Expert Systems with Applications 40 (2013) 5817–5821

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

A hybrid AC3-tabu search algorithm for solving Sudoku puzzles

Ricardo Soto [a,b,*], Broderick Crawford [a,c], Cristian Galleguillos [a], Eric Monfroy [d], Fernando Paredes [e]

[a] Pontificia Universidad Católica de Valparaíso, Av. Brasil 2950, Valparaíso, Chile
[b] Universidad Autónoma de Chile, Av. Pedro de Valdivia 641, Santiago, Chile
[c] Universidad Finis Terrae, Av. Pedro de Valdivia 1509, Santiago, Chile
[d] CNRS, LINA, University of Nantes, 2 rue de la Houssinière, Nantes, France
[e] Escuela de Ingeniería Industrial, Universidad Diego Portales, Manuel Rodríguez Sur 415, Santiago, Chile

ARTICLE INFO

Keywords:
Metaheuristics
Tabu search
Constraint satisfaction
Sudoku

ABSTRACT

The Sudoku problem consists in filling a $n^2 \times n^2$ grid so that each column, row and each one of the $n \times n$ sub-grids contain different digits from 1 to n^2. This is a non-trivial problem, known to be NP-complete. The literature reports different incomplete search methods devoted to tackle this problem, genetic computing being the one exhibiting the best results. In this paper, we propose a new hybrid AC3-tabu search algorithm for Sudoku problems. We merge a classic tabu search procedure with an arc-consistency 3 (AC3) algorithm in order to effectively reduce the combinatorial space. The role of AC3 here is do not only acting as a single pre-processing phase, but as a fully integrated procedure that applies at every iteration of the tabu search. This integration leads to a more effective domain filtering and therefore to a faster resolution process. We illustrate experimental evaluations where our approach outperforms the best results reported by using incomplete search methods.

1. Introduction

The Sudoku puzzle consists in a board commonly of size 9×9, subdivided into sub-grids of size 3×3 including pre-filled cells with digits that cannot be changed or moved (see Fig. 1). The puzzle is solved when the board is filled so that each row, column and sub-grid contain different digits from 1 to 9. The general Sudoku problem of $n^2 \times n^2$ board size including $n \times n$ sub-grids is known to be NP-complete (Yato, Seta, & Ito, 2003). Then exact methods may solve the problem in exponential time. Sudokus are often classified in terms of difficulty. The relevance and positioning of the problem may vary its difficulty, however the number of pre-filled cells has little or no incidence. Mantere and Koljonen (2007) classify Sudokus into tree categories: easy, medium, and hard.

Various approaches have been proposed during the last years to solve Sudoku puzzles. Most works range from complete search methods such as constraint programming (Moon & Gunther, 2006; Rossi, van Beek, & Walsh, 2006; Simonis, 2005) and Boolean satisfiability (Lynce & Ouaknine, 2006) to incomplete search methods such as genetic programming (Asif, 2009; Mantere & Koljonen, 2007) and metaheuristics in general (Moraglio, Togelius, & Lucas, 2006; Lewis, 2007; Moraglio & Togelius, 2007; Mantere & Koljonen, 2008a). Other less traditional techniques in this context such as rewriting rules (Santos-García & Palomino, 2007), Sinkhorn balancing (Moon, Gunther, & Kupin, 2009) and entropy minimization (Gunther & Moon, 2012) have also been proposed to tackle this problem.

In this paper we focus on incomplete search methods. We propose a new hybrid algorithm for Sudoku puzzles by combining a classic tabu search with an arc-consistency 3 (AC3) algorithm that acts as a domain reducer. The idea is to apply the AC3 procedure as a pre-processing phase but also at each iteration of the tabu search in order to actively filter the domains. This integration clearly reduce the number of tabu search iterations speeding up the solving process. We illustrate experimental results where our approach outperforms better than the incomplete methods reported in the literature.

This paper is structured as follows. The related work is presented in Section 2 followed by the preliminaries. The new hybrid AC3-tabu search algorithm for Sudokus is described in Section 4. Experiments are illustrated in Section 5. Finally, we conclude and give some directions for future work.

