# Chapter 3

# A New Sudoku Solver

## Overview

The problem being addressed here can be stated as to develop a new Sudoku solver which is absolutely guessed free. 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 3.1(a) and its solution is shown in Figure 3.1(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 | | |

**(a)**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

**(b)**

**Figure 3.1: (a)** An instance of the Sudoku problem. **(b)** A solution of the Sudoku instance shown in Figure 3.1(a).

As already mentioned, solving an instance of Sudoku problem is NP-complete [10]. So it is unlikely to develop a deterministic polynomial time algorithm for solving a Sudoku instance 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 [14,15,17].

All the earlier existing Sudoku solvers that are available in literature (and Internet) are entirely guess based and hence extremely time consuming. 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 thesis 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.

It has already been told that our approach developed in this thesis does not differentiate the instances, rather our approach computes a solution if it exists without guessing a possible value in a blank location and primarily minigrid based irredundant deterministic computations are involved over there. Later on, inter-minigrid valid combinations are computed in some fashion, either following spiral, or zigzag, or semi-spiral sequence of considering all the minigrids to compute a desired solution. At the end, in developing the algorithm, we have generalized the approach based on a graph theoretic formulation wherein we have proved that the modified comprehensive innovation is able to generate all valid solutions of a given Sudoku instance, if there be two or more such solutions.

As already mentioned, the existing works on developing algorithms are exclusively guess based and each of them moves towards a solution considering a blank location of the given Sudoku instance at a time. In this context, the algorithm for solving a Sudoku instance that has been developed in this thesis is free from any guess of a probable number for a blank position; rather it computes all valid solutions individually for each of the minigrids under consideration. The algorithm consists of four versions in a natural sequence of its development. In the first version (i.e. Version 3.1), minigrids are considered based on the number of given clues. In the second version (i.e. Version 3.2), minigrids are considered in a succession, either following spiral, or zigzag, or semi-spiral run for their probable valid combinations such that eventually a desired solution of the Sudoku instance is accomplished. These versions of the algorithm are included in this chapter of the thesis.

In the third version (i.e. Version 4.1), we consider all individual valid solutions of all the minigrids and with the help of a graph theoretic formulation among the individual valid solutions of the minigrids we generate all valid solutions (one or more) of the given Sudoku instance, or declare that the instance does not have any valid solution. In the fourth version (i.e. Version 4.2) of the proposed algorithm, we have further compacted the graph structure. This is the first attempt in developing a Sudoku solver, which is novel in many ways; for example, this algorithmic approach is completely guessed free where minigrids are considered instead of individual blank cells and secondly it computes all valid solutions of a given Sudoku instance, if the solution is not unique, and so on and so forth. These versions of the algorithm have been included in Chapter 4 of this thesis.

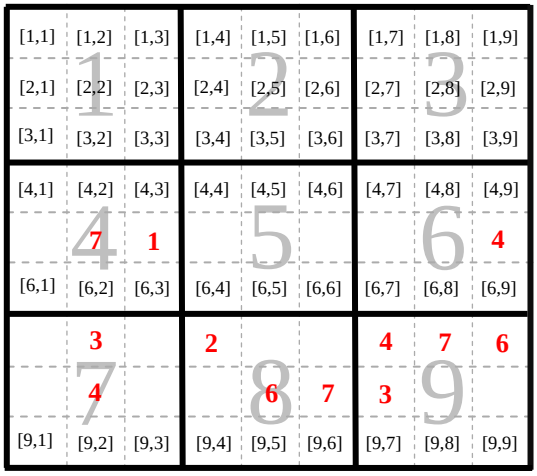



**Figure 3.2:** The structure of a 9×9 Sudoku puzzle (problem) with its nine minigrids of size 3×3 each as numbered (in grey outsized font) 1 through 9.

## 3.1 Version 3.1 of the New Guessed Free Sudoku Solving Algorithm

In this section we like to develop the first version of the algorithm, a new Sudoku solver, which is absolutely guessed free. The algorithm considers each of the nine minigrids 1 through 9 as shown in Figure 3.2. Minigrid 1 comprises cells [1,1], [1,2], [1,3], [2,1], [2,2], [2,3], [3,1], [3,2], and [3,3], minigrid 2 comprises cells [1,4], [1,5], [1,6], [2,4], [2,5], [2,6], [3,4], [3,5], and [3,6], and so on. In a similar way, minigrid 9 comprises cells [7,7], [7,8], [7,9], [8,7], [8,8], [8,9], [9,7], [9,8], and [9,9]. Each minigrid may or may not have some clues as digits / numbers that are given. In Figure 3.2, clues are represented in red colour. In this figure we may observe that each minigrid has exactly four *adjacent* minigrids along its row and column. As for example, minigrid

6 has its adjacent minigrids 3, 4, 5, and 9. Similarly, minigrid 7 has its adjacent minigrids 1, 4, 8, and 9, and so on.

We like 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, as also shown in Figure 3.2, 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 $1 \leq i, j, k, l, m \leq 9$. Moreover, each minigrid may or may not have some clues as numbers (or characters) 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.

