Hindawi Publishing Corporation
The Scientific World Journal
Volume 2014, Article ID 465359, 12 pages
http://dx.doi.org/10.1155/2014/465359

Research Article

A Prefiltered Cuckoo Search Algorithm with Geometric Operators for Solving Sudoku Problems

Ricardo Soto,[1,2] Broderick Crawford,[1,3] Cristian Galleguillos,[1] Eric Monfroy,[4] and Fernando Paredes[5]

[1] *Pontificia Universidad Católica de Valparaíso, 2362807 Valparaíso, Chile*
[2] *Universidad Autónoma de Chile, 7500138 Santiago, Chile*
[3] *Universidad Finis Terrae, 7501015 Santiago, Chile*
[4] *CNRS, LINA, University of Nantes, 44322 Nantes, France*
[5] *Escuela de Ingeniería Industrial, Universidad Diego Portales, 8370109 Santiago, Chile*

Correspondence should be addressed to Ricardo Soto; ricardo.soto@ucv.cl

Received 11 November 2013; Accepted 30 December 2013; Published 23 February 2014

Academic Editors: Z. Cui and X. Yang

Copyright © 2014 Ricardo Soto et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The Sudoku is a famous logic-placement game, originally popularized in Japan and today widely employed as pastime and as testbed for search algorithms. The classic Sudoku consists in filling a 9×9 grid, divided into nine 3×3 regions, so that each column, row, and region contains different digits from 1 to 9. This game is known to be NP-complete, with existing various complete and incomplete search algorithms able to solve different instances of it. In this paper, we present a new cuckoo search algorithm for solving Sudoku puzzles combining prefiltering phases and geometric operations. The geometric operators allow one to correctly move toward promising regions of the combinatorial space, while the prefiltering phases are able to previously delete from domains the values that do not conduct to any feasible solution. This integration leads to a more efficient domain filtering and as a consequence to a faster solving process. We illustrate encouraging experimental results where our approach noticeably competes with the best approximate methods reported in the literature.

1. Introduction

The Sudoku is a logic-based placement puzzle, widely present as pastime game in newspapers and magazines. It was initially popularized in Japan during the eighties, but today it is a worldwide popular game and a useful benchmark for testing artificial intelligence solving techniques. A Sudoku puzzle consists in filling a board of 9×9 subdivided into subgrids of size 3×3 so that each row, column, and subgrid contains different digits from 1 to 9. A Sudoku problem includes prefilled cells, namely, the “givens,” which cannot be changed or moved (see Figure 1). Certainly, the amount of givens has limited or no impact on the difficulty of the problem. The difficulty is mostly dependent on the positioning of the givens along the puzzle. A useful difficulty classification including easy, medium, and hard Sudokus has been proposed by Mantere and Koljonen [1].

The literature reports various approaches to solve Sudoku puzzles. For instance, exact methods such as constraint programming [2–4] and boolean satisfiability [5] are well-known candidates for the efficient handling of such kind of puzzles. In the context of approximate methods, genetic programming [1] and metaheuristics in general [6–10] have illustrated promising results. Additional, but less traditional, Sudoku solving techniques have been proposed as well, such as Sinkhorn balancing [11], rewriting rules [12], and entropy minimization [13].

In this paper, we focus on approximate methods. We propose a new algorithm for the efficient solving of Sudoku instances based on cuckoo search, geometric operators, and prefiltering phases. The cuckoo search is a relatively modern nature-inspired metaheuristic [14–18] to which we introduce geometric operators in order to correctly move to promising regions of a discrete space of solutions. This combination

Subgrid Column

Row

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

Figure 1: Sudoku puzzle instance.

is additionally enhanced with a domain reducer component based on local consistencies. The idea is to previously delete from the search space the values that do not conduct to any feasible solution. This integration straightforwardly alleviates the work of the metaheuristic leading to a faster solving process. We illustrate encouraging experimental results where our approach noticeably competes with the best approximate methods reported in the literature.

This paper is organized as follows. In Section 2, we describe the previous work. Section 3 presents the classic cuckoo search algorithm. The geometric operators and the prefiltering phase employed are illustrated and exemplified in Sections 4 and 5, respectively. The resulting new cuckoo search algorithm is presented in Section 6, followed by the corresponding experimental results. Finally, in Section 8, we conclude and give some directions for future work.

2. Related Work

Sudoku puzzles have been solved with various techniques during the last decades. For instance, complete methods such as Boolean satisfiability and constraint satisfaction can clearly be used to solve Sudokus [3–5, 19]. In this paper, we focus on incomplete search methods, specially on solving hard instances of such a puzzle. Within this scenario, different solutions have been suggested, mainly based on metaheuristics. For instance, Lewis [7] models the puzzle as an optimization problem where the number of incorrectly placed digits on the board must be minimized. The model is solved by using simulated annealing, but the approach is mostly focused on producing valid Sudokus than on the performance of the resolution. In [10], an ant colony algorithm is proposed, where the problem is modeled in an opposite form: maximizing the number of correctly filled cells. The best result completes only 76 out of 81 cells of the puzzle. In [8], a particle swarm optimizer (PSO) for solving Sudokus is presented, but the goal of authors was rather to validate the use of geometric operators in PSO for complex combinatorial spaces.

In [6], a hill-climbing algorithm for Sudokus is reported. The approach succeeds in solving easy Sudoku instances, failing for medium and hard ones. In the same work, a genetic algorithm (GA) outperforms the hill-climber previously presented. Such a GA is tuned with geometric operators, in particular Hamming space crossovers and swap space crossovers, reporting solutions for a hard Sudoku. In Mantere and Koljonen [1], another GA is proposed that succeeds for easy and medium instances, but it only reaches the optimum in 2 out of 30 tries for a hard Sudoku. A cultural algorithm is proposed by the same authors [9], but it is generally outperformed by the GA previously reported. In Soto et al. [20], a tabu search is tuned with a prefiltered phase, being capable of solving 30 out of 30 tries for a hard Sudoku.

3. Cuckoo Search

Cuckoo search is a nature-inspired metaheuristic, based on the principle of the brood parasitism of some cuckoo species. This kind of bird has an aggressive reproduction strategy, which is based on the use of foreign nests for incubation. Cuckoos proceed by laying their eggs in nests from other bird species, removing the other bird eggs to increase incubation probability. Eventually, cuckoo eggs may be discovered by the host bird, which might act in two ways: taking off the alien eggs or simply abandoning its nest and building a new one elsewhere. In practice, an egg represents a solution and cuckoo eggs represent potentially better solutions than the current ones in nests. In the simplest form each nest has only one egg.

