Journal of the Operational Research Society (2014) 65, 1602–1610 © 2014 Operational Research Society Ltd. All rights reserved. 0160-5682/14

www.palgrave-journals.com/jors/

A comparison of several enumerative algorithms for Sudoku

Leandro C Coelho[1,2*] and Gilbert Laporte[1,3]

[1]CIRRELT—Interuniversity Research Center on Enterprise Networks, Logistics and Transportation, Québec, Canada; [2]Université Laval, Québec, Canada; and [3]HEC Montréal, Montréal, Canada

Sudoku is a puzzle played of an $n \times n$ grid $\mathcal{N}$, where n is the square of a positive integer m. The most common size is $n = 9$. The grid is partitioned into n subgrids of size $m \times m$. The player must place exactly one number from the set $N = \{1, \ldots, n\}$ in each row and each column of $\mathcal{N}$, as well as in each subgrid. A grid is provided with some numbers already in place, called *givens*. In this paper, some relationships between Sudoku and several operations research problems are presented. We model the problem by means of two mathematical programming formulations. The first one consists of an integer linear programming model, while the second one is a tighter non-linear integer programming formulation. We then describe several enumerative algorithms to solve the puzzle and compare their relative efficiencies. Two basic backtracking algorithms are first described for the general Sudoku. We then solve both formulations by means of constraint programming. Computational experiments are performed to compare the efficiency and effectiveness of the proposed algorithms. Our implementation of a backtracking algorithm can solve most benchmark instances of size 9 within 0.02 s, while no such instance was solved within that time by any other method. Our implementation is also much faster than an existing alternative algorithm.

Journal of the Operational Research Society (2014) **65**(10), 1602–1610. doi:10.1057/jors.2013.114
Published online 25 September 2013

Keywords: Sudoku; puzzle; backtracking; linear and non-linear integer programming; constraint programming

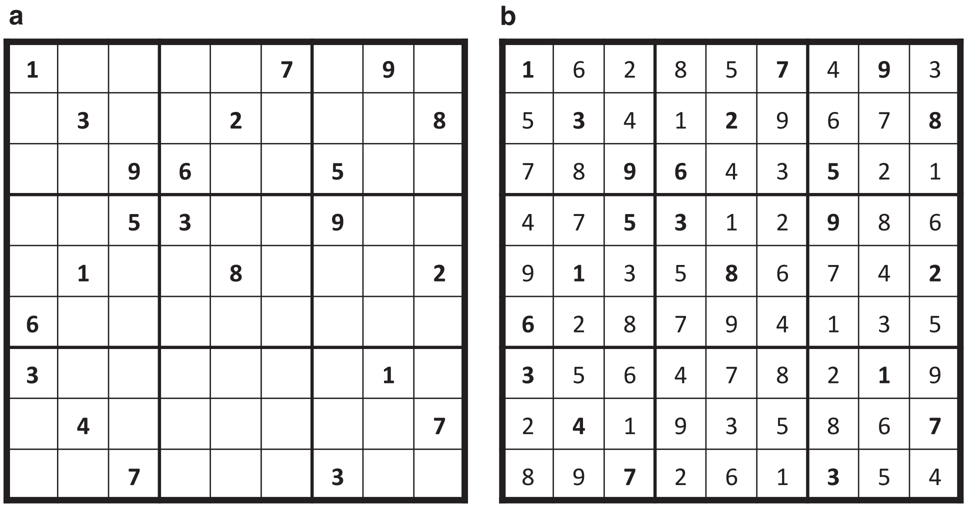
1. Introduction

Sudoku is a puzzle played on an $n \times n$ grid $\mathcal{N}$, where n is the square of a positive integer m. The most common size is $n = 9$. The grid $\mathcal{N}$ is partitioned into n subgrids of size $m \times m$. The player must place exactly one number from the set $N = \{1, \ldots, n\}$ in each row and each column of $\mathcal{N}$, as well as in each subgrid. A grid is provided with some numbers already in place, called *givens*. The origins of Sudoku can be traced back to the 19th century when number games first appeared in newspapers. The modern version of the puzzle was likely created in the 1970s (Garns, 1979) and became a world success in the following decade. Nowadays, many daily newspapers offer Sudoku alongside crosswords and other puzzles. Figure 1 depicts a 9×9 Sudoku. Figure 1(a) shows the initial grid with 22 givens, and Figure 1(b) provides the solution.

