Difficulty-Driven Sudoku Puzzle Generation

Martin Hunt
Christopher Pong
George Tucker

Harvey Mudd College
Claremont, CA

Advisor: Zach Dodds

Summary

Many existing Sudoku puzzle generators create puzzles randomly by starting with either a blank grid or a filled-in grid. To generate a puzzle of a desired difficulty level, puzzles are made, graded, and discarded until one meets the required difficulty level, as evaluated by a predetermined difficulty metric. The efficiency of this process relies on randomness to span all difficulty levels.

We describe generation and evaluation methods that accurately model human Sudoku-playing. Instead of a completely random puzzle generator, we propose a new algorithm, *Difficulty-Driven Generation*, that guides the generation process by adding cells to an empty grid that maintain the desired difficulty.

We encapsulate the most difficult technique required to solve the puzzle and number of available moves at any given time into a *rounds* metric. A round is a single stage in the puzzle-solving process, consisting of a single high-level move or a maximal series of low-level moves. Our metric counts the numbers of each type of rounds.

Implementing our generator algorithm requires using an existing metric, which assigns a puzzle a difficulty corresponding to the most difficult technique required to solve it. We propose using our rounds metric as a method to further simplify our generator.

The UMAP Journal 29 (3) (2008) 343–362.

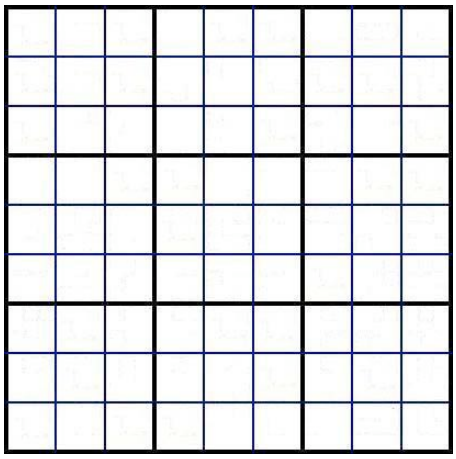


Figure 1. A blank Sudoku grid.

Introduction

Sudoku first appeared in 1979 in *Dell Pencil Puzzles & Word Games* under the name "Number Place" [Garns 1979]. In 1984, the puzzle migrated to Japan where the *Monthly Nikolist* began printing its own puzzles entitled "Suuji Wa Dokushin Ni Kagiru" or "the numbers must be single"—later shortened to Sudoku [Pegg Jr. 2005]. It was not until 2005, however, that this puzzle gained international fame.

We propose an algorithm to generate Sudoku puzzles at specified difficulty levels. This algorithm is based on a modified solver, which checks to ensure one solution to all generated puzzles and uses metrics to quantify the difficulty of the puzzle. Our algorithm differs from existing algorithms in using human-technique-based modeling to guide puzzle construction. In addition, we offer a new grading algorithm that measures both the most difficult moves and the number of available moves at each stage of the solving process. We ran basic tests on both algorithms to demonstrate their feasibility. Our work leads us to believe that combining our generation algorithm and metrics would result in a generation algorithm that creates puzzles on a scale of difficulty corresponding to actual perceived difficulty.

Terminology

Figure 1 shows a blank 9×9 cell Sudoku grid. In this grid, there are nine rows, nine columns, and nine blocks (3×3 disjoint cell groups defined by the thicker black lines). We use the terms:

- **Cell**: A single unit square in the Sudoku grid that can contain exactly one digit between 1 and 9 inclusive.
- **Adjacent Cell**: A cell that is in the same row, column or block as some other cell(s).

- **Hint**: One of the digits initially in a Sudoku puzzle.
- **Puzzle**: The 9×9 cell Sudoku grid with some cells containing digits (hints). The remaining cells are intentionally left empty.
- **Completed puzzle**: The 9×9 cell Sudoku grid with all cells containing digits.
- **Well-posed puzzle**: A Sudoku puzzle with exactly one solution (a unique completed puzzle). A Sudoku puzzle that is not well-posed either has no solutions or has multiple solutions (different completed puzzles can be obtained from the same initial puzzle).
- **Proper puzzle**: A Sudoku puzzle that can be solved using only logical moves—guessing and checking is not necessary. All proper puzzles are well-posed. Some newspaper or magazine problems are proper puzzles We concern ourselves with proper Sudoku puzzles only.
- **Candidate**: A number that can potentially be placed in a given cell. A cell typically has multiple candidates. A cell with only one candidate, for example, can simply be assigned the value of the candidate.

How to Play

The object of the game is to place the digits 1 through 9 in a given puzzle board such that every row, column and block contains each digit exactly once. An example puzzle with 28 hints is shown in **Figure 2a**. All of the digits can be placed using the logical techniques described subsequently. **Figure 2b** shows the completed solution to this problem. A well-posed Sudoku problem has only one solution.

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

a. Puzzle example.

