A New Algorithm for Singleton Arc Consistency

Roman Barták, Radek Erben

Charles University, Institute for Theoretical Computer Science
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
bartak@kti.mff.cuni.cz, erben@fbl.cz

Abstract

Constraint satisfaction technology emerged from AI research. Its practical success is based on integration of sophisticated search with consistency techniques reducing the search space by removing as many as possible inconsistent values from the problem. Singleton consistency is a meta-consistency technique that reinforces other pure consistency techniques by their repeated invocation and thus it achieves even better domain pruning. The paper presents a new algorithm for singleton arc consistency that improves practical time efficiency of singleton arc consistency by applying the principles behind AC-4.

Introduction

Constraint programming (CP) is a successful technology for solving combinatorial problems modelled as constraint satisfaction problems. The power behind CP is hidden in filtering variables' domains by removing inconsistent values, i.e., values that cannot be part of any solution. This technique reduces the search space and thus speeds up the solving process. Removing as many as possible inconsistencies from the constraint networks and detecting the future clashes as soon as possible are two main goals of consistency techniques. There exist many notions of consistency and many consistency algorithms were proposed to achieve these goals (Barták, 2001). Among these techniques, singleton consistency plays an exceptional role because of its meta-character (Debruyne and Bessière, 1997). Singleton consistency is not a "standalone" technique but it improves filtering power of another pure consistency technique like arc consistency or path consistency. Basically, the singleton consistency ensures that after assigning a value to the variable it is still possible to make the problem consistent in terms of the underlying consistency technique. By ensuring this basic feature for each value in the problem it is possible to remove more inconsistencies than by using the underlying consistency technique alone (Prosser, Stergiou, Walsh, 2000). Moreover, singleton consistency removes (inconsistent) values from the variables' domains so it does not change the structure of the constraint network like other stronger consistency techniques e.g. path consistency. Last but not least, provided that we have the algorithm for achieving some basic consistency level, it is easy to extend this algorithm to achieve a singleton consistency for this consistency level. On the other hand, the current singleton consistency algorithm suffers from the efficiency problems because it repeatedly invokes the underlying consistency algorithms until domain of any variable is changed. In some sense, the basic singleton consistency algorithm mimics behaviour of the old-fashioned AC-1 and PC-1 algorithms that were proposed many years ago (Mackworth, 1977). In this paper we propose to apply the improvement leading to AC-3 and AC-4 algorithms to singleton arc consistency (SAC). The basic idea of the new SAC-2 algorithm is to minimise the number of calls to the underlying AC algorithm (we use AC-4 there) by remembering the supporting values for each value. We compared the new algorithm with the original SAC-1 algorithm from (Debruyne and Bessière, 1997) on random CSPs (Gent et al. 1998). The empirical evaluation shows that the new algorithm is significantly faster in the area of phase transition where the hard problems settle.

The paper is organised as follows. First we formally introduce the notion of singleton arc consistency and we present the original SAC-1 algorithm. Then we describe the improved algorithm called SAC-2, we show soundness and completeness of this new algorithm and we describe its worst-case time and space complexity. The paper is concluded with experimental results comparing SAC-1 and SAC-2 on random constraint satisfaction problems.

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Preliminaries

A constraint satisfaction problem (CSP) P is a triple (X,D,C), where X is a finite set of variables, for each $x_i \in X$, $D_i \in D$ is a finite set of possible values for the variable x_i (the domain), and C is a finite set of constraints. In this paper we expect all the constraints to be binary, i.e., the constraint $c_{ij} \in C$ defined over the variables x_i and x_j is a subset of the Cartesian product $D_i \times D_j$. We denote by $P|D_i=\{a\}$ the CSP obtained from P by assigning a value a to the variable x_i.

Definition 1: The value a of the variable x_i is *arc consistent* if and only if for each variable x_j connected to x_i by the constraint c_{ij}, there exists a value $b \in D_j$ such that

$(a,b) \in c_{ij}$. The CSP is *arc consistent* if and only if every value of every variable is arc consistent.