2. Related work

The literature presents several techniques for solving, rating and generating Sudoku problems. Sudoku problems can definitely be solved by using brute-force algorithms, backtracking-like procedures or complete search methods in general (Crawford, Aranda, Castro, & Monfroy, 2008; Lynce & Ouaknine, 2006; Moon & Gunther, 2006; Simonis, 2005). In this paper, we focus on incomplete search methods to solve Sudoku puzzles, in particular hard instances of such a problem. In this context, different approaches

* Corresponding author at: Pontificia Universidad Católica de Valparaíso, Chile. Tel.: +56 32 2273659.
E-mail address: ricardo.soto@ucv.cl (R. Soto).

0957-4174/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.eswa.2013.05.019

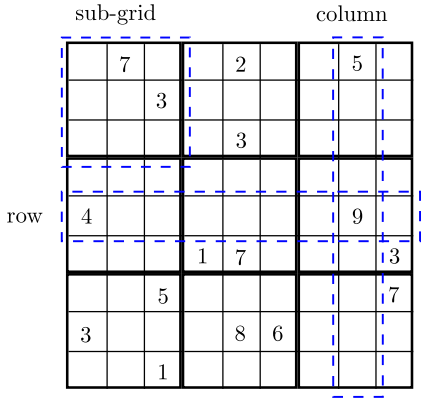

Fig. 1. Sudoku puzzle instance.

has been reported. For instance, Asif (2009) proposes an ant colony optimization algorithm for solving Sudoku problems. The author employs as heuristic information the number of digits correctly placed on the board. However, the best value reached is 76, 81 being the global optimum. In Moraglio and Togelius (2007) a geometric particle swarm optimization (GPSO) algorithm is proposed. Their goal was rather to validate the use of GPSO for non-trivial combinatorial spaces than the performance of results. Indeed they achieve a 72% of success; for 36 out of 50 tries the global optimum was reached.

Hill-climbers have also been tested to solve Sudoku puzzles (Moraglio et al., 2006), they are able to succeed for easy Sudokus failing for medium and hard ones. A genetic algorithm (GA) for Sudoku puzzles is illustrated in Moraglio et al. (2006) as well. The behaviour of geometric crossovers is studied, in particular Hamming space crossovers and swap space crossovers. Both approaches performs better than hill-climbers and mutations alone. But, they are not able to solve medium Sudokus; and only by using the swap space crossover, 15 out of 30 hard Sudokus are solved. In Mantere and Koljonen (2007) another GA approach is proposed. Here, different categories of Sudoku are solved: easy, medium, and hard. The algorithm is an extension of one devoted to solve magic square problems. Good results are exhibited for solving easy and medium Sudokus, being able to solve 2 out of 30 hard Sudokus. A similar approach using cultural algorithms is proposed in Mantere and Koljonen (2008a), but is in general superseded by the GA previously reported.

Lewis (2007) presents a simulated annealing algorithm for Sudokus. The idea is to model the puzzle as an optimization problem where the goal is to minimize the number of incorrectly placed digits on the board. However, the approach is mostly centered on creating solvable problem instances than solving hard Sudoku puzzles.

3. Preliminaries

3.1. Tabu search

Tabu search (TS), introduced by Glover, is a metaheuristic especially devoted to solve combinatorial optimization problems. It has successfully been used for tackling different kind of real-life problems as well as well-known problems from academic literature such as the travelling salesman problem, the knapsack problem, the quadratic assignment problem, or the timetabling problem.

The core idea of TS relies in employing a local search procedure that allows to iteratively move from one potential solution to another promising one until some stop criterion has been reached. This procedure is complemented with a memory structure called tabu list, which is perhaps a main feature that distinguishes TS from many incomplete methods. The goal of this memory structure is twofold: (1) to help the TS to escape from poor-scoring areas, (2) and to avoid returning to recent visited states.

Algorithm 1 depicts the classic procedure of tabu search for minimization. As input, it receives the size of the tabu list and as output it returns the best solution reached. Then, an initial solution is created, which is commonly chosen at random. At line 3, a while loop manages the iterations of the process until a given stop condition is met. Some examples of stop condition are a number of iterations limit or a threshold on the solution cost. At line 7, the neighboring solutions are added to the candidate list only if they do not contain elements on the tabu list. Then, a potential best candidate is selected, which commonly corresponds to the best quality solution according to the cost. At line 11, the cost of the selected candidate is evaluated. If it is better than the one of S_{best}, its features are added to the tabu list and the candidate becomes the new S_{best}. Finally, some elements are allowed to expire from the tabu list, generally in the same order they were added.