A well-designed Sudoku grid should contain enough givens to make the solution unique. It has been proved that the least number of givens needed to yield a unique solution to a 9×9 grid is 17 (McGuire *et al*, 2012). The number of feasible solutions for an empty 9×9 Sudoku grid is about 6×10^{21} (Felgenhauer and Jarvis, 2005), but the number of symmetries obtained by row and column permutations and by number relabellings is also huge. When symmetries are disregarded, the number of solutions is 5 472 730 538 (Russel and Jarvis, 2005)

The Sudoku was proved to be NP-complete (Colbourn *et al*, 1984; Takayuki and Takahiro, 2003), thus inciting operations research scientists to develop efficient algorithms for it. The Sudoku game can be reduced to a satisfiability problem. It has been solved by constraint programming (CP) (Simonis, 2005) by a hybrid of CP and stochastic search (Lewis, 2007a), by neighbourhood search metaheuristics (Lewis, 2007b), and by genetic algorithms (Mantere and Koljonen, 2006, 2007). A Sudoku is a special case of a Latin square, an $n \times n$ grid filled with n distinct numbers, each occurring exactly once in each row and in each column (Dénes and Keedwell, 1974). It can also be viewed as a colouring problem, widely studied in graph theory (see, eg, Jensen and Toft, 1994). The problem can be generalized if one interprets the n numbers as resources. Each resource is available n times and must be assigned to different tasks (i, j). By eliminating one of the dimensions, the problem can be interpreted as that of assigning people to jobs (Sellmann *et al*, 2002; Naveh *et al*, 2007), or of assigning colours to a map, or more generally, to a graph (Jensen and Toft, 1994). In two dimensions, one can assign a job to a machine at a certain time, such as in the job-shop problem, or in scheduling and timetabling problems (Applegate and Cook, 1991; Higgins *et al*, 1996). Studying and solving the Sudoku is therefore attractive not only as a puzzle, but also from an operations research perspective. In particular, developing and comparing Sudoku

**Correspondence: Leandro C Coelho, Operations and Decision Systems, CIRRELT and Université Laval, 2325 rue de la Terrasse, Quebec, QC, G1V0A6, Canada.*



a

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

b

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

Figure 1 An instance of a Sudoku: (a) A 9×9 Sudoku with 22 givens; (b) The completed Sudoku.

algorithms is of interest to our community, namely for large sizes ($m \geqslant 4$) which are rather difficult to solve manually.

We first model the problem by means of mathematical programming techniques. We present a three-index integer linear programming model for the general Sudoku, as well as a more compact two-index non-linear integer programming model. Similar models have been proposed in Tadei and Mancini (2006) and in Bartlett and Langville (2006). We then describe four enumerative algorithms. Our aim is to compare the relative performance of these algorithms, namely with respect to an existing algorithm developed by Lewis (2007b). This enables us to assess the efficiency of each algorithm with respect to the input data, that is, to the number of givens, and with respect to the size of instance, in order to evaluate which kind of algorithm scales best. Since these models can be adapted to solve related problems different from the Sudoku, we expect that our results will motivate other researchers to pursue the same objective in different contexts.

The remainder of the paper is organized as follows. Two mathematical programming models are described in Section 2, and the algorithms are presented in Section 3. This is followed by computational experiments in Section 4, and by conclusions in Section 5.

2. Mathematical models

We now present two exact mathematical formulations we have developed for the $n \times n$ Sudoku. In both formulations it is implicitly assumed that the variables corresponding to the givens are fixed at these values.

2.1. Three-index binary linear programming formulation

Our three-index binary linear programming formulation makes use of variables x_{ij}^k equal to 1 if and only if value $k \in N$ is present in the cell (i,j) of grid $\mathcal{N}$. A model similar to this one was presented by Tadei and Mancini (2006). The following sets of constraints must be satisfied:

$$\sum_{j \in N} x_{ij}^k = 1, \quad i,k \in N \tag{1}$$

$$\sum_{i \in N} x_{ij}^k = 1, \quad j,k \in N \tag{2}$$

$$\sum_{i=(h-1)m+1}^{mh} \sum_{j=(l-1)m+1}^{ml} x_{ij}^k = 1, \quad k \in N, \quad h,l \in \{1, \ldots, m\} \tag{3}$$

$$\sum_{k \in N} x_{ij}^k = 1, \quad i,j \in N \tag{4}$$

$$x_{ij}^k \in \{0,1\}, \quad i,j,k \in N \tag{5}$$

Constraints (1) and (2) mean that each value k must be present exactly once in each row and in each column. Constraints (3) ensure that each value k appears only once in each subgrid $\mathcal{M}$. Constraints (4) state that only one value k can be assigned to each position (i,j). Finally, constraints (5) define the domains of the variables. The model defined by (1)–(4) contains n^3 binary variables and $3n^2 + nm^2$ linear constraints.

2.2. Two-index non-linear integer programming formulation

A more compact model making use of two-index variables can be defined at the expense of using non-linear relations between the variables, as is more common when working under the CP paradigm. This formulation makes use of integer variables x_{ij} equal to the number assigned to cell (i,j) of grid $\mathcal{N}$. A similar model is presented in Bartlett and Langville (2006). The following sets of constraints must be satisfied:

$$x_{ij} \neq x_{kj}, \quad i,j,k \in N, \quad k \neq i \tag{6}$$

$$x_{ij} \neq x_{ik}, \quad i,j,k \in N, \quad k \neq j \tag{7}$$

$$\sum_{i \in N} x_{ij} = \sum_{i \in N} i, \quad j \in N \tag{8}$$

$$\sum_{j \in N} x_{ij} = \sum_{j \in N} j, \quad i \in N \tag{9}$$

$$x_{ij} \neq x_{kj} \quad p,q \in \{1, \dots, m\}, \quad i \in \{m(q-1)+1, \dots, mq\}, \quad j,k \in \{m(p-1)+1, \dots, mp\}, \quad k \neq i \tag{10}$$

