Sudoku Solver using Minigrid based Backtracking

Arnab Kumar Maji
Department of Information Technology
North Eastern Hill University
Shillong, India
arnab.maji@gmail.com

Rajat Kumar Pal
Department of Computer Science and Engineering
University of Calcutta
Kolkata, India
pal.rajatk@gmail.com

***Abstract*—'Sudoku' is a popular Japanese puzzle game that trains our logical mind. The word Sudoku means 'the digits must remain single'. 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, Secret image sharing with necessary reversibility, Encrypting SMS, Digital watermarking, Image authentication, Image Encryption, and so and so forth. All the existing Sudoku solving techniques are primarily guess based heuristic or computation intensive soft computing methodology. They are all cell based, that is why very much time consuming. Therefore, in this paper a minigrid based novel technique is developed to solve the Sudoku puzzle in guessed free manner.**

Keywords—Sudoku puzzle; Minigrid; Backtracking; Permutation; Permutation tree; Algorithm.

I. INTRODUCTION

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 9×9. 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* [1]. A *pentomino* is composed of five congruent squares, connected orthogonally. *Pentomino* is seen in playing the game *Tetris* [2],
- 16×16 grid (super Sudoku) [3],
- 25×25 grid (Sudoku, the Giant) [3], etc.

Solving an instance of Sudoku problem is NP-complete [8]. So it is unlikely to develop a deterministic polynomial time algorithm for solving a given Sudoku puzzle of size $n \times n$, where n is any large number such that the square root of n is an integer. But incidentally when the value of n is bounded by some constant, solutions may be obtained in reasonable amount of time [3].

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 [4].

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
1 (Extremely Easy)	05
2 (Easy)	04
3 (Medium)	03
4 (Difficult)	02
5 (Evil)	00

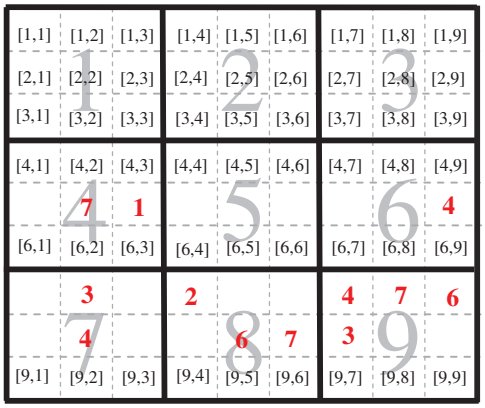


Figure 1: The structure of a 9×9 Sudoku puzzle (problem) with its nine minigrids of size 3×3 each as numbered (in grey outsized font) 1 through 9. Representation of each cell of a Sudoku puzzle and some example givens (or clues coloured by red) in the remaining cells. So, the cells are [1,1] through [9,9], and the distinct cells may have some clues as well. Minigrid numbered 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 numbered 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.

However, position of each of the empty cells also affects the level of difficulty. If two puzzles have the same number of

978-1-4799-2572-8/14/$31.00 ©2014 IEEE

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 [4] as shown in Table 2.

In our proposed approach, we have divided a 9×9 Sudoku puzzle into nine 3×3 minigrids. We have labelled each minigrid from 1 to 9, with minigrid 1 at the top-left corner and minigrid 9 at the bottom-right corner; minigrid numbers are shown in faded larger font size in Figure 1. Also we refer to each cell in a minigrid by its row number followed by its column number, as shown in the same figure.

Then we have generated permutations among minigrids based on the given clues. Then based on valid permutation amongst minigrids using backtracking approach, the final solution of the Sudoku puzzle has been generated, if the given puzzle has a solution.

II. Literature Survey on Existing Sudoku Solver

The basic technique that is used to solve a Sudoku Puzzle is backtracking. Some other techniques include *elimination based approach* [4] and *soft computing based approach* [3].

Now we review on the backtracking technique that has been adopted for solving Sudoku puzzles [4]. The basic backtracking algorithm 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 (or a minimum possible acceptable value). If the number that has to be altered is a 9, which cannot be raised by one in a standard 9×9 Sudoku grid, the process backtracks and increases the number in the previous cell (or the next to the last number placed) by one. Then it moves forward until it hits a new conflict.