The idea of singleton consistency was first proposed in (Debruyne and Bessière, 1997) and then studied in (Prosser, Stergiou, Walsh, 2000). Basically, singleton consistency extends some particular consistency level A in such a way that for any instantiation of the variable, the resulting sub-problem can be made A-consistent. In this paper we describe a new algorithm for singleton arc consistency so let us define singleton arc consistency first.

Definition 2: The value a of the variable x_i is *singleton arc consistent* if and only if the problem restricted to $x_i = a$ (i.e., $P|D_i=\{a\}$) is arc consistent. The CSP is *singleton arc consistent* if and only if every value of every variable is singleton arc consistent.

Debruyne and Bessière (1997) proposed a straightforward algorithm for achieving singleton arc consistency. This algorithm is shown in Figure 1; we call it *SAC-1*. SAC-1 first enforces arc consistency using some underlying AC algorithm and then it tests the SAC feature for each value of each variable. If the value is not singleton arc consistent, then the value is removed from the domain and arc consistency is established again. If any value is removed then the loop is repeated again and again until all the remaining values are singleton arc consistent.

```
procedure SAC_1(P);
 1 P <- AC(P);
 2 repeat
 3   change <- false;
 4   forall i ∈ X do
 5     forall a ∈ Di do
 6       if P|Di={a} leads to wipe out then
 7         Di <- Di \ {a};
 8         propagate the deletion of (i,a)
             in P to achieve AC;
 9         change <- true;
10 until change = false;
```

Figure 1. Algorithm SAC-1.

In some sense, SAC-1 behaves like AC-1 (Mackworth, 1977) – all the consistency checks are repeated for all remaining values until a fix point is reached. However, it implies that many checks may be repeated useless because they have nothing in common with the deleted value. By using this observation we propose an improvement of the SAC algorithm called *SAC-2* that performs only necessary checks after value deletion.

Algorithm SAC-2

As we mentioned above, the algorithm SAC-1 suffers from repeated checks of SAC that are not necessary. To decrease the number of checks we need to identify which values should be checked after deletion of a given value. A similar problem was resolved in the algorithm AC-3 that improves AC-1 (Mackworth, 1997). Thanks to a local character of arc consistency, AC-3 can only check the variables that are connected to the variable where some value has been deleted. AC-4 further improves this idea by remembering the direct relations between the values (Mohr, Henderson, 1986). In fact, AC-4 keeps a set of values that are supported by a given value – it is called a support set. If a value is removed from domain then only the values from the support set need to be checked for arc consistency.

Unfortunately, singleton arc consistency has a global character that complicates the definition of a support set. Recall that a value a of the variable x_i is singleton arc consistent if $P|D_i=\{a\}$ is arc consistent. If we make the problem $P|D_i=\{a\}$ arc consistent then this consistency can be broken only by removing some value from the current domain of some variable. In such a case SAC must be checked for a again. So it means that all the values in the domains of $P|D_i=\{a\}$ after making this problem arc consistent are supporters for singleton arc consistency of a. Using this principle we have proposed an algorithm SAC-2 that keeps a set of values supported by a given value. Like in AC-4, if the value is removed from domain then the values from the support set are checked for SAC.

