Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence 172 (2008) 29–41
Artificial Intelligence
www.elsevier.com/locate/artint

Theoretical analysis of singleton arc consistency and its extensions ☆

Christian Bessiere [a,*], Romuald Debruyne [b]

[a] *LIRMM (University of Montpellier/CNRS), 161 rue Ada, Montpellier, France*
[b] *École des Mines de Nantes/LINA, 4 rue Alfred Kastler, Nantes, France*

Received 13 July 2006; received in revised form 29 August 2007; accepted 3 September 2007

Available online 19 September 2007

Abstract

Singleton arc consistency (SAC) is a consistency property that is simple to specify and is stronger than arc consistency. Algorithms have already been proposed to enforce SAC, but they have a high time complexity. In this paper, we give a lower bound to the worst-case time complexity of enforcing SAC on binary constraints. We discuss two interesting features of SAC. The first feature leads us to propose an algorithm for SAC that has optimal time complexity when restricted to binary constraints. The second feature leads us to extend SAC to a stronger level of local consistency that we call Bidirectional SAC (BiSAC). We also show the relationship between SAC and the propagation of disjunctive constraints.

Keywords: Constraint satisfaction problems; Local consistency; Singleton arc consistency; Disjunctive constraints; Constructive disjunction; Bidirectional singleton arc consistency

1. Introduction

Constraint propagation is an important step of reasoning when we try to solve problems with constraints. Constraint propagation usually consists in enforcing arc consistency. However, in some cases, some stronger level of consistency could pay off. Singleton arc consistency (SAC) ensures that we can enforce arc consistency without failure after any assignment of a value to a variable. It has been introduced in [12], and further studied, both theoretically and experimentally in [13,25]. This idea of looking 'one step in advance' exists under other forms. It has been applied in constraint problems with numerical domains by limiting the tentative assignments to the bounds of the domains and ensuring that bound consistency does not fail after such assignments [18]. In SAT, the technique of 'failed literals' has been used as a way to compute more accurate heuristics for DPLL [14,20]. In mixed integer programming, 'strong branching' has been used before actually branching on some variable to test which of the fractional variables gives the best progress after performing a limited number of dual simplex pivots [1].

☆ Parts of this paper have been presented in [C. Bessiere, R. Debruyne, Theoretical analysis of singleton arc consistency, in: B. Hnich (Ed.), Proceedings ECAI'04 Workshop on Modelling and Solving Problems with Constraints, Valencia, Spain, 2004, pp. 20–29] and [C. Bessiere, R. Debruyne, Optimal and suboptimal singleton arc consistency algorithms, in: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05), Edinburgh, Scotland, 2005, pp. 54–59].

* Corresponding author.
E-mail addresses: bessiere@lirmm.fr (C. Bessiere), romuald.debruyne@emn.fr (R. Debruyne).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2007.09.001

Some nice properties give to SAC a real advantage over the other local consistencies that are stronger than the ubiquitous arc consistency. The definition of SAC is much simpler than that of restricted path consistency [4], max-restricted-path consistency [11], or other exotic local consistencies. Its operational semantics can be understood by a non-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 [24], k-consistency [15], which may add constraints and thus modify the topology of the network. Finally, implementing it can be done simply on top of any arc consistency algorithm.

Three algorithms enforcing SAC have already been proposed: *SAC1* [12], *SAC2* [2] and *SAC3* [17]. But they all have a high worst-case time complexity ($O(en^2d^4)$ on binary constraints). Moreover, there does not exist any study of the complexity of singleton arc consistency that would tell us the optimal worst-case time complexity of enforcing SAC.

