Chapter 4

A Novel Graph Theoretic Sudoku Solver

Overview

In this chapter of the thesis, we have developed the third version (i.e. Version 4.1) of the algorithm, which is based on a graph theoretic formulation of the minigrids, for all the valid permutations of each and every minigrid depending on the given clues. Then we represent these valid permutations of the minigrids as respective vertices of a graph, and based on the observation of compatibility of two valid permutations of two minigrids in a row (or column), we connect them by an edge. As a result the graph we obtain is an undirected graph. From this graph, the solution(s) of a given Sudoku puzzle is (are) computed following the algorithm developed in this chapter, if there exists one. The novelty of this algorithm is that it successfully computes all valid solutions of a given Sudoku instance, if there be two or more such solutions, and for that the algorithm does not take any additional time (or space) or does not bear any additional cost. The graph we compute is also capable to make a decision and conclude if a given puzzle is not a valid Sudoku instance (in terms of not getting a solution).

4.1 *GTSPS:* A Graph Theoretic Approach for Solving Sudoku Puzzle

In this section, we formulate the generalized algorithm for solving a Sudoku instance, as Version 4.1 of the algorithm developed in Chapter 3, using a simple, symmetric graph, $G = (V, E)$, where the graph structure consists of nine sets of vertices, and each such set comprises and represents a set of valid permutations for a minigrid of a given Sudoku puzzle P. In some other words, if there is (are) $p \geq 1$ valid permutation(s) for minigrid M_i, $1 \leq i \leq 9$, then we introducing p vertices into the graph, where each vertex represents a unique permutation of M_i. This means, if the total number of valid permutations for all individual nine minigrids be m, then the graph contains $m = |V|$ vertices. We may recall that all valid permutations for a minigrid are generated only for the given clues in P, individually for all the nine minigrids of the given puzzle, as we computed permutations in Version 3.2 of the algorithm in Chapter 3.

Now we state the criterion we reasonably adopt for introducing edges in G. We are aware of and have already discussed that each minigrid M_i, $1 \le i \le 9$, in a Sudoku problem is adjacent to exactly four other minigrids out of the eight remaining minigrids of the puzzle structure, where two are in row and the left behind two are in column of M_i. In our formulation, we introduce an edge between two vertices v_i and v_j, where v_i is a valid permutation of minigrid M_i and v_j is a valid permutation of minigrid M_j such that the permutations v_i and v_j match each other, and M_i and M_j are either two row minigrids (i.e. minigrids belonging to the same row) or two column minigrids (i.e. minigrids belonging to the same column), where $1 \le i, j \le 9$ but $j \ne i$. This introduction of an edge is based on the compatibility of two permutations for their corresponding minigrids belonging to the same row or to the same column of the Sudoku puzzle.

Two (valid) permutations of the same minigrid M_i are never compatible (based on the clues given); rather, it is not required to introduce an edge between the corresponding vertices in G that are belonging to the same set of valid permutations for any M_i, as for a given Sudoku puzzle P, we like to take exactly one valid permutation from each minigrid to compute a desired solution of P. On the other hand, it is unnecessary to judge whether two valid permutations of two minigrids M_i and M_j are compatible (or not) that are neither row minigrids nor column minigrids. This is because at the time of introducing edges we only consider two minigrids that are either vertically aligned in column or horizontally aligned in row; two diagonal minigrids are nowise directly related to each other. Two diagonal minigrids M_i and M_j do have two common intersectional minigrids M_{k1} and M_{k2} such that individual compatibility between M_i and M_j with M_{k1} and M_{k2} are to be maintained. As for example, two diagonal minigrids 1 and 8 have their intersectional minigrids 2 and 7, where minigrid pairs $\langle 1,2 \rangle$, $\langle 1,7 \rangle$, $\langle 2,8 \rangle$, and $\langle 7,8 \rangle$ are to be considered for judging compatibility for all of their valid permutations.

Now suppose M_i, M_j, and M_k are three row minigrids and p_{ix}, p_{jy}, and p_{kz} are three of their valid permutations, respectively, that are compatible to each other, where x, y, and z are three integers, $1 \le i, j, k \le 9$, and p_{ix} may be assumed as the xth valid permutation of the ith minigrid (i.e. minigrid M_i). That means the corresponding vertices v_{ix} and v_{jy} are to be connected by an edge, v_{jy} and v_{kz} are to be connected by an edge, and v_{ix} and v_{kz} are to be connected by an edge in G. This means that a set of compatible permutations of the minigrids in a row must form a triangle in G, if the given Sudoku puzzle P has a solution S. We may notice that a set of compatible permutations of the minigrids in a column must also form a triangle in *G*, if a valid solution S is there for P. This means that for each minigrid M_i, $1 \le i \le 9$, there must have at least one valid permutation p_{ix} such that the corresponding vertex v_{ix} in *G* would belong to two cliques of size three each, and there must be a subgraph of *G* which is isomorphic to the graph G_S, as shown in Figure 4.1. We may further observe that the degree of each of the vertices in G_S is four, which is analogous to that of an instance P where each minigrid M_i is to be compatible to four other minigrids (in terms of their valid permutations), out of which two are in the same row and the remaining two are in the same column of M_i.

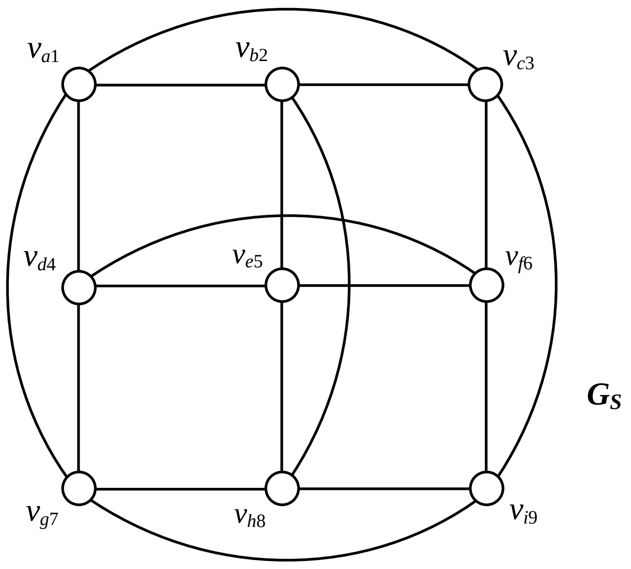

Figure 4.1: A graph G_S that contains nine vertices, and for the sake of simplicity we draw it in a 3×3 fashion, where the three vertices in a row form a triangle and the three vertices in a column form a triangle.

In this algorithm, our objective is to find out and extract a subgraph of *G* which is isomorphic to G_S. If there is only one induced subgraph that is isomorphic to G_S in *G*, then S for P is unique; otherwise, the number of such distinct induced subgraphs each of which is isomorphic to G_S in *G* is the number of valid solutions for the given Sudoku instance P. Here for any two such subgraphs G_{S1} and G_{S2} of *G*, the subgraphs may contain one or more common vertices in *G* (and certainly several other vertices in *G* are unlike based on their presence in the said subgraphs). It may be observed that for two such valid solutions S_1 and S_2 for a given Sudoku instance P, a minimum of two minigrids may have different valid permutations so that each of G_{S1} and G_{S2} of

G is isomorphic to G_S in Figure 4.1. For the Sudoku instance shown in Figure 1.2(a), two such solutions are shown in Figure 1.2(b). So, we may assume that if a valid permutation of one minigrid is changed towards obtaining another solution of a given Sudoku puzzle, then at least one more valid permutation of another minigrid must have to be changed.

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

Figure 4.2: A 35-clue Sudoku instance P that we consider for developing the graph theoretic technique.

Now we like to describe the algorithm with the help of a Sudoku instance P as shown in Figure 4.2, which is a 35-clue Sudoku puzzle. First of all, we compute all valid permutations of P obeying only the given clues in P. Minigrid-wise, for minigrid 1 (or M_1) through minigrid 9 (or M_9), the valid permutations that we obtain are as follows: Two valid permutations for M_1 are 978235 (or 1_a) and 978325 (or 1_b), two valid permutations for M_2 are 59634 (or 2_a) and 65934 (or 2_b), six valid permutations for M_3 are 678192 (or 3_a), 678291 (or 3_b), 678921 (or 3_c), 876192 (or 3_d), 876291 (or 3_e), and 876921 (or 3_f), one valid permutation for M_4 is 648 (or 4_a), two valid permutations for M_5 are 892357 (or 5_a) and 893257 (or 5_b), one valid permutation for M_6 is 362 (or 6_a), four valid permutations for M_7 are 142367 (or 7_a), 143267 (or 7_b), 162347 (or 7_c), and 163247 (or 7_d), nine valid permutations for M_8 are 46825 (or 8_a), 46852 (or 8_b), 48625 (or 8_c), 48652 (or 8_d), 56824 (or 8_e), 58624 (or 8_f), 62854 (or 8_g), 68425 (or 8_h), and 68452 (or 8_i), and six valid permutations for M_9 are 249315 (or 9_a), 259314 (or 9_b), 342915 (or 9_c), 349215 (or 9_d), 352914 (or 9_e), and 359214 (or 9_f).

Hence we introduce a total of 33 vertices into the graph *G*, and the vertices are also identified after 1_a through 9_f, as shown in Figure 4.3. Here each vertex represents a valid permutation of its corresponding minigrid obeying only the given clues in P. An edge is introduced between a pair of vertices only if the coupled permutations (of an associated pair of row or column minigrids) are compatible (or matching) to each other. To differentiate the valid permutations of the same minigrid we place all of them in the same level and permutations of different minigrids are placed in different levels of the graph. As for example, the vertices 1_a and 1_b are placed in the topmost stage, the vertices 2_a and 2_b are placed in the next to the topmost stage, and so on in the figure. Valid permutations of the same minigrid are also arranged in ascending order in placing them from left to right. Now we introduce edges for the valid permutations of a pair of row (and column) minigrids that are compatible (or matching) to each other. For two such minigrids if *p* and *q* be the number of valid permutations, then we are supposed to compare $p \times q$ pairs of permutations to judge their compatibility. Hence there are 18 pairs of row and column minigrids whose all possible valid permutations are to be considered and compared; if they match, the associated vertices are joined by an (undirected) edge, and otherwise, the corresponding pair of vertices is not connected by an edge.