Now to start with a minigrid as stated above, we find that the minigrid 3 contains a maximum number of clues, i.e. four, among all the minigrids, and each of the minigrids 1 and 2 contains less number of clues than that of minigrid 3 (see Figure 3.1(a)). For example, for the Sudoku instance as shown in Figure 3.1(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 locations 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 3.1(a) are: 9 in cell [1,8], 8 in cell [1,9], 1 in cell [2,8], and 5 in cell [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 cells 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 initial version of the 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 cells [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 cell [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 cell [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 belonging to 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 there is a solution of the given Sudoku instance. 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 3.1(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 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 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 3.1(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 (or next) 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 $t^{th}$ minigrid a valid permutation for the $(t+1)^{th}$ minigrid is obtained, we consider a second (or next) 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 following theorem if at least one solution of the given Sudoku puzzle exists.

**Theorem 3.1:** There is at least one valid permutation for the missing digits for their respective blank cells 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 3.1

**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 that are in the same row and column minigrid of N, 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 3.1(a). According to this instance the sequence $S_M$ of minigrids is $\langle 3, 1, 7, 6, 5, 9, 4, 8, 2 \rangle$ as it has been described and performed in Step 2 of the first version of the algorithm above.

We have already claimed in Theorem 3.1 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 now compute the computational complexity of the algorithm (Version 3.1) developed herein and claim the result in the form of the following theorem, either a solution is there or it is not there.

**Theorem 3.2:** Version 3.1 of the Sudoku solving algorithm developed in this thesis guarantees a valid solution of a Sudoku instance, if one exists, and it takes both time and space complexity $O((p!)^{\alpha n})$, where $p$ is the average number of blank cells in a minigrid of a Sudoku instance of size $n \times n$, and $\alpha$ is some positive constant less than or equal to one.

**Proof:** This solver 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 solving algorithm 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 (or countable) 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 complexity $O((p!)^n)$, as generation of all valid permutations of a minigrid is the dominating computations involved in this algorithm, where $p$ is the average number of blank cells in a minigrid of P of size $n \times n$. Here $(p!)^n$ is the size of the tree structures we compute while producing all valid permutations of the missing digits in a minigrid (in the worst case). In practice, the size of the tree structure (for each minigrid) is significantly less than the asymptotic upper bound mentioned herein, and as a result the overall time and space complexity could be mentioned as $O(p!(p!)^{\alpha n}) = O((p!)^{\alpha n+1}) = O((p!)^{\alpha n})$ for all minigrids of a given Sudoku puzzle, where $\alpha$ is a constant and $0 < \alpha \leq 1$. ♦

## 3.2 Permutation Generation Technique

Permutation is a way of ordering of a set of objects or symbols where each object occurs exactly once. As an example, there are six permutations of the set of numbers {1,2,3}, namely (1,2,3),

(1,3,2), (2,1,3), (2,3,1), (3,1,2), and (3,2,1). The number of permutations of $n$ distinct objects can be calculated as $n \times (n-1) \times (n-2) \times ... ... \times 2 \times 1$, which is essentially read as "factorial of $n$" and written as "$n$!". In computing, it may be required to generate permutations of a given sequence of values. The notion of permutation is used in the following contexts: (1) Group theory, (2) Combinatorics, etc. There are several permutation generation techniques, some of which are described in the following sections.

### 3.2.1 A Brief Review on Existing Permutation Generation Algorithms

In this section, we discuss on different permutation generation algorithms. There are several permutation generation techniques available in literature. Among many others, now we briefly talk about Random Permutation Generation Algorithm [38], Fisher-Yates Shuffle [39], and Permutation Tree [40] in the next three sections.

#### 3.2.1.1 Random Permutation Generation Algorithm

We can generate a random permutation of $k$ distinct objects in a computer, to order a given list randomly [38]. Permutation distribution is supposed to be unbiased so that each of the $k!$ permutations are generated with equal probability. Here a permutation has generated from a list of $k$ distinct random numbers $a_1, a_2, ..., a_k$. We can perform random permutation from this list by moving each number $a_i$ to position $i$, where $1 \leq i \leq k$ [38].

The naive way of generating such a list is as follows:

**Step 1:** Generate a trial random number $r$ from 1 to $k$.

**Step 2:** Check whether $r$ is already in the list. If yes, then go back to Step 1.

**Step 3:** Else, add $r$ to the list.

**Step 4:** If fewer than $k$ numbers have been added to the list, go back to Step 1.

But the main drawback of this algorithm is that the algorithm is not efficient. The time required to check whether each trial number is already in the list is of order $k$. Again, if nearly $k$ numbers have already been added, then the number of trial random numbers required, before one is found that is not already in the list, is also of order $k$. Thus, $O(k^2)$ is the overall running time of the algorithm briefed above.

#### 3.2.1.2 Fisher-Yates Shuffle

There is another popular way of generating a random permutation, which is called *Fisher-Yates shuffle* [39]. It is an iterative algorithm used for randomising a set of numbers. It is a paper and pen based method. Steps of this algorithm are as follows:

**Step 1:** Write down the numbers from 1 to $N$ in scratch.

**Step 2:** Pick a random number $k$ between one and the number of unstuck numbers remaining in the scratch. Store the value of $k$ in Roll.

