A Greedy Approach to Establish Singleton Arc Consistency

Christophe Lecoutre and **Stéphane Cardon**
CRIL-CNRS FRE 2499,
Université d'Artois
Lens, France
$\{lecoutre, cardon\}$@cril.univ-artois.fr

Abstract

In this paper, we propose a new approach to establish Singleton Arc Consistency (SAC) on constraint networks. While the principle of existing SAC algorithms involves performing a breadth-first search up to a depth equal to 1, the principle of the two algorithms introduced in this paper involves performing several runs of a greedy search (where at each step, arc consistency is maintained). It is then 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. Furthermore, both space and time complexities are quite competitive.

1 Introduction

Inference and search are two categories of techniques for processing constraints [Dechter, 2003]. 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 and tree clustering, 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. 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 [Sabin and Freuder, 1994]. MAC interleaves inference and search since at each step of a depth-first exploration with backtracking, a local consistency called arc consistency is maintained. However, since the introduction of stronger consistencies such as max-restricted path consistency [Debruyne and Bessière, 1997a] and singleton arc consistency [Debruyne and Bessière, 1997b], one issue has been the practical interest of utilizing such consistencies, instead of arc consistency, before or during search.

There is a recent focus about singleton consistencies, and more particularly about SAC (Singleton Arc Consistency), as illustrated by recent works of [Debruyne and Bessière, 1997b; Prosser *et al.*, 2000; Bartak, 2004; Bessière and Debruyne, 2004; 2005]. A constraint network is singleton arc consistent iff any singleton check does not show unsatisfiability, i.e., iff after performing any variable assignment, enforcing arc consistency on the resulting network does not entail a domain wipe-out.

In this paper, we propose two new algorithms, denoted SAC-3 and SAC-3+, to establish singleton arc consistency. While SAC algorithms introduced so far perform a breadth-first search up to a depth equal to 1, these two new algorithms perform several runs of a greedy search (where at each step, arc consistency is maintained). However, unlike SAC-3+, SAC-3 does not record the context of performed runs.

We have identified several advantages to adopt this approach:

- 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 the consistency,
- time complexity of both algorithms is quite competitive.

More precisely, the good space complexity of both algorithms allows to use them on large constraint networks. In particular, SAC-3 admits the space complexity of the underlying arc consistency algorithm. Next, when a greedy search maintaining arc consistency is used, we naturally benefit from the incrementality of arc consistency, i.e., the fact that iteratively establishing arc consistency on a more and more reduced search space is less penalizing than repeatedly establishing it on the original search space. Besides, when a dead-end is encountered during a greedy search, a no-good can be recorded and/or the origin of the failure taken into account. Also, some solutions may be found by the algorithm. Finally, if the instance contains a large under-constrained part, a very efficient time complexity can be expected.

2 Preliminaries

A constraint network consists of a finite set of variables such that each variable X has an associated domain dom(X) denoting the set of values allowed for X, and a finite set of constraints such that each constraint C has an associated relation rel(C) denoting the set of tuples allowed for the variables vars(C) involved in C. A solution to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied. A constraint network is said to be satisfiable if it admits at least a solution.

The Constraint Satisfaction Problem (CSP), whose task is to determine whether or not a given constraint network is satisfiable, is NP-complete. A constraint network is also called CSP instance. 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. Usually, constraint propagation algorithms, which are based on some constraint network properties such as arc consistency, remove some values which can not occur in any solution.

Definition 1 *Let $P = (\mathscr{X}, \mathscr{C})$ be a constraint network, $C \in \mathscr{C}$, $X \in vars(C)$ and $a \in dom(X)$. (X, a) is said to be arc consistent wrt C iff there exists a support of (X, a) in C, i.e., a tuple $t \in rel(C)$ such that $t[X] = a$. P is said to be arc consistent iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and $\forall C \in \mathscr{C}$, $\forall X \in vars(C)$, $\forall a \in dom(X)$, (X, a) is arc consistent wrt C.*

AC(P) will denote the constraint network obtained after enforcing Arc Consistency (AC) on a given constraint network P. AC(P) is such that all values of P that are not arc consistent have been removed. Note that a value will usually refer to a pair (X,a) with $X \in \mathscr{X}$ and $a \in dom(X)$. If there is a variable with an empty domain in AC(P), denoted AC(P) = $\perp$, then P is clearly unsatisfiable. $P|_S$ with $S \subset \{X = a | X \in \mathscr{X} \wedge a \in dom(X)\}$ is the constraint network[1] obtained from P by restricting the domain of X to the singleton $\{a\}$ for any variable assignment $X = a \in S$.

