Soft Computing (2019) 23:6585–6601
https://doi.org/10.1007/s00500-018-3307-6

METHODOLOGIES AND APPLICATION

General variable neighborhood search for solving Sudoku puzzles: unfiltered and filtered models

Aise Zulal Sevkli[1] · Khorshid Adel Hamza[2]

Published online: 15 June 2018

Abstract
In this paper, two novel models based on the Variable Neighborhood Search (VNS) algorithm are proposed to solve Sudoku puzzles. For the first model (Unfiltered-VNS), four neighborhood structures are proposed. The Filtered-VNS, which is the second model, uses filtering to reduce the number of partial infeasible solutions in the search area. Local search is performed by a novel mutation-based neighborhood structure. In both models, the neighborhood structures are implemented by using different local search improvement strategies. Two proposed models with the best configurations are tested on 57 well-known Sudoku benchmarks. The experimental results indicate that our models can solve all benchmarks. For easy-, medium-, and hard-level puzzles, Filtered-VNS shows better solution quality than Unfiltered-VNS. For very hard instances, performance of Unfiltered-VNS is better than Filtered-VNS. Except for two of the 57 benchmarks, Filtered-VNS improves the solution qualities of the previous studies.

Keywords Metaheuristic · Sudoku puzzles · Variable neighborhood search · Filtering

1 Introduction

The Sudoku puzzle game is popular around the world and can be found in most newspapers and on puzzle websites. It is also widely present in the world of mobile applications, which is evidenced by the large number of Sudoku applications in the Google Play store and the Apple Store. Today, millions of websites offer online Sudoku puzzles, and several free generator software packages are available, reflecting the fact that puzzlers find Sudoku puzzles to be highly entertaining (Gizmos Reviews 2014).

Although there are many Sudoku varieties, such as Shidoku, Squiggly, Jigsaw Sudoku, we study basic Sudoku puzzles that consist of n × n grids (usually 9 × 9 grids) and have n rows, n columns, and n sub-grids with some cells that already contain numbers, known as “givens” or “fixed”; these numbers cannot be changed or moved through the solving process. The rows, columns, and sub-grids on a sample puzzle are shown in Fig. 1. The player aims to fill all sub-grids with numbers from 1 to n (1–9 in our example, Fig. 1), but this should be done carefully with respect to the following three rules:

1. Each row contains the integers 1 to n exactly once.
2. Each column contains the integers 1 to n exactly once.
3. Each sub-grid contains the integers 1 to n exactly once.

From the theoretical point of view of computer science, (Yato 2003) proved that Sudoku is NP-complete by using a reduction from a Latin square to Sudoku. The difficulty of a game instance depends on many factors; there are 15–20 factors that are claimed to have an effect on the difficulty rating (Deng et al. 2012). The positioning of the given numbers on the board constitutes an important difficulty factor in addition to the number of fixed cells.

A Sudoku board can have more than one solution, which we call ambiguous. Some Sudoku problems have no solutions at all, and we call them invalid. In this paper, we deal with “proper” Sudoku puzzles, which have only one solution. Additionally, the puzzle has different levels, such as very easy, easy, medium, hard, and very hard.

Communicated by V. Loia.

✉ Aise Zulal Sevkli
zsevkli@gmail.com; sevklia@denison.edu

Khorshid Adel Hamza
Khorshid.adel@epu.edu.krd

[1] Department of Computer Science, Denison University, Granville, OH, USA

[2] Department of Managment, Erbil Polytechnic University, Erbil, Iraq

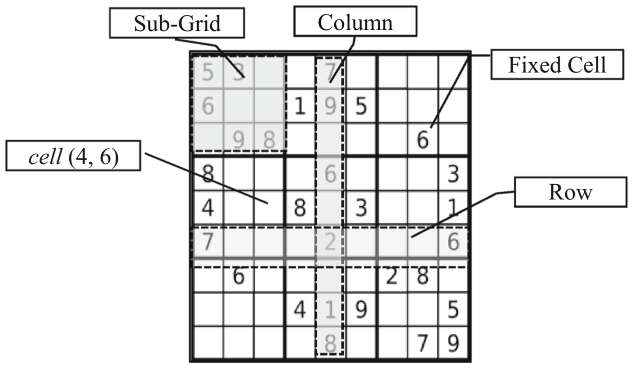


Fig. 1 A sample board of the Sudoku puzzle

Recently, many researchers have applied metaheuristic algorithms to solve Sudoku puzzles by modifying the general templates of the algorithms to fit the specifications of the problem. The differences between these researchers' results are seen in the rate of success, the number of iterations, and the amount of time needed to reach an optimal solution. When we study to previous research on solving Sudoku puzzles, we find that the most of those studies used population-based metaheuristics, such as the genetic algorithm, the harmony algorithm, ant colony optimization, to solve this problem. On the other hand, there are few single-solution-based metaheuristics implemented in this field. Since VNS which is a single-solution-based metaheuristic has few parameters and an easy implementation feature, it is widely used to solve many combinatorial optimization problems (Sevkli and Sevilgen 2006; Hansen and Mladenović 2010; Sevkli and Guler 2017). In this paper, we propose two novel VNS-based metaheuristic algorithms to solve different levels of Sudoku problems. To the best of our knowledge, this study is the first implementation of VNS for solving Sudoku. It presents new strategies (filtering, ranking, continuous improvement) and new components (five neighborhood structures) for VNS. The positive effects of new strategies and components on solution quality and processing time are demonstrated by testing them on 57 well-known Sudoku benchmarks.

The remainder of this paper is organized into five additional sections: Sect. 2 explains previous studies on Sudoku puzzles. VNS and its variations are explained in Sect. 3. Section 4 gives details of the proposed algorithms. Experimental problem sets and results are given in Sect. 5. Finally, Sect. 6 contains the discussion and concluding remarks.

2 Literature review

The literature contains many studies on solving Sudoku puzzles. Backtracking is a straightforward approach to solving Sudoku puzzle problems, but it cannot solve the difficult problem levels in a reasonable period of time, because Sudoku is known as NP-complete. Constraint programming and elimination-based algorithms can efficiently solve Sudoku, and they can also be used to generate Sudoku problems with different levels.

Recently, various metaheuristics have been proposed for solving Sudoku puzzles (Mishra et al. 2018). A list of previous studies is given in Table 1.

Bartlett et al. (2008) developed a mathematical model which presents a binary integer linear program using MATLAB software to solve Sudoku. Furthermore, their program could be used to solve variations of the traditional Sudoku puzzles. The Mathlab program could solve a Sudoku instance with 25 givens in 16.08 s. Unfortunately, there are no more experimental results on the other instances in the paper.

Mantere and Koljonen proposed two population-based algorithms: Genetic Algorithm (GA) (Mantere and Koljonen 2007) and Cultural Algorithm (CA) (Mantere and Koljonen 2008). The GA was tested 100 times with 27 Sudoku puzzles (see Table 6 for puzzles 1, 2, 3, 4, 5, E, C, D, and SD). The GA solved all puzzles; furthermore, it solved the hardest puzzle (Al Escargot) with a 100% success rate without giving any termination criteria. The same algorithm solved the same problem with just an 18% success rate when the termination criteria was given as 100,000 iterations. The CA is a special variation of evolutionary algorithms. The main difference between the CA and the GA is the use of a belief space. The CA improved the GA results by up to 8.67%.

A three additional GA-based algorithms were proposed by Deng et al. (2012): Improved Genetic Algorithm (IGA), New Genetic Algorithm (NGA), and Hybrid Genetic Algorithm (HGA). The IGA and the NGA were tested on four difficulty levels (easy, medium, difficult, and super difficult). The NGA showed better results as it solved the first two levels with an 80% success rate against the 17% success rate of the IGA, but both algorithms were disappointing when it came to the difficult and the super difficult puzzles—the NGA solved them with an 8% success rate, while the IGA was not able to solve them at all. The NGA results were compared to those of the GA (Mantere and Koljonen 2008). In general, the NGA has a better success rate and fewer numbers of generation. The HGA was able to improve the previous results of all previous algorithms significantly, even at the difficult levels.

The implementation of the Artificial Bee Colony (ABC) on Sudoku puzzles was proposed by Pacurib et al. (2009). The algorithm was tested on three Sudoku instances: one easy, one medium, and one hard. It was able to find optimal solutions with a success rate of 100%. In addition, it showed excellent results when tested on the Al Escargot problem.