**Step 3:** Counting from the low end, select and strike out the $k$th number from the scratch, if not yet struck out, and write it down elsewhere.

**Step 4:** Repeat from Step 2 until all the numbers have been struck out.

**Step 5:** The sequence of numbers written down in Step 3 is now a random permutation of the original numbers.

In 1964, a modern version of the Fisher-Yates shuffle, designed for computer use, was proposed by Durstenfeld [41]. The algorithm described by Durstenfeld differs from that given by Fisher and Yates in a small but significant way. This algorithm shuffles an array of any type.

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

#### 3.2.1.3 Permutation Tree

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

As there are different techniques for generating permutations available in literature, so for different types of such techniques different permutation trees can be constructed [40]. Some of them have been described below.

In 1989, Arnow developed an attractive method to generate the random permutations of $n$ elements [42]. He has inserted $n$ elements in $n$ successive levels of the tree, and for an integer $k \leq$

*n*, all permutations of the first *k* elements are obtained at the *k*th level of the tree. More specifically, the root of the tree represents the only permutation comprising the first element. All other successive levels have been formed by inserting the successive elements in all positions (extreme and in between) one after another (as child nodes of a smaller parent permutation). Hence for each of the nodes at the *k*th level must have exactly *k*+1 child nodes. As an example, the tree structure for $n = 3$ has been shown in Fig. 3.3, where the elements are 1, 2, and 3. Here 1 is the root of the tree. Then the child nodes have been formed by inserting 2 before 1 and 2 after 1. Similarly, the third level is formed by inserting 3 in three possible positions in each of 21 and 12, and we obtain all six permutations comprising the elements 1, 2, and 3.

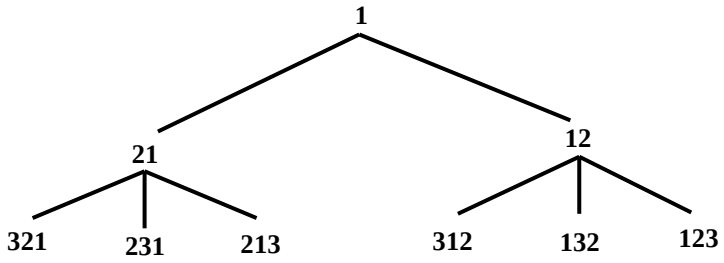



**Figure 3.3:** The permutation tree for three natural numbers 1, 2, and 3 following the algorithm of Arnow [42].

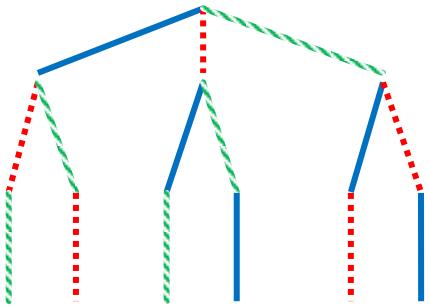


**Figure 3.4:** The permutation tree for three distinct colours following the algorithm of Latif [43].

In 2004, Latif developed another way of constructing permutation tree [43]. He used a random permutation technique for generating the permutations and by considering these permutations he constructs a permutation tree model. In his proposed structure the edges are represented by the elements to be permutated and he used different colours for each of them. Following an edge in the permutation tree is equivalent to selecting an item, and permutations are formed by following paths from the root to the leaves. Figure 3.4 shows a sample tree structure for three distinct elements represented by three different colours.

A permutation tree of $n$ items is generated recursively. At the root of the tree, all $n$ items are available for selection, so the root has $n$ edges of different colours leading to $n$ subtrees. At each of the subtrees of the root has only $n$–1 items available (as one item has already been selected), so each of the subtrees of the root has $n$–1 edges leading to respective leaf vertices. The colour of a missing edge corresponds to the item that has been selected now following the path from the root of the tree to the root of the subtree (see Figure 3.4). This process is repeated with the subtrees of the subtrees, and so on, until the leaves are reached. At this point all items have been selected; therefore, no edges remaining to be considered and to be continued the process further. This is how a path from the root of the tree to a leaf node provides a permutation of the elements under consideration.

### 3.2.2 A Novel Permutation Generation Technique

Computation of all valid permutations for the missing decimal digits (or numbers / symbols) in a minigrid of a 9×9 Sudoku puzzle P is an important task of the proposed 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 the Sudoku instance in Figure 3.1(a) is four, 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 3.5, 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 a 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 3.5.

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 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 2 as a given clue. So, this is how either a valid permutation is generated from the root of the tree 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 3.5 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 3.1(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 3.1(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.

