A hybrid AC3-tabu search algorithm for solving Sudoku puzzles

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

Keywords: Metaheuristics, Tabu search, Constraint satisfaction, Sudoku

ABSTRACT

The Sudoku problem consists of 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 combinational space. The role of AC3 here is do not only act 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 \( \times \) 9, subdivided into sub-grids of size 3 \( \times \) 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
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 perform 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 kinds 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 $S_{\text{best}}$, its features are added to the tabu list and the candidate becomes the new $S_{\text{best}}$. Finally, some elements are allowed to expire from the tabu list, generally in the same order they were added.

<table>
<thead>
<tr>
<th>Algorithm 1 – Tabu search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> TabuListsize</td>
</tr>
<tr>
<td><strong>Output:</strong> $S_{\text{best}}$</td>
</tr>
<tr>
<td>1: $S_{\text{best}} \leftarrow \text{ConstructInitialSolution}()$</td>
</tr>
<tr>
<td>2: TabuList $\leftarrow 0$</td>
</tr>
<tr>
<td>3: While $\neg$ StopCondition do</td>
</tr>
<tr>
<td>4: $\text{CandidateList} \leftarrow 0$</td>
</tr>
<tr>
<td>5: For $(S_{\text{candidate}} \in \text{CandidateList})$ do</td>
</tr>
<tr>
<td>6: If $\neg$ ContainsAnyFeatures($S_{\text{candidate}}$, TabuList)</td>
</tr>
<tr>
<td>7: $\text{CandidateList} \leftarrow S_{\text{candidate}} + \text{CandidateList}$</td>
</tr>
<tr>
<td>8: End If</td>
</tr>
<tr>
<td>9: End For</td>
</tr>
<tr>
<td>10: $S_{\text{candidate}} \leftarrow \text{LocateBestCandidate(CandidateList)}$</td>
</tr>
<tr>
<td>11: If $\text{cost}(S_{\text{candidate}}) \leq \text{cost}(S_{\text{best}})$</td>
</tr>
<tr>
<td>12: TabuList $\leftarrow \text{FeatureDifferences}(S_{\text{candidate}}, S_{\text{best}})$</td>
</tr>
<tr>
<td>13: $S_{\text{best}} \leftarrow S_{\text{candidate}}$</td>
</tr>
<tr>
<td>14: While TabuList $&gt; \text{TabuListsize}$ do</td>
</tr>
<tr>
<td>15: ExpireFeature(TabuList)</td>
</tr>
<tr>
<td>16: End While</td>
</tr>
<tr>
<td>17: End If</td>
</tr>
<tr>
<td>18: End While</td>
</tr>
<tr>
<td>19: Return $S_{\text{best}}$</td>
</tr>
</tbody>
</table>

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 $X(c) = (x_1, \ldots, x_{\text{nvar}})$, called the scheme of $c$. $c$ is the subset of $\mathcal{Z}^{X(c)}$ that contains the combinations of values (or tuples) $\tau \in \mathcal{Z}^{X(c)}$ 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 $c(x_1, \ldots, x_k)$.\n
![Fig. 1. Sudoku puzzle instance.](image-url)
Definition 2 (Constraint network). A constraint network also known as constraint satisfaction problem (CSP) is defined by a triple $N = (X, D, C)$, 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 $D(x_i) \subseteq \mathbb{Z}$ is the finite set of values that variable $x_i$ can take.
- $C$ is a set of constraints $C = \{c_1, \ldots, c_m\}$, where variables in $X(c_i)$ 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 name generalized arc-consistency (GAC).

Definition 4 ((Generalized) arc consistency). Given a network $N = (X, D, C)$, 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 $\tau$ 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., $D(x_i) \subseteq \pi_{x_i}(c \cap \pi_{x_i}(D))$.
- 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 $c_{23} : (x_2 = x_3)$. 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 \quad If $\exists \tau \in c \cap \pi_{x_i}(D)\text{with} |\tau| = v_i$ do
4 \quad \quad remove $v_i$ from $D(x_i)$
5 \quad CHANGE ← true
6 \quad 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 $\tau \in c \cap \pi_{x_i}(D)$, 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 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 \quad $Q = \{(x_i, c)|c \in C, x_i \in X(c)\}$
2 While $Q \neq \emptyset$ do
3 \quad select and remove $(x_i, c)$ from $Q$
4 \quad If Revise3$(x_i, c)$ then
5 \quad \quad If $D(x_j) = \emptyset$ then
6 \quad \quad \quad Return false
7 \quad \quad Else
8 \quad \quad \quad $Q \leftarrow Q \cup \{(x_j, c')|c' \in C \land c' \neq c \land x_j \in X(c') \land j \neq i\}$
9 \quad \quad End If
10 \quad 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 $(X, D, C)$ where:

- $X = (x_1, \ldots, x_{nm})$ is the sequence of variables, and $x_i \in X$ identifies the cell placed in the $i$th row and $j$th column of the Sudoku matrix, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. 
Fig. 3. Solution cost of a Sudoku puzzle.

Algorithm 4. Hybrid AC3-tabu search

1. \( S_{\text{best}} \leftarrow \text{AC3}(X, D, C) \)
2. \( \text{tabuList} \leftarrow \emptyset \)
3. \( \text{While} \ S_{\text{best}} > \) StopCondition \( \text{do} \)
4. \( \text{CandidateList} \leftarrow 0 \)
5. \( \text{For} (S_{\text{candidate}} \in S_{\text{best}}, \text{stopped}) \) \( \text{do} \)
6. \( \text{CandidateList} \leftarrow \text{CandidateGenerator} () \)
7. \( \text{End For} \)
8. \( S_{\text{candidate}} \leftarrow \text{LocateBestCandidate} (\text{CandidateList}) \)
9. \( S_{\text{best}} \leftarrow \text{AC3}(X, D_{\text{candidate}}, C) \)
10. \( \text{If} \ \text{cost}(S_{\text{candidate}}) < \text{cost}(S_{\text{best}}) \) \( \text{do} \)
11. \( \text{TabuList} \leftarrow \text{FeatureDifferences}(S_{\text{candidate}}, S_{\text{best}}) \)
12. \( S_{\text{best}} \leftarrow S_{\text{candidate}} \)
13. \( \text{While} \ \text{TabuList} > \text{TabuList}_{\text{size}} \) \( \text{do} \)
14. \( \text{ExpireFeature} (\text{TabuList}) \)
15. \( \text{End While} \)
16. \( \text{End If} \)
17. \( \text{End While} \)
18. \( \text{Return} S_{\text{best}} \)

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
forms 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


Table 2
Comparing the hybrid TS with the best-performing incomplete method for Sudokus considering 100,000 and unlimited iterations.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Hybrid TS</th>
<th>GA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unlimited iterations</td>
<td>100,000 iterations</td>
</tr>
<tr>
<td>Easy a</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Easy b</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Easy c</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Medium a</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Hard a</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>