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

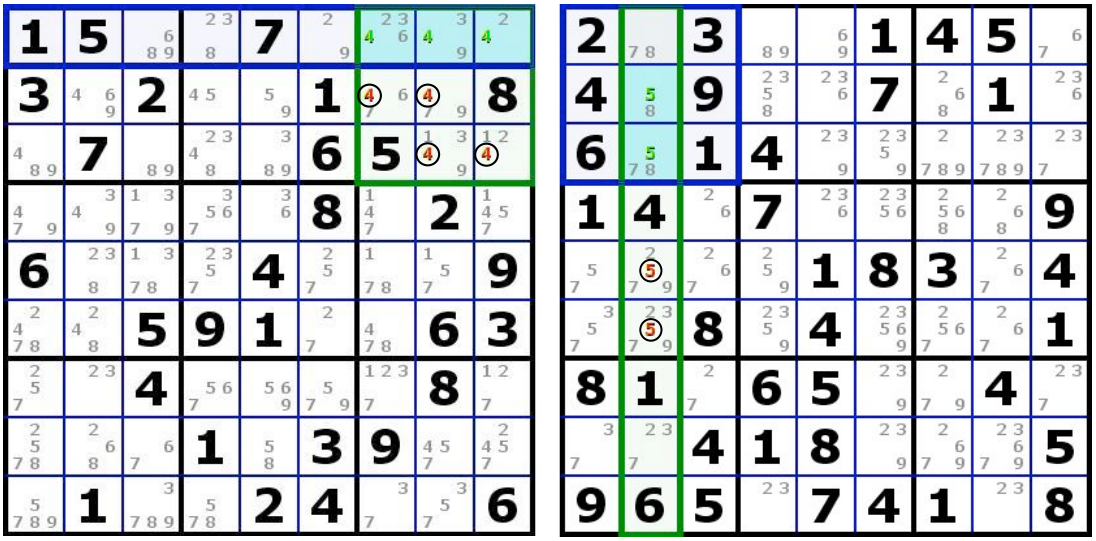
b. Puzzle solution.

Figure 2. Example of a Sudoku puzzle and its solution.

Techniques

We provide a list of commonly-used techniques; they require that the solver know the possible candidates for the relevant cells.

- **Naked Single**: A cell contains only one candidate, therefore it must be that number. This is called naked because there is only one candidate in this cell.
- **Hidden Single**: A cell contains multiple candidates but only one is possible given a row/column/block constraint, therefore it must be that number. This is called hidden because there are multiple candidates in the cell, but only one of them can be true due to the constraints.
- **Claiming**: This occurs when a candidate in a row/column also only appears in a single block. Since the row must have at least one of the candidate, these candidates "claim" the block. Therefore, all other instances of this candidate can be eliminated in the block. In **Figure 3a**, the circled 4s are the candidates that can be eliminated.
- **Pointing**: This occurs when a block has a candidate that appears only in a row/column. These candidates "point" to other candidates along the row/column that can be eliminated. In **Figure 3b**, the circled 5s are the candidates that can be eliminated.

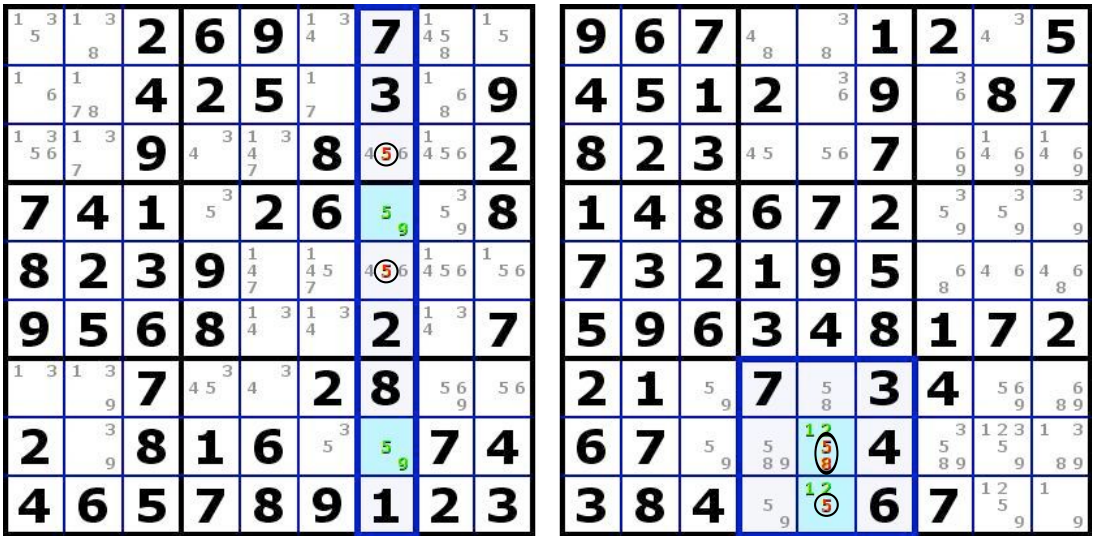


a. Claiming. **b.** Pointing.