In this way, the process 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. There are several means by which this algorithm can be improved: *constraint propagation*, *forward checking*, and *choosing most constrained value first* [4] are some of them.

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*, 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 2(a) and 2(b), respectively.

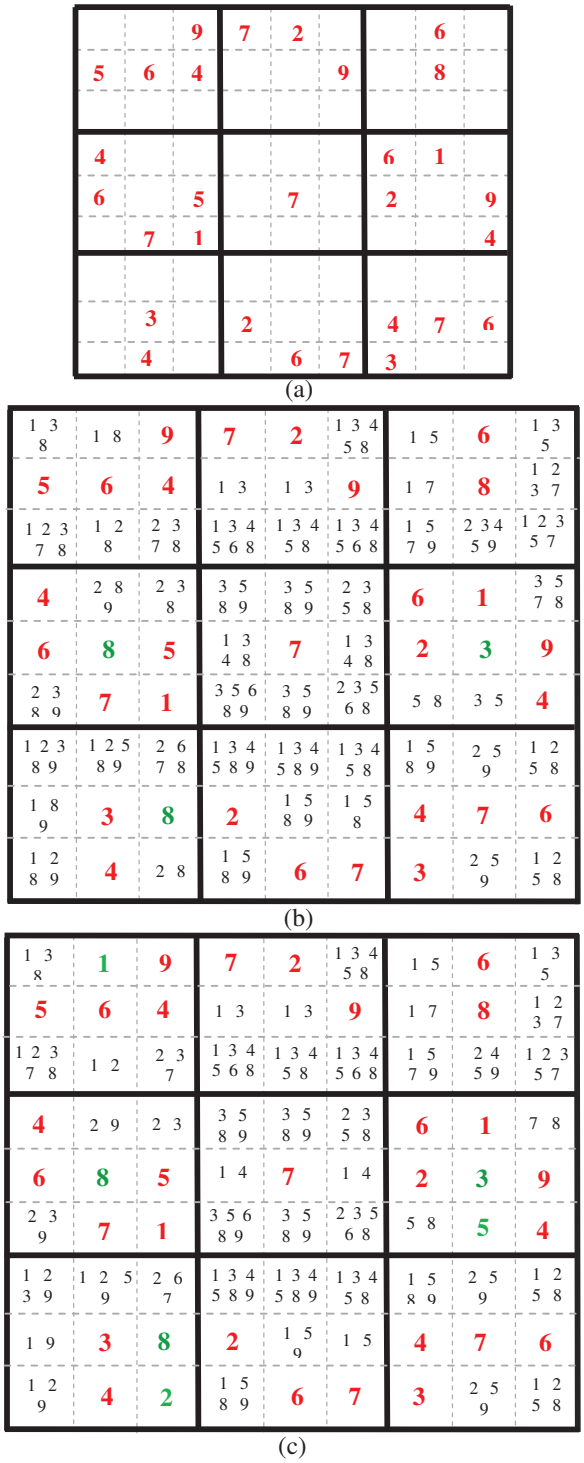


Figure 2: (a) An instance of a Sudoku puzzle. **(b)** Potential values in each blank cell are inserted based on the given clues of the Sudoku instance in Figure 2(a); here green digits are naked singles. **(c)** The concept of naked singles is preferably used to reduce the domain of probable candidate 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 9, as 2 and 5 have already been recognized as naked singles along row 9 and column 8; then 2 is a naked single for cell [7,8], as 5 and 9 are already identified naked singles along column 8, and so on.

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*, 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 2(a) and 2(b), respectively.

Naked single: If there is only one possible value existing in a blank cell, then that value is known as a *naked single* [3]. After assigning the probable values for each blank cell, as shown in Figure 2(b), we obtain the naked singles 8, 3, and 8 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 2(c), wherein several other naked singles could be found (and this process is recursive until no naked singles are found). As for example, the naked single for cell [9,8] is 9, as 2 and 5 have already been recognized as naked singles along row 9 and column 8; then 2 is a naked single for cell [7,8], as 5 and 9 are already identified naked singles along column 8, and so on.

