Optimal and Suboptimal Singleton Arc Consistency Algorithms

Christian Bessiere[1] and Romuald Debruyne[2]

[1] LIRMM-CNRS, 161 rue Ada F-34392 Montpellier Cedex 5
bessiere@lirmm.fr
[2] École des Mines de Nantes, 4 rue Alfred Kastler, F-44307 Nantes Cedex 3
romuald.debruyne@emn.fr

Abstract. Singleton arc consistency (SAC) is a local consistency that ensures that a constraint network can be made arc consistent after any assignment of a value to a variable. A first contribution of this paper is to show experimentally that the optimal-time algorithm proposed in [5] (which was analyzed only theoretically) is efficient in practice compared to previous SAC algorithms. However, it can be costly in space on large problems, even with the small improvements we propose at the beginning of this paper. To reduce this space consumption, we propose another SAC algorithm requiring less space but no longer optimal in time. An experimental study on random problems highlights the good performance of this second algorithm.

1 Introduction

Ensuring that a given local consistency does not lead to a failure when we enforce it after having assigned a variable to a value is a common idea in constraint solving. It has been applied (sometimes under the name 'shaving') in constraint problems with numerical domains by limiting the assignments to bounds in the domains and ensuring that bounds consistency does not fail [13]. In SAT, it has been used as a way to compute more accurate heuristics for DPLL [10, 14]. Finally, in constraint satisfaction problems (CSPs), it has been proposed and studied under the name Singleton Arc Consistency (SAC) ([9, 19]).

Some nice properties give to SAC a real advantage over the other local consistencies enhancing the ubiquitous arc consistency. Its definition is much simpler than restricted path consistency [3], max-restricted-path consistency [8], or other exotic local consistencies, and its operational semantics can be understood by a non-completely-expert of the field. Enforcing it only removes values in domains, and thus does not change the structure of the problem, as opposed to path consistency [17], k-consistency [11], etc. Finally, implementing it can be done simply on top of any AC algorithm.

Non optimal SAC algorithms were proposed in [9] and [1] while an optimal one has recently been described in [5]. In Section 3 of this paper, we rewrite the algorithm proposed in [5] in a slight different way that does not change its worst-case time and space complexities while improving slightly its practical

performance. We call it SAC-Opt. However, the optimal time complexity is kept at the cost of a high space complexity that prevents the use of this algorithm on large problems. We then propose in Section 4 another SAC algorithm, SAC-SDS, with a better worst-case space complexity but no longer optimal in time. Nevertheless, its time complexity remains better than the other SAC algorithms proposed in the past. The experiments presented in Section 5 highlight the good performance of both SAC-Opt and SAC-SDS.

2 Preliminaries

A finite *constraint network* P consists of a finite set of n variables $X = \{i, j, \ldots\}$, a set of domains $D = \{D_i, D_j, \ldots\}$, where the domain D_i is the finite set of values that variable i can take, and a set of constraints $C = \{c_1, \ldots, c_m\}$. Each constraint c_i is defined by the ordered set $var(c_i)$ of the variables it involves, and a set $sol(c_i)$ of allowed combinations of values. An assignment of values to the variables in $var(c_i)$ *satisfies* c_i if it belongs to $sol(c_i)$. A *solution* to a constraint network is an assignment of a value from its domain to each variable such that every constraint in the network is satisfied. We will use c_{ij} to refer to $sol(c)$ when $var(c) = (i, j)$. $\Phi(P)$ denotes the network obtained after enforcing Φ-consistency on P.

Definition 1 *A constraint network $P = (X, D, C)$ is said to be* ***Φ-inconsistent*** *iff $\Phi(P)$ has some empty domains or empty constraints.*

Definition 2 *A constraint network $P = (X, D, C)$ is* ***singleton arc consistent*** *iff $\forall i \in X, \forall a \in D_i$, the network $P|_{i=a}$ obtained by replacing D_i by the singleton $\{a\}$ is not arc inconsistent.*

3 An Optimal Algorithm for SAC

SAC-1 [9] has no data structure storing on which values rely the SAC consistency of each value. After a value removal, SAC-1 must check again the SAC consistency of all the other values.