The Cuckoo Search (CS) algorithm is relatively new. It was implemented by Soto et al. (2014) to solve Sudoku problems. The CS was tested on nine samples (easy-s10, medium-s11, and hard-s12) from the dataset proposed by Mantere and Koljonen (2007). The algorithm, with a maxi-

Table 1 List of algorithms applied to solve Sudoku

Type	Algorithm	References
Exact methodologies	An integer programming model for the Sudoku problem	Bartlett et al. (2008)
Trajectory-based	Simulated annealing (SA)	Lewis (2007)
Metaheuristics	Variable neighborhood search (VNS)	Hamza and Sevkli (2014)
Population-based metaheuristics	Genetic algorithm (GA)	Mantere and Koljonen (2007)
		Mantere and Koljonen (2007)
		Deng et al. (2011)
		Deng et al. (2012)
		Deng et al. (2013)
		Deng et al. (2013)
	Harmony search (HS)	Geem (2007)
		Mandal and Sadhu (2011)
		Mandal and Sadhu (2011)
	Cultural algorithm (CA)	Mantere and Koljonen (2008)
	Particle swarm optimization (PSO)	Hereford and Gerlach (2008)
		Singh and Deep (2016)
	Artificial bee colony (ABC)	Pacurib et al. (2009)
	Ant colony (AC)	Asif and Baig (2009)
	Differential evolutions (DE)	Boryczka and Juszczuk (2012)
	Genetic programming approach (GPA)	Pillay (2012)
	Cuckoo search (CS)	Soto et al. (2014)
Hybrid	Cultural genetic algorithm (CGA)	Perez and Marwala (2008)
	Repulsive particle swarm optimization (RPSO)	
	Quantum simulated annealing (QSA)	
	Genetic algorithm with simulated annealing (GASA)	
	AC3-tabu	Soto et al. (2013)

mum of 10,000 iterations, solved all three easy problems with a 100% success rate. However, for the medium level, it had a success rate of 59%, and it had 51.54% for the hard level.

Another algorithm was published by Mandal and Sadhu (2011) using the Harmony Search (HS) algorithm. This algorithm, in addition to solving different levels of Sudoku, can determine the level of the Sudoku game. The results were very good when solving easy levels, with a success rate of 100%, but these rates decreased to 10% as the difficulty increased to the hardest level.

Other metaheuristic implementations, such as ACO (citealtAsif9) and PSO (Hereford and Gerlach 2008), do not have a satisfactory solution quality. Recently, PSO with membrane algorithm (Singh and Deep 2016) is proposed to solve Sudoku. It shows efficient performance for 15 randomly chosen easy and medium level problems with success rate of 93.3 and of 84.6%, respectively. For hard-level puzzles, the success rate decreases to 47.3%.

There are also many hybrid algorithms that have been developed to solve Sudoku puzzles. Perez and Marwala (2008) developed a series of stochastic optimization approaches: Cultural Genetic Algorithm (CGA) with two versions, Repulsive Particle Swarm Optimization (RPSO), Quantum Simulated Annealing (QSA), and the Hybrid method, which combines the Genetic Algorithm with Simulated Annealing (HGASA). A hybrid AC3-tabu (Soto et al. 2013) merges a classic Tabu Search algorithm (TS) with an AC3 filtering procedure to remove the infeasible values from the search space. The AC3 algorithm in the hybrid implementation is employed on both the preprocessing phase and the filtering process in the TS.

Although there are many population-based heuristic algorithms, there are few trajectory-based (single-solution-based search) algorithms for solving Sudoku games. One of them is Simulated Annealing (SA) algorithm proposed by Lewis (2007). It solves different levels of (3 * 3) and (4 * 4) Sudoku games. Lewis also developed an algorithm to create solvable problem instances with different levels by using some rules and criteria. The results were amazing, as the algorithm solved all samples of all levels in a short time, although it needed more times to solve the 4 * 4 Sudoku problems. The main factor in reducing running time was calculating the fitness function by recalculating only affected rows and columns with the swap neighborhood structure operation. Unfortunately, the dataset he used in his experiments is unavailable for comparison of his results with ours. Recently, as a preliminary study of this work, we presented a trajectory-based VNS algorithm to solve Sudoku puzzles (Hamza and Sevkli 2014).

3 Variable neighborhood search

The Variable Neighborhood Search (VNS) algorithm was proposed by Hamza and Sevkli (2014). The basic idea of VNS is to successively explore a set of predefined neighborhoods to provide a better diversification of solutions. It systematically explores a set of neighborhoods to get different local optima and to escape from local optima. The VNS algorithm has three main phases: Shake, Local Search, and Move or Not. While Shake diversifies the solution, Local Search explores local areas thoroughly. The idea in VNS involves iterative exploration using a predefined set of neighborhoods (N_k where k : 1 to kmax) in Shake, generally ordered from small to large coverage areas ($N_1(\mathrm{S}) \subseteq N_2(\mathrm{S}) \subseteq \ldots \subseteq N_{\mathrm{kmax}}(S)$, S: solution). In VNS, the next candidate for a solution from the current neighborhood can be found by selecting a random element (shaking) from the same neighborhood (S'). Then, a Local Search is applied to improve it (S''). After that, depending on the quality of the solution (fitness function), we can choose S'' if it is better than S' and start from the first NS. If not, we move to the next NS. This procedure is repeated until we reach kmax without improving. All these operations can be iterated until a stopping condition is met. The stopping condition can be a maximum number of iterations for the whole process or a period of time for the current solution in which no improvement has been observed.

VNS is used as a Local Search algorithm if Shake is removed from the algorithm. This variation of VNS is called Variable Neighborhood Descent (VND), which thoroughly explores the current neighborhood before moving to the next neighborhood. The general VNS (GVNS) is another variation in which VND is used as a Local Search in VNS. The algorithm starts by defining two sets of neighborhood structures: $N_k(k = 1, \ldots, k_{\max})$ and $M_j(j = 1, \ldots, j_{\max})$. At each iteration, an initial solution is shaken by the current neighborhood N_k to generate a solution S' in the current neighborhood $N_k(\mathrm{S})$ set. VND is applied to the solution S' to generate the solution S''. The current solution is replaced by the new local optima S'' if and only if a better solution has been found. The pseudocode of GVNS is given in Fig. 2.

4 GVNS-based proposed models

In this section, two GVNS-based models are presented. Solution representation, initialization, and calculation of fitness value are the common subjects for two models. The other components of GVNS are explained for each model separately. To determine the configuration of each model, many schemas are experimented. In this paper, only the configuration for each model that produces the best results on experiments is presented. The other experimented schemas and their results can be found in Hamza (2015).

Fig. 2 Pseudocode of GVNS

```
Input: a set of neighborhood structures $N_k$ for k = 1,, $k_{max}$ for shaking.
          a set of neighborhood structures $M_j$ for j = 1,.. $j_{max}$ for local search.
S = $S_0$;
Repeat
   For k=1 to $k_{max}$ Do
      Shaking: pick a random solution S′ from the $k^{th}$ neighborhood $N_k(x)$ of S;
      Local search by VND;
         For j=1 To $j_{max}$ Do
            Find the best neighbor S″ of S′ in $M_j(S′)$;
            If Fitness (S″) is better than Fitness (S′) Then S′= S″ ; j=1;
            Otherwise j=j+1;
      Move or not:
         If S″ is better than S Then S = S″
            Continue to search with $N_k$ (k = 1);
         Otherwise k=k+1;
   End For
Until stopping criteria
Return S
```

4.1 Solution representation

We keep the same game board structure and game rules that are explained in the introduction section for representation of the solution. Each sub-grid is considered a one-dimensional array. We present each number in the Sudoku game board with a cell (i, j), in which *i* denotes the sub-grid number and *j* denotes the cell position (1–9) in the sub-grid. For instance, cell (4, 6) refers to the value of the cell whose sub-grid number is 4 and position number is 6. Figure 1 shows the position of the cell (4, 6).

4.2 Initialization

Each empty cell has a list of possible numbers that includes all the numbers that could be assigned to a cell without violating the three rules with respect to the fixed cell's value. Initialization is performed by selecting a random value from the list that constructed for each empty cell. For instance, for cell (2, 1) in Fig. 3, only the values {1, 8} are available because:

The fixed numbers in the first row are {2, 3, 4, 5, 7} (first rule).

The fixed numbers in the fourth column are {7, 9} (second rule)

The fixed numbers in the second sub-grid are {2, 6, 9} (third rule)

The union of sets of three rules is {2, 3, 4, 5, 6, 7, 9}. Finally, {1, 2, 3, 4, 5, 6, 7, 8, 9} \ {2, 3, 4, 5, 6, 7, 9} results in {1, 8}. By repeating this process for all other empty cells, we get possible values as they appear in Fig. 3. To obtain an initial solution, one of the possible values is randomly assigned to each cell.

