International Conference on Computational Intelligence: Modeling Techniques and Applications (CIMTA) 2013

An Algorithm for Generating only Desired Permutations for Solving Sudoku Puzzle

Arnab Kumar Maji\textsuperscript{a}\textsuperscript{*}, Sunanda Jana\textsuperscript{b}, Rajat Kumar Pal\textsuperscript{c}

\textsuperscript{a} Department of Information Technology, North Eastern Hill University, Shillong 793 022, Meghalaya, India
\textsuperscript{b} Department of Computer Science and Engineering, Haldia Institute of Technology, Haldia 721 657, West Bengal, India
\textsuperscript{c} Department of Computer Science and Engineering, University of Calcutta, Kolkata 700 009, West Bengal, India

Abstract

‘Sudoku’ is the Japanese abbreviation of a longer phrase, ‘Suji wa dokushin ni kagiru’, meaning ‘the digits must remain single’. It is a challenging numeric puzzle that trains our logical mind. Solving a Sudoku puzzle requires no math, not even arithmetic. Even so, the game poses a number of intriguing mathematical problems. In this paper, a novel technique is proposed to generate only the desired permutations among minigrids that can be used to solve a Sudoku puzzle in an efficient manner.

© 2013 The Authors. Published by Elsevier Ltd. Selection and peer-review under responsibility of the University of Kalyani, Department of Computer Science & Engineering

Keywords: Sudoku Puzzle; Minigrid; Permutation; Permutation tree.

1. Introduction

1.1 A Brief Background

A Sudoku puzzle is a grid of $n$ rows and $n$ columns, in which some pre-assigned clues or givens have been entered. The size of the grid can be $n \times n$, where $n$ is an integer. The most common size of such a (square) grid is $9 \times 9$. Besides the standard $9 \times 9$ grid, variants of Sudoku puzzles include the following:

- $4 \times 4$ grid with $2 \times 2$ minigrids,
- $5 \times 5$ grid with pentomino regions published under the name Logi-5 [1]. A pentomino is composed of five congruent squares, connected orthogonally. Pentomino is seen in playing the game Tetris [2],
- $16 \times 16$ grid (super Sudoku) [10],

* Corresponding author. Tel.: +913212210165
E-mail address: arnab.maji@gmail.com.
• 25×25 grid (Sudoku, the Giant) [10], etc.

There are quite a few logic techniques that researchers use to solve this problem. Some are basic simple logic, some are more advanced. Depending on the difficulty of the puzzle, a blend of techniques may be needed in order to solve a puzzle. In fact, most computer generated Sudoku puzzles rank the difficulty based upon the number of empty cells in the puzzle and how much effort is needed to solve each of them. Table 1 shows a comparison chart of the number of clues for different difficulty levels [15]. However, position of each of the empty cells also affects the level of difficulty.

In our proposed method, we have divided a 9×9 grid into nine 3×3 minigrids as shown in Figure 3(c). Then we have generated permutations among minigrids based on given clues. Then based on valid permutation amongst minigrids, the final solution of the Sudoku puzzle has been generated, if the given puzzle has a solution.

| Difficulty level   | Number of clues |
|--------------------|-----------------|
| 1 (Extremely Easy) | More than 46    |
| 2 (Easy)           | 36-46           |
| 3 (Medium)         | 32-35           |
| 4 (Difficult)      | 28-31           |
| 5 (Evil)           | 17-27           |

1.2 Outline of the Paper

The remaining part of the paper is organized as follows. A brief literature survey is introduced in Section 2. A permutation generation based algorithm for solving a given Sudoku instance has been developed in Section 3. The importance for solving a given Sudoku instance has been briefly discussed in Section 4 in the form of applications, and the paper is concluded in Section 5 with few necessary remarks.