The 18 pairs of minigrids that we have to consider for all their valid permutations are (M_1, M_2), (M_1, M_3), (M_1, M_4), (M_1, M_7), (M_2, M_3), (M_2, M_5), (M_2, M_8), (M_3, M_6), (M_3, M_9), (M_4, M_5), (M_4, M_6), (M_4, M_7), (M_5, M_6), (M_5, M_8), (M_6, M_9), (M_7, M_8), (M_7, M_9), and (M_8, M_9). Hence, considering all permutations for each such pair of minigrids, we do introduce 104 undirected edges into the graph, and the edges are as follows as shown in Figure 4.3: {1_a, 2_b}, {1_a, 3_d}, {1_a, 3_e}, {1_a, 3_f}, {1_a, 4_a}, {1_b, 4_a}, {1_a, 7_b}, {1_a, 7_d}, {1_b, 7_a}, {1_b, 7_c}, {2_a, 3_a}, {2_a, 3_b}, {2_a, 3_c}, {2_b, 3_d}, {2_b, 3_e}, {2_b, 3_f}, {2_b, 5_b}, {2_a, 8_e}, {2_b, 8_f}, {2_b, 8_g}, {3_b, 6_a}, {3_c, 6_a}, {3_e, 6_a}, {3_f, 6_a}, {3_b, 9_a}, {3_b, 9_b}, {3_b, 9_d}, {3_b, 9_f}, {3_c, 9_c}, {3_c, 9_e}, {3_e, 9_a}, {3_e, 9_b}, {3_e, 9_d}, {3_e, 9_f}, {3_f, 9_c}, {3_f, 9_e}, {4_a, 5_a}, {4_a, 5_b}, {4_a, 6_a}, {4_a, 7_a}, {4_a, 7_b}, {4_a, 7_c}, {4_a, 7_d}, {5_a, 6_a}, {5_b, 6_a}, {5_b, 8_c}, {5_b, 8_f}, {5_b, 8_h}, {6_a, 9_a}, {6_a, 9_b}, {6_a, 9_c}, {6_a, 9_d}, {6_a, 9_e}, {6_a, 9_f}, {7_a, 8_g}, {7_a, 8_h}, {7_a, 8_i}, {7_b, 8_h}, {7_b, 8_i}, {7_c, 8_a}, {7_c, 8_b}, {7_c, 8_c}, {7_c, 8_d}, {7_c, 8_e}, {7_c, 8_f}, {7_d, 8_a}, {7_d, 8_b}, {7_d, 8_c}, {7_d, 8_d}, {7_d, 8_e}, {7_d, 8_f}, {7_a, 9_e}, {7_a, 9_f}, {7_b, 9_b}, {7_c, 9_c}, {7_c, 9_d}, {7_c, 9_e}, {7_c, 9_f}, {7_d, 9_a}, {7_d, 9_b}, {8_a, 9_b}, {8_a, 9_e}, {8_a, 9_f}, {8_b, 9_b}, {8_b, 9_e}, {8_b, 9_f}, {8_c, 9_b}, {8_c, 9_e}, {8_c, 9_f}, {8_d, 9_b}, {8_d, 9_e}, {8_d, 9_f}, {8_e, 9_a}, {8_e, 9_c}, {8_e, 9_d}, {8_f, 9_a}, {8_f, 9_c}, {8_f, 9_d}, {8_h, 9_b}, {8_h, 9_e}, {8_h, 9_f}, {8_i, 9_b}, {8_i, 9_e}, and {8_i, 9_f}. This graph can also be visualized with the help of a linked list representation of a graph that contains 33 header nodes, and for an edge $\{v_i, v_j\}$ in G, a header node v_i contains a linked node v_j whereas a header node v_j contains a linked node v_i, as G is an undirected graph, as shown in Figure 4.4.

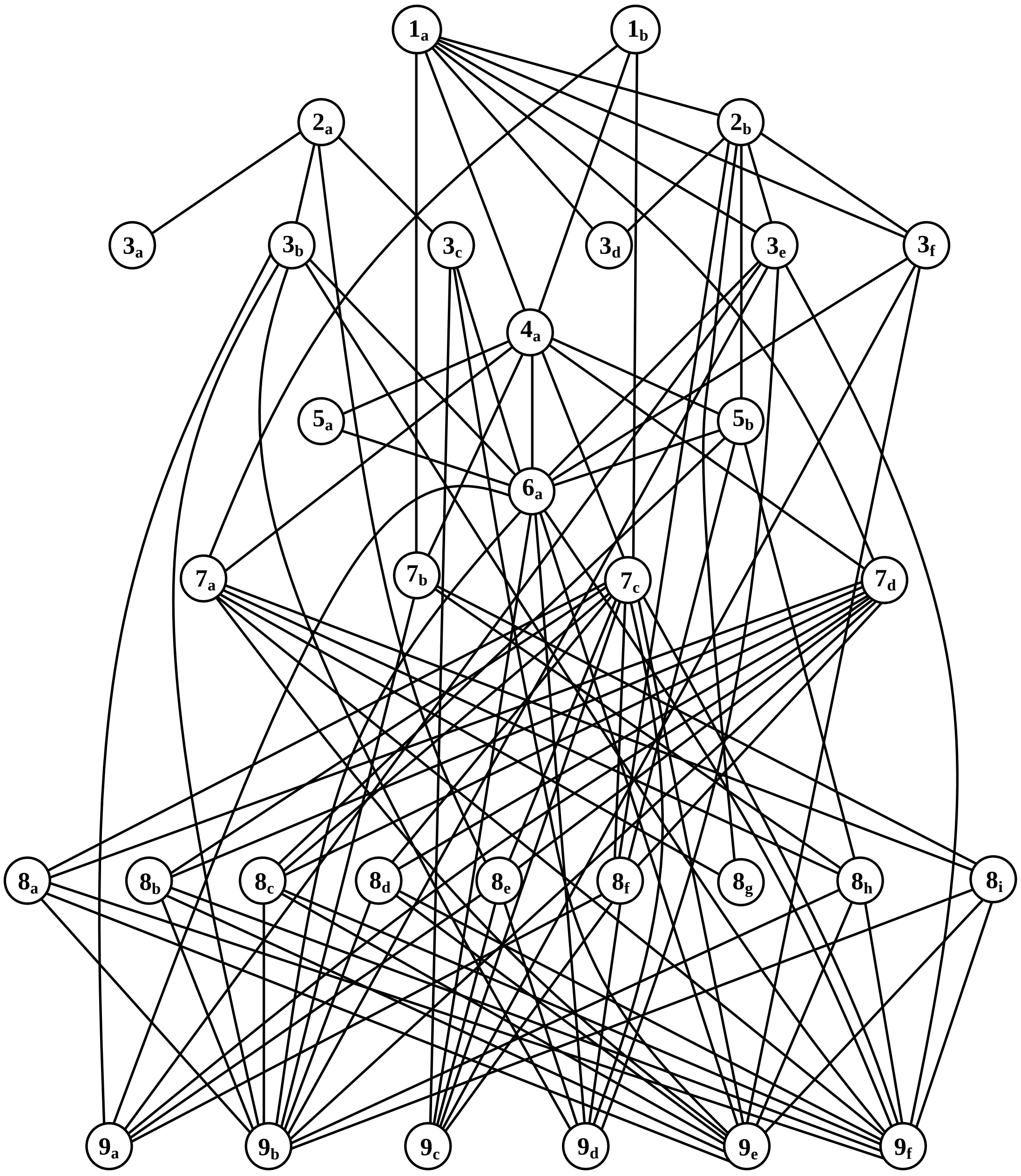

Figure 4.3: The graph G that is obtained for the 35-clue Sudoku instance P of Figure 4.2.

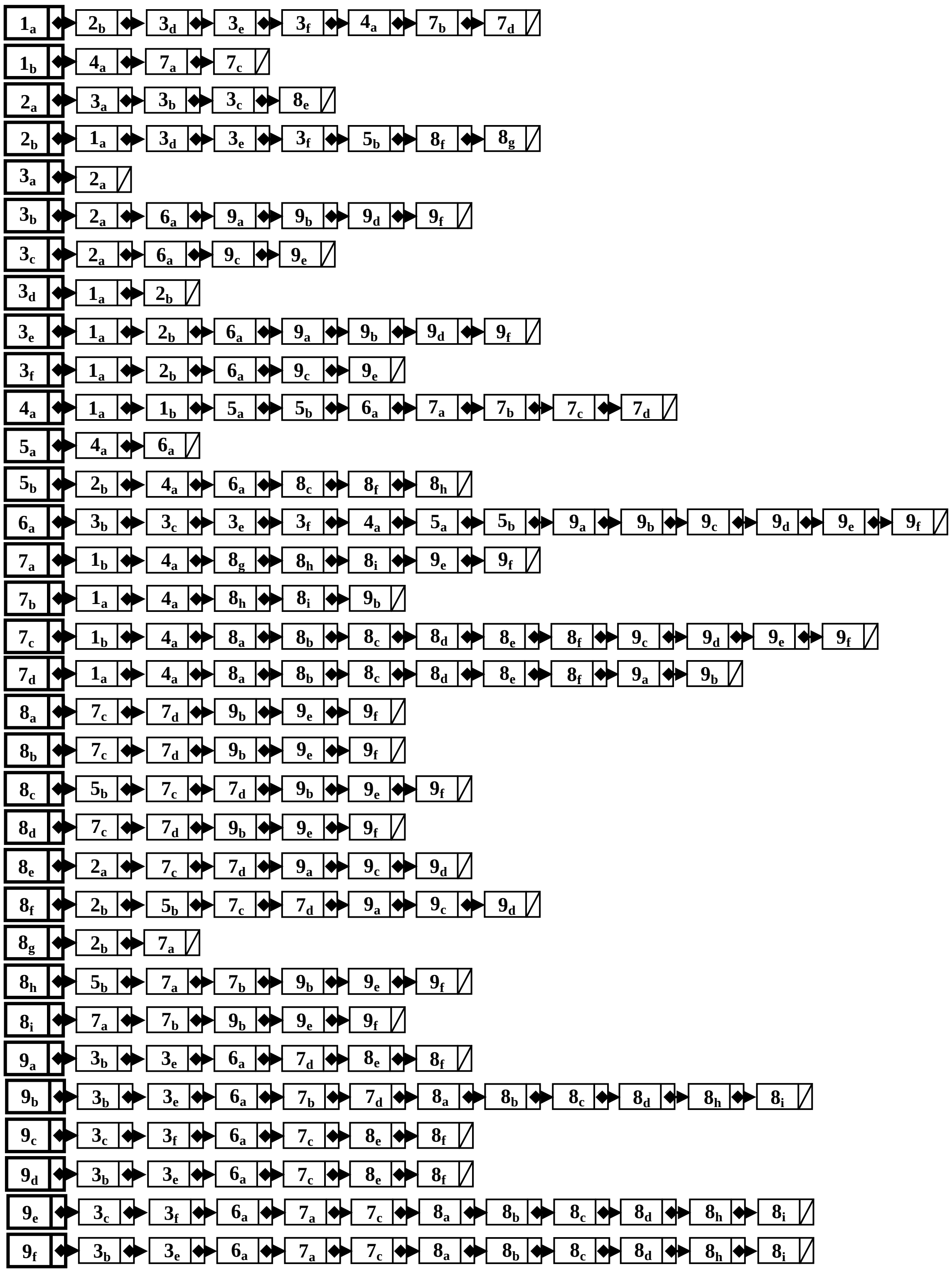
1_a 2_b 3_d 3_e 3_f 4_a 7_b 7_d
1_b 4_a 7_a 7_c
2_a 3_a 3_b 3_c 8_e
2_b 1_a 3_d 3_e 3_f 5_b 8_f 8_g
3_a 2_a
3_b 2_a 6_a 9_a 9_b 9_d 9_f
3_c 2_a 6_a 9_c 9_e
3_d 1_a 2_b
3_e 1_a 2_b 6_a 9_a 9_b 9_d 9_f
3_f 1_a 2_b 6_a 9_c 9_e
4_a 1_a 1_b 5_a 5_b 6_a 7_a 7_b 7_c 7_d
5_a 4_a 6_a
5_b 2_b 4_a 6_a 8_c 8_f 8_h
6_a 3_b 3_c 3_e 3_f 4_a 5_a 5_b 9_a 9_b 9_c 9_d 9_e 9_f
7_a 1_b 4_a 8_g 8_h 8_i 9_e 9_f
7_b 1_a 4_a 8_h 8_i 9_b
7_c 1_b 4_a 8_a 8_b 8_c 8_d 8_e 8_f 9_c 9_d 9_e 9_f
7_d 1_a 4_a 8_a 8_b 8_c 8_d 8_e 8_f 9_a 9_b
8_a 7_c 7_d 9_b 9_e 9_f
8_b 7_c 7_d 9_b 9_e 9_f
8_c 5_b 7_c 7_d 9_b 9_e 9_f
8_d 7_c 7_d 9_b 9_e 9_f
8_e 2_a 7_c 7_d 9_a 9_c 9_d
8_f 2_b 5_b 7_c 7_d 9_a 9_c 9_d
8_g 2_b 7_a
8_h 5_b 7_a 7_b 9_b 9_e 9_f
8_i 7_a 7_b 9_b 9_e 9_f
9_a 3_b 3_e 6_a 7_d 8_e 8_f
9_b 3_b 3_e 6_a 7_b 7_d 8_a 8_b 8_c 8_d 8_h 8_i
9_c 3_c 3_f 6_a 7_c 8_e 8_f
9_d 3_b 3_e 6_a 7_c 8_e 8_f
9_e 3_c 3_f 6_a 7_a 7_c 8_a 8_b 8_c 8_d 8_h 8_i
9_f 3_b 3_e 6_a 7_a 7_c 8_a 8_b 8_c 8_d 8_h 8_i