Figure 3. Examples of pointing and claiming techniques.

- **Naked Double**: In a given row/column/block, there are two cells that have the same two and only two candidates. Therefore, these candidates can be eliminated in any other adjacent cells. In **Figure 4a**, the circled 5s are the candidates that can be eliminated.

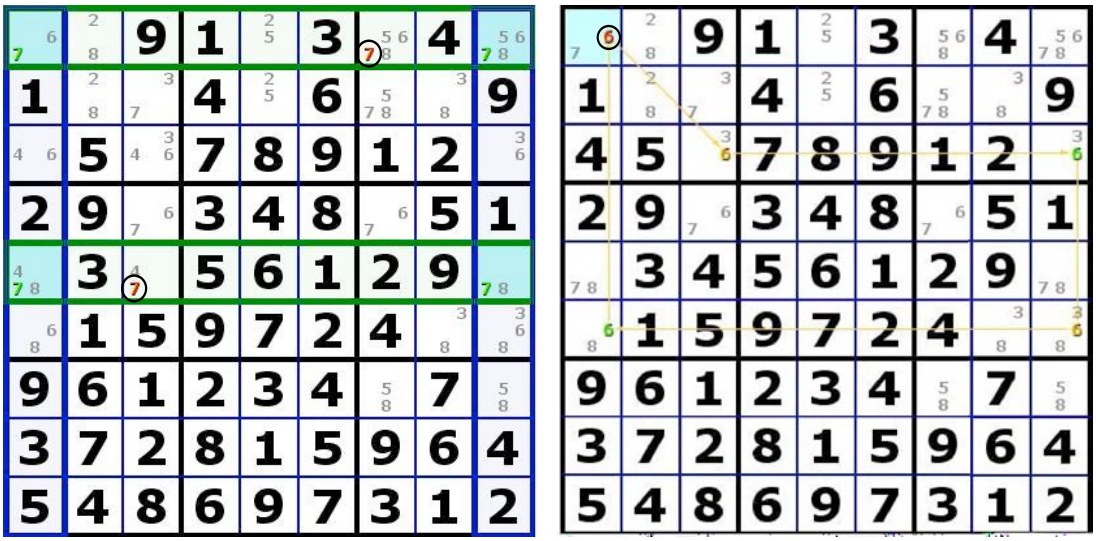
- **Hidden Double**: In a given row/column/block, two and only two cells can be one of two candidates due to the row/column/block constraint. Therefore any other candidates in these two cells can be eliminated. In **Figure 4b**, the circled 5s and 8 are the candidates that can be eliminated.



a. Naked Double. **b.** Hidden Double.

Figure 4. Examples of naked double and hidden double techniques.

- **Naked Subset (Triple, Quadruple, etc.)**: This is an extension of the naked double to higher numbers of candidates and cells, n. Because these n candidates must appear in the n cells, these candidates to be eliminated from adjacent cells.
- **Hidden Subset (Triple, Quadruple, etc.)**: This is an extension of the hidden double to higher numbers of candidates and cells, n. Due to row/column/block constraints, the n candidates must occupy n cells and other candidates in these cells can be eliminated.
- **X-Wing**: The X-Wing pattern focuses on the intersection of two rows and two columns. If two rows contain contain a single candidate digit in exactly two columns, then we can eliminate the candidate digit from all of the cells in those columns. The four cells of interest form a rectangle and the candidate digit must occupy cells on alternate corners, hence the name X-Wing. Note that the rows and columns can be interchanged. In **Figure 5a**, the circled 7s can be eliminated. X-Wings naturally lead to extensions such as XY-Wings, XYZ-Wings, Swordfish, Jellyfish, and Squirmbags. (Other advanced techniques are discussed at Stuart [2008].)
- **Nishio**: A candidate is guessed to be correct. If this leads to a contradiction, that candidate can be eliminated. In **Figure 5b**, the circled 6 can be eliminated.



a. X-Wing. **b.** Nishio.

Figure 5. Examples of X-Wing and Nishio techniques.

Generators

Solving Sudoku has been well-investigated. but generating puzzles is without much exploration. There are 6,670,903,752,021,072,936,960 possible completed puzzles [Felgenhauer and Jarvis 2006]. There are even more possible puzzles, since there are many ways of removing cells from completed puzzles to produce initial puzzles, though this number cannot be easily defined since the uniqueness constraint of well-formed puzzles makes them difficult to count.

We studied several generators, including Sudoku Explainer, jlib, Isanaki, PsuedoQ, SuDoKu, SudoCue, Microsoft Sudoku, and Gnome Sudoku. All rely on Sudoku solvers to verify that the generated puzzles are unique and sufficiently difficult. The two fastest generators work by randomly generating puzzles with brute-force solvers.

