"Sudoku Was Not Generated In a Day…"

Fast and True-to-Life Approaches to Sudoku Puzzle Generation

Jong Ho Kang
Jong Wook Kim
Sung Jin Oh

Korea Advanced Institute of Science and Technology

ABSTRACT

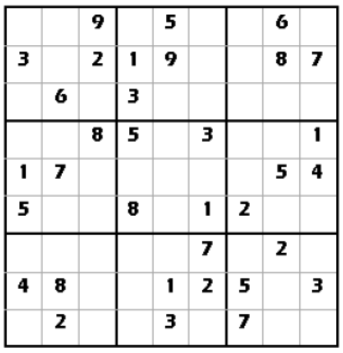
Our purpose was to construct an algorithm that generates Sudoku puzzles with varying difficulty, while also minimizing the complexity of algorithm. We formulate a method to produce Sudoku puzzles with unique solution; a solving algorithm is required to guarantee the uniqueness. Moreover, we focus upon defining a realistic metric of difficulty for Sudoku puzzles. Difficulty level of a Sudoku puzzle is defined to be the sum of complexity of each step involved in solving the puzzle using the solving algorithm.
Then we introduce three different models—the Greedy Back-tracking Model, the Procedural Sudoker Model, and the Probabilistic Sudoker Model—which provide both the solving algorithm and a measurement of the difficulty of Sudoku puzzles. These three have trade-offs in terms of algorithmic complexity and reality; our conclusion is that the Procedural Sudoker Model is the most appropriate model to achieve our purpose, because it provides results both efficiently and realistically. Sample Sudoku puzzles of difficulties Easy, Medium, Hard, and Very Hard generated by our algorithm are provided.
In addition to these works, we present an axiomatic formulation of Sudoku puzzle, which we use as the theoretical basis of the solving techniques in our Sudoker models.

Table of Contents

What is Sudoku?

Sudoku is a puzzle based on logical number placement. Though extremely simple a game, Sudoku puzzle solving often requires a considerable use of logic, and making it an interesting intellectual challenge.

A Sudoku puzzle contains a 9×9 grid, partially filled with numbers from 1 to 9, which is again divided into nine sub-grids (called boxes) of the size 3×3. The one-and-only rule of this puzzle is *to complete the grid with numbers from 1 to 9 so that each column, each row, and each box contain each digit (from 1 to 9) only once.* (To be explained more precisely in the following section) Actually, the name of the game, Sudoku(数独), is the short for a Japanese phrase "*Sūji wa dokushin ni kagiru* (数字は独身に限る)", which can be translated as "*the numbers must occur only once*". [1]



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

Figure 1 A Sudoku Puzzle

The modern puzzle was invented by an American architect, Howard Garns, in 1979 and published by Dell Magazines under the name "Number Place". It became popular in Japan in 1986, after it was published by Nikoli and given the name Sudoku. It became an international hit in 2005.

Not only being an interesting puzzle, Sudoku has lots to offer for mathematicians. Mathematically, completed Sudoku puzzles are a type of Latin square, with an additional constraint on the contents of individual regions (boxes). The study of Latin squares goes back to Leonhard Euler's paper "*De quadratis magicis*", which contains a discussion about magic and Latin squares. [2] It has been proven that the total number of valid Sudoku grids is about 6.67×10^{27}. [3] Even if the symmetry is considered, the total number is about 3.55×10^{12} [3]; simply put, you will never run out of puzzles to solve!

Terminologies and Rules

Here are some basic terminologies regarding Sudoku that we will use throughout the paper. Most of them are rather standard, borrowed from [4].

- **Sudoku grid**: A 9×9 grid of cells, on which the numbers from 1 to 9 is to be filled. (See Figure 2)
- **Column**: The set of nine cells on a Sudoku grid lying on a vertical line. We enumerate the columns starting from the left; hence the 1st and the 9th columns refer to the leftmost and the rightmost columns, respectively. (See Figure 2)
- **Row**: The set of nine cells on a Sudoku grid lying on a horizontal line. Rows are enumerated starting

from the top, so that the 1st and the 9th rows refer to the top and the bottom rows, respectively. (See Figure 2)

- **Box**: One of the nine (disjoint) 3×3 sub-grids that make up the Sudoku grid. (See Figure 2)
- **(*i,j*)th Cell**: The cell on the Sudoku grid lying on the *i*th column and the *j*th row.
- **Region**: We shall refer columns, rows, and boxes as **regions**.
- **Candidates for the (*i,j*)th Cell, or *Cand*(*i,j*)**: The collection of numbers which the (*i,j*)th cell can possibly have.