The cuckoo search procedure for minimization is described in Algorithm 1. The process begins by randomly generating an initial population of host nests. Next, the algorithm iterates until a given stop criterion is reached, which is commonly a maximum number of iterations. At line 3, a new solution is created, normally by employing a random walk via Lévy flights. Equation (1) describes such a random walk, where x_i^{t+1} is the new solution, t corresponds to the iteration number, and the product $\oplus$ means entrywise multiplications. The α parameter is the step size, where $\alpha > 0$, and determines how far the process can go for a fixed number of iterations. Then, at line 4, an Egg_j is randomly chosen to be then compared with the previous one in order to keep the egg exhibiting the best cost. Finally, the worse nests are abandoned depending on the probability p_a and new solutions are built:

$$x_i^{t+1} = x_i^t + \alpha \oplus \text{Lévy}(\lambda),$$
$$\text{Lévy} \sim u = t^{-\lambda}, \quad (1 < \lambda \leq 3). \tag{1}$$

4. Geometric Operators

The cuckoo search has been originally designed for continuous domains, while Sudokus own discrete values. Then, a discretization phase for the CS algorithm is mandatory to correctly explore the potential solutions. The discretization phase applied here has been inspired from the work reported in [8], where a particle swarm optimization algorithm is adapted to solve discrete domains. The idea relies on the use of geometric-based operators able to correctly move to

```
Input: Nest_size, α, λ
Output: Egg_best
(1)  Nests ← GenerateInitialPopulation(Nest_size)
(2)  While ¬ StopCondition do
(3)    Egg_i ← GetCuckooByLevyFlight(Nests, α, λ)
(4)    Egg_j ← ChooseRamdomlyFrom(Nests)
(5)    If cost(Egg_i) ≤ cost(Egg_j)
(6)      Egg_j ← Egg_i
(7)    End If
(8)    Egg_best ← FindCurrentBest(Nests)
(9)    Nests ← AbandonWorseNests(p_a, Nests)
(10)   Nests ← BuildNewSolutions(Egg_best)
(11) End While
```

Algorithm 1: Cuckoo search.

```
Input: Parent_1[], Parent_2[]
Output: Child[]
(1)  Child ← SelectRandomSegmentFrom(Parent_1)
(2)  List_val ← SelectNotCopiedToChild(Parent_2, Child)
(3)  For Each val ∈ List_val
(6)    lval ← val
(4)    V ← Parent_1[IndexOf(val, Parent_2)]
(5)    If IndexOf(V, Parent_2) ∈ SegmentOf(Child)
(6)      val ← V
(6)      Go To (4)
(7)    Else
(8)      Child[IndexOf(V, Parent_2)] ← lval
(9)    End If
(10) End For Each
(11) CopyRemaining(Child, Parent_2)
```

Algorithm 2: PMX crossover.

promising regions of a discrete search space. In particular, for this work, we employ the partially matched crossover, the geometric crossover, and the feasible geometric mutation, which are described in the following.

4.1. Partially Matched Crossover. The partially matched crossover (PMX) basically works with two parents, creating two crossover points that are selected at random and then PMX proceeds by position swap. The PMX process is described in Algorithm 2.

At the beginning, a segment from $Parent_1$ is randomly selected and copied to the child. Then, looking in the same segment positions in $Parent_2$, each value not copied to the child is stored in a list. Then, for each value in this list, the **V** value is located in $Parent_1$ in the position given by the index of *val* in $Parent_2$. Then, an if-else conditional operates as follows: if the index of **V** is present in the original segment, **V** becomes the new *val* and the process goes to line 4; otherwise, *lval* is inserted into the child in the position given by the index of **V** in $Parent_2$. Finally, the remaining positions from $Parent_2$ are copied to the child. The PMX operator applied to the Sudoku can be seen in Example 1.

Example 1 (PMX crossover). Let two rows located in the same position from different solutions of a Sudoku instance be the parents. The corresponding child produced by the PMX crossover is constructed as follows.

(1) A random segment of consecutive digits from $Parent_1$ is copied to the child. Assuming that 1 corresponds to the index of the first position, the segment has size 5, from index 4 to 8:

$$\begin{array}{r l} Parent_1: & 8\ 4\ 7\ \overline{3\ 6\ 2\ 5\ 1}\ 9 \\ Parent_2: & 9\ 1\ 2\ \overline{3\ 4\ 5\ 6\ 7}\ 8 \\ Child: & -\ -\ -\ \overline{3\ 6\ 2\ 5\ 1}\ - \end{array} \quad (2)$$

(2) "4" is the first value in the observed segment of $Parent_2$ that is not present in the child. Then, $val = 4$, and the index of *val* in $Parent_2$ is "5." Hence, the **V** value corresponds to "6." Next, the index of **V** in $Parent_2$ is "7." This index exists in the observed segment, so the process comes back to line 4 using "6" as *val*:

$$\begin{array}{r l} Parent_1: & 8\ 4\ 7\ \overline{3\ \mathbf{6}\ 2\ 5\ 1}\ 9 \\ Parent_2: & 9\ 1\ 2\ \overline{3\ \mathbf{4}\ 5\ \mathbf{6}\ 7}\ 8 \\ Child: & -\ -\ -\ \overline{3\ 6\ 2\ 5\ 1}\ - \end{array} \quad (3)$$

Input: $Parent_1[\,], Parent_2[\,], Parent_3[\,]$
Output: $Child[\,]$
(1) $Mask \leftarrow$ `InitializeMask()`
(1) **For Each** $i \in \{1, \ldots, Mask_{length}\}$
(2) **If** $Mask[i] = 1$
(3) $Child[i] \leftarrow Parent_1[i]$
(4) $Parent_2 \leftarrow \texttt{Swap}(Parent_1[i], Parent_2, i)$
(5) $Parent_3 \leftarrow \texttt{Swap}(Parent_1[i], Parent_3, i)$
(6) **Else If** $Mask[i] = 2$
(7) $Child[i] \leftarrow Parent_2[i]$
(8) $Parent_1 \leftarrow \texttt{Swap}(Parent_2[i], Parent_1, i)$
(9) $Parent_3 \leftarrow \texttt{Swap}(Parent_2[i], Parent_3, i)$
(10) **Else**
(11) $Child[i] \leftarrow Parent_3[i]$
(12) $Parent_1 \leftarrow \texttt{Swap}(Parent_3[i], Parent_1, i)$
(13) $Parent_2 \leftarrow \texttt{Swap}(Parent_3[i], Parent_2, i)$
(14) **End If**
(15) **End For Each**

ALGORITHM 3: Multiparental sorting crossover.