In this paper, we prove a lower bound on the worst-case time complexity of enforcing SAC on a network of binary constraints. We propose an algorithm whose complexity reaches this bound when restricted to binary constraints, thus being optimal for such constraints. We discuss two features inherent to singleton arc consistency. One of them, the non-locality of supports, justifies our approach for designing an optimal algorithm to enforce SAC. The other one, the non-bidirectionality of supports, leads us to the definition of a new local consistency, bidirectional singleton arc consistency (BiSAC), which is stronger than SAC. We propose a simple algorithm to enforce BiSAC. We theoretically compare the level of consistency of BiSAC with that of existing local consistencies. We also show the relationship between SAC and techniques to propagate disjunctions of constraints.

The rest of the paper is organized as follows. Section 2 contains the preliminary notions and notations necessary for reading the paper. In Section 3, we study the complexity of enforcing SAC and we show an unexpected feature of SAC: supports are not local. In Section 4 we propose *SAC-Opt*, the algorithm enforcing SAC with optimal worst-case time complexity. The relationship between SAC and propagation techniques for disjunctive constraints is presented in Section 5. In Section 6 we show a second unexpected feature of SAC (the non-bidirectionality of supports) and we present BiSAC, a new local consistency stronger than SAC that no longer has this feature; we give an algorithm that enforces BiSAC and we compare BiSAC to existing local consistencies. Section 7 concludes this work.

2. Background

A *constraint network* P consists of a finite set of n variables $X = \{X_1, \ldots, X_n\}$, a set of domains $D = \{D(X_1), \ldots, D(X_n)\}$, where the domain $D(X_i)$ is the finite set of at most d values that variable X_i can take, and a set of e constraints $C = \{c_1, \ldots, c_e\}$. Each constraint c_k is defined by the ordered set $var(c_k)$ of the variables it involves, and the set $sol(c_k)$ of combinations of values satisfying it. An assignment (or instantiation) of values to variables in a set $Y \subseteq X$ is *locally consistent* iff each variable in Y is assigned a value in its domain and all constraints c with $var(c) \subseteq Y$ are satisfied. A *solution* to a constraint network is an assignment of values to all variables in X that is locally consistent. When $var(c) = (X_i, X_j)$, we use c_{ij} to denote $sol(c)$ and we say that c is *binary*.

Given a value a in the domain of a variable X_i, denoted by (X_i, a), a *support* for the consistency of (X_i, a) on c_k is an assignment of the other variables in $var(c_k)$ to values in their domain such that together with (X_i, a) they satisfy c_k. If there is a constraint in C for which such a support does not exist, (X_i, a) is said to be *arc inconsistent*. A constraint c_k is *arc consistent* iff all the values of all the variables in $var(c_k)$ have a support on c_k. A constraint network $P = (X, D, C)$ is *arc consistent* (*AC*) iff all the constraints in C are arc consistent (or equivalently, D does not contain any arc inconsistent value). $AC(P)$ denotes the arc consistent *closure* of P, that is, the network obtained by iteratively removing arc inconsistent values from D until the remaining network is arc consistent. This process of computing $AC(P)$ is also called *enforcing* arc consistency on P. If enforcing arc consistency on P wipes out a domain (denoted by $AC(P) = \emptyset$), we say that P is *arc inconsistent*. We should bear in mind that if P is not arc consistent it is not necessarily arc inconsistent. Arc inconsistency is a stronger notion. It means that enforcing arc consistency on P is sufficient to detect that it has no solution.

A constraint network $P = (X, D, C)$ is *singleton arc consistent* (*SAC*) iff $\forall X_i \in X, \forall a \in D(X_i)$, the network $P|_{X_i=a} = (X, D|_{X_i=a}, C)$ obtained by replacing $D(X_i)$ by the singleton $\{a\}$ is not arc inconsistent. If $P|_{X_i=a}$ is arc inconsistent, we say that (X_i, a) is SAC inconsistent.