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* [4]. Suppose, if we re-examine the possible values in each cell of Figure 2(b), hidden single can easily be found in cell [7,2] whose value must be 5 as in other columns along minigrid numbered 7, 5 is already present. Similarly, for cell [4,9], the hidden single is 7 (as in other rows along minigrid numbered 6, 7 is already present). Most of the puzzles ranked as easy, extremely easy, and medium can simply be solved using these two techniques of singles.

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

Figure 3: An example row of a Sudoku puzzle with a lone ranger 8 in the second cell.

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 [4]. 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 3. 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.

Notice that the second cell is the only cell that contains the possible value 8. Since none of the remaining cells in this row can possibly contains 8, this cell can now be confirmed with the number 8. In this case, this 8 is known as a lone ranger.

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* [3]. 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 4. It can readily be found that minigrid numbered 6 should have 8 in the last row. So we can simply eliminate number 8 from cell [6,5] of minigrid numbered 5. Similarly, minigrid numbered 8 should have 8 in its first column. So, 8 can be eliminated as a possible candidate from cell [4,4].

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

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

Twin: If two same possible values are present for two blank cells in a row (or column) of a Sudoku puzzle, they are referred as *twin* [4]. Consider the partially solved Sudoku puzzle as shown in Figure 5(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 5(b), as the values cannot be probable candidates in other blank cells along the same row (or column) and in the same minigrid.

Triplet: If three cells in a row (or column) are marked with a set of same three possible values, they are referred as *triplet* [4]. 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 6(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 6(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 6(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.

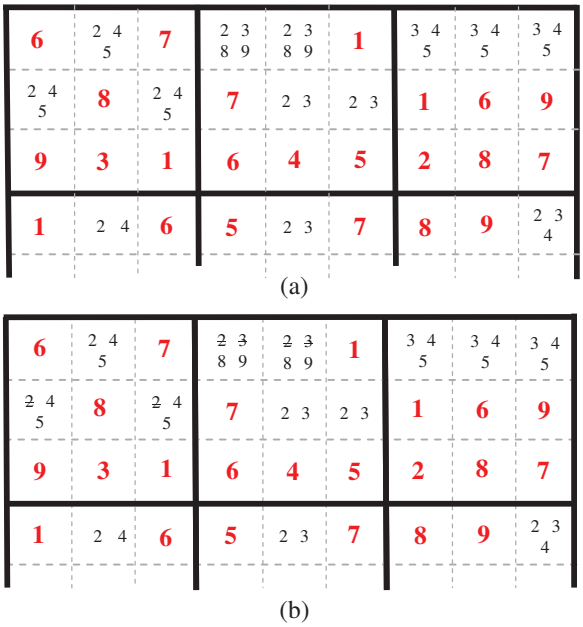


Figure 5: (a) A partial Sudoku instance with presence of twin 2 and 3 in cells [2,5] and [2,6]. **(b)** Elimination of probable values (that are 2 and 3) based on the twin from the second row (2 is deleted from cells [2,1] and [2,3]) and from the same minigrid (2 and 3 are deleted from cells [1,4] and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

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 [3]. 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 7 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.

An extended version of the above algorithm defines a set of terms, like XWing, swordfish, hidden subset, etc. but ultimately it is also a trial based algorithmic technique, which is a guess based, cell based Sudoku solver [3, 4]. The *XWing* method can be applied when there are two rows and columns for which a given value is possible to assign only to two blank (diagonal) cells. If these four cells are only at the intersections of two orthogonal rows and columns, then all other cells along these rows and columns will never get assigned to this value. The XWing method can easily be generalized, but *swordfish* is a procedure that further makes the possibility of assigning a value to a blank cell more specific. On the other hand, *hidden subset* is a kind of practice that is very similar to twin (or triplet or quad) that have already been explained above [3, 4].

