When Sudoku Comes across Information Theory

Team # 2513

February 18, 2008

Abstract

Sudoku is known to be a NP-Complete problem. Generally, the construction algorithms guarantee that a Sudoku puzzle has a unique solution through solving the puzzle many times by searching. Therefore, the efficiency of the search largely determines the efficiency of the solving part, hence the efficiency of the whole algorithm. So finding an effective search method is the key to optimize the construction algorithm.

In this paper, based on the ***information theory***, solving a Sudoku puzzle is abstracted into a process in which the ***entropy*** of an initial grid decreases along with the decrease of the number of the empty cells. The quicker the entropy decreases, the shorter the solving process will be. Based on the analysis above, to decrease the complicity of the searching method, we modify the order of filling empty cells to make the entropy decrease maximized in each step. Consequently, the solving part reaches the best in theory. Then a trial and error construction algorithm with the solving part optimized with the information theory is proposed. Moreover, the entropy of the initial Sudoku grid is chosen to define the difficulty levels.

The theoretical analysis and computer simulations show, based on the information theory, the complexity of the optimized searching method decreases evidently. Especially for an extremely difficult one, the time needed is 0.5% of that of the general methods. Therefore, optimizing under the guideline of the information theory is scientific and efficient.

Finally, our algorithm is illustrated in 6 difficulty levels.

Contents

Abstract 1
1 Introduction 3
1.1 Background 3
1.2 Problem Restatement 4
1.3 How to Define the Difficulty Levels 4
1.4 How to Guarantee the Uniqueness of Solution 5
1.5 Our Ideas 5
2 Basic Model 5
2.1 Two General Methods 5
2.2 The Metric to Define the Difficulty 6
2.3 Framework of the Algorithm 6
2.4 Description of the Algorithm 6
2.5 Algorithm of Solving a Sudoku Grid 7
3 Optimazing Model Based on Information Theory 10
3.1 Motivation 10
3.2 Basic Knowledge of Information Theory 10
3.3 Sudoku Puzzles in Eyes of Information Theory 11
3.4 The Metric to Define the Difficulty Levels 14
3.5 Review of the Algorithm 14
3.6 How to Measure the Information Given by Each Step 15
3.7 Description of the Algorithm 16
3.8 Algorithm of Solving a Sudoku Grid 16
4 Testing and Comparison 17
4.1 Testing of Our Second Algorithm 17
4.2 Comparison between Different Searching Strategies 17
5 Why Our Second Algorithm Is Effective 20
6 Conclusions 21
7 Algorithm Illustration Scheme 21
8 Strengths and Weaknesses 22
8.1 Strengths 22
8.2 Weaknesses 22
9 Future Attempt 23
References 24
A Sudoku puzzles of Six difficulty levels illustrated 27
B Algorithm programmes based on our model 28

1 Introduction

1.1 Background

Sudoku is a logic-based number placement puzzle. The aim of the most popular form is to fill a 9×9 matrix of cells with digits from 1 through 9. Each puzzle consists of a grid in which some digits have already been filled and the goal is to fill in the remaining cells so that each digit appears once in each row, column and subgrid. For a correctly formed Sudoku puzzle, there should be only one solution. **Fig. 1** (taken from [1]) shows one such puzzle with thirty-one clues given.

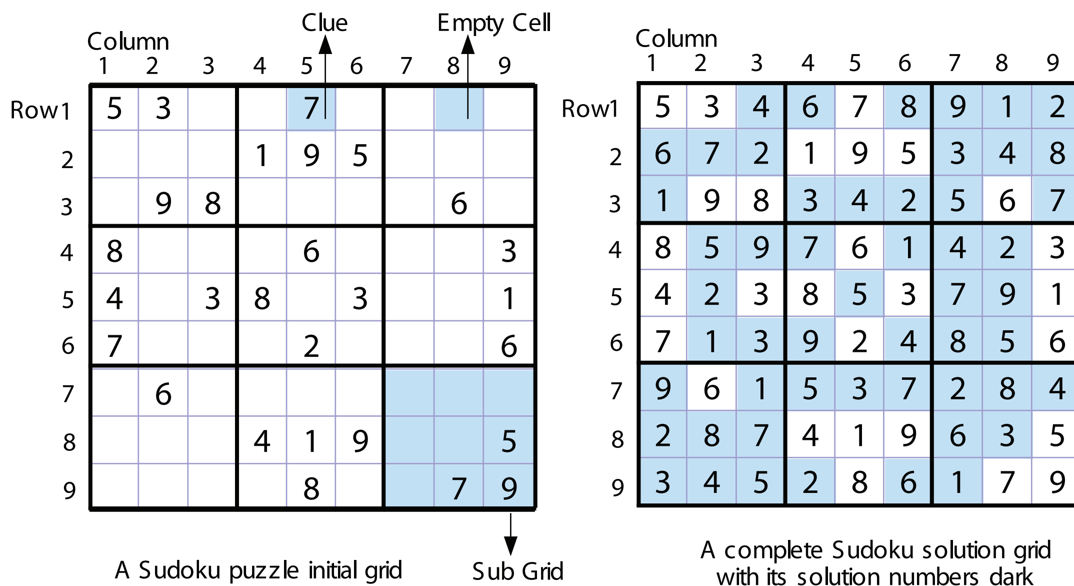

Fig 1: Sudoku Puzzle

There are some extremely interesting questions among Sudoku fans:

- How many Sudoku puzzles are there? The total number of classic 9×9 Sudoku solution grids was 6.67×10^{21} proved by Bertram Felgenhauer and Frazer Jarvis in 2005 [2].
- What is the minimum number of clues that a Sudoku puzzle should have? The answer to this question is not known, possibly 17 [3].
- What is the minimum number of different digits that a Sudoku puzzle can have? From the proof by graph theory, this number is 8 [4].

Though Sudoku is just a puzzle, the mathematical logic behind it is wondrous. Scientists attempt to research Sudoku by theories of other branches, such as graph theory, group theory, and so on. Various algorithms have also been proposed to solve the problem, such as, backtracking searching, genetic algorithm. Since the sufficient condition of uniqueness of the Sudoku puzzle has not been found, the algorithms which construct a Sudoku puzzle guarantee uniqueness of solution through searching all the solutions, and Sudoku is proved to be NP-complete problem [5], thus the development of efficient Sudoku construction algorithm with different difficulty levels is a challenging task.

1.2 Problem Restatement

The problem consists of four parts:

- Develop an algorithm to construct Sudoku puzzles with unique solution;
- Develop metrics to define difficulty levels;
- The algorithm and metrics should be extensible to at least 4 difficulty levels;
- Analyze the complexity of our algorithm.

Our objective is to minimize the complexity of the algorithm and meet the above constraints.

1.3 How to Define the Difficulty Levels

There have been many ways to define the difficulty of a Sudoku puzzle. One common metric is based on the complexity of the solving techniques required. Another approach relies on a median solving time obtained from the experience of a group of people who have tried to solve it.

We consider that the whole process of solving a Sudoku puzzle could be viewed as one of eliminating the uncertainty containing in the initial grid. And the uncertainty is relative to the number and positions of the clues. Therefore, we try to find a volume which can reflect the uncertainty reasonably as the metric of the difficulty levels.

1.4 How to Guarantee the Uniqueness of Solution

Over the years, experts dedicated to the problem: how to determine the uniqueness of solution of a Sudoku puzzle. In 2007, an article titled "Sudoku Squares and Chromatic Polynomials" applies a method to analyzing the puzzles with graph theory. It proved that the number of solutions can be given by a chromatic polynomial. But the polynomial can't be obtained easily. So we have to accept the fact that there are no specific guide lines, except for solving it, used to guarantee the solution's uniqueness.

1.5 Our Ideas

As we know, the mysteries hidden behind the Sudoku are related to many fields of knowledge. And there are many theories used to describe the Sudoku puzzles. Our aim is not only to develop an algorithm which satisfies the requirement of the problem, but to analyze the Sudoku puzzle deeply and expect to find a theory which can abstract Sudoku' s essential law. Then the algorithm to construct Sudoku puzzles could be optimized as much as possible under the guide of the theory, avoiding merely focusing on the local improvement of the algorithm with the common programming skills.