Given two variables X_i and X_j, $i \neq j$, given two values $a \in D(X_i)$ and $b \in D(X_j)$, the pair $((X_i, a), (X_j, b))$ is *path consistent* iff for any third variable $X_k, k \neq i, k \neq j$ there exists a value $v_k \in D(X_k)$ such that the binary constraints between X_i, X_j and X_k are satisfied by the assignment $((X_i, a), (X_j, b), (X_k, v_k))$. A constraint network

$P = (X, D, C)$ is *path consistent* (*PC*) iff for any pair of variables X_i, X_j $(i \neq j)$, for any pair of values $(a, b) \in D(X_i) \times D(X_j)$ not forbidden by a constraint c_{ij}, $((X_i, a), (X_j, b))$ is path consistent.

$\Phi(P)$ denotes the Φ closure for any local consistency Φ, and *enforcing* Φ denotes the process of computing $\Phi(P)$. We will use the relation 'stronger than' defined in [13] to compare local consistencies. 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.

3. Complexity analysis of SAC

In this section, we analyze singleton arc consistency from a theoretical point of view. We study its computational complexity, proving a lower bound to the worst-case time complexity of enforcing SAC. Then, we observe that the property of 'local support' usually used to build optimal algorithms for local consistencies does not hold on SAC.

3.1. Lower bound to the cost of SAC

On binary constraint networks, the three existing algorithms for enforcing singleton arc consistency, *SAC1* [12], *SAC2* [2], and *SAC3* [17], all have an $O(en^2d^4)$ time complexity. But this is not optimal. We first show a lower bound to the worst-case time complexity of enforcing SAC. We limit our analysis to binary constraint networks because the complexity of arc consistency, and that of SAC, depends on the arity of the constraints.

Theorem 1. *The worst-case time complexity of enforcing SAC on a network of binary constraints is bounded below by* $O(end^3)$*, where n is the number of variables, d is the size of the largest domain, and e is the number of constraints.*

Proof. The complexity of enforcing SAC on a constraint network of binary constraints is at least as high as the complexity of checking whether $P|_{X_i=a}$ is arc inconsistent for any value a in the domain of any variable X_i. According to [22], we know that checking whether a value (X_i, a) is such that AC does not fail in $P|_{X_i=a}$ is equivalent to verifying if in each domain $D(X_j)$ there is at least one value b locally consistent with a such that $((X_i, a), (X_j, b))$ is 'path consistent on (X_i, a)'. (A pair of values $((X_i, a), (X_j, b))$ is *path consistent on* (X_i, a) iff it is not deleted when enforcing path consistency only on pairs involving (X_i, a).) In the worst case, all the values b in $D(X_j)$ have to be considered. So, the path consistency of each pair $((X_i, a), (X_j, b))$ may need to be checked against every third variable X_k by looking for a value (X_k, v_k) compatible with both (X_i, a) and (X_j, b). However, it is useless to consider variables X_k not linked to X_j by a constraint. Indeed, if $((X_i, a), (X_j, b))$ is path inconsistent on (X_i, a) because of X_k, and X_j, X_k are not linked by a constraint, then there is no value $v_k \in D(X_k)$ locally consistent with (X_i, a). A lower bound to the worst-case time complexity of proving that path consistency on (X_i, a) does not fail is thus at least $\sum_{c_{jk} \in C} |D(X_j)| \times |D(X_k)| = ed^2$. Furthermore, checking whether $((X_i, a), (X_j, b))$ is path consistent on (X_i, a) does not help deciding whether $((X_j, b), (X_i, a))$ is path consistent on (X_j, b). Indeed, a pair $((X_i, a), (X_j, b))$ can be path consistent on (X_i, a) and the pair $((X_j, b), (X_i, a))$ can be path inconsistent on (X_j, b): Suppose $((X_i, a), (X_j, b))$ is path consistent on (X_i, a) because the pair $((X_k, v_k), (X_j, b))$ belongs to c_{kj} and $((X_j, b), (X_k, v_k))$ is path inconsistent on (X_j, b) because of some variable X_l. $((X_j, b), (X_k, v_k))$ is not removed by path consistency on (X_i, a) but it is removed by path consistency on (X_j, b). Then $((X_j, b), (X_i, a))$ is path inconsistent on (X_j, b). Thus, the worst-case time complexity of enforcing this form of path consistency of pairs on every value is at least as high as the complexity of enforcing it on each value in turn. Therefore, the worst-case time complexity of any algorithm enforcing SAC is bounded below by $O(nd \times ed^2) = O(end^3)$. □