Definition 2 *Let $P = (\mathscr{X}, \mathscr{C})$ be a constraint network, $X \in \mathscr{X}$ and $a \in dom(X)$. (X, a) is said to be singleton arc consistent iff AC($P|_{X=a}$) $\neq \perp$. P is said to be singleton arc consistent iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and $\forall a \in dom(X)$, (X, a) is singleton arc consistent.*

$\mathscr{X}$ will be called the domain of $P = (\mathscr{X}, \mathscr{C})$. We will note $(X, a) \in P$ (respectively, $(X, a) \notin P$) iff $X \in \mathscr{X}$ and $a \in dom(X)$ (respectively, $a \notin dom(X)$).

3 Overview of SAC algorithms

The first algorithm that has been proposed to establish singleton arc consistency is called SAC-1 [Debruyne and Bessière, 1997b]. The principle of this algorithm is to check the singleton arc consistency of all variables whenever a value is detected singleton arc inconsistent and then removed. Worst-case space and time complexities of SAC-1 are respectively O(md) and O(mn^2d^4) where n denotes the number of variables, d the size of the largest domain and m the number of constraints.

A second algorithm, denoted SAC-2, has been proposed by [Bartak, 2004]. The idea is to check (again) the singleton arc consistency of a value (Y,b) after the removal of a value (X,a) only if (X,a) does not support (Y,b), i.e., does not belong to AC($P|_{Y=b}$). Hence, this algorithm allows avoiding usefulness singleton checks by recording, for each value, the set of values supported by it. As expected and supported by the experimentation of [Bartak, 2004], SAC-2 offers a significant improvement of practical time efficiency with respect to SAC-1. Worst-case space and time complexities of SAC-2 are respectively O(n^2d^2) and O(mn^2d^4).

[Bessière and Debruyne, 2004] have remarked that SAC-2 does not present any improvement in terms of worst-case time complexity because whenever the singleton arc consistency of a value (X,a) must be checked again, one has to perform the arc consistency enforcement on $P|_{X=a}$ from scratch. In other words, SAC-2 does not exploit the incrementality of arc consistency. An arc consistency algorithm is said incremental if its worst-case time complexity is the same when it is applied one time on a given network P and when it is applied up to nd times on P where, between two consecutive executions, at least one value has been deleted. All current arc consistency algorithms are incremental. To benefit from the incrementality of arc consistency, [Bessière and Debruyne, 2004; 2005] have proposed a new algorithm, SAC-OPT, that duplicates the original constraint network into nd dedicated constraint networks, one for each value (X,a) of the instance. Simply, whenever the singleton consistency of a value (X,a) must be checked, the dedicated constraint network is used. Worst-case space and time complexities of SAC-OPT are respectively O(mnd^2) and O(mnd^3) which is the best time complexity that can be expected from an algorithm enforcing singleton arc consistency [Bessière and Debruyne, 2004].

Finally, from the observation that a space complexity in O(mnd^2) prevents the use of SAC-OPT on large constraint networks, [Bessière and Debruyne, 2005] have proposed another algorithm called SAC-SDS which represents a trade-off between time and space. With respect to each value, only the domain (called SAC-support) is recorded as well as a propagation list used for arc consistency. In return, the data structures required to establish arc consistency are no more dedicated but shared. An experimental study on random instances have highlighted the good performance of this algorithm. Worst-case space and time complexities of SAC-SDS are respectively O(n^2d^2) and O(mnd^4).

4 SAC-3

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 checks. In other words, we can try to build less branches of greater sizes using a greedy

[1]It will be assumed that $X = a \in S \wedge Y = b \in S \Rightarrow X \neq Y$ and, for convenience, $P|_{\{X=a\}}$ will be simply denoted by $P|_{X=a}$.