The most powerful solver that we encountered, Sudoku Explainer [Juillerat 2007], uses a brute-force solver to generate a completed puzzle. After producing the completed puzzle, it randomly removes cells and uses the same solver to verify that the puzzle with randomly removed cells still contains a unique solution. Finally, it uses a technique-based solver to evaluate the difficulty from a human perspective. Due to the variety of data structures and techniques already implemented in Sudoku Explainer, we use it as a framework for our code. Since generators rely heavily on solvers, we begin with a discussion of existing solver methods.

Modeling Human Sudoku Solving

Humans and the fastest computer algorithms solve Sudoku in very different ways. For example, the Dancing Links (DLX) Algorithm Knuth [2000] solves Sudoku by using a depth-first brute-force search in mere milliseconds. Backtracking and efficient data structures (2D linked lists) allow this algorithm to operate extremely quickly relative to other methods, although the Sudoku problem for a general $N \times N$ grid is NP-complete.

Because DLX is a brute-force solver, it can also find multiple solutions to Sudoku puzzles that are not well-posed (so from a computer's perspective, solving any Sudoku problem is a simple task!). However, most Sudoku players would not want to solve a puzzle in this manner because it requires extensive guessing and back-tracking. Instead, a player uses a repertoire of techniques gained from experience.

To determine the difficulty of a Sudoku puzzle from the perspective of a human by using a computer solver, we must model human behavior.

Human Behavior

We first consider Sudoku as a formal problem in constraint satisfaction [Simonis 2005], where we must satisfy the following four constraints:

1. **Cell constraint**: There can be only one number per cell.
2. **Row constraint**: There can be only one of each number per row.
3. **Column constraint**: There can be only one of each number per column.
4. **Block constraint**: There can be only one of each number per block.

We can realize these constraints in the form of a binary matrix. The rows represent the candidate digits for a cell; the columns represent constraints. In a blank Sudoku grid, there are 729 rows (nine candidates in each of the 81 cells) and 324 columns (81 constraints for each of the four constraints). A 1 is placed in the constraint matrix wherever a candidate satisfies a constraint. For example, the possibility of a 1 in the $(1, 1)$ entry of the Sudoku grid is represented as a row in the constraint matrix. The row has a 1 in the column corresponding to the constraint that the first box contains a 1. Therefore, each row contains exactly four 1s, since each possibility satisfies four constraints; and each column contains exactly nine 1s, since each constraint can be satisfied by nine candidates. An example of an abbreviated constraint matrix is in **Table 1**.

A solution is a subset of the rows such that each column has a single 1 in exactly one row of this subset. These rows correspond to a selection of digits such that every constraint is satisfied by exactly one digit, thus to a solution to the puzzle. The problem of finding in a general binary matrix a

Table 1.
Example of an abbreviated constraint matrix.

	One # in (2,3)	One # in (2,4)	#1 in Row 2	#2 in Row 2	#2 in Col. 4
#1@ (2,3)	1	.	1	.	.
#2@ (2,3)	1	.	.	1	.
#3@ (2,3)	1	.	.	.	.
#1@ (2,4)	.	1	1	.	.
#2@ (2,4)	.	1	.	1	1
#3@ (2,4)	.	1	.	.	.

subset of rows that sum along the columns to a row of 1s is known as the *exact-cover problem*.

The normal solution procedure for an exact-cover problem is to use DLX. The constraint-matrix form allows us to spot easily hidden or naked singles by simply looking for columns with a single 1 in them, which is already a step of the DLX algorithm. Moreover, the formalities of the constraint matrix allow us to state a general technique.

Theorem [Constraint Subset Rule]. *Suppose that we can form a set of n constraints A such that every candidate that satisfies a constraint of A only satisfies a single constraint of A and a set of n constraints B such that each candidate of A also satisfies a constraint in B. Then we can eliminate any candidate of B that is not a candidate of A.*

Proof: The assumption that every candidate that satisfies a constraint of A only satisfies a single constraint of A ensures that we must choose n candidates to satisfy the constraints in A (or else we could not possibly satisfy the n constraints of A). Each of those candidates satisfies at least one constraint in B, by assumption. Suppose for the sake of contradiction that we select a candidate that satisfies a constraint in B that does not satisfy any constraints A. Then B will have at most $n-1$ remaining constraints to be satisfied and the rows chosen to satisfy A will oversatisfy B (recall that each constraint must be satisfied once and only once). This is a contradiction, proving the claim. □

With specific constraints, the theorem translates naturally into many of the techniques discussed earlier. For example, if we a choose a singe block constraint as set A and a single row constraint as set B, then the Constraint Subset Rule reduces to the pointing technique.

Human Model

Using the constraint rule, we model the behavior of a human solver as:

1. Search for and use hidden and naked singles.

2. Search for constraint subsets of size one.
3. If a constraint subset is found, return to step 1. If none are found, return to step 2 and search for constraint subsets of one size larger.

