Theoretical Analysis of Singleton Arc Consistency

Christian Bessiere, Romuald Debruyne

Theoretical Analysis of Singleton Arc Consistency

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 enhances the pruning capability of arc consistency by ensuring that the network can be made arc consistent after any assignment of a value to a variable. While some algorithms have been proposed in the literature to enforce SAC, a more in-depth theoretical analysis of this local consistency has never been published. In this paper, we give a lower bound to the time complexity of enforcing SAC, and we propose an algorithm that achieves this complexity, thus being optimal. We characterize some properties of SAC which are unusual for a local consistency. Based on some of these properties, we extend SAC to a more powerful local consistency that we compare to the existing local consistencies.

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 [11]. In SAT, it has been used as a way to compute more accurate heuristics for DPLL [9, 12]. Finally, in constraint satisfaction problems (CSPs), it has been proposed and studied under the name Singleton Arc Consistency (SAC) in [7]. In [15], some additional results on the pruning efficiency of SAC and an empirical study on using SAC during a preprocessing step were presented.

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 [6], 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 [14], k-consistency [10], etc. Finally, implementing it can be done simply on top of any AC algorithm.

Algorithms for SAC have been proposed in [7] and [1]. However, singleton arc consistency lacks a careful analysis of its theoretical properties.

In this paper, we study the complexity of enforcing SAC and then propose an algorithm enforcing SAC with that optimal worst-case time complexity. Afterwards, we characterize some of the basic properties that local consistencies usually satisfy, and SAC does not. We show that a slight change in the definition leads to a stronger level of consistency that we compare to existing ones.

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 Complexity of SAC

Singleton arc consistency (SAC) was presented in [7]. In the same paper, a brute force algorithm for achieving SAC was proposed. Its time complexity is $O(en^2d^4)$. But it is not optimal.

Theorem 1 *The best time complexity we can expect from an algorithm enforcing SAC is in $O(end^3)$.*

Proof. Let P be a strongly path consistent network. Verifying if it is path consistent (PC) is in $O(end^3)$. For each pair of variables (i, j), we must check whether the triangles i, k, j with $C_{kj} \in C$ are path consistent. (Checking the other triangles is useless.) Then, $e \cdot n$ triangles are checked for path consistency. Each such test is in $O(d^3)$, and is done only once since the network is already path consistent (by assumption). Thus the complexity is $O(end^3)$. Now, if we look at the PC algorithm given in [13], we see that this algorithm is nothing more than a SAC algorithm that removes a pair (a, b) from C_{ij} each time a value (j, b) is found arc inconsistent in a subnetwork $P|_{i=a}$), and propagates the consequences. At the end of this process, P is path consistent. Now, by assumption, our network is already PC. Thus, applying the regular SAC (not removing the pairs) is sufficient to detect the path consistency of P. □

4 An optimal algorithm for SAC

SAC1 [7] has no data structure storing on which values rely the SAC consistency of each value. After a value removal, SAC1 must check again the SAC consistency of all the other values.

SAC2 [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), SAC2 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 SAC1 but the data structures of SAC2 are not sufficient to reach optimality since SAC2 may waste time redoing the enforcement of AC in $P|_{i=a}$ several times from scratch.

Algorithm 1 is an algorithm that enforces SAC in $O(end^3)$, the lowest time complexity which can be expected (see Theorem 1).

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 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.

Algorithm 1 can be decomposed in several sequential steps. In the following, `propagateAC`(P, S) denotes the function that incrementally propagates the removal of the values in S 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 $P|_{i=a}$, noted P_{ia} (line 4). Each value corresponding to a failing AC test is put in the list Q for future propagation (the value is SAC inconsistent –line 6), while subproblems succesful for the AC phases are put in the *PendingList* for future update (line 5).

Once this initialization phase has finished, we copy the list Q of removed values in all the local propagation lists Q_{ia} (line 7).