Figure 2 formally describes the algorithm SAC-2 together with all auxiliary procedures. Note that CSP P consists of the set of variables X, the set of constraints C, and domains D_i for variables. We use the notation similar to other AC algorithms, for example (Mohr, Henderson, 1986).

```
procedure initializeAC(P, M, Counter, S^AC,
                              listAC);
 1 forall (i,j) ∈ C do
 2   forall a ∈ Di do
 3     total <- 0;
 4     forall b ∈ Dj do
 5       if (a,b) ∈ cij then
 6         total <- total + 1;
 7         Sjb^AC <- Sjb^AC ∪ {(i,a)};
 8     if total = 0 then
 9       Di <- Di \ {a};
10       M[i,a] <- 1;
11       listAC <- listAC ∪ {(i,a)};
12    else Counter[(i,j),a] <- total;
```

```
procedure pruneAC(P, M, Counter, S^AC,
                  listAC, S^SAC, listSAC);
 1 while listAC ≠ 0 do
 2   choose and delete (j,b) from listAC;
 3   forall (i,a) ∈ Sjb^AC do
 4     Counter[(i,j),a] <-
                  Counter[(i,j),a] - 1;
 5     if Counter[(i,j),a]=0
          and M[i,a]=0 then
 6       Di <- Di \ {a};
 7       M[i,a] <- 1;
 8       listAC <- listAC ∪ {(i,a)};
 9       forall (k,c) ∈ Sia^SAC do
10         listSAC <- listSAC ∪ {(k,c)};
```

```
procedure initializeSAC(P,M,Counter,S^AC,
                        listAC,S^SAC,listSAC);
 1 forall i ∈ X do
 2   forall a ∈ Di do
 3     if P|Di={a} leads to wipe out then
 4       Di <- Di \ {a};
 5       M[i,a] <- 1;
 6       pruneAC(P, M, Counter, S^AC,
                 {(i,a)}, S^SAC, listSAC);
 7       forall (k,c) ∈ Sia^SAC do
 8         listSAC <- listSAC ∪ {(k,c)};
 9     else forall (j,b) ∈ P|Di={a} do
10       Sjb^SAC <- Sjb^SAC ∪ {(i,a)};

procedure pruneSAC(P, M, Counter, S^AC,
                   listAC, S^SAC, listSAC)
1 while listSAC ≠ 0 do
2   choose and delete (i,a) from listSAC;
3   if a ∈ Di then
4     if P|Di={a} leads to wipe out then
5       Di <- Di \ {a};
6       M[i,a] <- 1;
7       pruneAC(P, M, Counter, S^AC,
                {(i,a)}, S^SAC, listSAC);
8       forall (k,c) ∈ Sia^SAC do
9         listSAC <- listSAC ∪ {(k,c)};

procedure SAC_2(P);
 1 M <- 0;
 2 Counter <- 0;
 3 S^AC <- 0;
 4 listAC <- 0;
 5 S^SAC <- 0;
 6 listSAC <- 0;
 7 initializeAC(P, M, Counter, S^AC, listAC);
 8 pruneAC(P, M, Counter, S^AC,
           listAC, S^SAC, listSAC);
 9 initializeSAC(P, M, Counter, S^AC,
           listAC, S^SAC, listSAC);
10 pruneSAC(P, M, Counter, S^AC,
           listAC, S^SAC, listSAC);
```

Figure 2. Algorithm SAC-2.

The algorithm SAC-2 is tightly integrated with AC-4 so the arc consistency procedures in SAC-2 are derived from AC-4. However, note that SAC-2 is not the same as SAC-1 where AC-4 is used for consistency tests. SAC-2 uses the standard AC-4 data structures like the support sets S^{AC}, the counters C, and the queue `listAC` of values to be checked for AC. We use an array `M` to indicate whether the value is still in the domain. There are also similar data structures for checking singleton arc consistency, namely the support sets S^{SAC} and the queue `listSAC`.

First, the algorithm SAC-2 establishes arc consistency by using the procedures *initializeAC* and *pruneAC*. Note that these procedures are almost identical to the relevant procedures in AC-4; these new procedures only modify the new data structures S^{SAC} and `listSAC` in addition to the standard "AC job". Nevertheless, during the first run of *initializeAC* and *pruneAC*, the SAC data structures remain empty. The S^{SAC} and `listSAC` are filled during *initializeSAC*; S^{SAC} captures the support sets as described earlier in this section while `listSAC` keeps the values that need to be re-checked for SAC again. Note that as soon as S^{SAC} is filled, this data structure is no more updated during the algorithm (for simplicity of implementation). The rest of the algorithm uses more or less a standard principle: the value is removed from the queue, checked for SAC and in case of failure, the value is removed from the domain, AC is established, and the supported values are added to the queue. The main difference of SAC-2 from SAC-1 is that only the relevant values are re-checked for SAC.