Figure 4.4: The linked list representation of graph *G* in Figure 4.3.

There are 33 header nodes as there are 33 valid permutations in total for all the nine minigrids of the Sudoku instance in Figure 4.2. As *G* is an undirected graph, for each edge $\{v_i, v_j\}$ in *G*, a header node v_i (v_j) contains a linked node v_j (v_i). So for 104 edges in *G*, there are 208 linked nodes in this representation.

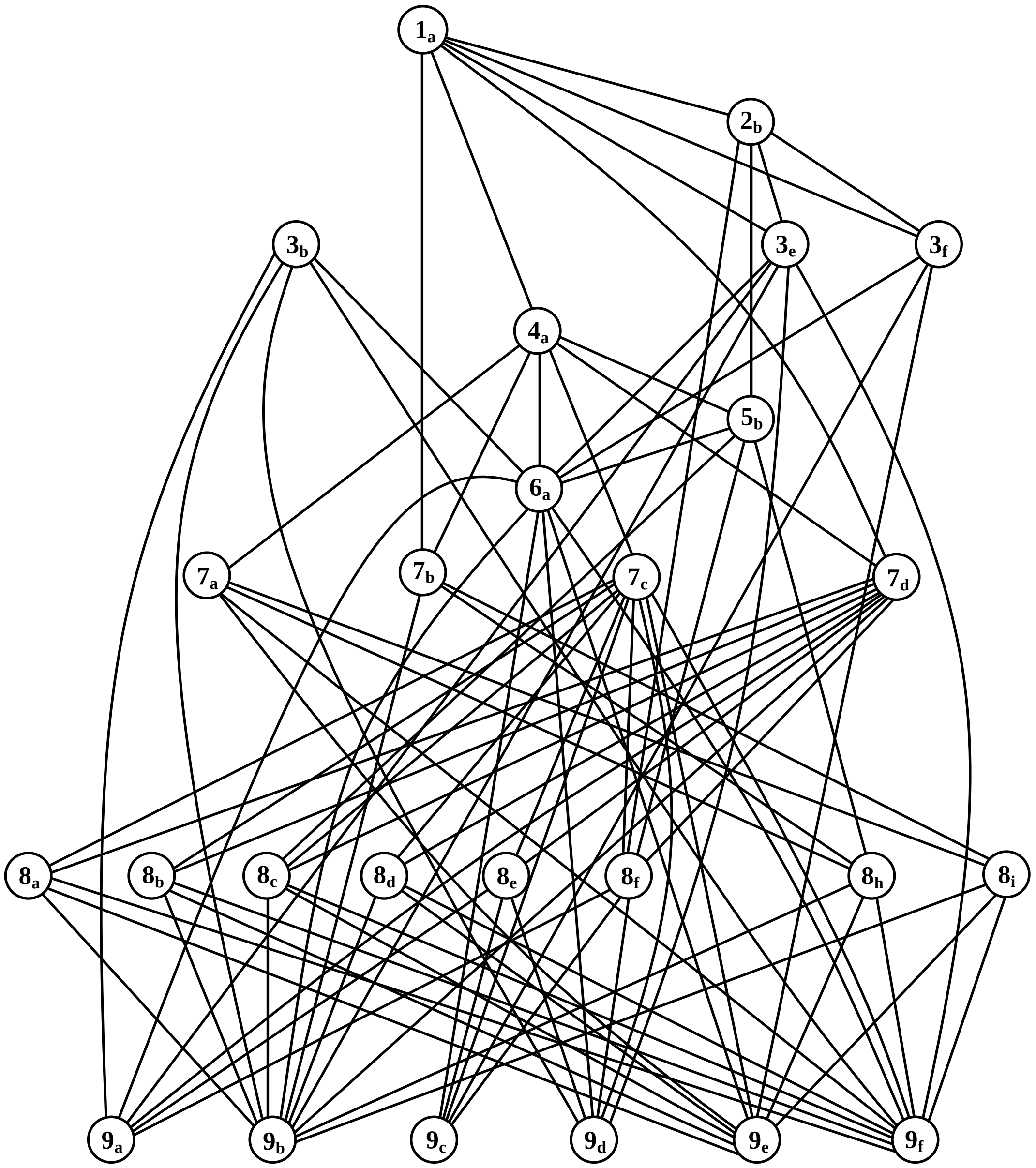

Figure 4.5: The modified graph *G* obtained by repetitively deleting vertices (along with their edges) with degree three or less starting from the graph, *G* in Figure 4.3.

Now our objective is to extract a subgraph (of *G*), which is isomorphic to that of the graph structure shown in Figure 4.1, as explained and described earlier. We know that the *subgraph* *isomorphism problem* is NP-hard [44]; so, we may not develop a polynomial time algorithm to identify a subgraph, if there is any subgraph of a general graph G_1, which is isomorphic to another graph G_2, where G_2 is smaller in size than that of G_1. But for a graph, which is similar to that of Figure 4.3, that contains only nine sets of vertices, the problem is solvable in reasonable amount of time, as from each such set only one vertex has to acquire for the subgraph, each vertex should have degree at least four, and there is a specific structure of adjacency for each of the recognized vertices.

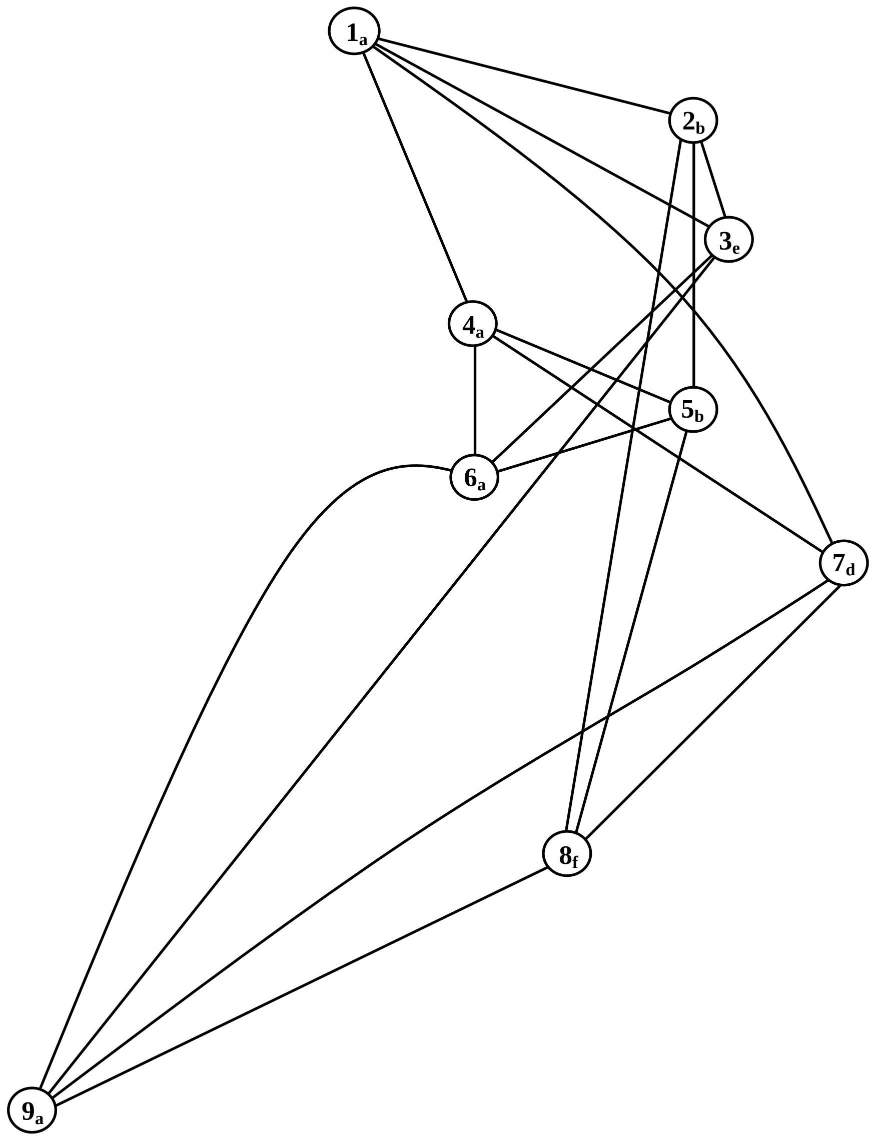

Figure 4.6: The finally modified graph *G*, obtained by repetitively deleting vertices (along with their edges) based on the compatibility among the corresponding permutations along rows and columns starting from the intermediate modified graph, *G* in Figure 4.5.