(3) Now, using "6" as *val*, the new **V** is "5." Then, the index of "5" in $Parent_2$ also appears within the segment. So, the process comes back again to line 4 using "5" as *val*:

$$\begin{array}{rccccccccc} Parent_1: & 8 & 4 & 7 & \overline{3} & \overline{6} & \overline{2} & \overline{\mathbf{5}} & \overline{1} & 9 \\ Parent_2: & 9 & 1 & 2 & \overline{3} & \overline{4} & \overline{\mathbf{5}} & \overline{\mathbf{6}} & \overline{7} & 8 \\ Child: & - & - & - & \overline{3} & \overline{6} & \overline{2} & \overline{5} & \overline{1} & - \end{array} \tag{4}$$

(4) Then, the **V** value is "2," and its index in $Parent_2$ does not appear within the segment. Hence, we obtain a position in the child for the value "4" from step 2:

$$\begin{array}{rccccccccc} Parent_1: & 8 & 4 & 7 & \overline{3} & \overline{6} & \overline{\mathbf{2}} & \overline{5} & \overline{1} & 9 \\ Parent_2: & 9 & 1 & \mathbf{2} & \overline{3} & \overline{4} & \overline{\mathbf{5}} & \overline{6} & \overline{7} & 8 \\ Child: & - & - & \mathbf{4} & \overline{3} & \overline{6} & \overline{2} & \overline{5} & \overline{1} & - \end{array} \tag{5}$$

(5) "7" is the next value from $Parent_2$ in the segment that is not already included in the child. Then, "1" is the **V** value, whose index does not appear within the segment as well. Hence, a position for the value "7" is obtained in the child:

$$\begin{array}{rccccccccc} Parent_1: & 8 & 4 & 7 & \overline{3} & \overline{6} & \overline{2} & \overline{5} & \overline{\mathbf{1}} & 9 \\ Parent_2: & 9 & \mathbf{1} & 2 & \overline{3} & \overline{4} & \overline{5} & \overline{6} & \overline{\mathbf{7}} & 8 \\ Child: & - & \mathbf{7} & 4 & \overline{3} & \overline{6} & \overline{2} & \overline{5} & \overline{1} & - \end{array} \tag{6}$$

(6) Now, everything else from $Parent_2$ is copied down to the child:

$$\begin{array}{rccccccccc} Parent_1: & 8 & 4 & 7 & \overline{3} & \overline{6} & \overline{2} & \overline{5} & \overline{1} & 9 \\ Parent_2: & \mathbf{9} & 1 & 2 & \overline{3} & \overline{4} & \overline{5} & \overline{6} & \overline{7} & \mathbf{8} \\ Child: & \mathbf{9} & 7 & 4 & \overline{3} & \overline{6} & \overline{2} & \overline{6} & \overline{1} & \mathbf{8} \end{array} \tag{7}$$

4.2. Multiparental Sorting Crossover. This operator may employ multiple parents; our approach based on [8] uses three, where each one represents a row of a potential solution: one from the best solution of all generations, one from the best solution of the current generation, and one from the current solution. Each parent is associated with a weight (w) according to (8) in order to control the relevance of each solution in the generation of the new one. The influence of parents given by the weights is reflected in a mask used in the process.

The multiparental crossover is described in Algorithm 3. The three parents and the mask are the input of the procedure, and the child resulting from the crossover is the output. A for each loop is used to scan the mask, where every entry indicates which parent the other two parents need to be equal to for that specific position. The replacement depends on the value of the mask and it is performed by swapping the corresponding values as indicated in the conditionals stated at lines 2, 6, and 10. The swapping process is described in Algorithm 4:

$$w_1 + w_2 + w_3 = 1, \text{ such that } w_i > 0 \quad \forall i \in \{1, 2, 3\}. \tag{8}$$

Example 2 (multiparental sorting crossover). Let $Parent_1$ be a row from the best solution of all generations, $Parent_2$ a row from the best solution of the current generation, and $Parent_3$ a row from the current solution. We employ 0.55 as the weight for $Parent_1$, 0.33 for $Parent_2$, and 0.12 for $Parent_3$. Those weights represent the percentage of appearances of the given parent within the mask. For instance, $Parent_1$ appears five times in the mask, $Parent_2$ three times, and $Parent_3$ once. The corresponding child produced by the multiparental sorting crossover is constructed as follows.

(1) A mask of parent length is randomly generated according to the parent weights:

$$\begin{array}{rccccccccc} Mask: & 3 & 1 & 2 & 1 & 1 & 1 & 1 & 2 & 2 \\ Parent_1: & 8 & 4 & 7 & 3 & 6 & 2 & 5 & 1 & 9 \\ Parent_2: & 9 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ Parent_3: & 4 & 7 & 9 & 3 & 6 & 2 & 5 & 1 & 8 \end{array} \tag{9}$$

Input: *value*, *Parent*[], *pos*
Output: *Parent*[]
(1) **If** $Parent[pos] \neq value$
(2) **For Each** $i \in \{1, \ldots, Parent_{length}\}$
(3) **If** $Parent[i] = value$
(4) $aux \leftarrow Parent[i]$
(5) $Parent[i] \leftarrow Parent[pos]$
(6) $Parent[pos] \leftarrow aux$
(7) **End If**
(8) **For Each**
(9) **End If**

Algorithm 4: Swap.

Input: *initSol*[][], *sol*[][]
Output: *sol*[][]
(1) *rows* ← `ChooseRowsRandomly`(*initSol*)
(2) **For Each** *row* ∈ *rows*
(3) *pos*1, *pos*2 ← `ChooseTwoEmptyCells`(*row*)
(4) *aux* ← `sol`[*row*][*pos*1]
(5) `sol`[*row*][*pos*1] ← `sol`[*row*][*pos*2]
(6) `sol`[*row*][*pos*2] ← *aux*
(7) **End For Each**

Algorithm 5: Feasible geometric mutation.