2 Basic Model

2.1 Two General Methods

The two main methods to generate Sudoku puzzles are listed below:

- **Method 1:** Generate a standard filled grid, then remove some digits and check whether there is a unique solution.
- **Method 2:** Start from an empty grid, fill it with digits step by step, and check whether it meets the constraint of unique solution.

In the former method, we develop two algorithms to generate a complete Sudoku solution grid. One is to fill all the cells with random digits and check the grid, looping if necessary, until it is valid. This algorithm is easy to implement but lack of efficiency. The other is to transform an existing complete grid into a new one by rotating, reflecting, exchanging its columns or rows and so on. This algorithm is efficient but unable to generate "essentially different" Sudoku grids.

In our models, the latter method is adopted for it is enables us to generate "essentially different" Sudoku grids and the efficiency of which is acceptable.

2.2 The Metric to Define the Difficulty

It is obvious that the initial Sudoku grid with 80 clues is much easier than the one with only 17 clues. Therefore, it seems that it is the number of the clues that determines the difficulty. For simplicity, we ignore the difference of the empty cells, view them equally, and use the number of the empty cells to measure the level of the difficulty. The bigger the number is, the harder the puzzle will be.

2.3 Framework of the Algorithm

Our algorithm includes three basic sections: trial part, solving part and measuring part.

The trail part has two operations: filling and removing. Filling is to put a proper digit into an empty cell. Removing is to empty a filled cell.

The function of the solving part is to work out the puzzle through searching. This part will be used frequently in the algorithm, because uniqueness of the solution is checked through it. So its performance will largely determine the efficiency of the whole algorithm.

The last part is the measuring part, testing whether the Sudoku puzzle created belongs to the required difficulty level.

2.4 Description of the Algorithm

At the beginning, there is an empty grid in the size of 9×9. The algorithm starts with the filling operation. For the empty cells are treated equally, one of them is chosen randomly and filled with an available random digit. Then the solving part begins to search for the solutions. If there is no solution, remove the digit of the cell filled before. If just one solution exists, go to the measuring part. If the solutions are more than 1, the filling operation is chosen again. Then repeat the cycle until the number of solutions is just 1.

In the measuring part, if the puzzle is fit for the required difficulty level, the algorithm stops. If not, randomly fill one empty cell if the puzzle is harder than requirement, or randomly remove one clue if easier until the Sudoku puzzle fits into the required difficulty level. If the number of solutions is not 1 after the operation, undo it and repeat the cycle. If the puzzle cannot reach the requirement finally, return to the very beginning–the empty grid.

Fig. 2 shows the basic flowchart of the algorithm above.

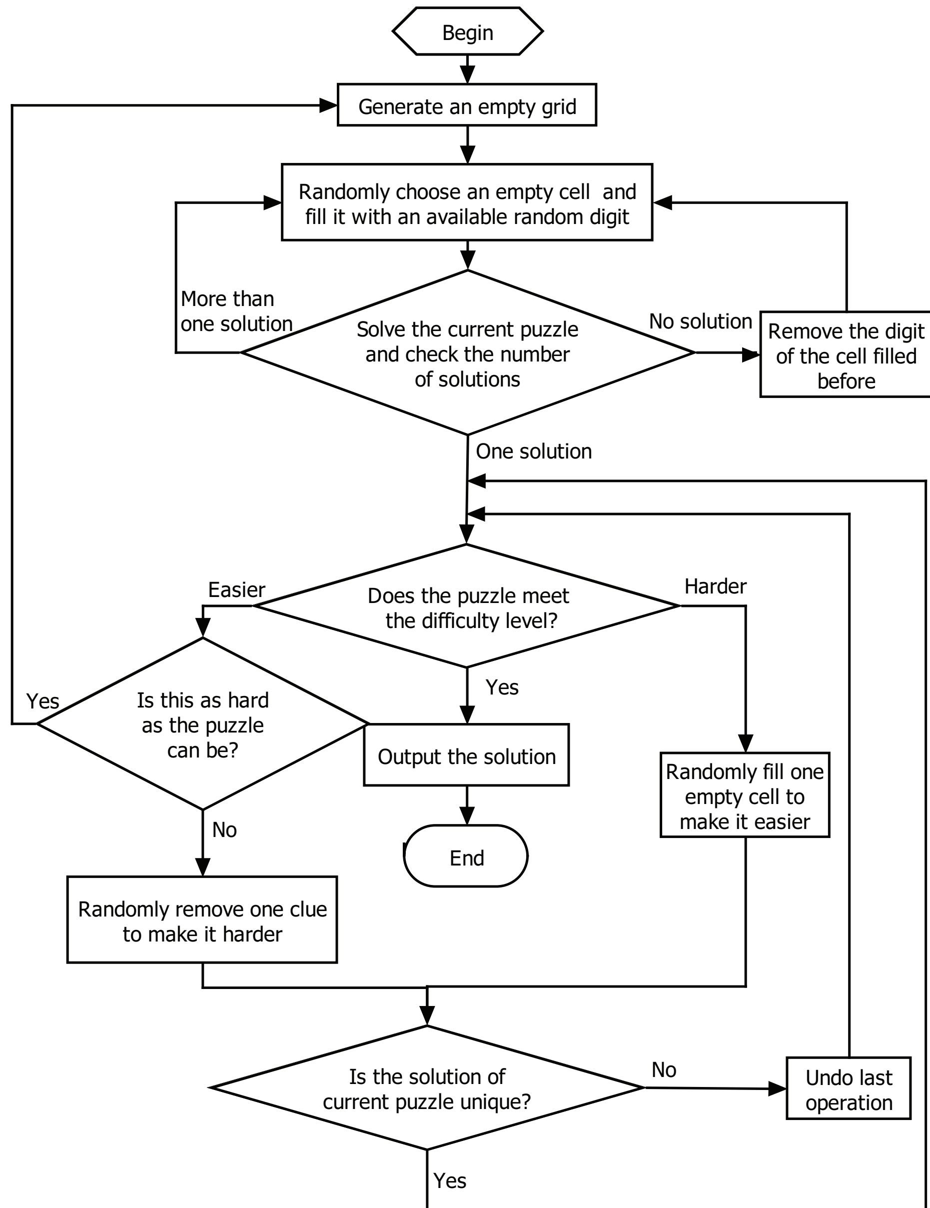

Fig 2: Basic flowchart of Sudoku algorithm

2.5 Algorithm of Solving a Sudoku Grid

As noticed above, this part will largely determine the efficiency of the whole algorithm, so great emphasis is put on it and detailed description is given below:

Definitions: Conflict can be defined as below:

One cell conflicting with other clues means that the digit in the cell has already appeared in the same row, column or subgrid.

Notations:

B	The set consists of all empty cells in a given Sudoku grid.
$num(B)$	The total number of elements in set B.
b_i	The ith cell of B.
V_i	The set consists of all the candidates of b_i .
v_{ij}	The jth digit of V_i.

Main Steps:

- **Step 1: Determining.**
  Determine the filling order of the empty cells, that is, which cell would be filled first, which would be filled next in searching. In this model, the filling order is simple: row by row.

- **Step 2: Searching.**
  Depth-first search the solution by the determined order.

Detailed Steps:
In this section, the main steps are separated into several further steps(see **Fig. 3**).

- **Step 1:** Scan the whole Sudoku grid row by row, and record the positions of all empty cells in B.

- **Step 2:** For each empty cell b_i in B , do Step 3 to Step 8.

- **Step 3:** Determine the candidates set V_i of b_i so that each candidate in V_i won't cause b_i conflicting with other clues.

- **Step 4:** Judge whether V_i is an empty set, if yes, let $i \leftarrow i-1$, and backtrack (go to Step 5) if $i > 0$, or go to Step 9.

- **Step 5:** For each value v_{ij} in V_i, do Step 6 and Step 7.

- **Step 6:** Fill b_i with v_{ij} so that b_i becomes a clue.

- **Step 7:** If $i < num(B)$, let $i \leftarrow i+1$, do recursion (go to Step 2), else output current grid as a solution.

- **Step 8:** Let $i \leftarrow i-1$, and if $i > 0$ backtrack (go to Step 5), else go to Step 9.

- **Step 9:** If the solution has not been found, the given grid has no solution.

