EUROPEAN ACADEMIC RESEARCH, VOL. I, ISSUE 6/ SEPEMBER 2013
ISSN 2286-4822, www.euacademic.org
IMPACT FACTOR: 0.485 (GIF)

A Novel Algorithmic approach for solving Sudoku puzzle in Guessed Free Manner

ARNAB K. MAJI
Department of IT, North Eastern Hill University
Shillong, Meghalaya
India
SUDIPTA ROY
Department of IT, Assam University
Silchar, Assam
India
RAJAT K. PAL
Department of CSE, University of Calcutta
Kolkata, West Bengal
India

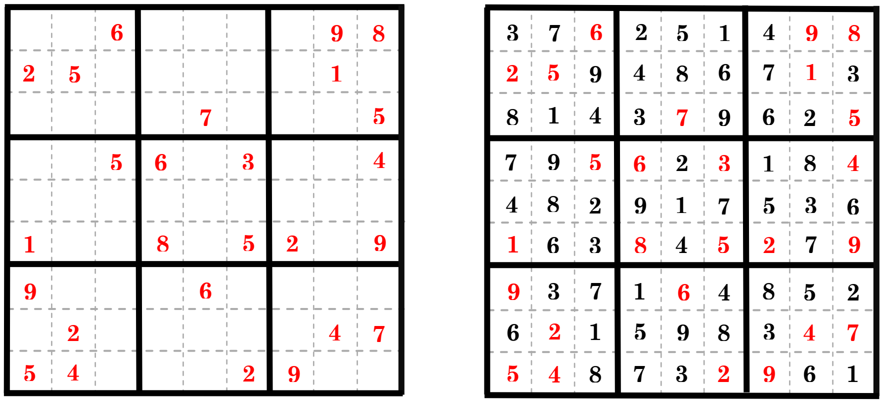
Abstract:

"Sudoku" is the Japanese abbreviation of a longer phrase, "Suuji 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. The problem of solving a given Sudoku puzzle finds numerous applications in practice. All the existing Sudoku solving techniques are primarily guess based heuristic or computation intensive soft computing methodology. In case of solving 9x9 Sudoku Puzzle, in each of these algorithms, we have to separately go through 81 cells and perform backtracking for the individual cells. In this paper, an attempt has been made to develop an algorithm which is minigrid based, i.e., we have to individually go through nine minigrids (instead of 81 cells) and perform backtracking only on them, which is less time consuming. Moreover, no guessing is involved in the whole computation.

Key words: Sudoku, Cell, Grid, Difficulty Level, Algorithm, Backtracking, Permutation, Elimination.

1. Introduction:

A Sudoku is usually a 9×9 grid based puzzle problem which is subdivided into nine 3×3 minigrids, wherein some clues are given and the objective 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 Figure 1(a) and its solution is shown in Figure 1(b).



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

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

Figure 1: (a) An instance of the Sudoku problem. (b) A solution of the Sudoku instance shown in 1(a).

Besides the standard 9×9 grid, variants of Sudoku puzzles include the following:

- 4×4 grid with 2×2 minigrids,
- 5×5 grid with *pentomino* regions published under the name *Logi-5*. A *pentomino* is composed of five congruent squares, connected orthogonally. *Pentomino* is seen in playing the game *Tetris*,
- 6×6 grid with 2×3 regions,

- 7×7 grid with six *heptomino* regions and a disjoint region,
- 16×16 grid (super Sudoku),
- 25×25 grid (Sudoku, the Giant),

A complete Sudoku solution grid may be arrived at in more than one way, as we can start from any given clues that are distributed over the minigrids of a given incomplete grid. Nobody has yet succeeded in determining how many different starting grids there are. Moreover, a Sudoku starting grid is really only interesting to a mathematician if it is minimal, i.e., if removing a single number means that the solution is no longer unique. No one has figured out the number of possible minimal grids, which amounts to the ultimate count of distinct Sudoku puzzles. It is a challenge that is sure to be taken up in the near future.

The Sudoku problem is important as it finds numerous applications in a variety of research domains with some sort of resemblance. Applications of solving a Sudoku instance are found in the fields of Steganography (Hong et al. 2008), Secret image sharing with necessary reversibility (Chang et al. 2010), Digital watermarking (Naini et al. 2010), Image authentication (Wu and Ren 2009), Image Encryption (Wu et al. 2011), Enhancement of genome sequence in DNA Sudoku (Enrich et al. 2009), Track maintenance through cooperating agents (Thaens, 2008) and so and so forth.

By the way, all the earlier existing Sudoku solvers that are available in literature (and Internet) are entirely guess based and hence extremely time consuming (Jussien 2007). In addition, each of these existing solvers solves an instance of the problem considering the clues one-by-one for each of the blank locations. Often guessing may not be guided by selecting a desired path of computing a solution and hence exhaustive redundant computations are involved over there. On the other hand, the solver developed in this paper is a minigrid based guessed free Sudoku solver which is a more deterministic algorithmic approach in the sense that redundancy is drastically reduced in this process of computations involved and that it always guarantees a solution if it exists in a reasonable amount of time.