Assumptions

- **Human solvers do not guess or use trial and error.** In certain puzzles, advanced Sudoku players use limited forms of trial and error, but extensive guessing trivializes the puzzle. So we consider only proper puzzles, which do not require guessing to solve.
- **Human solvers find all of the singles before moving on to more advanced techniques.** Sometimes players use more-advanced techniques while looking for singles; but for the most part, players look for the easiest moves before trying more-advanced techniques.
- **Human solvers search for subsets in order of increasing size.** The constraint subset rule treats all constraints equally, which allows us to generalize the advanced techniques easily. In practice, however, not all constraints are equal. Because the cells in a block are grouped together, for example, it is easier to focus on a block constraint than a row or column constraint. Future work should take this distinction into account.

Strengths

- **Simplicity.** By generalizing Sudoku to an exact cover problem, we avoid having to refer to specific named techniques and instead can use a single rule to govern our model.
- The model fairly accurately approximates how someone would go about solving a puzzle.

Weaknesses

- The constraint subset rule does not encompass all of the techniques that a human solver would employ; some advanced techniques, such as Nishio, do not fall under the subset rule. However, the constraint subset rule is a good approximation of the rules that human solvers use. Future work would incorporate additional rules to govern the model's behavior.

Using this model, we proceed to define metrics to assess the difficulty of puzzles.

Building the Metrics

One might assume that the more initial hints there are in a puzzle, the easier the puzzle is. Many papers follow this assumption, such as Tadei and Mancini [2006]. Lee [2006] uses a difficulty level based on a weighted sum of the techniques required to solve a puzzle, showing from a sample of 10,000 Sudoku puzzles that there is a correlation between the number of initially empty cells and the difficulty level.

There are many exceptions to both of these metrics that make them impractical to use. For example, puzzle A may only start with 22 hints but can be solved entirely with naked and hidden singles. Puzzle B may start with 50 hints but require the use of an advanced technique such as X-Wing to complete the puzzle. Puzzle C could start with 40 hints and after filling in 10 hidden singles be equivalent to Puzzle B (requiring an X-Wing). In practice, most people would find puzzles C and B equally difficult, while puzzle A would be significantly easier ($A < B = C$). The number of initial hints metric would classify the difficulties as $A > B > C$. The weighted-sum metric would do better, classifying $A < B < C$. Examples such as these show that counting the number of initial hints or weighting required techniques does not always accurately measure the difficulty of a puzzle.

We restrict our attention to evaluating the process involved in solving the puzzle. Our difficulty metric measures the following aspects:

- **Types of techniques required to solve the puzzle**: A more experienced Sudoku player will use techniques that require observing interactions between many cells.
- **The number of difficult techniques**: A puzzle with two X-Wings should be harder than a puzzle with one X-Wing.
- **The number of available moves**: It is easier to make progress in the puzzle when there are multiple options available, as opposed to a puzzle with only one logical move available.

To rate the difficulty of puzzles, Sudoku Explainer assigns each technique a numerical difficulty; the difficulty of the puzzle is defined to be the most difficult technique [Juillerat 2007]. The first problem with this type of implementation is that every technique must be documented and rated. The second problem is that many "medium" and even "hard" puzzles can be solved by finding singles, which have a low difficulty rating. The difference is that in difficult puzzles, the number of options at any given point is small. Sudoku Explainer's method of assigning difficulty has low granularity in the easy-to-hard category, which is the most important range of difficulties for most Sudoku players. The advanced techniques are not known by the general public and are difficult to use without a computer (one must write out all candidates for each cell).

We want our difficulty metric to distinguish between puzzles even when

the puzzle requires only basic techniques. One problem is that different orders of eliminating singles may increase or decrease the number of options available. So we measure difficulty by defining a *round*.

> A **round** is performing either every possible single move (hidden or naked) that we can see from our current board state *or* exactly one higher-level move.

Our metric operates by performing rounds until it either solves the puzzle or cannot proceed further. We classify puzzles by the number of rounds, whether they use higher-level constraint sets, or if they cannot be solved using these techniques.

Difficulty Levels

Our difficulty levels are:

1. **Easy**: Can be solved using hidden and naked singles.
2. **Medium**: Can be solved using constraint subsets of size one and hidden and naked singles.
3. **Hard**: Can be solved using constraint subsets of size two and easier techniques.
4. **Fiendish**: Cannot be solved using constraint subsets of size two or easier techniques.

Within each category, the number of rounds can be used to rank puzzles.

Strengths

- **Accounts for the number of available options.**
- **Provides finer granularity at the easier levels.** Within each category, the number of rounds is way to quantify which puzzles are harder.
- **Generalizes many of the named techniques** by expressing them in terms of eliminations in the exact cover matrix.
- **Conceptually very simple.** By formulating the human solver's behavior in terms of the constraint matrix, we avoid having to use and rate specific techniques. The Constraint Subset Rule naturally scales in difficulty as the subset size increases.

Weaknesses

- **Does not take into account the differences among row/column/block constraints.** It is easier to find a naked single in a block than in a row, because the cells are grouped together.

