

Efficient algorithms for singleton arc consistency

Christian Bessiere · Stéphane Cardon ·
Romuald Debruyne · Christophe Lecoutre

Published online: 2 October 2009
© Springer Science + Business Media, LLC 2009

Abstract In this paper, we propose two original and efficient approaches for enforcing singleton arc consistency. In the first one, the data structures used to enforce arc consistency are shared between all subproblems where a domain is reduced to a singleton. This new algorithm is not optimal but it requires far less space and is often more efficient in practice than the optimal algorithm SAC-Opt. In the second approach, we perform several runs of a greedy search (where at each step, arc consistency is maintained), possibly detecting the singleton arc consistency of several values in one run. It is an original illustration of applying inference (i.e., establishing singleton arc consistency) by search. Using a greedy search allows benefiting from the incrementality of arc consistency, learning relevant information from conflicts and, potentially finding solution(s) during the inference process. We present extensive experiments that show the benefit of our two approaches.

Keywords Singleton arc consistency · Filtering · Constraint propagation · Local consistency

This paper is a compilation and an extension of [5] and [18].

C. Bessiere
LIRMM–CNRS, Université de Montpellier, Montpellier, France
e-mail: bessiere@lirmm.fr

S. Cardon · C. Lecoutre (✉)
CRIL–CNRS, Université d’Artois, Lens, France
e-mail: lecoutre@cril.fr

S. Cardon
e-mail: cardon@cril.fr

R. Debruyne
LINA–CNRS, Ecole des Mines de Nantes, Nantes, France
e-mail: romuald.debruyne@emn.fr

1 Introduction

Inference and search are two categories of techniques for processing constraints [14, 17]. On the one hand, inference is used to transform a problem into an equivalent form which is either directly used to show the satisfiability or unsatisfiability of the problem, or simpler to be handled by a search algorithm. Inference aims at modifying a constraint network by employing structural methods such as variable elimination, or filtering methods based on properties such as arc consistency and path consistency. On the other hand, search is used to traverse the space delimited by the domains of all variables of the problem. For example, search can be systematic and complete by relying on breadth-first or depth-first exploration with backtracking, or stochastic and incomplete by relying on greedy exploration and randomized heuristics.

One of the most popular systematic search algorithms to solve instances of the *constraint satisfaction problem* (CSP) is called MAC [27]. MAC interleaves inference and search since at each step of a depth-first exploration with backtracking, a local consistency called arc consistency (AC) is maintained. However, since the introduction of stronger consistencies such as max-restricted path consistency (Max-RPC) [11, 13] and singleton arc consistency (SAC) [12, 13], one issue has been the practical interest of utilizing such consistencies, instead of arc consistency, before or during search.

There is a recent focus on singleton consistencies, and more particularly on singleton arc consistency, as illustrated by recent works [2, 4, 6, 12, 26]. A constraint network P is (singleton) arc-consistent if and only if each value of P is (singleton) arc-consistent. A value is singleton arc-consistent if assigning it to its variable does not produce an arc inconsistent network.

In this paper, we propose some original algorithms to establish singleton arc consistency. First, we note that to reach time optimality we have to fully exploit the incrementality of arc consistency, i.e., the fact that iteratively establishing AC on a more and more reduced search space is cheaper than repeatedly establishing it on the original problem. This requires for each value the duplication of the data structures used to enforce arc consistency. The time optimality can therefore be reached but at the cost of a prohibitive space complexity. With the algorithm SAC-SDS, we propose a good trade-off consisting, for each pair (x_i, a) composed of a variable x_i and a value a in the domain of x_i , in storing only the domains obtained by enforcing AC after the assignment of the value a to x_i . Storing these domains allows us both to know which values may no longer be singleton arc-consistent after a value removal, and not to check their singleton arc consistency from scratch. The other data structures used to enforce AC are shared. So the space required is far less important than with a duplication of these data structures. The resulting algorithm can be used on large constraint networks and shows very good time performances compared to previous SAC algorithms.

Second, we show how it is possible to establish singleton arc consistency by performing several runs of a greedy search (where at each step, arc consistency is maintained). This contrasts with the classical breadth-first search up to a depth equal to 1 performed by SAC algorithms introduced so far. We introduce two algorithms, denoted by SAC3 and SAC3-SDS: SAC3 does not record the context of runs already performed, whereas SAC3-SDS does record them in the same vein

as SAC-SDS. Although this approach has a high worst-case time complexity, it has several advantages:

- extra space requirement is limited,
- both algorithms benefit from the incrementality of arc consistency,
- using a greedy search enables learning relevant information from conflicts,
- it is possible to find solution(s) while establishing singleton arc consistency.

More precisely, the good space complexity of both algorithms allows to use them on large constraint networks. In particular, SAC3 admits the same space complexity as the underlying arc consistency algorithm. Furthermore, when a greedy search maintaining arc consistency is used, we naturally benefit from the incrementality of arc consistency since variables are instantiated in sequence on each branch. Besides, when a dead-end is encountered during a greedy search, a nogood can be recorded and/or the origin of the failure taken into account. Also, some solutions may be opportunistically found by the algorithm. Finally, if the instance contains a large under-constrained part, a very efficient practical time complexity can be expected.

The paper is organized as follows. Section 2 introduces technical background and Section 3 briefly presents AC algorithms. Section 4 provides an overview of existing SAC algorithms. The new algorithms, namely SAC-SDS, SAC3 and SAC3-SDS, are respectively developed in Sections 5, 6 and 7. An illustration of these new algorithms is proposed in Section 8. The results of a vast experimentation concerning both random and structured problems are presented in Section 9.

2 Background

A *constraint network* P consists of a finite set of n variables $X = \{x_1, \dots, x_n\}$, a set of domains $D = \{D(x_1), \dots, D(x_n)\}$, where the domain $D(x_i)$ is the finite set of at most d values that variable x_i can take, and a set of e constraints $C = \{c_1, \dots, c_e\}$. Each constraint c_k is defined by the ordered set $var(c_k)$ of at most r variables it involves, and the set $sol(c_k)$ of combinations of values satisfying it. We will note $(x_i, a) \in D$ (respectively, $(x_i, a) \notin D$) if and only if $x_i \in X$ and $a \in D(x_i)$ (respectively, $a \notin D(x_i)$). Given a set $Y \subseteq X$ of variables, an *assignment* A on Y assigns each variable $x_i \in Y$ a value in $D(x_i)$. $A[x_i]$ denotes the value assigned to x_i in A . When a subset $Y \subseteq X$ of variables are assigned and all constraints c with $var(c) \subseteq Y$ are satisfied, we say that the assignment is *locally consistent*. A *solution* to a constraint network is an assignment on X that is locally consistent. When $var(c) = (x_i, x_j)$, we use c_{ij} to denote $sol(c)$ and we say that c is *binary*.

The constraint satisfaction problem (CSP) consists in finding whether or not a given constraint network has solutions. It is NP-complete. To solve a CSP instance, a depth-first search algorithm with backtracking can be applied, where at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation. Constraint propagation algorithms usually enforce properties, such as arc consistency, that remove values that cannot occur in any solution.

Arc consistency (AC) is defined as follows. Given a value a in the domain of a variable x_i and a constraint c_k involving x_i , a *support* for (x_i, a) on c_k is an assignment A on the variables in $var(c_k)$ such that $A[x_i] = a$ and A satisfies c_k . If there exists a

constraint on which (x_i, a) has no support, (x_i, a) is said *arc-inconsistent*; otherwise, (x_i, a) is *arc-consistent*. A constraint network $P = (X, D, C)$ is *arc-consistent* if and only if D does not contain any arc-inconsistent value. $AC(P)$ denotes the network where all arc-inconsistent values have been removed from P . If the domain of a variable becomes empty when enforcing arc consistency, we call this a *domain wipe-out*. It is denoted by $AC(P) = \perp$ and we say that P is *arc-inconsistent*.

Singleton arc consistency (SAC) is defined as follows. Given a constraint network $P = (X, D, C)$, a value a in $D(x_i)$ for some variable $x_i \in X$ is *singleton arc-consistent* if and only if the network $P|_{x_i=a} = (X, D|_{x_i=a}, C)$ is not arc-inconsistent, i.e., if and only if $AC(P|_{x_i=a}) \neq \perp$, where $D|_{x_i=a}$ is obtained by replacing $D(x_i)$ by the singleton set $\{a\}$. If $P|_{x_i=a}$ is arc-inconsistent, we say that (x_i, a) is *singleton arc-inconsistent*. Checking whether a value is singleton arc-consistent is called a *singleton check*. A constraint network $P = (X, D, C)$ is *singleton arc-consistent* if and only if D does not contain any singleton arc-inconsistent value. $SAC(P)$ denotes the network where all singleton arc-inconsistent values have been removed from P . If there is a variable with an empty domain in $SAC(P)$, we say that P is *singleton arc-inconsistent*. Given an assignment A on a subset of X , $P|_A$ is the constraint network obtained from P by restricting the domain of each variable x_i involved in A to the singleton $\{A[x_i]\}$.

For any two constraint networks P and P' , we note $P' \preceq P$ if and only if $P = (X, D, C)$, $P' = (X, D', C)$ and $D' \subseteq D$. Here, $D' \subseteq D$ means that for any variable $x_i \in X$, we have $D'(x_i) \subseteq D(x_i)$. Informally, we have $P' \preceq P$ if P' can be obtained from P just by removing values from domains.

We can observe that all the definitions given in this section apply to binary or non-binary constraints indifferently. Hence, what is often called generalized arc consistency (GAC) in the literature, is referred here as arc consistency. In the rest of the paper, when no restriction is specified, the results hold for any constraint arity. When we focus on binary constraints, we consider that the network is connected (i.e., $n - 1 \leq e$) and normalized [1], that is, at most one constraint exists between two given variables (i.e., $e < n^2$). These restrictions will be assumed when giving complexities for binary constraint networks.

3 Enforcing arc consistency

Many general-purpose algorithms have been proposed so far to enforce arc consistency. They are classically classified as coarse-grained and fine-grained arc consistency algorithms. Propagation is guided by events concerning variables in coarse-grained algorithms whereas propagation is guided by deleted values in fine-grained ones. Examples of coarse-grained algorithms are (G)AC3 [21, 22], (G)AC2001 [7, 8, 30] and (G)AC3^m [19] whereas examples of fine-grained algorithms are (G)AC4 [23, 24] and AC6 [3]. AC2001, AC4 and AC6 are time-optimal algorithms for binary constraints.

To enforce SAC, we need to use an arc consistency algorithm since SAC is built on top of AC. However, in this paper we do not want to attach the presentation of SAC algorithms to a particular underlying arc consistency algorithm because we can adopt any generic AC algorithm or even any kind of propagators dedicated to specific constraints. So, to enforce AC in our algorithms, we will just call a function `propagateAC` without precisely describing its content.