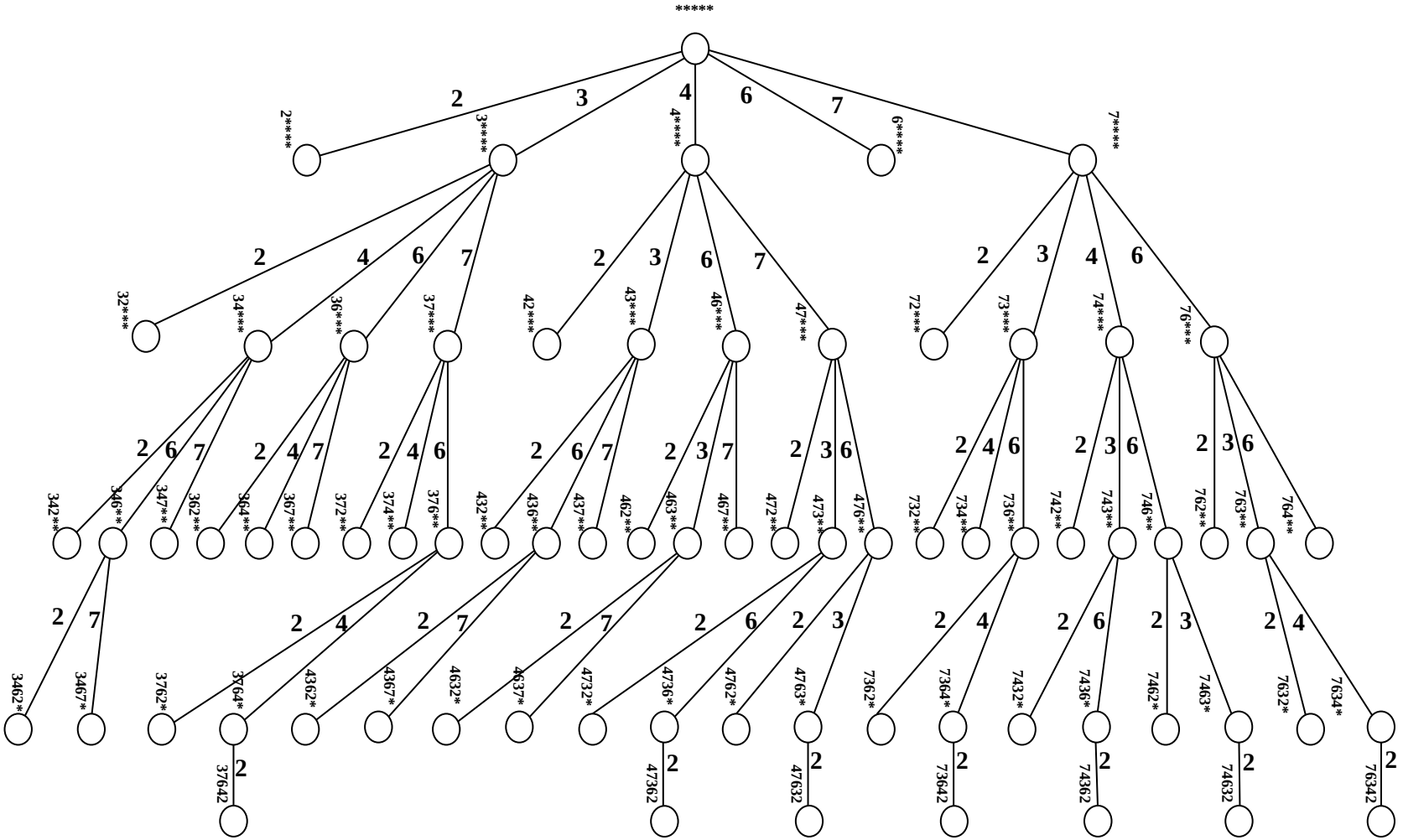



**Figure 3.5:** The permutation tree for generating only valid permutations of the missing digits in minigrid 3 of the Sudoku instance shown in Figure 3.1(a)

This is how the algorithm proceeds and generates all valid permutations of a minigrid under consideration obeying the given clues in P and a set of assumed valid permutations, one for each of the minigrids considered earlier in succession, up to this point in time.

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

Now we discuss about the size of the tree structure under consideration. If $p$ be the average number of blank cells in a minigrid and the Sudoku instance is of size $n \times n$, then the computational time as well as the computational space complexity of the guessed free Sudoku solver developed in this thesis 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.

Now we develop the next versions of the algorithm, which is a generalization of version 3.1 of the Sudoku solving algorithm, discussed in Section 3.1.

## 3.3 Version 3.2 of the New Guessed Free Sudoku Solving Algorithm

In the initial version of the algorithm to develop a guessed free Sudoku solver, we computed $S_M$, a sequence of minigrids that we used to follow the minigrids one after another and accordingly we generated valid permutations of a successive minigrid. There the valid permutations we generated for a subsequent minigrid were based on one valid permutation of each of the earlier

minigrids in $S_M$ along with the given clues in the specified Sudoku instance P. This generalized version of the guessed free Sudoku solver is different from the earlier one in two ways. Firstly, it follows a defined sequence $S_M$ of minigrids based on adjacency (instead of following a discrete sequence $S_M$ of minigrids), and secondly, it does not compute a set of valid permutations of a minigrid based on valid permutation(s) of some prior minigrid(s). We explicitly elucidate the alterations as follows.