We can reduce the number of cells to be filled by fixing the cells that have only one possible value. Cell (2, 3) in Fig. 3 can be fixed because it has only one possible value {1}. After fixing this cell, we can eliminate all {1} values from the cells on the same row, column, and sub-grid. The process of elimination and fixing one-possible-value cells is repeated until no more one-possible-value is found.

4.3 Fitness function

The initial solution is created based on rule 3, which means the numbers between 1 and 9 are not repeated in each sub-grid. It is clear that the appropriate fitness function is one that searches for violations of the remaining two rules (rules 1 and 2). To find the fitness value, repeated numbers in each row and column are calculated. The total numbers give the fitness value of the candidate solution. The calculation of the fitness value for a sample Sudoku puzzle is given in Fig. 4.

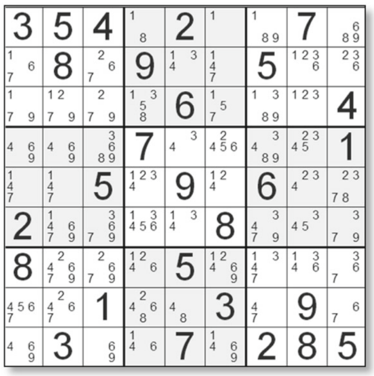


Fig. 3 Easy instance of Sudoku problem with possible numbers

3	**2**	**5**	4	1	**7**	6	3	**4**	2
9	8	**1**	9	8	**5**	7	**2**	1	3
7	6	4	3	**2**	6	**5**	9	8	1
5	2	9	6	3	**4**	**8**	7	1	0
1	3	6	7	9	8	2	4	5	0
4	8	**7**	**5**	1	2	6	3	**9**	0
6	1	**3**	8	**7**	4	2	8	**6**	2
7	**4**	5	**1**	6	3	**7**	5	3	3
8	9	2	**2**	5	9	**9**	**1**	4	3
1	2	1	0	1	1	3	1	2	**26**

Fig. 4 A candidate solution of the Sudoku puzzle with its fitness value

4.4 Proposed model I: Unfiltered-VNS

In this model, we propose the Unfiltered-VNS algorithm for solving Sudoku puzzles based on GVNS. The pseudocode of Unfiltered-VNS is shown in Fig. 5. Unfiltered-VNS starts by generating initial solution *S*. For this model, we define four neighborhood structures (NSs). While three of them (Insert, Swap and CPOEx) are used in the VND Local Search phase, one NS (Invert) is used in the Shaking phase. NSs are applied to one of the sub-grids on the Sudoku instance for each time. The Unfiltered-VNS loop begins with selecting a sub-grid randomly among the nine sub-grids. After Shaking the solution S by Invert NS, the VND procedure processes the solution S' as a returned solution from the Shaking phase. The new solution S'' is evaluated by a local search method which is called as DeepLocalSearch in the algorithm and compared to the solution *S*. If the solution has been improved, then the NSs are restarted from the first NS by setting $k = 1$, or if the solution (S) is not being improved, the VND will move to the next NS. The VND procedure continues until all NSs are exhausted (until $k_{\max}$). The Unfiltered-VNS performs until the stop condition of the main loop is met.

DeepLocalSearch method (Fig. 6) uses the best improvement strategy which exploits the whole current NS search

```
Function Unfiltered_VNS ()
   Generate initial solution S
   k=1   kmax=3
   while (stopping_condition)
      Set sub_grid randomly
      // Shaking phase
      S'=Invert(S,sub_grid)
      // VND local search
      while (k<=kmax)
         if (k==1) then S''= DeepLocalSearch(S', Insert, sub_grid)
         if (k==2) then S''= DeepLocalSearch(S', Swap, sub_grid)
         if (k==3) then S''= DeepLocalSearch(S', CPOEx, sub_grid)
         if (fitness(S'') < fitness(S))
            S= S''
            k=1
         else
            k++
      end while
   end while
   return S
End Function
```

Fig. 5 Pseudocodes of Unfiltered-VNS

```
procedure DeepLocalSearch (s, NS, sub_grid)
   //NS: neighborhood structure
   //s: initial solution
   // i and j are index of unfixed cells in the sub_grid
   sbest=s
   while fitness(s) is not worse than fitness(sbest)
      if NS equal Swap or NS equal Insert then
         for (i=0; i<max-1; i++)
            for (j=i+1; j<max; j++)
               s'= NS(s,i,j,sub_grid)
               if fitness(s') better then fitness(sbest)
                  sbest=s'
               end if
      else
         for (i=1; i<max-1; i++)
            s'= NS(s,i,sub_grid)
            if fitness(s') better then fitness(sbest)
               sbest=s'
            end if
      end if
      if fitness(sbest) better then fitness(s)
         s=sbest
      end if
   end while
   return sbest
end procedure
```

Fig. 6 Pseudocodes of DeepLocalSearch

area to find the best neighborhood solution. During implementation of the NSs, we consider the third rule so that all the NSs can perform only on one sub-grid of the puzzle each time. In other words, no operation of NSs can exceed other sub-grids. There are two general points that should be considered for all NSs: 1. Fixed cells cannot be removed or changed. 2. The sub-grid is considered to be a one-dimensional array.

The details of NSs are explained below.

(a) *Swap* This NS makes a small change in the solution. In Fig. 7, two unfixed cells in the same sub-grid are selected. Then, they are simply swapped as shown in the same figure.

The third and fifth cells are the cells to be swapped. That is, we exchange the values of these two cells so that the final permutation will be {1,6,5,4,3,2,7,8,9}.

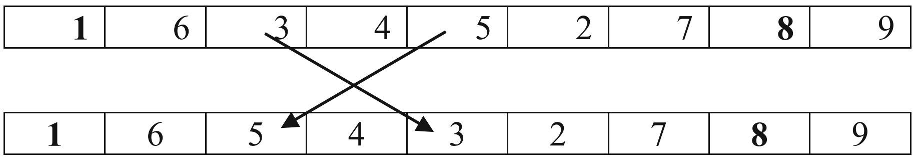
(b) *Insert* This structure performs an insertion of a cell chosen from the sub-grid, in front of another chosen cell. An example of insert NS is shown in Fig. 8. First, we assign the ninth cell as the starting point, and then we choose the fifth cell as an insertion point. This is done by inserting the number in the ninth cell in front of the fifth cell. This operation will shift other cells to the right. If the insertion point of a cell is fixed, the insertion does not work, but if the other cells are fixed as shown in Fig. 8, the insertion can be made without affecting the places of the fixed cells.

(c) *A Centered Point Oriented Exchange (CPOEx)* This structure is used to explore new solutions that are further away from the current solution. An example of CPOEx NS is shown in Fig. 9. First, a cell between the second and sixth cell is selected in a sub-grid. NS uses this cell as a center point to find exchange pairs. Starting from the cells nearest to the center point, exchange pairs are determined and then exchange NS is applied. The CPOEx continues to find appropriate exchange pairs until at least one cell of the pair consists of a fixed cell.

(d) *Invert* The idea is to select two cells in the sub-grid and then invert a sub-sequence of cells between the two selected cells. In Fig. 10, we select the third cell and the sixth cell, which means we have four cell positions {3, 4, 5, 2}. The inversion of these values will be {2, 5, 4, 3}.