2. Literature Review:

An $n^2 \times n^2$ Sudoku grid (consisting of $n \times n$ blocks) is an NP-complete problem (Yato and Seta 2003). So, it is unlikely to develop a polynomial time algorithm to solve this problem. There are quite a few logic techniques people use to solve this problem. Some are basic simple logic, some are more advanced. Depending on the difficulty of the puzzle, a mixture of techniques may be needed in order to solve a puzzle.

Usually Sudoku instances are available in literature based on their classification of different levels of difficulty like easy, moderate, diabolical, etc. These classifications are based on the algorithmic approaches developed and executed in solving different Sudoku instances. Sometimes it is told that the instances are easier or harder based on the number of clues given along with their relative locations in a given instance but there is no proof to support such claims. Incidentally, the Sudoku solver developed herein does not differentiate the instances in terms of any level of difficulty, and each of the instances is equally easy or hard to solve using our approach, if a reasonable number of clues are given. Table 1 shows a comparison chart of the number of clues for different difficulty levels (Lee 2006).

However, position of each of the empty cells also affects the level of difficulty. If two puzzles have the same number of clues at the beginning of a Sudoku game, the puzzle with the givens (or clues) in clusters is graded in higher level than that with the givens scattered over the space. Based on the row and column constraints, the lower bound on the number of clues are regulated in each row and column for each difficulty level (Lee 2006). as shown in Table 2.

Table 1: Number of clues given in a Sudoku puzzle in defining the level of difficulty of a Sudoku instance.

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

Table 2: The lower bound on the number of clues given in each row and column of a Sudoku instance for each corresponding level of difficulty.

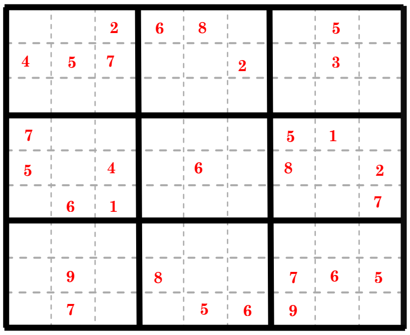
Difficulty level	**Lower bound on the number of clues in each row and column**
1 (Extremely Easy)	05
2 (Easy)	04
3 (Medium)	03
4 (Difficult)	02
5 (Evil)	00

It has already been told that our approach developed in this paper does not differentiate the instances, rather our approach computes a solution if it exists without guessing a possible value in a blank location and minigrid based irredundant deterministic computations are involved over there.

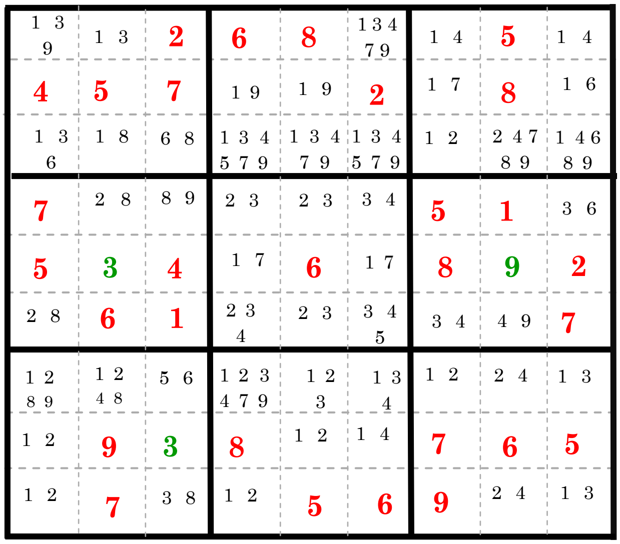
The basic technique that has been adopted for solving Sudoku puzzles is backtracking (Jussien 2007). It works as follows. The program places number 1 in the first empty cell. If the choice is compatible with the existing clues, it continues to the second empty cell, where it places a 1 (in some other row, column, and minigrid). When it encounters a conflict (which can happen very quickly), it erases the 1 just placed and inserts 2 or, if that is invalid, 3 or the next legal number. After placing the first legal number possible, it moves to the next cell and starts again with a 1. If the number that has to be changed is a 9 (which cannot be raised by one in a standard Sudoku grid), the program backtracks and increases the number in the previous cell (the next-to-last number placed) by one. Then it moves forward until it hits a conflict.

In this way, the program may sometimes backtrack several times before advancing. It is guaranteed to find a solution if there is one, simply because it eventually tries every possible number in every possible location.