2. Literature Survey

Permutation is a way of ordering of a set of objects or symbols where each object occurs exactly once. As an example, there are six permutations of the set of numbers {1,2,3}, namely (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), and (3,2,1). The number of permutations of \(n\) distinct objects can be calculated as \(n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1\), which is mainly denoted as “factorial of \(n\)” and written as “\(n!\)”. In computing, it may be required to generate permutations of a given sequence of values. The notion of permutation is used in the following contexts: (1) Group Theory, (2) Combinatorics, etc. In the existing literature, there are several permutation generation techniques, some of which are described in the following subsections.

2.1 Random Permutation Generation Algorithm

We can generate a random permutation [3] of \(k\) distinct objects in a computer, to order a given list randomly. Permutation distribution is supposed to be unbiased so that each of the \(k!\) permutations are generated with equal probability. Here a permutation has generated from a list of \(k\) distinct random numbers \(a_1, a_2, \ldots, a_k\), where \(1 \leq i \leq k\). We can perform random permutation from this list by moving each number \(a_i\) to position \(i\) [3].

The naive way of generating such a list is as follows:
1. Generate a trial random number \(r\) from 1 to \(k\).
2. Check whether \(r\) is already in the list. If yes, then go back to Step 1.
3. Else, add \(r\) to the list.
4. If fewer than \(k\) numbers have been added to the list, go back to Step 1.

But the main drawback of this algorithm is that the algorithm is not efficient. The time required to check whether each trial number is already in the list is of order \(k\). Again, if nearly \(k\) numbers have already been added, then the number of trial random numbers required, before one is found that is not already in the list, is also of order \(k\). Thus
O(k²) is the overall running time of this algorithm.

2.2 Fisher-Yates Shuffle

There is another popular way of generating a random permutation which is called Fisher-Yates shuffle [5]. It is an iterative algorithm used for randomising a set of numbers. It is a paper and pen based method. Steps of this algorithm are as follows:

1. Write down the numbers from 1 to \( N \) in scratch.
2. Pick a random number \( k \) between one and the number of unstuck numbers remaining in the scratch. Store the value of \( k \) in Roll.
3. Counting from the low end, select and strike out the \( k \)th number from the scratch, if not yet struck out, and write it down elsewhere.
4. Repeat from Step 2 until all the numbers have been struck out.
5. The sequence of numbers written down in Step 3 is now a random permutation of the original numbers.

In 1964, a modern version of the Fisher-Yates shuffle, designed for computer use, was proposed by Durstenfeld [12]. The algorithm described by Durstenfeld differs from that given by Fisher and Yates in a small but significant way. This algorithm shuffles an array of any type.

Computationally and mathematically this is a truly elegant algorithm. At each iteration \( i \), it chooses a random element \( \pi[j] \) from the unshuffled set \( \pi[0, 1, \ldots, n-1] \), and interchanges it with \( \pi[j] \). Thus, the time and space complexity of Durstenfeld’s algorithm is \( O(n) \), which is optimal.

2.3 Permutation Tree

If \( S_n \) is the set of permutations of set \( I = \{1, 2, \ldots, n\} \) (i.e., the set of first \( n \) natural numbers), then every permutation in \( S_n \) can be represented by a tree called permutation tree. So, we can also say that a permutation tree must correspond to a permutation. Extensive work has been done in the past to represent and generate permutations with the help of such trees.

As there are different techniques for generating permutations available in literature, so for different types of such techniques different permutation trees can be constructed [8]. Some of them have been described below.

Fig. 1. The permutation tree for three natural numbers 1, 2, and 3 following the algorithm of Arnow [6].

Fig. 2. The permutation tree for three distinct colours following the algorithm of Latif [4].

