Singleton Arc Consistency Revised

Roman Barták, Radek Erben

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

Abstract. Consistency techniques make a main flavour of most constraint satisfaction algorithms. Their goal is to remove as many as possible inconsistencies 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 and discusses some implementation details. The proposed algorithm improves practical time efficiency of singleton arc consistency by applying the principles behind AC-4.

1 Introduction

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 [1]. Among these techniques, singleton consistency plays an exceptional role because of its meta-character [3]. 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 [8]. 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 evokes 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 [6]. 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 [3] on random CSPs [5]. 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 prove soundness and completeness of this new algorithm and we give a theoretical study of time and space complexity of SAC-2. After that we give some implementation details and we prove that SAC-2 does not evoke AC filtering more times than SAC-1. The paper is concluded with experimental results comparing SAC-1 and SAC-2 on random constraints satisfaction problems.

2 Preliminaries

A constraint satisfaction problem (CSP) P is a triple (X,D,C), where X is s finite set of variables, for each $x_i \in X$, $D_i \in D$ is a set of possible values for the variable x_i (the domain), and C is a 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 [3] and then studied in [8]. 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.

In [3] Debruyne and Bessière proposed a straightforward algorithm for achieving singleton arc consistency. This algorithm is shown in Figure 1; we will call it *SAC-1*. *SAC-1* first enforces arc consistency using some underlying AC algorithm. Then it tests the SAC feature for each value of each variable and 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);
P <- AC(P);
repeat
  change <- false;
  forall i ∈ V do
    forall a ∈ $D_i$ do
      if P|$D_i$={a} leads to wipe out then
        $D_i$ <- $D_i$ \ {a};
        propagate the deletion of (i,a) in P to achieve AC;
        change <- true;
until change = false;
```

Fig. 1. Algorithm SAC-1

In some sense, the algorithm SAC-1 behaves like AC-1 and PC-1 [6] – 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 proposed an improvement of the SAC algorithm called *SAC-2* that performs only necessary checks after value deletion.

3 Algorithm SAC-2

As we mentioned in the previous section, 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* [6]. 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 [7]. 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 then only the values from the support set needs 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 the algorithm *SAC-2* that keeps a set of values supported by a given value. Like in *AC-4*, if the value is removed then the values from the support set are checked for SAC. Figure 2 formally describes the algorithm *SAC-2* together with all auxiliary procedures.

```
procedure initializeAC(P, M, Counter, $S^{AC}$, listAC);
forall (i,j) $\in$ E do
  forall a $\in$ $D_i$ do
    total <- 0;
    forall b $\in$ $D_j$ do
      if $R_{ij}$(a,b) then
        total <- total + 1;
        $S_{jb}^{AC}$ <- $S_{jb}^{AC}$ $\cup$ {(i,a)};
    if total = 0 then
      $D_i$ <- $D_i$ \ {a};
      M[i,a] <- 1;
      listAC <- listAC $\cup$ {(i,a)};
   else Counter[(i,j),a] <- total;

procedure pruneAC(P, M, Counter, $S^{AC}$, listAC, $S^{SAC}$, listSAC);
while listAC $\neq$ 0 do
  choose and delete (j,b) from listAC;
  forall (i,a) $\in$ $S_{jb}^{AC}$ do
    Counter[(i,j),a] <- Counter[(i,j),a] – 1;
    if Counter[(i,j),a] = 0 and M[i,a] = 0 then
      $D_i$ <- $D_i$ \ {a};
      M[i,a] <- 1;
      listAC <- listAC $\cup$ {(i,a)};
      forall (k,c) $\in$ $S_{ia}^{SAC}$ do
        listSAC <- listSAC $\cup$ {(k,c)};