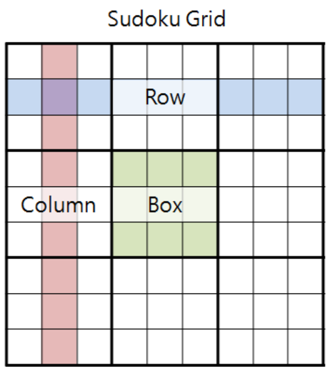


Figure 2 A Sudoku Grid

A Sudoku puzzle is a Sudoku grid which is partially filled with numbers. **The goal of a Sudoku puzzle** is *to fill the Sudoku grid with numbers from 1 to 9*, observing the following simple rule:

- **The Sudoku Rule**: Numbers from 1 to 9 must occur once and only once in each row, column, and box.

A completely filled Sudoku grid, obeying the Sudoku Rule, is called **a solution** to the Sudoku puzzle. A Sudoku puzzle must satisfy the following condition:

- **Existence and Uniqueness of Solution**: There must be one and only one solution to a Sudoku puzzle.

Statement of Purpose

- Our **Goal** is *to construct algorithms* which pose Sudoku puzzles of varying difficulty level.
- In achieving this goal, our **First Priority** is *to define precisely the notion of 'difficulty level' of Sudoku puzzles*, which most closely resembles how human Sudoku players would feel.
- Our **Second Priority** is *to minimize the complexity of the algorithm*.

An Overview of Our Approach

Our basic strategy in generating a Sudoku puzzle of a given difficulty level is simple:

First, we start with a completely filled Sudoku grid, and then

Second, we erase numbers from the grid one by one until

Third, we get a Sudoku puzzle of desired 'difficulty' (a term to be defined).

Figure 3 is the flowchart describing our approach. The final product of this process will be a Sudoku puzzle of desired difficulty, which we shall refer to as the **final Sudoku puzzle**. Sudoku puzzles which occur during this process will be called **intermediate**.

We now elaborate each step.

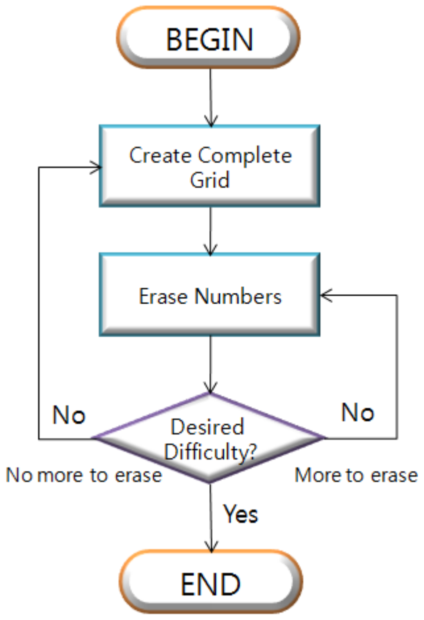


Figure 3 Flowchart of our approach

Step 1. Generating a Completely Filled Sudoku Grid

The reason why we start from a complete Sudoku grid is to ensure the existence of a solution for the final Sudoku puzzle. To generate a complete Sudoku grid, we used a simple back-tracking algorithm, using recursion. The following is the pseudo-code of the function `fillCell(index)`; here `grid[81]` represents the Sudoku grid as an array of 81 values. Provoking `fillCell(0)` will generate a random complete Sudoku grid.

```
bool fillCell(index) :
  if (index==81 ) :
    return true                // the grid has been filled

  set numbers = {1..9}         // this list is filled with numbers
                               1..9 in a random order

  for (i=0 to 9) :
    set grid[index] = numbers[i]

    if(grid is valid) :        // is this Sudoku grid still valid?
      if (fillGrid(index+1)) : // try to fill the next cell
        return true            // 'true' means that the grid has been filled

  return false                 // there is something wrong, so rollback!
```

Step 2. Erasing a number from the Sudoku grid

Each time we erase a number from the Sudoku grid, we check if the resulting intermediate Sudoku puzzle has a unique solution. We achieve this by checking if any of the Sudoku puzzles obtained by filling in the erased cell with numbers other than the original one have a solution. (To solve a Sudoku puzzle, we use one of the models which we will describe below. In this instance, any model would suffice, as long as it can always solve Sudoku puzzles which have solution) If a solution exists in some case, then we write the original number back, and try to erase another number from the grid. The following theorem ensures the uniqueness of solution for Sudoku puzzles generated by our approach:

Theorem. *A Sudoku puzzle obtained by repeatedly applying Step 2 to a complete Sudoku grid has a unique solution.*