In 1989, Arnow developed an attractive method to generate the random permutations of \( n \) elements [6]. He has inserted \( n \) elements in \( n \) successive levels of the tree, and for an integer \( k \leq n \), all permutations of the first \( k \) elements are obtained at the \( k \)th level of the tree. More specifically, the root of the tree represents the only permutation comprising the first element. All other successive levels have been formed by inserting the successive elements in all positions (extreme and in between) one after another (as child nodes of a smaller parent permutation). Hence for each of the nodes at the \( k \)th level must have exactly \( k+1 \) child nodes. As an example, the tree structure for \( n = 3 \) has been shown in Fig. 1, where the elements are 1, 2, and 3. Here 1 is the root of the tree. Then the child nodes have been formed by inserting 2 before 1 and 2 after 1. Similarly, the third level is formed by inserting 3 in three possible
positions, and we obtain all six permutations comprising the elements 1, 2, and 3.

In 2004, Latif developed another way of constructing permutation tree [4]. He used a random permutation technique for generating the permutations and by considering these permutations he constructs a permutation tree model. In his proposed structure the edges are represented by the elements to be permuted and he used different colours for each of them. Following an edge in the permutation tree is equivalent to selecting an item, and permutations are formed by following paths from the root to the leaves. Fig. 2 shows a sample tree structure for three distinct elements represented by three different colours.

A permutation tree of $n$ items is generated recursively. At the root of the tree, all $n$ items are available for selection, so the root has $n$ edges of different colours leading to $n$ subtrees. At each of the subtrees of the root has only $n-1$ items available (as one item has already been selected), so each of the subtrees of the root has $n-1$ edges leading to respective leaf vertices. The colour of a missing edge corresponds to the item that has been selected now following the path from the root of the tree to the root of the subtree (see Fig. 2). This process is repeated with the subtrees of the subtrees, and so on, until the leaves are reached. At this point all items have been selected; therefore, no edges remaining to be considered and to be continued the process further. This is how a path from the root of the tree to a leaf node provides a permutation of the elements under consideration.

In our proposed work we have developed a novel permutation tree to generate only valid permutations for each of the succeeding minigrids of a given Sudoku puzzle, and solve it subsequently, if it has a solution. As the problem of solving a Sudoku puzzle is NP-hard, it is less likely to build up a polynomial time algorithm for solving the same [13]. Anyway, a heuristic based deterministic algorithm has been developed in this paper that can successively solve a given Sudoku instance of size $9 \times 9$, if it has a solution, in reasonable amount of time.

### 3. The Proposed Algorithm

A Sudoku is usually a $9 \times 9$ grid based puzzle problem which is subdivided into nine $3 \times 3$ minigrids, wherein some clues are given and the objective of the problem is to fill it up for the remaining blank positions. Furthermore, the objective of this problem is to compute a solution where the numbers 1 through 9 will occur exactly once in each row, exactly once in each column, and exactly once in each minigrid independently obeying the given clues. One such problem instance is shown in Fig. 3(a) and its solution is shown in Fig. 3(b).

![Fig. 3. (a) An instance of the Sudoku problem. (b) A solution of the Sudoku instance shown in Fig. 3(a). (c) The structure of a 9×9 Sudoku puzzle (problem) with its nine minigrids of size 3×3 each as numbered 1 through 9.](image)

The proposed algorithm considers each of the minigrids that may be identified as 1 through 9 as shown in Fig. 3(c). Each minigrid may or may not have some clues as numbers that are given. We first consider a minigrid that contains a maximum number of clues, and if there are two or more such minigrids, we consider the one with the least minigrid number.

Needless to mention that each of the cells in a minigrid, either containing a clue or a blank cell, is somehow differentiated from each of the cells of another minigrid as the position of a cell in a Sudoku instance could be specified by its row number and column number, which is unique. So, a cell $[i, j]$ of minigrid $k$ may either contain a number $l$ as a given clue or a blank location that is to be filled in by inserting a number $m$, where $1 \leq i, j, k, l, m \leq 9$. 