- **Computationally slower.** Our metric searches for more basic types of moves than the Sudoku Explainer does. The benefits of expressing the rules in terms of the exact cover matrix far outweigh the decrease in speed.
- **Lack of regard for advanced techniques.** This metric does not consider many of the advanced techniques. Though most Sudoku players do not use these techniques, our basic implementation cannot classify puzzles with comparable granularity in the Hard and Fiendish categories. However, expanding the metric to include additional techniques is relatively simple using the constraint matrix formulation.

Results

Our metrics were tested with 34 Sudoku puzzles that appeared in the Los Angeles *Times* from 15 January 2008 to 17 February 2008. The results are tabulated in **Table 2**. We see that the terms "Gentle" and "Moderate" puzzles would both fall under our "Easy" category. These two difficulty levels would be distinguished within our "Easy" level because we can see that on average "Gentle" puzzles have fewer rounds than "Moderate" puzzles. However, there is some overlap. Perhaps our algorithm can distinguish between these categories better than the algorithm used by the *Times*, or the *Times*'s algorithm accounts for the fact that some singles may be easier to find than others. For example, a hidden single in a block might be easier to spot than a hidden single in a row or column; our algorithm treats all of these equally.

The *Times*'s Sudoku puzzles skip over our defined difficulty level of "Medium"—there are no puzzles with constraint subsets of size one. According to our metrics, the jump from "Moderate" to "Tough" is much larger than is justified. For example, there should be a level of difficulty between puzzles that use only naked and hidden singles and puzzles that require an X-Wing, consisting of puzzles that use pointing and claiming. In effect, our algorithm defines a higher granularity than that used by the Los Angeles *Times*.

Two "Tough" puzzles could not be solved using our Constraint Subset Rule. In fact, when checked with Sudoku Explainer, these puzzles require the use of Nishio and other advanced techniques that border on trial-and-error. Our metric ensures that puzzles such as these will not fall under the "Hard" level but will instead be moved up to the "Fiendish" level.

Building the Generator

Existing computer-based puzzle generators use one of two as random generation (RG) techniques, both of which amount to extending the func-

Table 2.
Results of metrics tested against Los Angeles *Times* Sudoku puzzles.

Difficulty	# of Rounds	# of Constraint Subsets of Size 1	# of Constraint Subsets of Size 2
Gentle	8	-	-
Gentle	7	-	-
Gentle	8	-	-
Gentle	10	-	-
Gentle	11	-	-
Gentle	9	-	-
Gentle	10	-	-
Gentle	7	-	-
Gentle	8	-	-
Moderate	8	-	-
Moderate	12	-	-
Moderate	9	-	-
Moderate	14	-	-
Moderate	9	-	-
Moderate	11	-	-
Moderate	10	-	-
Moderate	14	-	-
Moderate	9	-	-
Moderate	13	-	-
Moderate	12	-	-
Moderate	11	-	-
Moderate	11	-	-
Moderate	10	-	-
Moderate	10	-	-
Tough	12	8	1
Tough	11	5	1
Tough	14	3	1
Tough		Failed to solve	
Tough		Failed to solve	
Diabolical		Failed to solve	
Diabolical		Failed to solve	
Diabolical		Failed to solve	
Diabolical		Failed to solve	
Diabolical		Failed to solve	

tionality of a solver to guide the grid construction. (An introduction to making Sudoku grids by hand can be found at Time Intermedia Corporation [2007].)

RG1 (bottom-up generation). Begin with a blank grid.

(a) Add in a random number in a random spot in the grid

(b) Solve puzzle to see if there is a unique solution

  i. If there is a unique solution, proceed to next step

  ii. If there are multiple solutions, return to step (a)

(c) Remove unnecessary hints (any hints whose removal does not change the well-posed nature of the puzzle)

(d) Assess the difficulty of the puzzle, restarting generation if the difficulty is not the desired difficulty.

RG2 (top-down generation). Begin with a solved grid. There are many methods for constructing a solved grid. The one we primarily used builds grids by using a random brute-force solver with the most basic human techniques, placing numbers in obvious places for hidden and naked singles and randomly choosing numbers when no numbers can be placed logically (basic backtracking allows the algorithm to retry random number placement in the cases where it fails to generate a valid solution).

(a) Take out a number in a random spot

(b) Solve puzzle to determine if there is still a unique solution

  i. If there is a unique solution, go to step (a)

  ii. If there are multiple solutions, undo the last removal so that the grid again only has a single solution.

(c) Assess the difficulty of the puzzle, restarting generation if the difficulty is not the desired difficulty.

The first method is best implemented with a depth-first brute-force solver, because when the grid is nearly empty, multiple solutions need to be detected quickly. DLX is a natural choice for generators of this type.

The two methods are very similar: Steps RG1c and RG1d are essentially the same as steps RG2b and RG2c. The second method, however, runs the solver mostly on puzzles with unique solutions, while the first method runs the solver mostly on puzzles with multiple solutions. Solvers based on human techniques are slower and expect to use logic to deduce every square, operating best on multiple solutions. Thus, generators relying on solvers based on human techniques tend to favor the second method over the first.