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 if found, either the branch is of size 0 and a value is detected inconsistent, or all but last variable assignments correspond to singleton arc consistent values. This last statement relies on Proposition 1.

Proposition 1 *Let $P = (\mathscr{X}, \mathscr{C})$ be a constraint network and let $S \subset \{X = a | X \in \mathscr{X} \wedge a \in dom(X)\}$. If AC($P|_S$) $\neq \perp$ then any pair (X,a) such that $X \in \mathscr{X}$ and dom(X) = $\{a\}$ in AC($P|_S$) is singleton arc consistent.*

Proof. If AC($P|_S$) $\neq \perp$ then, clearly any element $X = a \in S$ is singleton arc consistent. It is a consequence of the monotony of arc consistency. One should observe that we can also find some values (Y,b) such that $Y \in \mathscr{X}$ and dom(Y) = $\{b\}$ in AC($P|_S$) with $Y = b \notin S$. These values are also clearly singleton arc consistent. □

As mentioned in the proof above, some values can be detected singleton arc consistent while checking the singleton arc consistency of another one(s). Proposition 1 can then be seen as a generalization of Property 2 in [Chmeiss and Sais, 2000], and is also related to the exploitation of singleton-valued variables in [Sabin and Freuder, 1997].

Although the primary goal of our approach was to exploit incrementality of arc consistency, other nice features have been observed. Indeed, using a greedy search, one may find solutions and one can learn from conflicts by recording no-goods or weighting failure culprits.

Below, we give the description of a first algorithm that uses a greedy search in order to establish singleton arc consistency. The description is given in the context of using an underlying coarse-grained arc consistency algorithm (e.g. AC3 [Mackworth, 1977] or AC3.2/3.3 [Lecoutre *et al.*, 2003]) with a variable-oriented propagation scheme.

First, let us introduce some notations. If $P = (\mathscr{X}, \mathscr{C})$, then AC($P$,$Q$) with $Q \subseteq \mathscr{X}$ means enforcing arc consistency on P from the given propagation set Q. For a description of AC, see, for instance, the function $propagateAC$ in [Bessière and Debruyne, 2005]. Q_{sac} is the set of values whose singleton arc consistency must be checked. A branch corresponds to a set of values that have been assigned. For any set of values $S \subseteq \{(X, a) \mid X \in \mathscr{X} \wedge a \in dom(X)\}$, vars($S$) = $\{X | (X, a) \in S\}$. Finally, an instruction of the form $P_{before} \leftarrow P$ should not be systematically considered as a duplication of the problem. Most of the time, it correspond to store or restore the domain of a network (and the structures of the underlying arc consistency algorithm).

Algorithm 2 starts by enforcing arc consistency on the given network. Then, all values are put in the structure Q_{sac} and in order to check their singleton arc consistency, successive branches are built. The process continues until a fix-point is reached. Algorithm 1 allows building a branch by performing successive variable assignments while maintaining arc consistency (line 6). When an inconsistency is detected for a non empty branch, one has to put back the last value in Q_{sac} (line 12) since we have no information about the singleton arc consistency or inconsistency of this value. If the branch is empty, we have to manage the removal of a value and to reestablish arc consistency (lines 16 to 19). Note that

Algorithm 1 buildBranch()

```
1: br ← ∅
2: P_before ← P
3: consistent ← true
4: repeat
5:    pick and delete (X,a) ∈ Q_sac s.t. X ∉ vars(br)
6:    P ← AC(P|_{X=a},{X})
7:    if P ≠ ⊥ then
8:       add (X,a) to br
9:    else
10:      consistent ← false
11:      if br ≠ ∅ then
12:         add (X,a) to Q_sac
13:   end if
14: until not consistent ∨ vars(Q_sac) − vars(br) = ∅
15: P ← P_before
16: if br = ∅ then
17:    remove a from dom(X)
18:    P ← AC(P,{X})
19:    Q_sac ← Q_sac − {(Y,b)|(Y,b) ∈ dom(P_before) − dom(P)}
20: end if
```

Algorithm 2 SAC-3($P = (\mathscr{X}, \mathscr{C})$: CSP)

```
1: P ← AC(P,𝒳)
2: repeat
3:    P_before ← P
4:    Q_sac ← {(X,a) | X ∈ 𝒳 ∧ a ∈ dom(X)}
5:    while Q_sac ≠ ∅ do
6:       buildBranch()
7: until P = P_before
```

no inconsistency is detected when a solution is found or when there is no way of extending the current branch. Finally, in order to maximally benefit from the incrementality of arc consistency, we have to build branches as long as possible. Hence (although not indicated in the algorithm), it is important to select first values (X,a) $\in Q_{sac}$ such that $a \in dom(X)$.