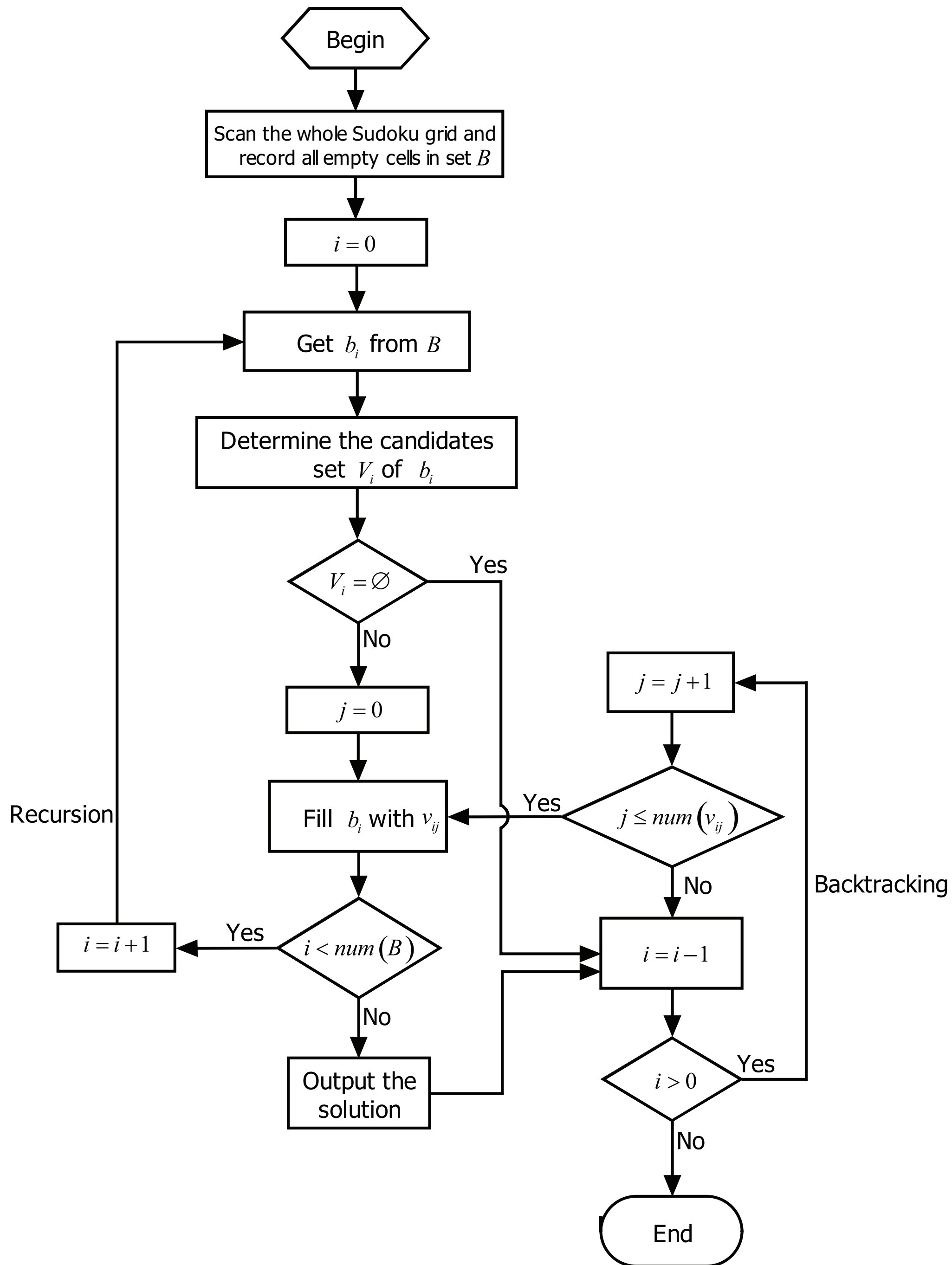

Fig 3: Flowchart of basic search algorithm

3 Optimazing Model Based on Information Theory

3.1 Motivation

What on earth determines the difficulty of a Sudoku puzzle? Sure, the smaller the number of the clues is, the more difficult the puzzle will be. But cells in a Sudoku puzzle are associated with each other through rows, columns and subgrids. It is common to find that an easy puzzle becomes a hard one after the clue positions are changed. So the overall arrangement of a Sudoku initial grid is also important to determine the difficulty. Then how to combine the clue number with the clue initial positions and furthermore, quantize the result to several difficulty levels?

The process of solving a Sudoku puzzle can be viewed as one in which the unknown situations are analyzed by what have been obtained. And each step of all the attempts will give out some hint, i.e., information of the given Sudoku puzzle's solution. Based on what is mentioned above, we use the information theory to develop the metric to define the difficulty of Sudoku puzzles.

3.2 Basic Knowledge of Information Theory

Information Theory [6]is a branch of applied mathematics and engineering involving the quantification of information. It is widely used in the study of communications now. To use conveniently, the introduction to the concepts of entropy[9], conditional entropy, and mutual information is given here.

Let X denote a discrete random variable with alphabet $\{x_i, i = 1, 2, \ldots, m\}$and probability mass function $p(x_i) = \Pr(X = x_i)$. Then the entropy of X , which deserves as the measure of the information or the amount of uncertainty contained in X is defined as

$$H(X) = -\sum_{i=1}^{m} p(x_i) \log_2 p(x_i) \tag{1}$$

Especially, when $x_1 = x_2 = \cdots = x_K = \frac{1}{K}$, $H(X) = \log K$.

Let (X, Y) denote a two-dimensional discrete random variable with alphabet $\{(x_i, y_j), i = 1, 2, \cdots, m, j = 1, 2, \cdots n\}$and joint distribution $p(x_i, y_j)$. The conditional entropy or conditional uncertainty of X given random variable Y is defined below:

$$H(X|Y) = -\sum_{i=1}^{m}\sum_{j=1}^{n} p(x_i, y_j) \log_2 p(x_i|y_j) \tag{2}$$

The mutual information of X relative to Y is given by

$$I(X;Y) = -\sum_{i=1}^{m}\sum_{j=1}^{n} p(x_i, y_j) \log_2 \frac{p(x_i|y_j)}{p(x_i)} \tag{3}$$

It measures the amount of information that can be obtained about one random variable by observing another.

The relationship between the three above can be indicated as follows:

$$I(X;Y) = H(X) - H(X|Y) \tag{4}$$

That is, knowing Y, we can obtain an average of $I(X;Y)$ bits information compared to not knowing Y.

3.3 Sudoku Puzzles in Eyes of Information Theory

When a Sudoku initial grid is given, the clues in it represent the certainty, and the empty cells the uncertainty contained in the puzzle. If we fill each of the blanks randomly with digits which are proper according to the initial clues, the varied completions will be obtained. Some of the completions are solutions to the sudoku puzzle, but the others are not because they violate the constraints (some digits appear twice or more in the same row, column or subgrid). Based on the theory of probability, the result of completions can be treated as a random variable, which is denoted by X. Each of the completions appears in accordance with a certain probability. According to the definition that has been given before, the entropy $H(X)$ is the value that can represent quite accurately the uncertainty in the initial Sudoku grid.

Now we choose an empty cell, fill it with an appropriate digit, denoted as Y. This operation will give us some further information of the completion, which could be measured with the mutual information $I(X;Y)$. The remaining amount of the uncertainty after this operation can be measured with conditional entropy $H(X|Y)$. The relationship between them can be described as:

$$H(X|Y) = H(X) - I(X;Y) \tag{5}$$

An example to illustrate the above analysis is given below.

Example:

Consider the initial Sudoku grid in **Fig. 4**. Let (i,j,k) denote the coordinates of the cell which lies in the ith row, the jth column and the kth subgrid (For instance, the digit in the cell with coordinates (1,2,1) is 1). Then let $n(i,j,k)$ denote the number of the digits that could be filled in the empty cell.

Denoted by D_i , the set of the digits, which have been filled in the ith row, E_j the set of the digits in the jth column, and F_k the set of the digits in the kth subgrid, $n(i,j,k)$ could be calculated by the formula below:

$$n(i,j,k) = 9 - \left|D_i \cup E_j \cup F_k\right| \tag{6}$$

where $|\cdot|$ denotes the cardinality of a set.