SAC-2 [1] uses the fact that if we know that AC does not lead to a wipe out in $P|_{i=a}$ then the SAC consistency of (i, a) holds as long as all the values in $AC(P|i = a)$ are in the domain. After the removal of a value (j, b), SAC-2 checks again the SAC consistency of all the values (i, a) such that (j, b) was in $AC(P|_{i=a})$. This leads to a better average time complexity than SAC-1 but the data structures of SAC-2 are not sufficient to reach optimality since SAC-2 may waste time redoing the enforcement of AC in $P|_{i=a}$ several times from scratch.

Algorithm 1, called SAC-Opt, is an algorithm that enforces SAC in $O(end^3)$, the lowest time complexity which can be expected. (See [5]).

The idea behind such an optimal algorithm is that we don't want to do and redo (potentially nd times) arc consistency from scratch for each subproblem $P|_{j=b}$ each time a value (i, a) is found SAC inconsistent. (Which represents

Algorithm 1: `The optimal SAC algorithm`

```
procedure SAC-Opt(in P:Problem);
    /* init phase */;
 1  P ← AC(P) ; PendingList ← ∅;
    foreach (i, a) ∈ D do P_ia ← nil;
 2  foreach (i, a) ∈ D do
 3      P_ia ← P /* we copy the network and its data structures */;
 4      if not(propagateAC(P_ia, D_i \ {a})) then
 5          D ← D \ {(i, a)};
 6          propagateAC(P, {(i, a)});
            foreach P_jb ≠ nil such that (i, a) ∈ P_jb do
 7              Q_jb ← Q_jb ∪ {(i, a)};
 8              PendingList ← PendingList ∪ P_jb;
    /* propag phase */;
 9  while PendingList ≠ ∅ do
        pop P_ia from PendingList;
10      if not(propagateAC(P_ia, Q_ia)) then
11          D ← D \ {(i, a)};
            foreach (j, b) ∈ D such that (i, a) ∈ P_jb do
12              Q_jb ← Q_jb ∪ {(i, a)};
13              PendingList ← PendingList ∪ P_jb;
```

n^2d^2 potential arc consistency calls.) To avoid such costly repetitions of arc consistency calls, we duplicate the problem nd times, one for each value (i, a), so that we can benefit from the incrementality of arc consistency on each of them. An AC algorithm is called 'incremental' when its complexity on a problem P is the same for a single call or for up to nd calls, where two consecutive calls differ only by the deletion of some values from P. The generic AC algorithms are all incremental.

SAC-Opt can be decomposed in several sequential steps. In the following, `propagateAC`(P, S) denotes the function that incrementally propagates in P the removal of the set S of values when an initial AC call has already been executed, initializing the data structures required by the AC algorithm in use.

First, after some basic initializations and making the problem arc consistent (line 1), the loop in line 2 duplicates nd times the arc consistent problem obtained in line 1, and propagates the removal of all the values different from a for i in each $P|_{i=a}$, denoted P_{ia} (line 4). If a subproblem P_{ia} has no arc consistent subdomain, the removal of (i, a) is propagated in P (line 6). The subproblems corresponding to the subsequent steps of the loop will benefit from this propagation because they are created by duplication of P (line 3).[3] For each already

[3] This is the main difference between SAC-Opt and the algorithm presented in [5]. SAC-Opt will thus build *less* and *smaller* subproblems.

checked subproblem P_{jb} having (i, a) in its domain, (i, a) is put in Q_{jb} for future propagation (line 7).

Once this initialization phase has finished, the removal of all SAC inconsistent value has been propagated in all the subproblems except in those in *PendingList*. Each problem P_{jb} in *PendingList* contains the removals that must be propagated in its local propagation list Q_{jb}. During the whole loop of the propagation phase, if the AC propagation of a list Q_{ia} in a subproblem P_{ia} fails (line 10), (i, a) is removed from D, and the list Q_{jb} of each subproblem P_{jb} having (i, a) in its domain is updated for a future propagation of this removal.

When *PendingList* is empty, all the removals have been propagated in the subproblems and all the values in D are SAC consistent.

Theorem 1 *SAC-Opt is a correct SAC algorithm with $O(end^3)$ optimal worst-case time complexity and $O(end^2)$ worst-case space complexity.*

Proof. (See [5].)