3.2. Globality of supports for SAC

Some of the intuitive characteristics of local consistencies are no longer true on singleton arc consistency. The most noticeable one is the loss of the notion of 'local support': the removal of a value (X_j, b) anywhere in the constraint network can provoke SAC inconsistency of a value (X_i, a) even if all supports for the arc consistency of (X_i, a) remain SAC.

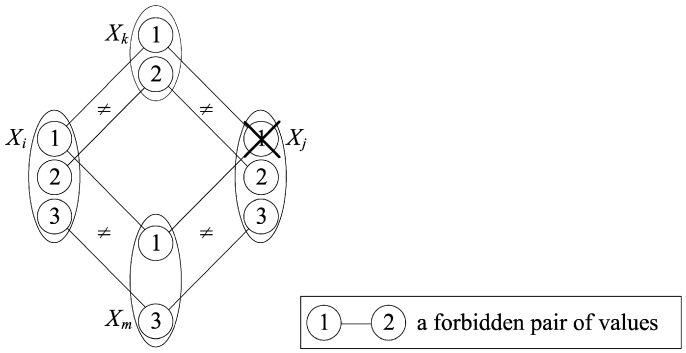

Fig. 1. Inconsistency graph of a constraint network on which $(X_i, 1)$ becomes SAC inconsistent after removing $(X_j, 1)$ whereas all values of neighboring variables remain SAC.

Local consistencies are properties usually defined on subsets of variables (or subsets of the constraints) of bounded size. Consequently, verifying that a value (or a partial instantiation in the case of k-consistencies) satisfies a local consistency property can be done by finding instantiations on the specified subsets of variables such that the specified subset of constraints is satisfied. Optimal algorithms can always be implemented on the same principle: storing 'supporting' assignments for each value (or for each partial instantiation) [10,23]. As long as its supporting assignments are consistent, the supported value verifies the given property. For arc consistency, for instance, the subnetworks to check are simply the constraints involving the given variable.

In the case of SAC, the notion of support is no longer local.

Observation 1. *Let $P = (X, D, C)$ be a SAC network, and (X_i, a) and (X_j, b) two values in P such that X_i and X_j are not involved in the same constraint. It is possible that removing (X_j, b) from P (denoted by $P \setminus \{(X_j, b)\}$) makes (X_i, a) SAC inconsistent whereas all the values of its neighbors belong to SAC$(P \setminus \{(X_j, b)\})$.*

Proof. Take the example in Fig. 1. When $(X_j, 1)$ is present, the whole network is SAC. If we remove $(X_j, 1)$, all values for X_k and X_m are still SAC. Nevertheless, $(X_i, 1)$ is not SAC because if we instantiate X_i with 1 and propagate arc consistency, values $(X_k, 1)$ and $(X_m, 1)$ are removed, which provokes the removal of $(X_j, 2)$ and $(X_j, 3)$, then wiping out X_j's domain. □

Observation 1 shows that if $AC(P|_{X_i=a}) \neq \emptyset$ we cannot guarantee that (X_i, a) will remain SAC as long as all the values in variables sharing a constraint with X_i remain SAC. The whole network $AC(P|_{X_i=a})$ supports (X_i, a). This observation will be used to build an optimal time SAC algorithm.

4. Optimal time algorithm for SAC

We propose an algorithm that enforces SAC with optimal time complexity. Such an algorithm maintains some data structures that permit to benefit from the incrementality property of optimal arc consistency algorithms.