Algorithm 1 – Tabu search

```
Input: TabuList_size
Output: S_best
1  S_best ← ConstructInitialSolution ()
2  TabuList ← 0
3  While ¬ StopCondition do
4    CandidateList ← 0
5    For (S_candidate ∈ S_best_neighboorhood ) do
6      If ¬ ContainsAnyFeatures(S_candidate,TabuList)
7        CandidateList← S_candidate + CandidateList
8      End If
9    End For
10   S_candidate ← LocateBestCandidate (CandidateList)
11   If cost(S_candidate) ⩽cost(S_best)
12       TabuList ← FeatureDifferences(S_candidate, S_best)
13       S_best ← S_candidate
14       While TabuList > TabuList_size do
15         ExpireFeature(TabuList)
16       End While
17     End If
18   End While
19   Return S_best
```

3.2. Arc consistency

As illustrated by Simonis (2005), Sudokus can be represented as constraint networks and as a consequence techniques from constraint satisfaction can be applied over them. Arc-consistency is one of the most used filtering techniques in constraint satisfaction for reducing the combinatorial space of problems. Arc-consistency is formally defined as a local consistency within the constraint programming field (Rossi et al., 2006). A local consistency defines properties that the constraint problem must satisfy after constraint propagation. Constraint propagation is simply the process when the given local consistency is enforced to the problem. In the following, some necessary definitions are stated (Bessière, 2006).

Definition 1 (*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 values (or 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 2 (*Constraint network*). A constraint network also known as constraint satisfaction problem (CSP) is defined by a triple $N = \langle X, D, C\rangle$, where:

- X is a finite sequence of integer variables $X = (x_1, \ldots, x_n)$.
- 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.
- C is a set of constraints $C = \{c_1, \ldots, c_e\}$, where variables in $X(c_j)$ are in X.

Definition 3 (*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.

As previously mentioned, arc-consistency is one of the most used ways of propagating constraints. Arc-consistency was initially defined for binary constraint (Mackworth, 1977a, 1977b), i.e constraints involving two variables. We here give the more general definition for non-arbitrary constraints named generalized arc-consistency (GAC).

Definition 4 (*(Generalized) arc consistency*). Given a network $N = \langle X, D, C\rangle$, a constraint $c \in C$, and a variable $x_i \in X(c)$,

- A value $v_i \in D(x_i)$ is consistent with $c \in D$ iff there exists a valid tuple τ satisfying c such that $v_i = \tau[\{x_i\}]$. Such a tuple is called a support for (x_i, v_i) on c.
- The domain D is (generalized) arc consistent on c for x_i iff all the values in $D(x_i)$ are consistent with c in D, i.e., $D(x_i) \subseteq \pi_{x_i}(c \cap \pi_{X(c)}(D))$.
- The network N is (generalized) arc consistent iff D is (generalized) arc consistent for all variables in X on all constraints in C.

As an example let us consider the non arc-consistent network N depicted on left side of Fig. 2. It considers three variables x_1, x_2 and x_3; and domains $D(x_1) = D(x_2) = D(x_3) = \{0, 1, 2\}$, and constraints $c_{12} : (x_1 < x_2)$ and $c_{23} : (x2 = x3)$. Enforcing arc-consistency allows one to eliminate some inconsistent values. For instance, when constraint c_{12} is verified, the value 2 from $D(x_1)$ is removed since there is no value greater that it in $D(x_2)$. The value 0 from $D(x_2)$ is also removed since no support for it exists in $D(x_1)$. Removing 0 from $D(x_2)$ leads to the removal of 0 from $D(x_3)$ when c_{23} is checked. The resulting arc-consistent network is depicted on the right side of Fig. 2.