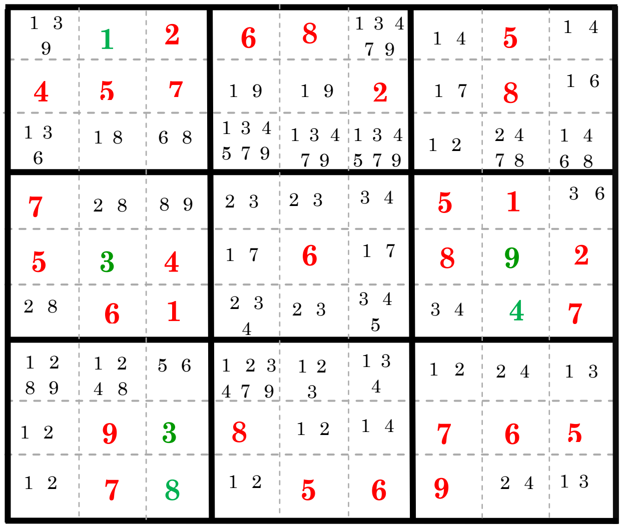
Figure 3. (a) An instance of a Sudoku puzzle. (b) Potential values in each blank cell are inserted based on the given clues of the Sudoku instance in Figure 3(a); here green digits are naked singles. (c) The concept of naked singles is preferably used to reduce the domain of probable candidate values in each blank cell, and the process is successive in nature to find out consequent naked singles, as much as possible. As for example, the naked single for cell [9,8] is 2, as 4 and 8 have already been recognized as naked singles along row 9 and column 8; then 8 is a naked single for cell [7,8], as 2 and 4 are already identified naked singles along column 8, and so on.



(a)



(b)



(c)

Some other techniques include *elimination based approach* (Lee 2006) and *soft computing based approach* (Jussien 2007). Let us now focus to review the elimination based approach. In this approach, based on the given clues a list of possible values for every blank cell is first obtained. Then using the following different methods such as *naked single*, *hidden single*, *lone ranger*, *locked candidate*, *twin*, *triplet*, *quad*, *X-wing*, *XY-wing*, *swordfish*, *coloring*, we eliminate the multiple possibilities of each and every blank cell, satisfying the constraints that each row, column, and minigrid should have the numbers 1 through 9 exactly once. An instance of a Sudoku puzzle and its possible values of each blank cell are shown in Figures 3(a) and 3(b), respectively.

B. Naked single

If there is only one possible value existing in a blank cell, then that value is known as a *naked single* (Lee 2006). After assigning the probable values for each blank cell, as shown in Figure 3(b), we obtain the naked singles 3, 9, and 3 at locations [5,2], [5,8], and [8,3], respectively. So, we can directly assign these values to these cells. Then we eliminate these digits (or naked singles) from each of the corresponding row, column, and minigrid. Hence, after elimination of these numbers, as stated above, we obtain a modified (reduced) status of each blank cell as shown in Figure 3(c), wherein several other naked singles could be found (and this process is recursive until no naked singles are found).

C. Hidden single

Sometimes there are blank cells that do, in fact, have only one possible value based on the situation, but a simple elimination of candidate in that cell's row, column and minigrid does not make it obvious. This kind of possible value is known as a *hidden single* (Lee 2006). Suppose, if we re-examine the possible values in each cell of Figure 3(b), hidden single can easily be found in cell [7,2] whose value must be 4 as in minigrid numbered 7, 4 is not there as probable values in other cells. Similarly, for cell [4,9], the hidden single is 6 (as in other cells of the same minigrid 6 is not present as probable values). Most of the puzzles ranked as easy, extremely easy and medium can simply be solved using these two techniques of singles.

D. Lone ranger

Lone ranger is a term that is used to refer to a number that is one of multiple possible values for a blank cell that appears only once in a row, or column, or minigrid (Lee 2006). To see what this means in practice, consider a row of a Sudoku puzzle with all its possibilities for each of the cells (red digits are either givens or already achieved), as shown in Figure 4. In this row, six cells (with red digits) have already been filled in, leaving three unsolved cells (second, eighth, and ninth) with their probable values written in them.

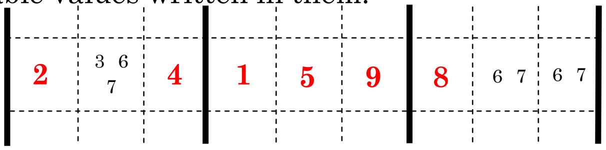


Figure 4. An example row of a Sudoku puzzle with a lone ranger 3 in the second cell.

Notice that the second cell is the only cell that contains the possible value 3. Since none of the remaining cells in this row can possibly contains 3, this cell can now be confirmed with the number 3. In this case, this 3 is known as a lone ranger.

E. Locked candidate

Sometimes it can be observed that a minigrid where the only possible position for a number is in one row (or column) within that block, although the position is not fixed for the number. That number is known as a *locked candidate* (Lee 2006). Since the minigrid must contain the number in a row (or column) we can eliminate that number not as a probable candidate along the same row (or column) in other minigrids. Consider the Sudoku puzzle along with its probable assignments for each blank cell, as shown in Figure 5. It can readily be found that minigrid numbered 6 should have 3 in the last row. So we can simply eliminate number 3 from cell [6,5] of minigrid numbered 5. Similarly, minigrid numbered 8 should have 3 in its first column. So, 3 can be eliminated as a possible candidate from cell [4,4].