The function `propagateAC`, Algorithm 1, is called with three parameters D , Q and UPD (plus an optional one). D is the domain of the (sub)problem that must be made AC. Q is a *variable-based* propagation list, that is, it contains the variables x_j for which the domain $D(x_j)$ has changed. Such a type of propagation list is well adapted to coarse-grained algorithms. Fine-grained algorithms, such as AC6 require the complete list of values (x_j, b) removed from $D(x_j)$. But this is essentially a matter of presentation.

Algorithm 1: The underlying AC algorithm

```

function propagateAC(inout  $D$ ; in  $Q$ ; in  $UPD$ ; inout  $Deleted$ ): Boolean
    //  $Deleted$  : optional parameter
    if  $UPD$  then
1   | propagate AC on  $D$  from the set  $Q$  of modified variables and update the data
    | structure of the AC algorithm
2 else
3   | propagate AC on  $D$  from the set  $Q$  of modified variables and do not update
    | the data structure of the AC algorithm
4 if  $\exists x_i \mid D(x_i) = \emptyset$  then
    |   return false
5 if optional parameter  $Deleted$  is present then
    |    $Deleted \leftarrow Deleted \cup \{\text{all values pruned from } D\}$ 
6 return true
    
```

In our presentation, we are interested in the data structure employed by the underlying AC algorithm. For example, this data structure corresponds to the *Last* pointers for AC2001 and to residual supports for AC3^{rm}. Because one priority is to save space, for all SAC algorithms presented later, the data structure of the underlying AC algorithm is not duplicated. However, when calling `propagateAC`, it must be decided whether this data structure must be updated or not. Hence the function `propagateAC` performs two types of propagation depending on whether its Boolean parameter UPD is set to true or not. When UPD is true, `propagateAC` uses and *updates* the data structure to propagate deletions in D . When UPD is false, `propagateAC` uses the data structure to propagate deletions in D but does *not* modify it. When enforcing SAC, we distinguish the main problem P from subproblems $P|_{x_i=a}$ corresponding to singleton checks of values (x_i, a) . Typically, when AC is applied on P , UPD is true and when AC is applied on a subproblem, UPD is false (however, we shall see that for greedy runs, this is different). Consequently, in the case of an AC2001 implementation, subproblems can benefit from the knowledge of *Last* supports computed in the main problem P , but not from computations already performed inside the subproblem itself.

Function `propagateAC` has another parameter, $Deleted$, which is optional. That extra parameter $Deleted$ is used when `propagateAC` is called to enforce AC in the main problem. In that case, in addition to enforcing AC, `propagateAC` stores in $Deleted$ the values removed from D during the propagation. As we shall see, those removals performed in the main problem can directly be exploited in the subproblems.

When analyzing complexities, we will have the optimal algorithm AC2001 in mind. Of course, if a non-optimal AC algorithm is used in `propagateAC`, the global time complexity of SAC algorithms can be higher than our claims.

4 Overview of SAC algorithms

Table 1 shows the complexities of existing SAC algorithms that we briefly describe in this section. In the rest of this paper, we will denote by T and S the time and space complexities of the AC algorithm used to enforce arc consistency inside a SAC algorithm.

The first algorithm that has been proposed to establish singleton arc consistency is called SAC1 [12]. Requiring no specific data structure, its space complexity is the same as the underlying arc consistency algorithm. The counterpart of having no data structures to store the reason why a value was found singleton arc-consistent is that after every value removal, SAC1 must check again the singleton arc consistency of all remaining values. This leads to a worst case time complexity in $O(n^2d^2T)$, where T is the cost of enforcing arc consistency on the network with the underlying arc consistency algorithm. On binary constraints, this gives a time complexity in $O(en^2d^4)$ if the arc consistency algorithm used is optimal.

The second algorithm, SAC2 [2], uses the fact that if $AC(P|_{x_i=a}) \neq \perp$ then the singleton arc consistency of (x_i, a) holds as long as all the values in $AC(P|_{x_i=a})$ are in the domain. For each value (x_j, b) , SAC2 records the set $Support[j, b]$ of values (x_i, a) such that $(x_j, b) \in AC(P|_{x_i=a})$. This algorithm avoids useless singleton checks because the singleton arc consistency of a value (x_i, a) has to be checked after the removal of a value (x_j, b) only if (x_i, a) belongs to $Support[j, b]$. As shown by the experiments in [2], SAC2 is faster than SAC1. However, SAC2 has the same worst-case time complexity as SAC1 because whenever the singleton arc consistency of a value (x_i, a) must be checked again, one has to perform the arc consistency enforcement on $P|_{x_i=a}$ from scratch.

To avoid doing and redoing arc consistency from scratch (potentially nd times for each subproblem $P|_{x_i=a}$), SAC-Opt [5] creates nd copies of the domains and data structures of the AC algorithm (one copy for each subproblem $P|_{x_i=a}$). Simply, whenever the singleton arc consistency of a value (x_i, a) must be checked, the dedicated domains and data structures are used. Hence, as opposed to SAC1 and SAC2, SAC-Opt benefits from the fact that arc consistency is incremental (namely, time complexity on a problem P is the same for a single call of arc consistency or

Table 1 Complexities of existing SAC algorithms

Algorithm	Non-binary CN		Binary CN	
	Time	Space	Time	Space
SAC1 [12]	$O(n^2d^2T)$	$O(S)$	$O(en^2d^4)$	$O(ed)$
SAC2 [2]	$O(n^2d^2T)$	$O(n^2d^2 + S)$	$O(en^2d^4)$	$O(n^2d^2)$
SAC-Opt [5]	$O(ndT)$	$O(nd(nd + S))$	$O(end^3)$	$O(end^2)$

T and S are the time and space complexities of the underlying (G)AC algorithm. For binary constraints, we consider an optimal AC algorithm, such as AC2001, with time in $O(ed^2)$ and space in $O(ed)$

for up to nd calls, where two consecutive calls differ only by the deletion of some values from P). This allows SAC-Opt to enforce SAC in $O(ndT)$. This has been shown to be the optimal time complexity for binary constraints when an optimal arc consistency algorithm is used (see [5, 6]). But the worst-case space complexity of SAC-Opt prevents its use on large constraint networks.

It seems difficult to reach the same time complexity as SAC-Opt with smaller space requirements because a SAC algorithm has to maintain AC on nd subproblems $P|_{x_i=a}$. To guarantee optimal total time on such subproblems in spite of the nd possible restrictions we need to use an optimal time AC algorithm. Since there does not exist any optimal AC algorithm requiring less than $O(erd)$ space, we obtain $nd \cdot erd$ for the whole SAC algorithm (recall that r is the greatest constraint arity). In the next section, we show how we can relax time optimality to save some space and still obtain an algorithm with better time complexity than SAC1 and SAC2.

5 SAC-SDS

We propose a new algorithm that relaxes time optimality to reach a satisfactory trade-off between space and time. Like SAC-Opt, our new algorithm, SAC-SDS (*Sharing Data Structures*), duplicates the domains for each value. But unlike SAC-Opt, it does not duplicate the data structures of the AC algorithm.

For every value (x_i, a) , SAC-SDS stores a propagation list Q_{ia} and the current domain D_{ia} of the subproblem $P|_{x_i=a}$. The subproblem having D_{ia} as domain is denoted by P_{ia} . Q_{ia} contains variables whose domain has changed in P_{ia} and for which the change has not yet been propagated. It is needed to ensure optimal (G)AC propagation on each subproblem. D_{ia} is used to avoid restarting each new AC propagation phase from scratch in P_{ia} when some values have been removed from P . The data structure of the AC algorithm (*Last pointers for AC2001*) is built and updated only in the main problem P (no duplication). Nevertheless, it can be used by all subproblems P_{ia} to avoid repeating constraint checks already done in P . Parameter *UPD* of function `propagateAC` allows this distinction. (See Section 4.)

SAC-SDS (see Algorithm 2) works as follows. After some initializations (lines 1–5), SAC-SDS repeatedly pops a value (x_i, a) from *PendingList* and enforces AC in P_{ia} (lines 7–10). Note that ' $D_{ia}=nil$ ' means that this is the first enforcement of AC in P_{ia} , so D_{ia} must be initialized (line 9). Each enforcement of AC in P_{ia} starts the propagation from the variables in Q_{ia} , which are the variables whose domain has changed since the previous enforcement of AC in P_{ia} . If enforcing AC in P_{ia} leads to an empty domain in D_{ia} (line 11), (x_i, a) is singleton arc-inconsistent. It is therefore removed from D (line 12) and this deletion is propagated to P using `propagateAC` (line 13). Any value removed from P , including (x_i, a) , is put in the set *Deleted* (lines 12 and 13). This set is used by `updateSubProblems` (line 14) to remove from the subproblem the values removed from P (line 19), and to update the lists Q_{jb} and *PendingList* for further propagation in these modified subproblems (lines 18 and 20). The advantage of propagating immediately in P (line 13) the singleton arc-inconsistent values detected in line 10 is twofold. First, if a value (x_k, c) is removed in line 13 before its subdomain D_{kc} is created, D_{kc} will never be generated. Second, the subdomains D_{jb} created after the removal of (x_k, c) will benefit from this propagation because they are created by duplication of D (line

9). For each already created subproblem P_{jb} with $c \in D_{jb}(x_k)$, x_k is put in Q_{jb} and (x_j, b) is put in *PendingList* for future propagation (line 14). Because the AC algorithm embedded in SAC-SDS does not store its data structures (which prevents it from being optimal), we refer to its worst-case time complexity by T' .¹

Algorithm 2: SAC-SDS

```

function SAC-SDS(inout  $P = (X, D, C)$ ): Boolean
1 initialize the data structure of the AC algorithm used in propagateAC
2 if  $\neg$ propagateAC( $D, X, true$ ) then return false
3  $PendingList \leftarrow \{(x_i, a) \mid x_i \in X, a \in D(x_i)\}$ 
4 foreach  $(x_i, a) \in PendingList$  do
5    $D_{ia} \leftarrow nil; Q_{ia} \leftarrow \{x_i\}$ 
6 while  $PendingList \neq \emptyset$  do
7   select and remove a value  $(x_i, a)$  from  $PendingList$ 
8   if  $a \in D(x_i)$  then
9     if  $D_{ia} = nil$  then  $D_{ia} \leftarrow (D \setminus D(x_i)) \cup \{(x_i, a)\}$ 
10    if propagateAC( $D_{ia}, Q_{ia}, false$ ) then  $Q_{ia} \leftarrow \emptyset$ 
11    else
12       $D(x_i) \leftarrow D(x_i) \setminus \{a\}; Deleted \leftarrow \{(x_i, a)\}$ 
13      if propagateAC( $D, \{x_i\}, true, Deleted$ ) then
14        updateSubProblems( $Deleted$ )
15      else return false
16 return true

procedure updateSubProblems(in  $Deleted$ : set)
17 foreach  $(x_j, b) \in D \mid D_{jb} \cap Deleted \neq \emptyset$  do
18    $Q_{jb} \leftarrow Q_{jb} \cup \{x_i \in X \mid D_{jb}(x_i) \cap Deleted \neq \emptyset\}$ 
19    $D_{jb} \leftarrow D_{jb} \setminus Deleted$ 
20    $PendingList \leftarrow PendingList \cup \{(x_j, b)\}$ 

```

Theorem 1 SAC-SDS is a correct SAC algorithm with $O(ndT')$ time complexity, where T' is the cost of enforcing arc consistency on the network with an AC algorithm that does not store any special data structure between two calls. This gives $O(end^4)$ on binary constraint. The space complexity of SAC-SDS is in $O(n^2d^2 + S)$, where S is the space complexity of the AC algorithm used.