The cells are selected randomly for Invert NS. On the other hand, Swap and Insert NSs are performed with all possible cell options into loop/nested-loop for following to the best improvement strategy in DeepLocalSearch as shown in

Fig. 7 An example of swap NS in a sub-grid



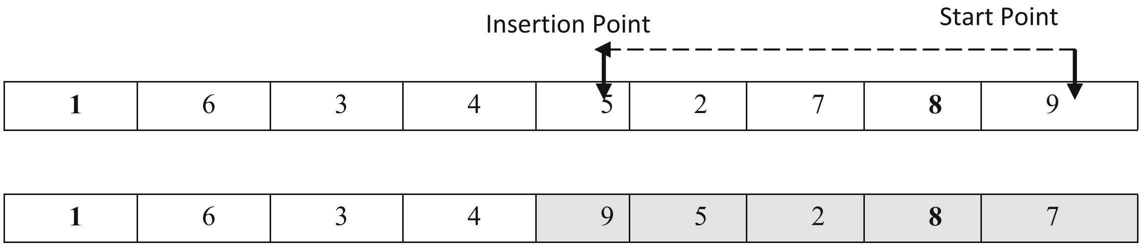


Fig. 8 An example of insert NS in a sub-grid

Fig. 9 An example of CPOEx NS

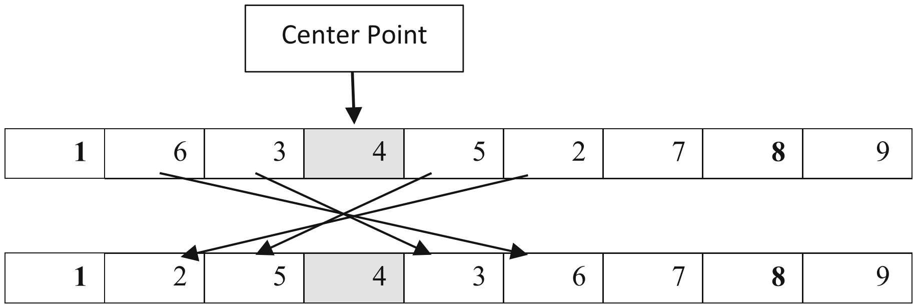


Fig. 10 An example of invert NS

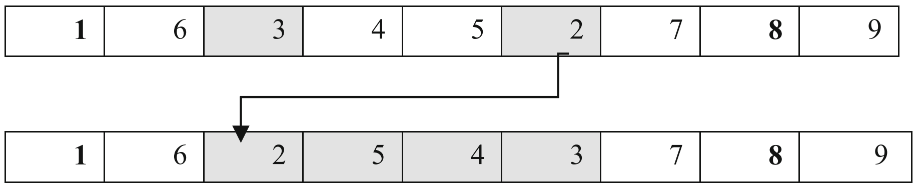


Fig. 6. Then, DeepLocalSearch restarts the whole procedure with the improved solution until no more improvements are found.

Note that the changes in NSs orders in the VND local search can produce different results depending on the problem. To determine the best order of NSs, all possible orders of NSs are tested. The best results were obtained by the following order: Insert, Swap, and CPOEx. Unfiltered-VNS follows this order as shown in Fig. 5.

4.5 Proposed model II: Filtered-VNS

The main process in this model is to find valid permutations for each sub-grid based on possible numbers. To explain how to find sub-grid permutations, an instance of a Sudoku puzzle with possible numbers for each cell is shown in Fig. 11.

We take sub-grid (5) of this example as a sample sub-grid. It has three unfixed cells {1st, 5th, 9th} as shown in Fig. 11. All valid and invalid combinations of sub-grid (5) are listed in Table 2. There are 12 combinations in total, and only two of them are valid combinations; in other words, only valid permutations lead to the optimal solution.

The total number of all valid combinations for all sub-grids, which is 70,255,180,800, can be calculated by multiplying all valid permutations given in Table 3.

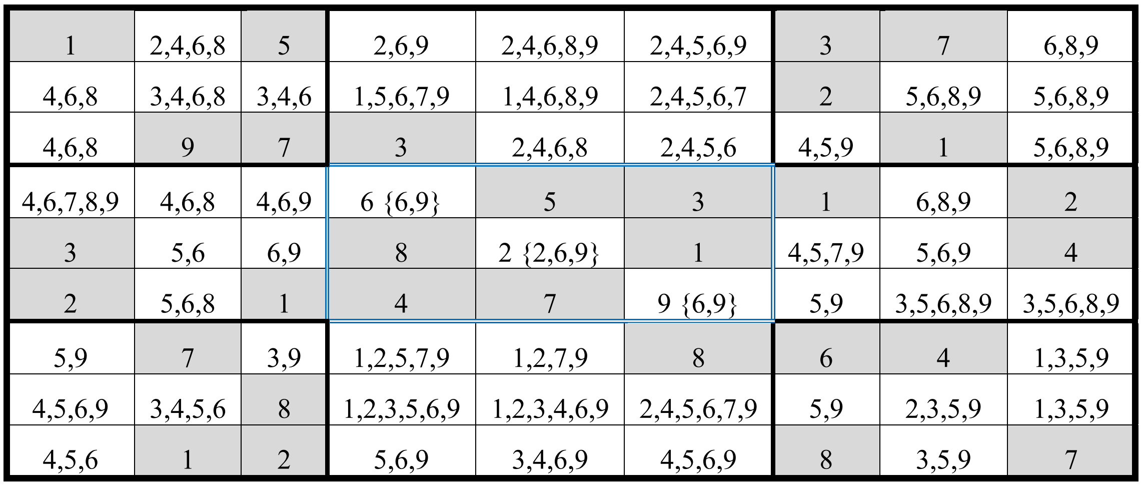
The idea here is to find these permutations and save them to be used later on. The search space is determined by these permutations. Therefore, we have to customize our NS to get these valid permutations.

4.5.1 K-mutation neighborhood structure

In Filtered-VNS, we have one NS that takes K as a parameter. The parameter K is the number of positions at which the corresponding numbers are different between two permutations in the same sub-grid. An example of finding K is explained on the sample sub-grid given in Table 4.

There are eight valid permutations that can be generated from the sample sub-grid: {[2 3 6 7 1 9 4], [2 3 6 9 1 7 4], [3 2 6 7 1 9 4], [3 2 6 9 1 7 4], [6 2 3 7 1 9 4], [6 2 3 9 1 7 4], [6 3 2 7 1 9 4], [6 3 2 9 1 7 4]}.

We can represent this permutation list as a forest of trees, as shown in Fig. 12. A path from root to leaf represents a valid permutation. Dotted nodes on the tree denote the current permutation for the sample sub-grid given in Table 4. Values of K between the current and other permutations are calculated. For example, the K value is 2 between the current and the left-most permutation shown in Fig. 12. In other words, the left-most permutation is two swaps away from the



1	2,4,6,8	5	2,6,9	2,4,6,8,9	2,4,5,6,9	3	7	6,8,9
4,6,8	3,4,6,8	3,4,6	1,5,6,7,9	1,4,6,8,9	2,4,5,6,7	2	5,6,8,9	5,6,8,9
4,6,8	9	7	3	2,4,6,8	2,4,5,6	4,5,9	1	5,6,8,9
4,6,7,8,9	4,6,8	4,6,9	6 {6,9}	5	3	1	6,8,9	2
3	5,6	6,9	8	2 {2,6,9}	1	4,5,7,9	5,6,9	4
2	5,6,8	1	4	7	9 {6,9}	5,9	3,5,6,8,9	3,5,6,8,9
5,9	7	3,9	1,2,5,7,9	1,2,7,9	8	6	4	1,3,5,9
4,5,6,9	3,4,5,6	8	1,2,3,5,6,9	1,2,3,4,6,9	2,4,5,6,7,9	5,9	2,3,5,9	1,3,5,9
4,5,6	1	2	5,6,9	3,4,6,9	4,5,6,9	8	3,5,9	7

Fig. 11 Hard Sudoku Puzzle (S05a) with possible numbers

Table 2 All valid and invalid combinations of sub-grid(5) of puzzle S05A in Fig. 11

# Permutation	Cell 1	Cell 5	Cell 9
1	6	2	6
2	**6**	**2**	**9**
3	6	6	6
4	6	6	9
5	6	9	6
6	6	9	9
7	**9**	**2**	**6**
8	9	2	9
9	9	6	6
10	9	6	9
11	9	9	6
12	9	9	9

current permutation. The K values for all permutations are listed below:

$K_{p1} = 2$, $K_{p2} = 0$ (current permutation), $K_{p3} = 4$, $K_{p4} = 2$, $K_{p5} = 5$, $K_{p6} = 3$, $K_{p7} = 4$, $K_{p8} = 2$

This process is repeated for each sub-grid. The idea in K-mutation is to modify the current permutation with the nearest permutation. This idea is important because it makes obtaining the optimal solution faster. The pseudocode for K-mutation NS is given in Fig. 13. The function using the permutation list (p), current solution (S) generates the new solution (S') by selecting the next K-nearest permutation in the p. If S' is better than the current solution S, then the continuous improvement strategy which allows the exploitation process to carry on with the improved neighbor solution is applied. The function returns the improvement solution as soon as possible.