Previous SAC algorithms are not optimal. *SAC1* and *SAC3* have no data structure storing the values that may become SAC inconsistent when a given value is removed. Hence, they must check again the SAC consistency of all remaining values. *SAC2* has data structures that use the fact that if AC does not lead to a wipe out in $P|_{X_j=b}$ then the SAC consistency of (X_j, b) holds as long as all the values in $AC(P|_{X_j=b})$ are in the domain. After the removal of a value (X_i, a), *SAC2* checks again the SAC consistency of all the values (X_j, b) such that (X_i, a) was in $AC(P|_{X_j=b})$. This leads to a better average time complexity than *SAC1* but the data structures of *SAC2* are not sufficient to reach optimality because *SAC2* may waste time re-enforcing AC in $P|_{X_j=b}$ several times from scratch.

Algorithm 1, called *SAC-Opt*, is an algorithm that enforces SAC in $O(end^3)$ if all constraints are binary. This is the lowest time complexity we can expect (see Theorem 1). The idea behind such an optimal algorithm is that we do not want to achieve (potentially nd times) arc consistency from scratch in each subproblem $P|_{X_j=b}$ each time a value (X_i, a) is found SAC inconsistent. To avoid such costly repetitions of arc consistency calls, we propose to duplicate

```
function SAC-Opt(inout P: Problem): Boolean;
     /* init phase */;
 1   if AC(P) then PendingList ← ∅ else return false;
 2   foreach (X_i, a) ∈ D do
 3     P_ia ← P|_{X_i=a}            /* copy only domains and data structures */;
 4     Q_ia ← D(X_i) \ {a};
 5     PendingList ← PendingList ∪{(X_i, a)};
     /* propag phase */;
 6   while PendingList ≠ ∅ do
 7     pop (X_i, a) from PendingList;
 8     if propagAC(P_ia, Q_ia) then Q_ia ← ∅;
 9     else
10       D ← D \ {(X_i, a)};
11       if D(X_i) = ∅ then return false;
12       foreach (X_j, b) ∈ D such that (X_i, a) ∈ D_jb do
13         D_jb(X_i) ← D_jb(X_i) \ {a};
14         Q_jb ← Q_jb ∪ {(X_i, a)};
15         PendingList ← PendingList ∪{(X_j, b)};
16   return true;
```

Algorithm 1. The optimal SAC algorithm.

nd times the domains and the data structures of the arc consistency algorithm we use (e.g., the lists S of supported values for AC-6 or the $Last$ supports for AC2001). That is, for each value (X_i, a), we store the domain of $P|_{X_i=a}$ and the associated data structures. These local domains and local data structures permit to avoid starting each new AC propagation phase from scratch. So, we can benefit from the incrementality of arc consistency on each $P|_{X_i=a}$. All generic AC algorithms are incremental, i.e., their asymptotic complexity 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. The local domains also permit to know that the values (X_j, b) that may no longer be SAC after the removal of a value (X_i, a) are those for which (X_i, a) was in the domain of $P|_{X_j=b}$.

In the following, P_{ia} denotes the subproblem in which *SAC-Opt* stores the current domain (noted D_{ia}) and the data structures corresponding to the AC enforcement in $P|_{X_i=a}$. propagAC(P, Q) denotes the function that incrementally propagates the removal of set Q of values in problem P when an initial AC call has already been executed. The initial AC call initializes the data structure required by the AC algorithm in use. It returns false iff P is arc inconsistent.

SAC-Opt is composed of two main steps. After making the 'master' problem P arc consistent (line 1), the loop in line 2 creates the subproblem P_{ia} by duplicating the domains of P together with the data structures necessary to enforce AC optimally. In this loop, we also put in Q_{ia} all values of $D(X_i)$ other than a for future propagation (line 4) and we put (X_i, a) in *PendingList*, the list of values for which some removals have not yet been propagated in their subproblem (line 5).