Row \ Column	1	2	3	4	5	6	7	8	9
Row 1		1							
2				3			8		
3	9						6		
4	7				1	2	4		
5	5		3						
6	8								
7				6					
8					4			2	
9				7				5	

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

Fig 4: An Sudoku example

Let N denote the number of the empty cells in the Sudoku initial grid. Then the total number of the completion of the given initial Sudoku grids M could be obtained by Multiplication Principle:

$$M = \prod_{i,j,k} n\,(i,j,k) \tag{7}$$

The probability of each completion is equal to $\frac{1}{M}$, so the entropy of the Sudoku puzzle in Figure 1 could be obtained by the formula:

$$\begin{aligned} H\,(X) &= -\sum_{m=1}^{M} \tfrac{1}{M} \log_2 \tfrac{1}{M} = \log_2 M \\ &= \log_2 \left(\prod_{i,j,k} n\,(i,j,k) \right) \\ &= \sum_{i,j,k} \log_2 n\,(i,j,k) \end{aligned} \tag{8}$$

Now we choose the cell $(3,1,1)$. The empty cell is filled with a digit, denoted by Y_1 under the rules of the Sudoku. This operation will affect the empty cells which are in the same row, column or subgrid. For example, the digit 9 is filled into the cell. Consequently, the cells which are dark in **Fig. 5** can not be filled with 9 in following steps. That is to say, as for these cells which could be filled with 9 before, the number of the available digits reduces by 1.

Row \ Column	1	2	3	4	5	6	7	8	9
Row 1		1							
2				3			8		
3	9						6		
4	7				1	2	4		
5	5		3						
6	8								
7				6					
8					4			2	
9				7				5	

Fig 5: The Sudoku example after one operation

Let $n_1(i,j,k)$ denote the number of the digits that could be filled in the remained empty cell after the operation. The total number of the updated Sudoku's completions M_1 could be obtained:

$$M_1 = \prod_{i,j,k} n_1(i,j,k) \tag{9}$$

The remaining amount of the uncertainty $H(X|Y_1)$ could be obtained by the formula:

$$\begin{aligned} H(X|Y_1) &= \log_2 M_1 \\ &= \log_2 \left(\prod_{i,j,k} n_1(i,j,k) \right) \\ &= \sum_{i,j,k} \log_2 n_1(i,j,k) \end{aligned} \tag{10}$$

So the mutual information $I(X;Y_1)$ of this operation is:

$$\begin{aligned} I(X;Y_1) &= H(X) - H(X|Y_1) \\ &= \sum_{i,j,k} \log_2 n(i,j,k) - \sum_{i,j,k} \log_2 n_1(i,j,k) \\ &= \sum_{i,j,k} \log_2 \frac{n(i,j,k)}{n_1(i,j,k)} \end{aligned} \tag{11}$$

After the step, the cell $(3,1,1)$ could be viewed as a clue. In our example, the digit of the cell is fixed to 9, and it will not be affected by other cells. Now the number of the empty cells decreases to $N-1$.

We choose another empty cell and denote it by Y_2 , then $H(X|Y_1Y_2)$ and $I(X|Y_1;Y_2)$ could be obtained like above. After the rth step, there are r equations:

$$\begin{aligned} I(X;Y_1) &= H(X) - H(X|Y_1) \\ I(X|Y_1;Y_2) &= H(X|Y_1) - H(X|Y_1Y_2) \\ &\vdots \\ I(X|Y_1,\cdots,Y_{r-1};Y_r) &= H(X|Y_1,\cdots,Y_{r-1}) - H(X|Y_1,\cdots,Y_r) \end{aligned} \tag{12}$$

Adding both sides of the equations, we obtain

$$H(X) = I(X;Y_1) + I(X|Y_1;Y_2) + I(X|Y_1,\cdots,Y_{r-1};Y_r) + \cdots + H(X|Y_1,\cdots,Y_r) \tag{13}$$

The equation shows that the remaining amount of the uncertainty after r steps is equal to the difference of the entropy of the initial Sudoku grid and the sum of the mutual information obtained in each step. The more the information given by each step is, the less the uncertainty will remain.

3.4 The Metric to Define the Difficulty Levels

As we have explained above, the entropy $H(X)$ can represent the uncertainty in the Sudoku initial grid. The difficulty is relative to the uncertainty directly. From the formula:

$$H(X) = \sum_{i,j,k} \log_2 n(i,j,k) \tag{14}$$

it can be seen that both the number and the overall arrangement of the clues in the Sudoku initial grid are involved in the entropy $H(X)$. Moreover, $H(X)$ is a quantified value and it could be calculated. So it is very suitable to be the metric to measure the difficulty of a Sudoku puzzle. The greater $H(X)$ is, the harder the puzzle will be.

Remark: In case $n_r(i,j,k) = 0$, then $M_r = 0$. That means the probability of the completion is 0, it follows that $H(X|Y_1,\cdots,Y_{r-1}) = 0$.

3.5 Review of the Algorithm

According to the information theory, the process of solving a problem is, in fact, to eliminate the uncertainty in the problem with an algorithm. In the sense, the so-called algorithm could be viewed as a series of operations which get the information together ceaselessly. So a good algorithm should involve as fewer operations as possible and meanwhile collect as much information as possible.

When turning back to the Sudoku creating algorithm, the optimization mainly relies on the improvement of the solving part, which largely determines the efficiency of the algorithm. As is shown in the first model, the solving part is realized in the way that all the probable digit groups are tried. In each attempt, the empty cells are filled one by one in some order. If the largest amount of information is obtained in each filling stepthe fewest steps will be needed. As a result, the complexity of the algorithm will be minimized.

3.6 How to Measure the Information Given by Each Step

When the step finishes the operation that fills an empty cell with a certain digit, it eliminates some uncertainty of the puzzle. The elimination of the uncertainty is defined as the information given about the Sudoku puzzle by the step. We denote it by W. It is made up of two parts. One is the mutual information between the operated cell and other empty cells, which is denoted by I_M . And the other part is about the decrease of the operated cell's entropy, denoted by ΔH . Owning to the complicated interactions between the cells, the information given by each step is approximately calculated with the formulas below.

Let Y still denote the empty cell to be filled in this step, and n_Y denote the number of candidates. Let $a(i,j,k)$ and $a'(i,j,k)$ denote the number of the available digits of another empty cell which is in the same row, column or subgird before and after the step. Then the amount of mutual information given by the step could be calculated by

$$I_M = \frac{1}{n_Y}\left(\sum_{i,j,k}\log_2 a(i,j,k) - \sum_{i,j,k}\log_2 a'(i,j,k)\right) \tag{15}$$

The decrease of the entropy of Y could be represented by

$$\Delta H(n_Y) = \log_2 n_Y - \log_2 (n_Y - 1) \tag{16}$$

Then the total information given by the step of filling the cell Y is obtained by

$$\begin{aligned} W(Y) &= I_M + \Delta H(n_Y) \\ &= \tfrac{1}{n_Y}\left(\sum_{i,j,k}\log_2 a(i,j,k) - \sum_{i,j,k}\log_2 a'(i,j,k)\right) + \log_2 n_Y - \log_2 (n_Y - 1) \end{aligned} \tag{17}$$

3.7 Description of the Algorithm

The framework of the algorithm is almost unchanged. In this algorithm, we measure the difficulty of the puzzle by its entire entropy instead of the number of empty cells in it. And the solving algorithm is guided by the steepest descent direction of the entropy, *i.e.*, at each step, the algorithm guarantee the information given about the Sudoku puzzle by the step defined above is maximized.

3.8 Algorithm of Solving a Sudoku Grid

Notations:

On the basis of definitions and notations in the former model, some notations are added below:

S	The stack used for storing the position of empty cells.
$push(S,i)$	The operation of pushing value i on the top of the stack S.
$pop(S,i)$	The operation of popping the value on the top of the stack S in i.

Remark:

For the program implementation, we modified the function $\Delta H(n)$ as below:

If $n > 1$, $\Delta H(n) = \log_2(n) - \log_2(n-1)$; $\Delta H(1)$ is defined to 10^6 (a number far larger than any other $\Delta H(n')$ when $n' > 1$). The value of the function when $n = 0$ is undefined.

Main Steps:

Step 1: Find the best empty cell b_i to fill so that the entire entropy of the Sudoku grid decreases most when the number of b_i's candidates decreases 1.

Step 2: Recursively search the solution by doing Step 1 to choose the best cell in each filling step.

Detailed steps:

In this section, the main steps are separated into several further steps(see **Fig. 6**).

(Here symbol i only means the position of current empty cell.)