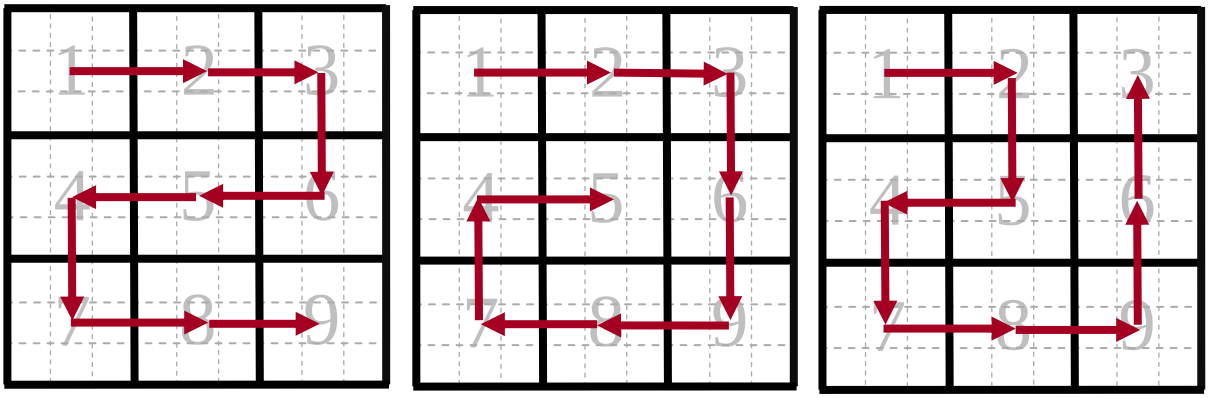


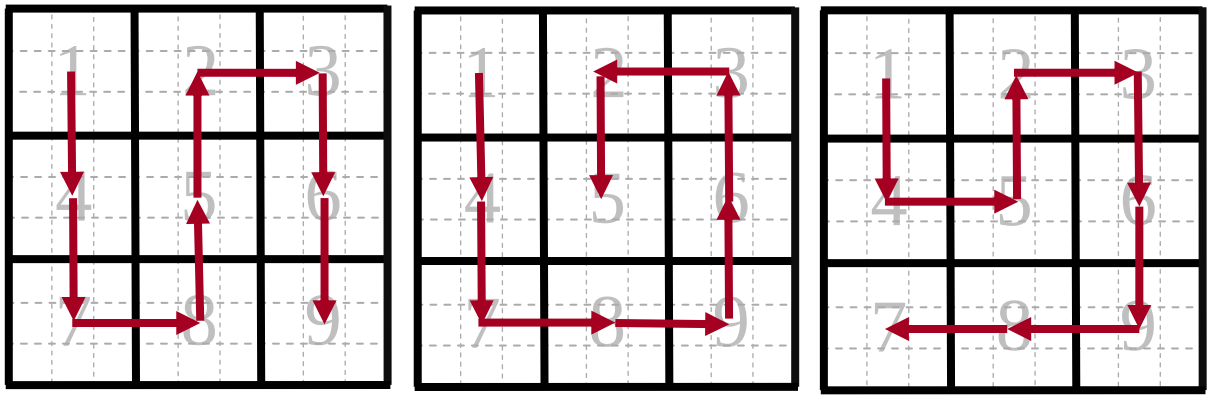

**Figure 3.6: (a)** A *zigzag* way of considering the minigrids starting with minigrid 1, following a row-major sequence, and ending with minigrid 9. **(b)** A *spiral* way of considering the minigrids starting with minigrid 1, following a row-major sequence, and ending with minigrid 5. **(c)** A *semi-spiral* way of considering the minigrids starting with minigrid 1, following a partial row-major sequence, and ending with minigrid 3.

We know how we computed $S_M$ in our earlier version of the Sudoku solver. There, two consecutive members in $S_M$ may not be physically consecutive (in terms of adjacency) of the minigrids. As for example, for the given Sudoku instance in Figure 3.1(a) and the way we numbered the minigrids as shown in Figure 3.2, minigrid 1 is not adjacent to minigrid 3, minigrid 7 is not adjacent to minigrid 1, minigrid 6 is not adjacent to minigrid 7, and so on and so forth, as these minigrids (numbers) are there in $S_M$ for this Sudoku instance. In this generalized version of the Sudoku solver, we consider an $S_M$ wherein each pair of consecutive minigrids is also adjacent to each other. There are several such possible sequences of $S_M$ starting with a corner minigrid, say 1, and initially following a row-major order, as shown in Figure 3.6, or the reverse or another.