Once the initialization phase has finished *PendingList* contains all values (X_i, a) still in D because the removal of the values $D(X_i) \setminus \{a\}$ (contained in Q_{ia}) has not yet been propagated in P_{ia}. The loop in line 6 propagates these removals (line 8). If propagation fails, this means that (X_i, a) is SAC inconsistent. (X_i, a) is then removed from the domain D of the master problem in line 10 and each subproblem P_{jb} containing (X_i, a) is updated together with its propagation list Q_{jb} for future propagation (lines 12–15). When *PendingList* is empty, all removals have been propagated in the subproblems. So, all the values in D are SAC.

Theorem 2. *SAC-Opt is a correct SAC algorithm with* $O(enkd^{k+1})$ *time complexity and* $O(enkd^2)$ *space complexity, where* k *is the greatest arity of the constraints.*

Proof. *Soundness.* Suppose some SAC values are removed by *SAC-Opt*. Let (X_i, a) be the first SAC value removed. It is necessarily removed in line 10 because P_{ia} was found arc inconsistent in line 8. All values already removed from P_{ia} in line 13 were removed from P in line 10 and are SAC inconsistent by assumption. So, P_{ia} cannot be made arc consistent without containing some SAC inconsistent values. So, (X_i, a) is SAC inconsistent and *SAC-Opt* is sound.

Completeness. Thanks to the way *PendingList* and Q_{ia} are updated in lines 4–5 and 12–15, we know that at the end of the algorithm, all P_{ia} are arc consistent. Thus, for any value (X_i, a) remaining in P at the end of *SAC-Opt*, $P|_{X_i=a}$ is not arc inconsistent and (X_i, a) is therefore SAC.

Complexity. Because 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 *GAC-schema* [8] or *GAC2001* [9] is used. (They have an $O(ekd)$ space complexity and an $O(ekd^k)$ time complexity.) Line 3 tells us that the space complexity will be nd times the complexity of the AC algorithm, namely $O(enkd^2)$. Regarding time complexity, the AC call on the master problem (line 1) is in $O(ekd^k)$. The first loop (line 2) copies the data structures and initializes the lists Q_{ia} and *PendingList*, two tasks which are respectively in $O(nd \cdot ekd)$ and $O(nd \cdot d)$. In the `while` loop (line 6), each subproblem can be called at most nd times for arc consistency, and there are nd subproblems. Now, *GAC-schema* and *GAC2001* are both incremental, which means that the complexity of nd restrictions on the same problem is ekd^k and not $nd \cdot ekd^k$. Thus the total cost of arc consistency propagation on the nd subproblems is $nd \cdot ekd^k$. We have to add the cost of updating the lists in line 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 *PendingList*. If $n < e$, the total time complexity is $O(enkd^{k+1})$. □

Corollary 1. *On networks of binary constraints, SAC-Opt has an optimal* $O(end^3)$ *time complexity.*

We do not claim optimality of *SAC-Opt* on non-binary constraints because Theorem 1 is restricted to binary constraints. However, we strongly conjecture it is also optimal for non-binary constraints.

Remarks. We chose to present the Q_{ia} lists as lists of values to be propagated because the AC algorithm used is not specified here, and value removal is the most accurate information we can have. When using a fine-grained AC algorithm, such as *AC4* [23], *AC6* [5], or *GAC-schema*, the lists will be used as they are. If we use a coarse-grained algorithm such as *AC3* [21] or *AC2001*, only the list of variables from which the domain has changed is needed. The modification is direct. We should bear in mind that if *AC3* is used, we decrease space complexity to $O(n^2d^2)$, but time complexity increases to $O(end^4)$—on binary constraint networks—because *AC3* is not optimal. As a comparison, *SAC1* and *SAC3* have the space complexity of the AC algorithm they use (i.e., $O(e + nd)$ if *AC3* or $O(ekd)$ if *GAC2001*), and *SAC2* has a space complexity in $O(n^2d^2)$.