Proof. We proceed by induction. With none of the numbers erased, the Sudoku puzzle obviously has a unique solution. Suppose we have a Sudoku puzzle *S*, whose Sudoku grid is not necessarily complete, with a unique solution. Applying Step 2 (if possible) to *S*, a number *n* in a (*i,j*)th cell is erased, and we obtain another Sudoku puzzle *S'*. To prove that *S'* has a unique solution, suppose the contrary. Then we have a solution which differs from the complete Sudoku grid we started with; let us call this *D*. Note that the number in the (*i,j*)th cell of *D*, which we shall call *m*, cannot be *n*; for then *D* becomes a solution of *S* while *S* was supposed to have a unique solution (namely, the one we began with). But then *D* is a solution to the Sudoku puzzle obtained by filling in the (*i,j*)th cell of *S'* by *m*; this is contradictory to Step 2. Hence *S'* has a unique solution. □

Step 3. Checking if the Sudoku puzzle is of the desired difficulty

To carry out this step, we first define a metric of difficulty of Sudoku puzzles. Using this metric, we measure how difficult the intermediate Sudoku puzzle is. If it is right of the desired difficulty, then this becomes our final Sudoku puzzle. However, if the difficulty of the intermediate Sudoku puzzle does not meet our demand, we need to improve the puzzle. If more cells can be erased via Step 2, we go back to Step 2; however, if no more cells are there to be erased (while guaranteeing uniqueness of solution), we go back to Step 1 and start the whole process all over again.

To effectively carry out Step 3, it is crucial to achieve our **First Priority**; to state it again, we need *to define precisely the notion of 'difficulty level' of Sudoku puzzles*, which most closely resembles how human Sudoku players would feel. The obstacles we must overcome when evaluating difficulty of a Sudoku puzzle is as follows:

- **Requirement 1**: Design a model emulating a human Sudoku player, and
- **Requirement 2**: See how 'difficult' it is for that model to solve the given Sudoku puzzle.

Actually, **Requirement 2** is not so hard a problem, for it only involves analyzing the steps the model used; the more intricate and numerous the steps, the more difficult the Sudoku puzzle is. Hence, accomplishing the **First Priority** is largely reduced to achieving **Requirement 1**:

Designing a model which solves Sudoku puzzles, in a way as close as possible to a human Sudoku player

Next three sections describe three models we set up, successively more sophisticated and closer to an actual Sudoku player.

Model 1. The Greedy Back-tracking Model

Model Description

This is the most elementary of the three models, in the sense that it is easiest to implement on computer. The underlying idea is short and simple:

We consecutively guess the value of each cell, until we finally get the whole puzzle right

We obviously want to make plausible guesses; therefore we always make a guess in the cell which has the least number of possible values (or **candidates**) so far. (Hence the adjective 'Greedy') Each guess will eliminate candidates in other cells, according to the **Sudoku Rule**. Of course, some guess may lead to a dead-end, in which some cells have no candidates at all. This proves that the guess was wrong, so we back-track to the point where we made that incorrect guess, and take another shot. (Hence the name 'Back-tracking Model') This process is repeated until the Sudoku puzzle is correctly completed. More precise implementation of the Greedy Back-tracking Model is provided below, in the form of pseudo-code.

```
bool solve() :
   if(the puzzle is solved) :
     return true               // the puzzle is solved!

   find a cell with the smallest number of candidates
   (if there are multiple such cells, repeat the following for each cell)

   for(each candidate of the found cell) :
     save current status
     put in the candidate in the cell
     if(solve()) :             // if the above guess leads to a solution,
       return true             // return true
     else :
       restore status          // if not, come back and guess again

  return false                 // the whole guess was wrong, so roll back!
```

Defining Difficulty

As we can see above, the `solve()` function works recursively, and is called each time we guess a value. Intuitively, the more guesses we have to make, the harder the Sudoku puzzle is; hence it will make sense to make the following definition:

- **The difficulty of a Sudoku puzzle** is *the total number the total number of times we had to provoke the* `solve()` *function it took to solve the puzzle.*

Model 2. The Procedural Sudoker Model

Model Description

This is a model which employs more non-trivial techniques to solve Sudoku puzzles, rather than just guessing all the way. We chose those techniques which are often used when an actual Sudoku player is solving a Sudoku puzzle. The following is the list of the techniques (we mostly followed the terminology of [4])

- **Technique 1**. Single Position
- **Technique 2**. Single Candidate
- **Technique 3**. Candidate Line
- **Technique 4**. Multiple Lines
- **Technique 5**. Naked Pairs/Triples
- **Technique 6**. Hidden Pairs/Triples
- **Technique 7**. X-Wings

- **Technique 8**. Swordfish
- **Technique 9**. Guessing

(For detailed description of each of these techniques, see [4])

The list above is roughly in the order of increasing complexity. The model is 'procedural' in the sense that it always applies the simplest technique (that is, one that is nearest to the top in the above list) available. Figure 2 below is the flowchart which describes how this model proceeds.