Figure 3.6(a) shows a *zigzag* way of considering the minigrids starting with minigrid 1, following a row-major sequence, and ending with minigrid 9. Here the minigrids followed are 1→2→3→6→5→4→7→8→9. Similarly, a zigzag way of considering the minigrids starting with minigrid 1 (or from some other corner minigrid), following a column-major sequence (or row-major sequence), and ending with minigrid 9 (or up to a corresponding opposite corner minigrid) can also be there. Figure 3.7(a) shows a *zigzag* way of considering minigrids in column-major sequence. Here the minigrids followed are 1→4→7→8→5→2→3→6→9. Figure 3.6(b) shows a *spiral* way of considering the minigrids starting with minigrid 1, following a row-major sequence, and ending with minigrid 5. Here the minigrids followed are 1→2→3→6→9→ 8→7→4→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. Figure 3.7(b) shows a *Spiral* way of considering minigrids in column-major sequence. Here the minigrids followed are 1→4→7→8 →9→6→3→2→5. In a *semi-spiral* way of considering the minigrids starting with minigrid 1, following a partial row-major sequence, and ending with minigrid 3 is shown in Figure 3.6(c). Here the minigrids followed are 1→2→5→4→7→8→9→6→3. Instead of row major sequence, we can also follow a column-major sequence during the selection of sequence of minigrids, as shown in Figure 3.7(c). Here the minigrids followed are 1→4→5→2→3→6→9→8→7.

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.

Now we like to state the second variation of this version of the algorithm over the earlier one. 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 (and irrespective of allowing for *valid* permutation(s) of the minigrid(s) for its (their) blank locations considered earlier in $S_M$). Let us assume that $S_{Mi}$, $1 \leq i \leq 9$, 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, if it exists. This claim has been stated in the following lemma.




**(a)** **(b)** **(c)**

**Figure 3.7: (a)** A *zigzag* way of considering the minigrids starting with minigrid 1, following a column-major sequence, and ending with minigrid 9. **(b)** A *spiral* way of considering the minigrids starting with minigrid 1, following a column-major sequence, and ending with minigrid 5. **(c)** A *semi-spiral* way of considering the minigrids starting with minigrid 1, following a partial column-major sequence, and ending with minigrid 7.

**Lemma 3.1:** 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 $M_i$, $1 \leq i \leq 9$, two in the row and two in the column, in P. So, clues in $M_i$ inform about the missing digits in the $i$th minigrid and the four adjacent minigrids of $M_i$ guide to generate only the valid set of permutations for $M_i$. If $M_j$, $1 \leq j \leq 9$ but $j \neq i$, 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, if P has a valid solution, and thus the lemma is concluded. ♦

Now it is worthwhile to state that this generalized version of the Sudoku solver never generates redundant permutations for the blank locations in a minigrid. This is because a so-called valid permutation generated for a subsequent minigrid may not be a valid permutation, as a permutation we assumed as valid for some of its earlier minigrid(s) may not lead to an appropriate permutation at the end of the computation. So, such permutations, that are so-called valid permutations but do not lead to compute a solution for a given Sudoku instance P, may direct to generate a huge number of redundant (valid) permutations for subsequent minigrids, which are not in fact acceptable. The earlier version of the algorithm has several advantages but is lacking in that respect, where this generalized version of the algorithm overrules.

### 3.3.1 The Algorithm (Version 3.2) at a Glance

Here in this section, we like to see the algorithm at a glance that has been devised and 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 to the earlier minigrid(s).

**Step 5.1:** If it matches to 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 to 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, if one exists.

**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 the first minigrid (i.e. minigrid 1), following a row-major sequence, and ending with minigrid 9, as shown in Figure 3.6 (a). At the beginning of the algorithm, we have to compute all $S_{Mi}$'s, $1 \leq i \leq 9$, 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 minigrid 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 minigrids 1, 2, and 3, are obtained matched to each other, we go for searching a valid permutation for minigrid 6, and then we match its permutation to the already obtained valid permutation for minigrid 3 (only for the time

being, as minigrid 3 belongs to the same column of minigrid 6). If one such valid permutation for minigrid 6 is obtained, we move to minigrid 5 to search for its desired permutation; otherwise, a new set of valid permutations for minigrids 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 5, we are supposed to match a generated permutation of this minigrid to the already identified permutations for minigrids 2 and 6 (as minigrid 5 belongs to the same column of minigrid 2 and to the same row of minigrid 6, up to this point in time). If one such valid permutation of minigrid 5 is obtained, we move to minigrid 4 to search for its desired permutation; otherwise, a new set of valid permutations for minigrids 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 4, we are supposed to match a generated permutation of this minigrid to the already identified permutations for minigrids 1, 5, and 6 (as minigrid 1 belongs to the same column of minigrid 4 and to the same row of minigrids 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 to 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). It is straightforward to mention that when we consider a permutation for minigrid 7, we are supposed to check it with the already identified permutations of minigrids 1 and 4 (only along the column), while doing the same for minigrid 8, we are supposed to check its permutation with the already identified permutations of minigrids 2 and 5 along the second column (of minigrids), and minigrid 7 along the third row (of minigrids). For the remaining minigrid, i.e. for minigrid 9, we are supposed to consider the already identified permutations for minigrids 3 and 6 along the third column (of minigrids), and minigrids 7 and 8 along the third row (of minigrids) of P, if pair-wise they all match to a permutation of minigrid 9.