Algorithm 2 – Revise3

Input: x_i, c
Output: *CHANGE*
1 *CHANGE* ← 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* ← true
6 **End If**
7 **End Foreach**
8 **Return** *CHANGE*

Such a filtering process can be carried out by using Algorithms 2 and 3. As previously illustrated, the main idea of this process is the revision of arcs, i.e., to eliminate every value in $D(X_i)$ that is inconsistent with a given constraint c. This notion is encapsulated in the function `Revise3`. This function 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

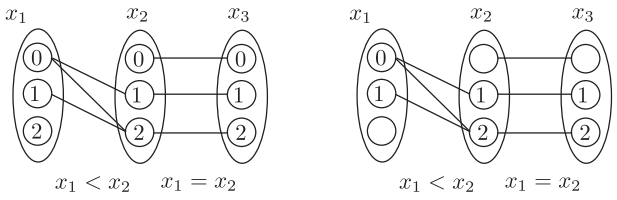

Fig. 2. Enforcing arc-consistency.

constraint c (line 3). If support does not exist, the value v_i is eliminated from $D(x_i)$. Finally, the function informs if $D(X_i)$ has been changed by returning true, or false otherwise (line 8).

The Algorithm 3 is responsible for ensuring that every domain is consistent with the set of constraints. This is done by using a loop that verifies arcs until no change happens. The function begins by filling a list Q with pairs (x_i, c) such that $x_i \in X(c)$. The idea is to keep the pairs for which $D(x_i)$ is not ensured to be arc-consistent w.r.t c. This allows to avoid useless calls to `Revise3` as done in more basic algorithms such as AC1 and AC2. Then, a loop takes the pairs (x_i, c) from Q (line 2) and `Revise3` is called (line 4). If `Revise3` return true, $D(X_i)$ is checked whether it is an emptyset. If so, the algorithm returns false. Otherwise, normally, a value for another variable x_j has lost its support on c. Thus, all pairs (x_i, c) such that $x_i \in X(c)$ must be reinserted in the list Q. The algorithm ends once Q is empty, and it returns true when all arcs have been verified and remaining values of domains are arc-consistency w.r.t all constraints.

Algorithm 3 – AC3/GAC3

Input: X, D, C
Output: `Boolean`
1 $\{Q \leftarrow (x_i, c) | c \in C, x_i \in X(c)\}$
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 $Q \leftarrow Q \cup \{(x_j, c') | c' \in C \wedge c' \neq c \wedge x_i, x_j \in X(c') \wedge j \neq i\}$
9 **End If**
10 **End If**
11 **End While**
12 **Return** true

4. The hybrid tabu search

The hybrid AC3-tabu search proposed, merges a classic tabu search algorithm with an AC3 filtering procedure in order to remove in advance the values that do not lead to any solution. The idea is to reduce the number of iterations needed to reach a solution and as a consequence to accelerate the solving process. The AC3 procedure previously introduced acts doubly: as a pre-processing phase and as a filtering component of the iteration process within the tabu search. The pre-processing phase allows to contract the combinatorial space of the initial solution, and the filtering component reduce in turn the domains of the candidate solutions.

Algorithm 4 depicts the new hybrid algorithm. As input, it receives the size of the tabu list and the Sudoku problem to be solved, which is stated as a constraint network $\langle X, D, C\rangle$ where:

- $X = (x_{11}, \ldots, x_{nm})$ is the sequence of variables, and $x_{ij} \in X$ identifies the cell placed in the ith row and jth column of the Sudoku matrix, for $i = 1, \ldots, n$ and $i = 1, \ldots, m$.

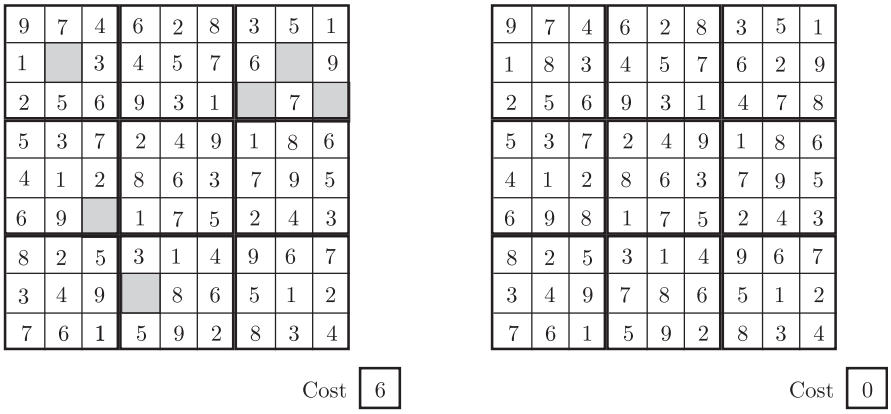

Fig. 3. Solution cost of a Sudoku puzzle.