- **Step 1:** Clear stack S.
- **Step 2:** Determine the empty cell set B of current Sudoku grid.
- **Step 3:** If $B = \emptyset$, output the current grid as a solution. If S is empty, go to Step 10; else $pop(S,i)$, backtrack (go to Step 6).
- **Step 4:** Determine V_k of each b_k so that each candidate in V_k won't cause b_k conflicting with other clues. If $V_k = \emptyset$ for any k then: if S is empty, $pop(S,i)$, backtrack (go to Step 5); else go to Step 10.
- **Step 5:** Find b_i so that the value of $W(b_i)$ is maximum.
- **Step 6:** For each v_{ij} in V_i , do Step 7 Step 8:

- **Step 7:** Fill b_i with v_{ij} so that b_i becomes a clue.
- **Step 8:** $push(S,i)$, do recursion (go to Step 2).
- **Step 9:** If S is not empty, $pop(S,i)$, backtrack (go to Step 5).
- **Step 10:** If the solution has not been found, the given grid has no solution.

4 Testing and Comparison

As shown above, our second algorithm based on the information theory optimizes the searching process by decreasing the entropy of entire Sudoku grid at the fastest speed. But how effective is it? Some tests have to be conducted next.

4.1 Testing of Our Second Algorithm

We generate several puzzles of varied difficulty and solve them. The difficulty of each of the puzzle and the total steps of recursion during solving process are shown in **Fig. 7**.

From **Fig. 7** we can see:

- Our metric of difficulty is appropriate, because the number of recursion steps increase alone with the difficulty.
- Our solving part is effective for the number of the recursion steps is very small, even when the difficulty of the puzzle is very high.

For further testing, we especially choose some puzzles from the Internet[9], which are acknowledged as extremely difficult. And our algorithm discussed above is applied to solving them. The result is: the range of the puzzles' difficulty is about 132 to 141, and the range of the number of the recursion steps by solving them is about 245 to 20192. It can be seen that our algorithm is still effective even when facing with quite difficult puzzles.

4.2 Comparison between Different Searching Strategies

A comparison will be made between the efficiencies of two algorithms which are in different orders of filling empty cells during searching: row by row and the order in the second model – our best theoretic order. It is expected the decreasing velocity of the entire entropy of the latter is faster than the former's, thus the latter is more efficient.

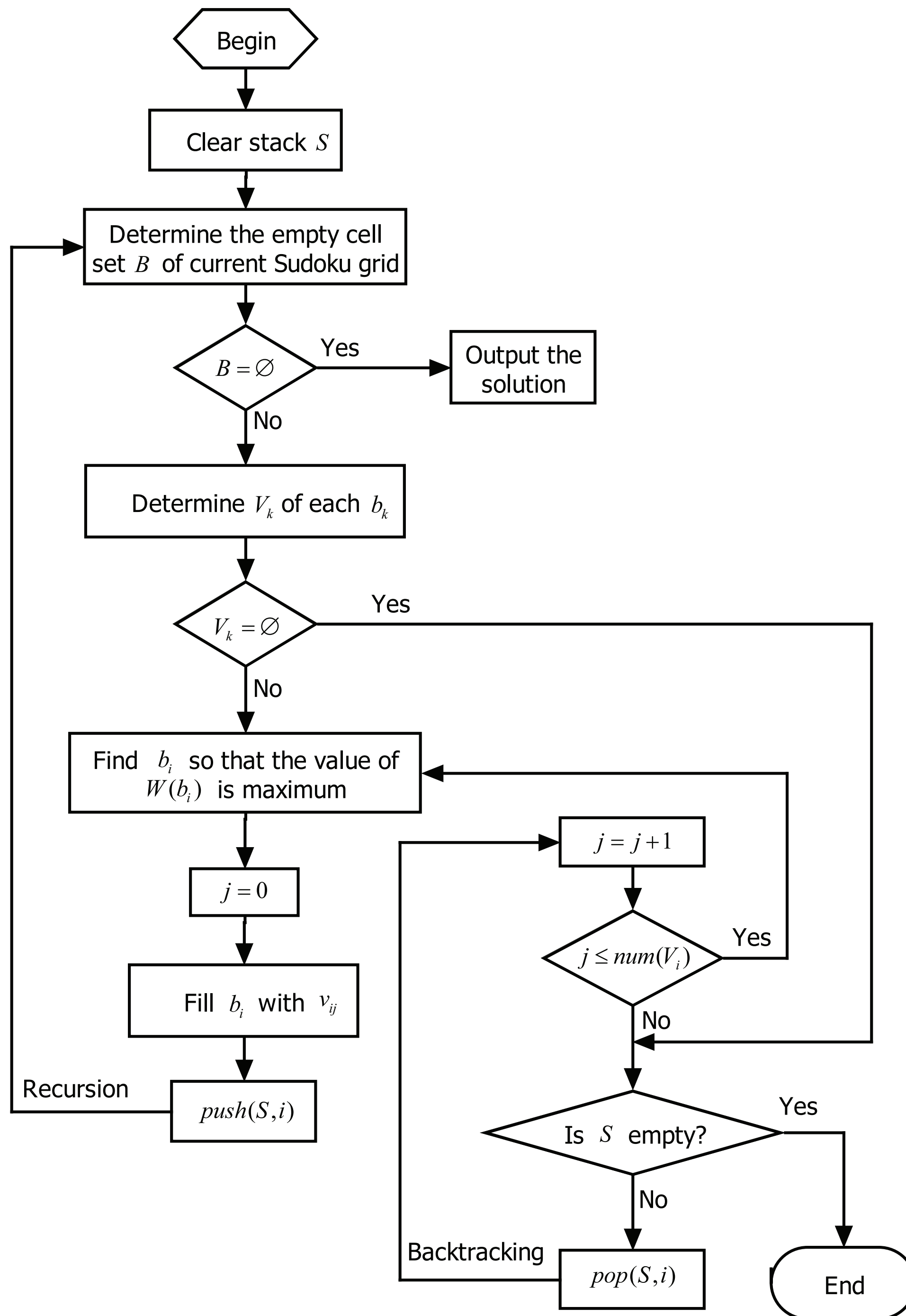

Fig 6: Flowchart of search algorithm optimized by information theory

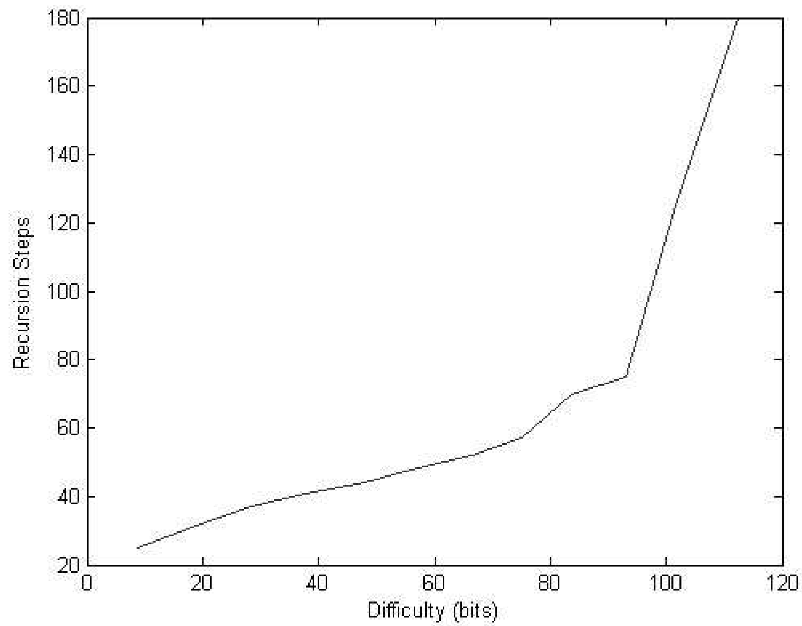

Fig 7: The relationship between total steps of recursion and difficulty level measured by entropy

We generate several puzzles of varied difficulty and solve them respectively using the two algorithms mentioned above. The result is listed in **Tab.1**:

It's obvious that the result above fits into our theory very well. The efficiency of the algorithm evidently increases when the filling order improves.