procedure initializeSAC(P, M, Counter, $S^{AC}$, listAC, $S^{SAC}$,
listSAC);
forall i $\in$ V do
  forall a $\in$ $D_i$ do
    if P|$D_i$={a} leads to wipe out then
      $D_i$ <- $D_i$ \ {a};
      M[i,a] <- 1;
      pruneAC(M, Counter, $S^{AC}$, {(i,a)}, $S^{SAC}$, listSAC);
      forall (k,c) $\in$ $S_{ia}^{SAC}$ do
        listSAC <- listSAC $\cup$ {(k,c)};
    else forall (j,b) $\in$ P|$D_i$={a} do
      $S_{jb}^{SAC}$ <- $S_{jb}^{SAC}$ $\cup$ {(i,a)};

procedure pruneSAC(P, M, Counter, $S^{AC}$, listAC, $S^{SAC}$, listSAC)
while listSAC $\neq$ 0 do
  choose and delete (i,a) from listSAC;
  if a $\in$ $D_i$ then
    if P|$D_i$={a} leads to wipe out then
      $D_i$ <- $D_i$ \ {a};
      M[i,a] <- 1;
      pruneAC(M, Counter, $S^{AC}$, {(i,a)}, $S^{SAC}$, listSAC);
      forall (k,c) $\in$ $S_{ia}^{SAC}$ do
        listSAC <- listSAC $\cup$ {(k,c)};
```