Proposition 2 *SAC-3 is a correct algorithm with a worst-case space complexity in O(md) and a time complexity in O(bmd^2) where b denotes the number of branches built by SAC-3.*

Proof. Correctness results from Proposition 1. If SAC-3 uses an optimal coarse-grained arc consistency algorithm such as AC3.2, then the overall space complexity is O(md) since space complexity of AC3.2 is O(md), the data structure Q_{sac} is O(nd) and each branch built is O(n). The overall time complexity is O(bmd^2) since, due to incrementality, each branch built by the algorithm is O(md^2). □

Remark that b must include the "empty" branches that correspond to the detection of inconsistent SAC values. With respect to already singleton arc consistent constraint networks, Corollary 1 indicates that SAC-3 can outperform SAC-OPT and SAC-SDS (admitting then a time complexity in O(mnd^3)). More interestingly, it suggests that SAC-3 can outperform SAC-OPT and SAC-SDS on structured (not necessarily singleton arc consistent) instances that contain large under-constrained parts as can be expected in real-world applications.

Corollary 1 *SAC-3 admits a worst-case time complexity in O(mn^2d^4) but, when applied to an already singleton arc consistent constraint network, SAC-3 admits a best-case*[2] *time complexity in O(md^3) and a worst-case time complexity in O(mnd^3).*

Proof. In the worst-case, $b = \frac{n^2d^2+nd}{2}$, hence, we obtain O(mn^2d^4). When applied to an already singleton arc consistent constraint network, the best and worst cases correspond to branches of maximum size and of size 1 (1 consistent assignment followed by an inconsistent one), respectively. We have then respectively $b = d$ (all branches delivering a solution) and $b = nd$ branches. □

5 SAC-3+

It is possible to improve the behaviour of the algorithm SAC-3 by recording the domain of the constraint networks obtained after each greedy run, that is to say, for each branch. When a value is removed, it is then possible to determine which previously built branches must be reconsidered. Indeed, if a removed value does not support a branch, i.e. does not belong to the domain associated with the branch, all values of the branch remain singleton arc consistent. On the other hand, if it 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, SAC-3+ exploits incrementality as SAC-SDS does.

In order to manage domain and propagation of constraint networks corresponding to branches, we consider two arrays denoted $P[]$ and $Q[]$. For a given branch br, $P[br]$ corresponds to the constraint network associated with the branch br (in fact, we only need to record the domain of the constraint network) whereas $Q[br]$ contains the variables that have lost some value(s) and that should be considered when re-establishing arc consistency.

After enforcing arc consistency on the given network, Algorithm 6 builds successive branches by calling the function $buildBranch+$. Once the singleton arc consistency of all values of Q_{sac} have been tested, we have to check the validity of the branches that have been built and recorded in brs by a call to the function $checkBranches$ since some values may have been deleted after a branch has been built. For each branch br, we re-establish arc consistency on $P[br]$ (line 2 of Algorithm 5) and in case of a domain wipe-out, we delete this branch and update Q_{sac} (lines 4 and 5).

Algorithm 4 differs from Algorithm 1 on two aspects. First, we need to record the domain of the constraint network corresponding to the branch that is built (line 22) and add this branch to brs (line 23). Note that if the last variable assignment entails a domain wipe-out, P_{store} is not updated (line 9). For the implementation, $P[br]$ can be directly set (backtracking one step if necessary) without any duplication of domain. Second, after re-establishing arc consistency (line 19), all values that have been removed including the singleton arc inconsistent one (line 18) must be taken into account in order

[2] Even if the worst-case time complexity of the underlying arc consistency algorithm is considered.