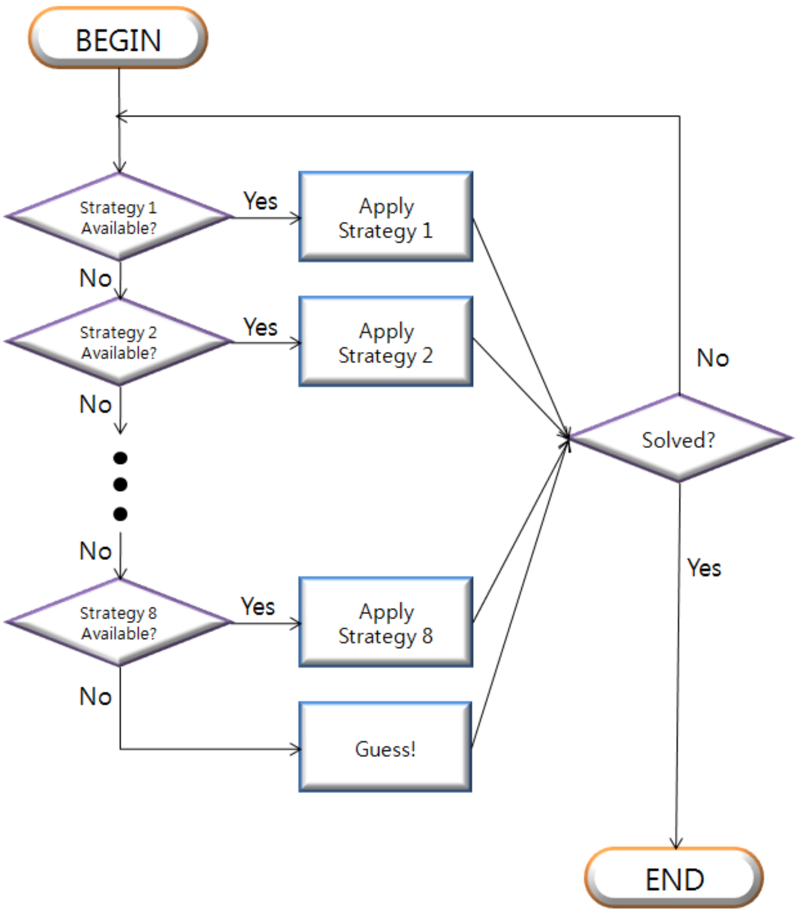


Figure 4 Flowchart describing Model 2

Defining Difficulty

Recall that in this model, different techniques have varying degree of complexity. Thus, in contrast with **Model 1**, not only the number of steps used, but also the intricacy of technique used at each step must be considered to measure the level of difficulty of Sudoku puzzles. Therefore, in this case, it is reasonable to make the following definitions:

- **The difficulty score of a technique** is a number which indicates how complex the technique is.
- **The difficulty of a Sudoku puzzle** is defined to be the sum of difficulty scores of techniques used at each step in the way of solving the puzzle.

Difficulty scores were chosen based on empirical grounds, so that the resulting metric of difficulty seems the closest to how we perceive. The specific values for the difficulty scores of the techniques are given in the following table:

No	Name	Difficulty Score
1	Single Position	10
2	Single Candidate	10
3	Candidate Line	30
4	Multiple Lines	30
5	Naked Pairs/Triples	30
6	Hidden Pairs/Triples	40
7	X-Wings	80
8	Swordfish	120
9	Guessing	50

Table 1 Difficulty Scores of Techniques

Model 3. The Probabilistic Sudoker Model

Model Description

Although **Model 2**, the Procedural Sudoker Model, constructs quite analogous an algorithm to that of a human Sudoku player, it is yet questionable whether a human Sudoku player really does play Sudoku following all the procedures. A more realistic alternative would be as follows: of numerously many ways to proceed with the puzzle, a human Sudoku player finds a configuration where a technique can be applied, namely, a **characteristic spot**.

Spots to apply elementary techniques are relatively easier to detect compared to that of advanced techniques. Thus we can model the **Probabilistic Sudoker** with the following points in mind:

- At each instance, the Probabilistic Sudoker detects a characteristic spot.
- The (average) time taken to find the spot is proportional to the difficulty score of the respective technique.
- The difficulty of a given Sudoku problem is determined by total amount of time taken to solve the puzzle.