Tab 1: Total recursion steps comparison between two algorithms

Difficulty of puzzles (bits)	Recursion steps of row by row order	Recursion steps of the order in model 2
0-9.99	28	25
10∼19.99	41	31
20∼29.99	55	37
30∼39.99	89	41
40∼49.99	96	44
50∼59.99	223	47
60∼69.99	658	53
70∼79.99	1445	56
80∼89.99	2626	66
90∼99.99	8561	76
100∼109.99	24826	134
110∼119.99	>50000	251

5 Why Our Second Algorithm Is Effective

From results above we can conclude that our second algorithm is highly effective. It is considered that this algorithm takes good advantages of the information theory for optimization.

For optimizing an existing depth-first search algorithm, common method is to add varied appropriate pruning algorithm as much as possible. It is like cutting as many branches of a tree as possible to simplify the shape of the tree. But most pruning algorithm is complex to implement and the improvement of efficiency may not be very evident. In other words, even after a person tries hard to cut many a branch of a tree, the shape of it is still complex because it's too huge.

Our optimization algorithm is essentially different from what is mentioned above. Via improving the order of filling empty cells during searching, the number of recursion steps is efficiently reduced. That is to say, we simplify the shape of a huge tree by rearranging the tree's nodes instead of cutting any branch of it.

Fig. 8 shows the "decision tree" of the three different searching algorithms mentioned above:

It can be imagined that if our optimization method combines with pruning methods, the algorithm will surely be much more efficient.

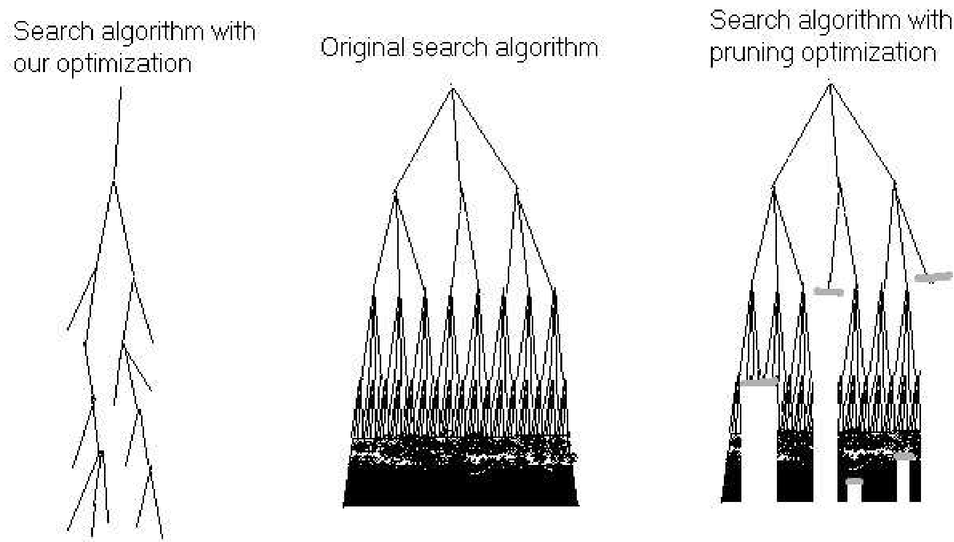

Fig 8: "Decision tree" of the three different searching algorithms

6 Conclusions

From the results of the testing and comparison, it can be concluded that the performance of the algorithm based on the information theory is excellent, both in dealing with the extremely difficult Sudoku puzzles, and in comparing the problem-solving efficiency with other algorithm. In terms of determining the difficulty levels of a Sudoku puzzle, the entropy is in accordance with the number of the recursion steps needed for solving it. They both confirm that the information theory is very suitable to analyze and address the issue of Sudoku puzzles.

Based on the information theory, if we view a problem as an unknown system, then the function of an algorithm will be collecting information of the system. In other words, it eliminates the uncertainty of the system ceaselessly by each step. Therefore, to optimize an existing algorithm, like the one in the first model, could be realized by accelerating the elimination of the entropy as much as possible. Maybe for some problems, developing an exact formula to calculate the entropy is not easy, but we still firmly believe that the information theory has indicated a new direction for universal optimization algorithms.

7 Algorithm Illustration Scheme

As analyzed above, we develop an algorithm to construct Sudoku puzzles of varying difficulty levels with the reasonable metric of entropy. The algorithm with 6 difficulty levels is listed in **Tab.2**:

Tab 2: Six difficulty levels illustrated

Difficulty Levels	Level Name	Value of Entropy (bits)
Level 1	Easy	0∼21
Level 2	Medium	21∼42
Level 3	Challenging	42∼63
Level 4	Difficult	63∼84
Level 5	Extremely Difficult	84∼105
Level 6	Evil	>105

The six examples of created Sudoku puzzles are attached in **Appendix A**.

8 Strengths and Weaknesses

8.1 Strengths

- **Scientific angle of view.** We manage to reflect the essence of the Sudoku with a scientific angle of view by information theory.
- **Reasonable quantified difficulty levels.** We define the difficulty levels by entropy. This metric is scientific, avoiding the influence of people's subjective factors and the random factors.
- **High searching efficiency.** Our algorithm enjoys many advantages of the information formula, which guides the way of searching in an effective direction and decreases the complexity of the algorithm.
- **Extensibility.** Slightly modified, our algorithm can be easily extended to construct lots of Sudoku puzzles with different sizes.

8.2 Weaknesses

- **Lower efficiency when no solution.** Our algorithm works worse when there is no solution to a puzzle. Because the quantified function we've developed is just an approximate description of the change of entropy and not precise enough. The errors will increase with the algorithm meeting a puzzle with no solution.

9 Future Attempt

In our model, there is some simplification in the process of analyzing the Sudoku puzzle with the information theory. In fact, one cell in a Sudoku grid is associated with any other through the rows, columns and subgrids. If the 9 rows, 9 columns and 9 subgrids are viewed as 27 variables, maybe we could use functions of the 27 variables to represent the amounts of the uncertainty and the information in a puzzle. And the condition under which the Sudoku puzzle has the unique solution may be found out with the function. But we can not analyze the subject further this time because of the limits in time. It is of great pleasure for us to reflect the character of the Sudoku puzzle from the perspective of information theory, and we will continue to work hard to explore it!

References

[1] Wikipedia: *Sudoku*. Available via WWW:
http://en.wikipedia.org/wiki/Sudoku (cited Feb.18 2008)

[2] JEAN-PAUL DELAHAYE: *The Science behind Sudoku.* Scientific American, June 2006

[3] *Minimum Sudoku.* Available via WWW:
http://people.csse.uwa.edu.au/gordon/sudokumin.php (cited Feb.18 2008)

[4] Agnes M. Herzberg and M. Ram Murty: *Sudoku Squares and Chromatic Polynomials.* Notice of the AMS, Volume 54, Number 6

[5] Takayuki YATO and Takahiro SETA: *Complexity and Completeness of Finding Another Solution and Its Application to Puzzles.* Available via WWW:
www-imai.is.s.u-tokyo.ac.jp/ yato/data2/SIGAL87-2.pdf (cited Feb.18 2008)

[6] Wikipedia: *Information theory.* Available via WWW:

`http://en.wikipedia.org/wiki/Information_theory`

(cited Feb.18 2008)

[7] Wikipedia: *Entropy.* Available via WWW:

`http://en.wikipedia.org/wiki/Entropy`

(cited Feb.18 2008)

[8] *Top 10 most difficult in DB.* Available via WWW:

`http://www.menneske.no/sudoku/eng/top10.html`

(cited Feb.18 2008)

[9] Wikipedia: *Algorithmics of Sudoku.* Available via WWW:

`http://en.wikipedia.org/wiki/Algorithmics_of_Sudoku`

(cited Feb.18 2008)

[10] Wikipedia: *Mathematics of Sudoku.* Available via WWW:

http://en.wikipedia.org/wiki/Mathematics_of_Sudoku

(cited Feb.18 2008)

[11] Wikipedia: *Quantities of information.* Available via WWW:

http://en.wikipedia.org/wiki/Quantities_of_information

(cited Feb.18 2008)

[12] Wikipedia: *Depth-first search.* Available via WWW:

http://en.wikipedia.org/wiki/Depth-first_search

(cited Feb.18 2008)

[13] *Sudoku Puzzle Theory.* Available via WWW:

http://www.sudokudragon.com/sudokutheory.htm

(cited Feb.18 2008)

[14] LAUREN AARONSON: *Sudoku Science.* IEEE Spectrum, February 2006, NA

[15] Bertram Felgenhauer and Frazer Jarvis: *Enumerating possible Sudoku grids.* Available via WWW:

http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf

(cited Feb.18 2008)

[16] Laura Taalman: *Taking Sudoku Seriously.* Available via WWW:

www.math.jmu.edu/~taal/sudoku_mathhorizons.pdf

(cited Feb.18 2008)

[17] Bastian Michel: *Mathematics of NRC-Sudoku (December 5,2007).* Available via WWW:

`www.math.ruu.nl/people/vdkallen/webfiles/sudoku.pdf`

(cited Feb.18 2008)

[18] Rhyd Lewis: *Metaheuristics can Solve Sudoku Puzzles.* Available via WWW:

`www.cardiff.ac.uk/carbs/faculty/lewisr9/META_CAN_SOLVE_SUDOKU.pdf`

(cited Feb.18 2008)

[19] *Timo Mantere and Janne Koljonen: Solving and Rating Sudoku Puzzles with Genetic Algorithms.* New Developments in Artificial Intelligence and the Semantic Web Proceedings of the 12th Finnish Artificial Intelligence Conference STeP 2006.