- D is the corresponding set of domains, where $D(x_{ij}) \in D$ is the domain of the variable x_{ij}.
- C is the set of constraints defined as follows:
  - To ensure that values are different in rows and columns: $x_{k,i} \neq x_{k,j} \wedge x_{i,k} \neq x_{j,k}, \forall(k \in [1,9], i \in [1,9], j \in [i+1,9])$
  - To ensure that values are different in sub-squares: $x_{(k1-1)*3+k2,(j1-1)*3+j2} \neq x_{(k1-1)*3+k3,(j1-1)*3+j3}$, $\forall(k1, j1, k2, j2, k3, j3 \in [1,3] | k2 \neq k3 \wedge j2 \neq j3)$.

As output, the procedure returns S_{best}, which is the best solution reached by the algorithm. A solution is therefore an assignment $\{x_{11} \rightarrow a_{11}, \ldots, x_{nm} \rightarrow a_{nm}\}$ such that $a_{ij} \in D(x_{ij})$ for $i = 1, \ldots, n$ and $i = 1, \ldots, m$. The pre-processing phase is triggered at the first line of the procedure. The AC3 algorithm receives as input the constraint network $\langle X, D, C \rangle$ of the Sudoku and returns a new initial solution S_{best}. It also updates the set of domains D by deleting the unfeasible values. At line 2, the tabu list is initialized. Then, a while loop controls the iterations of the process until the stop condition is reached, which in this case is a given number of iterations. At line 4, the candidate list is set to 0. Then a for loop manages the candidate generation, which is carried out by a roulette-wheel based procedure (Xhafa, Carretero, Dorronsoro, & Alba, 2009; Razali & Geraghty, 2011). Next, the best candidate is located, which corresponds to the matrix that has less empty cells. This candidate is then submitted to a new AC3 phase in order to filter its set of domains $D_{S_{candidate}}$. Let us note that constraint propagation via AC3 is also able in several cases to reduce a domain $D(x_{ij})$ to a unique possible value. This allows one to fill the corresponding cell x_{ij} with the only remaining value and as a consequence to improve the quality of $S_{candidate}$. At line 10, the cost of the $S_{candidate}$ is compared to the best reached solution. If the cost of $S_{candidate}$ is better than the one of S_{best}, $S_{candidate}$ becomes the new S_{best}. The cost of a solution corresponds to the sum of empty cells of the matrix as shown in Fig. 3. Clearly, a solution with cost zero is optimal. Finally, the tabu list is updated according to a classic tabu search procedure.

Algorithm 4. Hybrid AC3-tabu search

Input: $TabuList_{size}, X, D, C$
Output: S_{best}
1 $S_{best}, D \leftarrow$ `AC3(`X, D, C`)`
2 `tabuList` $\leftarrow 0$
3 **While** $\neg$ `StopCondition` **do**
4 `CandidateList` $\leftarrow 0$
5 **For** ($S_{candidate} \in S_{best_{neighboorhood}}$) **do**
6 `CandidateList`$\leftarrow$ `CandidateGenerator ()`
7 **End For**
8 $S_{candidate} \leftarrow$ `LocateBestCandidate (CandidateList)`
9 $S_{candidate}, D \leftarrow$ `AC3(`$X, D_{S_{candidate}}, C$`)`
10 **If** `cost(`$S_{candidate}$`)` $\leqslant$ `cost(`S_{best}`)`
11 $TabuList \leftarrow$ `FeatureDifferences(`$S_{candidate}, S_{best}$`)`
12 $S_{best} \leftarrow S_{candidate}$
13 **While** $TabuList > TabuList_{size}$ **do**
14 `ExpireFeature(`$TabuList$`)`
15 **End While**
16 **End If**
17 **End While**
18 **Return** S_{best}

Table 1
Solving Sudokus with the hybrid TS considering 1000 iterations.