Proof We instantiate the proof in the case where propagateAC uses AC2001 to enforce arc consistency. The data structure of AC2001 is the *Last* structure which

¹For instance, standard AC2001 has $T = O(ed^2)$ optimal time complexity on binary constraint networks whatever the number of times we call it after some value deletions in the network. When we do not store its data structure between two calls, its complexity becomes $T' = O(ed^3)$.

maintains in $Last(x_i, a, c_k)$ the *smallest* support for (x_i, a) on c_k . The smallest support is defined according to any total ordering o on the tuples on $var(c_k)$. Any other AC algorithm can be used.

Soundness. Note first that the structure $Last$ is updated only when achieving AC in P so that any support for (x_i, a) is greater than or equal to $Last(x_i, a, c_k)$. The domains of the subproblems being subdomains of D , any support for a value (x_i, a) on c_k in a subproblem is also greater than or equal to $Last(x_i, a, c_k)$. This explains that $propagateAC(D_{ia}, Q_{ia}, false)$ can benefit from the structure $Last$ without losing any support. Thus, the tests of arc consistency on subproblems are sound. Suppose now that some singleton arc-consistent values are removed by SAC-SDS. Let (x_i, a) be the first singleton arc-consistent value removed. It is necessarily removed in line 12, thus P_{ia} was found arc-inconsistent in line 10. P_{ia} was initially arc-consistent because we suppose that (x_i, a) is singleton arc-consistent. Furthermore, $propagateAC(D_{ia}, Q_{ia}, false)$ is sound. So, the arc inconsistency of (x_i, a) comes from values removed from P_{ia} in line 19. Those values were removed from P in lines 12 or 13 and are singleton arc-inconsistent by our assumption that (x_i, a) is the first singleton arc-consistent value removed. So, (x_i, a) is singleton arc-inconsistent too, which contradicts our assumption. Therefore, SAC-SDS is sound.

Completeness. Completeness comes from the fact that any deletion is propagated. After initialization, $PendingList = D$ (line 3) and so, the main loop of SAC-SDS processes all subproblems $P|_{x_i=a}$ at least once. Each time a value (x_i, a) is found singleton arc-inconsistent in P because $P|_{x_i=a}$ is arc-inconsistent (line 10) or because the deletion of some singleton arc-inconsistent value makes it arc-inconsistent in P (detected by $propagateAC$ in line 13), (x_i, a) is removed from the subproblems (line 19). $PendingList$ and the local propagation lists are updated for future propagation (lines 18 and 20). At the end of the main loop, $PendingList$ is empty, so all the removals have been propagated and for any value $(x_i, a) \in D$, D_{ia} is a non empty arc-consistent subdomain of $P|_{x_i=a}$.

Complexity. Each of the nd domains D_{ia} can contain nd values so they require a space in $O(n^2d^2)$. There are at most n variables in the nd local propagation lists Q_{ia} , which requires a total space in $O(n^2d)$. As a result, the space complexity of SAC-SDS is in $O(n^2d^2 + S)$ if S is the space required by the AC algorithm used. It gives $O(n^2d^2 + erd)$ with AC2001, or $O(n^2d^2)$ with AC3. So, considering space requirements, SAC-SDS is similar to SAC2 as long as we use AC3 or if $er < n^2d$, which is true in most practical cases. On binary networks, space complexity becomes $O(n^2d^2)$ even with AC2001.

Regarding time complexity, SAC-SDS first duplicates the domains (line 9), which is in $nd \cdot nd$. Each value found singleton arc-inconsistent in line 10 is removed and this removal is propagated to all subproblems $P|_{x_i=a}$ via an update of $PendingList$, Q_{ia} and D_{ia} (lines 17–20). This update requires nd operations per removal, so $nd \cdot nd$ operations in total. Each subproblem can in the worst case be called nd times for arc consistency in line 10, and there are nd subproblems. The domain of each subproblem $P|_{x_i=a}$ is stored in D_{ia} so that the AC propagation is launched with the domain in the state in which it was at the end of the previous AC propagation in the subproblem. In addition, the variables of $P|_{x_i=a}$ that have lost values since the last call to AC are stored in Q_{ia} so that AC is incrementally propagated from the modified variables and not on the whole network from scratch. Thus, in spite of the several AC propagations on a subproblem, the propagation of all the value

removals costs $O(T')$ in a subproblem.² (Note that we cannot reach the optimal $O(T)$ time complexity for arc consistency on these subproblems because we do not duplicate the data structures necessary for AC optimality.) Therefore, the total cost of arc consistency propagations in all subproblems is in $O(nd \cdot T')$ (i.e., $O(nd^4)$ on binary constraints). \square

Like SAC2, SAC-SDS performs a less brute-force propagation than SAC1 because after the removal of a value (x_i, a) from D , SAC-SDS checks the arc consistency of the subproblems $P|_{j=b}$ only if they have (x_i, a) in their domains (and not all the subproblems as SAC1). But this is not sufficient to have a better worst-case time complexity than SAC1. SAC2 has indeed the same time complexity as SAC1. SAC-SDS improves this complexity because it stores the *current* domain of each subproblem, and so, it does not propagate in the subproblems each time from scratch. This is independent on the AC algorithm: using AC3 gives the same time complexity. However, we can expect a better average time complexity with an algorithm like AC2001 because the shared data structure reduces the number of constraint checks required, even if it does not permit to obtain optimal worst-case time complexity. Finally, in SAC1 and SAC2, each AC enforcement in a subproblem must be done on a new copy of D built at runtime (potentially $nd \cdot nd$ times) whereas such duplication is performed only once for each value in SAC-SDS (by creating subdomains D_{ia}).

6 SAC3

All algorithms previously mentioned involve performing a breadth-first search up to a depth equal to 1. Each branch (of size 1) of this search corresponds to check the singleton arc consistency of a value, and allows removing this value if an inconsistency is found (after establishing arc consistency). One alternative is to check the singleton arc consistency of a value in the continuity of previous singleton checks. In other words, we can try to build less branches of greater size using a greedy search (where at each step, arc consistency is maintained). As long as, for a current branch, no inconsistency is found, we try to extend it. When an inconsistency is found, either the branch is of size 1 and that single value is detected singleton arc-inconsistent, or all but last variable assignments correspond to singleton arc-consistent values. This last statement relies on Proposition 1.

Proposition 1 *Let $P = (X, D, C)$ be a constraint network, S be an assignment on a subset $Y \subseteq X$ of variables, and $P' = (X, D', C) = AC(P|_S)$. If $P' \neq \perp$, then for any $x_i \in X$ such that $|D'(x_i)| = 1$, the pair (x_i, a) , where a is the unique value in $D'(x_i)$, is singleton arc-consistent in P .*

Proof Arc consistency is contracting for \leq , that is, for any constraint network P , we have $AC(P) \leq P$. Hence, $AC(P|_S) \leq P|_S$, and we deduce that $P' \leq P$ because $P' = AC(P|_S)$ and $P|_S \leq P$. We also know from [1] that arc consistency is monotonic, that is, for any constraint networks P_1 and P_2 with $P_1 \leq P_2$, we have $AC(P_1) \leq$

²As an example, nd calls to the propagation phase of AC2001 on a binary network P_{ia} are in $O(ed^3)$ —the same as AC3—instead of $O(nd \cdot ed^2)$ if domains D_{ia} were re-initialized at each call or if the local propagation list Q_{ia} was not maintained.

$AC(P_2)$. Furthermore, for any $x_i \in X$ such that $|D'(x_i)| = 1$ and a is the unique value in $D'(x_i)$, we have $P'|_{x_i=a} = P'$. Since $P' \preceq P$, we know that $P'|_{x_i=a} \preceq P|_{x_i=a}$, from which we deduce that $AC(P'|_{x_i=a}) \preceq AC(P|_{x_i=a})$. By assumption we have $P' \neq \perp$ and $P' = AC(P')$. As a result, $AC(P'|_{x_i=a}) = AC(P') = P' \neq \perp$, which implies that $AC(P|_{x_i=a}) \neq \perp$. Therefore, (x_i, a) is singleton arc-consistent in P . \square

As mentioned in the proposition above, some values can be detected singleton arc-consistent while checking the singleton arc consistency of others. Proposition 1 can then be seen as a generalization of Property 2 in [10], and is also related to the exploitation of singleton-valued variables in [28].

Below, we give the description of SAC3 (see Algorithm 3), a first algorithm that uses a greedy search in order to establish singleton arc consistency. As for SAC-SDS, *PendingList* is used to store the set of values whose singleton arc consistency must be checked, and *Deleted* the set of values removed after a call to function `propagateAC`.

Algorithm 3 starts by enforcing arc consistency on the network (line 2). Then, as in SAC-SDS, all values are put in *PendingList* for checking their singleton arc consistency (line 4). In order to check the singleton arc consistency of the values in *PendingList*, successive branches are built. We start by picking a value (x_i, a) in *PendingList* (line 6) that will be the first assignment of the branch to be built. Note that a heuristic can be employed to select this value (as well as the next ones along the branch being built). To build a branch, the function `buildBranch` is called with the value (x_i, a) as parameter (line 8). Function `buildBranch` returns false if $D_{x_i=a}$ is arc-inconsistent, true otherwise. In the first case, (x_i, a) has been detected singleton arc-inconsistent. As in SAC-SDS, this value must be removed and arc consistency must be re-established (lines 9–10). We also need to set the flag *CHANGE* to true to know that some changes have occurred since the initialization of *PendingList* (line 11). When the set *PendingList* becomes empty, this flag is used to determine whether we have to test again the singleton arc consistency of all values (lines 12–14). Observe that, as opposed to SAC-SDS, SAC3 cannot update *PendingList* incrementally because it does not store the subdomains D_{ia} . These subdomains allowed SAC-SDS to know which values have to be retested for singleton arc consistency.