$$x_{ij} \neq x_{il} \quad p,q \in \{1, \dots, m\}, \quad i,l \in \{m(q-1)+1, \dots, mq\}, \quad j \in \{m(p-1)+1, \dots, mp\}, \quad l \neq j \tag{11}$$

$$x_{ij} \neq x_{kl} \quad p,q \in \{1, \dots, m\}, \quad i,l \in \{m(q-1)+1, \dots, mq\}, \quad j,k \in \{m(p-1)+1, \dots, mp\}, \quad k \neq i \text{ or } l \neq j \tag{12}$$

$$\sum_{i=m(p-1)+1}^{mp} \sum_{j=m(q-1)+1}^{mq} x_{ij} = \sum_{i \in N} i, \quad p,q \in \{1, \dots, m\} \tag{13}$$

$$x_{ij} \in N, \quad i,j \in N \tag{14}$$

Constraints (6)–(9) indicate that each value k must be present exactly once in each row and each column. Note that constraints (6) and (7) alone are sufficient to ensure that a number will not be repeated in the same row or column. As a result, constraints (8) and (9) are redundant but tighten this part of the formulation. Constraints (10)–(12) ensure that each element is different from all other elements in the same subgrid. As before, constraints (13) are redundant but yield a tighter formulation. Constraints (14) define the domains of the variables. The model defined by (6)–(13) contains n^2 integer variables, $2n+m^2$ linear constraints and $2n^3+3m^5$ non-linear constraints.

3. Enumerative algorithms

The algorithms developed and compared in this study are all enumerative. The first two are backtracking procedures in which numbers are iteratively assigned to cells as long as this is feasible. Each assignment automatically restricts the available options for some of the remaining cells, as per the rules of the puzzle. Whenever an infeasibility is detected, backtracking takes place. The first algorithm assigns the numbers in increasing order and considers the cells according to a predefined order. The second algorithm gives priority to the cells with the fewest remaining feasible assignments.

The next two algorithms consist of solving the two mathematical formulations presented in Section 2 by means of the CP functions contained in CPLEX Studio. The roots of CP can be traced back to the work of Golomb and Baumert (1965) who were the first to formally formulate the early ideas of Walker (1960). Initially, the domains of the variables corresponding to the givens each contain a single value. This algorithm assigns values to variables with finite domains which are gradually reduced as the search proceeds, and it makes use of backtracking. This algorithm works rather effectively in situations for which the aim is to determine a feasible solution to a highly constrained problem, such as Sudoku and manpower scheduling (see, eg, Laporte and Pesant, 2004). Over time, CP has evolved into a rich research area. For overviews, see Apt (2003), Lustig and Puget (2001) and Rossi *et al* (2006). The CP functionality contained in CPLEX Studio was used as a black box and no description of its inner workings is available.

In a sense, the first two algorithms used in the comparison can also be viewed as an application of CP because of the way they operate on variable domains. As a result, all four algorithms are based on the same paradigm but make use of different implementation rules. We now describe our two backtracking algorithms.

3.1. Simple backtracking algorithm

Our first implementation of the backtracking algorithm differs from a brute-force method because it verifies the feasible assignments for each empty cell. When a cell has no feasible assignment, the algorithm backtracks and changes the assignment of an already filled cell. It considers cells sequentially and assigns values to empty cells starting with the lowest possible number with respect to all other assignments, taking into consideration the rules of the puzzle. Algorithm 1 shows the pseudocode for this algorithm. Note that the backtracking procedure takes place at line 16, when the process goes back to a previously made assignment and tries another possibility.

3.2. Backtracking algorithm with heuristic moves

The second implementation of the backtracking algorithm also takes into consideration the rules of the puzzle and all numbers already assigned to other cells. However, instead of visiting the cells sequentially, the next cell to be visited is the one having the least number of remaining feasible assignments. Our motivation is to avoid re-evaluating assignments too often. Algorithm 3.2 shows the pseudocode for this algorithm. Note the extra procedure in lines 8–10 to select the next cell to work on.

4. Computational experiments

We now describe the computational experiments we have carried out to evaluate the algorithms. All computations were performed on a desktop PC equipped with an Intel Core i7™ processor running at 3.66 GHz with 8 GB of RAM installed, with the Oracle Linux 6.3 operating system. The algorithms were coded in C++. We have used the IBM Concert Technology and CPLEX Optimization Studio 12.5 as the CP solver. In addition, we have obtained the code of Lewis (2007b).

This algorithm is also coded in C++ and we have compiled the program and run the tests on it in our computing environment.