(2) The first value of the mask corresponds to parent "3", and then the first value of $Parent_1$ and $Parent_2$ needs to be equal to the first value of $Parent_3$, which is "4." To this end, in $Parent_1$ and $Parent_2$, the first cell is swapped with the cell that holds the value "4":

$$\begin{array}{rccccccccc} Mask\text{:} & \mathbf{3} & 1 & 2 & 1 & 1 & 1 & 2 & 2 & 3 \\ Parent_1\text{:} & \underline{4} & \underline{8} & 7 & 3 & 6 & 2 & 5 & 1 & 9 \\ Parent_2\text{:} & \underline{4} & 1 & 2 & 3 & \underline{9} & 5 & 6 & 7 & 8 \\ Parent_3\text{:} & \mathbf{4} & 7 & 9 & 3 & 6 & 2 & 5 & 1 & 8 \\ \hline Child & 4 & & & & & & & & \end{array} \quad (10)$$

(3) Next, the second value from the mask is "1." The swapping process is analogous:

$$\begin{array}{rccccccccc} Mask\text{:} & 3 & \mathbf{1} & 2 & 1 & 1 & 1 & 2 & 2 & 3 \\ Parent_1\text{:} & 4 & \mathbf{8} & 7 & 3 & 6 & 2 & 5 & 1 & 9 \\ Parent_2\text{:} & 4 & \underline{8} & 2 & 3 & 9 & 5 & 6 & 7 & \underline{1} \\ Parent_3\text{:} & 4 & \underline{8} & 9 & 3 & 6 & 2 & 5 & 1 & \underline{7} \\ \hline Child & 4 & 8 & & & & & & & \end{array} \quad (11)$$

(4) Following the same procedure, the last step is shown below obtaining 4 8 2 3 6 7 9 5 1 as the new child:

$$\begin{array}{rccccccccc} Mask\text{:} & 3 & 1 & 2 & 1 & 1 & 1 & 2 & 2 & 3 \\ Parent_1\text{:} & 4 & 8 & 2 & 3 & 6 & 7 & 9 & 5 & 1 \\ Parent_2\text{:} & 4 & 8 & 2 & 3 & 6 & 7 & 9 & 5 & 1 \\ Parent_3\text{:} & 4 & 8 & 2 & 3 & 6 & 7 & 9 & 5 & 1 \\ \hline Child & 4 & 8 & 2 & 3 & 6 & 7 & 9 & 5 & 1 \end{array} \quad (12)$$

4.3. Feasible Geometric Mutation. This is a simple operator used to maintain diversity in the solutions. It swaps two non-fixed elements in a row guaranteeing that mutation is applied only over the cells with no given value. The procedure is described in Algorithm 5.

Example 3 (feasible geometric mutation). Let us consider a given Sudoku row and a solution candidate row, as shown below:

$$\begin{array}{rccccccccc} \text{Sudoku problem row:} & - & - & - & 3 & - & - & - & - & - \\ \text{Solution candidate row:} & 9 & 7 & 4 & 3 & 6 & 2 & 5 & 1 & 8 \end{array} \quad (13)$$

The mutation is allowed in any cell except for cell 4, which owns the value 3 as given for the Sudoku instance. Examples of allowed and forbidden mutations are depicted below:

$$\begin{array}{rccccccccc} \text{allowed mutation:} & 9 & \mathbf{6} & 4 & 3 & \mathbf{7} & 2 & 5 & 1 & 8 \\ \text{forbidden mutation:} & 9 & 7 & 4 & \mathbf{4} & 6 & 2 & 5 & 1 & 8 \end{array} \quad (14)$$

5. Prefiltering Phase

In the presence of unfeasible solutions, the cuckoo procedure is responsible for detecting and discarding them in order to conduct the search to feasible regions of the space of solutions. The goal of the prefiltering phase is to alleviate the work of the cuckoo algorithm by previously eliminating those unfeasible values. This is possible by representing the Sudoku as a constraint network [4] and then applying efficient filtering techniques from the constraint satisfaction domain. In this context, arc-consistency [2] is a widely employed

local consistency for filtering algorithms. Arc-consistency was initially defined for binary constraint [21, 22]. We employ here the more general filtering for nonarbitrary constraints named generalized arc-consistency (GAC). The idea is to enforce a local consistency to the problem in a process called constraint propagation. Before detailing this process, let us introduce some necessary definitions [23].

Definition 4 (constraint). A constraint c is a relation defined on a sequence of variables $X(c) = (x_{i_1}, \ldots, x_{i_{|X(c)|}})$, called the scheme of c; c is the subset of $\mathbb{Z}^{|X(c)|}$ that contains the combinations of tuples $\tau \in \mathbb{Z}^{|X(c)|}$ that satisfy c. $|X(c)|$ is called the arity of c. A constraint c with scheme $X(c) = (x_1, \ldots, x_k)$ is also noted as $c(x_1, \ldots, x_k)$.

Definition 5 (constraint network). A constraint network also known as constraint satisfaction problem (CSP) is defined by a triple $N = \langle X, D, C \rangle$, where

(i) X is a finite sequence of integer variables $X = (x_1, \ldots, x_n)$;

(ii) D is the corresponding set of domains for X; that is, $D = D(x_1) \times \cdots \times D(x_n)$, where $D(x_i) \subset \mathbb{Z}$is the finite set of values that variable x_i can take;

(iii) C is a set of constraints $C = \{c_1, \ldots, c_e\}$, where variables in $X(c_j)$ are in X.

Example 6 (the Sudoku as a constraint network). Let $\langle X, D, C \rangle$ be the constraint network, which is composed of the following.

(i) $X = (x_{1,1}, \ldots, x_{n,m})$ is the sequence of variables, and $x_{i,j} \in X$ identifies the cell placed in the ith row and jth column of the Sudoku matrix, for $i = 1, \ldots, n$ and $i = 1, \ldots, m$.

(ii) D is the corresponding set of domains, where $D(x_{i\,j}) \in D$ is the domain of the variable x_{ij}.

(iii) C is the set of constraints defined as follows.

(a) To ensure that values are different in rows and columns, $x_{k,i} \neq x_{k,j} \wedge x_{i,k} \neq x_{j,k}$, for all $(k \in [1, 9],\ i \in [1, 9],\ j \in [i + 1, 9])$.

(b) To ensure that values are different in subgrid: $x_{(k1-1)*3+k2,(j1-1)*3+j2} \neq x_{(k1-1)*3+k3,(j1-1)*3+j3}$, for all $(k1, j1, k2, j2, k3, j3 \in [1, 3] \mid k2 \neq k3 \wedge j2 \neq j3)$.