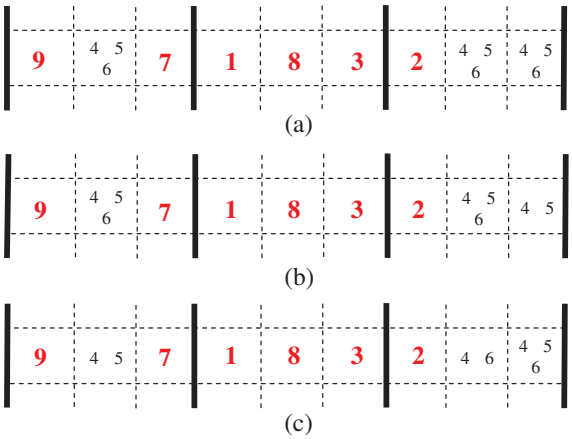


Figure 6: Example rows of Sudoku puzzles with different varieties of triplet. **(a)** A triplet of *Variety# 1* with same three possible values present in three cells. **(b)** A triplet of *Variety# 2* with same three possible values present in two cells and the other cell containing any two of them. **(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.

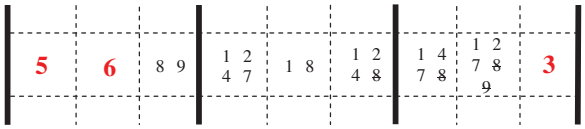


Figure 7: An example row of Sudoku puzzle with quad comprising digits 1, 2, 4, and 7 present in columns 4, 6, 7, and 8. To support the digits present in the quad in the stated cells, other probable values (like 8 and 9 in columns 6, 7, and 8) are eliminated from these cells of the quad, as these values (that are 8 and 9) cannot be probable digits for the specified cells.

On the other hand, all the soft computing based Sudoku solvers, either use genetic algorithm [6] or bee colony [7] or some other soft computing based technique, are truly exhaustive and extremely time consuming. Needless to mention, that these approaches use their own sets of operators to execute the respective algorithms. *Genetic algorithms* belong to the larger class of evolutionary algorithms that generate solutions to combinatorial optimization problems using schemes inspired by natural evolution, such as selection, crossover, mutation, and inheritance [6].

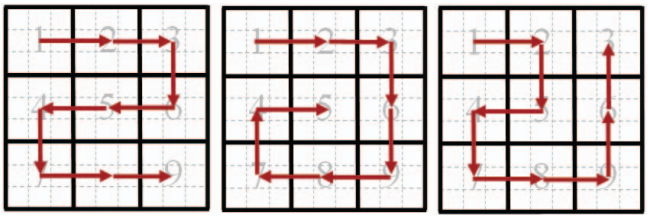
The *simulated annealing* based Sudoku solver is a probabilistic Sudoku solver. The general design is capable to solve a Sudoku instance of order up to 15. It has been claimed that the solver has solved in actual hardware Sudoku puzzles of order up to 12 within the competition imposed time limits [5].

III. The Proposed Sudoku Solver

All the above algorithms discussed in reviewing the existing literature, are guess based; hence, excessive redundant computation is involved and very much time consuming as

these are all cell based Sudoku solvers. 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 and no redundant computation is performed during the whole computation.

Our proposed algorithm considers each of the minigrids that may be identified as 1 through 9 as shown in Figure 1. Each minigrid may or may not have some clues as numbers that are given. We first consider the first minigrid and then find out the permutations among the missing numbers based on the clues present in the minigrids belongs to the same row and column. Then it moves to the next minigrid and go on considering the other minigrids in a definite sequence S_M of minigrids based on adjacency. There are several such possible sequences of S_M starting with a corner minigrid (number), say 1, and initially following a row-major order, as shown in Figure 8, or the reverse or another.