So, if possible, we like to delete all the vertices with degree three or less, along with their associated edges, straightway from the graph (i.e. graph *G*) of Figure 4.3, as none of them is responsible in obtaining a required graph structure similar to that of Figure 4.1. Notice that the vertices 1_b, 3_a, 3_d, 5_a, and 8_g have their degree 3, 1, 2, 2, and 2, respectively. So, we delete all these vertices along with all their incident edges from graph *G*, and obtain a modified (graph) *G*. Then we further apply and continue this step if a vertex in *G* is of degree three or less, and stop when we find that none of the vertices in the resulting modified (graph) *G* is of degree three or less, which has been shown in Figure 4.5. Hence, the degree of each of the vertices of this modified graph *G* is at least four.

Now we may notice that a vertex in the resulting (modified graph) *G* (in Figure 4.5) with degree four or more for each of its vertices, may not be assumed as a member of the subgraph (isomorphic to the graph in Figure 4.1) due to its structural non-adjacency.

As for example, a vertex corresponding to a permutation for minigrid 1 (i.e. M_1) must be adjacent to similar vertices of M_2, M_3, M_4, and M_7, where the matching vertices of M_2 and M_3 have to be connected by an edge and the matching vertices of M_4 and M_7 have also to be connected by an edge in this graph. Independently, this is to be a fact for at least a vertex (for its related permutation) for each of the minigrids, such that along with vertices of its corresponding row and column minigrids, in combination we get a subgraph of this graph, which is isomorphic to the graph in Figure 4.1, if a solution S for a given Sudoku puzzle P is there.

Succeeding to the facts of the structural adjacency as stated above, we may observe that in graph *G* in Figure 4.5, vertex 1_a is not adjacent to the vertices 3_b, 7_a, and 7_c. So, we delete the vertices 3_b, 7_a, and 7_c along with their adjacent edges, as 1_a is the vertex for the only valid permutation of M_1 up to this point in time (where M_3 is a row minigrid and M_7 is a column minigrid of M_1). Also, 2_b is the only vertex available now for minigrid M_2 (similar to 1_a for M_1), which is adjacent to only 8_f out of eight available (valid) permutations of minigrid 8 (i.e. M_8). So, we delete all the remaining seven vertices of M_8 along with their adjacent edges and simplify the graph structure towards a desired one. Also we may observe that the vertex 7_b is connected to 1_a as well as 4_a, which are also connected by an edge, but 7_b is not connected to 8_f. So, we delete 7_b along with its edges from *G* that is still being modified. This step is followed repeatedly till we fail to point out such a vertex that is not adjacent to the vertices matching to the permutations of all its row as well as column minigrids, and in due course we finally obtain a modified graph *G*, depicted in Figure 4.6.

As for example, in this process of deleting vertices (and their adjacent edges), we may further observe that now there is only one permutation of minigrid 7 (i.e. M_7), which is 7_d whereas the only one permutation of minigrid 8 (i.e. M_8) is 8_f, which are adjacent to each other and also jointly adjacent to 9_a, the only permutation of minigrid 9 (i.e. M_9). Hence we delete all the remaining permutations of minigrid 9 (that are 9_b through 9_f) along with their adjacent edges. Moreover, we may observe that at this point in time the degree of each of these vertices (of M_9, except 9_a) is less than four; so each of them must not be a member permutation of the overall desired solution of the given Sudoku puzzle. We now may observe that the degree of vertex 3_f is three, which is also to be deleted from the graph along with its edges. Interestingly, this is how we obtain a graph which is exactly similar (or isomorphic) to the graph shown in Figure 4.1. Hence, we conclude that the permutations for each of the minigrids matching to the vertices present in the final graph (in Figure 4.6) in combination provide the only desired solution S, as shown in Figure 4.7, for the assumed 35-clue Sudoku instance shown in Figure 4.2.

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

Figure 4.7: The solution of the 35-clue Sudoku instance P shown in Figure 4.2 that we considered for developing the graph theoretic version of the algorithm *GTSPS.*

Now we expose the next versions (i.e. Version 4.2) of the algorithm in the next section with necessary further updates to visualize the modifications better and clear.

4.2 Version 4.2 of the Algorithm: A Different Adaptation of *GTSPS*

In this section, we develop an alternative and compacted version (i.e. Version 4.2) of the algorithm based on graph theory, designed and described in the preceding section. We take the help of the same 35-clue Sudoku instance shown in Figure 4.2, to build up and explain this version of the algorithm. So, we may consider either the graph of Figure 4.3 or the graph of Figure 4.5 to start with edifying the algorithm. This is a vertex contraction algorithm that we like to achieve based on the theory that if three distinct permutations of the three minigrids in a row or in a column be compatible to each other, then we may combine them to form a three-tuple super-vertex. Hence it is very clear that the other parts of the graph, which does not take role to form a three-tuple super-vertex, are right away excluded from the graph as redundant information. In this form of the graph we connect two super-vertices corresponding to two sets of row minigrids (or two sets of column minigrids), only if the allied and correlated column minigrids (or row minigrids) are attuned for their respective permutations.

Now we observe the modified graph *G* in Figure 4.5 and find the graph that contains 35 three-tuples. Two three-tuples, based on two rows of minigrids or two columns of minigrids, may be considered adjacent only if all the corresponding orthogonal permutations match each other. As for example, from Figure 4.5, we may note that the created three-tuples are as follows: $\langle 1_a\text{-}2_b\text{-}3_e\rangle$, $\langle 1_a\text{-}2_b\text{-}3_f\rangle$, $\langle 4_a\text{-}5_b\text{-}6_a\rangle$, $\langle 7_a\text{-}8_h\text{-}9_e\rangle$, $\langle 7_a\text{-}8_h\text{-}9_f\rangle$, $\langle 7_a\text{-}8_i\text{-}9_e\rangle$, $\langle 7_a\text{-}8_i\text{-}9_f\rangle$, $\langle 7_b\text{-}8_h\text{-}9_b\rangle$, $\langle 7_b\text{-}8_i\text{-}9_b\rangle$, $\langle 7_c\text{-}8_a\text{-}9_e\rangle$, $\langle 7_c\text{-}8_a\text{-}9_f\rangle$, $\langle 7_c\text{-}8_b\text{-}9_e\rangle$, $\langle 7_c\text{-}8_b\text{-}9_f\rangle$, $\langle 7_c\text{-}8_c\text{-}9_e\rangle$, $\langle 7_c\text{-}8_c\text{-}9_f\rangle$, $\langle 7_c\text{-}8_d\text{-}9_e\rangle$, $\langle 7_c\text{-}8_d\text{-}9_f\rangle$, $\langle 7_c\text{-}8_e\text{-}9_c\rangle$, $\langle 7_c\text{-}8_e\text{-}9_d\rangle$, $\langle 7_c\text{-}8_f\text{-}9_c\rangle$, $\langle 7_c\text{-}8_f\text{-}9_d\rangle$, $\langle 7_d\text{-}8_a\text{-}9_b\rangle$, $\langle 7_d\text{-}8_b\text{-}9_b\rangle$, $\langle 7_d\text{-}8_c\text{-}9_b\rangle$, $\langle 7_d\text{-}8_d\text{-}9_b\rangle$, $\langle 7_d\text{-}8_e\text{-}9_a\rangle$, $\langle 7_d\text{-}8_f\text{-}9_a\rangle$, $\langle 1_a\text{-}4_a\text{-}7_b\rangle$, $\langle 1_a\text{-}4_a\text{-}7_d\rangle$, $\langle 2_b\text{-}5_b\text{-}8_f\rangle$, $\langle 3_e\text{-}6_a\text{-}9_a\rangle$, $\langle 3_e\text{-}6_a\text{-}9_b\rangle$, $\langle 3_e\text{-}6_a\text{-}9_d\rangle$, $\langle 3_f\text{-}6_a\text{-}9_c\rangle$, $\langle 3_f\text{-}6_a\text{-}9_e\rangle$. In this graph theoretic representation, we introduce a vertex into the graph for each of the three-tuples cited above, and if two parallel three-tuples, based on the associated permutations of either row or column minigrids, are matching to each other, derived from the permutations of the minigrids available in the orthogonal direction, we introduce an edge between the corresponding super-vertices. Hence the graph we obtain is *G** that has been revealed in Figure 4.8.

We may draw our attention in viewing the graph in Figure 4.8 that most of the super-vertices are isolated in this figure. This means that the corresponding permutations are only locally compatible either in a row or in a column of minigrids. Hence we conclude that this graph in Figure 4.8 can be reduced further if we consider the final graph obtained by deleting vertices (along with their edges) over and over again based on the compatibility of the corresponding permutations along rows and columns (starting from the intermediate modified graph *G* in Figure 4.5), as shown in Figure 4.6. Also, the graph in Figure 4.8 can even further be packed in only considering the row (column) three-tuples (wherein the permutations are compatible to each other) as super-vertices and draw the graph based on column (row) matching for all the three sets of corresponding permutations.