Definition 7 (projection). A projection of c on Y is denoted as $\pi_{Y(c)}$, which defines the relation with scheme Y that contains the tuples that can be extended to a tuple on $X(c)$ satisfying c.

Definition 8 ((generalized) arc-consistency). Given a network $N = \langle X, D, C \rangle$, a constraint $c \in C$, and a variable $x_i \in X(c)$,

(i) a value $v_i \in D(x_i)$ is consistent with $c \in D$ if and only if there exists a valid tuple tau satisfying c such that $v_i = \tau[\{x_i\}]$; such a tuple is called a support for (x_i, v_i) on c;

(ii) the domain D is (generalized) arc-consistent on c for x_i if and only if all the values in $D(x_i)$ are consistent with c in D; that is, $D(x_i) \subseteq \pi_{x_i}(c \cap \pi_{X(c)}(D))$;

(iii) the network N is (generalized) arc-consistent if and only if D is (generalized) arc-consistent for all variables in X on all constraints in C.

The filtering process is achieved by enforcing the arc-consistency on the problem. This can be carried out by using Algorithms 6 and 7. The idea is to revise the arcs (the constraint relation between variables) by removing the values from $D(X_i)$ that lead to inconsistencies with respect to a given constraint. Such a revision process is done by Algorithm 6, which takes each value $v_i \in D(x_i)$ (line 2) and analyses the space $\tau \in c \cap \pi_{X(c)}(D)$, searching for a support on constraint c (line 3). If support does not exist, the value v_i is eliminated from $D(x_i)$. Finally, the procedure informs if the domain has been modified by returning the corresponding Boolean value (line 8).

The role of Algorithm 7 is to guarantee that every domain is arc-consistent. This is done by iteratively revising the arcs by performing calls to Algorithm 6. At the beginning, a list called Q is filled with pairs (x_i, c) such that $x_i \in X(c)$. Pairs for which $D(x_i)$ is not ensured to be arc-consistent are kept in order to avoid useless calls to Algorithm 7. This is a main advantage of AC3 with respect to its predecessor Algorithms AC1 and AC2. Then, at line 2, a while statement controls the calls to `Revise3`. If a true value is received from `Revise3`, $D(X_i)$ is verified and if no value remains within the domain, the algorithm returns false. If there still exist values in $D(X_i)$, normally, a value for another variable x_j has lost its support on c. Hence, the list Q must be refilled with all pairs $(x_i; c)$. The process finishes and returns true when all remaining values within domains are arc-consistent with respect to all constraints.

Example 9 (enforcing AC3 on a Sudoku puzzle). Let us exemplify the work of the prefiltering phase by enforcing the AC3 on three constraints of a Sudoku instance. We begin by enforcing the AC3 on a given subgrid (enclosed with dashed lines in Figure 2) of the Sudoku puzzle. The subgrid has three variables with no value assigned ($x_{4,9}$, $x_{5,8}$, and $x_{6,9}$). Then, enforcing AC3 via the GAC3 algorithm with respect to the subgrid constraint (second constraint from Example 6) leads to the elimination of six values from $D(x_{4,9})$, $D(x_{5,8})$, and $D(x_{6,9})$. Those values have no support on the verified constraint; that is, they have been already taken for another cell on the same subgrid. Thus, the original domains for the three variables are reduced to $\{5, 6, 8\}$.

In Figure 3, the AC3 is enforced to a row of the puzzle. This row has four variables ($x_{5,2}$, $x_{5,4}$, $x_{5,6}$, and $x_{5,8}$) with no assigned value. The GAC3 algorithm filters from domains five values with no support from $D(x_{5,2})$, $D(x_{5,4})$, and $D(x_{5,6})$. The variable $x_{5,8}$ has been refiltered, remaining only two

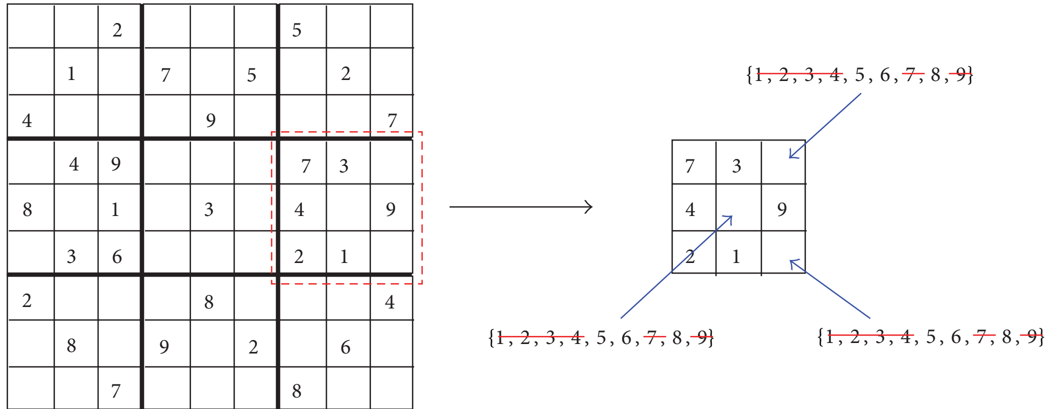

Figure 2: Enforcing AC3 to a subgrid constraint.

Input: x_i, c
Output: $CHANGE$
(1) $CHANGE \leftarrow$ false
(2) **Foreach** $v_i \in D(x_i)$ **do**
(3) **If** $\nexists \tau \in c \cap \pi_{X(c)}(D)$ with $\tau[x_i] = v_i$ **do**
(4) remove v_i from $D(x_i)$
(5) $CHANGE \leftarrow$ true
(6) **End If**
(7) **End Foreach**
(8) **Return** $CHANGE$

Algorithm 6: Revise3.

Input: X, D, C
Output: `Boolean`
(1) $Q \leftarrow (x_i, c) \mid c \in C, x_i \in X(c)$
(2) **While** $Q \neq \emptyset$ **do**
(3) select and remove (x_i, c) from Q
(4) **If** `Revise3`(x_i, c) **then**
(5) If $D(x_i) = \emptyset$ **then**
(6) **Return** false
(7) **Else**
(8) $Q \leftarrow Q \cup \{(x_j, c') \mid c' \in C \wedge c' \neq c \wedge x_i, x_j \in X(c') \wedge j \neq i\}$
(9) **End If**
(10) **End If**
(11) **End While**
(12) **Return** true

Algorithm 7: AC3/GAC3.

possible values. Finally, in Figure 4, four values are filtered from four variables, remaining only one possible value for variable $x_{5,8}$.