Function `buildBranch` works as follows. First, it saves the data structure of the AC algorithm used (line 16). This is necessary if we want to ensure optimal time complexity on each branch. Then, the assignment (x_i, a) passed as a parameter is performed (line 17). If arc consistency on the subproblem obtained leads to a wipe-out, we know that (x_i, a) is singleton arc-inconsistent and the function returns false after having restored the data structure as they were before entering the function (lines 18–20). If the assignment $x_i = a$ has not provoked a wipe-out, the function tries to continue that branch (rooted on $x_i = a$) as long as arc consistency does not lead to a wipe-out (lines 21–26). The values that are selected to continue the branch are removed from *PendingList* (line 22). This is only when a wipe-out is detected that the last value of the branch is put back in *PendingList* (line 25). All the other values of the branch are necessarily singleton arc-consistent thanks to Proposition 1. Note that we can exit the while loop of line 21 without going through the break in line 26. This happens when $PendingList \cap D = \emptyset$, that is, there is no more way of extending the current branch with values in *PendingList* because all values remaining in *PendingList* have been removed from the domain of the subproblem obtained on this branch. A new branch will have to be started by the main procedure. But in the

extreme case, the current branch is of length n , which means that a solution has been found (an explicit use of this case has been omitted in the pseudo-code presented in Algorithm 3.)

Algorithm 3: SAC3

```

function SAC3(inout  $P = (X, D, C)$ ): Boolean
1 initialize the data structure of the AC algorithm used in propagateAC
2 if  $\neg$ propagateAC( $D, X, true$ ) then return false
3  $CHANGE \leftarrow false$ 
4  $PendingList \leftarrow \{(x_i, a) \mid x_i \in X, a \in D(x_i)\}$ 
5 while  $PendingList \neq \emptyset$  do
6   select and remove a value  $(x_i, a)$  from  $PendingList$ 
7   if  $a \in D(x_i)$  then
8     if  $\neg$ buildBranch( $(x_i, a), D, PendingList$ ) then
9        $D(x_i) \leftarrow D(x_i) \setminus \{a\}$ 
10      if  $\neg$ propagateAC( $D, \{x_i\}, true$ ) then return false
11       $CHANGE \leftarrow true$ 
12   if  $PendingList = \emptyset \wedge CHANGE$  then
13      $PendingList \leftarrow \{(x_i, a) \mid x_i \in X, a \in D(x_i)\}$ 
14      $CHANGE \leftarrow false$ 
15 return true

function buildBranch(in  $(x_i, a)$ ; in  $D$ ; inout  $PendingList$ ): Boolean
16 save the data structure of the AC algorithm
17  $D(x_i) \leftarrow \{a\}$ 
18 if  $\neg$ propagateAC( $D, \{x_i\}, true$ ) then
19   restore the data structure of the AC algorithm
20   return false
21 while  $(PendingList \cap D \neq \emptyset)$  do
22   select a value  $(x_j, b)$  from  $PendingList \cap D$ ; remove it from  $PendingList$ 
23    $D(x_j) \leftarrow \{b\}$ 
24   if  $\neg$ propagateAC( $D, \{x_j\}, true$ ) then
25      $PendingList \leftarrow PendingList \cup \{(x_j, b)\}$ 
26     break the while loop
27 restore the data structure of the AC algorithm
28 return true

```

We can observe that in lines 18 and 24, `propagateAC` is called without the set *Deleted* because we are in a subproblem and removing a value from the subproblem does not mean that this value is globally inconsistent. But we call `propagateAC` with the parameter *UPD* set to true if we want to keep optimality of AC on the branch. This is why in lines 16, 19 and 27, we save and restore the original data structure. A simpler version of SAC3 can be obtained by setting *UPD* to false and removing lines 16, 19 and 27. For example, by using residual supports [19], one can expect to obtain

a good behavior in practice. This is the version that will be used in some experiments of Section 9.

Theorem 2 *SAC3 is a correct SAC algorithm with $O(bT)$ time complexity where b denotes the total number of branches built by SAC3. This gives $O(bed^2)$ on binary constraints. The space complexity of SAC3 is $O(nd + S)$ where S is the space complexity of the AC algorithm used.*

Proof Correctness results from Proposition 1.

The data structure *PendingList* has size nd and S is the space complexity of the AC algorithm used. Hence, SAC3 has an overall space complexity in $O(nd + S)$. If SAC3 embeds an optimal arc consistency algorithm such as AC2001, then the overall space complexity is in $O(erd)$.

The cost of saving and restoring the data structure on a branch (see lines 16, 19 and 27) cannot be more than the time complexity T of the AC algorithm using them. Furthermore, due to incrementality, the cost of AC on each branch built by SAC3 is in $O(T)$. Consequently, the overall time complexity of SAC3 is in $O(bT)$ (i.e., $O(bed^2)$ on binary constraints). \square

Notice that b takes into account the branches of size 1 that correspond to the detection of singleton arc-inconsistent values. Thus, in the worst-case, we have $b = \frac{n^2d^2+nd}{2}$ because we may have to build up to nd branches before removing a first value, up to $nd - 1$ additional branches before removing a second value, etc. We then obtain a worst-case time complexity (with AC2001 embedded) in $O(n^2d^2T)$, that is, the same complexity as SAC1. However, when a constraint network is already singleton arc-consistent, we can make the following observation.

Observation 1 *On a constraint network that is singleton arc-consistent, SAC3 builds between d and nd branches.*

The number of branches built by SAC3 is minimal when each branch (of size n) leads to a solution and uses as much as possible values not yet used by other branches. Each branch contains at most one value per variable. Hence, d branches are required to cover all values of the largest domain (of size d). The maximal number of branches is reached when all branches are of size 2 (one consistent assignment followed by an inconsistent one). In this case, only the first value of each branch is proved singleton arc-consistent, which requires nd branches to cover all of them.

As a consequence of this observation, on a constraint network that is already singleton arc-consistent, SAC3 embedding an optimal AC algorithm such as AC2001, has a time complexity in $O(ndT)$ (i.e., $O(end^3)$ on binary constraints) because it explores at most nd branches of size 2. Interestingly, this suggests that SAC3 may be quite competitive with respect to SAC-SDS on structured (not necessarily singleton arc-consistent) instances that contain large under-constrained parts as can be expected in real-world applications.

SAC3 shows another interesting feature. Although the primary goal of the greedy approach is to efficiently enforce singleton arc consistency by exploiting the incrementality of arc consistency, one may opportunistically find solutions during one of the greedy runs.

7 SAC3-SDS

It is possible to improve the behavior of the algorithm SAC3 by adapting the idea of storing domains of subproblems as in SAC-SDS. As an extension of SAC3, we can then decide to record the domain of the constraint networks obtained after each greedy run, that is to say, for each branch. Consequently, as in SAC-SDS, when a value is removed, it is possible to determine which previously built branches must be reconsidered. Indeed, if a removed value does not belong to the domain associated with a given branch, all values of this branch remain singleton arc-consistent. On the other hand, if a removed value supports a branch, we have to verify that the branch still remains valid by re-establishing arc consistency from the recorded domain. When a branch is no more valid, we have to delete it. In summary, SAC3-SDS can (partially) exploit incrementality as SAC-SDS does.³

A branch corresponds to a set Br of values that have been (explicitly) assigned. We record all branches built by the algorithm in a set called *Branches*. In order to manage domains and propagation of subproblems corresponding to branches, we also consider two arrays denoted $D[\]$ and $Q[\]$. For a given branch Br , $D[Br]$ is the local domain of the subproblem associated with the branch Br and $Q[Br]$ is its local propagation list. $Q[Br]$ contains the variables that should be considered when re-establishing arc consistency in the subproblem because they have lost values. Those $D[Br]$ and $Q[Br]$ play the same role as D_{ia} and Q_{ia} in SAC-SDS. More rigorously, we should associate an integer with each branch to be used as index value for $D[\]$ and $Q[\]$. But we omit it to simplify the presentation.

SAC3-SDS (see Algorithm 4) works as follows. After some initializations (lines 1–4), SAC3-SDS enters the main loop (line 5) and repeatedly pops an object from *PendingList* (line 6). Observe that *PendingList* may contain two types of objects: it may contain values (x_i, a) that have to be checked for singleton arc consistency, and it may also contain branches Br that have lost values in their associated subdomain. If the popped object is a branch Br (of the set of branches *Branches*), SAC3-SDS enforces arc consistency on its subdomain $D[Br]$, knowing that the variables modified since the last pass of AC are in the propagation list $Q[Br]$ (line 8). A wipe-out when enforcing AC on $D[Br]$ means that the values in the branch Br are no longer guaranteed to be singleton arc-consistent, and we have to put them back in *PendingList* and remove Br from the set of branches (lines 9–10). If the object popped in line 6 is a value (x_i, a) , we have to check its singleton arc consistency. As in SAC3, this is done by calling `buildBranch` with (x_i, a) as a parameter (line 11). In addition to building a branch, function `buildBranch` returns false if $D_{x_i=a}$ is arc-inconsistent, true otherwise. If false is returned, (x_i, a) has been detected singleton arc-inconsistent, it must be removed from the main problem, and arc consistency must be re-established (lines 12–15). All branches Br such that values from $D[Br]$ have been removed from the main problem by the arc consistency pass of line 13 have to be put back in *PendingList* for revisiting them. This is done by function `updateBranches` (line 14). If the AC call of line 13 has led to a wipe-out, the problem is proved to be inconsistent and SAC3-SDS returns false (line 15).

³SAC3-SDS is a refinement of the algorithm SAC3+ presented in [18]. The new algorithm is made more flexible (since at each step either a branch or a value can be selected) and consequently more reactive. Worst-case complexities remain the same.

Algorithm 4: SAC3-SDS

```

function SAC3-SDS(inout  $P = (X, D, C)$ : problem): Boolean
1 initialize the data structure of the AC algorithm used in propagateAC
2 if  $\neg$ propagateAC( $D, X, true$ ) then return false
3  $Branches \leftarrow \emptyset$ 
4  $PendingList \leftarrow \{(x_i, a) \mid x_i \in X, a \in D(x_i)\}$ 
5 while  $PendingList \neq \emptyset$  do
6     select and remove an object from  $PendingList$ 
7     if object is a branch  $Br \in Branches$  then
8         if  $\neg$ propagateAC( $D[Br], Q[Br], false$ ) then
9              $PendingList \leftarrow PendingList \cup \{(x_j, b) \in Br\}$ 
10             $Branches \leftarrow Branches \setminus \{Br\}$ 
11        else /* object is a value  $(x_i, a)$  */
12            if  $\neg$ buildBranch( $(x_i, a), D, PendingList$ ) then
13                 $D \leftarrow D \setminus (x_i, a)$ ;  $Deleted \leftarrow \{(x_i, a)\}$ 
14                if propagateAC( $D, \{x_i\}, true, Deleted$ ) then
15                    updateBranches( $Deleted$ )
16                else return false