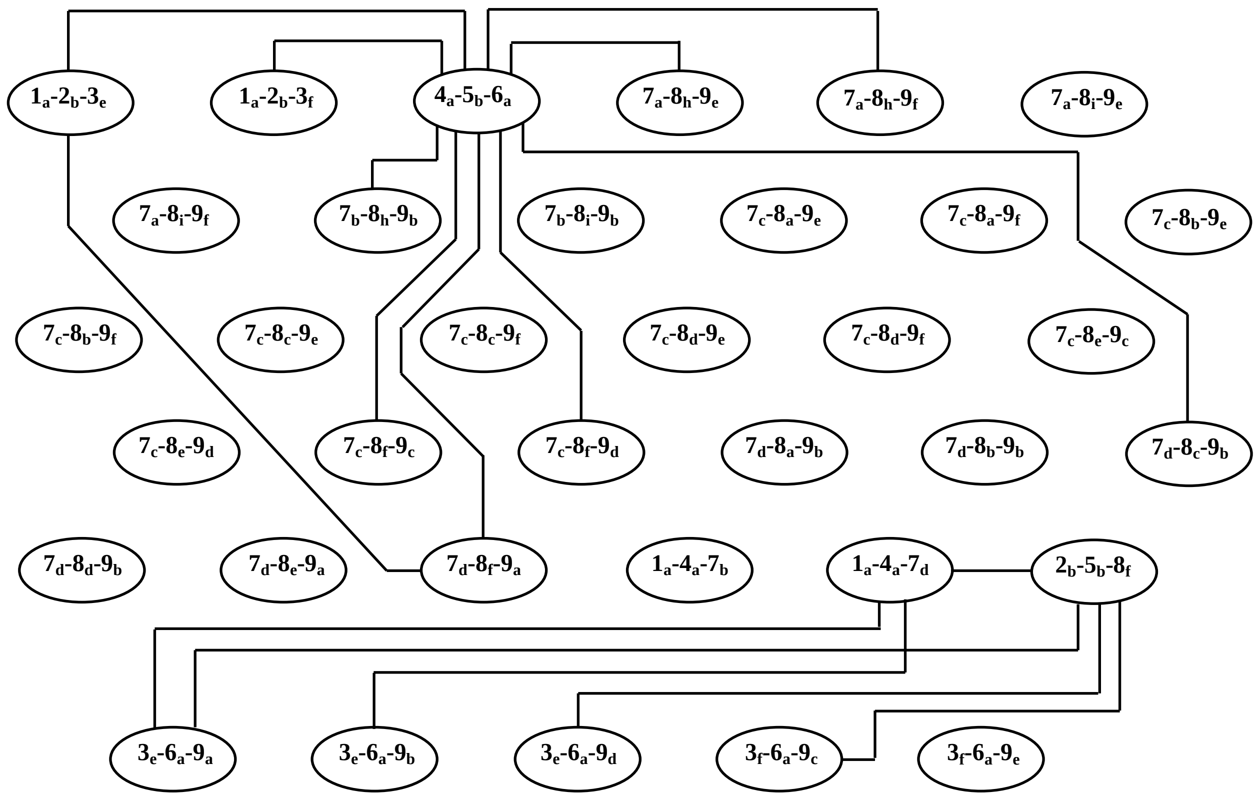

Figure 4.8: The graph *G**, derived from the finally modified graph *G* (in Figure 4.5) by combining row-wise and column-wise matching permutations, where the matched permutations in a row (or in a column) form a super-vertex of size three each, and the two such super-vertices are connected by an edge only if the related permutations also match to each other for all the three orthogonal minigrids.

Now it is worthwhile to state that if we find three three-tuples that are computed row-wise are also matching column-wise for each pair of subsequent matched permutations, then each twosome of the said three-tuples must be connected by an edge, and they also form a three-cycle in the graph. We may eventually ensure that the reverse is also true, and that is why we find two triangles in Figure 4.8, each of which computes the same solution as shown in Figure 4.7.

In some other words, the super-vertices corresponding to the three-tuples $\langle 1_a\text{-}2_b\text{-}3_e \rangle$, $\langle 4_a\text{-}5_b\text{-}6_a \rangle$, and $\langle 7_d\text{-}8_f\text{-}9_a \rangle$ form a three-cycle when initially we consider the permutations along the row minigrids and compatibility is judged among the permutations column-wise. The same solution is obtained when initially we consider the permutations along the column minigrids and compatibility is judged among the permutations row-wise, where the super-vertices corresponding to the three-tuples $\langle 1_a\text{-}4_a\text{-}7_d \rangle$, $\langle 2_b\text{-}5_b\text{-}8_f \rangle$, and $\langle 3_e\text{-}6_a\text{-}9_a \rangle$ form another three-cycle. So, as we visualise in Figure 4.8, if there are two solutions of a given Sudoku instance P, we must have four such three-cycles in such a graph. If there are three solutions of a given Sudoku instance P, we must have six such three-cycles in such a graph, and so on. This is how we may compute all valid solutions of a given Sudoku instance, if there are two or more; we establish this result in the following theorem.

Theorem 4.1: The graph theoretic guessed free Sudoku puzzle solver *GTSPS* computes all m valid solutions for a given Sudoku puzzle P of size $n \times n$, if $m \geq 1$, and indeed the number of solutions for P is never $m+1$ or more, where $\sqrt{n} \geq 2$ is an integer. The algorithm also successfully declares if there is no valid solution for the given Sudoku puzzle P without any extra cost.

4.3 The Algorithm at a Glance

In this section, we like to see the algorithm *GTSPS* at a glance that has been developed and exemplified in the previous sections, as follows. By the way, the algorithm considers a standard 9×9 Sudoku puzzle P as input and computes all its valid solutions, one or more. Even if the given instance has no valid solution, the algorithm identifies the same following the nine steps below as it has been developed. Notice that a valid solution of a given Sudoku instance P is viewed in the form of a subgraph (of the whole graph structure we obtain for P), which is isomorphic to the graph in Figure 4.1, wherein each vertex represents a (compatible) permutation of a particular minigrid.

Algorithm: *GTSPS* (A Graph Theoretic Sudoku Puzzle Solver)

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

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

Step 1: Compute the number of digits given as clue in each minigrid. Also identify the digits that are given, if any, and the missing digits to be inserted to each of the minigrids in P.

Step 2: Compute all valid permutations of the missing digits (for the blank locations in ascending order) in each of the (nine) minigrids based on the existing clues in P (in other row and column minigrids), and store them.

Step 3: Construct a graph $G = (V, E)$, where V is the set of vertices such that a vertex $v_{ex} \in V$ represents a valid permutation p_{ix} of minigrid M$_i$ and $x \geq 1$ is an integer. So, for altogether p valid permutations of all nine minigrids in P, $p = |V|$.

Step 4: Two vertices v_{ix} and v_{jy} corresponding to two valid permutations p_{ix} and p_{jy} of a pair of row or column minigrids M$_i$ and M$_j$ are connected by an edge $\{v_{ix}, v_{jy}\} = e \in E$, only if the permutations are compatible to each other, where $1 \leq i, j \leq 3$ and $x, y \geq 1$ are integers.

Step 5: Compute degree of each of the vertices in G. If the degree of a vertex is less than four, delete the vertex along with its adjacent edges from the graph. Repeat this step until a vertex in G is found with degree three or less. Hence, at last a modified (graph) G is obtained that contains no vertex with degree less than four.

Step 6: Observe groups of three-cycles among the valid permutations (present in the finally obtained modified graph G in Step 5) for each set of row minigrids and also for each set of column minigrids, and obtain a reduced graph G^* with only three-tuple super-vertices.

Step 7: For a pair of sets of row (column) permutations (present in the super-vertices), introduce an edge in G^*, only if all the corresponding valid permutations present in the assumed sets are compatible column-wise (row-wise).

Step 8: If three such sets of permutations are found in G^* that again form a three-cycle, then they collectively generate a valid solution S of P. If initially (three) valid permutations of (three) row (column) minigrids form the super-vertices and three such super-vertices form a three-cycle in G^*, then the matching valid permutations of (three) column (row) minigrids also form super-vertices and three such super-vertices also form a three-cycle in G^*, and all these (nine) valid permutations in combination produce the same solution S of P.

Step 9: If there is no such combination of super-vertices forming three-cycles in G^*, then there is no valid solution for the Sudoku puzzle P. Otherwise, if there are $q \geq 1$ such combination(s) of super-vertices forming three-cycles in G^*, then there is (are) q valid solution(s) for the Sudoku puzzle P.

Hence, following the above nine steps of the algorithm we can compute all valid solutions of a given Sudoku puzzle P. By the way, here the algorithm has been developed for a given Sudoku instance of size 9×9, but it can also be applied for any larger Sudoku instance of size $n \times n$, where n is bounded by some constant, and to achieve a similar setting (where each minigrid is a square) we may consider that the value of $\sqrt{n}$ is either 4, or 5, or more.

Now we briefly discuss some of the significant features relating to the algorithm. Without loss of generality, in representing the graph G in Step 3, we may keep all the valid permutations of a minigrid in the same level and in a more specified form the valid permutations of subsequent minigrids could be placed in successive levels. See Figures 4.3, 4.5, and 4.6, for example, where we have explicitly drawn and shown such graphs for an assumed 35-clue Sudoku instance.

It is inevitable that we are supposed to review compatibility only among the valid permutations of row and column minigrids while introducing edges into pairs of corresponding vertices in G. If there is no conflict in terms of all pairs of digits in assigning them (as permutations) to rows as well as to columns, then the associated vertices are connected by an edge. This implies that the allied pair of valid permutations of two row or column minigrids forms a matched pair.

After an application of formation of three-cycles over the graph obtained in Step 5, we may observe that there might have valid permutations not present in any of the three-cycles. This implies that such a valid permutation is not compatible in combination to other valid permutations along its row minigrids and also along its column minigrids. So we delete this vertex (allied to such a valid permutation) while computing G^* in Step 6 of the algorithm. Hence, each vertex in G^* is a super-vertex comprising three valid permutations in a row or three valid permutations in a column that are compatible to each other.

In G^*, we introduce an edge to connect two super-vertices consisting of permutations of row (column) minigrids in Step 7, only if all the matching permutations along the column (row) minigrids are compatible to each other. Hence, we obtain the graph G^* as shown in Figure 4.8 for the assumed 35-clue Sudoku instance. We may observe that most of the super-vertices in this graph are isolated, rather not present in a three-cycle. We may at once delete each such super-vertex from G^*. Anyway, what we essentially examine in G^* is whether it contains a three-cycle among the super-vertices. Our observation is that there is always an even number of three-cycles, if any, in G^*. This is because if we initially find three super-vertices based on rows (columns) form a three-cycle, then there must have three super-vertices based on columns (rows) forming a three-cycle, wherein the same set of nine valid permutations must be present in the orthogonal minigrids.

Hence in G^*, the number of three-cycles formed is always even, if there is a valid solution of the given Sudoku puzzle, and the number of solutions is precisely the half of the number of three-cycles in G^*, as a pair of three-cycles produces the same solution S for P. Next we compute the computational complexity of the algorithm developed in the subsequent section and conclude some important upshots.

4.4 Computational Complexity of the Algorithm

Now we discuss about the size of the tree structure we compute for each of the minigrids based on the clues of a given Sudoku instance P. One such tree structure is shown in Figure 3.5, whose computation dominates the overall computational complexity of the algorithm developed in this thesis (and we have made this computation there in Chapter 3 as well).