```
procedure SAC_2(P);
M <- 0;
Counter <- 0;
$S^{AC}$ <- 0;
listAC <- 0;
$S^{SAC}$ <- 0;
listSAC <- 0;
initializeAC(P, M, Counter, $S^{AC}$, listAC);
pruneAC(P, M, Counter, $S^{AC}$, listAC, $S^{SAC}$, listSAC);
initializeSAC(P, M, Counter, $S^{AC}$, listAC, $S^{SAC}$, listSAC);
pruneSAC(P, M, Counter, $S^{AC}$, listAC, $S^{SAC}$, listSAC);
```

Fig. 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* (we discuss usage of *AC-4* later in this section). Therefore, *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 remains 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. 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. 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*"). 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 [2]. This algorithm 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. However, *AC-4* pays off when the initialisation is done just once but propagation is repeated many times. Then *AC-4* seems better than *AC-6*. Because this is the case of calling AC from the SAC framework, we decided for *AC-4*. Moreover, we use the principles of *AC-4* in *SAC-2* so a more tighten integration is desirable. Last but least let us highlight one hidden feature of the above algorithms. For simplicity reasons we removed the tests of domain emptiness from the algorithms. If any domain is made empty then the affected procedure stops and it returns a failure to the top procedure.

3.1 Soundness and completeness

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 (soundness) when *SAC-2* terminates. Moreover, we also need to prove that SAC-2 really terminates.

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

Proof: The algorithm SAC-2 consists of four components processed in a sequential order without any loops between them. Thus it is enough to show that each of these components terminates.

First, the procedure `initializeAC()` is performed (line 7). Because this procedure is defined via two nested *for* loops, it terminates trivially (it is the feature of *for* loops).

Second, the procedure `pruneAC()` is called. This is a standard AC-4 algorithm extended by filling the `listSAC`. Note that filling `listSAC` terminates because it is done via a *for* loop. Moreover, it does not influence in any way the run of the AC-4 algorithm because the AC-4 algorithm does not use the `listSAC`. In [7], the authors proved that AC-4 terminates so altogether the procedure `pruneAC()` terminates.

Third, there is the procedure `initializeSAC()`. Again, this procedure is defined via nested *for* loops. Inside these loops, `pruneAC()` is called which has been just proven to terminate. There is also one more hidden call to AC, in the test `P|`D_i`={a} leads to wipe out`. We can expect that the underlying AC tester terminates as well (in fact, we use a version `pruneAC()` without modifying `listSAC`) so altogether, the procedure `initializeSAC()` terminates.

Last but not least, there is the main SAC procedure called `pruneSAC()`. This procedure consists of a single *while* loop and we have already proved that the inner part of the loop terminates (there are only the calls to already explored AC procedures). Note that `listSAC` is finite (it contains different pairs (*variable,value*), where *value* is from a finite domain of the *variable*) and during every pass of the loop, one element from `listSAC` is removed. It is possible that some elements are added to `listSAC` by `pruneAC()` but this happens only when a value is removed from a domain of some variable. Because the number of values in the domains is finite, the `listSAC` can be extended this way only a finite number of times. Then either some domain is empty and the procedure fails, or during the next passes of the loop, the `listSAC` is being emptied till emptiness and the loop terminates.

Altogether, the algorithm *SAC-2* is finite.

❑

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

Proof: A value can be removed from the domain in two cases only. First, the value is removed by `pruneAC()` procedure because this value is not arc consistent. Visibly, if the value is not arc consistent then it is not singleton arc consistent as well. Second, the value *a* is removed by `pruneSAC()` procedure because it does not pass the test `P|D_i={a} leads to wipe out`. Note that it means that at a given time this value is not singleton arc consistent. Because the domains of the variables are not extended, the value *a* will never become singleton arc consistent later during the run of the algorithm. Thus the algorithm *SAC-2* does not remove any value, which satisfies the conditions of singleton arc consistency; it removes only such values of the variables, which do not satisfy the condition of singleton arc consistency.