5. SAC and disjunctive constraints

Most constraint solvers allow to express constraints as disjunctions of other constraints. In this section, we show how singleton arc consistency can be seen as a propagation property for some special disjunctive constraints.

Constraint networks are defined as conjunctions of constraints on a set of variables. Constraint solvers usually contain libraries of 'elementary' constraints already defined together with their propagation algorithm. From a modelling point of view, it can be useful to post a disjunctive constraint, that is a constraint c which is defined as a disjunction of elementary constraints $c_1 \vee \cdots \vee c_k$ [26]. In this section, we will note $C \cup C_{disj}$ the set of constraints defining a constraint network, where C_{disj} contains those constraints explicitly defined as disjunctions of elementary constraints. In order to simplify the presentation, we will suppose that elementary constraints are all propagated via an algorithm enforcing arc consistency. Propagating a constraint $c = c_1 \vee \cdots \vee c_k$ could be done with a generic arc consistency algorithm such as *GAC-schema* [8] but it would be highly inefficient if the number of variables involved in c is large. The literature has proposed two main techniques to propagate disjunctive constraints.

5.1. Propagating disjunctive constraints

The simplest form of propagation for a disjunctive constraint $c = c_1 \vee \cdots \vee c_k$ in C_{disj} is to remove all components of the disjunction that are disentailed[1] and to move disjunctions with a single component from C_{disj} to C to be made arc consistent. We call *delayed-disjunction* consistency the level of local consistency achieved by such a technique.

[1] A constraint c_i is disentailed when enforcing arc consistency on it wipes a domain out.

Definition 1 *(Delayed-disjunction consistency).* Given a constraint network $P = (X, D, C \cup C_{disj})$ and a constraint $c \in C_{disj}$ such that $c = c_1 \vee \cdots \vee c_k$, c is *delayed-disjunction consistent* (denoted by DC) iff $k > 1$ and $\forall i \in 1..k$, c_i is not disentailed. P is DC iff all the constraints in C_{disj} are DC.

As shown in [19,26], DC is weaker than AC. Consider for instance the constraint $c = c_1 \vee c_2$ where c_1 is $Z = X$, c_2 is $Z = Y$, and the domains are $D(X) = \{1, 2\}$, $D(Y) = \{2, 3\}$, $D(Z) = \{1, 2, 3, 4\}$. Enforcing DC on c does not prune anything whereas the value $(Z, 4)$ does not have any support for AC on c.

In [26], Van Hentenryck et al. proposed constructive disjunction to overcome the weak level of consistency achieved by delayed-disjunction consistency. Given a constraint $c = c_1 \vee \cdots \vee c_k$, constructive disjunction propagates each disjunct c_i independently and stores the set S_i of values removed when propagating c_i. Then, the set of values removed by constructive disjunction on c is the intersection of all S_i.

Definition 2 *(Constructive-disjunction consistency).* Given a constraint network $P = (X, D, C \cup C_{disj})$ and a constraint $c = c_1 \vee \cdots \vee c_k$ in C_{disj}, c is *constructive-disjunction consistent* (denoted by CC) iff $\bigcap_{i \in 1..k} S_i = \emptyset$, where S_i is the set of values removed from D when enforcing AC on c_i. P is CC iff all the constraints in C_{disj} are CC.

Let us consider again the example above where $c = c_1 \vee c_2$, with c_1 is $Z = X$ and c_2 is $Z = Y$ with domains $D(X) = \{1, 2\}$, $D(Y) = \{2, 3\}$, $D(Z) = \{1, 2, 3, 4\}$. $S_1 = \{(Z, 3); (Z, 4)\}$ and $S_2 = \{(Z, 1); (Z, 4)\}$. $S_1 \cap S_2 = \{(Z, 4)\}$. Thus, $(Z, 4)$ is not CC. In [19], Lhomme shows that a disjunctive constraint is CC iff it is AC.