[20] Timo Mantere and Janne Koljonen: *Solving, Rating and Generating Sudoku Puzzles with GA.* 2007 IEEE Congress on Evolutionary Computation (CEC 2007)

A Sudoku puzzles of Six difficulty levels illustrated

Here are six examples of created Sudoku puzzles:

Level 1

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

Level 2

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

Level 3

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

Level 4

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

Level 5

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

Level 6

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

B Algorithm programmes based on our model

Here are algorithm programmes based on our model as follow.

Input C++ source:

```
#include <stdlib.h>
#include <memory.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <float.h>

//#define RND_ORDER
#define ENT_ORDER

//#define RECORDORDER

#define LEVEL_NUM 6

const float LEVEL_ENT[LEVEL_NUM+1] = { 0.0, 21.0, 42.0, 63.0, 84.0, 105.0, FLT_MAX };

typedef struct
{
        int cho[9];
        int chonum;
} CHO, *LPCHO;

class Sudoku
{
public:
        void Init(void);
        int Solve( int sud[], int solution[] );// solve a sudoku
        bool Gen( int sud[], int level );// generate a sudoku of specified level

        void setforcecut( bool bforce )
        {
                bforcecut = bforce;
        }

        // for testing
        void Gen( int sud[], float difficulty );
        int GetSteps(void)
        {
                return step;
        }

        static int MAX_STEP;

private:
        float CalcDifficulty( int sud[] );
        bool input( int val[] );
```

```
        bool updateorder( int start );
        int solve( int p );

        int *x;// sudoku puzzle
        int solnum;// solution number

        int *sol;// buffer to store solution

        int z[81];// store index of empty cells
        int znum;// number of empty cells

        CHO cho[81];

        // used flags
        int user[9][10];
        int usec[9][10];
        int useb[9][10];

        // row, col, block index, constants
        int row[81];
        int col[81];
        int block[81];

        float entdec[81];

        int count[81];
        int step;
        bool bforcecut;

#ifdef RECORDORDER
        int order[81];
#endif

        static float lb[10]; // log2
        static float lbd[10]; // log2(i) - log2(i-1)
};

int Sudoku::MAX_STEP = 5000;
float Sudoku::lb[10];
float Sudoku::lbd[10];

bool Sudoku::Gen( int sud[], int level )
{
        static int sol[81];
        int P[81];
        float diff;
        int s, res, a;
        int i;

        if( level < 0 || level >= LEVEL_NUM )
                {
                return false;
```

```
          }//end if

    for( ;; )
    {
    x = sud;

    do        {
          memset( sud, 0, sizeof( int[81] ) );
          do        {
                do        {
                      i = rand()%81;
                      } while( sud[i] != 0 );
                sud[i] = 1 + rand()%9;

                res = Solve( sud, NULL );
                // add a random cell and solve it. No solution ==> remove it
                      again.
                // Not yet a unique solution ==> continue adding cells
                if( res < 1 )
                      sud[i]=0;

                } while( res != 1 );

          } while( Solve( sud, NULL ) != 1 );

    //return;

    Solve( sud, sol );

    diff = CalcDifficulty( sud );
    if( diff >= LEVEL_ENT[level] && diff < LEVEL_ENT[level+1] )
          {
          return true;
          }
    else{
          P[0] = P[1] = 0;
          for(i = 1; i < 81; i++)
                {
                a = rand() % i;
                P[i] = P[a];
                P[a] = i;
                }//end for
          if( diff >= LEVEL_ENT[level + 1] ) // add cells
                {
                for( i = 0; i < 81; i++ )
                      {
                      a = P[i];
                      if( sud[a] == 0 )
                            {
                            sud[a] = sol[a];
                            diff = CalcDifficulty( sud );
                            if( diff >= LEVEL_ENT[level] && diff <
```

```
                                LEVEL_ENT[level+1] )
                                  {
                                  return true;
                                  }
                              else if( diff < LEVEL_ENT[level] )
                                  {
                                  sud[a] = 0;
                                  }//end if
                              }//end if
                          }//end for
                      }
                  else{// remove cells
                      for( i = 0; i < 81; i++ )
                          {
                          a = P[i];
                          s = sud[a];
                          if( s != 0 )
                              {
                              sud[a] = 0;
                              if( Solve( sud, NULL ) != 1 )
                                  {
                                  sud[a] = s;
                                  }
                              else{
                                  diff = CalcDifficulty( sud );
                                  if( diff >= LEVEL_ENT[level] && diff
                                      < LEVEL_ENT[level+1] )
                                      {
                                      return true;
                                      }
                                  else if( diff >= LEVEL_ENT[level+1] )
                                      {
                                      sud[a] = s;
                                      }//end if
                                  }//end if
                              }//end if
                          }//end for
                      }//end if
                  }//end if
              }//end loop

              return true;
    }//end Sudoku::Gen

    // for testing
    void Sudoku::Gen( int sud[], float difficulty )
    {
              static int sol[81];
              int P[81];
              float diff;
              int s, res, a;
              int i;
```

```
for( ;; )
{
x = sud;

do      {
        memset( sud, 0, sizeof( int[81] ) );
        do      {
                do      {
                        i = rand()%81;
                        } while( sud[i] != 0 );
                sud[i] = 1 + rand()%9;

                res = Solve( sud, NULL );
                // add a random cell and solve it. No solution ==> remove it
                     again.
                // Not yet a unique solution ==> continue adding cells
                if( res < 1 )
                        sud[i]=0;

                } while( res != 1 );

        } while( Solve( sud, NULL ) != 1 );

//return;

Solve( sud, sol );

diff = CalcDifficulty( sud );
if( diff > difficulty - 5 && diff < difficulty + 5 )
        {
        return;
        }
else{
        P[0] = P[1] = 0;
        for(i = 1; i < 81; i++)
                {
                a = rand() % i;
                P[i] = P[a];
                P[a] = i;
                }//end for
        if( diff > difficulty + 5 ) // add cells
                {
                for( i = 0; i < 81; i++ )
                        {
                        a = P[i];
                        if( sud[a] == 0 )
                                {
                                sud[a] = sol[a];
                                diff = CalcDifficulty( sud );
                                if( diff > difficulty - 5 && diff < difficulty +
                                     5 )
```

```
                                        {
                                        return;
                                        }
                                else if( diff < difficulty - 5 )
                                        {
                                        sud[a] = 0;
                                        }//end if
                                }//end if
                        }//end for
                }
        else{// remove cells
                for( i = 0; i < 81; i++ )
                        {
                        a = P[i];
                        s = sud[a];
                        if( s != 0 )
                                {
                                sud[a] = 0;
                                if( Solve( sud, NULL ) != 1 )
                                        {
                                        sud[a] = s;
                                        }
                                else{
                                        diff = CalcDifficulty( sud );
                                        if( diff > difficulty - 5 && diff <
                                            difficulty + 5 )
                                                {
                                                return;
                                                }
                                        else if( diff > difficulty + 5 )
                                                {
                                                sud[a] = s;
                                                }//end if
                                        }//end if
                                }//end if
                        }//end for
                }//end if
        }//end if
}//end loop

}//end Sudoku::Gen

float Sudoku::CalcDifficulty( int sud[] )
{
        float diff;
        int chonum;
        int i, j, k, m;

        input( sud );

        diff = 0;
        for( i=0, k=0; i<9; ++i )
```

```
                {
                for( j=0; j<9; ++j,++k )
                        {
                        if( x[k] == 0 )
                                {
                                chonum = 0;
                                for( m = 1; m <= 9; ++m )
                                        {
                                        if( !( user[i][m] || usec[j][m] || useb[block[k
                                              ]][m] ) )
                                                {
                                                ++chonum;