(a) (b) (c)

Figure 8: (a) A *zigzag* way of considering the minigrids starting with minigrid number 1, following a row-major sequence, and ending with minigrid number 9. Here the minigrid numbers followed are 1→2→3→6→5→4→7→8→9. **(b)** A *spiral* way of considering the minigrids starting with minigrid number 1, following a row-major sequence, and ending with minigrid number 5. Here the minigrid numbers followed are 1→2→3→6→9→8→7→4→5. **(c)** A *semi-spiral* way of considering the minigrids starting with minigrid number 1, following a partial row-major sequence, and ending with minigrid number 3. Here the minigrid numbers followed are 1→2→5→4→7→8→9→6→3.

Figure 8(a) shows a *zigzag* way of considering the minigrids starting with minigrid number 1, following a row-major sequence, and ending with minigrid number 9. Similarly, a zigzag way of considering the minigrids starting with minigrid number 1 (or from some other corner minigrid), following a column-major sequence (or row-major sequence), and ending with minigrid number 9 (or up to a corresponding opposite corner minigrid) can also be there. Figure 8(b) shows a *spiral* way of considering the minigrids starting with minigrid number 1, following a row-major sequence, and ending with minigrid number 5. A spiral way of considering the minigrids (or that may start with some other corner minigrid and following minigrids) in a column-major sequence (or row-major sequence), and always ending with minigrid number 5 can also be there. In a *semi-spiral* way of considering the minigrids starting with minigrid number 1, following a partial row-major sequence, and ending with minigrid number 3 is shown in Figure 8(c). In this case, a similar generalization can also be taken up starting from some other corner minigrid. A reverse sequence (or another) can also be adopted, not necessarily always starting with a corner minigrid, in computing S_M and followed accordingly.

A. Computationsof Valid Permutations of Minigrid

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 $1 \le i, j, k, l, m \le 9$.

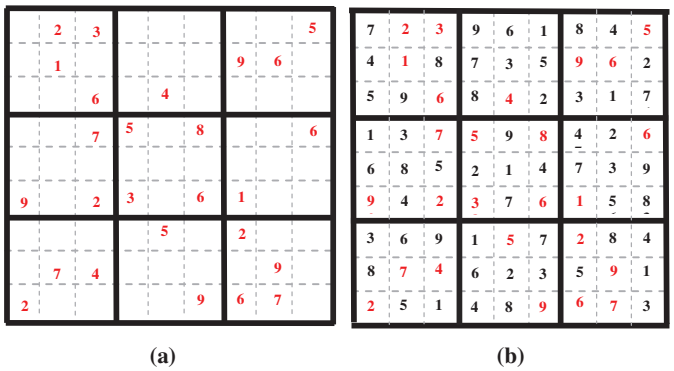
Now to start with a minigrid as stated above, we find that minigrid 1 contains only four clues, as shown in Figure 9(a). Here we denote a cell location of a Sudoku instance by [row number, column number], where each row number and column number vary from 1 to 9. Hence, the blank locations are [1, 1], [2, 1], [2, 3], [3, 1], and [3, 2], and the missing digits are 4, 5, 7, 8, and 9.

We compute all possible permutations of these missing digits in minigrid 1, 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 2, 3, 4, and 7). As for example, if we consider the last permutation 45789 and place the missing digits, respectively, in order in locations [1, 1], [2, 1], [2, 3], [3, 1], and [3, 2], which are arranged in ascending order, we find that this permutation is not a valid permutation. This is because location [4, 3] already contains 7 as a clue of minigrid 4; so we cannot place 7 at [2, 3] as the permutation suggests. Also we cannot place 9 at location [1, 1] as location [6, 1] 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 1, we construct a tree structure as shown in Figure 10. 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, 1], [2, 1], [2, 4], [3, 1], and [3, 2]. 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 1 of minigrid 6 contains 9 as given clue (at location [1, 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, 9] contains 5 as given clue. Up to this point in time, as either 8, or 7, or 4 could be placed at [1, 1], we expand each of the child vertices starting with permutations 8, and 7, and 4, as shown in Figure 10.



(a) (b)

Figure 9: (a) An instance of the Sudoku problem. (b) A solution of the Sudoku instance shown in Figure 9(a).

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, 3] as [2, 7] contains 9 as a 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, 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 1 of the Sudoku instance shown in Figure 9(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. 9(a)), out of these seven valid permutations only one will finally be accepted following the subsequent steps of the algorithm developed in the next section.