Algorithm 3 update(set : Set of Values)

```
1: Q_sac ← Q_sac − set
2: for each br ∈ brs do
3:   for each (X,a) ∈ set do
4:     if (X,a) ∈ P[br] then
5:       remove (X,a) from P[br]
6:       add X to Q[br]
7:     endif
```

Algorithm 4 buildBranch+()

```
 1: br ← ∅
 2: P_before ← P
 3: consistent ← true
 4: repeat
 5:   pick and delete (X,a) ∈ Q_sac s.t. X ∉ vars(br)
 6:   P ← AC(P|_{X=a},{X})
 7:   if P ≠ ⊥ then
 8:     add (X,a) to br
 9:     P_store ← P
10:   else
11:     consistent ← false
12:     if br ≠ ∅ then
13:       add (X,a) to Q_sac
14:   end if
15: until not consistent ∨ vars(Q_sac) − vars(br) = ∅
16: P ← P_before
17: if br = ∅ then
18:   remove a from dom(X)
19:   P ← AC(P,{X})
20:   update({(Y,b)|(Y,b) ∈ dom(P_before) − dom(P)})
21: else
22:   P[br] ← P_store
23:   add br to brs
24: end if
```

Algorithm 5 checkBranches()

```
1: for each branch br ∈ brs do
2:   P[br] ← AC(P[br], Q[br])
3:   if P[br] = ⊥ then
4:     Q_sac ← Q_sac ∪ br
5:     remove br from brs
6:   end if
7: end for
```

Algorithm 6 SAC-3+(P = ($\mathscr{X}$, $\mathscr{C}$) : CSP)

```
1: P ← AC(P,𝒳)
2: brs ← ∅
3: Q_sac ← {(X,a) | X ∈ 𝒳 ∧ a ∈ dom(X)}
4: while Q_sac ≠ ∅ do
5:   while Q_sac ≠ ∅ do
6:     buildBranch+()
7:   checkBranches()
8: end while
```

to update the state of all branches (line 20). For each branch (line 2 of Algorithm 3), we have to remove these values (line 5) and update the propagation list (line 6).

Proposition 3 *SAC-3+ is a correct algorithm with a space complexity in O($b_{max}nd + md$) and a time complexity in O(bmd^2) where b_{max} denotes the maximum number of branches recorded by SAC-3+ and b denotes the number of times a branch is built or checked by SAC-3+.*

Proof. Correctness comes from Proposition 1 and the fact that once, the singleton arc consistency of all values have been checked and some branches recorded, one verify that the property still holds by calling $checkBranches$. In addition to the space requirement in O(md) of the underlying optimal coarse-grained arc consistency algorithm, it is necessary to record the domain of the constraint networks corresponding to the valid branches that have been built. As recording a domain is in O(nd), we obtain O($b_{max}nd + md$). □

Remark that Corollary 1 also holds for SAC-3+. However, one should be optimistic about the average time complexity of this algorithm since it avoids building new branches when unnecessary.

6 Experiments

To prove the practical interest of the algorithms introduced in this paper, we have implemented them as well as the algorithms SAC-1 and SAC-SDS, the latter being considered as the most current efficient SAC algorithm [Bessière and Debruyne, 2005]. We have used AC3.2 [Lecoutre *et al.*, 2003] as an underlying arc consistency algorithms. We have conducted an experimentation on a PC Pentium IV 2,4GHz 512Mo under Linux with respect to different classes of random, academic and real-world instances. Performances have been measured in terms of the number of singleton arc consistency checks (#scks) and the cpu time in seconds (cpu). For information, is also given, for each instance, the number (#×) of values removed by any SAC algorithm (when #×=0, it means that the instance is initially singleton arc consistent).

First, we have experimented the two classes of random binary CSP instances introduced in [Bessière and Debruyne, 2005]. However, due to lack of space, we only present the figure depicting results about the class (100,20,0.05,t) corresponding to sparse constraint networks with 100 variables, 20 values per domain and a density of 0.05 (i.e., 248 constraints). t denotes the constraint tightness, i.e., the proportion of unallowed tuples in the relations associated with the constraints.