```

```

function buildBranch(in  $(x_i, a)$ ; in  $D$ ; inout  $PendingList$ ): Boolean
16 save the data structure of the AC algorithm
17  $D(x_i) \leftarrow \{a\}$ 
18 if  $\neg$ propagateAC( $D, \{x_i\}, true$ ) then
19     restore the data structure of the AC algorithm
20     return false
21  $Br \leftarrow \{(x_i, a)\}$ ;  $D_{pred} \leftarrow D$ 
22 while  $(PendingList \cap D \neq \emptyset)$  do
23     select a value  $(x_j, b)$  from  $PendingList \cap D$ ; remove it from  $PendingList$ 
24      $D(x_j) \leftarrow \{b\}$ 
25     if propagateAC( $D, \{x_j\}, true$ ) then
26          $Br \leftarrow Br \cup \{(x_j, b)\}$ ;  $D_{pred} \leftarrow D$ 
27     else
28          $PendingList \leftarrow PendingList \cup \{(x_j, b)\}$ 
29         break the while loop
29  $D[Br] \leftarrow D_{pred}$ ;  $Q[br] \leftarrow \emptyset$ 
30  $Branches \leftarrow Branches \cup \{Br\}$ 
31 restore the data structure of the AC algorithm
32 return true

```

```

function updateBranches(in  $Deleted$ : set)
33 foreach  $Br \in Branches \mid D[Br] \cap Deleted \neq \emptyset$  do
34      $Q[Br] \leftarrow Q[Br] \cup \{x_i \in X \mid D[Br](x_i) \cap Deleted \neq \emptyset\}$ 
35      $D[Br] \leftarrow D[Br] \setminus Deleted$ 
36      $PendingList \leftarrow PendingList \cup \{Br\}$ 

```

Function `buildBranch` works on the same principle as in SAC3, except that it has to store the branch Br and the domains $D[Br]$ of the subproblem associated with the branch Br . So lines 16 to 20 are the same. The first difference happens in line 21 where the branch Br is initialized with (x_i, a) once it has been proved that $P|_{x_i=a}$ is not arc-inconsistent. The second difference is in line 26, when extending the branch with value (x_j, b) has not led to arc inconsistency: Br is extended to contain (x_j, b) and the current domain is stored in D_{pred} for future use in line 29. If a wipe-out is detected when trying to extend Br , the last value of the branch is put back in `PendingList` and the loop is broken (lines 27–28). At this point, domain D corresponds to the subdomain which has failed because of the last value we tried to add to Br . We need to “backtrack” to the previous state, that is, to the domain stored in D_{pred} . This is done in line 29, but an implementation in a solver will use the backtracking facilities of the solver. Finally, Br is added to the set `Branches` (line 30) and the data structures of the AC algorithm are restored before returning true.

Function `updateBranches` (lines 33 to 36) is the same as `updateSubProblems` from SAC-SDS except that it deals with branches instead of values.

Theorem 3 *SAC3-SDS is a correct SAC algorithm with $O(b_{tot}T')$ time complexity, where T' is the cost of enforcing arc consistency on the network with an AC algorithm that does not store any special data structure between two calls and b_{tot} is the total number of times a branch is built by SAC3-SDS. This gives $O(b_{tot}ed^3)$ on binary constraints. The space complexity of SAC3-SDS is $O(b_{max}nd + S)$, where S is the space complexity of the AC algorithm used and b_{max} is the maximum number of branches recorded at the same time by SAC3-SDS.*

Proof Correctness. Soundness is obvious. As for completeness, each time the main loop of SAC3-SDS starts, every value in D is ensured to belong either to `PendingList` or to a branch in `Branches`. Any branch Br in `Branches` is ensured either to correspond to an arc-consistent subdomain $D[Br]$ or to be in `PendingList`. Since SAC3-SDS stops on empty `PendingList`, we are guaranteed that all values remaining in D are in a branch Br for which $D[Br]$ is arc-consistent. By Proposition 1, we have that all values in D are singleton arc-consistent.

Complexity. In addition to the space requirement in $O(S)$ of the underlying arc consistency algorithm, it is necessary to record the domain of the subproblems corresponding to the valid branches that have been built. As recording a domain is in $O(nd)$, we obtain $O(b_{max}nd + S)$.

Concerning time complexity, the total cost is dominated by the cost of building and updating branches. As in SAC3 the cost of building each branch in `buildBranch` is in $O(T)$. However, branches may need to be updated (line 8). The total cost on all these updates is *not* $O(ndT)$ because the domain of the subproblem is stored from one call to another. On the other hand, the total cost for each branch (including both the construction and the successive updates) is *not* $O(T)$ because the data structures of the AC algorithm are not saved from one call to another (UPD is false when calling `propagateAC` at line 8). The total cost is $O(T')$, the cost of arc consistency without storing data structures. So, the total time complexity of SAC3-SDS is $O(b_{tot}T')$ (i.e., $O(b_{tot}ed^3)$ on binary constraints). \square

Note that $b_{max} \leq b_{tot}$ ($= O(n^2d^2)$) because b_{max} is the maximum number of branches SAC3-SDS *simultaneously* stores in the set `Branches` whereas b_{tot} is the

Table 2 Complexities of the new SAC algorithms

Algorithm	Non-binary CN		Binary CN	
	Time	Space	Time	Space
SAC-SDS	$O(ndT')$	$O(n^2d^2 + S)$	$O(end^4)$	$O(n^2d^2)$
SAC3	$O(bT)$	$O(nd + S)$	$O(bed^2)$	$O(ed)$
SAC3-SDS	$O(b_{tot}T')$	$O(b_{max}nd + S)$	$O(b_{tot}ed^3)$	$O(b_{max}n + e)d$

T and S are the time and space complexities of the underlying (G)AC algorithm. T' is the cost of enforcing (G)AC with an algorithm that does not store any special data structure between two calls. b_{tot} is the total number of times a branch is built by SAC3-SDS and b_{max} is the maximum number of branches recorded at the same time by SAC3-SDS. For binary constraints, we consider an optimal AC algorithm, such as AC2001, with time in $O(ed^2)$ and space in $O(ed)$

total number of branches SAC3-SDS will have stored at some point during its execution. b_{max} and b_{tot} are both smaller than the number b^+ of times SAC3-SDS rechecks arc consistency of one of its branches (line 8 of Algorithm 4). Note that b^+ is not necessarily equal to b (see SAC3) because SAC3 and SAC3-SDS do not propagate *PendingList* in the same ordering. Nevertheless, as in Observation 1 for SAC3, we obtain $d \leq b^+ \leq nd$ on constraint networks that are already singleton arc-consistent. In addition, on such networks, each branch is built once and never rechecked, for a total cost in $O(T)$ instead of $O(T')$. As a consequence, on a constraint network that is already singleton arc-consistent, SAC3-SDS embedding an optimal AC algorithm such as AC2001 has a time complexity in $O(ndT)$ like SAC3 (i.e., $O(end^3)$ on binary constraints). Overall, one should be optimistic about the average time complexity of SAC3-SDS because, as opposed to SAC3, it avoids building new branches when unnecessary.

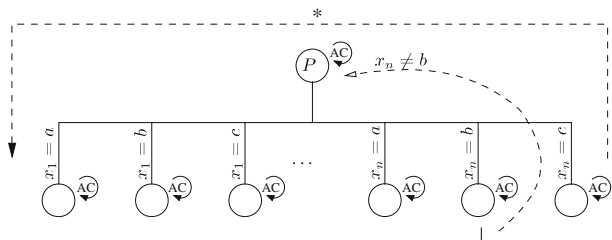
Table 2 shows the complexities of the new algorithms presented in this paper.

8 Illustrations

In this section, we propose a visual illustration of the different algorithms mentioned in this paper. Here, we suppose a constraint network involving n variables x_1, \dots, x_n with $D(x_i) = \{a, b, c\} \forall i \in 1..n$.

Figure 1 provides an illustration of the algorithm SAC1. After having established AC, all singleton checks are performed in turn; this is the first pass. Here, the value (x_n, b) is assumed to be proved inconsistent. It means that a second pass is necessary: each remaining value (after propagating $x_n \neq b$) has to be checked; see the dotted arrow. Imagine that during the second pass, the value (x_j, a) is shown singleton

Fig. 1 Illustration of SAC1. After identifying (x_n, b) as a singleton arc-inconsistent value, $x_n \neq b$ is propagated and a second pass is performed



be singleton arc-inconsistent. With SAC3, we know that a second pass is necessary, starting from scratch. However, for SAC3-SDS, only branches containing (x_n, b) in their associated subdomain have to be considered again. The opportunity of finding a solution during the inference process is shown in the figure.

9 Experiments

In order to show the practical interest of the algorithms presented in this paper, we have performed several experiments. For random problems, all algorithms have been implemented in C++ and ran on a Pentium IV-1600 Mhz with 512 Mb of memory under windows XP whereas for structured problems, we have used the constraint solver Abscon and run it on a cluster of Xeon 3.0 GHz with 1GiB of RAM under Linux.⁴ We have compared existing SAC algorithms, namely SAC1 and SAC2, with those presented in this paper: SAC-SDS, SAC3 and SAC3-SDS. SAC-Opt was also tested, but shown to be too space consuming in general.

9.1 Random problems

We compared the performance of the SAC algorithms on random uniform constraint networks of binary constraints generated with the generator in [15], which produces instances according to Model B [25]. These instances are characterized by the four parameters $\langle n, d, p_1, p_2 \rangle$, where n is the number of variables, d the size of all domains, p_1 the *density*, i.e., the proportion of pairs of variables linked by a binary constraint (among the $n(n-1)/2$ possible pairs), and p_2 the *tightness*, i.e., the proportion of forbidden pairs of values inside a constraint (among the d^2 possible pairs). The results reported are for versions of the algorithms based on AC2001. Note that for SAC2 the implementation of the propagation list has been done according to the recommendations made in [2]. The heuristic used for selecting values in *PendingList* is *lifo*.⁵ For each combination of parameters tested, we generated 50 instances for which we report mean cpu times.

9.1.1 Experiments on sparse constraint networks

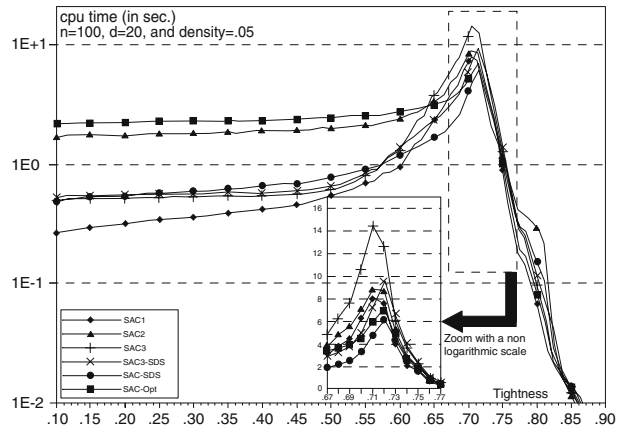
Figure 4 presents performance on constraint networks having 100 variables, 20 values in each domain, and a density of .05. In these relatively sparse constraint networks the variables have five neighbors on average.