6. The Prefiltered Cuckoo Search via Geometric Operators

The cuckoo search proposed here combines geometric operators with prefiltering phases. The goal is to enhance the performance of the cuckoo search as well as to allow it to correctly explore a discrete search space. Algorithm 5 illustrates the new hybrid algorithm. Now the input set is quite larger. It considers the size of the nest, the constraint network $\langle X, D, C\rangle$ representing the Sudoku problem, and two parameters that define probabilities for regulating the usage of geometric operators. As output, the procedure returns the best egg reached by the algorithm. The prefiltering phase via the AC3 algorithm is triggered at the beginning. The AC3 algorithm receives as input the constraint network $\langle X, D, C\rangle$ of the Sudoku and reduces if possible the

```
Input: Nest_size, X, D, C, P_pmx-multi, P_mutate
Output: Egg_best
(1)   D ← AC3(X, D, C)
(2)   Nests ← GenerateInitialPopulation(Nest_size, D)
(3)   While ¬ StopCondition do
(4)       Egg_i ← GetRandomlyFrom(Nests)
(5)       Egg_i ← GeometricOperators(P_pmx-multi, P_mutate)
(6)       Egg_j ← ChooseRamdomlyFrom(Nests)
(7)       If cost(Egg_i) ≤ cost(Egg_j)
(8)           Egg_j ← Egg_i
(9)       End If
(10)      Egg_best ← FindCurrentBest(Nests)
(11)      Nests ← AbandonWorseNests(p_a, Nests)
(12)      Nests ← BuildNewSolutions(Egg_best, D)
(13)  End While
```

Algorithm 8: Prefiltered discrete cuckoo search.

```
Input: Egg, Egg_best, Egg_lastbest, P_pmx-multi, P_mutate
Output: Egg
(1)   Foreach Egg ∈ Nests
(2)       If (Egg ⟨⟩ Egg_best)
(3)           If (rand() < P_pmx-multi)
(4)               Egg ← PMXCrossover(Egg, Egg_best)
(5)           Else
(6)               Egg = MultiParentalSortingCrossover(Egg, Egg_best, Egg_lastbest)
(7)           End If
(8)           If (rand() < P_mutate)
(9)               Egg ← FeasibleGeometricMutation(D, Egg)
(10)          End If
(11)      End If
(12)  End Foreach
```

Algorithm 9: Geometric operators.

set of domains D by deleting the unfeasible values. Then, an initial population of nests is generated but, unlike the classic cuckoo, the generation is bounded to the reduced set of domains D. Between lines 3 and 13, a while loop manages the iteration process until the stop condition is reached, which for the current implementation corresponds to a maximum number of iterations. At line 4, an Egg_i is randomly chosen from nests to which the geometric operators are then applied. The usage of geometric operators is illustrated in Algorithm 9, where they apply only whether the evaluated egg is not the best one. The PMX and multi-parent sorting crossovers act depending on a random value and on the $P_{pmx\text{-}multi}$ probability. Analogously, the mutation operates using the P_{mutate} probability, and D is used as input of the mutation to validate that only feasible mutations are carried out. At line 6, an Egg_j is randomly chosen to be then compared with the previous one in order to keep the one exhibiting the best cost. The cost of a solution corresponds to the sum of wrong values in subgrids, columns, and rows (see Figure 5). At line 11, the worse nests are abandoned depending on probability p_a. Finally, new solutions are built, but again, the set of filtered domains D is considered in order to avoid unfeasible solutions. Algorithm 8 illustrates the prefiltered discrete cuckoo search.

7. Experimental Results

Different experiments have been performed in order to validate our approach. The Sudoku benchmarks used have been taken from [24], which are organized in three difficulty levels: easy, medium, and hard. All tested Sudokus have a unique solution. The algorithms have been implemented in Octave 3.6.3, and the experiments have been performed on a 2.0 GHz Intel Core2 Duo T5870 with 1 Gb RAM running Fedora 17. The configuration of the proposed cuckoo search is the following, which corresponds to the best one achieved after a tuning phase: $P_a = 0.25$, $P_{pmx\text{-}multi} = 0.7$, $P_{mutate} = 0.9$, and $Nest_{size} = 10$.

Table 1 illustrates the results of solving 9 problems, 3 from each difficulty level, by using the proposed cuckoo search algorithm considering 10000 iterations. From left to right, the table states the number of tries performed, the number of tries solved, the minimum solving time reached, the average

Table 1: Solving Sudokus with the prefiltered discrete cuckoo search considering 10000 iterations.

	Tries	Solved	Min. solving time (sec)	$\overline{x}$ solving time (sec)	Max. solving time (sec)	SD time (sec)
Easy a	50	50	1.302	1.310	1.329	0.004
Easy b	50	50	1.005	1.007	1.029	0.003
Easy c	50	50	1.019	1.021	1.036	0.003
Medium a	50	50	2.501	384.341	1171.312	291.336
Medium b	84	50	38.32	729.673	1932.691	516.867
Medium c	67	50	26.807	800.471	1923.442	561.877
Hard a	97	50	385.652	2059.690	3777.340	984.921
Hard b	62	50	49.608	1484.692	6810.882	1473.731
Hard c	71	50	9.497	771.211	3561.042	825.516

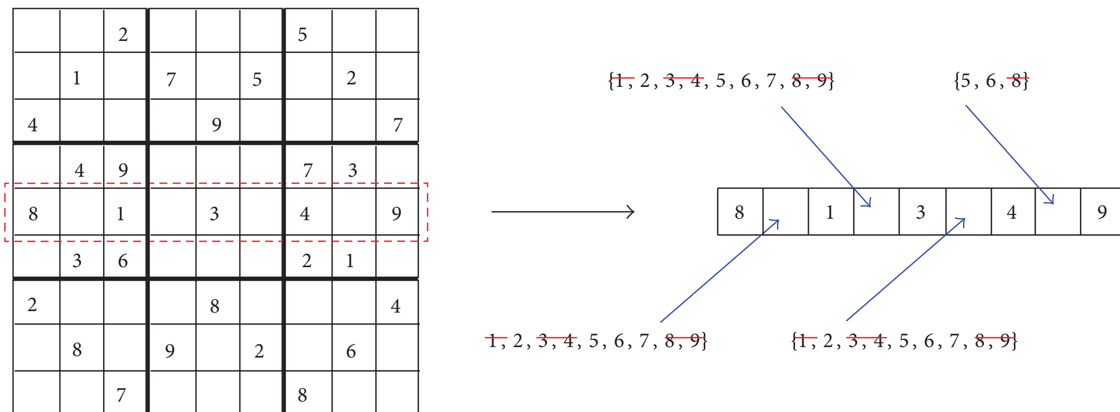

Figure 3: Enforcing AC3 to a row constraint.