The third main step concerns the AC propagation of the removal of values found SAC inconsistent (line 8). During the whole loop, a subproblem P_{ia} in *SacList* is one which currently has propagated the removal of all SAC inconsistent values (that are put in the Q_{ia} lists). One in *PendingList* has its Q_{ia} non empty (except in the first execution of the loop if all values were SAC in the initialization phase). A subproblem P_{ia} for which the AC propagation of Q_{ia} fails (line 10) is removed from the process, and (i, a) is put in all remaining Q_{jb} lists for propagation.

Algorithm 1: `The optimal SAC algorithm`

```
procedure SAC(P);
   /* init phase */;
1  P ← AC(P) ; Q ← ∅ ; SacList ← ∅ ; PendingList ← ∅;
2  foreach (X_i, a) ∈ D do
3  |  P_ia ← P /* we copy the network and its data structures nd times */;
4  |  if propagateAC(P_ia, D_i \ {a}) then
5  |  |  PendingList ← PendingList ∪ {P_ia};
6  |  else Q ← Q ∪ {(i, a)};
   /* propag phase */;
7  foreach P_ia ∈ PendingList do Q_ia ← Q;
8  while PendingList ≠ ∅ do
   |  pop P_ia from PendingList;
9  |  if propagateAC(P_ia, Q_ia) then
   |  |  SacList ← SacList ∪ {P_ia};
10 |  else
   |  |  foreach P_jb ∈ SacList ∪ PendingList do
11 |  |  |  Q_jb ← Q_jb ∪ {(i, a)};
12 |  |  |  if P_jb ∈ SacList then P_jb goes from SacList to PendingList;
13 foreach (i, a) ∈ D do  if P_ia ∉ SAClist then D_i ← D_i \ {a};
```

When *PendingList* is empty, *SacList* contains all AC subproblems. A value (i, a) is SAC iff $P_{ia} \in$ *SacList* (line 13).

Theorem 2 *Algorithm 1 is a correct SAC algorithm with optimal time complexity.*

Proof. Correctness. Suppose a value a is removed from D_i by Algorithm 1 while it should not. (i, a) is removed in line 13 only if P_{ia} is not in *SacList*. If P_{ia} was found arc inconsistent in the initialization phase (line 4), (i, a) would not be SAC. Thus, P_{ia} passed in *PendingList* at least once. If it finally does not belong to *SacList*, this means that at some point in the process, P_{ia} was arc inconsistent in line 9. Since by assumption, (i, a) should be in SAC(P), this means that at least one of the values removed from P_{ia} and that caused its arc inconsistency was SAC. By induction, there is necessarily a value (j, b) that is the first incorrectly removed value. This value cannot exist since P_{jb} has been found arc inconsistent while only SAC inconsistent values had been removed from it. Thus, Algorithm 1 is sound.

Completeness comes more directly. Thanks to the way *SacList* and *PendingList* are used in the while loop, we know that at the end of the loop, all P_{ia} are arc consistent. Thus, the values remaining after the end of Algorithm 1 are necessarily SAC.

Complexity. The complexity analysis is given for networks of binary constraints since the optimality proof was given for binary constraint networks only. But algorithm 1 works on constraints of any arity. Since any AC algorithm can

be used to implement our SAC algorithm, the space and time complexities will obviously depend on this choice. Let us first discuss the case where an optimal time algorithm such as AC-6 [4] or AC2001 [5] is used. Line 3 tells us that the space complexity will be nd times the complexity of the AC algorithm, namely $O(end^2)$. Regarding time complexity, the first loop copies the data structures and propagates arc consistency on each subproblem (line 4), two tasks which are respectively in $nd \cdot ed$ and $nd \cdot ed^2$. The `while` loop (line 8) is more tricky to evaluate. Each subproblem can in the worst case be called nd times for arc consistency, and there are nd subproblems. But it would be false to conclude on a $nd \cdot nd \cdot ed^2$ complexity since arc consistency is incremental for both AC-6 and AC2001. This means that the complexity of nd restrictions on them is still ed^2. Thus the total cost of arc consistency propagations is $nd \cdot ed^2$. We have to add the cost of updating the lists in lines 11 and 12. In the worst case, each value is removed one by one, and thus, nd values are put in nd Q lists, leading to n^2d^2 updates of *SacList* and *PendingList*. The total time complexity is thus $O(end^3)$, which is optimal. □