Problem	Tries	Solved	Min. solving time	$\bar{x}$ Solving time	Max. solving time
Easy a	30	30	1.324	1.331	1.354
Easy b	30	30	1.026	1.032	1.049
Easy c	30	30	1.046	1.067	1.113
Medium a	30	30	2.296	5.170	13.389
Medium b	31	30	5.151	69.807	245.346
Medium c	33	30	3.687	78.042	252.876
Hard a	35	30	2.608	88.550	259.610
Hard b	49	30	3.875	112.667	309.799

5. Experiments

In this section we provide a performance evaluation of the proposed hybrid AC3-Tabu search for Sudokus. 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 benchmarks tested have been taken from (Mantere & Koljonen, 2008b), which are classified in three levels of complexity: easy, medium, and hard. All Sudokus have a unique solution.

Table 1 illustrates the results of solving eight problems with the proposed hybrid tabu search: three easy problems, three medium problems, and two hard problems. From left to right, the table indicates the number of tries performed, the number of tries solved, the minimum solving time reached, the average solving time, and the maximum solving time. Only 1000 iterations have been considered for these experiments. The results show that the algorithm is able to rapidly solve the easy Sudokus, reaching a 100% of success (30 out of 30 tries are solved). For problems of medium

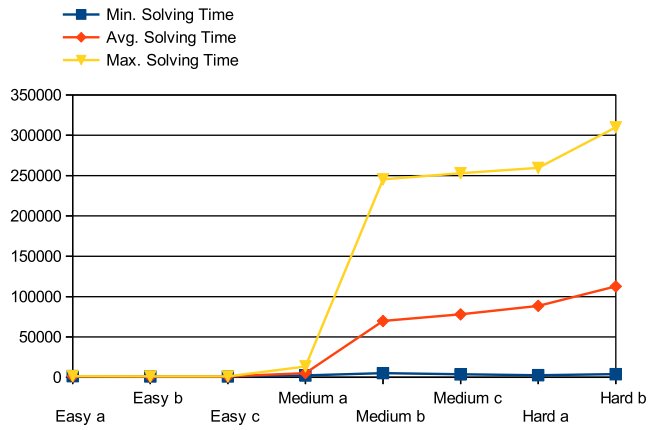

Fig. 4. Comparing solving times for Sudoku.

Table 2
Comparing the hybrid TS with the best-performing incomplete method for Sudokus considering 100.000 and unlimited iterations.

Problem	Hybrid TS		GA	
	Unlimited iterations	100.000 iterations	Unlimited iterations	100.000 iterations
Easy a	30	30	30	30
Easy b	30	30	30	30
Easy c	30	30	30	30
Medium a	30	30	30	22
Hard a	30	30	30	2

complexity, the % of success is also greatly high (30/30 for medium a, 30/31 for medium b, and 30/33 for medium c). However the runtime logically increases (see Fig. 4). In the presence of hard Sudokus, the algorithm is able to solve 30 out of 35 tries for hard b, and 30 out of 49 tries for hard b.

Table 2 compares the results of the proposed hybrid tabu search with the best-performing incomplete method reported – a genetic algorithm (GA) – for solving Sudokus (Mantere & Koljonen, 2007). We compare the amount of problems solved taking into account a total of 30 tries considering both unlimited iterations and 100.000 iterations. The results show that no difference can be seen for easy Sudokus. However, for medium Sudokus the hybrid tabu search is already better that GA, which is only able to solve 22 out of 30 tries after 100.000 iterations. Finally, for the hard a problem, the performance of the hybrid tabu search is remarkably better compared to GA, which is only capable to solve 2 out of 30 tries after 100.000 iteration, while our proposal reaches a 100% of success.

6. Conclusions

In this paper, we have proposed a new hybrid algorithm, which combines a classic tabu search procedure with AC3 in order to efficiently filter the combinatorial space. The AC3 applies both at the initial solution generation and within the iteration process. This allows one to a priori filter from domains values that do not lead to any solution, and consequently to speed up the resolution process. The experiments carried out exhibited promising results where easy, medium, and hard Sudokus are solved. Indeed, in the presence of hard Sudokus, the hybrid tabu search clearly outperforms the best-performing incomplete method from the literature. Such an algorithm is able to solve 2 out of 30 tries, while the hybrid tabu search reach a 100% of success solving 30 out of 30 tries, after 100.000 iterations.