![Fig. 3. (c) The structure of a 9×9 Sudoku puzzle (problem) with its nine minigrids of size 3×3 each as numbered 1 through 9.](image)
Now to start with a minigrid as stated above, we find that minigrid 3 contains a maximum number of clues, i.e., 4, among all the minigrids, and each of the minigrids 1 and 2 contains less number of clues than that of minigrid 3 (see Fig. 3(a)). For example, for the Sudoku instance as shown in Fig. 3(a), each of the minigrids 3, 5, and 7 contains four clues each; hence, at the beginning, we consider minigrid 3 for computing all its valid permutations of the missing numbers for its blank locations (as 3 is the minimum minigrid number).

Here we denote a cell location of a Sudoku instance by [row number, column number], where each of row number and column number varies from 1 to 9. Hence the blank locations are [1, 7], [2, 7], [2, 9], [3, 7], and [3, 8], and the missing digits are 4, 5, 7, 8, and 9.

We compute all possible permutations of these missing digits in minigrid 3, where the first permutation may be 98754 (the maximum number) and the last permutation may be 45789 (the minimum number) using the missing digits. Here as the number of blank locations is five, the total number of permutations is 5!, which is equal to 120. Now the algorithm considers each of these permutations one after another and identifies only the valid set of permutations based on the given clues available in rows and columns in other minigrids (that are minigrids 1, 2, 6, and 9). As for example, if we consider the last permutation 45789 and place the missing digits, respectively, in order in locations [1, 7], [2, 7], [2, 9], [3, 7], and [3, 8], which are arranged in ascending order, we find that this permutation is not a valid permutation. This is because location [4, 9] already contains 7 as a clue of minigrid 6, so we cannot place 7 at [2, 9] as the permutation suggests. Also we cannot place 9 at location [1, 7] as location [6, 7] contains 9 as a clue of minigrid 6; hence permutation 98754 is also not a valid permutation.

Similarly, we may find that there are many of the permutations that are not valid permutations. To compute only the valid permutations for the missing digits in minigrid 3, we construct a tree structure as shown in Fig. 4. Here the missing digits are 9, 8, 7, 5, and 4. The proposed algorithm likes to place each of the permutations of these missing digits in the blank locations [1, 7], [2, 7], [2, 9], [3, 7], and [3, 8]. Naturally as the root of the tree structure does not contain any permutation of the missing five digits, it is represented by ‘*****’. This root is having five children where the first child leads to generate all valid permutations staring with 9, the second child leads to generate all valid permutations staring with 8, and so on.

Now note that none of the permutations starting with 9 is a valid permutation as column 7 of minigrid 6 contains 9 as given clue (at location [6, 7]). So, we do not expand this vertex (i.e., vertex with permutation ‘9****’) further in order to compute only the set of valid permutations. Similarly, we do not expand the child vertex with permutation ‘5****’, as location [1, 3] contains 5 as given clue. Up to this point in time, as either 8, or 7, or 4 could be placed at [1, 7], we expand each of the child vertices starting with permutations 8, and 7, and 4, as shown in Fig. 4.

Similarly, we expand the tree structure inserting a new missing number at its respective location (for a blank cell) leading from a valid permutation (as vertex) in the previous level of the tree. Correspondingly, we verify whether the missing digit could be placed at the particular location for a blank cell of the given Sudoku instance P. If the answer is ‘yes’, we further expand the vertex; otherwise, we stop expanding the vertex in some earlier level of the tree structure prior to the last level of valid leaf vertices only. As for example, the vertex with permutation ‘479**’ is not expandable, because we cannot place 9 at [2, 9] as [2, 1] contains 9 as given clue. So, this is how either a valid permutation is generated from the root of the tree structure reaching to a bottommost leaf vertex, or the process of expansion is terminated in some earlier level of the tree that must generate other than valid (or unwanted) permutations at this point in time.