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, AC-7, or AC-4, the lists will be directly used like this. If it is AC-3 or AC2001, 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 AC3 is used, we decrease the space complexity to $O(n^2d^2)$, but time complexity increases to $O(end^4)$ since AC3 is not optimal.

5 Properties and weaknesses of SAC

In this section we analyse some of the properties we can expect from a local consistency, and that SAC does not possess.

The first strange behavior of SAC that we can observe is illustrated by Fig 1. On this constraint network, if we check the SAC consistency of the values in a lexicographic order, all the values preceding (l, a) are detected SAC consistent. However the deletion of the SAC inconsistent values (l, a) and (l, b) leads to the SAC inconsistency of (i, a) while none of the values in the neighbourhood of i become SAC inconsistent.

Most of the filtering algorithms are based on the notion of support. To know whether a local consistency holds for a value (i, a), a local test is performed. The aim of this test is to identify a subnetwork $Support(i, a)$ in the consistency graph that is sufficient to guarantee local consistency of (i, a). If such a subnetwork $Support(i, a)$ is found, we are sure that the local consistency holds for (i, a) as long as $Support(i, a)$ remains a subnetwork of the consistency graph. For arc consistency the subnetwork is a star composed of at least one allowed

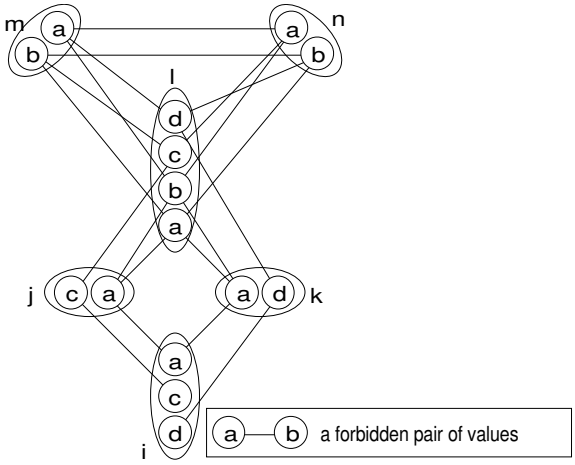

Fig. 1. Inconsistency graph of a CSP where (i, a) becomes SAC inconsistent after the deletion of (l, a) and (l, b) while the values of j and k remain SAC consistent.

arc $((i, a), (j, b))$ for each j linked to i. Let us introduce the SAC-Neighboring-Support of a value (i, a) as the subnetwork obtained by reducing $AC(P|_{i=a})$ to the neighborhood of i.

Definition 3 (SAC-N-Support) *Let (i, a) be a value of a constraint network $P = (X, D, C)$. The **SAC-N-Support** of (i, a) is the set of values in the neighborhood of i arc consistent in $P|_{i=a}$:*

$$\textit{SAC-N-Support}(i, a) = \{(j, b) \text{ s.t. } C_{ij} \in C \text{ and } (j, b) \in AC(P|_{i=a})\}.$$

Observation 1 *Let D' be the domain of $SAC(P)$ and (i, a) a value in P.*

$$\textit{SAC-N-Support}(i, a) \subseteq D' \not\Rightarrow (i, a) \in D'$$

Proof. See Fig. 1. SAC-N-Support(i, a)=$\{(j, a), (j, c), (k, a), (k, d)\}$ is included in $SAC(P)$, but once (l, a) and (l, b) are removed (because they are not SAC), (i, a) is no longer SAC. □

Observation 1 highlights that if $AC(P|_{i=a}) \neq \emptyset$ we cannot say that (i, a) will remain SAC consistent as long as all the values of SAC-N-Support(i, a) remain SAC consistent. So, the whole network $AC(P|_{i=a})$ supports (i, a), which leads to the following extended definition of SAC-support.