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

Figure 5. A Sudoku puzzle with probable locked candidates in the last row of minigrid 6 (and here the locked candidates are 3 and 5 in cells [6,7] and [6,8]), in the first column of minigrid 8 (and here the locked candidates are 9 and 3 in cells [8,4] and [9,4]), and so on.

F. Twin

If two same possible values are present for two blank cells in a row (or column) of a Sudoku puzzle, they are referred to as *twin* (Lee 2006). Consider the partially solved Sudoku puzzle as shown in Figure 6(a). Observe the two cells [2,5] and [2,6]. They both contain the values 2 and 3 (means either 2 or

3). So, if cell [2,5] takes value 2, then cell [2,6] must contain 3, or vice versa. This type of situation is an example of twin.

Once a twin is identified, these values can be eliminated by striking through from the same row, column, and minigrid as shown in Figure 6(b), as the values cannot be probable candidates in other blank cells along the same row (or column) and in the same minigrid.

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

(a)

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

(b)

Figure 6. (a) A partial Sudoku instance with presence of twin 2 and 3 in cells [2,5] and [2,6]. (b) Elimination of probable values (that are 2 and 3) based on the twin from the second row (2 is deleted from cells [2,1] and [2,3]) and from the same minigrid (2 and 3 are deleted from cells [1,4] and [1,5]).

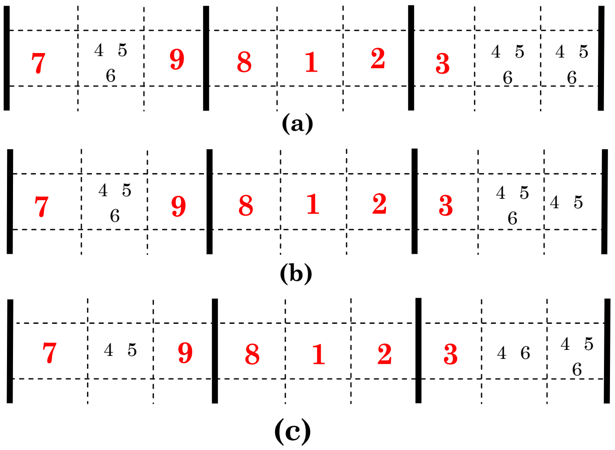


Figure 7. Example rows of Sudoku puzzles with different varieties of triplet. (a) A triplet of *Variety# 1* with same three possible values present in three cells. (b) A triplet of *Variety# 2* with same three possible values present in two cells and the other cell containing any two of them. (c) A triplet of *Variety# 3* with three possible values present in one cell and the two other cells containing two different subsets of two possible values of the earlier three values.

G. Triplet

If three cells in a row (or column) are marked with a set of same three possible values, they are referred to as *triplet* (Lee 2006). Like twins, triplets are also useful for eliminating some other possible values for other blank cells. Triplet has several variations like the following.

Variety# 1: Three cells with same three possible values, as shown in Figure 7(a).

Variety# 2: Two cells with same three possible values and the other cell containing any two of the possible values, as shown in Figure 7(b).

Variety# 3: One cell with three possible values and the two other cells containing two different subsets of two possible values of the former three values, as shown in Figure 7(c).

Once a triplet is found, we can eliminate all the values of the triplet that are there as possible candidates in other blank cells along the same row (or column) and in the same minigrid.

H. Quad

Analogous to triplet, a *quad* consists of a set of four possible values and these values are present in some form in four blank cells in a row (or column) of the Sudoku instance (Lee 2006). That is, if the values only exist in four (blank) cells in a row (or column), while each cell contains at least two of the four values, then other values (or numbers except the specified four values) can be eliminated from each of the assumed cells (forming the quad). Figure 8 shows a row of a Sudoku puzzle where the quad comprising the digits 1, 2, 4, 7 formed by the cells in column four, six, seven, and eight. So other possible values can straightway be eliminated from these cells, as shown by striking through the inapplicable digits in the figure.

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

Figure 8. An example row of a Sudoku puzzle with quad comprising digits 1, 2, 4, and 7 present in columns four, six, seven, and eight. To support the digits present in the quad in the stated cells, other probable values (like 8 and 9 in columns six, seven, and eight) are eliminated from these cells of the quad, as these values (that are 8 and 9) cannot be probable digits for the specified cells.

An extended version of the above algorithm(Jussien 2007) defines a set of terms like XWing, Swordfish, Hidden subset, etc, but ultimately it is also a trial based algorithmic way-out which is a guess based, cell based Sudoku solver (Jussien 2007). The *XWing* technique can be applied when there are two rows or columns for which a given value is possible to assign only to two blank cells. If those four cells are in only two orthogonal rows or columns, then all other cells in those regions will never get assigned to this value. The *XWing* technique can easily be generalized but *swordfish* is a technique that further makes the possibility of assigning a digit to a blank cell more specific. On the other hand, *hidden subset* is a kind of technique that is very similar to twin or triplet or quad that have already been explained above.