Remarks. We can remark that the Q lists contain values to be propagated. This is written like this because the AC algorithm chosen is not specified here, and value removal is the most accurate information we can have. If the AC algorithm chosen is AC-6 [4], AC-7 [6] , or AC-4 [16], the lists will be directly used like this. If it is AC-3 [15] or AC-2001 [7], only the variables from which the domain has changed are necessary. This last information is trivially obtained from the list of removed values.

We can also point out that if AC-3 is used, we decrease the space complexity to $O(n^2d^2)$, but time complexity increases to $O(end^4)$ since AC-3 is not optimal.

4 Losing Time Optimality to Save Space

SAC-Opt cannot be used on large constraint networks because of its $O(end^2)$ space complexity. Moreover, it seems difficult to reach optimal time complexity with smaller space requirements. Indeed, a SAC algorithm has to enforce AC in each subproblem $P|_{i=a}$ and to be optimal in time it must store sufficient data to never redo some work in a subproblem. Optimal AC algorithms use at least a space in $O(ed)$ and it seems therefore unavoidable that an optimal time SAC algorithm requires nd times more space.

In this section we propose to relax time optimality to reach a satisfactory trade-off between space and time. To avoid a too general discussion, we instantiate this idea on a AC-2001 oriented SAC algorithm. The "suboptimal" algorithm we present uses AC-2001 data structures, but the same idea could be implemented with other low-space optimal AC algorithms such as AC-6 and AC-7. The algorithm SAC-SDS ('Sharing Data Structures') tries to use the incrementality of the AC algorithms to avoid redundant work, without duplicating on each subproblem $P|_{i=a}$ the data structures required by optimal AC algorithms. This algorithm requires less space than SAC-Opt but is not optimal in time.

Algorithm 2: `The SAC-SDS algorithm`

```
procedure SAC-SDS-2001(in P: Problem);
1  P ← AC-2001(P) ; PendingList ← ∅;
2  foreach (i, a) ∈ D do
3      Support_ia^SAC ← nil ; Q_ia ← {i}; PendingList ← PendingList ∪ {(i, a)};
4  while PendingList ≠ ∅ do
       pop (i, a) from PendingList ;
       if a ∈ D_i then
5          if Support_ia^SAC = nil then Support_ia^SAC ← (D \ D_i) ∪ {(i, a)};
6          if not( propagateSubAC( (X, Support_ia^SAC, C), Q_ia)) then
7              D_i ← D_i \ {a} ;
8              updateSubproblems((i, a)) ;
9              propagateAC( (X, D, C), {i}) ;

function propagateAC(in (X, D, C): Problem, in Q: set): Boolean;
   while Q ≠ ∅ do
       pop j from Q ;
       foreach i ∈ X such that ∃C_ij ∈ C do
           foreach a ∈ D_i such that Last_ija ∉ D_j do
10             if ∃b ∈ D_j such that b > Last_ija ∧ C_ij(a, b) then
11                 Last_ija ← b /* not in propagateSubAC */;
               else
12                 D_i ← D_i \ {a} ;
                   Q ← Q ∪ {i} ;
13                 updateSubproblems((i, a)) /* not in propagateSubAC */;
           if D_i = ∅ then return false;
   return true;
procedure updateSubproblems(in (i, a): Value);
   foreach (j, b) ∈ D such that (i, a) ∈ Support_jb^SAC do
14     Support_jb^SAC ← Support_jb^SAC \ {(i, a)} ;
       Q_jb ← Q_jb ∪ {i} ;
15     PendingList ← PendingList ∪ {(j, b)} ;
```

The main idea in SAC-SDS is that for each value (i, a), we store a local propagation list and the domain of $AC(P|_{i=a})$, denoted by $Support_{ia}^{SAC}$ and called its SAC-support. Thanks to these SAC-supports, we know which values may no longer be SAC consistent after a removal. These SAC-supports are also used to follow the AC enforcement in each subproblem $P|_{i=a}$ with the domains in the state in which they were at the end of the last AC propagation.