Definition 4 (SAC-Support) *Let (i, a) be a value of a constraint network $P = (X, D, C)$. The **SAC-Support** of (i, a) is $AC(P|_{i=a})$.*

The second strange behavior of SAC is that conversely to other local consistencies, the supports are not "bidirectional". The fact that a value (j, b) is not in the SAC-support of a value (i, a) does not imply that (i, a) is not in the SAC-support of (j, b). In Fig 2, $(i, a) \in AC(P|_{j=a})$ but $(j, a) \notin AC(P|_{i=a})$.

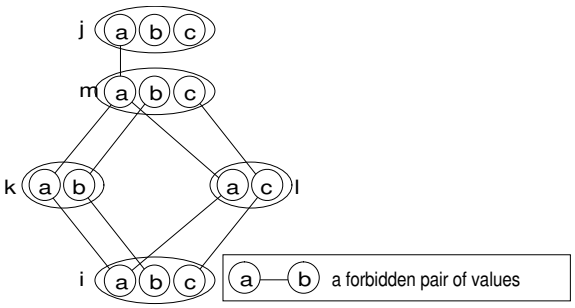

Fig. 2. Inconsistency graph of a CSP where (i, a) and (j, a) are not mutually compatible or incompatible wrt SAC.

Observation 2 *Let (i, a) and (j, b) be values in P.*

$$(j, b) \in SAC\text{-}Support(i, a) \not\Rightarrow (i, a) \in SAC\text{-}Support(j, b)$$

Proof. See Fig. 2 in which $(i, a) \in AC(P|_{j=a})$ while $(j, a) \notin AC(P|_{i=a})$. □

In fact, it is not required to enforce full arc consistency on $P|_{i=a}$ to know whether (i, a) is SAC consistent or not. Determining if AC can wipe out a domain in $P|_{i=a}$ is sufficient. So, we can use a lazy approach (like in lazy-AC7 [16]) to test the SAC consistency of the values. With a lazy approach, we obtain smaller subnetworks supporting the values. But these supports are still not bidirectional.

6 An extension of SAC

In the previous section, we gave two observations that prevent us from using classical constructions that lead to efficient local consistency algorithms. But observation 2 not only has negative consequences on the implementation of efficient SAC algorithms. It also shows a weakness in the pruning capability of SAC. If (j, b) does not belong to $AC(P|_{i=a})$, the SAC test on (j, b) could be done on a network from which (i, a) has been removed since we are guaranteed that (i, a) and (j, b) cannot belong to the same solution. Then, there would be more chances of wipe out when testing (j, b). This leads to a slightly modified definition of SAC. We obtain a stronger consistency level that we will compare to existing ones.

Definition 5 (Singleton Propagated Arc Consistency) *A constraint network $P = (X, D, C)$ is **singleton propagated arc consistant** (SPAC) iff $\forall i \in X, \forall a \in D_i$, $AC(T^{ia}) \neq \emptyset$, where $T^{ia} = (X, D^{ia}, C)$, with $D^{ia}_j = \{b \in D_j / (i, a) \in AC(P|_{j=b})\}$ (So, $D^{ia}_i = \{a\}$).*

This local consistency makes use of observation 2, which is close to what is done in 1-AC, the local consistency defined in [2].