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 chapter is $(p!-x)^n = O((p!)^{\alpha n})$, where α is a positive constant, whose value is not more than one, and x is the average number of invalid permutations based on the clues given in the Sudoku puzzle P. Our observation is that for a given instance P, x is very close to $p!$ and hence $p!-x$ is a reasonably small number and in our case the value of n is equal to 9. Hence, the experimentations made by this algorithm take negligible amount of clock (or CPU) time, of the order of micro-to-milliseconds. We conclude that the algorithm developed herein guarantees a valid solution of a given Sudoku puzzle, if one exists, and theoretically it takes time and space exponential in size; these has been stated in the following theorem and proved thereafter.

Theorem 4.2: The guessed free Sudoku solver *GTSPS* certifies a valid solution of a Sudoku puzzle P, if one exists, and it takes both time and space complexity $O(m^2 n^6 \sqrt{n})$, where $m = (p! - x)^{\sqrt{n}}$; p being the average number of blank cells in a minigrid of P of size $n \times n$. In fact, the assumed solver either declares if there is no (genuine) solution or computes all valid solutions of P, if there are one or more, using the same space and time.

Proof: The guessed free Sudoku solver *GTSPS* developed in this thesis considers all the minigrids of a given Sudoku instance P and computes only the valid permutations for each of the minigrids based on the given clues only. Hence, if there be a solution S for P, then each minigrid must has 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.

The algorithm has two parts: at first, it computes only valid permutations of each of the minigrids, and then it computes the graph G and its subsequent reduced graph G^*, etc. Now if p be the average number of blank cells of a minigrid in P, which is a Sudoku instance of order n, then the height of the computed tree is $O(p)$ and the number of vertices present at the lowermost level (i.e., the leaf level with only valid permutations) of the tree is $O(p!)$. Thus, the computational time as well as the computational space complexity of generating the tree is $O(p \times p!)$. Hence, for a P of size $n \times n$, as there are n minigrids, the overall computational time and space complexity is $O(n \times pp!)$.

Incidentally, the guessed free Sudoku solver generates a set of only valid permutations for each minigrid, and a required valid permutation for the minigrid in S must be a member of this set. Now if the average number of valid permutations per minigrid is q, then the total number of vertices introduced into the graph, G is $O(nq)$, and verifying the compatibility among the permutations along row and column minigrids takes time as well as space $O(n^2 q^2)$. Then the computation of all subsequent graphs and checking for degree of a vertex in G or G^*, deletion of vertices and edges, and so on and so forth, none of these tasks takes time as well as space more than $O(n^2 q^2)$.

Now, as the number of minigrids in each row is $\sqrt{n}$, therefore the total number of edges involved in checking compatibility among the minigrids in a row is ${}^{\sqrt{n}}C_2 = O(n)$. Then for all rows of minigrids (excluding columns) the total number of edges to be considered among valid permutations of row minigrids is $O(n\sqrt{n})$. Now the time required to compute a supervertex among a row of minigrids is $O((\sqrt{n})^2 \times n) = O(n^2)$. If m be the average number of supervertices for a row of minigrids, then the said complexity becomes $O(mn^2)$. As there are $\sqrt{n}$ number of row minigrids, the complexity to compute all supervertices of P is $O(mn^2\sqrt{n})$, following algorithm *GTSPS* (Version 4.2). Thus, the complexity to compute the total number of possible edges is $O((mn^2\sqrt{n})^2) = O(m^2n^5)$. Now to compute the compatibility between a pair of supervertices of two minigrids in two different rows we require $O(n\sqrt{n})$ cost. Hence, the complexity to verify the total compatibility becomes $O(m^2n^5 \times n\sqrt{n}) = O(m^2n^6\sqrt{n})$, where $m = (p!-x)^{\sqrt{n}}$ for an average number of p blank cells in a minigrid of P of size $n \times n$. ♦

It is worthwhile to mention that the algorithm developed in this chapter is also exponential time computable, but it is reasonable to accept such an algorithm where the values of the variables involved are bounded by some constant for a problem which is beyond polynomial time (or exponential time) computable (or intractable in nature) [45]. We have already mentioned that computing a solution of a given Sudoku puzzle is an NP-complete problem [10]. So, development of an exponential time algorithm is one of the adequate solving techniques, when the size of the Sudoku instance is reasonably small [17,45]. In fact, our experimentations take much less time in computing all solutions for instances of size 9×9.

Furthermore, in spite of the hypothetical exponential time computation of a tree while generating barely all valid permutations of the missing digits in a minigrid, in practice we find that the number of valid permutations is only a very few, as in reality, the size of the tree structure is extensively less than the asymptotic upper bound mentioned herein. Moreover, the same algorithm can easily be adopted for computing a desired solution for each of the instances of size 16×16 or 25×25, or more, instead of 9×9 as considered in this section of the thesis. We further claim that this is the first guessed free minigrid based Sudoku puzzle solver that algorithmically guarantees a solution, if one exists, for a given Sudoku instance in reasonable amount of clock (or CPU) time, and it is successful in computing all solutions if there are two or more.

4.5 Experimental Results

We have experienced and examined our algorithm with a large number of Sudoku instances in Intel Dual Core computing environment with the support of 3.0 GB RAM in Windows platform. In a bit more detail, we like to mention that we have considered 850 Sudoku instances in total out of which 200 are easy, 200 medium, 200 hard, and 200 are evil puzzles, and each of the 50 more puzzles have two or more solutions that are downloaded from [48]. For the same set of instances with different difficulty levels we have performed experimentations on our proposed *Graph Theoretic solver* (i.e. *GTSPS*), *Brute-Force solver* [47], and *Pencil-and-Paper based solver* [46]. The experimental results are highlighted in this section. We have compared our algorithm based on two parameters, like (i) Average number of iterations and (ii) Average CPU time. The results are very much interesting. We have found that our algorithm is best for the Sudoku puzzles with multiple solutions. We have taken 50 Sudoku instances with multiple solutions. Out of them, the instance shown in Figure 4.9 is having 127 distinct solutions and others are having less number of solutions.

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

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

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

Figure 4.9: (a) A Sudoku puzzle P. **(b)** Two of the valid solutions of P.

In Figure 4.9, we may observe that, the valid permutations that produce the solutions differ only in the first and second minigrids. Only the digits 2 and 7 change their positions in columns 1 and 5 in the first two rows of digits in these two minigrids, as these two digits are present in the third row of the third minigrid. Similarly, other 125 different other solutions can be produced by exchanging the positions of the digits present in different rows and columns. The comparative result is shown in Table 4.1. We have found that, our proposed graph based solver can detect all solutions at a single run, whereas *Brute-Force solver* [47] also can detect multiple solutions, but the program needed to be executed multiple times. Moreover, prior to computation of multiple solutions, we cannot tell about the number of solutions possible using the *Brute-Force solver*. Therefore, the CPU time and the number of iteration are too high. On the other hand, the *Pencil-and-Paper based solver* [46] cannot find multiple solutions. Therefore, we have not considered it for comparison.

From Table 4.1, we can see that the average number of iterations for the proposed *Graph Theoretic solver* (*GTSPS*) is pretty less than that needed for the Brute-Force solver. Similarly, we may notice that the average CPU time for the Brute-Force solver is 99.07 milliseconds, whereas for *GTSPS*, it is only 68.13 milliseconds. The total number of clues present in each instance ranges from 21 to 32.

Table 4.1: Comparison based on average number of iterations and CPU time for Sudoku instances with multiple solutions.

Sudoku Instances with Multiple Solutions	**Brute-Force Solver [47]**	**Proposed Graph Theoretic Solver (*GTSPS*)**
Average Number of Iterations	80881.72	43407.2
Average CPU Time (in milliseconds)	99.07	68.13
Number of Clues	21-32	

Table 4.2: Comparison based on average number of iterations and CPU time for Sudoku instances of difficulty level 'Easy'.

Easy Instances	**Brute-Force Solver [47]**	**Pencil-and-Paper based Solver [46]**	**Proposed Graph Theoretic Solver (*GTSPS*)**
Average Number of Iterations	8082.13	5525.07	11067.25
Average CPU Time (in milliseconds)	17.97	11.09	23.04
All Instances are Solvable?	Yes	Yes	Yes
Number of Clues	36-46		

Then experimentations are also carried out on the instances with Easy, Medium, Hard, and Evil difficulty levels. The summary of comparison is highlighted in Tables 4.2, 4.3, 4.4, and 4.5, respectively.

In Table 4.2, we can observe that the *Brute-Force solver* takes an average of 8082.13 iterations, whereas the *Pencil-and-Paper based solver* and the proposed *Graph Theoretic solver* (*GTSPS*) take 5525.07 and 11067.25 iterations, respectively, for solving easy instances, on an average. Similarly, the average CPU time needed for finding solutions of an instance is 17.97, 11.09, and 23.04 milliseconds using *Brute-Force solver* [47], *Pencil-and-Paper based solver* [46], and the proposed *Graph Theoretic solver*, respectively.

Table 4.3: Comparison based on average number of iterations and CPU time for Sudoku instances with difficulty level 'Medium'.

Medium Instances	**Brute-Force Solver [47]**	**Pencil-and-Paper based Solver [46]**	**Proposed Graph Theoretic Solver (*GTSPS*)**
Average Number of Iterations	22238.19	17512.58	25544.34
Average CPU Time (in milliseconds)	34.2	24.07	36.09
All Instances are Solvable?	Yes	Yes	Yes
Number of Clues	32-35		

In Table 4.3, we can observe that the average number of iterations for the *Brute-Force solver* [47], *Pencil-and-Paper based solver* [46], and the proposed *Graph Theoretic solver* (*GTSPS*) are 22238.19, 17512.58, and 25544.34, respectively. The average CPU time of 34.2 milliseconds, 24.07 milliseconds, and 36.09 milliseconds are consumed by *Brute-Force*, *Pencil-and-Paper based solver*, and the proposed *Graph Theoretic solver* for the given number of clues of 32 to 35 for the instances with difficulty level medium.

Table 4.4 highlights the comparison for different Sudoku solvers with difficulty level hard, having total number of clues ranging between 28 and 31. It is observed that the average number of iterations for the *Brute-Force solver* is 38215.18, whereas the *Pencil-and-Paper based solver* and the proposed *Graph Theoretic solver* (*GTSPS*) are having average number of iterations

26950.58 and 38706.07, respectively. The average CPU times consumed by the *Brute-Force solver*, *Pencil-and-Paper based solver,* and the proposed *Graph Theoretic solver* are 61.34, 33.1, and 61.97 milliseconds, respectively. Although the *Pencil-and-Paper based solver* takes very less time than other solvers, the most disadvantage of this solver is that it cannot solve all the hard Sudoku instances. Out of 200 instances with difficulty level hard, we have verified that the solver fails solve 37 instances, which is around 19% of the total assumed instances.

Table 4.4: Comparison based on average number of iterations and CPU time for Sudoku instances with difficulty level 'Hard'.