Naturally, the algorithms for singleton arc consistency are based on some underlying arc consistency algorithm. While SAC-1 is more or less independent of the AC algorithm, in SAC-2 we decided to integrate AC-4 as the underlying AC algorithm. AC-4 has the best worst-case time complexity but due to complex initialisation the average complexity is close to the worst complexity. To remove this difficulty, AC-6 algorithm was proposed in (Bessière, 1994). It remembers just one support for each value and when this support is lost, it looks for another one. Thus AC-6 spreads the initialisation phase over the propagation phase.

In SAC-2, the AC algorithm is called to establish arc consistency (*pruneAC*) as well as to test SAC of a given value ("*$P|D_i$={a} leads to wipe out*"). In the second case, we use a modified version of *pruneAC* applied to a copy of the problem where the domain of X_i is reduced to {*a*}. This helps us to "recover" domains and other data structures after the test. Note also that SAC data structures are not used during the test on SAC (as they are useless) while AC data structures are taken up from the main algorithm. Therefore initialisation of data structures is not repeated (they are just copied) so we believe that AC-4 pays off there. However, we did not test AC-6 yet.

Note finally that for simplicity reasons we removed the tests of domain emptiness from the description of the algorithms. If any domain is made empty then the affected procedure stops and returns a failure to the top procedure.

Theoretical Analysis

To show the correctness of the algorithm SAC-2 it is necessary to prove that every SAC inconsistent value is removed (completeness) and that no SAC consistent value is removed when SAC-2 terminates (soundness). Moreover, we also need to prove that SAC-2 terminates.

Proposition 1: The algorithm SAC-2 terminates for any constraint satisfaction problem.

Proof: The initialisation procedures consist of *for* loops only so they terminate. The pruning procedures use a *while* loop over the list of pairs (variable, value). During each cycle, one element is removed from the list. The elements are added to this list only when a value is removed from some domain. Thus, it is possible to add only a finite number of elements to the list (some elements can be added

repeatedly). Together, the pruning procedures terminates and the algorithm SAC-2 terminates too.

❑

Proposition 2: The algorithm SAC-2 does not remove any SAC consistent value from the variables' domains.

Proof: Notice that a value is removed from the domain only if the value is not arc consistent (in *pruneAC*) or the value is not singleton arc consistent (in *pruneSAC*). Thus, the algorithm SAC-2 does not remove any SAC consistent value from the variables' domains so the algorithm is sound.

❑

Proposition 3: When the algorithm SAC-2 terminates, then the domains of variables contain only singleton arc consistent values (or some domain is empty).

Proof: Every value in the domain passed the SAC test and since then the validity of the test was not violated by deleting any supporter.

❑

The worst-case time complexity of the algorithm AC-4 is $O(d^2e)$, where d is a maximum size of the domains and e is a number of constraints (Mohr and Henderson, 1986). The worst-case time complexity of the algorithm SAC-2 is thus $O(n^2d^4e)$. If SAC-1 uses AC-4 algorithm as the underlying arc consistency solver then its worst-case time complexity is $O(n^2d^4e)$ too. Nevertheless, the average complexity of SAC-2 is better because it does not need to perform so many repetitive consistency checks (see Experiments).

The space complexity of AC-4 is $O(d^2e)$, where d is a maximum size of the domains and e is a number of constraints. In addition to AC-4 data structures, the algorithm SAC-2 uses two additional data structures: `listSAC` and S^{SAC}. Thus, the space complexity of the algorithm SAC-2 is $O(n^2d^2)$ which is comparable to AC-4 for problems with a very high density of the constraints where $e = O(n^2)$.

Implementation of Data Structures

Many current research papers discuss constraint satisfaction algorithms from the theoretical point of view but without giving the implementation details. However, the practical efficiency of the algorithm is strongly influenced by the used data structures as well as by applied programming techniques. We believe that providing implementation details is at least as important as the theoretical study. In this section we show how the choice of the data structure implementing the list used in SAC-2 algorithm may influence the efficiency.