### 3.3.2 A Brief Comparison between Version 3.1 and Version 3.2 of the Algorithm

We now briefly state the variation of this version of the algorithm over the earlier one. The primary innovation of the initial version of the algorithm is that it starts generating valid permutations with a minigrid having more given clues. As a result, the number of valid permutations generated is much less. Just for an example, this could be verified for generating valid permutations for minigrid 3 of the Sudoku puzzle shown in Figure 3.1(a), for which out of 120 possible permutations only seven are found as valid permutations, as these have been generated using the tree structure shown in Figure 3.5. Subsequent consideration of minigrids in $S_M$ is also wisely logical. The key limitation of that (earlier) version of the Sudoku solver is to generate a vast number of redundant permutations based on some deceptive permutation(s) (that we generated and considered as valid permutation(s)) for some prior minigrid(s) in $S_M$.

In contrast to the earlier one, here we may assume any sequence $S_M$ of minigrids we like to, either zigzag, or spiral, or semi-spiral. Let us assume the semi-spiral way of considering the minigrids starting with minigrid 1, following a column-major sequence, and ending with minigrid 7, as shown in Figure 3.7(c). At the beginning of the algorithm, we have to compute all $S_{Mi}$'s, $1 \leq i \leq 9$, 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 minigrid 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 4 matches to it. If it does not match, we consider a second valid permutation of minigrid 4. If one valid permutation of minigrid 4 matches to that of minigrid 1, we go for searching a valid permutation of minigrid 5; otherwise, if there is no

such valid permutation for minigrid 4, we consider a second valid permutation of minigrid 1, and so on.

In this process, when three (assumed) valid permutations, one each for minigrids 1, 4, and 5, are obtained that match pairwise in sequence (permutations for minigrids 1 and 4 as column minigrids, and permutations for minigrids 4 and 5 as row minigrids), we go for searching a valid permutation for minigrid 2, and match its permutation with the already assumed valid permutation for minigrid 1 (as row minigrids) and the assumed valid permutation for minigrid 5 (as column minigrids) only for the time being. If one such valid permutation for minigrid 2 is obtained, we move to minigrid 3 to search for its desired permutation; otherwise, a new set of valid permutations for minigrids 5, 4, and 1 might need to be explored in reverse order, if the permutations are exhausted for the said minigrids (not necessarily all new but that would again be matched pairwise in succession, as all these minigrids are not belonging to the same row or column).

While finding out the necessary valid permutation for minigrid 3, we are supposed to match an assumed valid permutation of this minigrid with the already identified (or assumed) permutations for minigrids 1 and 2 (as minigrids 1, 2, and 3 row minigrids), up to this point in time. If one such valid permutation of minigrid 3 is obtained, we move to minigrid 6 to search for its desired permutation; otherwise, a new set of valid permutations for minigrids 1, 4, 5, and 2 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 6, we are supposed to match a generated permutation of this minigrid with the already identified permutations for minigrids 4, 5, and 3 (as minigrid 3 belongs to the same column of minigrid 6 and to the same row of minigrids 4 and 5). 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). It is easy to observe that when we consider a permutation for minigrid 9 (after minigrid 6), we are supposed to check its (assumed) permutation with the already identified permutations of minigrids 3 and 6 (only along the column), while doing the same for minigrid 8, we are supposed to check its permutation with the already identified permutations of minigrids 2 and 5 along the second column (of minigrids), and minigrid 9 along the third row (of minigrids). For the remaining minigrid, i.e. for minigrid 7, we are supposed to consider the already identified permutations for minigrids 1 and 4 along the first column (of minigrids), and minigrids 8 and 9 along the third row (of minigrids) of P, if pair-wise they all match with a permutation of minigrid 7.

We can visualize the validation of the algorithm in computing a solution S for some given Sudoku instance P, where we formulate this algorithm in the form of a simple, symmetric, connected graph. Nevertheless, the algorithm generates all such solutions S for some given P, if the solution is not unique. We may observe that if there are more clues in P, the instance is more constrained (or more guided) and the generation of valid permutations for each of the individual minigrids is less, and the computation of S is relatively simpler towards computing, hopefully, a unique solution for the given P. On the other hand, less clues in a given P means more flexibility in producing the permutations, and eventually more valid permutations for the minigrids are generated that may produce two or more pertinent solutions for the given P. This graph based visualization of the problem towards computing a valid Sudoku solution is devised in Chapter 4 of this thesis. Undeniably, this version (i.e. Version 4.1) of the algorithm is greatly generalized as it is capable of computing all solutions of a given Sudoku instance, if it has two or more solutions, or declares the given instance as invalid, if there is no valid solution of the given Sudoku instance.

### 3.3.3 Computational Complexity of the Algorithm

If $p$ be the average 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 valid permutations of a minigrid based on the Sudoku solver (i.e. Version 3.2 of the algorithm) developed in this chapter 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 (or CPU) time, of the order of milliseconds (or less). Of course, from complexity point of view, this version (i.e. Version 3.2) of the algorithm takes no more time and space than that required for executing Version 3.1 of the algorithm; we conclude the computational (time and space) complexity of Version 3.2 of the algorithm in the following theorem.