In Figure 1, we can observe that when $t < 0.6$ (the beginning of a phase transition), SAC-3 and SAC-3+ have the same behaviour and outperform SAC-1 and SAC-SDS. In the phase transition, SAC-3 and SAC-SDS respectively become the worst and the best approaches. For complete networks of the class (100,20,1,t), similar results (not depicted here) are obtained, and at the pic of difficulty, SAC-SDS (181 s) is about three times more efficient than SAC-1 (478 s) and SAC-3 (553 s) and two times more efficient than SAC-3+ (307 s). It is not really a surprise since the generated instances have no structure, which corresponds to the worst-case for SAC-3 and SAC-3+ as the average size of the branches that are built (at the critical point) is quite small (≈ 3).

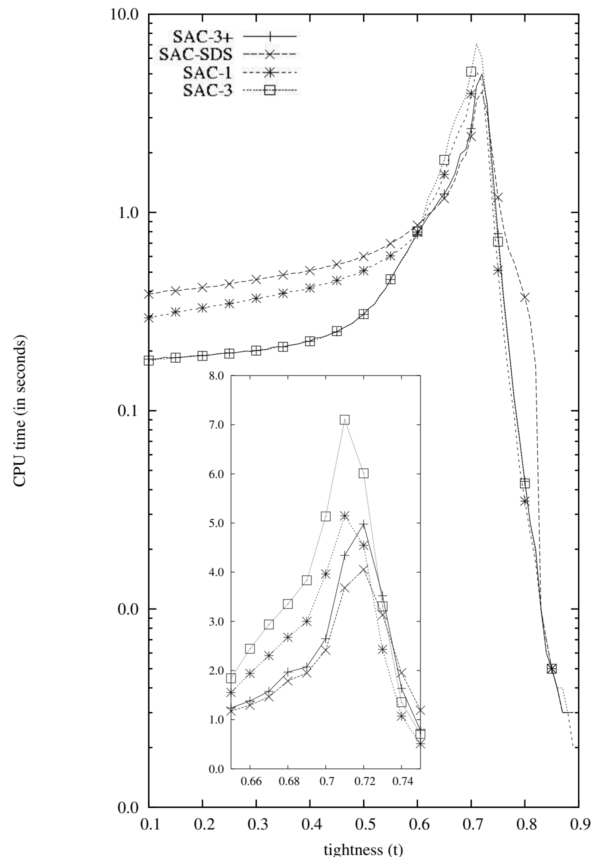

Figure 1: mean cpu time on 50 random instances of class (100,20,0.05,t) at each value of t

Next, we have dealt with the following academic instances:

- two chessboard coloration instances, denoted cc-20-2 and cc-20-3, involving quaternary constraints,
- two Golomb ruler instances, denoted gr-34-8 and gr-34-9, involving binary and ternary instances,
- two prime queen attacking instances, denoted qa-5 and qa-6, involving only binary constraints.

Table 1 shows that SAC-3, and especially SAC-3+, have on some instances a much better behaviour than SAC-1 and SAC-SDS. Roughly speaking, it can explained by the fact that such instances have some regular structure.

Next, we have tested real-world instances, taken from the FullRLFAP archive, which contains instances of radio link frequency assignment problems. Table 2 shows the results obtained on some representative instances. As expected, on already singleton arc consistent instances ($scen02$, $graph14$), a significant improvement is obtained. But it is also true for the other instances as they contain large under-constrained parts. It clearly appears that, on such structured instances, using SAC-3 and SAC-3+ is the best approach, especially as SAC-SDS has been out of memory on some instances.

It is interesting to note that SAC-3 and SAC-3+ can be much faster than SAC-1 and SAC-SDS even if the number of singleton checks (see for example, cc-20-2 and $scen02$) is similar. It results from the exploitation of the incrementality of arc consistency (when building branches). Another point

		SAC-1	SAC-SDS	SAC-3	SAC-3+
cc-20-2	cpu	14.44	14.43	3.46	3.48
(#×=0)	#scks	800	800	819	819
cc-20-3	cpu	22.61	22.71	7.01	7.02
(#×=0)	#scks	1200	1200	1200	1200
gr-34-8	cpu	66.08	17.43	38.28	18.62
(#×=351)	#scks	6335	3340	5299	1558
gr-34-9	cpu	111.44	31.29	91.50	32.03
(#×=513)	#scks	8474	4720	11017	2013
qa-5	cpu	2.47	2.50	.93	.96
(#×=9)	#scks	622	622	732	732
qa-6	cpu	27.55	14.34	8.23	4.38
(#×=48)	#scks	2523	1702	2855	1448