Algorithm 1 Pseudocode for the simple backtracking algorithm

```
 1: Input the Sudoku grid with the givens.
 2: Let $c$ be a cell.
 3: Let $v(c)$ be the value assigned to cell $c$.
 4: Let $S(c, i)$ return true if number $i$ can be assigned to cell $c$.
 5: for all empty cells $c$ do
 6:     $v(c)=0$
 7: end for
 8: Select the next empty cell $c$
 9: for $i=v(c)+1$ to $n$ do
10:     $v(c)=0$.
11:     if $S(c, i)=$ true then
12:         Assign $i$ to $c$;
13:         Go to 20
14:     end if
15: end for
16: if $v(c)=0$ then
17:     Backtrack to the previously selected cell.
18:     Go to 9
19: end if
20: if there are empty cells then
21:     Go to 8
22: end if
23: Return the solved Sudoku.
```

Algorithm 2 Pseudocode for the backtracking algorithm with heuristic moves

```
 1: Input the Sudoku grid with the givens.
 2: Let $c$ be a cell.
 3: Let $v(c)$ be the value assigned to cell $c$.
 4: Let $S(c, i)$ return true if number $i$ can be assigned to cell $c$.
 5: for all empty cells $c$ do
 6:     $v(c)=0$
 7: end for
 8: for All empty cells $c$ do
 9:     Select the one with the least number of candidates;
10:     Break ties randomly.
11: end for
12: for $i=v(c)+1$ to $n$ do
13:     $v(c)=0$
14:     if $S(c, i)=$ true then
15:         Assign $i$ to $c$;
16:         Go to 23.
17:     end if
18: end for
19: if $v(c)=0$ then
20:     Backtrack to the previously selected cell.
21:     Go to 12
```

Algorithm 2 Pseudocode for the backtracking algorithm with heuristic moves

```
22: end if
23: if there are empty cells then
24:     Go to 8.
25: end if
26: Return the solved Sudoku.
```

We have first performed some experiments with empty grids in order to assess the performance of each algorithm on instances of size $n=9$, 16, 25 and 36, within a 2-h time limit. The results shown in Table 1 indicate that neither backtracking algorithm performs well on empty grids of size larger than 16. On the other hand, both mathematical formulations perform better, the two-index model being superior in terms of time and scalability. The neighbourhood search algorithm of Lewis (2007b) solves as many instances as the three-index model within the allotted time. The three-index model performs better on the smaller instances ($n \leqslant 16$), whereas the metaheuristic of Lewis (2007b) is faster on instances of size $n=25$. We have also tried to solve an empty 49×49 grid, but this took more than 2 h for all algorithms and the tests were aborted. Note that on small instances, implementing our own standalone backtracking solver is orders of magnitude faster than a state-of-the-art CP solver.

We have then studied a very large data set containing 49 151 instances of size 9 with 17 givens obtained from Royle. This data set contains instances with the minimum number of givens necessary to yield a unique solution. Moreover, these instances are not symmetric, that is, they cannot be converted into one another. We have attempted to solve these instances with the simple backtracking algorithm, but the running times were prohibitive, in excess of 60 s per instance, so this option was abandoned. We have solved only the first 2500 instances of this data set using the algorithm of Lewis (2007b). This choice is due to the high running time of this algorithm, also in excess of 60 s for many instances. With a time limit of 20 s, this algorithm was able to solve only 920 out of the 2500 instances. The average running time for the solved instances was 8.42 s. We have then increased the running time to 60 s and we have attempted to solve the first 500 instances of the data set.

Table 1 Running time in seconds of each algorithm to solve one empty Sudoku grid of varying sizes

Size n	*Metaheuristic of* Lewis (2007b)	*Simple backtracking*	*Backtracking with heuristic moves*	*Three-index model*	*Two-index model*
9	0.29	0.0004	0.0002	0.15	0.02
16	5.27	0.003	0.002	1.55	0.09
25	36.86	>7200	>7200	264	0.30
36	>7200	>7200	>7200	>7200	12.55

The success rate increased from 36 to 69%, and 349 instances were solved, with an average running time of 20.3 s for the solved instances.

We were able to solve in much shorter running times all instances with our second backtracking algorithm and with CP applied to the two models. We have plotted in Figure 2 the