Interestingly, Fig. 4 shows the reality that the number of possible permutations of five missing digits is 120, and out of them only seven are valid for minigrid 3 of the Sudoku instance shown in Fig. 3(a). Note that the given clues in P are nothing but constraints and we are supposed to obey each of them. So, usually, if there are more clues, P is more constrained and hence the number of valid permutations is even much less, and the solution, if it exists, is unique in most of the cases. On the contrary, if there are fewer clues in P, more valid permutations for some minigrid of P could be generated; computation of a solution for P might take more time. In any case, if there is a solution of the assumed Sudoku instance (in Fig. 3(a)), out of these seven valid permutations only one will finally be accepted following the subsequent steps of the algorithm.
To find out the next minigrid to be considered, we go through the row and column minigrids of minigrid 3 in the Sudoku instance of Fig. 3(a) (that are minigrids 1, 2, 6, and 9), and among these minigrids we find that minigrid 1 contains a maximum number of clues, i.e., 3 (which is equally true for each of the minigrids 6 and 9), and its minigrid number is minimum. Now the algorithm considers one valid permutation (out of the seven permutations) of minigrid 3 and all given clues in P, and generates all valid permutations for minigrid 1. If at least one valid permutation for minigrid 1 is obtained, we proceed for generating all valid permutations for minigrid 7 obeying all given clues in P and the assumed valid permutations of minigrids 3 and 1; otherwise, a second valid permutation of minigrid 3 is considered, for which in a similar way, we generate all valid permutations for minigrid 1, and so on.

Interestingly, Fig. 4 shows the reality that the number of possible permutations of five missing digits is 120, and out of them only seven are valid for minigrid 3 of the Sudoku instance shown in Fig. 3(a). Note that the given clues in P are nothing but constraints and we are supposed to obey each of them. So, usually, if there are more clues, P is more constrained and hence the number of valid permutations is even much less, and the solution, if it exists, is unique in most of the cases. On the contrary, if there are fewer clues in P, more valid permutations for some minigrid of P could be generated; computation of a solution for P might take more time. In any case, if there is a solution of the assumed Sudoku instance (in Fig. 3(a)), out of these seven valid permutations only one will finally be accepted following the subsequent steps of the algorithm.

To find out the next minigrid to be considered, we go through the row and column minigrids of minigrid 3 in the Sudoku instance of Fig. 3(a) (that are minigrids 1, 2, 6, and 9), and among these minigrids we find that minigrid 1 contains a maximum number of clues, i.e., 3 (which is equally true for each of the minigrids 6 and 9), and its minigrid number is minimum. Now the algorithm considers one valid permutation (out of the seven permutations) of minigrid 3 and all given clues in P, and generates all valid permutations for minigrid 1. If at least one valid permutation for minigrid 1 is obtained, we proceed for generating all valid permutations for minigrid 7 obeying all given clues in P and the assumed valid permutations of minigrids 3 and 1; otherwise, a second valid permutation of minigrid 3 is considered, for which in a similar way, we generate all valid permutations for minigrid 1, and so on.

It is straightforward to note that here the minigrid that is to be considered is one among the minigrids 2, 4, 6, 7, and 9 as the row and column minigrids of minigrids 3 and 1 (for which we have already computed valid permutation(s) one after another); we consider minigrid 7 for computing all its valid permutations allowing for one valid permutation of minigrid 3 and then one subsequent valid permutation of minigrid 1, as each of the minigrids 2, 4, 6, and 9 contains less number of clues than that of minigrid 7.

This is how the algorithm proceeds and generates all valid permutations of a minigrid under consideration obeying the given clues in P and a set of assumed valid permutations, one for each of the minigrids considered earlier in succession, up to this point in time.
Note that at the time of computing a set of valid permutations for a minigrid, we have to consider clues and (earlier computed) valid permutations in only four of the remaining eight minigrids that are adjacent to the minigrid (currently) under consideration. As for example, while computing valid permutations for minigrid 7, we have to consider one valid permutation of minigrid 1 and the clues given in minigrids (1,) 4, 8, and 9 only; here the assumed valid permutation of minigrid 3 has no use while computing valid permutations for minigrid 7. In the same way, while computing valid permutations for minigrid 6, only we have to consider the assumed valid permutation of minigrid 3 (up to this point in time) and the clues given in minigrids (3,) 4, 5, and 9 only; here the assumed valid permutations of minigrids 1 and 7 have no use while computing valid permutations for minigrid 6, and so on.