Table 4 A sample sub-grid adjoined with a possible number list for each cell

2{2,3,6}	3{2,3}	6{2,3,6}
9(3,7,9)	1(1,3)	**8**
7(7,9)	4(1,4)	**5**

4.5.2 Filtered-VNS algorithm

The pseudocode of Filtered-VNS is shown in Fig. 14. Firstly, the solution S is initialized as described above. To apply NSs on the Sudoku instance for each time, one of the nine sub-grids should be selected. Unlike random sub-grid selection in Unfiltered-VNS, a list of sub-grids is generated by using a ranking strategy which calculates the weight of each sub-grid based on four criteria and then sorts the sub-grids in descending order. In this way, neighborhood structure is applied to the topN sub-grids only. To calculate the total weight of each sub-grid, the following four criteria are computed for each sub-grid:

- How many cells are repeated in the row and column?
- How many cells conflict with the fixed cells?

Table 3 Tree representation of eight valid permutations of the sub-grid in Fig. 4

No. of valid permutation	Sub 1	Sub 2	Sub 3	Sub 4	Sub 5	Sub 6	Sub 7	Sub 8	Sub 9
	10	127	14	5	2	14	12	294	8

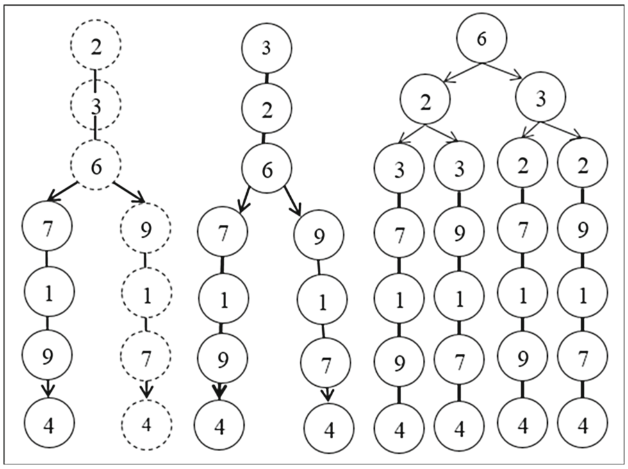


Fig. 12 Tree representation of eight valid permutations of the sub-grid in Table 4

- How many cells are not in the possible value list?
- How many fixed cells are there?

The main loop of Filtered-VNS starts with the calculation of all valid permutations for each sub-grid (permutation_list). Then, Shaking phase selects a random valid permutation (S') from the permutation_list for the current sub-grid. Depending on the value of S', parameter K is calculated for each other valid permutation in the permutation_list. In the inner loop for each sub-grid setting, with the value of K from 1 to Kmax, which denotes the highest degree of the number of differences between current and neighbor permutations, VND is performed to find a better solution (S''). If S'' is better than S, VND performs K-mutation NS with $K = 1$. If not, K is increased by 1 to search the neighborhood solutions with two differences from the current solution. This procedure is repeated until we reach $K_{\max}$without improving. All these operations are iterated until a stopping condition is met.

5 Experimental results

Two proposed GVNS algorithms are implemented using the C# under the Visual Studio 2013 environment. They are tested on several benchmark problems. All experiments are performed on Intel ® Core^{TM} i3 CPU M 380 @ 2.53 GHz, RAM 4.00 GB (3.67 GB usable).

5.1 Datasets

There are many datasets used as benchmarks to determine algorithm performance and the ability to solve the different Sudoku levels. Some papers use their own dataset, but many experimental results use the 15 benchmarks with a, b, and c (http://lipas.uwasa.fi/~timan/sudoku/). This benchmark appeared for the first time in (Mantere and Koljonen 2007); it became very popular and consequently has been used in many papers. Table 5 displays the level and the sample's name in this benchmark. The sample named Al Escargot could be considered the most difficult one among them. Each of these samples has a unique solution. Other Sudoku instances are found in "Appendix".

5.2 Parameter settings

Unlike the many other metaheuristics, VNS has few parameters. Therefore, it provides simpler design ways than other methods (Hansen and Mladenović 2010). For our models, there are just two parameters: topN ranking value and amount of iterations for stopping condition. For two proposed models, we tested four different values for sub-grid rankings: no-ranking (0), top 3, top 6, and top 9. The best results were obtained by top 6 for Filtered-VNS and no-ranking for Unfiltered-VNS (Hamza 2015). For all experiments, stopping conditions of two proposed models are set 2000 iterations.

Fig. 13 K-mutation pseudocode

```
Function K-mutation (Solution S, PermutationList p, int K)
  Set subK_perm_list= a list of valid permutation with the value of K from p
  While (subK_perm_list is not null)      //check the list is null or not
     S'= subK_perm_list (pnext:next k-nearest permutation )
     If (fitness(S') < fitness(S))
        S= S'
        Calculate K values between pnext and other valid permutations in p
        Update subK_perm_list
     End If
  End While
  Return (S)
End Function
```

Fig. 14 Pseudocode of Filtered-VNS

```
Function Filtered_VNS ()
  Generate initial solution S
  Set topN_sub_grids_list (S)
  While (stopping_condition)
      For Each (sub_grid in topN_sub_grids_list)
        Set permutation_list= a list of valid permutations for the sub_grid
        //Shake
        S'= Select a valid permutation (prand) from permutation_list (sub_grid) randomly
        Calculate K values between prand and other valid permutations in permutation_list
        K=1
        // VND
        While (K<=$K_{max}$)
          S''= K-mutation (S', permutation_list, K)
          If (f(S'') <f(S)
            K=1
            S=S''
          Else
            K=K+1
        End While
      End For
      Update topN_sub_grids_list (S)
  End While
End Function
```

Table 5 The 15 benchmarks with A, B, C, and AL escargot

Difficulty rating	Sudoku sample		
	a	b	c
1	s01a	s01b	s01c
2	s02a	s02b	s02c
3	s03a	s03b	s03c
4	s04a	s04b	s04c
5	s05a	s05b	s05c
E	s06a	s06b	s06c
C	s07a	s07b	s07c
D	s08a	s08b	s08c
SD	s09a	s09b	s09c
Easy	s10a	s10b	s10c
Medium	s11a	s11b	s11c
Hard	s12a	s12b	s12c
GA-E	s13a	s13b	s13c
GA-M	s14a	s14b	s14c
GA-H	s15a	s15b	s15c
Al Escargot	s16		

5.3 Comparison with literature results

We compared the results of the Unfiltered-VNS and Filtered-VNS with the results of previous studies listed in Table 1. Our algorithms repeat a predefined value times for each problem instance. The results are compared based on the success rate (SR%) and average CPU time (seconds). SR indicates how many times the algorithm reaches the optimum solution over the repetitions. To demonstrate the performance of our models better, a paired *t*-test (Armitage et al. 2001) is employed to evaluate the mean difference between Unfiltered-VNS and Filtered-VNS and each our model with each previous algorithm.

Because previous studies tested their algorithms on different Sudoku instances, we present our comparisons under separate subtitles.

5.3.1 Comparison with the results of harmony search

For Harmony Search (HS), we consider two papers for this study. The authors share their implementations and samples. The first study (Geem 2007) tries to solve an easy sample ("Appendix Fig. 16") and a hard one ("Appendix Fig. 17") with HS. The HS algorithm is able to solve the easy sample in the range of 3–8 s in the best combination of parameters. Unfortunately, it could not solve the hard sample. Table 6 shows the results by comparison with our two models.

In Table 6, all models solve the easy sample with a 100% success rate while Geem's HS fails to solve the hard sample. Our Unfiltered-VNS shows acceptable results, as it solves the hard sample with a rate of success equal to 23%. The

Table 6 Performance of Unfiltered-VNS and Filtered-VNS against hs by Geem (2007) (*SR* success rate, *AVG CPU* average time in seconds)

Samples	*HS*	Unfiltered-VNS		Filtered-VNS	
	SR%	SR%	Avg CPU	SR%	Avg CPU
Easy	100	100	0.189	100	0.011
Hard	0	23	45.5	100	0.033
Average	50	61.5	22.844	100	0.022

best results are achieved with Filtered-VNS, which solves the hard sample with a 100% success rate in 33 ms.

The second paper (Mandal and Sadhu 2011) tries to improve the previous HS results by implementing new modifications on HS. The algorithm is tested on five Sudoku samples. The results of HS and those of our proposed models are listed in Table 7.

The results indicate that HS can find the solution for all repetition for just sample 1. For the other samples, SR is under 50% and even equal 0% for the last sample. On the other hand, Unfiltered-VNS is superior to HS with average SR is equal to 90%. Furthermore, Filtered-VNS finds solutions for all repetition of all samples in very short time compared to CPU time of Unfiltered-VNS.