The following pseudo-code describes more concretely how we can implement this model:

```
bool solve() :
   until(the puzzle is solved) :
      survey all possible characteristic spots
      for (every moment):
          detect one of the characteristic spots according to its probability
          if(detected):
              apply strategy to the detected spot
              break;
```

In order to implement this model, it is unavoidable to survey the all characteristic spots every step, which drastically intensify the complexity of the algorithm. We will have a more thorough discussion of this fact in the following section.

Defining Difficulty

Being the most realistic of our three models, the Probabilistic Sudoker Model is more like an emulation of an actual Sudoku player rather than just a solving algorithm. A plausible way for a human Sudoku player to determine the difficulty of a Sudoku puzzle would be to measure the time taken to solve it. With the Probabilistic Sudoker Model we can measure the solving time of the puzzle; therefore, we make a definition as follows:

- **The difficulty of a Sudoku puzzle** is the total time elapsed to solve the Sudoku puzzle using the Probabilistic Sudoker Model.

Comparison of the Models

So far we have presented three different models which construct algorithms for solving Sudoku puzzles. All three models can be used for achieving our **Goal**, namely for generating puzzles of varying difficulty; however, each has different strengths and weaknesses. Our conclusion is that **Model 1** is least realistic while computationally the fastest, whereas **Model 3** is most realistic while computationally the slowest. **Model 2** shows moderate performance in both aspects. The diagram below summarizes it.

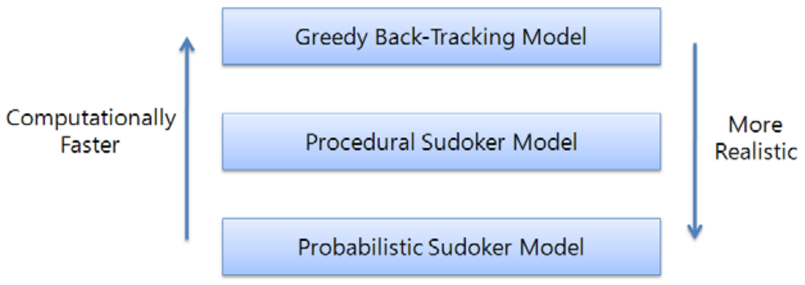


Figure 5 Summary of the Tendency

Quite a few evidences support our conclusion. One of them is the correlation between the average solving time of actual Sudoku players and the difficulty computed according to each model. A hundred Sudoku puzzles, each with the average time it takes for a person to solve it, were obtained from [5]. We measured the difficulty of these puzzles by using our models and compared the results with the given data. The followings are the graphs and the computed correlation coefficients.

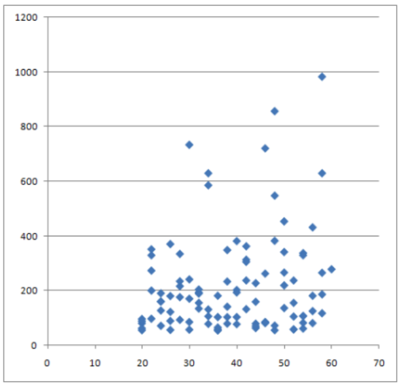


Figure 6 Model 1
Correlation: 0.193

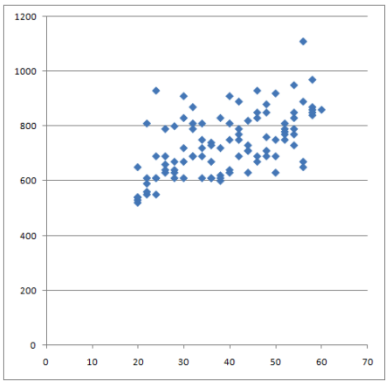


Figure 7 Model 2
Correlation: 0.568

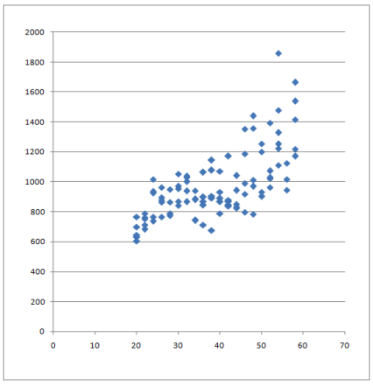


Figure 8 Model 3
Correlation: 0.630

The horizontal axis is the solving time of actual people and the vertical axis is the difficulty computed according to each model. Comparison of the correlation coefficients shows that the difficulty metric of **Model 3** has the strongest positive correlation with the solving time of actual people, and that of **Model 1** has the weakest correlation. Loosely put, **Model 3** reflects reality the best, while **Model 1** does the worst job in this respect.