On the other hand, all the soft computing based Sudoku solvers either use Genetic algorithm (Mantere and Koljonen 2007) or Bee colony (Pacurib et al. 2009), which is exhaustive and extremely time-consuming. Needless to mention that these techniques use their own set of operators to execute the respective algorithms. Genetic algorithms belong to the larger class of evolutionary algorithms that generate solutions to optimization problems using techniques inspired by natural evolution, such as selection, crossover, mutation, and inheritance.

The Simulated annealing based Sudoku solver is a probabilistic Sudoku solver. The general design is capable of solving a Sudoku instance of order up to fifteen [7]. It has been claimed that the solver has solved in actual hardware Sudoku puzzles of up to order 12 within the competition-imposed time limits.

3. Proposed Algorithm for solving Sudoku puzzle in a guessed free manner:

All the previous algorithms discussed in the literature survey are cell based and some amount of guessing is always involved in all the technique. In this paper an attempt has been made to develop an algorithm which is minigrid based, i.e., we have to individually go through nine minigrids (instead of 81 cells) and perform backtracking only on them, which is less time consuming. Moreover, no guessing is involved and no redundant computation is performed during the whole computation.

Our proposed algorithm considers each of the minigrids that may be numbered as 1 through 9 as shown in Figure 9. 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 $1 \leq i, j, k, l, m \leq 9$.

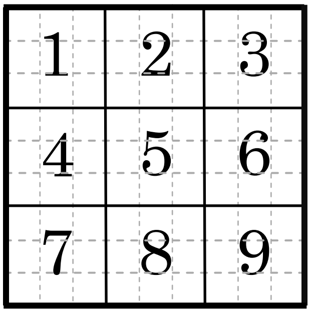


Figure 9: The structure of a 9×9 Sudoku puzzle (problem) with its nine minigrids of size 3×3 each as numbered 1 through 9. Minigrid number 1 consists of the cell locations [1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], and [3, 3], minigrid number 2 consists of the cell locations [1, 4], [1, 5], [1, 6], [2, 4], [2, 5], [2, 6], [3, 4], [3, 5], and [3, 6], and so on.

Now to start with a minigrid as stated above, we find that the 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 Figure 1(a)). For example, for the Sudoku instance as shown in Figure 1(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).

Besides, for a given Sudoku instance, we know all the clues given as well as the clue positions among the cells of a minigrid and subsequently the blank cells are also known to us. For example, the given clues in minigrid 3 of Figure 1(a) are 9 at location [1, 8], 8 at location [1, 9], 1 at location [2, 8], and 5 at location [3, 9]. 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 2, 3, 4, 6, and 7.

In this algorithm, we compute all possible permutations of these missing digits in minigrid 3, where the first permutation may be 23467 (the minimum number) and the last permutation may be 76432 (the maximum 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 first permutation 23467 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 the location [6, 7] already contains 2 as a clue of minigrid 6, and we cannot place 2 at [1, 7] as the permutation suggests. Also the location [3, 5] contains 7 as a clue of minigrid 2, and we cannot place 7 at [3, 8] as it is supposed to place.

Similarly, we may find that the last permutation 76432 is also not a valid permutation as location [4, 9] already contains 4 as a clue of minigrid 6, and we cannot place 4 at [2, 9] as the permutation suggests. But we may observe that 74362 is a valid permutation as we may safely place 7 at [1, 7], 4 at [2, 7], 3 at [2, 9], 6 at [3, 7], and 2 at [3, 8] based on the other clues in the corresponding rows and columns of other minigrids (that are minigrids 1, 2, 6, and 9).

This is how we may compute all valid permutations of minigrid 3, and proceed for a next minigrid that belongs to among the row and column minigrids of minigrid 3 which contains a maximum number of clues but the minigrid number is minimum. Among all the valid permutations (for their respective blank locations) of minigrid 3, at least one permutation must last at the end of computation of valid permutations of each of the remaining minigrids if the solution of the given Sudoku instance is unique. 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 Figure 1(a) (that are minigrids 1, 2, 6, and 9), and among these minigrids we find that the 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 the minimum.

So, now we consider minigrid 1, and as done before for minigrid 3, we find the given clues and the missing digits therein along with their locations. Here we do exactly the same as we did earlier in computing all permutations of the missing digits in minigrid 3. At the time of identifying all valid permutations of minigrid 1, we consider one valid permutation (at their respective blank locations) of minigrid 3 in addition to all given clues of the instance under consideration. If we get at least one valid permutation for minigrid 1 (obeying an assumed valid permutation of minigrid 3), we consider it for some subsequent computation of permutations of another minigrid; otherwise, we consider a second valid permutation of minigrid 3, and based on that we compute another set of valid permutations for minigrid 1, and so on.