When a paired t test is applied to the result of Table 7, the test with a confidence level of 95% shows that both Unfiltered-VNS and Filtered-VNS are significantly better than HS, with $t(7) = 2.4$, $p = 0.042$ and $t(7) = 3.3$, $p = 0.013$, respectively.

5.3.2 Comparison with the results of Hybrid AC3-tabu and Prefiltered Cuckoo Search

The hybrid AC3-tabu (Soto et al. 2013) and the Prefiltered Cuckoo Search (Soto et al. 2014) tested on nine Sudoku instances which are indexed as easy, medium, and hard in Table 5.

In the first paper, the goal is to find the number of attempts needed to solve each of nine samples 30 times, while in the Prefiltered Cuckoo Search, the goal is to find the number of attempts needed to reach optimal solutions for the same dataset 50 times. We made a mathematical proportion and converted the number of attempts to the success rate as follows: The authors in (Soto et al. 2013) declared that the hybrid AC3-tabu algorithm runs 84 times for medium b to reach the optimal solution 50 times. Same result can be written as roundToInteger ((50/84) $\times$ 100) = 60% in terms of success rate. The complete comparison can be found in Table 8.

As shown in Table 8, we see no difference between the results of the three easy samples (easy a, b, and c). All algorithms solve the samples with a 100% success rate. For the medium level, Filtered-VNS has the best results; it solves all three medium samples with a 100% success rate,

Table 7 Performance of Unfiltered-VNS and Filtered-VNS against hs by Mandal and Sadhu (2013) (*SR* success rate, *AVG CPU* average time in seconds)

Samples	HS	Unfiltered-VNS		Filtered-VNS	
	SR%	SR%	Avg CPU	SR%	Avg CPU
1	100	60	121	100	0.038
2	35	100	0.155	100	0.009
3	15	100	0.158	100	0.007
4	10	100	260	100	5.1
5	0	90	231	100	18.9
Average	40	90	122.46	100	4.81

while Hybrid AC3 achieves 100, 97, and 91% success rates. The Prefiltered Cuckoo Search and Unfiltered-VNS achieve success rates of 100, 60, and 75% and 45, 61, and 73%, respectively. The hard samples also show that Filtered-VNS has the best SR results, followed by Unfiltered-VNS, Hybrid A3, and Prefiltered Cuckoo Search.

Note that these algorithms use constraint programming to filter and reduce the search space to obtain these satisfactory results and reduce time as well. When a paired t test is applied to the result of Table 8, the test with a confidence level of 95% shows that there is no significant difference between Unfiltered-VNS, Filtered-VNS, and other two algorithms.

Although hardware configurations are different and time performance comparisons cannot be done fairly under this condition, some facts are clear. For instance, Prefiltered Cuckoo Search takes a large amount of time compared to Hybrid AC3. On average, Filtered-VNS is approximately three times faster than Unfiltered-VNS. On the other hand, Hybrid AC3 and Filtered-VNS's time performances are comparable.

5.3.3 Comparison with results of Improved Artificial Bee Colony Algorithm (ABC)

Three samples given in "Appendix Fig. 18" are tested 30 times by the ABC algorithm which is proposed in (Pacurib et al. 2009). Our proposed models run 30 times for each of these samples. The experimental results are listed in Table 9.

Table 9 shows that the success rates of three algorithms are 100% at the easy level while only ABC and Filtered-VNS can reach the 100% success rate for medium level. Filtered-VNS has the worst result (34%), while ABC and Unfiltered-VNS reach to success rates of 100 and 66%, respectively, for the hard sample. In addition to this, we can see from Table 9 that Filtered-VNS is faster than Unfiltered-VNS in easy and medium levels, but for hard level, Unfiltered-VNS needed less time than Filtered-VNS. However, when a paired t test is applied to the result of Table 9, the test with a confidence level

Table 8 Performance of Unfiltered-VNS and Filtered-VNS against prefiltered Cuckoo search and hybrid AC3 (*SR* success rate, *AVG CPU* average time in seconds)

Sample		Prefiltered Cuckoo search		Hybrid AC3		Unfiltered-VNS		Filtered-VNS	
		SR%	Avg CPU	SR%	Avg CPU	SR%	Avg CPU	SR%	Avg CPU
Easy	a	100	1.31	100	1.331	100	0.146	100	0.002
	b	100	1.007	100	1.032	100	0.138	100	0.003
	c	100	1.021	100	1.067	100	0.14	100	0.003
Medium	a	100	384.341	100	5.17	45	101	100	0.395
	b	60	729.673	97	69.807	61	60.7	100	0.015
	c	75	800.471	91	78.042	73	135	100	0.006
Hard	a	52	2059.69	88	88.55	94	130	58	223
	b	81	1484.692	67	112.667	75	178	100	6.3
	c	70	771.211		–	70	250	100	61.3
Average		82	692.601	93	44.708	80	95.013	95	32.336
Hardware Config		2.0GHz Intel Core2 Duo T5870 with 1 Gb RAM running Fedora 17.				Intel Core i3 CPU M 380 @ 2.53 GHz, RAM 4.00 GB			

Table 9 Performance of Unfiltered-VNS and Filtered-VNS against ABC (*SR* success rate, *AVG CPU* average time in seconds)

Samples	ABC		Unfiltered-VNS		Filtered-VNS	
	SR%	CPU Avg	SR%	CPU Avg	SR%	CPU Avg
Easy	**100**	4.100	**100**	0.279	**100**	0.003
Medium	**100**	17.900	76	56.200	**100**	0.005
Hard	**100**	267.500	66	282.000	34	369.000
Average	100	96.500	80	94.310	78	123.260

of 95% shows that there is no significant difference between Unfiltered-VNS, Filtered-VNS, and ABC algorithm.

Deng et al. (2013) implement HGA, which is the successor of Improved Genetic Algorithm (IGA) (Deng et al. 2011) and New Genetic Algorithm (NGA) (Deng et al. 2012). HGA has better results than most of the recent and previous algorithms. Deng and Li made a complete comparison with GA, CA, ACO, and GA/ACO hybrid by listing the number of samples solved out of 100 tries. It is important to mention that HGA uses 2000 iterations as a stopping condition, while other algorithms use different maximum number iterations. Consequently, we decided to test our two models with two different stopping condition values: 2000 iterations and 5000 iterations. Two models are performed for each instance 100 times. The average success rate results in Table 10 indicate that Filtered-VNS produces the best results. When a paired t test is applied to the result of Table 10, the test with a confidence level of 95% shows that Filtered-VNS algorithm is significantly better than Unfiltered-VNS, $t(72.11) = 4.86$, $p = 0.0001$, HGA, $t(62.76) = 4.03$, $p = 0.0001$ and other four algorithms (GA,CA,ACO and GA/ACO). Although there is no significant difference between Unfiltered-VNS and HGA algorithm, Unfiltered-VNS is significantly better than other four algorithms too. In summary, Filtered-VNS performed the best, followed by Unfiltered-VNS and HGA. Finally, it is important to mention that Unfiltered-VNS has the best result in the hardest sample AI_E by solving it 33 and 53 times out of 100 tries with 2000 and 5000 iterations, respectively.

5.3.4 Comparison with results of Hybrid Genetic Algorithm (HGA)

Figure 15 shows the cumulative results in solving each sample (46 problem instances) out of 100 tries, and the high performance of our two models is obvious. It should be mentioned that if an algorithm solves each instance with 100% SR, the cumulative SR on the left side in Fig. 15 would be equal to 1600, which is the top value. The graph indicates that the cumulative SR of Filtered-VNS lies between 1400 and 1600, which is the best result among the other algorithms. Unfortunately, no data are available about the amount of time consumed. Table 11 shows the time average of our two models (Unfiltered-VNS and Filtered-VNS).