A clear direction for future work could be the hybridization of AC3 with additional metaheuristics such as simulated annealing, particle swarm optimization, or ant colony optimization to solve Sudokus or any combinatorial problem. The integration of the autonomous search component described in Crawford et al. (2013) to the presented hybrid algorithm will be an interesting research direction to follow as well.

Acknowledgements

The author Fernando Paredes is supported by FONDECYT-Chile Grant 1130455.

References

Asif, M. (2009). Solving NP-complete problem using ACO algorithm. In *International conference on emerging technologies* (pp. 13–16). IEEE Computer Society.
Bessière, C. (2006). *Handbook of constraint programming*. Elsevier [Chap. Constraint propagation].
Crawford, B., Aranda, M., Castro, C., & Monfroy, E. (2008). Using constraint programming to solve Sudoku puzzles. In *Proceedings of the third international conference on convergence and hybrid information technology (ICCIT)* (pp. 926–931). IEEE Computer Society.
Crawford, B., Soto, R., Monfroy, E., Palma, W., Castro, C., & Paredes, F. (2013). Parameter tuning of a choice-function based hyperheuristic using particle swarm optimization. *Expert Systems with Applications, 40*(5), 1690–1695.
Glover, F., & Laguna, M. (1997). *Tabu search*. Kluwer Academics Publishers.
Gunther, J., & Moon, T. K. (2012). Entropy minimization for solving Sudoku. *IEEE Transactions on Signal Processing, 60*(1), 508–513.
Lewis, R. (2007). Metaheuristics can solve Sudoku puzzles. *Journal of Heuristics, 13*(4), 387–401.
Lynce, I., & Ouaknine, J. (2006). Sudoku as a SAT problem. In *International symposium on artificial intelligence and mathematics (ISAIM)*.
Mackworth, A. (1977a). Consistency in networks of relations. *Artificial Intelligence, 8*(1), 99–118.
Mackworth, A. (1977). On reading sketch maps. In *Proceedings of the international joint conference on artificial intelligence (IJCAI)* (pp. 598–606).
Mantere, T., & Koljonen, J. (2007). Solving, rating and generating Sudoku puzzles with GA. In *IEEE congress on evolutionary computation* (pp. 1382–1389). IEEE Computer Society.
Mantere, T., & Koljonen, J. (2008a). Solving and analyzing Sudokus with cultural algorithms. In *IEEE congress on evolutionary computation* (pp. 4054–4061). IEEE Computer Society.
Mantere, T., & Koljonen, J. (2008). Sudoku research page (Visited 1/2013). <http://lipas.uwasa.fi/~timan/sudoku/>
Moon, T. K., & Gunther, J. H. (2006). Multiple constraint satisfaction by belief propagation: An example using Sudoku. In *IEEE mountain workshop on adaptive and learning systems* (pp. 122–126).
Moon, T. K., Gunther, J. H., & Kupin, J. J. (2009). Sinkhorn solves Sudoku. *IEEE Transactions on Information Theory, 55*(4).
Moraglio, A., & Togelius, J. (2007). Geometric particle swarm optimization for the Sudoku puzzle. In *Proceedings of the genetic and evolutionary computation conference* (pp. 118–125). ACM Press.
Moraglio, A., Togelius, J., & Lucas, S. (2006). Product geometric crossover for the Sudoku puzzle. In *IEEE congress on evolutionary computation* (pp. 470–476). IEEE Computer Society.
Razali, N. M., & Geraghty, J. (2011). Genetic algorithm performance with different selection strategies in solving TSP. In *Proceedings of the world congress on engineering 2011 (WCE)* (Vol. II, pp. 1134–1139).
Rossi, F., van Beek, P., & Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.
Santos-García, G., & Palomino, M. (2007). Solving Sudoku puzzles with rewriting rules. *Electronic Notes in Theoretical Computer Science, 176*(4), 79–93.
Simonis, H. (2005). Sudoku as a constraint problem. In *4th International workshop on modelling and reformulating constraint satisfaction problems* (pp. 13–27).
Xhafa, F., Carretero, J., Dorronsoro, B., & Alba, E. (2009). A tabu search algorithm for scheduling independent jobs in computational grids. *Computing and Informatics, 28*(2), 237–250.
Yato, T., Seta, T., & Ito, T. (2003). Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals, E86-A*(5), 1052–1060.