Definition 6 (1-AC) *A constraint network $P = (X, D, C)$ is **1-AC** iff:*

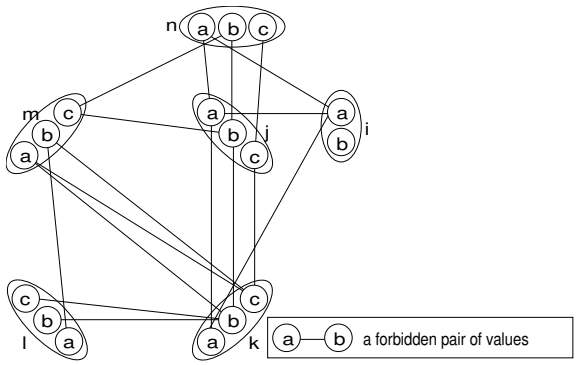

Fig. 3. Inconsistency graph of a 1-AC consistent CSP on which (i, a) is not SPAC.

– *P is SAC,*
– $\forall j \in X, \forall i \in X, \forall a \in D_i$, $\exists b \in D_j$ *such that* $a \in AC(P|_{j=b})$.

1-AC and SPAC look very similar. Nevertheless, they are different, as we will next see by comparing them wrt the relation 'stronger than' defined in [8]. Briefly, a local consistency Φ is stronger than another local consistency Φ' if any constraint network which is Φ-consistent is also Φ'-consistent. Consequently, any algorithm enforcing Φ removes at least all the values removed by an algorithm enforcing Φ'. Φ is strictly stronger than Φ' (denoted $\Phi \succ \Phi'$) if Φ is stronger than Φ' and there exists a constraint network on which Φ' holds and Φ does not.

We first give a lower bound to the level of consistency ensured by SPAC.

Property 1 *SPAC is strictly stronger than 1-AC.*

Proof. Let (i, a) removed by 1-AC and not by SPAC. (i, a) is then such that $\exists j / \forall b \in D_j, (i, a) \notin AC(P|_{j=b})$. SPAC will test $AC(T^{ia})$ with $D_j^{ia} = \emptyset$. Therefore, (i, a) is removed by SPAC.

Strictness is shown in Fig. 3. (i, a) is removed by SPAC and not by 1-AC. □

We now give an upper bound to the level of consistency ensured by SPAC. Following [8], strong path consistency, denoted AC(PC), and which consists in enforcing arc consistency after the network is made path consistent, is the tightest upper bound we can expect.

Property 2 *AC(PC) is strictly stronger than SPAC.*

Proof. Consider a SPAC inconsistent value (i, a) of a constraint network P. According to the definition of SPAC, the propagation of the deletion from $P|_{i=a}$ of all the values (j, b) such that $(i, a) \notin AC(P|_{j=b})$ leads AC to wipe out a domain D_k. In [13], McGregor enforces path consistency by removing all the pairs of values $((i, a), (j, b))$ such that (i, a) is arc inconsistent in $P|_{j=b}$. So, if we use this PC algorithm on P, all the pairs $((i, a), (j, b))$ such that $(i, a) \notin AC(P|_{j=b})$ will be removed from P when the PC algorithm will check $AC(P|_{i=a})$. Therefore,

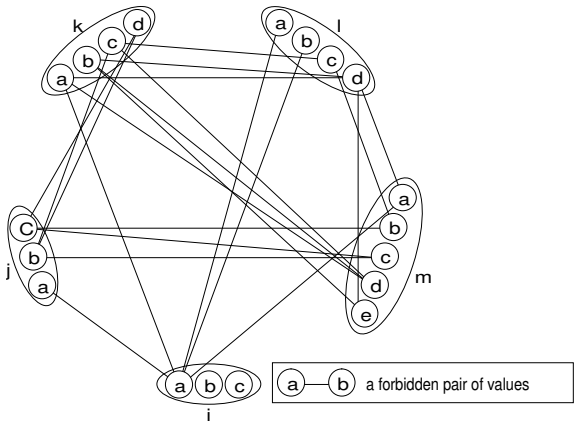

Fig. 4. Inconsistency graph of a SPAC consistent CSP on which strong path consistency removes (i, a).