Additional evidence is the different time complexities of the three models. Let us briefly analyze the time complexity of each algorithm. **Model 1** is basically a back-tracking algorithm, so it has a non-polynomial worst-case time complexity; however the execution time shows that its average-case time complexity is fast enough. In the algorithms of **Model 1** and **Model 2**, finding characteristic spots is quite a difficult job for computers because it has to check all combinations of possible regions for each technique. The depth of iteration statements is at most 6 in our implementation.

Following the analysis of complexities of our models above, it is natural to expect that given a Sudoku puzzle **Model 3** generally would take the longest time to solve it, and that **Model 1** would require the least time. The following figure, which shows average time for solving the same set of a hundred Sudoku puzzles, supports our expectation:

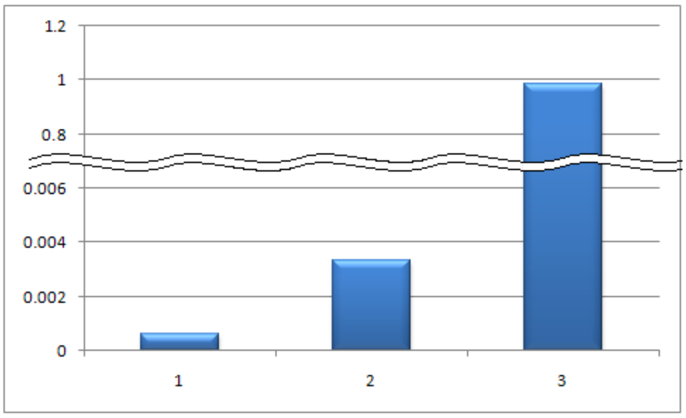


Figure 9 Average solving time for Models 1, 2, and 3

The average time it took for **Model 1** to solve the given problems was 6.21×10^{-4} s. The average time for **Model 2** was 3.33×10^{-3} s, and the average time for **Model 3** was 0.984 s. This shows that **Model 1** was about 1,500 times faster than **Model 3**, and **Model 2** is about 300 times faster than **Model 3**.

The final evidence for our conclusion is the following interesting example. The Sudoku puzzle below is introduced in Wikipedia as the "near worst case" puzzle for brute-force solver, which is just another name for back-tracking solving algorithm. [6]

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

Figure 10 The "Near-Worst" Puzzle for Brute-force Solver

Solving this puzzle via different models yields an interesting result. Whereas this puzzle is solved in **Model 2** and **Model 3** without any guessing, **Model 1** had to make 6,045 back-tracks to solve it. (Compare this result with the fact that most of the conventional Sudoku puzzles are solved with less than 10 back-tracks) This example illustrates that a puzzle easy for men, plus **Models 2** and **3**, may be extremely difficult for **Model 1**. This buttresses our conclusion that **Model 1** is the least realistic of all.

Conclusions & Results

From the results and discussions above, we know that **Models 1**, **2**, and **3** mimic successively better an actual Sudoku player. Recall that the problem of achieving the **First Priority** has largely been reduced to finding a model which most closely resembles a human Sudoku player; in this respect, **Model 3** seems to be the best fit.

However, we must also take into account our **Second Priority**: to minimize the complexity of the algorithm. As we discussed earlier, analysis of complexities of the models shows that **Model 3** has the highest complexity. In particular, Figure 9 indicates that given the data of a hundred conventional Sudoku puzzles, **Model 1** and **Model 2** performed 1,500 and 300 times faster than **Model 3**, respectively. Simply put, while it sure does closely follow a human Sudoku solver, **Model 3** is just too slow. The best alternative is **Model 2**, which is both low in complexity and highly realistic. (**Model 1** is not just realistic enough; recall what happened when Model 1 tried to solve the puzzle in Figure 10) Hence we make our main conclusion:

It is best to implement ***Model 2****, the* ***Procedural Sudoker Model****, to make our difficulty level realistic and our overall algorithm less complex.*

Therefore, we shall use **Model 2** to complete our algorithm, which generates Sudoku puzzles of varying difficulty.

Recall that the difficulty of a Sudoku puzzle is defined to be the sum of difficulty scores of techniques used to solve the puzzle. We divide the whole range of difficulty into four levels as follows:

- Easy : ~600
- Medium : 600~800
- Hard : 800~1000
- Very Hard : 1000~

These figures were chosen based on comparisons with the database of Sudoku puzzles from [5]. Now using our algorithm, we can make Sudoku puzzles of any desired difficulty. As an example, we include below Sudoku puzzles of various difficulty levels produced by our algorithm. For those who dare, enjoy!