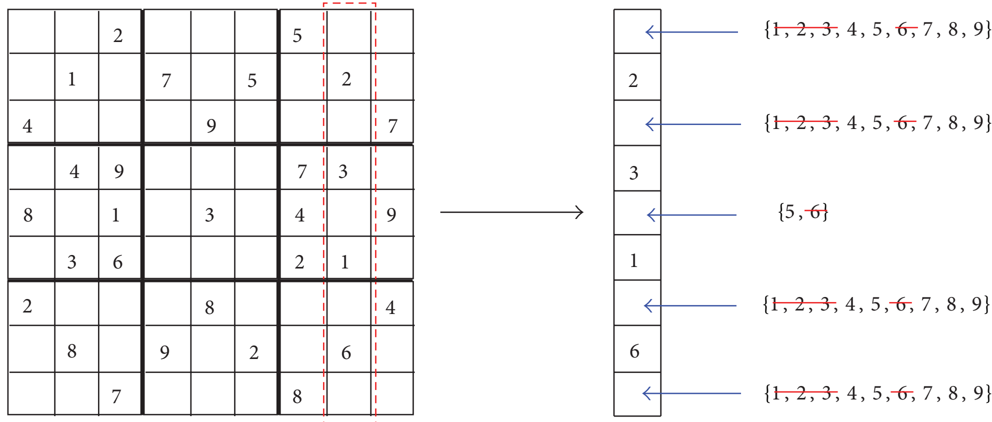

Figure 4: Enforcing AC3 to a column constraint.

solving time, the maximum solving time, and the standard deviation (SD). Regarding the easy Sudokus, the proposed cuckoo search is able to rapidly solve them reaching a 100% of success (50 out of 50 tries solved). Then, increasing one difficulty level, the percentage of success shortly diminishes, keeping the 100% of success for the "medium a" benchmark. Finally, for hard Sudokus, the runtime naturally gets bigger (see Figure 6); however, the performance is reasonable, reaching a 51% (50 out of 97 tries solved) of success for "hard a," 80% (50 out of 62 tries solved) of success for "hard b," and a 70% (50 out of 71 tries solved) of success for "hard c."

In Table 2, the performance of the proposed cuckoo search is compared with the best-performing incomplete methods reported in the literature: a genetic algorithm (GA) [1] and a hybrid tabu search (hybrid TS) [20]. We contrast the number of problems solved from a total of 30 tries

Table 2: Comparing the prefiltered discrete cuckoo search with the best-performing incomplete methods for Sudokus considering 100000 and unlimited iterations.

Problem	Prefiltered discrete CS		Hybrid TS		GA	
	Unlimited iterations	100000 iterations	Unlimited iterations	100000 iterations	Unlimited iterations	100000 iterations
Easy a	30	30	30	30	30	29
Easy b	30	30	30	30	30	30
Easy c	30	30	30	30	30	30
Medium a	30	30	30	30	30	10
Medium b	30	30	30	30	—	—
Medium c	30	30	30	30	—	—
Hard a	30	30	30	30	30	2
Hard b	30	30	30	30	—	—
Hard c	30	30	30	30	—	—

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

Cost 6

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

Cost 0

Figure 5: Solution cost of a Sudoku puzzle.

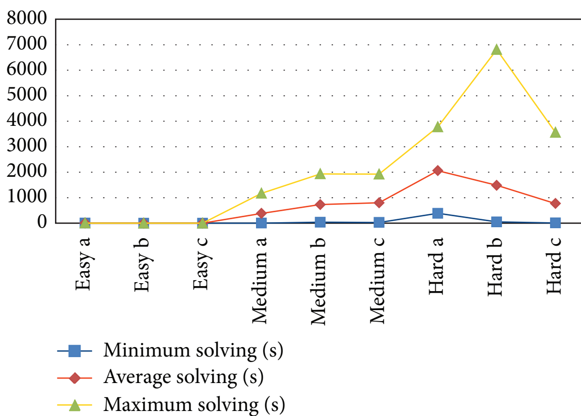

Figure 6: Comparing solving times for Sudoku.

taking into account both unlimited iterations and 100000 iterations. The results depict that the three techniques are able to easily succeed for the first Sudoku level. In the presence of medium Sudokus, the GA begins to decrease its performance, being able to solve a medium Sudoku with a 33% of success (10 out of 30 tries solved). The cuckoo search and the hybrid TS keep their 100% of success. Finally, observing hard Sudokus, the performance of the cuckoo search and the hybrid TS is considerably better than GA, which solves only 2 out of 30 tries, while our proposal as well as the hybrid TS is able to reach a 100% of success.

8. Conclusions and Future Work

In this paper, we have presented a prefiltered cuckoo search algorithm tuned with geometric operators. The geometric operators allow one to drive the search to promising regions of a discrete space of solutions, while the prefiltering phase attempts to previously delete from domains the values that do not conduct to any feasible solution. In this way, the work of the metaheuristic is alleviated leading to a faster solving process. We have performed a set of experiments in order to compare our approach with the best-performing approximate methods reported in the literature. We have considered Sudokus from different difficulty levels: easy, medium, and hard. In the presence of easy Sudokus, the proposed cuckoo search is able to reach a 100% of success.

When solving medium difficulty Sudokus, the cuckoo search keeps its 100% of success, while the best GA reported is able only to solve 10 out of 30 tries. Finally, regarding hard Sudokus, the cuckoo search noticeably competes against the best incomplete method reported for Sudokus, both reaching a 100% of success considering 100000 iterations.