One of the main data structures in SAC-2 is the list `listSAC` keeping the values that need to be re-checked for singleton arc consistency. Visibly the ordering of values in this list may influence the number of SAC checks. On the other hand, the choice of the data structure does not influence the result of the algorithm, i.e., at the end, the same values will be removed from the domains independently of the data structure used for `listSAC`.

Let us first summarise what operations over `listSAC` are required by the SAC-2 algorithm. It must be possible to add a new value to the list. In this case the list should behave as a set, i.e., each value can be placed at most once in the list. We also need to select and delete a value from the list and to check whether the list is empty. All these operations should be performed in a constant time, if possible. The space complexity of `listSAC` should be $O(nd)$, where n is a number of variables and d is a maximum size of the domains. Note that this is the smallest possible space complexity because the `listSAC` may contain all the values for all the variables in the problem.

We have explored the standard implementations of lists using stack (LIFO) and queue (LILO) combined with a binary array modelling the set features of the list. The practical efficiency of the stack was not very good, the queue behaved well but it is hard to compare this data structure with SAC-1. Therefore we proposed a new data structure called a *cyclic list*. The cyclic list is basically a binary array indicating which elements are currently in the list. There is a pointer to this array indicating which element was explored last. The elements are explored in a lexicographic order, i.e. all values for a single variable are explored first before going to the next variable. The exploration always starts at the position of the last selected value. We call the structure a cyclic list because when the last value of the last variable is explored then the first value of the first variable is tested. Naturally, only the values marked as being in the list are checked for SAC (see Figure 3). To simplify checking emptiness of the list, there is also a counter indicating the number of elements in the list. Note that in some sense the cyclic list implements a priority queue where the elements closer (in lexicographic ordering) to the current element are preferred.

Visibly, inserting a new element to this list as well as checking emptiness requires a constant time. However, time complexity of selecting and deleting the next element from the list is $O(nd)$, where n is a number of variables and d is a maximum size of the domains. Nevertheless, this is a hypothetical complexity of the worst case which does not influence the complexity analysis of the algorithm SAC-2 in the previous section. The space complexity is $O(nd)$ as we requested.

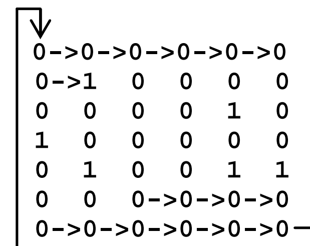

Figure 3. Values in the cyclic list are explored in a lexicographic order defined by the variables and ordering of values in the variable domain. Each row corresponds to domain of a variable.

The empirical study showed that real efficiency of the cyclic list is similar to a standard queue. However, thanks to nature of the cyclic list we can now formally prove that SAC-2 with the cyclic list does not perform more SAC tests than the SAC-1 algorithm.

Proposition 4: If the queue `listSAC` is implemented using the data structure *cyclic list* in the algorithm SAC-2, then SAC-2 does not perform more SAC tests than the SAC-1 algorithm.

Proof: During the first pass of the repeat-until loop, the SAC-1 algorithm checks all the variables' values for SAC, the same is done by SAC-2 during the procedure *initializeSAC*(). If no SAC inconsistency is detected then both algorithms terminate: SAC-1 because the flag of variable change remains false, SAC-2 because the `listSAC` remains empty. If some value is removed due to SAC inconsistency then SAC-1 repeats the tests for all the variables' values again starting from the first value of the first variable. SAC-2 explores the same set of variables' values in the procedure *pruneSAC*() but only the values that are possibly affected by the deletion are tested (these values are in the `listSAC`). Moreover, the values are explored in the same order as in SAC-1 thanks to the nature of the cyclic list. Thus during the second pass of the loop, SAC-2 will probably test a smaller number of values but definitely it does not test a larger number of values. Again, if some value is deleted then SAC-1 repeats the loop completely while SAC-2 tests only the affected values. Moreover, if the affected values are in `listSAC` before the "turnover" then these values will be tested in the same loop while SAC-1 requires one more run of the loop. Together, SAC-2 performs a smaller or equal number of loops than SAC-1 and in each loop SAC-2 tests a smaller or equal number of values. Thus the proposition holds.