Now it is straightforward to declare 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); note that neither of minigrids 5 and 8 is a row or column minigrid of minigrids 3 and 1. Hence following the instance in Figure 1(a), 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, in addition to all given clues of the instance under consideration, as each of the minigrids 2, 4, 6, and 9 contains less number of clues than that of minigrid 7. Here in computing all valid permutations of minigrid 7, we may not consider an assumed valid permutation of minigrid 3, as this minigrid is neither in a row nor in a column of minigrid 7, but we have to consider a valid permutation of minigrid 1 and all given clues in the Sudoku instance (primarily the clues given in minigrids 4, 8, and 9).

This process is continued till a valid permutation of a minigrid (or a set of valid permutations of a group of minigrids) is propagated to compute a valid permutation of a subsequent minigrid, and eventually a valid permutation of the last minigrid (i.e., the ninth minigrid; not necessarily minigrid number 9) is computed, which altogether generate a desired solution of the given Sudoku instance. It may so happen that up to t minigrids t valid permutations that we consider in a series match each other towards a valid combination of the given Sudoku instance but there is no valid permutation for the (t+1)th minigrid obeying the earlier assumed t valid permutations, where $1 < t < 9$. Then we consider a second valid permutation of the t^{th} minigrid, and after that we try to compute a valid permutation for the $(t+1)^{th}$ minigrid, if one exists. If for none of the valid permutations of the tth minigrid a valid permutation for the $(t+1)^{th}$ minigrid is obtained, we consider a second valid permutation for the $(t-1)^{th}$ minigrid that leads to compute a new set of valid permutations for the t^{th} minigrid, and so on.

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 a 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.

To see the algorithm at a glance, let us write it in the form as follows:

Algorithm: A Guessed Free Sudoku Solver – Version 1

Input: A Sudoku instance, P of size 9×9.

Output: A solution, S of the given Sudoku instance, P.

Step 1: Compute the number of clues, digit(s) given as clue, and the missing digits in each of the minigrids of P.

Step 2: Compute S_M, a sequence of minigrids that contains all the minigrids in succession, wherein $M \in S_M$ is the minigrid (and the first member in S_M) with a maximum number of clues and whose minigrid number is minimum. In S_M, a member N is a minigrid which is either in the row or in the column of any of its earlier members in S_M including M that contains a maximum number of clues and whose minigrid number is minimum, where $1 < N \leq 9$.

Step 3: Compute all valid permutations for the missing digits in M, and store them.

Step 4: For all the remaining minigrids in succession in S_M do the following:

Step 4.1: Consider a next minigrid, $N \in S_M$, and compute all its valid permutations for the missing digits in N assuming a valid permutation for each of the earlier minigrids up to M, and store them.

Step 4.2: If one valid permutation for N is obtained, then consider a next minigrid of N in S_M, if any, and compute all its valid permutations for the missing digits in this minigrid assuming a valid permutation for each of the earlier minigrids up to M, and store them.

Else consider a next valid permutation, if any, of the immediately previous minigrid of N, and compute all its valid permutations for the missing digits in N assuming a valid permutation for each of the earlier minigrids up to M, and store them.

Step 5: If all the valid permutations of the immediate successor minigrid of M are exhausted to obtain a valid combination for all the nine minigrids in S_M, then consider a next valid permutation of M and go to Step 4. The process is continued until a valid combination for all the nine minigrids in S_M is obtained as a desired solution S for P; otherwise, the algorithm declares that there is no valid solution for the given instance P. Now it is straightforward to compute S_M for a given Sudoku instance P. As for example, consider the Sudoku instance given in Figure 1(a). According to this instance the sequence S_M of minigrids is ⟨3, 1, 7, 6, 5, 9, 4, 8, 2⟩ as it has been described and performed in Step 2 of the first version of the algorithm above.

Computation of all valid permutations for the missing digits in a minigrid is an important task of the present algorithm. 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).

As for example, the number of clues given in minigrid 3 of Figure 1(a) is 4, and the missing digits are 2, 3, 4, 6, and 7. 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]. Here the tree structure we like to compute is shown in Figure 10, whose root does not contain any permutation of the missing five digits, and it is represented by '×××××'. This root is having five children where the first child leads to generate all valid permutations staring with 2, the second child leads to generate all valid permutations staring with 3, and so on.

Now note that none of the permutations starting with 2 is a valid permutation as column 7 of minigrid 6 contains 2 as given clue (at location [6, 7]). So, we do not expand this vertex (i.e., vertex with permutation '2××××') further in order to compute only the set of desired valid permutations. Similarly, we do not expand the child vertex with permutation '6××××', as location [1, 3] contains 6 as given clue. Up to this point in time, as either 3, or 4, or 7 could be placed at [1, 7], we expand each of the child vertices starting with permutations 3, and 4, and 7, as shown in Figure 10.