EASY:

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

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

MEDIUM:

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

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

HARD:

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

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

VERY HARD:

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

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

Strengths and Weaknesses

Strengths

- We provided three different models, each appropriate for fast, balanced, and realistic applications.
- All our works were based on mathematically and logically sound methodologies.
- Metrics of difficulty of our models are extensible to a varying number of difficulty levels.

Weaknesses

- The logical basis for selecting difficulty scores of techniques (in **Model 2**) was rather weak, since we could not provide more than just an empirical justification.
- Much larger database of puzzles, each with an average solving time of actual Sudoku players, is needed to strengthen our conclusions

Possible Improvements

Generalization to Grids of Other Sizes

All of our models can easily be generalized to Sudoku grids of any other sizes. As an example, 4x4 and 16x16 Sudoku grids are shown below. Since all models use fast back-tracking or polynomial time algorithms, varying size of the grid affects no more than multiplication by a constant on the time complexities.

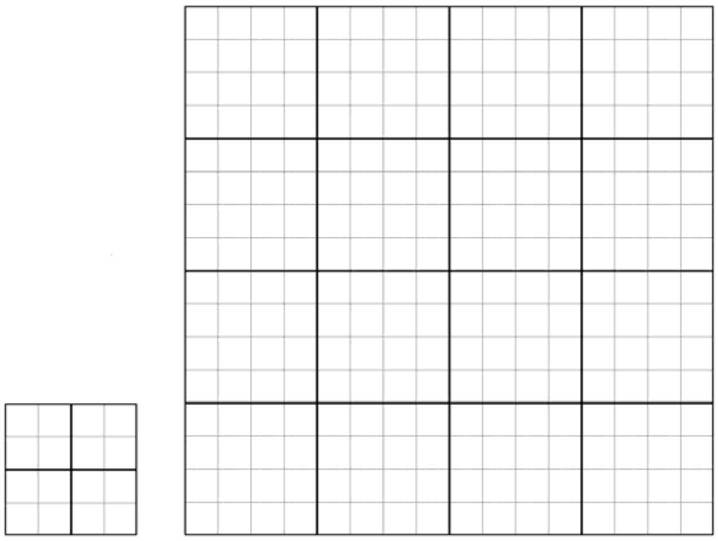


Figure 11 4x4 and 16x16 Sudoku grids

Introduction of a Parametric Binary Search Algorithm

It is possible to introduce a parametric binary search algorithm in the process of making a Sudoku puzzle of desired difficulty. Solvability of the puzzle can be quickly determined by the back-tracking algorithm (**Model 1**), and difficulty can be measured only at sparse points, where a parametric binary search is used. This approach change linear time complexity into logarithmic time complexity, effectively reducing the amount of calculations.

Appendix . Axiomatic Theory of Sudoku

In the appendix we provide justifications for the techniques that we implemented in **Model 2** and **Model 3**. To provide rigor, we take an axiomatic approach toward the game of Sudoku. We start with basic definitions:

Definitions.

i. *N is the set {1, 2, 3, 4, 5, 6, 7, 8, 9}, and* $G = N \times N$ *(G symbolizes Grid)*

ii. *A column is a set of the form* $\{i\} \times N$, *for* $i \in N$

iii. *A row is a set of the form* $N \times \{j\}$, *for* $j \in N$

iv. *A box is a set of the form* $\{(i,j) | i,j \in S\}$, *where* $S = \{1,2,3\}, \{4,5,6\}$, *or* $\{7,8,9\}$.

Let $O \subset G$ and $\varphi: O \to N$ (φ is a function). The set O represents the cells of the Sudoku puzzle which is filled from the beginning; $\varphi(i,j)$ for $(i,j) \in O$ represents the number inside the $(i,j)^{th}$ cell.

The **Sudoku Rule** and the **Existence and Uniqueness of Solution** are expressed as the following axiom:

Axiom. *There exists one and only one function* $\sigma: G \to N$ *such that*

i. $\sigma|O = \varphi$

ii. $\sigma(R) = N$ *for every region* R

The following definition is often useful:

Definition. *A subset M of N such that* $\sigma(S) \subseteq M$ *is called a set of candidates for S*

The following lemma is the cornerstone for the whole axiomatic theory.

Lemma. (1-1 Lemma) $\sigma|R : R \to N$ *is 1-1 correspondence for every region R*

Proof. Obviously, $\sigma|R$ is an onto function. If $\sigma|R$ is not one-to-one, then clearly $\#(\sigma|R) < \#R = 9$; however $9 = \#N = \#(\sigma|R) = \#R$. This shows that $\sigma|R$ is one-to-one, too. □

Now we justify the techniques we mentioned above, each under the name of a **Theorem**.