❑

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: In SAC-2, every value of every variable is tested for singleton arc consistency at least ones – during the run of the procedure `initializeSAC()`. If the value *a* is not removed from the domain, then it is singleton arc consistent for the structure of domains during the test. After the successful test, the value *a* is added to the support sets S^{SAC} of all the values that remain in the domains after the test (line 10 of the procedure `initializeSAC`). Thus, SAC of the value *a* can be broken only when any of these supporting values is deleted. However, in such a case, the value *a* is added to the `listSAC` so it will be checked for SAC again (line 10 of `pruneAC`, line 10 of `initializeSAC`, and line 9 of `pruneSAC`). Thus at the end of the algorithm *SAC-2* every value in the domain of any variable satisfies the condition of singleton arc consistency because it passed the SAC test and since then the validity of the test was not violated.

❑

3.2 Time and Space Complexity

Let us now study the worst-case time complexity of the algorithm *SAC-2*. *SAC-2* uses *AC-4* algorithm for enforcing arc consistency as well as for testing SAC of a given value. 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 [7]. We can assume that the number of constraints is between $n - 1$ and $(n^2 - n)/2$, where *n* is a number of variables[1].

The *AC-4* algorithm is called at most 2*nd times during `initializeSAC()` and construction of `listSAC` and S^{SAC} data structures requires $O(n^2d^2)$ time in the worst case. Because the number of constraints is at least $n - 1$, the worst-case time complexity of the procedure `initializeSAC()` is $O(nd^3e)$.

[1] We expect that the constraint satisfaction problem cannot be decomposed into independent parts. Otherwise, the consistency algorithms can be run on these parts independently.

A value is added to the `listSAC` if and only if one of its supporters is deleted. Because the maximal number of supports per value is $O(nd)$, the total length of `listSAC` during the run of the algorithm, i.e., the number of passes of the while loop in `pruneSAC()` is $O(n^2d^2)$. During each pass, the *AC-4* algorithm is called at most two times so the worst-case time complexity of the procedure `pruneSAC()` is $O(n^2d^4e)$.

Altogether, 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. This expectation is confirmed by an experimental study presented later.

Let us now study the space complexity of the algorithm *SAC-2*. 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}. The size of the list `listSAC` is at most nd, where n is a number of variables and d is a maximum size of the domains. This is because the list can contain every value of each variable and each value can be present at most once there. Each value can be supported by all values of all the rest variables, i.e., there is at most $(n - 1)d + 1$ supporting values. The total number of values of variables is nd, therefore the overall size of the structure S^{SAC} is at most $n^2d^2 - nd^2 + nd$. Altogether, the space complexity of the algorithm *SAC-2* is thus $O(n^2d^2)$. The space complexity of *SAC-2* is thus comparable to *AC-4* for the problems with a very high density of the constraints.

4 Discussion on Implementation

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 the 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 list 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[2] 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. 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 well.

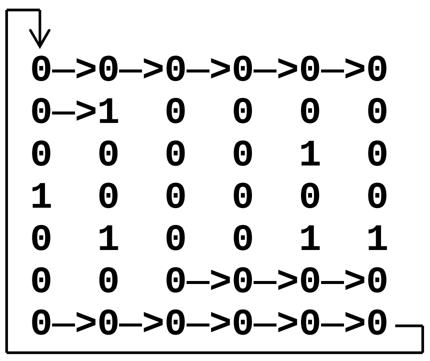


Fig. 3. Values in the cyclic list are explored in a lexicographic order defined by the variables and ordering of values in the variable domain. Each row corresponds to domain of a variable.

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 show that SAC-2 using the cyclic list performs at most the same number of SAC tests as the SAC-1 algorithm.

[2] We do not include the detail empirical comparison there due to space restrictions. The results can be found in [4].

Proposition 4: If we use a data structure *cyclic list* to implement the list `listSAC` in the algorithm *SAC-2*, then *SAC-2* performs at most the same number of SAC checks as *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 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. *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 `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 the listSAC before the "turn" 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.

❑

5 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 compared them using Random CSP.