❑

Experimental Results

To confirm our expectations about better practical efficiency of SAC-2 in comparison to SAC-1 we have implemented both algorithms in Java and we compared them using random constraint satisfaction problems. The experiments run on 1.7 GHz Pentium 4 and 512 MB RAM.

Random CSP

Random constraint satisfaction problems became a de facto standard for testing constraint satisfaction algorithms (Gent et al., 1998). Random CSP is a binary constraint satisfaction problem specified by four parameters:

- *n* - a number of variables,
- *d* - a size of domains,
- *density* - defines how many constraints appear in the problem,
- *tightness* - defines how many pairs of values are inconsistent in a constraint.

We have implemented a generator of Random CSP that works as follows. First, *n* variables are introduced together with the domains of size *d*. A random path of constraints between all the variables is generated to ensure that the constraint network is continuous. Then additional constraints between the random pairs of variables are added until the number of constraints is $\lfloor density*n*(n-1)/2 \rfloor$. A complete domain is defined for each constraint and $\lfloor tightness*d*d \rfloor$ random pairs of values are removed from this domain.

Three different areas can be identified in Random CSP:

- the under-constrained problems, where almost every value is singleton arc consistent so SAC is run just once for each value (small tightness),
- the over-constrained problems, where arc consistency already discovered the clashes so SAC is not run at all (large tightness),
- a phase transition area where the hard problems settle.

Comparison

We compare both the number of SAC tests performed by both algorithms and the total running time. Usually, the number of variables (*n*) and the size of domains (*d*) is fixed, while density and tightness is variable to get a set of different problems. We present the results for *n*=50 and *d*=20, the results for other pairs *n*/*d* are quite similar – we have performed the experiments for all combinations of $n \in \{10,20,30\}$ and $d \in \{10,20,30\}$. We have generated a hundred instances of each problem and for every instance both algorithms were run.

When comparing the number of SAC tests performed by SAC-1 and SAC-2 (Figure 4), we can see that in the under-constrained and over-constrained areas, the number of tests is identical for both algorithms. This is not surprising because both algorithms finish in the first stage either with a failure for the over-constrained problems or with a success for the under-constrained problems. However, in the phase transition area SAC-2 performs a visibly smaller number of tests. Actually, SAC-2 performs as 40% less SAC tests than SAC-1. The reason is that SAC-2 checks only the values that are possibly affected by some domain reduction and it may do the test earlier than SAC-1, if the affected value is already in the `listSAC`. Moreover as we showed in Proposition 4, SAC-2 never performs more tests than SAC-1, if we use the proposed data structure for `listSAC`.

When comparing the running times (Figure 5), in the area of over-constrained problems both algorithms run at the same speed simply because the SAC part is not invoked at all – the clash is already detected during establishing arc consistency. In the area of under-constrained problems, SAC-2 is slightly handicapped because of the overhead with the initialisation of the data structures. Still the running time is comparable to SAC-1. In the area of phase transition, SAC-2 again significantly outperforms SAC-1.

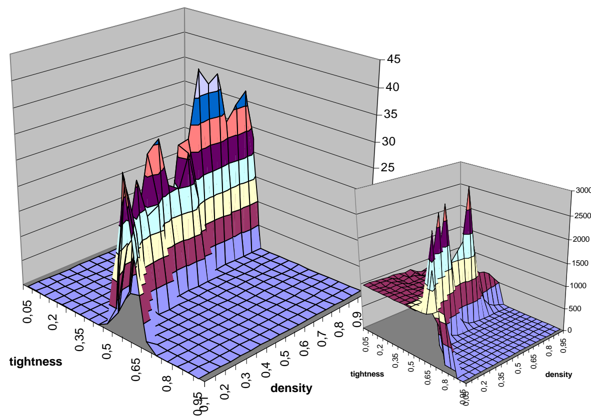

Figure 4. A comparison of the number of SAC tests performed by SAC-1 and SAC-2. The big graph (left) shows a relative comparison (100*(SAC1 - SAC2)/SAC1), the small graph (right) shows an absolute number of SAC tests performed by SAC-2.