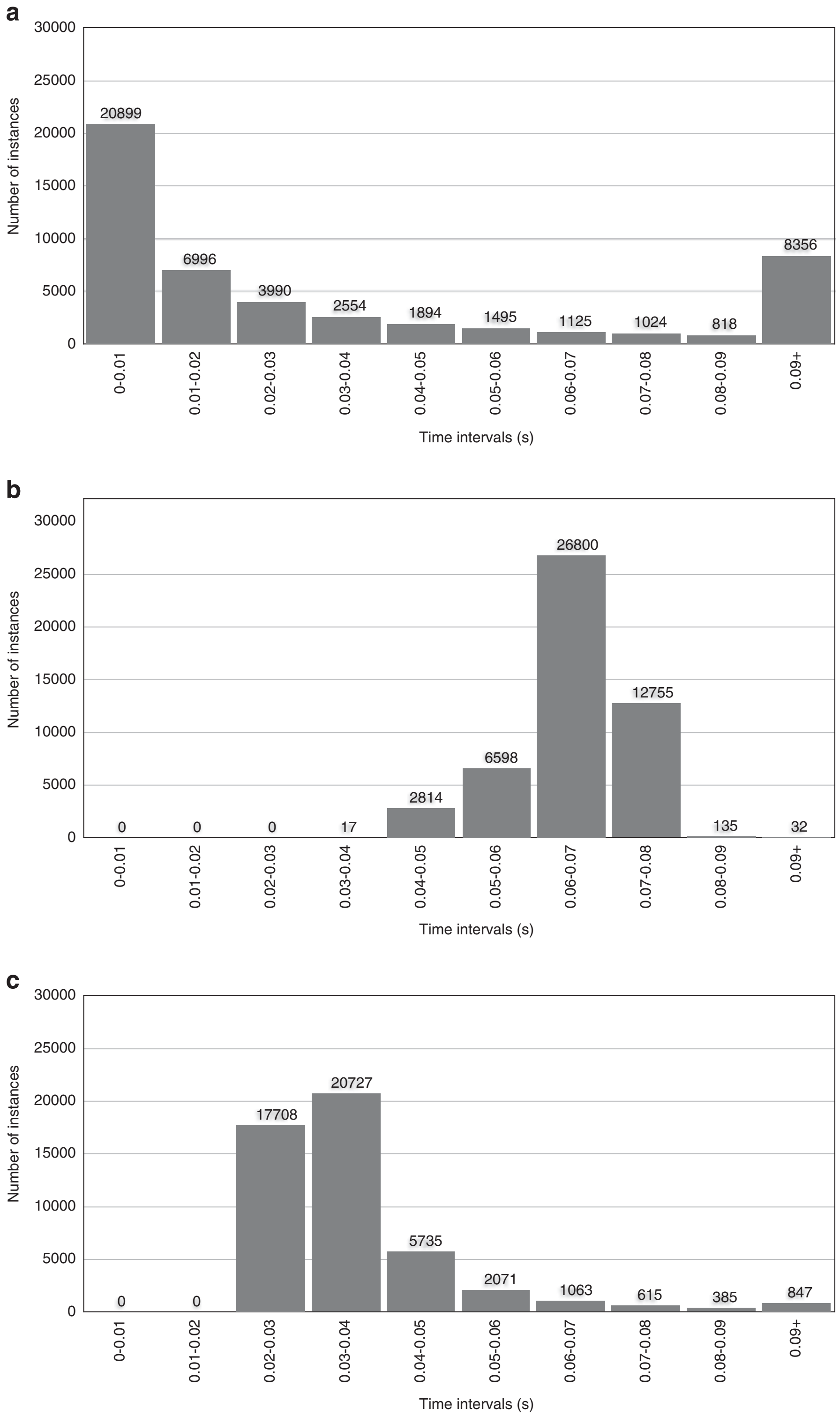


Figure 2 Distribution of the running times for the three algorithms used in the experiments: (a) Running times in seconds for the backtracking algorithm with heuristic; (b) Running times in seconds for CP applied to the three-index formulation; (c) Running times in seconds for CP applied to the two-index formulation.

distributions of the running times of these three algorithms on all 49 151 instances. We have used the same scale and intervals to ease the comparison. Most instances were solved faster by our standalone implementation, but the standard deviation of the running times is also the largest. However, while none of the instances was solved within less than 0.02 s by CP applied to either of the mathematical formulations, more than half of the instances were solved within this time limit by our backtracking algorithm with heuristic moves. The high variation of the running times for the backtracking algorithm over the instances of this data set can be partially explained by the simplicity of its inner workings. As a matter of fact, since the problem is NP-complete, it is expected that some instances will require a large computing time. In the worst case, all possible assignments have to be considered before a solution can be obtained. Owing to the large number of instances in this data set, it is expected that some of them will require extensive enumerations.

Table 2 presents the running time in seconds for the three algorithms used in our experiments. We see that the backtracking implementation is competitive with the state-of-the-art CP solver applied to the three-index model, with similar average running times. However, both algorithms fare worse than CP applied to the two-index model, which takes, on average, only approximately half the time to solve all instances. Interestingly, the CP algorithm works better on the two-index model than on the three-index model when there are givens, whereas the reverse is true when there are no givens (see Table 1). This is due to the fact that permanently assigning number h to position (i, j) means fixing variable $x_{ij}^h = 1$, which in turn fixes all other $x_{ij}^k = 0$ with $k \neq h$, and also all $x_{pj}^h = 0$ with $p \neq i$, $x_{iq}^h = 0$ with $q \neq j$, and $x_{pq}^h = 0$ for (p, q) belonging to the same subgrid as (i, j). Thus, fixing one variable, that is, having one given on the grid, significantly reduces the size of the instance for the three-index model. On the other hand, the number of variables in the two-index model is reduced by only one, that is, the position with the given number.