**Theorem 3.3:** Version 3.2 of the Sudoku solving algorithm guarantees a valid solution of a Sudoku instance P, if one exists, and it takes time and space complexity $O((p!)^{\alpha n})$, where $p$ is the average number of blank cells in a minigrid of P of size $n \times n$, and $\alpha$ is some positive constant less than or equal to one.

**Proof:** This proof is very similar to that of the theorem for computing computational complexity of the first version of the algorithm, which is performed in Theorem 3.2. In that computation, there were plenty of redundant generation of permutations of some subsequent minigrid based on consideration of an invalid permutation of some (prior) minigrid. In this second version of the algorithm, no such redundant computation is involved; rather, here we compute only valid sets of permutations for all the minigrids in isolation based on the given clues in a Sudoku puzzle. This version of the algorithm starts from a valid permutation of some assumed minigrid and in view of this valid permutation as a desired one it searches for a matched permutation among the computed valid permutations of an adjacent minigrid following a defined sequence of minigrids, either zigzag, or spiral, or semi-spiral. The algorithmic technique that is followed in this version of the algorithm is based on backtracking, but instead of individual blank cells here we do the same minigrid-wise.

Now computation of all valid permutations of a minigrid takes time and space $O(n \times p!)$, as generation of all valid permutations of a minigrid is the dominating computation involved in this algorithm, where $p$ is the average number of blank cells in a minigrid of P of size $n \times n$. Moreover, $O(n^2 p!)$ is the size of the tree structure we compute at the time of generating all valid permutations of the missing digits in all $n$ minigrid (in the worst case). Now for computing pairwise compatible valid permutations among the row and column minigrids in isolation, the total time and space required is $O((p!)^n)$. However, as we have seen in reality that the number of permutations computed in the worst case is appreciably less than the actual number of permutations possible, so this complexity could be written as $O((p!)^{\alpha n})$ for the given Sudoku puzzle, where $\alpha$ is a positive constant not more than 1, and thus, the worst case time and space complexity of this version (i.e., Version 3.2) of the algorithm is $O((p!)^{\alpha n})$. ♦

Nevertheless, whatever be the size of the permutation tree for each minigrid in a given Sudoku puzzle P, from the asymptotic upper bound point of view we may note that the aforesaid complexity is $O(n \times p!)$ for both the parameters stated above and also for both the versions of the algorithm. Furthermore, this complexity is exponentially bounded (and not polynomial time computable); so, as if it is not feasible in computing a desired (or optimal) solution of a given Sudoku instance in reasonable amount of (CPU) time. Incidentally, we may execute an exponential time algorithm for computing solutions of some optimization problem, which is NP-complete, if the size of the problem instance is small. In some other words, we may accept an exhaustive algorithm as a probable way-out towards computing a desired solution, when the input(s) of the variable(s) involved in the problem is (are) bounded by some constant.

Now it is very clear that the Sudoku instances considered in this thesis are all 9×9. That means the value of $n$ is always 9 and the value of $p$ is also at most 9. Whatsoever, even for some larger instances, where $n$ = 16, or 25, or 36, or more, $n$ is somehow bounded by some constant. So, the versions of the Sudoku solver developed in this thesis could safely be executed for each of the said larger instances as well. Experimental results may show some interesting data relating to the average CPU time needed in computing such larger instances.

Now it is interesting to note that both the versions of the algorithm is able to compute a solution of a given Sudoku instance P, if at least one (valid) solution is there for P. On the contrary, each version of the algorithm is also capable to declare if P is an invalid instance (or P does not have a

valid solution). The proof relating to the termination of the algorithm is intrinsic (in the way the algorithm has been developed) for both the versions (of the algorithm) with a desired solution or a declaration after a specified amount of computation (for smaller values of $n$). Hence we conclude these results in the form of the following theorem.

**Theorem 3.4:** The Sudoku solver, for both of its versions 3.1 and 3.2 , solves a given Sudoku instance P and computes a valid solution S for P, if P has at least one valid solution S, in time and space $O((p!)^{\alpha n})$, where $p$ is the average number of blank cells in a minigrid of P of size $n \times n$, and $0 < \alpha \leq 1$. Moreover, the algorithm terminates in a reasonable amount of CPU time either with a solution S (as a valid solution of P) or with a declaration that P does not have a valid solution, if $n$ is bounded by some constant.

## 3.4 Summary

In this chapter of the thesis, we have developed an exclusive minigrid based Sudoku solving algorithm, which is completely guessed free. The solving algorithm consists of two versions. The first version follows a sequence of minigrids that may not be adjacent but based on the number of clues in the minigrids in succession, whereas the second version of the algorithm follows a user defined path of minigrids based on their adjacency. Both the versions of the algorithm considers each of the minigrids (instead of blank cells in isolation) of size 3×3 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 particularly 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 entirely 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 chapter 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, hard, evil, etc. Incidentally, the Sudoku solver developed in this chapter 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 a desired solution for a given Sudoku instance.