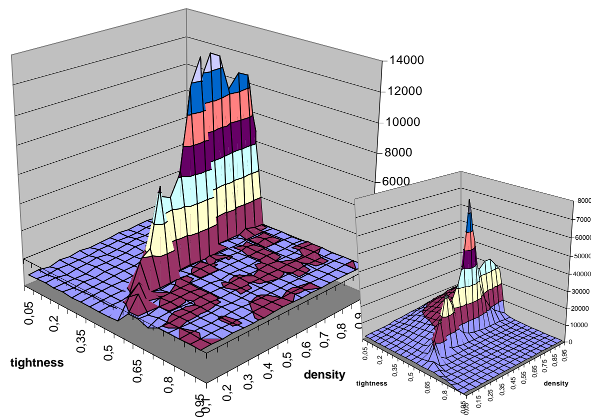

Figure 5. A comparison of running times for SAC-1 and SAC-2. The big graph (left) shows a time difference (SAC1 - SAC2) while the small graph (right) shows an absolute time of SAC2. Time is measured in milliseconds.

Conclusions

The paper presents a new algorithm for achieving singleton arc consistency called SAC-2. The algorithm is based on the ideas of AC-4 and it uses AC-4 as the underlying consistency algorithm. We formally proved that the new algorithm does not invoke the AC procedure to test the SAC condition ("*$P|D_i=\{a\}$ leads to wipe out*") more times than the original SAC-1 algorithm. In fact the empirical study shows that it invokes the AC procedure a significantly smaller number of times. Due to maintaining internal data structures, the SAC-2 algorithm has larger overhead than the SAC-1 algorithm but the empirical results show that SAC-2 still achieves better running times than SAC-1 in the area of phase transition where the hard problems settle. Still the time and space complexity of SAC-2 is not neglecting so SAC should be applied with caution. We believe that SAC can help to prune the search space before the labelling procedure starts or when variable domains are small, for example binary.

SAC-2 is useful especially when AC-4 is applied in the solving procedure as the long initialisation stage of AC-4 pays of there. Other improvements of AC-4 like AC-6 might or might not help; we have no evidence about it yet.

SAC-2 has the same time complexity as SAC-1 and this complexity is not optimal (personal communication to Christian Bessière). Nevertheless, SAC-2 can be further improved by updating the data structure S^{SAC} during the SAC tests and by using it to update the domains before doing the SAC tests. This improvement might lead to the optimal SAC algorithm but the question is whether the additional overhead pays-off.

Note finally that the ideas of SAC-2 can be extended to n-ary constraints as well as to other consistency levels like path consistency as showed in (Erben, 2002).

Acknowledgements

The research is supported by the Czech Science Foundation under the contract no. 201/01/0947. We would like to thank the reviewers for useful comments and for pointing our attention to the study of the optimal SAC complexity.

References

Barták R., 2001. Theory and Practice of Constraint Propagation. In *Proceedings of the Third Workshop on Constraint Programming in Decision and Control* (CPDC), 7-14. Silesian University.

Bessière C., 1994. Arc-consistency and arc-consistency again. *Artificial Intelligence* 65: 179-190.

Debruyne R. and Bessière C., 1997. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 412-417. Morgan Kaufmann.

Erben R., 2002. Consistency Techniques for Constraint Satisfaction, Master Thesis, Charles University in Prague.

Gent I.P., MacIntyre E., Prosser P., Smith B.M., and Walsh T., 1998. Random constraint satisfaction: Flaws and structure. Technical Report APES-08-1998, APES Research Group.

Mackworth A.K., 1977. Consistency in networks of relations. *Artificial Intelligence* 8:99-118.

Mohr R., Henderson T.C., 1986. Arc and Path Consistency Revisited. *Artificial Intelligence* 28:225-233.

Prosser P., Stergiou K., Walsh T., 2000. Singleton Consistencies. In *Principles and Practice of Constraint Programming (CP)*, 353-368. Springer Verlag.