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 respective 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 leaf vertices only. As for example, the vertex with permutation '742××' is not expandable, because we cannot place 2 at [2, 9] as [2, 1] contains a 2 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 (unwanted) permutations at this point in time.

Interestingly, Figure 10 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 Figure 1(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, and P may have two or more valid solutions. In any case, if there is a unique solution of the assumed Sudoku instance (in

Figure 1(a)), out of these seven valid permutations only one will finally be accepted following the subsequent steps of the algorithm.

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.

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.

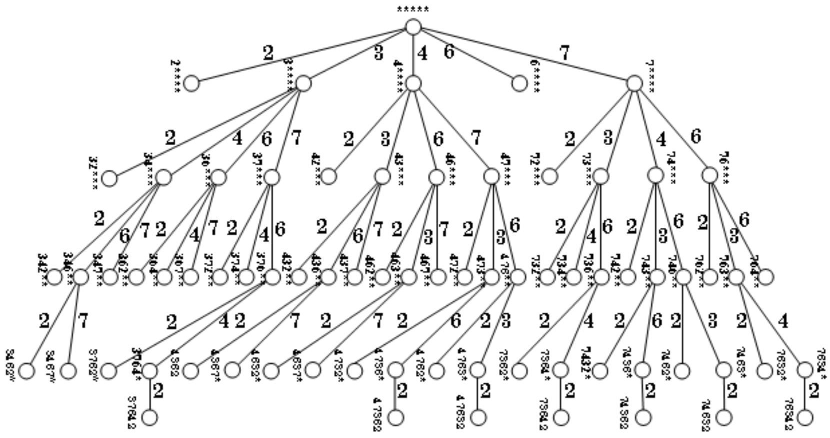


Figure 10: The permutation tree for generating only valid permutations of the missing digits in minigrid 3 of the Sudoku instance shown in Figure 1(a).

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 complexities of the guessed free Sudoku solver developed herein is $(p!-x)^n = O(p^n)$, where x is the number of other than valid 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. Hence, the experimentations made by this algorithm take negligible amount of clock time, of the order of milliseconds. We conclude that the algorithm developed in this paper guarantees a valid solution of a given Sudoku puzzle, if one exists, and theoretically it takes time and space exponential in size; these has been stated in the following theorem and proved thereafter.

Theorem 2: The guessed free Sudoku solver developed in this paper guarantees a valid solution of a Sudoku instance, if one exists, and it takes time and space complexities O(p^n), where p is the number of blank cells in a minigrid of a Sudoku instance of size $n \times n$.

Proof: The guessed free Sudoku solver developed in this paper considers the minigrids of a given Sudoku instance P in a particular sequence in a deterministic way. Hence, if there be a solution S for P, then each minigrid must have its own valid permutation for its missing digits and in combination of all of them an overall valid solution S for P is obtained, if S is unique. Incidentally, the guessed free Sudoku solver generates a set of only valid permutations for a minigrid, and the required valid permutation for the minigrid in S must be a member of this set. If the algorithm starts from a valid permutation for a minigrid, or considers a valid permutation for some subsequent minigrid, which may not be a valid permutation towards computing S, then the algorithm must reach to a point when no valid permutation for a later minigrid will be generated, and we have to revert back to the previous minigrid to consider its next valid permutation.

In this process of computation, if S is unique for P, the algorithm must consider the valid permutation of the initial minigrid, which will be the final valid permutation for the minigrid in S, then generation of permutations of the subsequent minigrids and their consideration will eventually lead to the desired solution S for P. This process might have several backtrackings, but as the number of valid permutations for a minigrid is negligibly less and the number of minigrids is confined to nine only, S is guaranteed to be computed in a reasonable amount of time, if it exists.

Now it is straightforward to prove that the algorithm takes both computational time and space O(p^n), as generation of all valid permutations of a minigrid is the dominating computations involved in this algorithm, where p is the number of blank cells in a minigrid of P of size $n \times n$. Here p^n is the size of the tree structure we compute while generating all valid permutations of the missing digits in a minigrid. In practice, the size of the tree structure is significantly less than the asymptotic upper bound mentioned herein.

4. Conclusion

In this paper we have proposed an exclusive minigrid based Sudoku solver algorithm, which is completely guessed free. The solver considers each of the minigrids (instead of blank cells in isolation) of size 3×3 each that has been developed for the first time in designing such an algorithm for a given Sudoku puzzle of size 9×9. In our algorithm a pre-processing is there for computing only all valid permutations for each of the minigrids based on the clues in a given Sudoku puzzle.

It has been observed in most practical situations that the number of valid permutations is appreciably less than the total number of possible permutations for each of the minigrids, and even if there are less clues in some minigrid of an instance, clues present in four adjacent row and column minigrids severely help in reducing the ultimate number of valid permutations for that minigrid too. Anyway, this approach of minigrid-wise computation of valid permutations and checking their compatibility among row minigrids and column minigrids is absolutely new and done for the first time in this domain of work.