SAC-SDS-2001 relies on AC-2001. The data structure *Last* of this algorithm will be used for the propagation of AC in P but it will also be used to help the enforcement of AC in the subproblems $P|_{i=a}$. This data structure is therefore shared since it is not duplicated while being used for achieving AC in P and all the subproblems $P|_{i=a}$.

In SAC-SDS-2001, a value (i, a) is in *PendingList* if some removals have to be propagated in $P|_{i=a}$. In such a case, Q_{ia} is a non empty list composed of all the variables j such that values in D_j have been removed since the last AC enforcement in $P|_{i=a}$. After some initializations, SAC-SDS-2001 repeatedly pops a value from *PendingList* and propagates AC in $(X, Support_{ia}^{SAC}, C)$, namely $P|_{i=a}$, since $Support_{ia}^{SAC}$ is the current domain of $P|_{i=a}$. Remark that if $Support_{ia}^{SAC} = nil$ this is the first enforcement of AC in $P|_{i=a}$ and $Support_{ia}^{SAC}$ must be initialized (line 5). If $P|_{i=a}$ is arc inconsistent, (i, a) is not SAC consistent. It is therefore removed from P (line 7) and from the subproblems (using *updateSubproblems* in line 8) before the propagation of this removal (line 9).

The function *propagateSubAC* used to propagate arc consistency in the subproblems is almost similar to *propagateAC* in AC-2001. The difference comes from line 11 where the structure *Last* is not updated. Indeed, this data structure is useful to achieve AC in the subproblems more quickly (line 10) since we know that there is no support for (i, a) on C_{ij} lower than $Last_{ija}$ in P (and so in subproblems) but since this data structure is not duplicated for each subproblem it must not be updated by *propagateSubAC*.

Obviously, while achieving AC in P using *propagateAC* the data structure *Last* is updated and the only difference with AC-2001 is the line 13 where *updateSubproblems* is used to remove the SAC inconsistent value (i, a) in all the subproblems and to update the local propagation lists for future propagation of these removals.

By using *updateSubproblems*, SAC-SDS-2001 tries to avoid redoing the same propagations in all the subproblems. Each removal of a SAC inconsistent value is first propagated in P before being propagated in the subproblems. Thanks to *updateSubproblems*, all the subproblems will benefit from the removals in P.

Theorem 2 *SAC-SDS is a correct SAC algorithm with $O(end^4)$ time complexity and $O(n^2d^2)$ space complexity.*

Proof. Correctness. Note first that the structure *Last* is updated only while achieving AC in P so that any support of (i, a) in D_j is greater than or equal to $Last_{ija}$. The domains of the subproblems being subdomains of D, any support of a value (i, a) on C_{ij} in a subproblem is also greater or equal to $Last_{ija}$. This explains that *propagateSubAC* can benefit (line 10) from the structure *Last* without losing any support.

Suppose that SAC-SDS-2001 is not sound on a problem P and let (i, a) be the first SAC consistent value it removes while it should not. If (i, a) is removed at line 7 it would be a SAC inconsistent value since only deleted values are put in the local propagation lists and by assumption any previously removed value is SAC-inconsistent. So, (i, a) is removed at line 9 because it is no longer arc consistent after the removal of some SAC inconsistent values and (i, a) is therefore not SAC consistent. So, any removed value is SAC inconsistent and SAC-SDS-2001 is sound.

Completeness comes from the fact that any deletion is propagated. After the initialization (lines 2-3), $PendingList = D$ and so, the main loop of SAC-SDS-2001 considers any subproblem $P|_{i=a}$ at least once. Each time a value (i, a)

is found SAC inconsistent in P, because $P|_{i=a}$ is arc inconsistent (line 6) or because the deletion of some SAC-inconsistent values make it arc inconsistent in P (lines 9 and 12 of *propagateAC*), (i, a) is removed from the subproblems (using *updateSubproblems* and *PendingList*) and the local propagation lists are updated for future propagation. At the end of the main loop, *PendingList* is empty, so all the removals have been propagated and for any value $(i, a) \in D$ $Support_{ia}^{SAC}$ is a non empty arc consistent subdomain of $P|_{i=a}$.