Now, in a similar fashion, the permutation generation algorithm computes all valid permutations of each of the subsequent minigrids based on the clues in an assumed minigrid along with the clues given in its row and column minigrids. Then for a valid permutation of minigrid 1, the Sudoku solver finds whether there is a valid permutation of minigrid 2. If it does not, a next valid permutation of minigrid 1 is considered and a valid permutation of minigrid 2 is searched out, if one exists; otherwise, a valid permutation of minigrid 3 is found out, which is supposed to be compatible with the assumed valid permutations of minigrids 1 and 2. This is how the algorithm proceeds, if a valid permutation of a new minigrid is found, whose inclusion in combination may consider a valid permutation of another minigrid; otherwise, we backtrack to consider another valid permutation (if any) of the prior minigrid, such that ultimately a combination of nine valid permutations of nine minigrids gives the solution of the given Sudoku puzzle P.

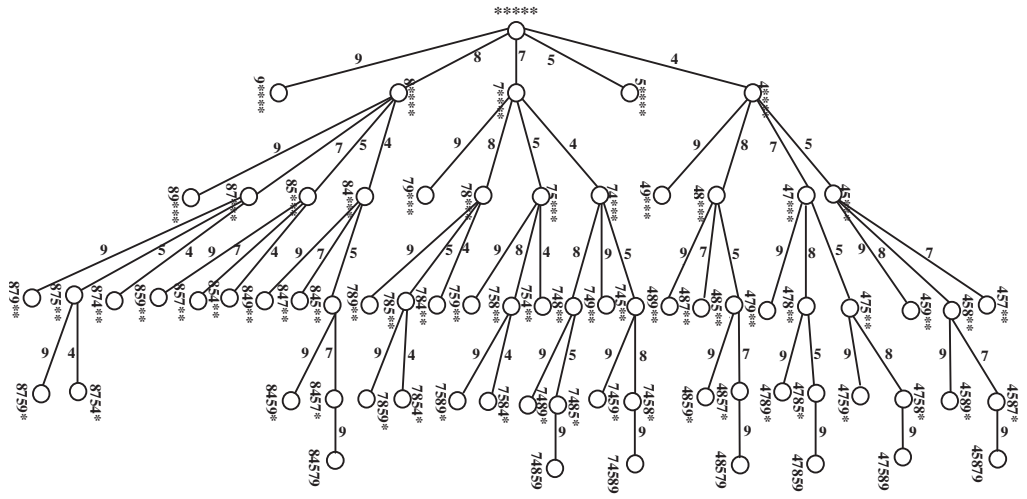


Figure 10: The permutation tree for generating only valid permutations of the missing digits in minigrid 1 of the Sudoku instance shown in Figure 9(a).

Here we separately generate individual sets of all valid permutations only for each of the minigrids obeying the given clues in the given Sudoku instance P. Let us assume that S_{Mi}, $1 \le i \le 9$, be the set of all valid permutations separately generated for minigrid i, only taking into consideration the given clues in P. To have a unique solution S of P, if it exists, we claim that S_{Mi} must include only one valid permutation for minigrid i such that in combination of all those nine valid permutations, one for each of the minigrids, S is assured to be computed. This claim has been stated in the following lemma.

Lemma 1: While computing individual valid permutations for each of the minigrids, obeying only the set of given clues in a Sudoku instance, at least a set of nine valid permutations must be there, one for each of the minigrids, whose combination certify a valid solution of the given Sudoku instance, if one exists.

