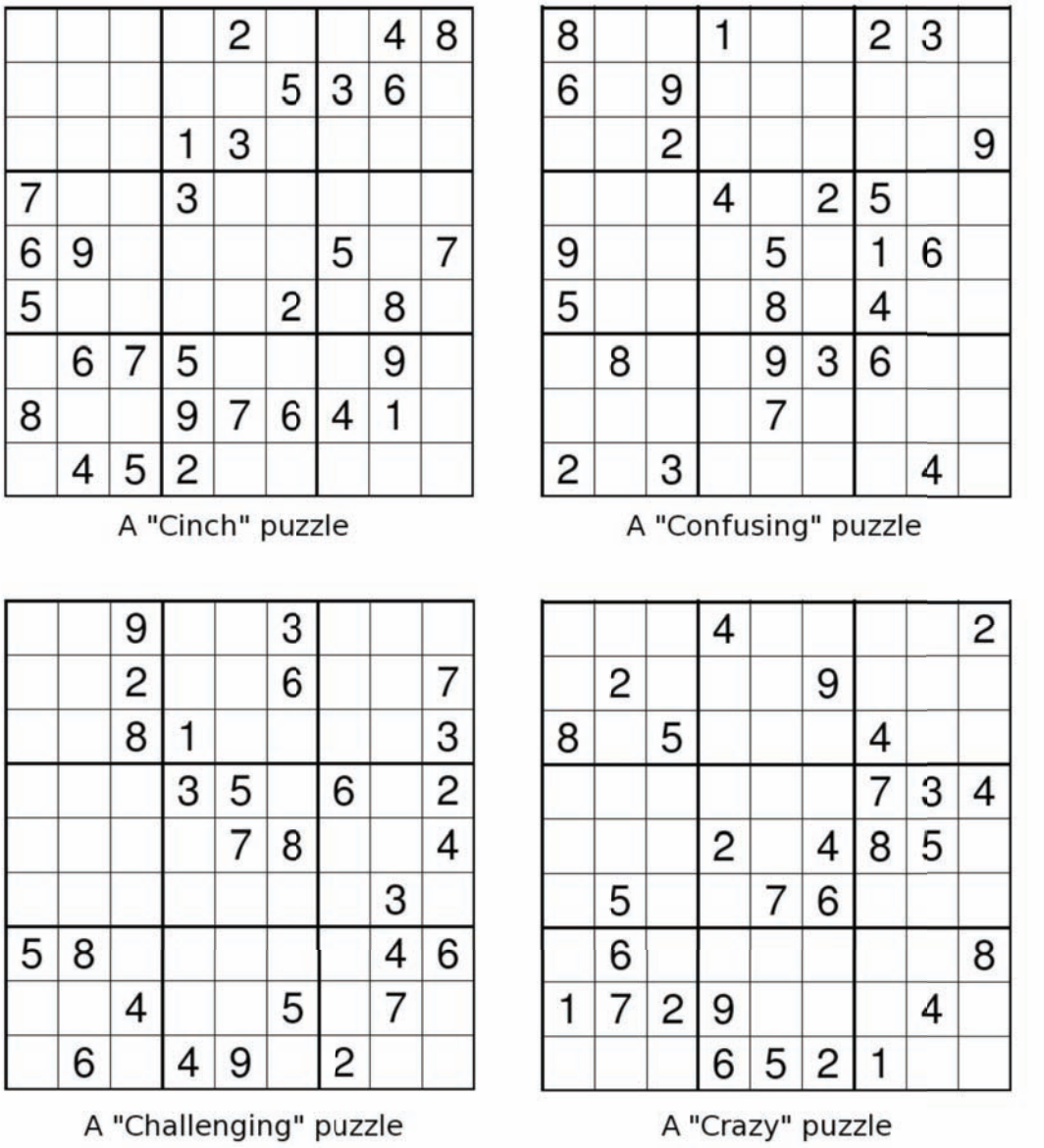

Figure 3. Four puzzles created by our generator, increasing in difficulty from left to right and top to bottom.

leading us to believe that creating hard Sudoku puzzles is hard.

We found a large number of very challenging Sudoku puzzles at Websites, which we tested with our metric. In particular, a Website called SudoCue [Werf 2007] offers some of the most challenging boards that we could find. Its "Daily Nightmare" section scored above 0.8 in almost all of our tests, with many puzzles above 0.9.

Our metric is highly sensitive to its parameter values. This is good, since it allows different users to tweak the metric to reflect their individual difficulty levels; but it is bad, because two different parameter sets can lead to vastly different difficulty scores. In **Table 2**, we show the parameter

values that we use; they were empirically generated. We maintain that they are informative but acknowledge that they could be changed to yield different outcomes.