Complexity. The data structure *Last* requires a space in $O(ed)$. Each of the nd $Support_{ia}^{SAC}$ can contain nd values and there is at most n variables in the nd local propagation lists. Since $e < n^2$, the space complexity of SAC-SDS-2001 is in $O(n^2d^2)$. So, considering space requirements, SAC-SDS-2001 is similar to SAC-2 [1].

Regarding time complexity, SAC-SDS-2001 first duplicates the data structures and propagates arc consistency on each subproblem (lines 5 and 6), two tasks which are respectively in $nd \cdot nd$ and $nd \cdot ed^2$. Each value removal is propagated to all $P|_{i=a}$ problems via an update of *PendingList* and $Support_{ia}^{SAC}$ sets (lines 8 and 13). This requires $nd \cdot nd$ operations. Each subproblem can in the worst case be called nd times for arc consistency, and there are nd subproblems. The domains of each subproblem are stored so that the AC propagation is launched with the domains in the state in which they were at the end of the previous AC propagation in the subproblem. Thus, in spite of the several AC propagations on a subproblem, a value will be removed at most once and, thanks to incrementality of arc consistency, the propagation of these nd value removals is in $O(ed^3)$. (Remark that we cannot reach the optimal ed^2 complexity for arc consistency on these subproblems since we do not duplicate the data structures necessary for AC optimality.) Thus the total cost of arc consistency propagations is $nd \cdot ed^3$. The total time complexity is $O(end^4)$. □

As SAC-2, SAC-SDS performs a better propagation than SAC-1 since after the removal of a value (i, a) from D, SAC-SDS checks the arc consistency of the subproblems $P|_{j=b}$ only if they have (i, a) in their domains (and not all the subproblems as SAC-1). But this is not sufficient to have a better worst-case time complexity than SAC-1. The time complexity of SAC-2 is indeed $O(en^2d^4)$ as SAC-1. SAC-SDS improves this complexity because it does not propagate in the subproblems from scratch since the *current* domain of each subproblem is stored (using $Support^{SAC}$). Furthermore, we can expect a better average time complexity since the shared structure *Last* can reduce the number of arc consistency tests required. Finally, SAC-SDS does not duplicate data structures to test the arc consistency of a subproblem, so, no restoration of data structures is required after such a test.

5 Experimental Results

To compare the performances of the SAC algorithms, we used the random uniform constraint network generator of [12] which produces instances according to the Model B [18]. All the algorithms have been implemented in C++. SAC-1 and

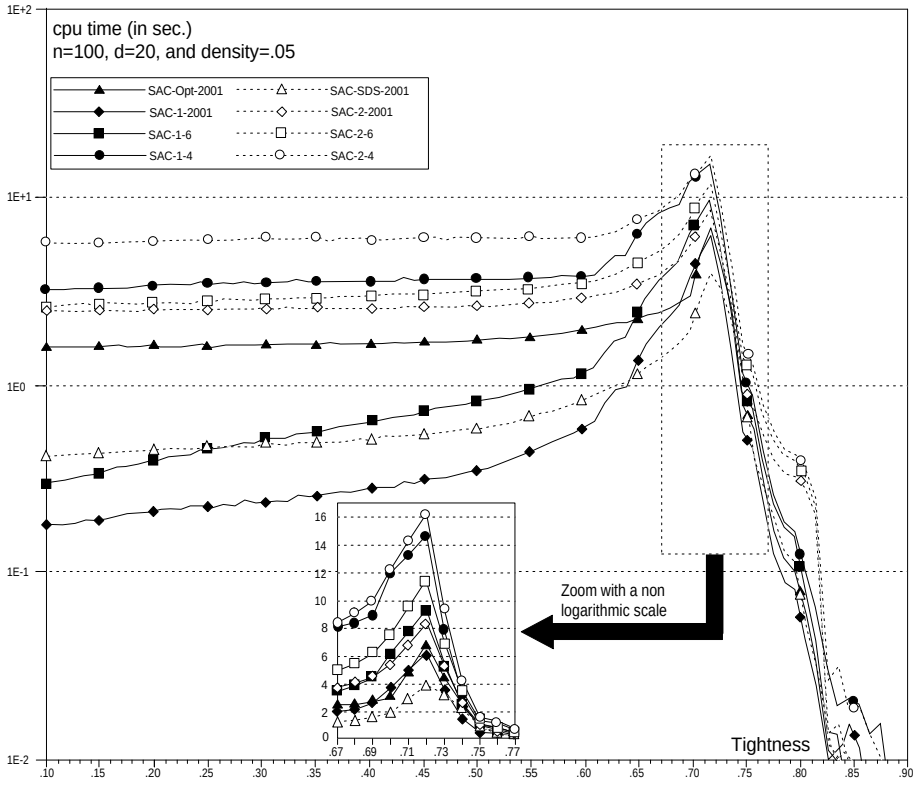

Fig. 1. cpu time results on constraint networks with n=100, d=20, and density=.05.