Now we discuss about the size of the tree structure under consideration. If $p$ be the number of blank cells in a minigrid and the Sudoku instance is of size $n \times n$, then the computational time as well as the computational space complexity of the guessed free Sudoku solver developed in this paper is $(p! - x)^n = O(p^n)$, where $x$ is the number of other than valid (or unwanted) permutations based on the clues given in the Sudoku instance $P$. Our observation is that for a given Sudoku instance $P$, $x$ is very close to $p!$, and hence $p! - x$ is a reasonably small number and in our case the value of $n$ is equal to 9 (or any number bounded by some constant, other than 9). Hence, the experimentations made by this algorithm take negligible amount of clock time, of the order of milliseconds.

Computation of all valid permutations for the missing digits in a minigrid is an important task. At the time of computing only all valid permutations for the missing digits, we follow a tree data structure, where the degree of the root of the tree is same as the number of missing digits, and level-wise it reduces to one to obtain the leaf vertices, where each leaf at the lowest level is a valid permutation of all the missing digits based on the clues given in $P$ (and the assumed valid permutation(s) in other minigrid(s) in subsequent iterations).

We claim that we must acquire at least one valid permutation for each of the minigrids one after another, obeying at least one valid permutation computed for each of the minigrids considered earlier in the process of assuming the minigrids in succession; we claim this result in the form of the following theorem if at least one solution of the given Sudoku puzzle exists.

**Theorem 1:** There is at least one valid permutation for the missing digits for their respective blank locations in each of the minigrids such that the combination of all such (nine) valid permutations for all the (nine) minigrids produces a desired solution, if there exists a solution of the given Sudoku instance.

**Proof:** The proof of the theorem is straightforward following the steps of the inherent development of the algorithm as stated above, if a feasible solution of the given Sudoku instance is there. We may start with one valid permutation for some earlier assumed minigrid that may not be a valid partial solution in combination for the whole Sudoku instance; then we must reach to a point of computing a valid permutation of some subsequent minigrid when no such permutation is obtained for that minigrid. In that case we are supposed to return back to the former minigrid we had to consider a next valid permutation, if any, for the same (i.e., for the previous minigrid) and move to the current minigrid for computing its valid permutations accordingly. Hence it is clear that if one valid permutation for some earlier assumed minigrid is not a valid partial solution in combination for the whole Sudoku instance, then we must have to return back to that prior minigrid to consider a new valid permutation of the same to continue the process again in computing all valid permutations of its subsequent minigrid, and so on. In this way, a set of individual valid permutations is to be differentiated so that in combination of all of them a desired solution of the given Sudoku instance is computed, if one such solution exists.

4. Applications

In this section we like to mention some exceptionally important spheres of application, where we may find that how much intrinsic occurrence in solving a given Sudoku instance might be, that different groups of researchers have pointed out mostly in last few years of their work. The word Steganography is of Greek origin and means *concealed writing*. Steganography is the art of hiding the existence of information within seemingly harmless carriers such as image, video, or audio. A message in cipher text may arouse suspicion while an invisible message is not. A digital image is a flexible medium used to carry a secret message because a minor modification of some cover image is hard to distinguish by human eyes. But the main drawback of this type of schemes is that, when it is revealed, the covert image may get distorted due to truncation of the greyscale secret. Using a $16 \times 16$ Sudoku puzzle...
and dividing it into sixteen 4×4 blocks we get as minigrids, may provide better quality in revealing the images [15].

Secret sharing refers to the method for distributing of undisclosed information amongst a group of participants, each of whom is allocated a share of secret, known as shadow. The hidden information can only be reconstructed when a sufficient number of shares is combined together, where individual shares are of no use on their own. Sudoku puzzle logic can be used for dividing the secrets into shadows and combining those shadows to reconstruct the original secret information [7].