Proof: Based on the given clues in a given Sudoku puzzle P, we may find that there are only four adjacent minigrids for each of the minigrids M_i, $1 \le i \le 9$, two in the row and two in the column, in P. So, clues in M_i inform about the missing digits in the ith minigrid and the four adjacent minigrids of M_i guide to generate only the valid set of permutations for M_i. If M_j, $1 \le j \le 9$ but $j \ne i$, is a minigrid either in the row or in the column of M_i, then at least one valid permutation of M_j must be there which is entirely well-matched to at least one valid permutation of M_i, if there is a solution of P. This should be true for each pair of minigrids, where M_i is a member (of the pair). So, for each minigrid M_i, four such pairs of minigrids are to be compared and matched. Hence, a total of ((9×4)÷2), or 18 number of pair-wise matching is necessary, and if P has a solution, such matching in each pair of (row-wise and column-wise available) minigrids is guaranteed to be established, and thus the lemma is concluded. ♦

B. The Algorithm

Here in this section, we like to see the algorithm at a glance that has been outlined in the previous section, as follows.

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

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

Step 1: Compute the 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 a minigrid (and the first member in S_M). The S_M can be Zigzag, Spiral or Semi-spiral as the case may be, either row-major or column-major.

Step 3: Compute all valid permutations of the missing digits in each of the nine minigrids based on the existing clues and store them.

Step 4: Now consider minigrid M (i.e., the first minigrid of the sequence S_M) and place the first valid permutation in the respective blank locations (in ascending order of the location).

Step 5: For all minigrids $N \in S_M$, do the following:

Consider a next minigrid, $N \in S_M$, and place the first valid permutation in the respective blank locations of this minigrid, and verify whether the permutation matches with the earlier minigrid(s).

> **Step 5.1:** If it matches with the permutation of earlier minigrids $M \in S_M$ in the same row and/or column, go on proceeding with the next minigrids in the sequence S_M and perform the same operation;

Otherwise,

> **Step 5.2:** Place a next valid permutation of the minigrid $N \in S_M$, and check whether it matches with the earlier minigrids in the same row and/or column. If it matches, go on proceeding with successor minigrids; otherwise, consider a next valid permutation of the predecessor minigrid in the same fashion.

Step 6: If all the valid permutations of the successor minigrids of N (including M) are exhausted to obtain a valid combination for all the nine minigrids in S_M, then consider a next valid permutation of the predecessor minigrid of N, and go to Step 5. The process is continued until a valid combination of nine distinct permutations for all the nine minigrids in S_M is obtained, row-wise and column-wise compatible to each other, as a desired solution S for P; otherwise, the algorithm declares that there is no valid solution for the given Sudoku instance P.

Here we may assume any sequence S_M of minigrids we like to, either zigzag, or spiral, or semi-spiral. Let us assume the zigzag way of considering the minigrids starting with minigrid number 1, following a row-major sequence, and ending with minigrid number 9, as shown in Figure 8(a). At the beginning of the algorithm, we have to compute all S_{Mi}'s, $1 \le i \le 9$, where S_{Mi} is the set of all valid permutations separately generated for minigrid i obeying only the given clues in the given Sudoku instance P. We like to start with the minigrid number 1; so we consider a valid permutation of minigrid 1. Based on this valid permutation of minigrid 1, we verify whether a valid permutation of minigrid 2 matches to it. If it does not match, we consider a second valid permutation of minigrid 2. If one valid permutation of minigrid 2 matches to that of minigrid 1, we go for searching a valid permutation of minigrid 3; otherwise, if there is no such valid permutation for minigrid 2, we consider a second valid permutation of minigrid 1, and so on.

In this process, when three valid permutations, one each for minigrid numbers 1, 2, and 3, are obtained matched to each other, we go for searching a valid permutation for minigrid number 6, and then we match its permutation with the already obtained valid permutation for minigrid number 3 (only for the time being, as minigrid number 3 belongs to the same column of minigrid number 6). If one such valid permutation for minigrid number 6 is obtained, we move to minigrid number 5 to search for its desired permutation; otherwise, a new set of valid permutations for minigrid numbers 1, 2, and 3 might

need to be explored (not necessarily all new but that would again be matched to each other).

While finding out the necessary valid permutation for minigrid number 5, we are supposed to match a generated permutation of this minigrid with the already identified permutations for minigrid numbers 2 and 6 (as minigrid number 5 belongs to the same column of minigrid number 2 and to the same row of minigrid number 6, up to this point in time). If one such valid permutation of minigrid number 5 is obtained, we move to minigrid number 4 to search for its desired permutation; otherwise, a new set of valid permutations for minigrid numbers 1, 2, 3, and 6 might need to be discovered (where minigrid-wise all the permutations may not be a new one).