We visualize different directions for future work; perhaps the clearest one is the introduction of prefiltering phases to additional metaheuristics such as particle swarm optimization, ant, or bee colony algorithms to solve Sudokus or any combinatorial problem. Evaluating the behaviour of geometric operators in other swarm-based metaheuristics is another interesting work to develop. Finally, the use of autonomous search [25–28] for the self-tuning of a metaheuristic interacting with prefiltering phases will be also an appealing research direction to pursue.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

Ricardo Soto is supported by Grant CONICYT/FONDECYT/INICIACION/11130459, Broderick Crawford is supported by Grant CONICYT/FONDECYT/REGULAR/1140897, and Fernando Paredes is supported by Grant CONICYT/FONDECYT/REGULAR/1130455.

References

[1] T. Mantere and J. Koljonen, "Solving, rating and generating sudoku puzzles with GA," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '07)*, pp. 1382–1389, IEEE Computer Society, Singapore, September 2007.

[2] F. Rossi, P. van Beek, and T. Walsh, *Handbook of Constraint Programming*, Elsevier, 2006.

[3] T. K. Moon and J. H. Gunther, "Multiple constraint satisfaction by belief propagation: an example using sudoku," in *Proceedings of the IEEE Mountain Workshop on Adaptive and Learning Systems*, pp. 122–126, Logan, Utah, USA, July 2006.

[4] H. Simonis, "Sudoku as a constraint problem," in *Proceedings of the 4th International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pp. 13–27, Barcelona, Spain, 2005.

[5] I. Lynce and J. Ouaknine, "Sudoku as a SAT problem," in *Proceedings of the International Symposium on Artificial Intelligence and Mathematics (ISAIM '06)*, Fort Lauderdale, Fla, USA, 2006.

[6] A. Moraglio, J. Togelius, and S. Lucas, "Product geometric crossover for the sudoku puzzle," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '06)*, pp. 470–476, IEEE Computer Society, Vancouver, Canada, July 2006.

[7] R. Lewis, "Metaheuristics can solve sudoku puzzles," *Journal of Heuristics*, vol. 13, no. 4, pp. 387–401, 2007.

[8] A. Moraglio and J. Togelius, "Geometric particle swarm optimization for the sudoku puzzle," in *Proceedings of the 9th Annual Genetic and Evolutionary Computation Conference (GECCO '07)*, pp. 118–125, ACM Press, July 2007.

[9] T. Mantere and J. Koljonen, "Solving and analyzing sudokus with cultural algorithms," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '08)*, pp. 4053–4060, IEEE Computer Society, Hong Kong, June 2008.

[10] M. Asif and R. Baig, "Solving NP-complete problem using ACO algorithm," in *Proceedings of the International Conference on Emerging Technologies (ICET '09)*, pp. 13–16, IEEE Computer Society, Islamabad, Pakistan, October 2009.

[11] T. K. Moon, J. H. Gunther, and J. J. Kupin, "Sinkhorn solves sudoku," *IEEE Transactions on Information Theory*, vol. 55, no. 4, pp. 1741–1746, 2009.

[12] G. Santos-García and M. Palomino, "Solving sudoku puzzles with rewriting rules," *Electronic Notes in Theoretical Computer Science*, vol. 176, no. 4, pp. 79–93, 2007.

[13] J. Gunther and T. K. Moon, "Entropy minimization for solving sudoku," *IEEE Transactions on Signal Processing*, vol. 60, no. 1, pp. 508–513, 2012.

[14] X.-S. Yang and S. Deb, "Cuckoo search via lévy flights," in *Proceedings of the World Congress on Nature and Biologically Inspired Computing (NABIC '09)*, pp. 210–214, IEEE, Coimbatore, India, December 2009.

[15] X.-S. Yang and S. Deb, "Multiobjective cuckoo search for design optimization," *Computers & Operations Research*, vol. 40, no. 6, pp. 1616–1624, 2013.

[16] P. R. Srivastava, A. Varshney, P. Nama, and X.-S. Yang, "Software test effort estimation: a model based on cuckoo search," *International Journal of Bio-Inspired Computation*, vol. 4, no. 5, pp. 278–285, 2012.

[17] M. K. Marichelvam, "An improved hybrid cuckoo search (ihcs) metaheuristics algorithm for permutation flow shop scheduling problems," *International Journal of Bio-Inspired Computation*, vol. 4, no. 4, pp. 200–205, 2012.

[18] A. Gherboudj, A. Layeb, and S. Chikhi, "Solving 0-1 knapsack problems by a discrete binary version of cuckoo search algorithm," *International Journal of Bio-Inspired Computation*, vol. 4, no. 4, pp. 229–236, 2012.

[19] B. Crawford, M. Aranda, C. Castro, and E. Monfroy, "Using constraint programming to solve sudoku puzzles," in *Proceedings of the 3rd International Conference on Convergence and Hybrid Information Technology (ICCIT '08)*, pp. 926–931, IEEE Computer Society, Busan, Republic of Korea, November 2008.

[20] R. Soto, B. Crawford, C. Galleguillos, E. Monfroy, and F. Paredes, "A hybrid ac3-tabu search algorithm for solving sudoku puzzles," *Expert Systems with Applications*, vol. 40, no. 15, pp. 5817–5821, 2013.

[21] A. K. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977.

[22] A. Mackworth, "On reading sketch maps," in *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI '77)*, pp. 598–606, 1977.

[23] C. Bessière, *Handbook of Constraint Programming*, Elsevier, 2006.

[24] T. Mantere and J. Koljonen, "Sudoku research page," 2008, http://lipas.uwasa.fi/~timan/sudoku/.

[25] B. Crawford, R. Soto, E. Monfroy, W. Palma, C. Castro, and F. Paredes, "Parameter tuning of a choice-function based hyperheuristic using Particle Swarm Optimization," *Expert Systems with Applications*, vol. 40, no. 5, pp. 1690–1695, 2013.

[26] E. Monfroy, C. Castro, B. Crawford, R. Soto, F. Paredes, and C. Figueroa, "A reactive and hybrid constraint solver," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 25, no. 1, pp. 1–22, 2013.

[27] R. Soto, B. Crawford, E. Monfroy, and V. Bustos, "Using autonomous search for generating good enumeration strategy blends in constraint programming," in *Proceedings of the 12th International Conference on Computational Science and Its Applications (ICCSA '12)*, vol. 7335 of *Lecture Notes in Computer Science*, pp. 607–617, Springer, Berlin, Germany, 2012.

[28] B. Crawford, R. Soto, C. Castro, and E. Monfroy, "Extensible cp-based autonomous search," in *Proceedings of the HCI International*, vol. 173 of *Communications in Computer and Information Science*, pp. 561–565, Springer, Berlin, Germany, 2011.