A comparison of these results with the findings of Lewis (2007a, b) and with our own experiments using the algorithm of Lewis (2007b) shows that the methods proposed in this paper can solve Sudokus with the same number of givens within less computing time. Neighbourhood search (Lewis, 2007b) and the hybrid CP (Lewis, 2007a) algorithms have proved to scale well for instances of size $n = 25$, solving them in about 200 s. Neither of our two backtracking implementations could solve an instance of this size; however, CP could find a solution within 264 s when applied to the three-index model, and within 0.30 s when applied to the two-index model.

We have also analysed the correlations of the running times for every pair of algorithms. In Table 3 we present these correlations, together with Pearson-*r* correlation statistic (Rodgers and Nicewander, 1988) for the null hypothesis that the true correlation is zero. Note that if the the *p*-value is below a given significance threshold (eg, $p < 0.01$), the null hypothesis is then rejected. We have found a significant positive correlation between the backtracking algorithm and CP applied to the two-index formulation. However, we have observed no significant correlation between the running times of CP applied to the two mathematical formulations, or between the backtracking algorithm and CP applied to the three-index model. Thus if an instance is harder for one algorithm it does not follow that it will be harder for the other two algorithms.

In order to better illustrate the effect of the number of givens on the running time of each algorithm, we have obtained the instances used to evaluate the algorithms of Mantere and Koljonen (2006, 2007), which contain different numbers of givens. There are 46 instances of size 9 and the number of givens ranges from 22 to 39. We have not observed any relationship between the number of givens and the running time, but this new set of experiments helps understand the robustness of each method with respect to the input data. The average running times over the 46 instances are shown in Table 4. One can observe that the neighbourhood search algorithm of Lewis (2007b) is able to obtain solutions on an empty grid and on the instances of Mantere and Koljonen (2006, 2007) within comparable running times. Note, however, that the standard deviation of the running times is relatively high on the instances of Mantere and Koljonen (2006, 2007). The longest it took to solve one particular instance was 1.94 s. The backtracking algorithm, on the other hand, performs rather poorly. It has a significantly higher running time, and also a high standard deviation. This can be partly attributed to the fact that some instances take only a few seconds to solve, but one particular instance takes 13.24 s. Both mathematical formulations solved by CP performed very well, with stable running times much lower than in the previous experiments. Here, the standard deviations are very low, particularly for the

Table 2 Average computation time in seconds on the 49 151 instances of size 9 of Royle

Backtracking with heuristic moves	*Three-index model*	*Two-index model*
0.0648	0.0654	0.0370

Table 3 Correlation of the running times and *p*-values on the 49 151 instances of size 9 of Royle obtained with three algorithms

Algorithm	*Backtracking with heuristic moves*	*Three-index model*	*Two-index model*
Backtracking with heuristic moves	1.00	−0.011	0.276
		$p = 0.011$	$p < 0.00001$
Three-index model	—	1.00	−0.006
			$p = 0.147$
Two-index model	—	—	1.00

Table 4 Average running time (standard deviation) in seconds on empty grids and on the 46 instances of size 9 of Mantere and Koljonen (2006, 2007)

Input	*Metaheuristic of Lewis (2007b)*	*Backtracking with heuristic moves*	*Three-index model*	*Two-index model*
Empty grid	0.29	0.0002	0.15	0.02
Instances of Mantere and Koljonen (2006, 2007)	0.33 (0.449)	0.57 (2.176)	0.042 (0.040)	0.00039 (2.465×10^{-5})