Hard Instances	**Brute-Force Solver [47]**	**Pencil-and-Paper based Solver [46]**	**Proposed Graph Theoretic Solver (*GTSPS*)**
Average Number of Iterations	38215.18	26950.58	38706.07
Average CPU Time (in milliseconds)	61.34	33.1	61.97
All Instances are Solvable?	Yes	No	Yes
Number of Clues	28-31		

Table 4.5: Comparison based on average number of iterations and CPU time for Sudoku instances with difficulty level 'Evil'.

Evil Instances	**Brute-Force Solver [47]**	**Pencil-and-Paper based Solver [46]**	**Proposed Graph Theoretic Solver (*GTSPS*)**
Average Number of Iterations	53292.1	33981.27	46164.67
Average CPU Time (in milliseconds)	84.07	42.98	78.7
All Instances are Solvable?	Yes	No	Yes
Number of Clues		17-27	

From Table 4.5, we can observe that the average number of iterations and the average CPU time taken by our proposed *Graph Theoretic solver* (*GTSPS*) are 46164.67 and 78.7 milliseconds only. The *Pencil-and-Paper based solver* and *Brute-Force solver* take 42.98 and 84.07 milliseconds of CPU time, respectively. The average numbers of iterations for the *Brute-Force* *solver* and *Pencil-and-Paper based solver* are 53292.1 and 33981.27, respectively. Thus, we can easily notice that, the *Pencil-and-Paper based solver* requires the least number of iterations and it consumes the least CPU time, even though we have noticed that out of 200 instances, 73 instances are not solvable by this solver, which is roughly 37% of the total evil instances.

Table 4.6: Comparison based on average number of nodes in the initial graph structure obtained using of the proposed *Graph Theoretic solver* (*GTSPS*).

Difficulty Level	**Average Number of Nodes in the Graph**
Easy	227
Medium	358
Hard	528
Evil	720

The average number of nodes in the initial graph structure is shown in Table 4.6. It actually reflects the number of valid permutations based on the given clues. We can eventually understand that the average numbers of valid permutations for puzzles with difficulty level Easy, Medium, Hard, and Evil instances are 228, 358, 528, and 720, respectively. From this table, we also notice that the total number of valid permutations is greatly less than the actual number of possible permutations, as the instances are bounded by constraints (or clues). Therefore, the proposed *Graph Theoretic solver* (*GTSPS*) takes very less time in the order of milliseconds.

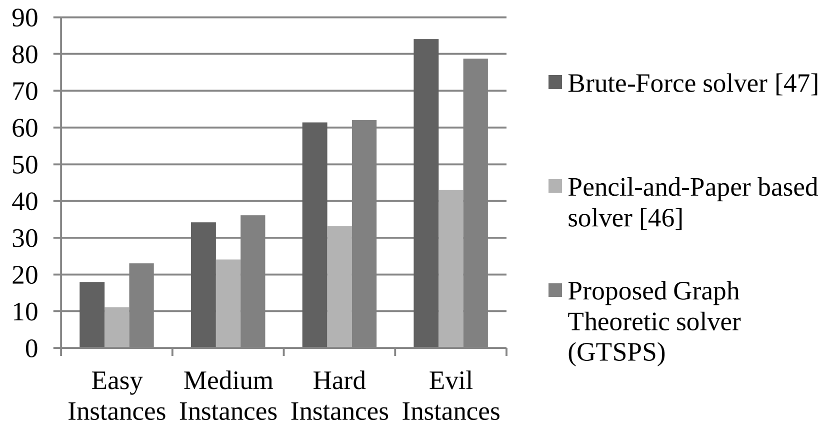

Figure 4.10: Comparative bar chart based on average CPU times (in milliseconds).

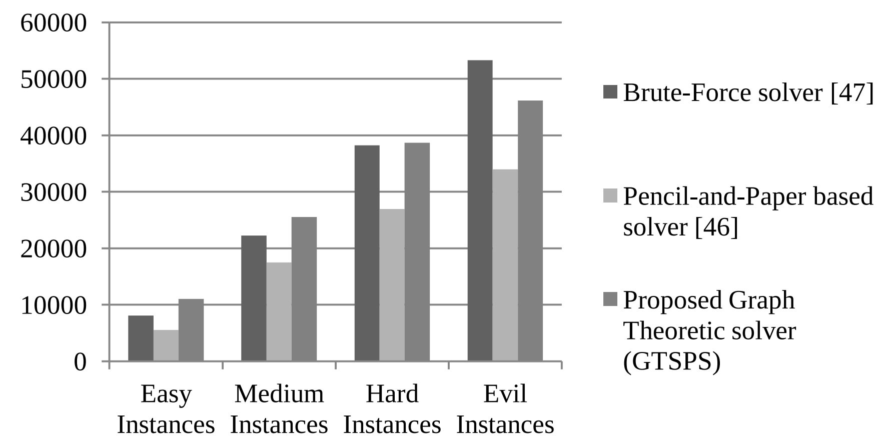

Figure 4.11: Comparative bar chart based on average numbers of iterations.

4.5.1 Analysis of Experimental Results

The comparative bar chart based on average CPU times (in milliseconds) and average numbers of iterations for *Brute-Force solver* [47], *Pencil-and-Paper based solver* [46], and the proposed *Graph Theoretic solver* (*GTSPS*) are highlighted in Figures 4.10 and 4.11, respectively.

From Figures 4.10 and 4.11, we can easily understand that the *Pencil-and-Paper based solver* can solve a given Sudoku puzzle with least iterations and consuming least CPU time. However, this solver cannot solve all the puzzle instances of difficulty levels Hard and Evil. Our proposed *Graph Theoretic solver* and the *Brute-Force solver* can solve all the instances. For Easy and Medium instances, although the *Brute-Force solver* takes less CPU time and number of iterations on an average, for Hard instances there are only marginal differences for these two parameters in comparison to our proposed solver. Our proposed solver (i.e. *GTSPS*) takes less CPU time and number of iterations than those of the *Brute-Force solver*, on an average, for the puzzles with difficulty level Evil. Results show that the performance of our proposed algorithm is best for solving Sudoku puzzles with multiple solutions. Both *Brute-Force* and the proposed solver can compute multiple solutions. However, as our proposed solver, *GTSPS*, can find multiple solutions in a single run, the *Brute-Force solver* is required to run multiple times, and thus takes more iterations and more CPU time.

We have also compared our algorithm with the solver proposed by Andrew-Stuart [49]. It has been found that, all Sudoku instances, which are categorized as ‘Extreme’, are not solvable by this solver [49]. We have found that all such 40 available puzzle instances, which are categorized as ‘Extreme’, are straightforwardly solvable by our solver with an average CPU time of 19 seconds only though the average number of iterations is large enough, roughly 21836771. We have also noticed for the *Peter-Norvig’s solver* [50] that takes the CPU time for solving the most time consuming puzzle is 1439 seconds, our solver solves the same instance within 312 seconds only (having the number of iterations 22119736).

The work on the proposed graph theoretic Sudoku solver (i.e. *GTSPS*) is a *complete* Sudoku solver, where the concept of the level of difficulty for such a puzzle is relatively less relevant. The *Pencil-and-Paper based solver*, *Peter-Norvig’s solver*, or the *Andrew-Stuart’s solver* used different techniques for puzzles of different difficulty levels. Rather, the Sudoku solver, devised in this thesis, treats all the puzzles likewise and views all of them from a different height. Parenthetically, there exist as many as 6,670,903,752,021,072,936,960 distinct Sudoku puzzles of size 9×9 each [37], and we have considered only a few of them. Based upon the Sudoku instances under consideration, sometimes the numbers of iterations and CPU times may vary.

We further observe that, the average time of computing a Sudoku instance using the *Pencil-and-Paper based solver* [46] is ~27.81 milliseconds, whereas that for the *Brute-Force solver* [47] is ~49.40 milliseconds and for the proposed *Graph Theoretic solver* (*GTSPS*) is ~49.95 milliseconds. Thus, for solving all the existing Sudoku instances, as mentioned above, the *Pencil-and-Paper based solver* would take time approximately 5,882,731,904,607 years and the *Brute-Force solver* would take approximately 10,449,728,733,822 years, whereas our proposed *Graph Theoretic solver* (*GTSPS*) would take about 10,566,071,867,499 years. So, sampling all Sudoku instances are also not promising by any of the solvers. Random sampling does not show the actual experimental results. As we have noticed that the experimental results are very much dependent on instances. We have found several other factors to compare our algorithm with many others existing in literature, as mentioned and briefly detailed in the following section.

4.5.2 Comparative Judgement among Different Sudoku Solvers

In this chapter, we have developed a Sudoku solver that we claim as *unique* and *complete* in many respects in comparison to all earlier Sudoku solvers; we like to mention some of these as follows and include in Table 4.7 for several measuring criteria.

We have taken into consideration different other Sudoku solvers, such as Pencil-and-Paper based solver [46], Brute-Force solver [47], solver by Andrew Stuart [49], and solver by Peter Nerving [50]. Then we made a comparative judgement amongst all of them, which is shown in Table 4.7.

The proposed Sudoku solver is the first as well as exclusive graph theoretic algorithm that could be implemented with the help of a computer in a systematic way. The proposed method is completely guessed free, as stated several times, which considers minigrids only (instead of blank cells) of size 3×3 each that has been devised for the first time in designing such an algorithm for a given Sudoku instance of size 9×9.

In our proposed algorithm a pre-processing of computing all valid permutations only is there for all the minigrids based on the clues in a given Sudoku instance. It has been observed in most practical situations that the number of valid permutations is significantly 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 radically help in reducing the ultimate number of valid permutations for that minigrid too. Anyway, this technique 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 cells, therefore, the computations involved in the algorithm is significantly reduced. In the case of a 9×9 Sudoku puzzle, there are 81 cells with some clues (which is less) and the remaining blank cells (which is more), whereas there are only nine minigrids each of which consists of 3×3 cells. Here the mode of viewing a Sudoku puzzle is not by searching of missing numbers cell-by-cell (as if searching for an address by moving through streets or lanes), 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 who is flying in the sky).

In spite of the controversy that a Sudoku puzzle must have only one unique solution [51], a given Sudoku instance may have two or more valid solutions that finds application in different domains of research. The devised Sudoku solver in this chapter is capable of computing all those solutions, if they exist. The proposed method is unique in many a sense like the way we have observed the problem under consideration (i.e. to solve a given Sudoku puzzle), the application of graph as a tool to solve this problem, the extracted graph composition (e.g., as shown in Figure 4.1 for a Sudoku puzzle of size 9×9) and its presence in the overall graph theoretic representation for some instance of the puzzle, and the technique we have anticipated to solve the same. In fact, the algorithm is capable of computing all those solutions, if there are two or more. These are just a few; detailed comparative debates could be observed in Table 4.7 for several other parameters or measuring issues.