As we consider minigrids for finding only the valid solutions of a given Sudoku puzzle instead of considering the individual (blank) cells, therefore, the computations involved in the algorithm is significantly reduced. In the case of a 9×9 Sudoku puzzle, there are 81 such cells with some clues (which is less) and the remaining blank cells (which is more), whereas there are only nine such minigrids each of which consists of 3×3 cells. Here the observation to a Sudoku puzzle is not by searching of missing numbers cell-by-cell (as if searching for an address by moving through streets); rather, it is one step above the ground of the puzzle by considering groups of cells or minigrids (and searching the same from a bird's eye view).

The brilliancy of the algorithm developed in this paper is that the same logic can also be straightway applied for larger Sudoku instances such as 16×16, 25×25, or of any other rectangular sizes (with their respective objective functions).

The level of difficulty is another important issue that almost all the earlier Sudoku solvers consider while developing an algorithm. There are no hard and fast rules that state the difficulty level of a Sudoku puzzle. A sparsely filled Sudoku puzzle may be extremely easy to solve, whereas a densely filled Sudoku puzzle may actually be more difficult to crack. From a programming viewpoint, we can determine the difficulty level of a Sudoku puzzle by analyzing how much effort must be expended to solve the puzzle, and the different levels of difficulty are easy, medium, difficult, extremely difficult, etc. Incidentally, the Sudoku solver developed in this paper does not depict any level of difficulty; rather, all the Sudoku instances are having the same level of difficulty. In some cases, more valid permutations may be generated for some minigrid, but in general, the number of valid permutations is much less, and the minigrids with smaller number of valid permutations in fact guide to compute eventually all the desired solutions for a given Sudoku instance.

BIBLIOGRAPHY:

Chang, C.-C., P.-Y. Lin, Z. H. Wang, and M. C. Li. 2010. "A Sudoku based Secret Image Sharing Scheme with Reversibility." *Journal of Communications* 5(1): 5-12.

Erlich, Y., K. Chang, A. Gordon, R. Ronen, O. Navon, M. Rooks, and G. J. Hannon. 2009. "DNA Sudoku: Harnessing

High-throughput Sequencing for Multiplexed Specimen Analysis." *Genome Research Journal* 19(7): 1243-1253.

Hong, W., T.-S. Chen, and C.-W. Shiu. 2008. "Steganography using Sudoku Revisited." Proc. 2nd International Symp. Intelligent Information Technology Application: 935-939. Accessed June 2013, Available at http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4739900&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D4739900.

Jussien, N. 2007. *A-Z of Sudoku.* USA: ISTE Ltd.

Lee, W.-M. 2006. *Programming Sudoku.* USA: Apress.

Mantere, T. and J. Koljonen. 2007. "Solving, Rating and Generating Sudoku Puzzles with GA." *IEEE Congress on Evolutionary Computation*: 1382-1389.

Naini, P. M., S. M. Fakhraie, and A. N. Avanaki. 2010. "Sudoku Bit Arrangement for Combined Demosaicking and Watermarking in Digital Camera." Proc. 2nd International Conf. Advances in Databases, Knowledge and Data Applications: 41-44. Accessed June 2013, Available at http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=5477147&contentType=Conference+Publications.

Pacurib, J. A., G. M. M. Seno, and J. P. T. Yusiong. 2009. "Solving Sudoku Puzzles using Improved Artificial Bee Colony Algorithm." In: Proc. 4th Int. Conf. Innovative Computing, Information and Control: 885-888. Accessed July 2013, Available at http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5412260&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F5410746%2F5412194%2F05412260.pdf%3Farnumber%3D5412260.

Thaens, R. 2008. "Track Maintenance through Cooperating Agents." Proc. 11th International Conference on Information Fusion: 1-8. Accessed July 2013, Available at http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4632285&url=http%3A%2F%2Fieeexplore.ieee.org
%2Fxpls%2Fabs_all.jsp%3Farnumber%3D4632285.

Wu, W.-C. and G.-R. Ren. 2009. "A New Approach to Image Authentication using Chaotic Map and Sudoku Puzzle", Proc. 5th Int. Conf. Intelligent Information Hiding and Multimedia Signal Processing: 628-631. Accessed May 2013, Available at http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5337415&contentType=Conference+Publications&queryText%3DA+New+Approach+to+Image+Authentication+Using+Chaotic+Map+and+Sudoku+Puzzle.

Wu, Y., J. P. Noonan, and S. Agaian. 2011. "Image Encryption using the Rectangular Sudoku Cipher." Proc. International Conference on System Science and Engineering: 704-709. Accessed July 2013, Available at http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=5961994&contentType=Conference+Publications.

Yato, T. and T. Seta. 2003. "Complexity and Completeness of Finding Another Solution and Its Application to Puzzles." *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences* E86-A(5): 1052–1060.