For a tightness lower than .55, all the values are singleton arc-consistent. On these under-constrained networks, the SAC algorithms check the arc consistency of each subproblem at most once. Storing support lists (as in SAC2) or duplicating for each value the data structures of the AC algorithm (as in SAC-Opt) is useless and prohibitive. Storing only local subdomains (as in SAC-SDS) or using a greedy

⁴The use of two different computers is due to our two platforms, one limited to binary constraints in which all algorithms were implemented, and the other one allowing any type of constraints, but in which some algorithms were not available.

⁵Other heuristics have been tested but due to lack of structure in random instances, the difference of behavior between heuristics is small.

Fig. 4 cpu time on random problems with $n = 100$, $d = 20$, and density = .05



approach (as in SAC3 and SAC3-SDS) is far less costly, but the brute-force algorithm SAC1 shows the best performance.

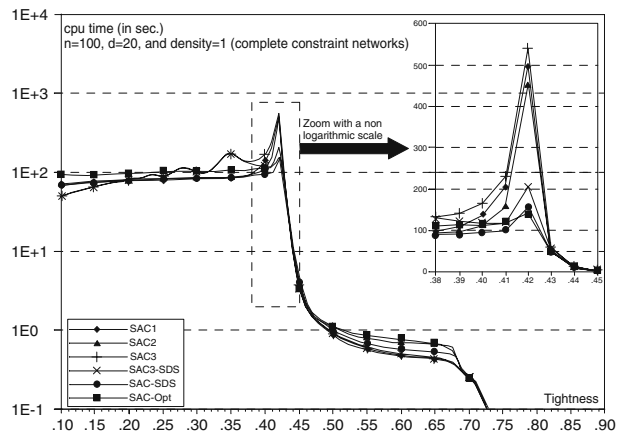
On problems having tighter constraints, some singleton arc-inconsistent values are removed and at tightness .72 we can see a peak of complexity. As mentioned in [2], the improved propagation of SAC2 is useless on sparse constraint networks. SAC2 is always more expensive than SAC1. Around the peak of complexity, SAC-SDS is the clear winner. SAC-Opt and SAC1 are around 1.7 times slower, and all the others are between 2.1 and 3 times slower.

9.1.2 Experiments on dense constraint networks

Figure 5 presents performance on constraint networks having 100 variables, 20 values in each domain and a complete graph of binary constraints.

SAC1 and SAC2 show very close performance. When all values are SAC (tightness lower than .37) the additional data structure of SAC2 is useless because there is no propagation. Nevertheless, the cost of building this data structure is not important compared to the overall time and the time required by SAC2 is almost the same

Fig. 5 cpu time on random problems with $n = 100$, $d = 20$, and density = 1



as SAC1. Around the peak of complexity, SAC2 requires slightly less time than SAC1. The reason is that SAC2 rechecks the arc consistency of less subproblems than SAC1. On very tight constraints, SAC1 requires less time than SAC2 because the inconsistency of the problem is found with almost no propagation and building the data structure of SAC2 is useless.

The best results are obtained with SAC-Opt and SAC-SDS which are between 1.41 and 3.74 times faster than the others at the peak. The smarter propagation mechanism of these two algorithms pays off on these problems. However, the interest of SAC-Opt can be seen only at the peak since it remains very costly elsewhere.

When constraints are not tight, SAC3 and SAC3-SDS build long branches and clearly show the best performance. However, as on sparse problems, SAC3 is the most costly at the peak. Without storing the branches on which rely the singleton consistency of the values, SAC3 performs useless tests. The enhanced version SAC3-SDS has not this drawback and highlights a better behavior at the peak.

9.2 Structured problems

For structured problems, we have decided to focus on the most efficient algorithms identified above, while keeping SAC1 as baseline algorithm. SAC2 was removed as a clear loser in the previous experiments. SAC-Opt was tested on a few binary structured problems, but it reached the space limit so often that we decided not to include it. For algorithms based on greedy search, we have tested several heuristics to select values when building branches. Two representative ones are *lifo*⁶ already used in our experiments with random instances, and *dom/wdeg*. Let us call a value of $PendingList \cap D$ a valid choice. While *lifo* selects the youngest valid choice put in *PendingList*, *dom/wdeg* selects first the variable with the best ratio “domain size” on “weighted degree” [9] among variables for which at least a valid choice exists. For *dom/wdeg*, if x is the selected variable, the youngest valid choice for x is considered. There is one case where *dom/wdeg* is overridden: when we encounter a failure that does not correspond to an identified singleton arc-inconsistent value (i.e., when a branch with more than one assignment terminates with a domain wipe-out), a new branch is systematically started with the variable-value pair assignment that led to the wipe-out. Note that *lifo* automatically behaves in a similar way after each encountered failure. The main benefit is that the last variable assignment before the wipe-out is more likely to fail again.

The (G)AC algorithm embedded in the various SAC algorithms used for this experiment (concerning structured problems) is (G)AC3sm. Although non-optimal in theory, this algorithm is known to be quite efficient in practice, even for non-binary problems [19]. For problem series, such as *renault* and *crosswords*, that involve constraints in extension of arity strictly greater than 3, we have used the algorithm STR to enforce GAC. This algorithm [16, 29] belongs to state-of-the-art GAC algorithms for constraints in extension (so called table constraints).

We have experimented on series of structured CSP instances that are available from <http://www.cril.fr/~lecoutre>. These series represent a large spectrum of instances, and importantly, allow anyone to easily reproduce our experimentation.

⁶The heuristic *fifo* is rather close to *lifo* in terms of performance, but *lifo* has the advantage of checking in priority the last value assigned when a branch ends with a failure.

Among the series, we find instances from the scheduling job-shop problem (instances with prefix *ewddr*), graph-coloring problem (instances with prefix *fpsol*), all-interval series problem (instances with prefix *series*), Langford problem, radio-link frequency assignment problem (so-called RLFAP instances with prefix *graph* or *scen*), frequency assignment problem with polarization (so-called FAPP instances with prefix *fapp*), Renault configuration problem, crossword problem (instances with prefix *cw*), queen attacking problem, Golomb ruler problem (instances with prefix *ruler*), and primes problem.

We have distinguished between three main categories of instances: those that are already singleton arc-consistent, those that are singleton arc-inconsistent, and finally those that are neither singleton arc-consistent nor singleton arc-inconsistent. Even if the most interesting category is the last one, we briefly analyze some results concerning the two other categories below. Results are given for representative structured instances since contrary to random problems, computing mean results per series is almost always irrelevant for SAC. Below the name of each instance, we give three numbers, *i-j-k*. *i* is the number of initial values, *j* is the number of values removed by AC, and *k* is the number of values removed by SAC (including values removed by AC, i.e., we have $k \geq j$). For example, for the instance *graph3* in Table 5, we know that the number of initial values is 7820, the number of removed arc-inconsistent values is 340 and the total number of values removed after enforcing SAC is 1274. For each algorithm tested, the cpu time (in seconds) as well as the number of singleton checks (#scks) are given.

Table 3 shows the results obtained on some representative instances for which enforcing SAC has no impact: these instances are already singleton arc-consistent. Consequently, applying a SAC algorithm on such instances is nothing more than a waste of time. Nevertheless, it is interesting to see the relative behavior of the different algorithms. First, note that SAC1 and SAC-SDS behave equivalently since

Table 3 Results obtained on instances (already) singleton arc-consistent

Instance		SAC1/	SAC3		SAC3-SDS	
		SAC-SDS	<i>lifo</i>	<i>dom/wdeg</i>	<i>lifo</i>	<i>dom/wdeg</i>
<i>ewddr2-1</i> (6720-0-0)	cpu	2.52	1.16	1.51	1.25	1.52
	#scks	6,720	6,868	6,851	6,868	6,851
<i>ewddr2-10</i> (6895-0-0)	cpu	2.28	0.98	1.35	1.07	1.35
	#scks	6,895	6,924	6,934	6,924	6,934
<i>fpsol2-i-1-15</i> (4035-0-0)	cpu	21.3	10.7	14.1	10.7	13.5
	#scks	4,035	4,378	4,415	4,378	4,415
<i>fpsol2-i-1-20</i> (5380-0-0)	cpu	26.5	13.0	18.1	12.7	18.9
	#scks	5,380	5,689	5,755	5,689	5,755
<i>series-40</i> (3121-0-0)	cpu	9.42	12.5	10.2	12.7	10.1
	#scks	3,121	3,205	3,204	3,205	3,204
<i>series-50</i> (4901-0-0)	cpu	23.1	36.7	31.4	34.4	31.6
	#scks	4,901	5,004	5,010	5,004	5,010
<i>langford-3-40</i> (9240-0-0)	cpu	102	21.9	21.9	21.0	21.8
	#scks	9,240	9,331	9,327	9,331	9,327
<i>langford-3-50</i> (14550-0-0)	cpu	253	53.7	53.5	54.0	53.4
	#scks	14,550	14,652	14,648	14,652	14,648

SAC-SDS has a behavior similar to SAC1 because a value never needs to be tested twice

never a value has to be tested twice. The only difference between the two algorithms is the overhead of managing the data structures that are specific to SAC-SDS. As it is quite negligible here, we present the results for both algorithms within a single column. Second, for the greedy algorithms SAC3 and SAC3-SDS, using a “clever” heuristic to build branches has a very limited impact. This is not surprising because these instances are (highly) under-constrained. Whatever the heuristic is, the chance of building long similar branches is important. Finally, in terms of results, if the greedy algorithms are overall faster than SAC1/SAC-SDS, in some cases (see series-50) they can be penalized. Indeed, on certain instances, whereas each isolated singleton check is cheap as involving no propagation, building a branch leads to a point where propagation starts to be effective, which is counter-productive. Nevertheless, note that the gain obtained by SAC3 and SAC3-SDS is sometimes substantial: on Langford instances, both algorithms are five times faster than SAC1/SAC-SDS.