5.3.5 Comparison with results of Hybrid Algorithms

Our two models are compared with the three hybrid algorithms of Perez and Marwala (2008). The algorithms are given below:

Table 10 Success rate performance of Unfiltered-VNS and Filtered-VNS against HGA and previous studies

Samples	GA			CA			ACO			GA/ACO hybrid			HGA			Unfiltered-VNS (1) 2000 iterations			Unfiltered-VNS (2) 5000 iterations			Filtered-VNS (1) 2000 iterations			Filtered-VNS (2) 5000 iterations		
	a	b	c	a	b	c	a	b	c	a	b		a	b	c	a	b	c	a	b	c	a	b	c	a	b	c
1	58	68	48	56	76	46	70	54	41	48	53	42	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
2	41	22	31	42	22	22	10	23	30	21	26	27	100	79	90	69	100	100	89	100	100	100	100	100	100	100	100
3	34	15	16	42	19	17	22	6	5	18	11	5	87	88	100	100	73	50	100	100	50	100	100	100	100	100	100
4	9	5	11	7	7	13	4	7	3	7	5	7	69	89	94	86	33	42	93	46	42	100	100	100	100	100	100
5	5	12	10	4	8	9	4	1	0	6	10	0	58	85	71	53	53	44	73	73	55	100	95	100	100	100	100
E	82	83	77	72	86	69	74	54	79	44	67	59	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
C	33	10	17	27	8	22	12	2	21	31	3	23	96	85	98	81	100	100	87	100	100	100	100	100	100	100	100
D	6	11	22	11	13	25	0	5	14	6	18	15	2	0	47	53	26	40	60	47	53	100	100	100	100	100	100
SD	5	8	5	4	8	3	0	2	1	2	3	5	0	0	0	46	40	40	66	73	80	100	100	100	100	100	100
Easy	34	62	50	39	60	55	58	62	48	37	53	39	100	99	100	100	100	100	100	100	100	100	100	100	100	100	100
Med.	10	2	29	13	2	26	11	1	3	15	4	25	96	100	94	55	7	35	66	7	66	100	100	100	100	100	100
Hard	2	10	4	3	6	1	0	0	2	0	3	4	63	3	2	60	46	60	80	69	80	30	100	80	60	100	100
GA-E	37	52	35	28	52	36	43	39	16	23	29	16	97	99	100	100	100	100	100	100	100	100	100	100	100	100	100
GA-M	14	11	15	11	9	8	7	9	3	11	9	9	90	92	94	53	47	60	92	47	60	100	100	100	100	100	100
GA-H	3	5	4	0	4	1	0	0	0	2	7	2	7	9	6	26	53	13	53	66	26	100	100	68	100	100	92
AI_E	5			3			0			1			8			**33**			**53**			5			14		
Average	25	25	25	24	25	24	21	18	18	17	20	19	67	69	69	70	63	64	82	74	73	90	100	91	92	100	94

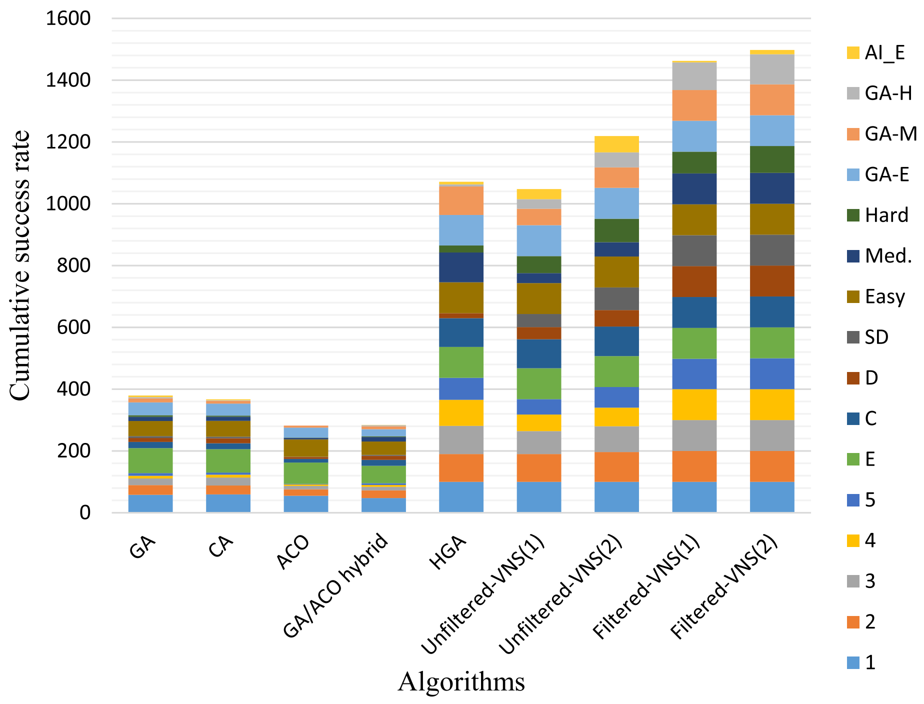


Fig. 15 Cumulative success rate of the 46 problem instances in Table 10

Table 11 CPU average time performance of Unfiltered-VNS and Filtered-VNS in solving the 46 sudoku problem instances (5000 iterations)

Samples	Unfiltered-VNS average CPU time			Filtered-VNS average CPU time		
	a	b	c	a	b	c
1	0.186	0.155	0.152	0.007	0.007	0.007
2	127	0.166	0.171	0.012	0.008	0.008
3	3.289	185	38	0.009	0.053	1.945
4	127	184	59	0.045	0.043	4.983
5	176	202	109	0.097	47.3	7.751
E	0.16	0.187	0.16	0.008	0.007	0.008
C	0.178	0.189	0.175	0.013	0.007	0.007
D	115	277	158	5.729	0.735	0.301
SD	184	219	269	0.097	23	1.5
Easy	0.17	0.174	0.154	0.004	0.004	0.004
Med.	120	5.6	123	0.371	0.019	0.008
Hard	170	182	194	191	6.014	53.2
GA-E	0.17	0.174	180	0.003	0.004	0.004
GA-M	147	54	51	0.005	0.913	2.371
GA-H	213	153	171	8.87	9.4	72
AI_E	400			428		
Average*	92.2102	116.415	90.187	13.751	32.219	9.606

*The CPU time of AI_E is not considered on average calculation

Table 12 Performance of Unfiltered-VNS and Filtered-VNS against CGA and others (*SR* success rate, *AVG CPU* average time in

Sample	CGA	QSA	HGASA	Unfiltered-VNS		Filtered-VNS	
	SR%	SR%	SR%	SR%	Avg CPU	SR%	Avg CPU
	100	75	100	100	0.180	100	0.004

- Cultural Genetic Algorithm (CGA)
- Quantum Simulated Annealing (QSA)
- Genetic Algorithm with Simulated Annealing (HGASA)

All the experiments are performed on one Sudoku puzzle given in "Appendix Fig. 19". In Table 12, we can see that our two models (Unfiltered-VNS and Filtered-VNS) can find the optimal solution in a very short time. In addition, all algorithms can find the optimal solution with a 100% success rate, except for QSA which has 75% success rate.

6 Discussion and Conclusion

There are many metaheuristic studies in the literature for solving Sudoku puzzles. We made a comprehensive survey of many previous studies about Sudoku for this paper, which can be used as a reference resource to find many algorithms and datasets related to Sudoku puzzles.

To the best of our knowledge, there is only one trajectory-based metaheuristic—simulated annealing—that is studied for solving Sudoku problems (Lewis 2007). In this study, we propose two novel models, called Unfiltered and Filtered Variable Neighborhood Search (VNS), to solve Sudoku puzzles. To find the best configuration of the two models, different strategies are developed. The local search is the powerful part of the VNS. Due to the constraints (fixed cells, etc.) on the search area, it is difficult to apply the predefined neighborhood structures of other VNS studies directly to Sudoku. We propose four neighborhood structures (swap, insert, CPOEx, and invert) for Unfiltered-VNS and one novel neighborhood structure (K-mutation) for Filtered-VNS.

These neighborhood structures are performed into the local search (VND) by using a new improvement strategy-continuous improvement (continues the searches with better solutions), which is different from the first improvement and best improvement strategies in the literature. While running a VND local search on sub-grids, the ranking of the sub-grids depending on the contribution quality of the solution is another novel idea that decreases the time needed to find the best solution in the Filtered-VNS.

Another contribution is the filtering technique on the Filtered-VNS model. The novel idea is to segment the Sudoku puzzle into the nine sub-grids. Then, it optimizes each sub-grid separately using permutation filtering. The new K-mutation neighborhood structure based on this filtering technique is performed in the local search phase by using the continuous improvement strategy.

We experiment two models on 57 Sudoku instances. The results obtained from our two models (Unfiltered-VNS and Filtered-VNS) outperform the most of the previous results. Unfiltered-VNS and Filtered-VNS improve the rate of success of the hard sample proposed by Geem (2007) from 0 to 100%. Mandal and Sadhu (2011) tested their algorithm on five different samples. The average success rates of their results are improved 125 and 150% by Unfiltered-VNS and Filtered-VNS, respectively. Hybrid AC3 (Soto et al. 2013) can be considered as an efficient algorithm for solving Sudoku puzzles. The average results of this algorithm are also improved 2.24% by Filtered-VNS. The average results of the Cuckoo Search algorithm are improved 14.1% by Filtered-VNS. Finally, HGA (Deng et al. 2013), which is a good, recent algorithm that has been outperforming many previous results. The results obtained by HGA are also improved by our two models. Unfiltered-VNS outperforms HGA by increasing average success rate to 11.44%, and Filtered-VNS improves the results of HGA 39.91% on 46 Sudoku benchmark instances.