                                                }//end if
                                        }//end for
                                diff += lb[chonum];
                                }//end if
                        }//end for
                }//end for

        return diff;
}//end Sudoku::CalcDifficulty

void Sudoku::Init(void)
{
        int i, j, r, c;

        srand( time( NULL ) );
        for( i = 0; i < 9; ++i )
                {
                for( j = 0; j < 9; ++j )
                        {
                        row[i*9+j] = i;
                        col[i*9+j] = j;
                        }//end for
                }//end for

        for( i = 0; i < 3; ++i )
                {
                for( j = 0; j < 3; ++j )
                        {
                        for( r = 0; r < 3; ++r )
                                {
                                for( c = 0; c < 3; ++c )
                                        {
                                        block[ ( i * 3 + r ) * 9 + j * 3 + c ] = i * 3 +
                                              j;
                                        }//end for
                                }//end for
                        }//end for
                }//end for

        for( i = 1; i <= 9; ++i )
```

```
                {
                lb[i] = log( (float)i ) / log( 2.0f );
                }//end for
        for( i = 9; i > 1; --i)
                {
                lbd[i] = lb[i] - lb[i-1];
                }//end for
        lbd[1] = 1e6;

        bforcecut = true;
}//end Sudoku::init

int Sudoku::solve( int p )
{
        int r,c,b;
        int pos;
        int t;
        int i;

        ++count[p];
        if( bforcecut )
                {
                ++step;
                }//end if
        if( step >= MAX_STEP )
                {
                if( solnum == 1 )
                        {
                        solnum = 2;
                        }//end if
                return 0;
                }//end if

#ifdef ENT_ORDER
        if( !updateorder( p ) )
                {
                return false;
                }//end if
#endif // ENT_ORDER

#ifndef RND_ORDER
        pos = z[p];
#else
        do      {
                i = rand()%81;
                } while( x[i] != 0 );
        z[p] = pos = i;
#endif

#ifdef RECORDORDER
        order[pos] = p + 1;
#endif
```

```
#ifdef ENT_ORDER
        for( i = 0, t = x[pos] = cho[pos].cho[i]; i < cho[pos].chonum; ++i, t = x[pos] =
            cho[pos].cho[i] )
#else
        for( t = x[pos]=1; x[pos]<=9 && solnum < 2 && step < MAX_STEP; ++x[pos],
            ++t )
#endif
                {
                r=row[pos];
                c=col[pos];
                b=block[pos];

#ifndef ENT_ORDER
                if( user[r][t] || usec[c][t] || useb[b][t] ) continue;
#endif

                user[r][t]=1;
                usec[c][t]=1;
                useb[b][t]=1;

                if( p+1 < znum )
                        {
                        solve( p+1 );
                        if( solnum > 1 || ( solnum == 1 && sol != NULL ) )
                                {
                                return solnum;
                                }//end if
                        }
                else{
                        ++solnum;
                        if( solnum == 1 )
                                {
                                if( sol != NULL )
                                        {
                                        memcpy( sol, x, sizeof( int[81] ) );
                                        return 1;
                                        }//end if
                                }
                        else{// solnum > 1
                                return 2;
                                }//end if
                        }//end if

                user[r][t]=0;
                usec[c][t]=0;
                useb[b][t]=0;
                }//end for
        x[pos] = 0;

        return solnum;
}//end Sudoku::solve
```

```
bool Sudoku::updateorder( int start )
{
        static float ent[81];
        float maxe;
        int pos;
        int i, j, k, m;
        int p, r, c;
        int upper, upper2;

        memset( ent, 0, sizeof( ent ) );
        memset( entdec, 0, sizeof( entdec ) );
        for( i=0, k=0; i<9; ++i )
                {
                for( j=0; j<9; ++j,++k )
                        {
                        if (x[k]==0)
                                {
                                cho[k].chonum = 0;
                                for( m = 1; m <= 9; ++m )
                                        {
                                        if( !(user[i][m] || usec[j][m] || useb[block[k]][
                                            m]) )
                                                {
                                                cho[k].cho[ cho[k].chonum++ ] = m;
                                                }//end if
                                        }//end for
                                if( cho[k].chonum == 0 )
                                        {
                                        return false;
                                        }//end if

                                ent[k] = lbd[cho[k].chonum];
                                for( p = j; p < 81; p += 9 )
                                        {
                                        if( p != k )
                                                {
                                                entdec[p] += ent[k];
                                                }//end if
                                        }//end for
                                for( p = i * 9, upper = p + 9; p < upper; ++p )
                                        {
                                        if( p != k )
                                                {
                                                entdec[p] += ent[k];
                                                }//end if
                                        }//end for
                                for( r = i - i % 3, upper = r + 3; r < upper; ++r )
                                        {
                                        for( c = j - j % 3, upper2 = c + 3; c < upper2;
                                            ++c )
                                                {
```

```
                                                if( r != i && c != j )
                                                        {
                                                        entdec[r * 9 + c] += ent[k];
                                                        }//end for
                                                }//end for
                                        }//end for

                                }//end if
                        }//end for
                }//end for

        maxe = -FLT_MAX;
        for( i=0, k=0; i<9; ++i )
                {
                for( j=0; j<9; ++j,++k )
                        {
                        if (x[k]==0)
                                {
                                entdec[k] /= cho[k].chonum;
                                entdec[k] += ent[k];
                                if( entdec[k] > maxe )
                                        {
                                        maxe = entdec[k];
                                        pos = k;
                                        }//end if
                                }//end if
                        }//end for
                }//end for

        z[start] = pos;

        return true;
}//end Sudoku::updateorder

bool Sudoku::input( int val[] )
{
        int i, j, k;

        x = val;

        znum=0;
        memset(user,0,sizeof(user));
        memset(usec,0,sizeof(usec));
        memset(useb,0,sizeof(useb));

        for( i=0, k=0; i<9; ++i )
                {
                for( j=0; j<9; ++j,++k )
                        {
                        if (x[k]==0)
                                {
                                z[znum++]=k;
```

```
                        }
                else{
                        if( user[i][x[k]] || usec[j][x[k]] || useb[block[k]][x[k
                            ]] )
                                {
                                return false;
                                }//end if
                        user[i][x[k]]=usec[j][x[k]]=useb[block[k]][x[k]]=1;
                        }//end if
                }//end for
        }//end for

    return true;
}//end Sudoku::input

int Sudoku::Solve( int val[], int solution[] )
{
    int ret;
    int i;

    if( !input( val ) )
        {
        return 0;
        }//end if
    sol = solution;

#ifdef RECORDORDER
    for( i = 0; i < 81; ++i )
        {
        printf( "%-3d", x[i] );
        if( i % 9 == 8 )
            {
            putchar( '\n' );
            }
        }//end for
    putchar( '\n' );
    memset( order, 0, sizeof( order ) );
#endif

    memset( count, 0, sizeof( count ) );
    step = 0;

    solnum = 0;
    ret = solve( 0 );

    for( i = 0; i < znum; ++i )
        {
        x[ z[i] ] = 0;
        }//end for

#ifdef RECORDORDER
    for( i = 0; i < 81; ++i )
```

```
            {
            printf( "%-3d", order[i] );
            if( i % 9 == 8 )
                {
                putchar( '\n' );
                }
            }//end for
        putchar( '\n' );
        putchar( '\n' );
#endif

        return ret;
}//end Sudoku::solve

Sudoku sudoku;
int sol[81];

int main()
{
        static int val[81];
        int j,k;

        sudoku.Init();

        Sudoku::MAX_STEP = 5000;
        sudoku.setforcecut( true );

        FILE* file = fopen( "sudoku.txt", "w" );
        for( k = 0; k < LEVEL_NUM; ++k )
            {
            sudoku.Gen( val, k );

            for( j = 0; j < 81; ++j )
                {
                putc( val[j] + '0', file );
                }//end for
            putc( '\n', file );
            putc( '\n', file );
            }//end for

        fclose( file );
        return 0;
}
```