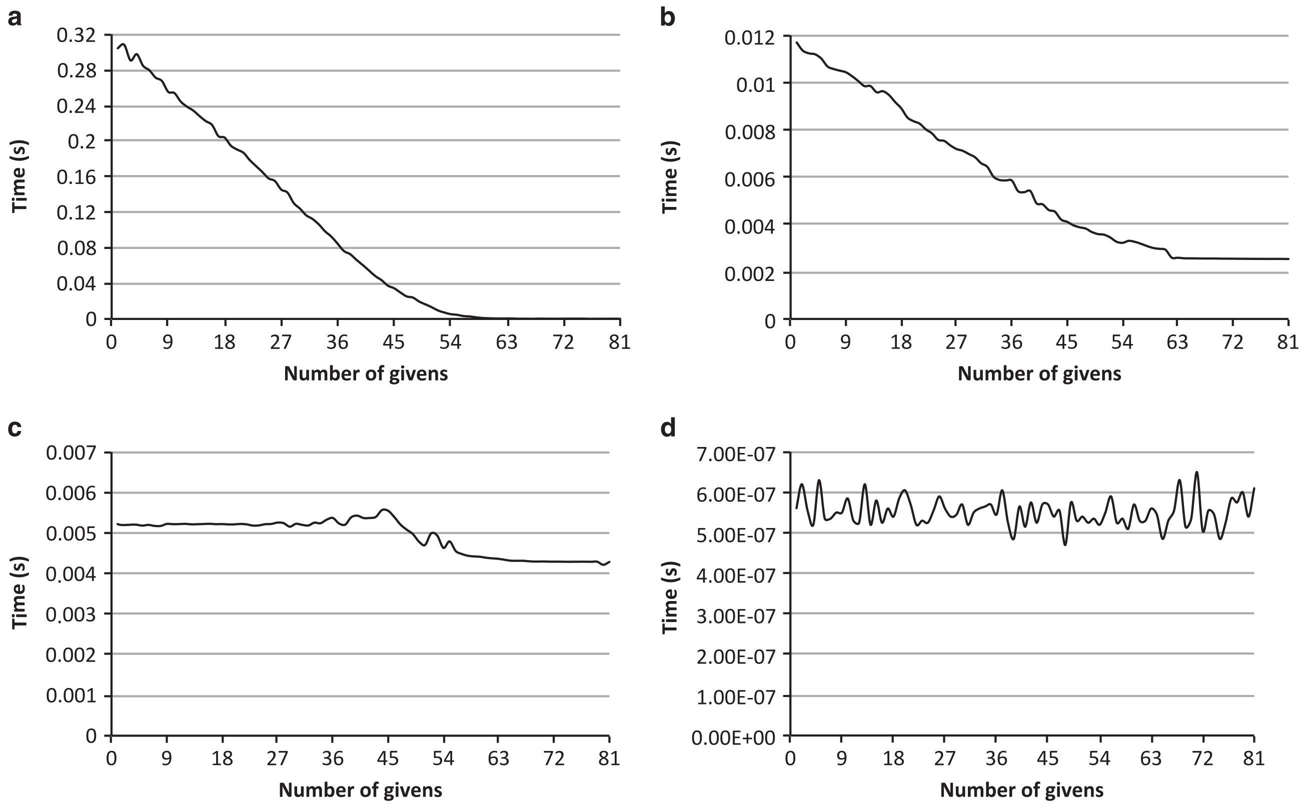


Figure 3 Performance of four algorithms with respect to the number of givens on Sudokus of size 9: (a) Algorithm of Lewis (2007a); (b) CP applied to the three-index model; (c) CP applied to the two-index model; (d) Backtracking with heuristic moves.

two-index model. This suggests that they tend to perform better with more givens. These instances were also solved by genetic algorithms in Mantere and Koljonen (2006, 2007). These experiments were executed in a different computing environment (2.8 GHz quad-core 64-bit Intel Xeon processor), but the running times were significantly longer, with an average of 3.8 s.

In order to assess the performance of the algorithms with respect to the number of givens, we have taken 100 solved Sudoku instances of size 9 and performed the following experiments. We have created 81 instances out of each of them, starting from an empty grid, and gradually adding a given randomly selected from the solved instance, thus obtaining 81 instances with 1–81 givens. Repeating the process for the 100 instances, we have then generated 100 different instances with one given, 100 instances with two givens, and so on. We have then solved each of the 8100 instances using all four algorithms just evaluated.

We depict in Figure 3 the performance of each algorithm measured by the average running time with respect to the number of givens. Note the differences in vertical scale in the four figures. We observe that the algorithms of Lewis (2007b) (Figure 3(a)) and CP applied to the three-index model (Figure 3 (b)) are highly dependent on the number of givens, with a running time clearly decreasing when the number of givens increase. Moreover, we observe that these algorithms are able to obtain a solution virtually instantaneously after about 60 givens. Yet, it is clear that there is an increase in performance of about one order of magnitude between the metaheuristic of Lewis and CP applied to the three-index model. The CP algorithm applied to the two-index model (Figure 3(c)) performs orders of magnitude faster than either of the first two algorithms on instances with less than 40 givens, but is almost unaffected by the number of givens. Finally, these experiments confirm that our backtracking algorithm with heuristic moves is clearly the fastest algorithm for Sudokus of size 9, as shown in Figure 3(d). It also performs very well irrespective of the number of givens. The running times are insignificant and the variations are mostly noise.

5. Conclusion

We have described two mathematical models and we have compared four enumerative algorithms for Sudokus of varying sizes. The first two algorithms are simple backtracking algorithms while the remaining two consist of applying CP to three-index and two-index mathematical programming formulations. The first backtracking algorithm is clearly dominated by the second one that selects cells to be filled based on a heuristic criterion. In addition, CP works better on the non-linear two-index formulation than on the linear three-index formulation. We have shown through extensive computational experiments on almost 50 000 instances that our implementation of a backtracking algorithm clearly outperforms CP applied to either formulation in terms of solving most of the instances faster, but is slower on average. Moreover, our algorithm outperforms existing neighbourhood search and genetic algorithms on instances of sizes $n=9$ and 16. We have also evaluated the performance of several algorithms with respect to the number of givens. While neighbourhood search and CP applied to the three-index model perform better when the number of givens increases, our backtracking algorithm and CP applied to the two-index model remain rather stable with respect to the number of givens in the input data. We have shown that for highly constrained instances CP is a viable tool to solve the problem quickly. Moreover, we have shown that a well-designed non-linear model can outperform a linear one in some applications. Finally, we have shown that backtracking policies can have a major effect on the ability to obtain a solution and on the running times if one takes advantage of the structure of the problem.