5.1 Random CSP

Random constraint satisfaction problems became a de facto standard for testing consistency satisfaction algorithms [5]. 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.

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 $\lfloor \text{density}*\text{n}*(\text{n} - 1)/2 \rfloor$. For each constraint a complete domain is defined and from this domain random pairs are removed until the given number ⌊tightness*d*d⌋ of pairs is removed.

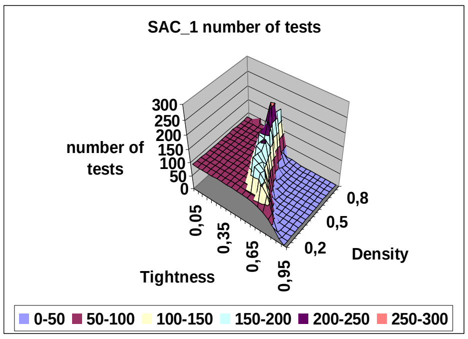
For each type of the random CSP (i.e., for each quadruple ⟨n, d, density, tightness⟩) we have generated a hundred instances of the problem and for each instance both algorithms were run. We present the tests for n=10 and d=10, for other problems the results are similar.

5.2 Comparison

In this section we present the empirical comparison of *SAC-1* and *SAC-2* algorithms. We show both the number of SAC tests performed by both algorithms and the total running time. In all the following figures we can identify three different areas:

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

Figure 4 shows the number of SAC tests performed by *SAC-1* and *SAC-2*. We can see that in the under-constrained and over-constrained areas, the number of tests is identical for both algorithms. However in the phase transition area *SAC-2* performs a visibly smaller number of tests. *SAC-2* never performs a higher number of tests than *SAC-1* as we proved in Proposition 4.



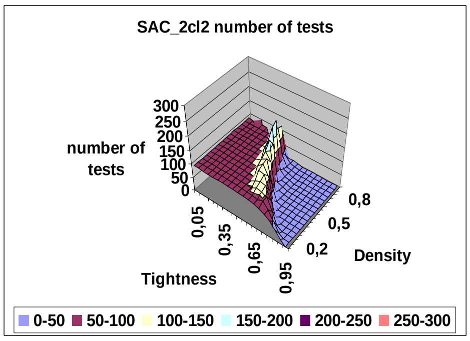


Fig. 4. Absolute comparison of the number of AC tests.

Figure 5 shows a comparison of the total running times of both algorithms (tested on Pentium 120 MHz with 48 MB RAM). In the area of over-constrained problems both algorithms run at the same speed simply because SAC test is not evoked 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 initialisation of the data structures. Still the running time is comparable to *SAC-1*. In the area of phase transition, *SAC-2* again outperforms *SAC-1*.

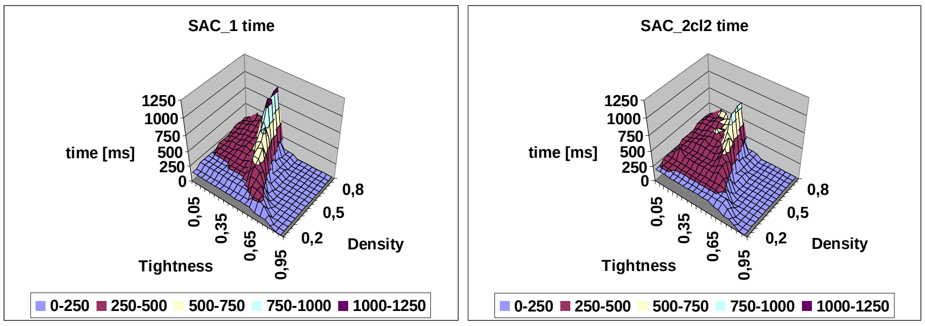


Fig. 5. Absolute comparison of the total running times.

Figure 6 shows a relative comparison of both algorithms. We can see that at the phase transition area, *SAC-2* performs as 50% less SAC tests than *SAC-1* and it is as 70% faster. However, in the under-constrained area, SAC-2 is slower than SAC-1 due to overhead with preparing the internal data structures.

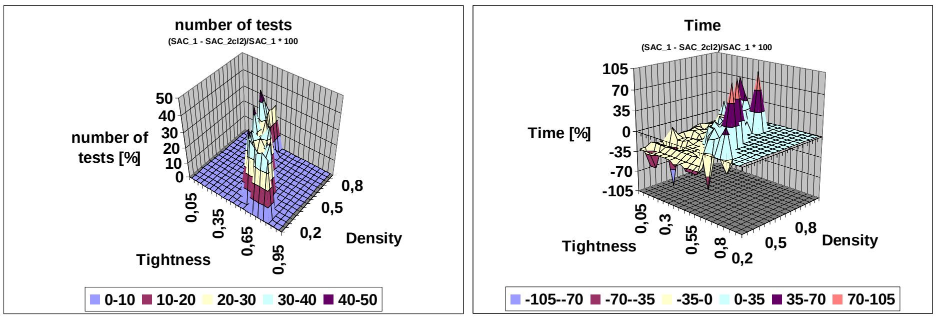


Fig. 6. Relative comparison of the algorithms (left – the number of tests, right – time)

6 Conclusions

The paper presents a new algorithm for achieving singleton arc consistency called *SAC-2*. The algorithm is based on ideas of *AC-4* and it uses *AC-4* as the underlying consistency algorithm. We showed that the new algorithm does not evoke the AC procedure more times than the original *SAC-1* algorithm. In fact the empirical study shows that it evokes 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*. Moreover the improvement of *SAC-2* is even stronger 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 it can help to prune the search space before the labelling procedure starts. *SAC-2* is useful especially when *AC-4* is applied in the solving procedure because the long initialisation stage of *AC-4* pays of there. On the other hand, the next improvements of *AC-4* like *AC-6* will not probably improve the practical time efficiency of the SAC algorithm because they have more overhead when detecting inconsistencies. Last but not least, the *SAC-2* algorithm can be naturally extended to other consistency levels. The background message of the paper is that even for global consistencies it pays off to use some supporting data structures that decrease the number of consistency checks.

Acknowledgements

The first author is supported by the Grant Agency of the Czech Republic under the contract no. 201/01/0942.

References

1 Barták R.: Theory and Practice of Constraint Propagation, In J. Figwer (editor) Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control, pages 7-14, Poland, 2001.

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

3 Debruyne R., Bessière C.: Some Practicable Filtering Techniques for the Constraint Satisfaction Problem, Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97), pages 412-417, Morgan Kaufmann, 1997.

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

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

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

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

8 Prosser P., Stergiou K., Walsh T.: Singleton Consistencies, in Proc. Principles and Practice of Constraint Programming (CP2000), pages 353-368, Springer Verlag, 2000.