Table 2.
Parameter values chosen for our metric. The first three represent the difficulty of the puzzle, and the last four are scaling and shift parameters.

Parameter	α	β	γ	δ	σ	A	B
Value	0.1	0.5	15	0.5	1	0.25	10

In analyzing our metric, we found some puzzles that are scored reasonably and some that are scored outrageously. Our generator gave us a puzzle rated .85 that had no oracle visits. Suspicious of our metric, one member of our team tried solving the puzzle. He did so in less than 7.5 min, even though a photographer interrupted him to take pictures of the team. This discovery caused us to realize that perhaps our parameters do not mean what we thought they did. Our intent in weighting naked singles positively was to make our metric say that nearly-completely-filled-in puzzles are easier than sparsely-filled-in puzzles. We considered the average number of singles per scan rather than the total number because we wanted to somehow include the number of iterations in our metric. However, with our current parameter values, our metric says that a puzzle with a high number of singles per scan is easier than a puzzle with a low number of singles per scan, which is wrong and not what we intended. Parameters α and β should probably be negative, or else we should change what N and H mean.

Future Work

Any generator that has to solve the puzzle to check for the uniqueness of the solution will inherently have an exponential running time (assuming P$\neq$NP), since solving Sudoku has been shown to be NP-complete. Thus, to produce a generator with a better runtime, it would be necessary to find some other means of checking the uniqueness of the solution. One possible approach would be to analyze the configurations that occur when a puzzle does not have a unique solution. Checking for such configurations could result in a more efficient method of checking for solution uniqueness and thus would potentially allow for generators with less than exponential running time.

While we are excited by the potential that the oracle brings in rating Sudoku puzzles, we recognize that our metric as it stands is not as effective as it could be. We have a variety of ideas as to how it could be improved:

- The simplest improvement would be to run more experiments to deter-

mine better parameter values for our scoring function and the best places to insert difficulty breaks. A slightly different scoring function, perhaps one that directly considers the number of iterations needed to solve the Sudoku puzzle, could be a more accurate measure.

- Alternatively, we would like to devise a scoring function that distinguishes puzzles that can be solved using only the singles techniques vs. others.
- There is a much larger set of techniques that Sudopedia refers to as the Simple Sudoku Technique Set (SSTS) that advanced solvers consider trivial. If we were to add another layer to our model, so that we first did as much as possible with singles, then applied the rest of the SSTS and only went to the oracle when those techniques had been exhausted, it could give a better delineation of "medium-level" puzzles. If there were a threshold score dividing puzzles solvable by SSTS and puzzles requiring the oracle, it would allow advanced solvers to determine which puzzles they would find interesting.
- Another extension would be altering the oracle to eliminate possible cell candidate(s) rather than reveal a new cell. This alteration could potentially allow for greater differentiation among the hardest puzzles.

Conclusion

We devised a metric which uses an oracle to model techniques employed by the Sudoku community. This approach has the advantage of not depending on a specific set of techniques or any particular hierarchy of them. The large number of parameters in our scoring function leaves it open to adjustment and improvement.

In addition, we developed an algorithm to generate puzzles with unique solutions across a wide range of difficulties. It tends towards creating easier puzzles.

There is an increasing demand for more Sudoku puzzles, different puzzles, and harder puzzles. We hope that we have contributed insights into its levels of difficulty.

References

Ajanki, Antti. 2006. Dancing links Sudoku solver. Source code available online at `http://users.tkk.fi/~aajanki/sudoku/index.html`.

Eppstein, David. 2005. Nonrepetitive paths and cycles in graphs with application to sudoku. `http://arxiv.org/abs/cs/0507053v1`.

Felgenhauer, Frazer, and B. Jarvis. 2006. Mathematics of Sudoku I. http://www.afjarvis.staff.shef.ac.uk/sudoku/.

Knuth, Donald E. 2000. Dancing links. In *Millenial Perspectives in Computer Science*, 187–214. http://www-cs-faculty.stanford.edu/~uno/preprints.html.

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.

Solving technique. http://www.sudopedia.org. Accessed 16 Feb 2008, then last modified 11 Jan 2008.

van der Werf, Ruud. 2007. SudoCue—home of the Sudoku addict. http://www.sudocue.net.

Wikipedia. 2008a. Mathematics of Sudoku. http://en.wikipedia.org/wiki/Mathematics_of_Sudoku.

_____. 2008b. Sudoku. http://en.wikipedia.org/wiki/Sudoku.

Frederick Johnson, Sarah Fletcher, David Morrison, and advisor Jon Jacobsen.