AC will wipe out the domain D_k. For all $c \in D_k$, the pair of values $((i, a), (k, c))$ will be removed by the PC algorithm leading to the arc inconsistency of (i, a). So, (i, a) is also strong path inconsistent.

Strictness is shown in Fig. 4. (i, a) is removed by AC(PC) and not by SPAC. □

The proof of property 2 highlights the relations between SAC, SPAC and strong PC. We can easily obtain a SAC algorithm from a strong PC algorithm based on the algorithm of McGregor by not storing the pairs of values found path inconsistent. To test the SPAC consistency of a value (i, a) only the path inconsistency of the pairs involving (i, a) is considered. This explains that the filtering capability of SPAC is between those of SAC and strong PC.

We can summarize the relations between SPAC and the other local consistencies by:

$$AC(PC) \succ SPAC \succ \text{1-}AC \succ SAC.$$

Obviously, for any local consistency Φ we can introduce the singleton propagated Φ consistency than will enhance the filtering capability of singleton Φ consistency in a similar way to SPAC enhancing SAC.

Finally, we can point out that even if SPAC does not have the weakness of SAC on the bidirectionality of supports (Observation 2), it has other drawbacks. First, as in SAC, the support of a value (i, a) involves values in all the network, and not only in the neighborhood of i. (This can be observed on Fig. 1 since (i, a) becomes SPAC inconsistent after the deletion of (l, a) and (l, b) while the values of j and k remain SPAC consistent.) Second, we cannot use a lazy approach: AC has to be completely enforced on $P|_{i=a}$ to detect all the values SPAC inconsistent with (i, a). Third, for each value (i, a), the domain of $AC(P|_{i=a})$ has to be stored since the values (j, b) in this domain are the ones for which (i, a) can participate

to the support. This leads to an $\Omega(n^2d^2)$ space complexity and SPAC is therefore as expensive in space as strong PC.

7 Summary and Conclusion

We have made a theoretical study of singleton arc consistency. We have proposed a SAC algorithm having optimal worst case time complexity. We have observed some bad behaviors of SAC. These observations have led us to conclude that SAC is definitely not as 'local' as the other local consistencies, with some bad consequences on the way to efficiently enforce it. Another consequence is that we can enhance the pruning efficiency of SAC by changing its definition to avoid one of its observed drawbacks. The filtering capability of the new local consistency obtained is also studied.

References

1. R. Barták. A new algorithm for singleton arc consistency. In *Proceedings FLAIRS'04*, Miami Beach, Fl, 2004. AAAI Press.
2. H. Bennaceur and M.S. Affane. Partition-k-ac: an efficient filtering technique combining domain partition and arc consistency. In *Proceedings CP'01*, pages 560–564, Paphos, Cyprus, 2001. Short paper.
3. P. Berlandier. Improving domain filtering using restricted path consistency. In *Proceedings IEEE-CAIA'95*, 1995.
4. C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.
5. C. Bessière and J.C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings IJCAI'01*, pages 309–315, Seattle WA, 2001.
6. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proceedings CP'97*, pages 312–326, Linz, Austria, 1997.
7. R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings IJCAI'97*, Nagoya, Japan, 1997.
8. R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
9. J.W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, Philadelphia PA, 1995.
10. E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, Nov 1978.
11. O. Lhomme. Consistency techniques for numeric csps. In *Proceedings IJCAI'93*, pages 232–238, Chambry, France, 1993.
12. C.M. Li and Ambulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings IJCAI'97*, pages 366–371, Nagoya, Japan, 1997.
13. J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphism. *Information Science*, 19:229–250, 1979.
14. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.
15. P. Prosser, K. Stergiou, and T Walsh. Singleton consistencies. In *Proceedings CP'00*, pages 353–368, Singapore, 2000.
16. T. Schiex, J.C. Régin, G. Verfaillie, and C. Gaspin. Lazy arc-consistency. In *Proceedings AAAI'96*, pages 216–221, Portland OR, 1996.