SAC-2 have been tested using several AC algorithms. In the following, we note AC-1-X, AC-2-X and SAC-Opt-X the versions of SAC-1, SAC-2 and SAC-Opt based on AC-X. Note that for SAC-2 the implementation of the propagation list has been done according to the recommendations made in [2, 1].

5.1 Experiments on sparse constraint networks

Fig. 1 presents cpu time performances on constraint networks having 100 variables, 20 values in each initial domain, and a density of .05. These constraint networks are relatively sparse since the variables have five neighbors on average. For each tightness, 50 instances were generated. Fig. 1 shows mean values obtained on a Pentium IV-1600 MHz with 512 Mb of memory under Windows XP.

For a tightness lower than .55, all the values are SAC consistent. On these under constrained network, the SAC algorithms check the arc consistency of each subproblem at most once. Storing the SAC-supports to enhance the propagation, as in SAC-2 and in SAC-SDS-2001, does not pay-off on these problems.

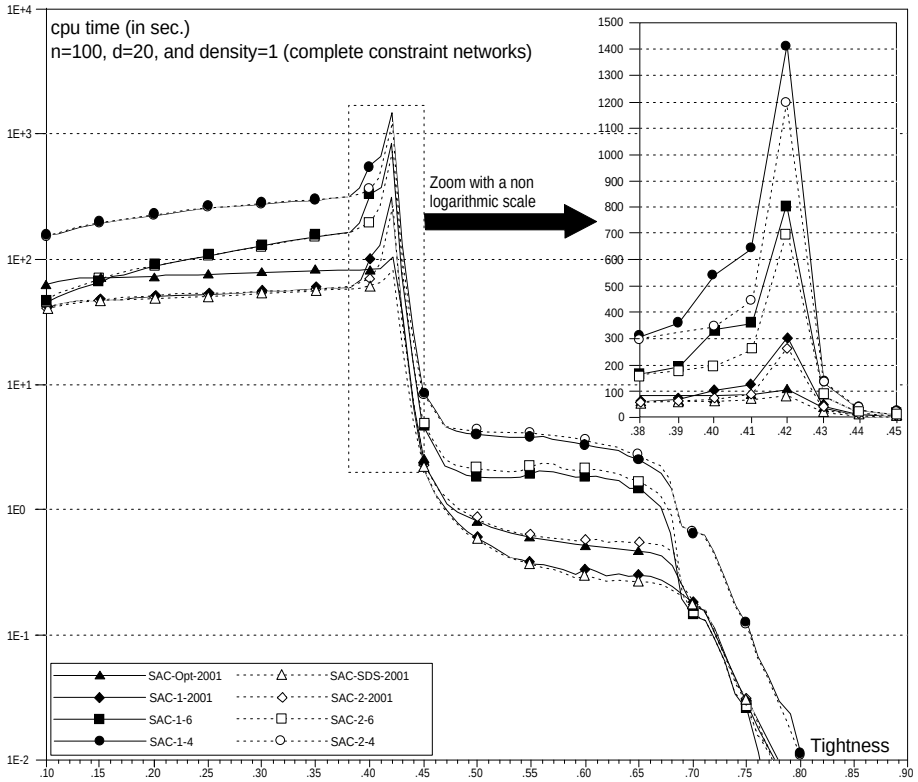

Fig. 2. cpu time results on complete constraint networks with n=100 and d=20.

A brute-force algorithm such as SAC-1 is sufficient. SAC-1-2001 shows the best performance.