Table 1: Academic instances

		SAC-1	SAC-SDS	SAC-3	SAC-3+
$scen02$	cpu	20.97	20.73	4.09 (16)	4.08 (16)
(#×=0)	#scks	8004	8004	8005	8005
$scen05$	cpu	11.79	20.03	1.55 (1)	1.87
(#×=13814)	#scks	6513	4865	4241	2389
$graph03$	cpu	215.88	136.93	74.97	39.10
(#×=1274)	#scks	20075	17069	22279	8406
$graph10$	cpu	1389.59	-	675.64	349.53
(#×=2572)	#scks	74321	-	82503	29398
$graph14$	cpu	154.16	-	31.17 (10)	32.72 (10)
(#×=0)	#scks	36716	-	36719	36719

Table 2: RLFAP instances

to be mentioned is that some solutions (enclosed in brackets near cpu time) have been found during the inference process on some instances. For example, 16 solutions have been found on $scen02$ by SAC-3 and SAC-3+.

Finally, we have experimented some realistic scheduling problems. We have selected 2 representative instances from the set of 60 job shop instances proposed by [Sadeh and Fox, 1996]. It is quite interesting to note (see Table 3) that SAC-3 and SAC-3+ have found solution(s) to these two instances quite more efficiently than MAC (when run to find one solution). It can be explained by the restart aspect of both algorithms.

		SAC-1	SAC-SDS	SAC-3	SAC-3+	MAC
js-1	cpu	29.61	29.58	18.17 (4)	19.04 (4)	181.68
(#×=0)	#scks	5760	5760	6029	6029	-
js-2	cpu	40.40	38.58	31.13 (1)	31.07 (1)	211.52
(#×=0)	#scks	6315	6315	6718	6718	-

Table 3: Job-shop instances

7 Conclusion

Establishing singleton arc consistency can be justified if it is effective (allows some filtering). On the contrary, applying a SAC algorithm on a constraint network that is already singleton arc consistent is a problem as it may involve a large waste of time. The two algorithms, SAC-3 and SAC-3+, introduced in this paper combine inference and search and can be understood as an answer to this problem. Indeed, when an instance is under-constrained or, more generally, contains easy large parts, as can be expected in real-world applications, exploiting search during inference can pay off. It has been confirmed by our experimentation. Besides, one should have noted the limited space requirement of both algorithms which makes them applicable on large constraint networks.

We believe that this approach deserves further investigation in order to determine if it could be applied to other local consistencies and, to what extent, maintaining SAC during search could be a viable alternative to MAC.

Acknowledgments

This paper has been supported by the CNRS, the “programme COCOA de la Région Nord/Pas-de-Calais” and by the “IUT de Lens”.

References

[Bartak, 2004] R. Bartak. A new algorithm for singleton arc consistency. In *Proceedings of FLAIRS'04*, 2004.

[Bessière and Debruyne, 2004] C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency. In *Proceedings of ECAI'04 workshop on modeling and solving problems with constraints*, pages 20–29, 2004.

[Bessière and Debruyne, 2005] C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, 2005.

[Chmeiss and Sais, 2000] A. Chmeiss and L. Sais. About the use of local consistency in solving CSPs. In *Proceedings of ICTAI'00*, pages 104–107, 2000.

[Debruyne and Bessière, 1997a] R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proc. of CP'97*, pages 312–326, 1997.

[Debruyne and Bessière, 1997b] R. Debruyne and C. Bessière. Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.

[Dechter, 2003] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[Lecoutre *et al.*, 2003] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.

[Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:118–126, 1977.

[Prosser *et al.*, 2000] P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proceedings of CP'00*, pages 353–368, 2000.

[Sabin and Freuder, 1994] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the PPCPA'94*, 1994.

[Sabin and Freuder, 1997] D. Sabin and E. Freuder. Understanding and improving the MAC algorithm. In *Proceedings of CP'97*, 1997.

[Sadeh and Fox, 1996] N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86:1–41, 1996.