Theorem. (Single Position) $\sigma(i,j) = n$ *if for a region R containing (i, j), there exists a set of candidates* $cand(R\backslash\{(i,j)\})$ *for* $R\backslash\{(i,j)\}$ *such that* $n \notin cand(R\backslash\{(i,j)\})$

Proof. If there exists a set of candidates $cand(R\backslash\{(i,j)\})$ for $R\backslash\{(i,j)\}$ such that $n \notin cand(R\backslash\{(i,j)\})$ where *R* is a region containing (i,j), then we have $n \notin \sigma(R\backslash\{(x,y)\}) = \sigma|R(R\backslash\{(x,y)\})$. Since $\sigma|R(R) = N$, $n \in N,$ and $\sigma|R : R \to N$ is an 1-1 correspondence by the 1-1 Lemma, we have $\sigma(x,y) = n$. □

Theorem. (Candidate Line) *Let* R_1, R_2 *be regions. If there exists a set of candidates* $Cand(R_1\backslash R_2)$ *of* $R_1\backslash R_2$ *such that* $n \notin Cand(R_1\backslash R_2)$*, then* $n \notin \sigma(R_1\backslash R_2)$ *and* $n \in \sigma(R_1 \cap R_2)$*.*

Proof. Since $\sigma(R_1\backslash R_2) \subset Cand(R_1\backslash R_2)$,

$$n \notin Cand(R_1\backslash R_2) \Rightarrow n \notin \sigma(R_1\backslash R_2) = \sigma|R_1(R_1\backslash R_2).$$

If $n \notin \sigma(R_1 \cap R_2) = \sigma|R_1(R_1 \cap R_2)$, then we must have $n \in \sigma|R_1\big(R_1\backslash(R_1 \cap R_2)\big) = \sigma|R_1(R_1\backslash R_2)$ since $n \in \sigma|R_1(R_1) = N$ and $\sigma|R_1 : R_1 \to N$ is an 1-1 correspondence by 1-1 Lemma; this is a contradiction. Hence $n \in \sigma(R_1 \cap R_2)$. □

Theorem. (Naked Pair, Triple, and Etc) *Let* $Cand(S)$ *be a set of candidates for* S*. For a region R,* $S \subseteq R$*, if* $\#S = \#Cand(S)$*, then* $\sigma(S) = Cand(S)$*.*

Proof. If $\sigma(S) \neq Cand(S)$, $\sigma(S) < Cand(S)$. Since $\sigma(S) = \sigma|R(S)$, *S* and *N* are finite sets, and $\sigma|R : R \to N$ is an 1-1 correspondence by the 1-1 Lemma, we have $\#\sigma(S) = \#S = \#Cand(S)$; this is a contradiction. Hence $\sigma(S) = Cand(S)$. □

Corollary. (Single Candidate theorem) *Let* $Cand(S)$ *be a set of candidates for* S*. For a region R,* $S \subseteq R$*. If* $\#S = \#Cand(S) = 1$*, then* $\sigma(S) = Cand(S)$*.*

Theorem. (Hidden Pair, Triple, and Etc) *Let* M *be a subset of* $N = \{1, \ldots ,9\}$*. If there exists a set* $Cand(R\backslash S)$ *of candidates for* $R\backslash S$ *such that* $Cand(R\backslash S) \cap M = \emptyset$*, where R is a region and* $S \subseteq R$*, and if #S=#M, then*

$\sigma(S) = M$

Proof. Since $\sigma(R\backslash S) \subset Cand(R\backslash S)$,

$$Cand(R\backslash S) \cap M = \emptyset \Rightarrow \sigma(R\backslash S) \cap M = \sigma|R(R\backslash S) \cap M = \emptyset.$$

Since $\sigma|R : R \to N$ is an 1-1 correspondence by the 1-1 Lemma, $\sigma|R(R\backslash S) \cup \sigma|R(S) = \sigma|R(R) = N$. Since $M \subseteq N$ and $\sigma|R(R\backslash S) \cap M = \emptyset$, $M \subseteq \sigma|R(S)$. Note that $\#\sigma|R(S) = \#S = \#M$. Therefore it follows that $M = \sigma|R(S) = \sigma(S)$. □

References

1. History of Sudoku- http://www.sudoku-tips.com/about_sudoku.php
2. Leonhard Euler. On magic squares- http://arxiv.org/abs/math.CO/0408230
3. http://www.shef.ac.uk/~pm1afj/sudoku/
4. Sudoku of the Day- http://sudokuoftheday.com
5. (Korean Book) 샘 그리피스존스, "두뇌의 힘을 키우는 스도쿠 논리퍼즐 2". 황금부엉이
6. Algorithmics of Sudoku (Wikipedia)- http://en.wikipedia.org/wiki/Algorithmics_of_sudoku