On problems having tighter constraints, some SAC inconsistent values are removed and at tightness .72 we can see a peak of complexity. However, as mentioned in [2], the enhanced propagation of SAC-2 is useless on sparse constraint networks and SAC-2-X (with X$\in \{4, 6, 2001\}$) is always more expensive than SAC-1-X on the generated problems.

Around the peak of complexity, SAC-SDS-2001 is the clear winner. SAC-Opt-2001 and SAC-1-2001 are around 1.7 times slower, and all the others are between 2.1 and 10 times slower.

5.2 Experiments on dense constraint networks

We used the same computer to evaluate the performance of the SAC algorithms on complete constraint networks. For each tightness, 50 instances were generated and Fig. 2 shows mean values.

The performance of SAC-2 and SAC-1 is very close. When all the values are SAC consistent (tightness lower than .37) the additional data structure of SAC-2

is useless since there is no propagation. However the cost of building this data structure is not important compared to the overall time and the time required by SAC-2 is almost the same as SAC-1. Around the peak of complexity, SAC-2-X (with X$\in \{4, 6, 2001\}$) requires a little less time than SAC-1-X. SAC-2 has to repeatedly recheck the arc consistency of less subproblems than SAC-1 but the cost of testing a subproblem remains the same. On very tight constraints, SAC-1 requires less time than SAC-2 since the inconsistency of the problem is found with almost no propagation and building the data structure of SAC-2 is useless.

Conversely to what is supposed in [1], using AC-4 in SAC-1 (or in SAC-2) instead of AC-6 or AC-2001 is not worthwhile. The intuition was that since the data structure of AC-4 has not to be updated, the cost of its creation would be light compared to the profit we can expect. However, SAC-1-4 and SAC-2-4 are far more costly than their versions based on AC-6 or AC-2001.

The best results are obtained with SAC-Opt-2001 and SAC-SDS-2001 which are between 2.6 and 17 times faster than the others at the peak. These two algorithms have a better propagation between subproblems than SAC-1 but they also avoid some redundant work and so reduce the work performed on each subproblem.

6 Summary and Conclusion

We have presented SAC-Opt, a slightly modified version of the optimal worst-case time complexity SAC algorithm presented in [5]. However, the $O(end^2)$ space complexity of this algorithm prevents its use on large constraint networks. Therefore, we have proposed another SAC algorithm, SAC-SDS, that is not optimal in time but that requires less space than SAC-Opt. Like the optimal algorithm, SAC-SDS tries to avoid redundant work. Experiments show the good performance of these new SAC algorithms.

References

1. R. Barták. A new algorithm for singleton arc consistency. In *Proceedings FLAIRS'04*, Miami Beach, Fl, 2004. AAAI Press.
2. R. Barták and R. Erben. Singleton arc consistency revised. In *ITI Series 2003-153*, Prague, 2003.
3. P. Berlandier. Improving domain filtering using restricted path consistency. In *Proceedings IEEE-CAIA'95*, Los Angeles, CA, 1995.
4. C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence 65*, pages 179–190, 1994.
5. C. Bessiere and R. Debruyne. Theoretical analysis of singleton arc consistency. In B. Hnich, editor, *Proceedings ECAI'04 workshop on Modelling and Solving Problems with Constraints*, Valencia, Spain, 2004.
6. C. Bessière, E.C. Freuder, and J.C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.
7. C. Bessière and J.C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings IJCAI'01*, pages 309–315, Seattle, WA, 2001.

8. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proceedings CP'97*, pages 312–326, Linz, Austria, 1997.
9. R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings IJCAI'97*, pages 412–417, Nagoya, Japan, 1997.
10. J.W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, Philadelphia PA, 1995.
11. E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.
12. D. Frost, C. Bessière, R. Dechter, and J.C. Régin. Random uniform csp generators. In *http://www.ics.uci.edu/˜ frost/csp/generatotr.html*, 1996.
13. O. Lhomme. Consistency techniques for numeric csps. In *Proceedings IJCAI'93*, pages 232–238, Chambéry, France, 1993.
14. C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings IJCAI'97*, pages 366–371, Nagoya, Japan, 1997.
15. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
16. R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
17. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.
18. P. Prosser. An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.
19. P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proceedings CP'00*, pages 353–368, Singapore, 2000.