Table 4 presents the results obtained on some instances that are singleton arc-inconsistent but not arc-inconsistent. Again, we merge the results obtained with SAC1 and SAC-SDS because on all these instances, except *graph8-f11*, only one pass is necessary to prove that the instances are unsatisfiable (i.e., each values is tested at most once). As a consequence, SAC-SDS is not given the opportunity to exploit the incrementality of arc consistency algorithms. On *graph8-f11* the difference observed between SAC1 and SAC-SDS is small. As a first general observation, it appears that greedy SAC algorithms are almost always beneficial. It is partially due to the incrementality exploited when building branches. Another explanation is the natural inclination of greedy SAC algorithms to “explore” different portions of the search space. Indeed, instead of focusing on values of the same variable, SAC3 and SAC3-SDS necessarily test values for different variables. It allows them to quickly find the most constrained variables and potentially singleton arc-inconsistent values since both heuristics *lifo* and *dom/wdeg* “reason” from the last conflict. By additionally learning from conflicts through constraint weighting, *dom/wdeg* is capable of progressively focusing on the hard region(s) of the constraint network, those where singleton arc-inconsistent values are expected to appear. This is confirmed by the

Table 4 Results obtained on instances singleton arc-inconsistent

Instance		SAC1/ SAC-SDS		SAC3		SAC3-SDS	
				<i>lifo</i>	<i>dom/wdeg</i>	<i>lifo</i>	<i>dom/wdeg</i>
<i>graph8-f11</i> (19322-6306-19322)	cpu		3.36	2.52	2.0	4.44	3.47
	#scks	21,996		19,149	13,898	18,881	13,878
<i>scen7-w1-f5</i> (14176-4836-14176)	cpu		0.24	0.25	0.06	0.32	0.06
	#scks	3,846		3,442	135	3,442	135
<i>fapp5-350-9</i> (79311-15943-79311)	cpu		149	23.2	2.54	21.6	2.66
	#scks	5,709		595	61	595	61
<i>fapp5-350-10</i> (79311-9349-79311)	cpu		221	50.1	1.81	50.6	1.79
	#scks	9,541		1,886	35	1,886	35
<i>renault-mod-12</i> (562-24-562)	cpu		1.15	0.07	0.10	0.08	0.11
	#scks	474		131	214	131	214
<i>renault-mod-15</i> (562-14-562)	cpu		1.13	0.05	0.12	0.05	0.14
	#scks	481		71	243	71	243

SAC-SDS has a behavior similar to SAC1 because only one pass is necessary to prove inconsistency (except for instance *graph8-f11*)

number of singleton checks performed by the different algorithms. For example, on instance *fapp5-350-10*, only 35 singleton checks are necessary to SAC3 and SAC3-SDS to prove unsatisfiability when using *dom/wdeg*. When using *lifo*, they require 1,886 singleton checks. When a classical approach is used, almost 10,000 singleton checks are required. Obviously, it has a dramatic impact in terms of cpu time.

Finally, we have selected representative instances that are neither singleton arc consistent nor singleton arc inconsistent. First, we have tested instances originated from real-world applications (see Table 5). On RLFAP instances, the best identified algorithm is SAC3-SDS combined with *dom/wdeg*: it is from 3 to 10 times faster than the basic SAC1. The interest of using *dom/wdeg* instead of *lifo* is clear as it permits to be twice faster. SAC-SDS is not very effective here as it runs out of memory on *graph10*. However, on FAPP instances, i.e., instances from the Frequency Assignment Problem with Polarization, SAC-SDS has a good behavior. It is quite close to SAC3-SDS-*dom/wdeg* and largely outperforms the other SAC algorithms. On non-binary instances from *renault* and *crossword* series, the observed difference between using *lifo* and *dom/wdeg* is rather small. In fact, these instances contain a more regular structure than the one present in frequency allocation problem instances. Consequently, constraint weighting has a limited⁷ impact. Whereas SAC3-SDS is the most efficient approach on *renault* instances, this is not so clear on *crossword* ones. Roughly speaking, SAC-SDS and SAC3-SDS are the best algorithms for these instances but they are “only” 50% faster than SAC1. SAC3 is even outperformed by SAC1 when dictionary *uk* is used. Exploiting the incrementality of arc consistency on crossword instances seems then little effective. Here, we have to mention that if STR is replaced by a generic GAC algorithm such as GAC2001, the results obtained by SAC3 and SAC3-SDS are damaged, compared to SAC1 and SAC-SDS. It illustrates the difficulty of experimentally comparing algorithms based on approaches that are significantly different. The results obtained on academic instances (Table 6) confirm our previous observations. On some series, SAC3-SDS is clearly the best approach (*queenAttacking*), but on some other series (*ruler* and *primes*), SAC3-SDS and SAC-SDS have close performances.

For SAC1 and SAC-SDS, we have also tested heuristics such as *dom/wdeg* to select values from *PendingList*. On a limited number of instances, we obtain an improvement of the algorithm performance, but this remains unusual. The difference between SAC algorithms based on a breadth-first approach and SAC algorithms based on a greedy approach is that the latter have the possibility of quickly learning and diversifying exploration. This renders the use of an adaptive heuristic such as *dom/wdeg* quite relevant.

What general lessons can we draw from these experiments? Clearly, there is no algorithm always dominating all the others. We can always find an instance, which, because of its structure or because of its constrainedness, will be favorable to an algorithm and not to another. Nevertheless, there are general trends that are quite visible. When problems are not trivially under-constrained, the incrementality of SAC-SDS and of SAC3-SDS pays off. When problems are structured, as real problems often are, algorithms based on a greedy search show the best performance. All in all, SAC3-SDS appears as an overall winner, benefiting from both incrementality and greedy search.

⁷Besides, the higher the arity, the lower the preciseness of constraint weighting.

Table 5 Results obtained on real-world instances

Instance		SAC1	SAC-SDS	SAC3		SAC3-SDS	
				<i>lifo</i>	<i>dom/wdeg</i>	<i>lifo</i>	<i>dom/wdeg</i>
Binary instances							
<i>graph3</i>	cpu	22.2	11.0	8.51	5.02	5.04	2.82
(7820-340-1274)	#scks	20,075	15,798	22,164	21,001	8,375	7,475
<i>graph4</i>	cpu	82.2	39.8	33.4	18.7	23.9	14.0
(15592-776-2876)	#scks	51,957	37,394	44,912	41,996	16,508	15,370
<i>graph10</i>	cpu	141	Mem out	82.9	50.7	55.5	29.5
(26980-386-2572)	#scks	74,321		83,046	78,288	29,093	27,557
<i>scen5</i>	cpu	0.85	1.38	0.31	0.27	0.32	0.27
(15768-12046-13814)	#scks	6,261	3,966	4,215	4,148	2,487	2,242
<i>fapp1-200-4</i>	cpu	77.3	39.2	108	102	56.9	33.8
(26963-13029-16038)	#scks	33,955	41,832	53,773	62,839	17,285	14,373
<i>fapp1-200-6</i>	cpu	83.5	33.1	80.3	65.7	33.9	26.4
(26963-11039-12082)	#scks	45,200	41,425	49,846	48,351	17,491	16,685
<i>fapp1-200-8</i>	cpu	85.4	32.6	71.2	52.8	27.3	20.4
(26963-9155-9523)	#scks	52,603	37,061	56,374	54,882	19,127	18,576
<i>fapp1-200-10</i>	cpu	38.6	20.0	28.0	23.2	15.2	12.1
(26963-5976-6053)	#scks	41,889	38,484	42,762	42,394	21,440	21,250
Non-binary instances							
<i>renault</i>	cpu	2.0	1.15	0.55	0.49	0.33	0.29
(462-13-14)	#scks	754	409	781	781	392	392
<i>renault-mod-18</i>	cpu	2.11	1.51	0.71	0.74	0.47	0.4
(562-36-51)	#scks	1,013	962	1,177	1,167	596	591
<i>renault-mod-44</i>	cpu	3.85	2.14	1.42	1.0	0.63	0.5
(462-27-108)	#scks	1,739	1,565	1,893	1,403	665	623
<i>cw-lex-15-08</i>	cpu	15.6	9.72	14.5	14.8	7.48	7.7
(4836-231-235)	#scks	9,206	7,789	10,211	10,726	5,105	5,365
<i>cw-lex-19-08</i>	cpu	30.4	25.4	34.2	39.8	16.6	21.8
(7670-431-441)	#scks	14,468	14,372	15,895	16,741	7,953	8,374
<i>cw-lex-21-08</i>	cpu	40.0	31.6	39.3	42.9	20.9	21.8
(9490-357-360)	#scks	18,263	12,445	19,907	21,148	9,951	10,566
<i>cw-lex-23-08</i>	cpu	54.8	45.9	57.9	61.7	30.1	31.0
(11804-562-566)	#scks	22,480	22,313	24,624	26,944	12,316	13,462
<i>cw-uk-vg14-15</i>	cpu	187	128	224	230	135	116
(5460-248-480)	#scks	15,174	13,503	22,634	21,672	7,819	7,429
<i>cw-uk-vg14-16</i>	cpu	203	155	323	342	163	141
(5824-337-798)	#scks	20,586	15,304	40,134	39,029	8,690	8,410
<i>cw-uk-vg15-15</i>	cpu	133	125	204	193	140	141
(5850-300-850)	#scks	15,567	15,222	24,440	23,859	8,840	8,529
<i>cw-uk-vg15-16</i>	cpu	174	160	301	328	193	195
(6240-400-1444)	#scks	30,069	33,733	58,867	58,170	10,481	10,390

9.3 Enforcing SAC before and during search

The previous subsections show a comparison of the performance of SAC algorithms when enforcing SAC alone. However, each time new algorithms for a local consistency are proposed, the natural questions that arises is whether or not these new algorithms improve the search for solutions. In this section we briefly address this question in two ways. First, we compare MAC and SAC followed by MAC on a

Table 6 Results obtained on academic instances

Instance		SAC1	SAC-SDS	SAC3		SAC3-SDS	
				<i>lifo</i>	<i>dom/wdeg</i>	<i>lifo</i>	<i>dom/wdeg</i>
Binary instances							
<i>queenAttacking-6</i>	cpu	1.94	1.24	0.84	0.78	0.47	0.42
(1302-33-48)	#scks	2,523	1,702	2,752	2,788	1,388	1,400
<i>queenAttacking-7</i>	cpu	6.12	4.07	2.33	2.39	1.24	1.25
(2411-45-65)	#scks	4,712	3,255	4,996	5,052	2,507	2,536
<i>queenAttacking-8</i>	cpu	19.1	12.8	9.86	6.77	3.29	3.56
(4106-108-160)	#scks	7,942	5,664	12,661	8,557	4,269	4,299
<i>queenAttacking-9</i>	cpu	53.1	30.3	24.5	17.3	8.79	9.43
(6576-132-197)	#scks	12,819	9,045	20,200	13,598	6,785	6,828
Non-binary instances							
<i>ruler-34-8-a3</i>	cpu	38.8	11.0	37.4	13.5	12.0	7.5
(1232-168-351)	#scks	6,335	3,340	9,726	3,947	1,755	1,423
<i>ruler-34-9-a3</i>	cpu	50.8	15.7	55.6	36.9	18.6	14.2
(1539-240-513)	#scks	8,474	4,720	13,184	11,135	2,175	2,101
<i>ruler-44-9-a3</i>	cpu	189	46.4	185	123	64.2	36.5
(1989-240-513)	#scks	12,074	6,445	18,340	15,484	2,934	2,631
<i>ruler-44-10-a3</i>	cpu	257	64.9	241	181	78.1	53.1
(2430-330-718)	#scks	15,810	8,829	21,993	21,000	3,571	3,436
<i>primes-10-20-3-1</i>	cpu	40.1	21.3	20.7	13.3	11.4	8.36
(2800-386-484)	#scks	2,139	1,526	3,421	2,240	1,348	1,139
<i>primes-10-20-3-3</i>	cpu	140	91.3	202	205	123	104
(2800-180-225)	#scks	2,942	2,787	3,282	3,292	1,725	1,677
<i>primes-10-20-3-5</i>	cpu	355	128	246	383	133	127
(2800-111-133)	#scks	5,503	5,092	3,982	5,895	2,086	2,003
<i>primes-20-60-3-1</i>	cpu	5.5	6.96	3.75	4.2	4.04	4.55
(7000-3368-5938)	#scks	3,231	3,171	3,388	2,983	2,864	2,651
<i>primes-20-60-3-3</i>	cpu	487	227	674	567	268	227
(7000-629-1877)	#scks	25,288	20,585	34,982	29,909	9,380	8,540
<i>primes-20-60-3-5</i>	cpu	647	350	830	840	448	455
(7000-437-573)	#scks	12,850	11,422	14,305	14,574	7,255	7,480

selection of instances from subsections above. Second, we compare MAC and a brute force implementation of maintaining SAC during search.