These methods are driven primarily by random numbers, which is beneficial because random techniques theoretically offer fairly unbiased access to the entire domain of possible puzzles. However, no research that we are

aware of has yet proved that a particular generation method is bias-free. Due to the size of the domain, we are not too worried about small biases. More importantly, these techniques are popular because they operate very quickly, converging for most difficulty levels to valid puzzles within a few seconds. Extremely hard puzzles cannot be dependably generated easily using these techniques, since they are very rare.

Difficulty-Driven Generation

There are two drawbacks to the previously mentioned methods:

- They do not operate with any notion of difficulty, hoping to stumble upon difficult puzzles by chance.
- They require many calls to the solver (on the order of hundreds) due both to the difficulty requirement and the backtracking in case random placements or removals fail. Particularly when using a human-solver, these calls can be very expensive. We do not concern ourselves too much with the runtime, since even generators that require several seconds to generate hard puzzles are tolerable to human users. Online puzzles, puzzles in magazines and newspapers, and puzzles on handheld devices are frequently pre-generated anyway.

We propose a new method of generating puzzles to address the first concern. Our goal is to develop a method that can produce a puzzle of a specified difficulty by guiding the placement of numbers in cells. We call this method - *Generation (DDG)*. DDG is based on merging the human-technique based solvers frequently found in the top-down RG2 method with the bottom-up RG1 method.

1. Begin with an empty grid and a desired difficulty, d. (We use real numbers for our difficulty levels.) Look for a puzzle difficulty $d' \in \mathbb{R}$ with $|d' - d| < t$ for some threshold $t \in \mathbb{R}$.
2. Fill in some number of cells to initialize the grid (well-posed Sudoku puzzles with fewer than 17 cells have not been found, so initializing the empty grid with some number of cells less than 17 is feasible).
3. Solve forward logically with the human-technique solver to remove any obvious cells from consideration.
4. For $i = 1$ to n: (n controls how much the search behaves like a depth-first search and how much it behaves like a breadth-first search)
   (a) Pick a random cell c_i.
   (b) Compute the possible values for c_i, choose a value v_i at random, and fill c_i with v_i.
   (c) Solve forward logically as far as possible. Record the difficulty, d_i.

(d) If we have solved the puzzle and $|d_i - d| < t$, stop and return the grid.

(e) Unfill c_i.

5. Find the value j that minimizes $|d_j - d|$.

6. Recursively call this procedure (enter at step 3) with c_j filled with v_j.

Implementing DDG is highly dependent on the choice of metrics, because it makes the following assumption:

> **DDG Assumption:** During bottom-up grid generation, choosing a cell that makes the puzzle better fit to the desired difficulty at a given iteration will make the final puzzle closer to the overall difficulty.

The construction of the DDG algorithm aims to make this assumption true. Ideally, the choice of later cells in the DDG approach will not significantly change the difficulty of the puzzle, since all cells in the DDG algorithm are subject to the same difficulty constraints. This assumption is not entirely true, particularly due to the complexity of the interactions in Sudoku. However, once we decide on particular metrics, we can alter DDG to make the assumption hold more often.

DDG with the Most Difficult Required Technique

As an example, we consider the metric of the *Most Difficult Required Technique (MDRT)*. Regardless of how many easy moves a player makes when solving a puzzle, the player must be able to perform the most difficult required technique to solve the puzzle. To code DDG with the MDRT, we began with the human-technique based solver of Sudoku Explainer. This solver cannot handle multiple solutions, so we modified it to return partial solutions when it could no longer logically deduce the next move. The solver works by checking a database of techniques (in order of increasing difficulty) against the puzzle to see if any are applicable. When applied, the technique returns both the areas affected by the technique and the specific cells affected (either potential candidates are removed from cells or cells are forced to certain values). Some techniques use the assumption of well-posed puzzle uniqueness to make deductions, which can lead to the solver falsely reporting the puzzle was solved entirely with logical deduction when in reality the puzzle has multiple solutions. To counter this, we use a fast brute-force solver to identify two solutions when there are multiple solutions.

To help ensure the DDG Assumption, we limit the possible locations where cells can be filled in Step 4a. During each recursive step, we find the most difficult technique used in the previous solve step and try to ensure that technique or similar techniques will continue to be required by the puzzle. Sudoku Explainer techniques are ranked using floating point numbers between 1.0 and 11.0 depending on their complexity in the particular

instance when they are applied (for example, hidden singles in blocks are ranked as easier than hidden singles in rows or columns). During Step 4a, no cells or regions affected by the most difficult technique in the previous solve can be filled with values.

Additionally, we use the intersection of the two solutions found by the brute-force solver to indicate which cells are most likely to make the puzzle converge to a unique solution, limiting our cell choices in Step 4a to those cells that were not common to the two solutions. The brute-force solver we use computes the two solutions by depth-first searching in the 'opposite' direction, picking candidates in the search starting at 1 for one solution and 9 for the the other solution, guaranteeing maximal difference between the solutions.