In the similar way, to detect the desired valid permutation for minigrid number 4, we are supposed to match a generated permutation of this minigrid with the already identified permutations for minigrid numbers 1, 5, and 6 (as minigrid number 1 belongs to the same column of minigrid number 4 and to the same row of minigrid numbers 5 and 6). This is how the algorithm progresses for exploring desired permutations for the remaining minigrids, and if one is obtained for a minigrid, we move forward following the (assumed) sequence of minigrids in S_M; otherwise, we are supposed to retrace our steps to the previous minigrid if there is any remaining valid permutation that may match and move forward again; or else go back to search for a matched permutation from the remaining valid permutations (if any) of a previous minigrid (currently again under consideration). As soon as a valid permutation that matches for a minigrid is obtained, we move forward; otherwise, we move backward to search for a desired permutation from the remaining set of valid permutations (if any) of some earlier minigrid (following the reverse sequence in S_M), which is at present again under consideration.

In this way of judging compatibility of a permutation for some minigrid with the already identified permutations of its earlier minigrids eventually provides a solution S for a given Sudoku instance P, if one exists, and as we approach towards the end of the sequence in S_M, the matched permutation checking process between the minigrids becomes faster (as options for matching is reduced, or the selection of a desired permutation for the current minigrid gets more guided by already identified permutation(s) of other earlier row and column minigrid(s) of the current minigrid). Needless to mention that when we consider a permutation for minigrid number 7, we are supposed to check it with the already identified permutations of minigrid numbers 1 and 4 (only along the column), while doing the same for minigrid number 8, we are supposed to check its permutation with the already identified permutations of minigrid numbers 2 and 5 along the second column, and minigrid number 7 along the third row. For the remaining minigrid, i.e., for minigrid number 9, we are supposed to consider the already identified permutations for minigrid numbers 3 and 6 along the third column, and minigrid numbers 7 and 8 along the third row of P, if pair-wise they all match with a permutation of minigrid number 9. This is how, following the algorithm developed in this paper, we obtain the valid solution shown in Figure 9(b) for the Sudoku instance shown in Figure 9(a).

C. Computational Complexity of the Algorithm

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 for computing all possible permutations of a minigrid based on the 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. Hence we conclude the computational complexity of the algorithm in the following theorem.

Theorem 1: The Sudoku solver developed in this paper guarantees a valid solution of a Sudoku instance P, if one exists, and it takes time and space complexities $O(np^n)$, where p is the number of blank cells in a minigrid of P of size $n \times n$.

Proof: The 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 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. ♦

IV. Conclusion

In this paper we have developed an exclusive minigrid based Sudoku solver, 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.

References

[1] A. C. Clarke, Ascent to Orbit: A Scientific Autobiography, John Wiley & Sons, New York, 1984.

[2] H. Intelm, How to Solve Every Sudoku Puzzle, Vol.2, Geostar Publishing LLC. 2005.

[3] N. Jussien, A-Z of Sudoku, ISTE Ltd., USA, 2007.

[4] W.-M. Lee, Programming Sudoku, Apress, USA, 2006.

[5] P. Malakonakis, M. Smerdis, E. Sotiriades, and A. Dollas, "An FPGA based Sudoku Solver based on Simulated Annealing Methods", In: Proc. Int. Conf. Field-Programmable Technology, 2009, pp. 522-525.

[6] T. Mantere, and J. Koljonen, "Solving, Rating and Generating Sudoku Puzzles with GA", In: Proc. IEEE Congress on Evolutionary Computation, 2007, pp. 1382-1389.

[7] J. A Pacurib., G. M. M. Seno, and J. P. T. Yusiong, "Solving Sudoku Puzzles using Improved Artificial Bee Colony Algorithm", In: Proc. 4th Int. Conf. Innovative Computing, Information and Control, 2009, pp. 885-888.

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