One may wonder about the impact of enforcing SAC at preprocessing time before running MAC; what we will be denoted by SAC+MAC. Clearly, it will depend on the number (or proportion) of singleton arc-inconsistent values in the original instance to be solved. If there is none, enforcing SAC is just a waste of time (but sophisticated SAC algorithms moderate this drawback). At the other extreme, enforcing SAC may be sufficient to solve the instance either by proving its unsatisfiability (when the instance is singleton arc-inconsistent) or by opportunistically finding one solution

Table 7 Solving singleton arc-inconsistent instances using MAC and SAC3

Instance	MAC	SAC3
<i>queensKnights-10-5-mul</i>	0.5	0.4
<i>queensKnights-20-5-mul</i>	3.2	0.7
<i>queensKnights-30-5-mul</i>	12.8	1.7
<i>queensKnights-40-5-mul</i>	> 3600	46

Table 8 CPU time in seconds (and number of visited nodes) for instances of the queens and RLFAP problems, given a 30 min cutoff

Instance	MAC time (#nodes)	Maintaining	
		SAC1 time (#nodes)	SAC3 time (#nodes)
<i>100-queens</i>	4.2 (119)	time-out	17.4 (1)
<i>110-queens</i>	time-out	time-out	37.9 (1)
<i>120-queens</i>	1,636 (323K)	time-out	16.7(1)
<i>scen11-f12</i>	3.6 (696)	1,072 (42)	418 (6)
<i>scen11-f10</i>	4.4 (863)	1,732 (53)	814 (9)
<i>scen11-f8</i>	67.8 (14K)	time-out	time-out

(when greedy runs are performed). To illustrate the case where there are SAC inconsistent values in the instance, let us consider some instances of the *queen-sKnights* problem as described in [9]. Instances of the form *queensKnights-n-5-mul* are singleton arc-inconsistent (n denotes the size of the chessboard), and so enforcing SAC allows us to avoid exploring a search tree. Table 7 shows the cpu time required to solve some of these instances (for $n = 10, 20, 30$ and 40) when MAC is used (using *dom/wdeg* as variable ordering heuristic⁸ and *lexico* as value ordering heuristic) but also when SAC is enforced at preprocessing (using algorithm SAC3-*dom/wdeg*). The interest of SAC at preprocessing is immediate here.

The second question is about the cost of maintaining SAC during search; what we will be denoted by MSAC. In Table 8 we compare MAC and MSAC both on satisfiable and unsatisfiable instances. Results are given for MAC, MSAC1 and MSAC3 (where MSACX maintains SACX during search). We chose SAC1 and SAC3 because they do not involve complex data structures to maintain, so that we expect them not to be too much penalized by our brute force implementation for maintaining SAC during search.

The top of Table 8 illustrates what happens on satisfiable instances. The instances used are satisfiable instances of the *n-queens* problem. It is interesting to note that for all these satisfiable instances, SAC3 visits a single node. This is because the solution is found by one of the greedy runs of the first pass of SAC. This makes MSAC3 much more efficient than MAC. However, MSAC1 is terribly inefficient. This confirms that efficient (and opportunistic) algorithms for SAC are useful.

The bottom of Table 8 illustrates the behavior of the same algorithms on unsatisfiable instances. The instances used are some difficult (modified) unsatisfiable RLFAP instances. The results show that maintaining SAC significantly reduces the number of nodes that have to be visited. Compared to MAC, MSAC1 reduces the number of nodes by one order of magnitude whereas MSAC3 reduces it by two orders of magnitude. This means that one order of magnitude of reduction of the search tree comes from the pruning power of SAC whereas the second order of magnitude comes from the fact that SAC3 learns from failures (of greedy runs) via use of the *dom/wdeg* heuristic. Unfortunately, this strong reduction of the size of the search space does not compensate for the cost of applying SAC at each node. Time performance of MSAC algorithms are much worse than those of MAC.

⁸Results are even worse when classical non-adaptive variable ordering heuristics such as *dom* (choosing at each step the variable with the smallest domain size) are used.

The high cost of applying SAC at each node has already been pointed out in [20]. The authors proposed relaxed versions of SAC that could be maintained during search at a lower cost than full SAC. For instance, they proposed existential SAC, that stipulates that at least one value in each domain is SAC-consistent. We have adapted MSAC3 for maintaining existential SAC. We call it MESAC3. We ran it on the instances of Table 8. It has approximately the same performance as MSAC3 on the satisfiable instances. But on unsatisfiable ones, it is significantly faster than MSAC3. For instance, on the *scen11-f10* instance, it explores 26 nodes in 38.3 s for solving the problem. This is 20 times faster than MSAC3 for only three times more nodes. It solves *scen11-f8* in 290 s and 213 nodes.

Of course, the results given in this subsection are extremely preliminary. They just give a first idea of the questions that will have to be tackled. This will deserve a thorough study.

10 Conclusion

We have reviewed existing algorithms for singleton arc consistency and we have given their complexities. Two of them have a high time complexity (SAC1, SAC2) and one has a high space complexity (SAC-Opt). Our first contribution, SAC-SDS, is a trade-off between these two extremes. Like SAC-Opt, SAC-SDS stores the domains of the subproblems created for checking singleton arc consistency of values. Unlike SAC-Opt, it does not store all the data structures used by arc consistency in these subproblems. This leads to a higher time complexity and a lower space complexity than SAC-Opt. The data stored by SAC-SDS are enough to obtain a lower time complexity than SAC1 and SAC2. Our second contribution, SAC3, is an original approach that mixes inference and search to enforce singleton arc consistency. A side effect of building a branch while maintaining arc consistency is that all its values except the last are guaranteed to be singleton arc-consistent. Such an approach has several advantages. We can benefit from the guidance of standard heuristics usually used during search, we can check singleton arc consistency quickly when the network is already singleton arc-consistent, and we can find a solution 'by chance' while building a branch. The third algorithm presented in this paper, SAC3-SDS, combines our two first contributions in a single algorithm. SAC3-SDS builds branches, like SAC3, and SAC3-SDS stores the domains of the subproblems built, like SAC-SDS. This paper also contains an extensive experimental evaluation. This shows the benefit of our contributions compared to existing algorithms for SAC. Overall, SAC3-SDS seems to be the more frequent winner thanks to its combination of the good characteristics of SAC-SDS and SAC3.

Acknowledgements The first author was supported by the ANR project ANR-06-BLAN-0383-02. The second and fourth authors were supported by the CNRS and by the "IUT de Lens".

References

1. Apt, K. R. (2003). *Principles of constraint programming*. Cambridge: Cambridge University Press.

2. Bartak, R., & Erben, R. (2004). A new algorithm for singleton arc consistency. In *Proceedings of FLAIRS'04* (pp. 257–262).
3. Bessiere, C. (1994). Arc consistency and arc consistency again. *Artificial Intelligence*, 65, 179–190.
4. Bessiere, C., & Debruyne, R. (2004). Theoretical analysis of singleton arc consistency. In *Proceedings of ECAI'04 workshop on modelling and solving problems with constraints* (pp. 20–29).
5. Bessiere, C., & Debruyne, R. (2005). Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05* (pp. 54–59).
6. Bessiere, C., & Debruyne, R. (2008). Theoretical analysis of singleton arc consistency and its extensions. *Artificial Intelligence*, 172(1), 29–41.
7. Bessiere, C., & Régin, J. (2001). Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01* (pp. 309–315).
8. Bessiere, C., Régin, J. C., Yap, R., & Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2), 165–185.
9. Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04* (pp. 146–150).
10. Chmeiss, A., & Sais, L. (2000). About the use of local consistency in solving CSPs. In *Proceedings of ICTAI'00* (pp. 104–107).
11. Debruyne, R., & Bessiere, C. (1997a). From restricted path consistency to max-restricted path consistency. In *Proceedings of CP'97* (pp. 312–326).
12. Debruyne, R., & Bessiere, C. (1997b). Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97* (pp. 412–417).
13. Debruyne, R., & Bessiere, C. (2001). Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14, 205–230.
14. Dechter, R. (2003). *Constraint processing*. San Francisco: Morgan Kaufmann.
15. Frost, D., Dechter, R., Bessiere, C., & Régin, J. C. (1996). *Random uniform CSP generators*. <http://www.lirmm.fr/~bessiere/generator.html>.
16. Lecoutre, C. (2008). Optimization of simple tabular reduction for table constraints. In *Proceedings of CP'08* (pp. 128–143).
17. Lecoutre, C. (2009). *Constraint networks: Techniques and algorithms*. New York: ISTE/Wiley.
18. Lecoutre, C., & Cardon, S. (2005). A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05* (pp. 199–204).
19. Lecoutre, C., & Hemery, F. (2007). A study of residual supports in arc consistency. In *Proceedings of IJCAI'07* (pp. 125–130).
20. Lecoutre, C., & Prosser, P. (2006). Maintaining singleton arc consistency. In *Proceedings of CPAI'06 workshop held with CP'06* (pp. 47–61).
21. Mackworth, A. K. (1977a). Consistency in networks of relations. *Artificial Intelligence*, 8(1), 99–118.
22. Mackworth, A. K. (1977b). On reading sketch maps. In *Proceedings of IJCAI'77* (pp. 598–606).
23. Mohr, R., & Henderson, T. C. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28, 225–233.
24. Mohr, R., & Masini, G. (1988). Good old discrete relaxation. In *Proceedings of ECAI'88* (pp. 651–656).
25. Prosser, P. (1996). An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81, 81–109.
26. Prosser, P., Stergiou, K., Walsh, T. (2000). Singleton consistencies. In *Proceedings of CP'00* (pp. 353–368).
27. Sabin, D., & Freuder, E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94* (pp. 10–20).
28. Sabin, D., & Freuder, E. C. (1997). Understanding and improving the MAC algorithm. In *Proceedings of CP'97* (pp. 167–181).
29. Ullmann, J. R. (2007). Partition search for non-binary constraint satisfaction. *Information Science*, 177, 3639–3678.
30. Zhang, Y., & Yap, R. (2001). Making AC3 an optimal algorithm. In *Proceedings of IJCAI'01* (pp. 316–321).