Table 4.7: The novelty of *GTSPS*, the guessed free Sudoku solver developed in this thesis in comparison to other existing Sudoku solvers for different measuring issues.

Sl. No.	**Parameters / Measuring Issues**	**Pencil-and-Paper based Solver** [46]	**Brute-Force Solver** [47]	**Solver by Andrew Stuart** [49]	**Solver by Peter Nerving** [50]	***GTSPS:* The Proposed Sudoku Solver**
1.	Techniques / Methods used	Twin, Triplet, Locked candidates, Hidden subset, Lone ranger, Naked subsets, etc.	Backtracking	All the methods used in [46] along with X-Wing, XY-Wing, Swordfish, XYZ-Wing, etc.	Backtracking, Constraint propagation, and Searching	Permutation generation, Graph theory
2.	Type of Algorithm	Guess based	Guess based	Guess based	Guess based	Guessed free
3.	Filling Criterion	Cell based	Cell based	Cell based	Cell based	Minigrid based
4.	Application of Graph in Viewing a Sudoku Puzzle	No	No	No	No	Yes
5.	Ability to Identify Whether a Given Puzzle is Valid	No	No	No	No	Yes
6.	Ability to Identify All Solutions, if Two or More Solutions Exist	No	Yes, but extremely time consuming	No	No	Yes
7.	Number of Solutions Generated (at a Time)	Only one	Only one	Only one	Only one	All solutions, if there are many
8.	Whether Able to Solve Sudoku Puzzles of Any Level of Difficulty	No	Yes, but redundant computations are involved	No	Yes, as the author has claimed	Yes, computes only the minigrid based irredundant permutations
9.	Whether the Same Algorithmic	No	Yes, but extremely time consuming	No	Yes	Yes

	Technique is Applicable to Solve Any Sudoku Puzzle					
10.	Modularity of the Algorithmic Technique Adopted	No; applicable only for 9×9 Sudoku puzzles	Yes; applicable for any larger Sudoku puzzle	No; applicable only for 9×9 Sudoku puzzles	No; applicable only for 9×9 Sudoku puzzles	Yes; applicable for any larger Sudoku puzzle
11.	Applicability in Steganographic Information Processing	No	No	No	No	Yes

A Sudoku instance might have two or more solutions that have some inherent usefulness in one application where we like to hide some information. Particularly, let us assume a Sudoku instance is sent with some concealed information. An interloper may access the instance and may make one valid solution of this instance. By the way, this solution may not interpret the concealed information as it is kept in some other valid solution of the same instance.

We have observed that, even if there are more clues, a Sudoku puzzle may produce more solutions. If there are many solutions for some given Sudoku instance, our proposed method does not incur any additional cost for that; all these solutions can be produced at the same time.

The brilliance of the devised algorithm is that the same logic can also be straight away applied in a modular form for larger Sudoku instances such as 16×16, 25×25, or of any other rectangular size. Similar logic of extracting an appropriate graph structure (some sort of graph as shown in Figure 4.1 for a 9×9 puzzle) based on a larger Sudoku instance under consideration is to be carried out. As for example, for a 25×25 puzzle, there are five minigrids of size 5×5 each in each row and five minigrids of size 5×5 each in each column, making a total of 25 minigrids in such an instance. So, here the graph structure (to be extracted) is of a 5×5 grid, where each set of five vertices in a row must form a complete graph (or a clique of size five) and each set of five vertices in a column must also form a complete graph (or a clique of size five). Now the task of finding such a graph isomorphic to a subgraph of a bigger one that we compute for some larger Sudoku instance is particularly difficult, tedious, and time-consuming that such algorithmic line of attack (as invented in the thesis) may solve in a reasonable amount of time.

The level of difficulty is another important issue that almost all the earlier Sudoku solvers consider while developing an algorithm or a technique towards computing a solution. 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 solve. 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 as Easy, Medium, Difficult, Evil, etc. Incidentally, the Sudoku solver developed in this chapter needs further investigation to reopen the issue of different difficulty levels, as its angle of viewing and judging such instances are different. 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 eventually guide to compute all the desired solutions for a given Sudoku instance.

4.6 Relative Comparisons among the Versions of the Newly Devised Sudoku Solver

In this thesis, four versions of a new Sudoku solver have been devised and discussed. Two versions (that are Versions 3.1 and 3.2) are included in Chapter 3, whereas the other two versions of the solver (that are Versions 4.1 and 4.2) are incorporated in this chapter. The initial version of the algorithm (i.e. Version 3.1) starts generating valid permutations with a minigrid having more given clues. As a result, the number of valid permutations generated is much less. The key limitation of this 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 contrast to the Version 3.1, Version 3.2 assumes any sequence S_M of minigrids which can be either zigzag, or spiral, or semi-spiral. 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 in isolation. Then, based on the sequence S_M, the starting minigrid is chosen and a valid permutation for that minigrid is considered. 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). 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 matching 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).

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). Here we can observe that the numbers of redundant permutations are less in number. But both the versions of the devised Sudoku solver are unable to find multiple solutions in a single run, if it exists. Also it fails to find out whether a Sudoku puzzle is having a valid solution or not.

In Version 4.1, we formulate the generalized algorithm for solving a Sudoku instance, using a simple, symmetric graph, $G = (V, E)$, where the graph structure consists of nine sets of vertices, and each such set comprises and represents a set of valid permutations for a minigrid of a given Sudoku puzzle P. In some other words, if there is (are) $p \geq 1$ valid permutation(s) for minigrid M_i, $1 \leq i \leq 9$, then we introduce p vertices into the graph, where each vertex represents a unique permutation of M_i. This means, if the total number of valid permutations for all individual nine minigrids be m, then the graph contains $m = |V|$ vertices. We may recall that all valid permutations for a minigrid are generated only for the given clues in P, independently for all the nine minigrids of the given puzzle, as we computed permutations in Version 3.2 of the algorithm in Chapter 3.

Table 4.8: A table of comparison among the different versions of the devised Sudoku solver for a Sudoku instance P of size $n \times n$, p and x being the average number of blank cells and the average number of invalid permutations computed for a minigrid.

Sl. No.	**Parameters**	**Version 3.1**	**Version 3.2**	**Version 4.1**	**Version 4.2**
1.	Algorithmic Technique	Backtracking based on non-adjacency among the minigrids	Backtracking based on adjacency among the minigrids	Without backtracking; based on sub-graph isomorphism	Without backtracking; based on complete sub-graph isomorphism of size $\sqrt{n}$
2.	Representational structure	Tree	Tree	Graph	Graph
3.	Number of solutions generated	One (at a time in one run)	One (at a time in one run)	All at a time, if they exist	All at a time, if they exist
4.	Validity of a given Sudoku instance	Cannot check	Cannot check	Can check	Can check
5.	Sequence of choosing minigrids	Dense minigrids with more clues are chosen first	Zigzag, Spiral, and Semi-Spiral	Does not arise	Does not arise
6.	Redundant permutations	More	Less	Less	Less
7.	Number of minigrids considered in each vertex	Does not occur	Does not occur	One	Three
8.	Time and Space Complexity	$O(p!)^{\alpha n}$), where $0 < \alpha \leq 1$	$O(p!)^{\alpha n}$), where $0 < \alpha \leq 1$	$O(\sqrt{n}q^{3n/2})$, where $q = p! - x$	$O(m^2 n^6 \sqrt{n})$, where $m = (p! - x)^{\sqrt{n}}$

A more compacted representation of Version 4.1 is discussed in Version 4.2 of the Sudoku solver; here instead of representing only a valid permutation of a minigrid, a node represents valid permutations of three compatible minigrids in each row and in each column. In a summary, all these versions of the devised Sudoku solver are compared in Table 4.8, for a set of reasonable parameters included in the table.

4.7 Summary

In this chapter, we have designed an exclusive graph theory based Sudoku solver in two succeeding versions (Versions 4.1 and 4.2) of the same (algorithm), which are entirely 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 this algorithm, a pre-processing is there for computing only all valid permutations of the missing digits 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 rigorously help in reducing the eventual number of valid permutations for that minigrid too. The devised algorithmic procedure of minigrid based computation of valid permutations of the missing digits (in a minigrid) and checking their compatibility among row minigrids and column minigrids is completely new and done for the first time in this domain of work for solving a given Sudoku puzzle.

As we consider minigrids for finding only the valid solutions of a given Sudoku puzzle instead of considering the individual (blank) cells, therefore, the amount of 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 (or digits) cell-wise (as if searching for an address by walking through lanes); rather, it is one step above the ground of the puzzle by considering groups of cells or minigrids (where searching the same is made from a bird's eye view who is flying in the sky).

In spite of the controversy that a Sudoku puzzle must have only one unique solution [25], we show that a given Sudoku instance may have two or more valid solutions as well, and in practice it also finds applications in different domains of importance. The Sudoku solver developed in this chapter is capable of computing all those solutions, if there exist two or more, not including any extra cost. Even in a case when a supposed Sudoku instance is given without having a single valid solution, this algorithm is able to identify that as well without any additional algorithmic cost of time and space.

A Sudoku instance might have two or more solutions that have some inherent usefulness in one application where we like to hide some information. Particularly, let us assume a baffling Sudoku puzzle is sent with some concealed information. An interloper may access the instance and may make one valid solution of this puzzle. By the way, this solution may not interpret the concealed information as it is kept in some other valid solution of the same instance. The concealed information could also be distributed over all the $x \geq 2$ number of valid solutions of an assumed Sudoku instance in some mysterious enigmatic fashion.

The novelty of the algorithm devised herein is that the algorithm is modular in nature; the same logic can also be straightway applied for larger Sudoku instances such as 16×16, 25×25, or of any other rectangular size (with their respective objective function). Similar logic of extracting an appropriate graph structure (some sort of the graph in Figure 4.1 for a 9×9 puzzle) based on a larger Sudoku instance under consideration is to be executed. We may right away guess that the associated tasks for larger Sudoku instances are extremely difficult, tedious, and time consuming that such algorithmic methodology (as developed in this chapter) may solve in a reasonable amount of clock time.

The level of difficulty is another important issue that almost all the earlier Sudoku solvers consider while developing an algorithm. They use different techniques for different difficulty levels of the Sudoku puzzle. However, 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 as Easy, Medium, Hard, Evil, etc. By the way, the Sudoku solver designed in this chapter does not depict any level of difficulty, as a so-called Hard instance may take less CPU time and a Medium instance might require more iterations. Our devised solver follows the same algorithmic technique for solving a Sudoku puzzle of any level of difficulty.