To demonstrate the performance of our two models better, a paired *t* test is employed to evaluate the mean difference between each proposed model and other algorithms. The results show that our proposed models are either significantly better than previous algorithms or no significant differences with the others.

Although the proposed two models show mostly high performance over the other compared algorithms, we can observe that Filtered-VNS shows higher performance than Unfiltered-VNS and it is more robust in solving most of the Sudoku instances, except for some very hard instances. On the other hand, Unfiltered-VNS is superior especially in solving the Al Escargot instance, which is considered one of the hardest Sudoku problem instances.

There can be many reasons behind this performance differences. First of all, Unfiltered-VNS searches the solution in a larger search space compared to search space of Filtered-VNS. Because of that it needs to execute local search method deeply with the best improvement strategy which takes more time. On the other hand, Filtered-VNS reduces the search space by filtering. Furthermore, a new neighborhood structure (k-mutation) that is suitable for searching in the filtered search space is employed in its local search component. These two improvements with ranking strategy effect on the local search performance of Filtered-VNS and decrease the total time of the algorithm. In other words, Filtered-VNS does not need to search deeply as in Unfiltered-VNS. Conversely, Unfiltered-VNS shows superior performance on the hard instances where its optimal solution can be hidden between many local optima. Therefore, for hard instance, deep local search with various neighborhood structures as provided by Unfiltered-VNS can be much more important to reach optimal solution and to escape the local optima.

All these results support that our proposed GVNS-based algorithms produce satisfactory results for all level of Sudoku puzzles. As future work, we would like to implement these proposed algorithms to solve other problems in game development and game design.

Compliance with ethical standards

Conflicts of interest Authors declares that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

Appendix: Sudoku samples

See Figs. 16, 17, 18, and 19.
newpage

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

Fig. 16 Easy Sudoku puzzle

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

Fig. 17 Hard Sudoku puzzle (Geem 2007)

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

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

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

Fig. 18 Three Sudoku samples: easy, medium, and hard (Karaboga et al. 2014)

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

Fig. 19 Sudoku Sample (Perez and Marwala 2008)

References

Armitage P, Berry G, Matthews JNS (2001) Statistical methods in medical research, 4th edn. Blackwell Science, Oxford

Asif M, Baig R (2009) Solving NP-complete problem using ACO algorithm. In: 2009 International conference on emerging technologies, pp 13–16

Boryczka U, Juszczuk P (2012) Solving the sudoku with the differential evolution. Zeszyty Naukowe Politechniki Białostockiej: Informatyka 9:5–16

Bartlett A, Chartier TP, Langville AN, Rankin TD (2008) An integer programming model for the Sudoku problem. J Online Math Appl vol 8,

Crawford B, Castro C, Monfroy E (2013) Solving Sudoku with constraint programming. In: Shi Y, Wang S, Peng Y, Li J, Zeng Y (eds) Cutting-edge research topics on multiple criteria decision making SE-52, vol 35, pp 345–348

Deng X, Li J, Li G (2013) Research on Sudoku puzzles based on metaheuristics algorithm. J Mod Math Front 2(1):25–32

Deng X, Li Y, Cai R (2011) Solving Sudoku puzzles based on improved genetic algorithm. Comput Appl Softw 28(3):68–70

Deng X, Li Y, Cai R (2012) Solving Sudoku with New Genetic Algorithm. Lecture Notes in Information Technology, vol 12, pp 431–440

Geem W (2007) Harmony search algorithm for solving Sudoku. Knowledge-based intelligent information and engineering systems SE-46, vol 4692. Springer, Berlin, pp 371–378

Geem W, Kim H, Loganathan GV (2001) A new heuristic optimization algorithm: harmony search. Simulation 76(2):60–68

Hansen P, Mladenović N (2010) Handbook of metaheuristics. Variable neighborhood search. Kluwer Academic Publishers, Dordrecht

Hansen P, Mladenović N, Moreno Pérez J (2010) Variable neighbourhood search: methods and applications. Ann Oper Res 175(1):367–407

Hamza KA (2015) A study on metaheuristic algorithms for solving Sudoku puzzles. Master thesis, graduate school of sciences and engineering, Fatih University, https://drive.google.com/file/d/0BwGzOrhCZ2Q01DaFRRcGVSVU0/view?usp=sharing

Hamza KA, Sevkli AZ (2014) A variable neighborhood search for solving Sudoku puzzles, BT-ECTA 2014. In: Proceedings of the international conference on evolutionary computation theory and applications, part of IJCCI 2014, Rome, Italy

Hereford JM, Gerlach H (2008) Integer-valued Particle Swarm Optimization applied to Sudoku puzzles. In: Swarm intelligence symposium, SIS 2008. IEEE

Karaboga D, Gorkemli B, Ozturk C, Karaboga N (2014) A comprehensive survey: artificial bee colony (ABC) algorithm and applications. Artif Intell Rev 42(1):21–57

Lewis R (2007) Metaheuristics can solve sudoku puzzles. J Heuristics 13(4):387–401

Mandal S, Sadhu S (2011) An efficient approach to solve Sudoku problem by harmony search algorithm. Int J Eng Sci 4:312–323

Mandal S, Sadhu S (2013) Solution and level identification of Sudoku using harmony search. Int J Mod Educ Comput Sci 5(3):49–55 April 16

Mantere T, Koljonen J (2007) Solving, rating and generating Sudoku puzzles with GA. In: 2007 IEEE congress on evolutionary computation, pp 1382–1389

Mantere T, Koljonen J (2008) Solving and analyzing Sudokus with cultural algorithms. In: Evolutionary computation, 2008. CEC 2008. (IEEE world congress on computational intelligence)

Mantere T, Koljonen J (2014) Sudoku research page. Retrieved August 29, 2014, from http://lipas.uwasa.fi/~iman/sudoku/

Mladenović N, Hansen P (1997) Variable neighborhood search. Comput Oper Res 24(11):1097–1100

Mishra DB, Mishra R, Das KN, Acharya AA (2018) Solving Sudoku puzzles using evolutionary techniques—a systematic survey. In: Pant M, Ray K, Sharma T, Rawat S, Bandyopadhyay A (eds) Soft computing: theories and applications. Advances in intelligent systems and computing, vol 583. Springer, Singapore

Pacurib JA, Seno GMM, Yusiong JPT (2009)Solving Sudoku puzzles using improved artificial bee colony algorithm. In: Fourth international conference on innovative computing, information and control (ICICIC), pp 885–888

Perez M, Marwala T (2008) Stochastic optimization approaches for solving Sudoku arXiv Preprint arXiv:0805.0697

Pillay N (2012) Finding solutions to Sudoku puzzles using human intuitive heuristics. South Afr Comput J 49:25–34

Sevkli AZ, Guler B (2017) A multi-phase oscillated variable neighbourhood search algorithm for a real-world open vehicle routing problem. Appl Soft Comput 58:128–144

Sevkli Z, Sevilgen FE (2006) Variable neighborhood search for orienteering problem. Lect Notes Comput Sci 4263:134–143

Singh G, Deep K (2016) A new membrane algorithm using the rules of Particle Swarm Optimization incorporated within the framework of cell-like P-systems to solve Sudoku. Appl Soft Comput 45:27–39

Soto R, Crawford B, Galleguillos C, Monfroy E, Paredes F (2013) A hybrid AC3-tabu search algorithm for solving Sudoku puzzles. Expert Syst Appl 40(15):5817–5821

Soto R, Crawford B, Galleguillos C, Monfroy E, Paredes F (2014) A prefiltered cuckoo search algorithm with geometric operators for solving Sudoku problems. The Sci World J

Weller M (2008) Counting, generating, and solving Sudoku, pp 1–34, Retrieved from http://theinf1.informatik.uni-jena.de/publications/sudoku-weller08.pdf

Yato T, Seta T (2003) Complexity and completeness of finding another solution and its application to puzzles. In: IEICE Transactions on fundamentals Electronics, communications Computer Sciences (Inst Electron Inf Commun Eng), vol E86-A, pp 1052–1060