Improvements to Difficulty-Driven Generation

DDG currently can produce a range of solutions for various difficulty levels. However, it operates much slower than the random generation solvers. To optimize the algorithm, we recommend the following adjustments. These adjustments significantly increase the complexity of the algorithm far beyond a guided breadth/depth first search.

- **Tune DDG's desired difficulty levels** at different stages of recursion. The DDG Assumption is entirely valid in practice due to the fact that fewer cells are available to choose from as the algorithm progresses. We recommend that the desired difficulty level start initially high with a high tolerance for the acceptable range of difficulties d_i that should be tried in a recursive step. As more cells are added to the puzzle, the difficulty naturally declines, so the generator should aim higher at the beginning. As the generator progresses, a process similar to simulated annealing should occur with the difficulty level being 'cooled' at the appropriate rate to converge to the desired difficulty level at the same time the puzzle converges to a unique solution of the same difficulty.
- **Make better use of the metrics** to determine exactly which cells can be added at each step with the dual goal of maximizing closeness to uniqueness and minimizing changes to the previously found most difficult required technique. This approach may also allow such algorithms to look for particular techniques. One could imagine that a player desiring extra practice with forcing chains of a certain length could generate several puzzles explicitly preserving forcing chains found during the search process.
- **Sample cell values from a known solution grid** rather than a random distribution. Our experimental results indicate that certain solution grids have a maximum difficulty regardless of how much searching is performed over the grid for a good initial grid. Thus, if this suggestion

is used, after a certain amount of the solution space is searched, the maximum difficulty for that solution grid should be estimated. If the estimated difficulty does not meet the desired difficulty, the solution grid should be regenerated.

DDG Complexity

The DDG algorithm is more complex than RG algorithms, and we believe that the potential benefits outweigh the additional complexity. For generating puzzles with targeted difficulty levels, DDG has the potential to be much more efficient than RG in terms of expected running time and search complexity (number of iterations and branches).

Overall complexity of any generator is highly dependent on the solver used. By using the human technique-based solver, our runtime significantly exceeds the runtime of RG methods using brute-force solvers. We believe that our work on DLX-based difficulty assessment could replace the reliance on the human technique-based solver, significantly accelerating the generator to make it competitive with RG methods in terms of running time. We hypothesize that an optimized DDG will outperform RG for extremely high difficulty levels since the probability of complicated structures arising at random is very small. RG requires several seconds to compute extremely difficult puzzles. (We consider "extremely difficult" puzzles to be those rated over 8.0 on the Sudoku Explainer scale.)

Conclusion

We cast Sudoku as an exact cover problem. Then we present a model of human solving strategies, which allows us to define a natural difficulty metric on Sudoku puzzles. This metric provides four difficulty levels: Easy, Medium, Hard, and Fiendish, each with an additional granularity determined by the number of rounds to complete the puzzle. This metric was tested against puzzles from the Los Angeles *Times*. Our 'Easy' metric encompasses both the 'Gentle' and 'Moderate', but is able to distinguish between puzzles in the category by the number of rounds it takes to complete the puzzle. It also offers additional granularity at the higher end by defining another level that contains constraint sets of size one, which the *Times* lacks. This metric also provides additional accuracy by separating out puzzles from the "Hard" level into a "Fiendish" level that require much more advanced techniques such as Nishio.

Our Difficulty-Driven Generation algorithm customizes to various definitions of difficulty. It builds from existing ideas of random generation algorithms, combining the bottom-up approach with human-technique based solvers to generate puzzles of varying difficulties. Though the generator

requires additional tuning to make it competitive with current generators, we have demonstrated its ability to generate a range of puzzle difficulties.

References

Felgenhauer, Bertram, and Frazer Jarvis. 2006. Mathematics of sudoku I. http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf .

Garns, Howard. 1979. Number place. *Dell Pencil Puzzles & Word Games* #16 (May 1979): 6.

Juillerat, Nicolas. 2007. Sudoku Explainer. http://diuf.unifr.ch/people/juillera/Sudoku/Sudoku.html .

Knuth, Donald E. 2000. Dancing links. 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 .

Lee, Wei-Meng. 2006. Programming Sudoku. Berkeley, CA: Apress.

Mancini, Simona. 2006. Sudoku game: Theory, models and algorithms. Thesis, Politecnico di Torino. http://compalg.inf.elte.hu/~tony/Oktatas/Rozsa/Sudoku-thesis/tesi_Mancini_Simona%2520SUDOKU.pdf .

Pegg, Ed, Jr. 2005. Sudoku variations. Math Games. http://www.maa.org/editorial/mathgames/mathgames_09_05_05.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 .

Stuart, Andrew. 2008. Strategy families. http://www.scanraid.com/Strategy_Families .

Time Intermedia Corporation. 2007. Puzzle generator Japan. http://puzzle.gr.jp/show/English/LetsMakeNPElem/01.

Chris Pong, Martin Hunt, and George Tucker.