Acknowledgements—The authors thank the associate editor and the anonymous referees for their valuable comments. This work was partly supported by the Canadian Natural Sciences and Engineering Research Council under grant 39682-10. This support is gratefully acknowledged.

References

Applegate DL and Cook WJ (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on Computing* **30**(2): 149–156.

Apt K (2003). *Principles of Constraint Programming*. Cambridge University Press: Cambridge.

Bartlett AC and Langville AN (2006). An integer programming model for the Sudoku problem. http://www.cofc.edu/~langvillea/Sudoku/sudoku2.pdf, accessed 21 May 2013.

Colbourn CJ, Colbourn MJ and Stinson DR (1984). The computational complexity of recognizing critical sets. In: Koh K and Yap H (eds). *Graph Theory Singapore 1983, Volume 1073 of Lecture Notes in Mathematics*. Springer: Heidelberg: pp 248–253.

Dànes J and Keedwell AD (1974). *Latin Squares and their Applications*. English Universities Press: London.

Felgenhauer B and Jarvis F (2005). Enumerating possible Sudoku grids. http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf, accessed 7 January 2013.

Garns H (1979). Number place. *Dell Pencil Puzzles & Word Games* (16): 6.

Golomb SW and Baumert LD (1965). Backtrack programming. *Journal of the ACM* **120**(4): 516–524.

Higgins A, Kozan E and Ferreira L (1996). Optimal scheduling of trains on a single line track. *Transportation Research Part B: Methodological* **300**(2): 147–161.

Jensen TR and Toft B (1994). *Graph Coloring Problems*. Vol. 39. Wiley-Interscience: New York.

Laporte G and Pesant G (2004). A general multi-shift scheduling system. *Journal of the Operational Research Society* **55**(11): 1208–1217.

Lewis R (2007a). On the combination of constraint programming and stochastic search: The Sudoku case. http://rhydlewis.eu/papers/HM2007.pdf, accessed 7 January 2013.

Lewis R (2007b). Metaheuristics can solve Sudoku puzzles. *Journal of Heuristics* **130**(4): 387–401.

Lustig IJ and Puget J-F (2001). Program does not equal program: Constraint programming and its relationship to mathematical programming. *Interfaces* **310**(6): 29–53.

Mantere T and Koljonen J (2006). Solving and rating Sudoku puzzles with genetic algorithms. In: Hyvönen E, Kauppinen T, Kortela J, Raiko T, Laukkanen M and Viljanen K (eds). *New Developments in Artificial Intelligence and the Semantic Web-Proceedings of the 12th Finnish Artificial Conference*. Finnish Artificial Intelligence Society: Espoo, Finland: pp 86–92.

Mantere T and Koljonen J (2007). Solving, rating and generating Sudoku puzzles with GA. In: *2007 IEEE Congress on Evolutionary Computation*. IEEE: Singapore, pp 1382-1389.

McGuire G, Tugemann B and Civario G (2012). There is no 16-clue Sudoku: Solving the Sudoku minimum number of clues problem. http://arxiv.org/pdf/1201.0749v1.pdf, accessed 7 January 2013.

Naveh Y, Richter Y, Altshuler Y, Gresh DL and Connors DP (2007). Workforce optimization: Identification and assignment of professional workers using constraint programming. *IBM Journal of Research and Development* **51**(3.4): 263–279.

Rodgers JL and Nicewander WA (1988). Thirteen ways to look at the correlation coefficient. *The American Statistician* **420**(1): 59–66.

Rossi F, Van Beek P and Walsh T (2006). *Handbook of Constraint Programming*. Vol. 2. Elsevier: Amsterdam.

Royle G. Sudoku dataset: 49151 instances with 17 givens, University of Western Australia. http://school.maths.uwa.edu.au/~gordon/sudokumin.php, accessed 20 September 2013.

Russell E and Jarvis F (2005). There are 5472730538 essentially different Sudoku grids. http://www.afjarvis.staff.shef.ac.uk/sudoku/sudgroup.html, accessed 30 May 2013.

Sellmann M, Zervoudakis K, Stamatopoulos P and Fahle T (2002). Crew assignment via constraint programming: Integrating column generation and heuristic tree search. *Annals of Operations Research* **1150**(1): 207–225.

Simonis H (2005). Sudoku as a constraint program. http://4c.ucc.ie/~hsimonis/sudoku.pdf, accessed 7 January 2013.

Tadei R and Mancini S (2006). Sudoku game theory, models and algorithms. http://www.polito.it/polymath/htmlS/Studenti/Tesi/SudokuGame/tesi_Mancini_Simona%20SUDOKU.pdf, accessed 21 May 2013.

Takayuki Y and Takahiro S (2003). Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* **860**(5): 1052–1060.

Walker RJ (1960). An enumerative technique for a class of combinatorial problems. In: Bellman RE and Hall M (eds) *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society: Providence, RI. Vol. 10, pp 91–94.

Received 08 January 2013;
accepted 12 August 2013 after two revision