SMS (short message service) is one of the popular mobile phone services today. In this service people can write short messages and send to a receiver through mobile communications. Securing short message services is very much important as there are possibilities of revelation of confidential and personal document during exchange of information among various systems. Better security can be achieved by hiding the SMS data in a Sudoku puzzle using some Steganographic algorithm. Then the puzzle is sent in a way that it does not attract any attention of some possible intruder. After solving the Sudoku puzzle on the receiving end, one may extract the hidden data in accordance with order of numbers 1 through 9 in one of the certain rows or columns, which is equivalent with hidden information [9].

Distribution of digital data over public networks such as the Internet is not really safe due to copyright violation, counterfeiting, forgery, and deception. Therefore, the protection of digital data, especially for confidential data, is in high demand. Suitable watermarking scheme may be used to protect the data from copyright violation. But in most of the digital watermarking schemes, achieving high hiding capability with good image quality is very difficult. By the way, it has been claimed that if we use the logic of Sudoku puzzle to create a reference matrix for digital watermarking, we can attain both the issues [11].

5. Conclusion

In this paper we have studied the different techniques of generating permutations of a given set of numbers, and also we have developed a new one for generating only valid permutations for each of the successive minigrids of a given Sudoku puzzle. We know that the problem of solving a Sudoku puzzle is beyond polynomial time computable; hence developing heuristic algorithms is one of the solving schemes. Our proposed algorithm can successfully solve a given Sudoku puzzle of size 9×9, if it has a solution. The algorithmic technique we use is backtracking, but this time we apply backtracking minigrid-wise instead of blank cells, which is extremely time consuming. The heuristic developed in this paper is also applicable for super Sudoku, Giant Sudoku, or for any large Sudoku instance, where the size of the instance is bounded by some constant.

References

1. Pentomino. http://www.en.wikipedia.org/wiki/Pentomino
2. Tetris. http://www.en.wikipedia.org/wiki/Tetris
3. Random Permutation Generation. http://www.chris-j.co.uk/randperm.php
4. Permutation Tree. http://www.techuser.net/randpermgen.html
5. Wikipedia: The Free Encyclopaedia, wikipedia.org/wiki/Fisher-Yates_shuffle
6. Arnow BJ. Representation of Permutations, Combinations and Dihedral Elements as Tree, Journal of Computers, Mathematics and Its Applications; 1990, Vol. 20(3), pp. 63-67.
7. Chang C-C, Lin P-Y, Wang ZH, Li MC. A Sudoku based Secret Image Sharing Scheme with Reversibility, Journal of Communications; 2010, Vol. 5(1), pp. 5-12.
8. Bóna Miklós, Combinatorics of Permutations, Chapman Hall-CRC, Paris, 2004.
9. Shirali-Shahreza MH, Shirali-Shahreza M. Steganography in SMS by Sudoku Puzzle, In IEEE/ACS International Conference on Computer Systems and Applications; Mar. 2008, pp. 844-847.
10. Jussien N. A-Z of Sudoku, ISTE Ltd., USA, Jan. 2007.
11. Naini PM, Fakhrizal SM, Avanaki AN. Sudoku Bit Arrangement for Combined Demosaicking and Watermarking in Digital Camera, In Second International Conference on Advances in Databases, Knowledge and Data Applications; Apr. 2010 pp. 41-44.
12. Durstenfeld R. Algorithm 235: Random Permutation, Communications of ACM; 1964, Vol. 7(7), pp. 410-20.
13. Yato T, Seta T. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles, IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences; 2003, Vol. E86-A(5), pp. 1052–1060.
14. Hong W, Chen T-S, Shiu C-W. Steganography using Sudoku Revisited, In Second International Symposium on Intelligent Information Technology Application; Dec. 2006, pp. 925-939.
15. Lee W-M. Programming Sudoku, Apress, USA, 2006.