5.2. Extended disjunctions

When we have a disjunctive constraint $c = c_1 \vee \cdots \vee c_k$, we can use the distributivity of disjunction over conjunction to reformulate the constraint networks containing c. Given the constraint network $P = (X, D, C \cup \{c\})$, given the constraint networks $P_i = (X, D, C \cup \{c_i\})$, $i \in 1..k$, the set of solutions $sol(P)$ of P is equal to the union $\bigcup_{i \in 1..k} sol(P_i)$ of the solutions of the k networks P_i. Thus, We can extend DC and CC the following way.

Definition 3 *(Extended-delayed-disjunction consistency).* Given a constraint network $P = (X, D, C \cup C_{disj})$ and a constraint $c \in C_{disj}$ such that $c = c_1 \vee \cdots \vee c_k$, c is *extended-delayed-disjunction consistent* (denoted by EDC) iff $k > 1$ and $\forall i \in 1..k$, $(X, D, C \cup \{c_i\})$ is not arc inconsistent. P is EDC iff all the constraints in C_{disj} are EDC.

We see that the condition to be EDC is stronger than the condition to be DC because EDC does not only require a disjunct not to be disentailed, but also not to lead to a failure when propagated by AC *together with* all constraints in C.

Theorem 3. *EDC is strictly stronger than DC.*

Proof. Let us prove that EDC is stronger than DC. Suppose a constraint $c \in C_{disj}$ is not DC. If this is because c contains a single disjunct, it is then trivially not EDC. If it contains several disjuncts, suppose c_i is a disjunct that is disentailed. $(X, D, C \cup \{c_i\})$ is thus obviously arc inconsistent and c is not EDC.

To prove EDC is strictly stronger than DC, consider the network with variables X_1, X_2, X_3, domains $D(X_1) = D(X_2) = D(X_3) = \{1, 2, 3\}$ and constraints $C = \{X_1 \leqslant X_2, X_2 \leqslant X_3\}$ and $C_{disj} = \{(X_1 < X_3 \vee X_1 > X_3)\}$. None of the two disjuncts of the single disjunctive constraint are disentailed. Thus the network is DC. However, if we apply arc consistency on $C \cup \{X_1 > X_3\}$, we obtain a failure. Therefore, the network is not EDC. □

Definition 4 *(Extended-constructive-disjunction consistency).* Given a constraint network $P = (X, D, C \cup C_{disj})$ and a constraint $c = c_1 \vee \cdots \vee c_k$ in C_{disj}, c is *extended-constructive-disjunction consistent* (denoted by ECC) iff $\bigcap_{i \in 1..k} S_i = \emptyset$, where S_i is the set of values removed from D when enforcing AC on $(X, D, C \cup \{c_i\})$. P is ECC iff all the constraints in C_{disj} are ECC.

Similarly to DC and EDC, extended CC is a local consistency stronger than CC because ECC does not only require an empty set of values removed by all disjuncts but also an empty set of values removed when propagating AC on each disjunct *together with* all constraints in C.

Theorem 4. *ECC is strictly stronger than CC.*

Proof. Let us prove that ECC is stronger than CC. Suppose a constraint $c = c_1 \vee \cdots \vee c_k$ in C_{disj} is not CC. Hence, there exists a value v in some variable domain that is removed by AC on any disjunct c_i of c. Thus, v is removed by AC on $C \cup \{c_i\}$ for any disjunct c_i of c. Therefore, the constraint c is not ECC.

To prove ECC is strictly stronger than CC, consider the network with variables X_1, X_2, X_3, domains $D(X_1) = D(X_2) = D(X_3) = \{1, 2, 3\}$ and constraints $C = \{X_1 \leqslant X_2, X_2 \leqslant X_3\}$ and $C_{disj} = \{(X_1 < X_2 \vee X_2 < X_3)\}$. Arc consistency on the first disjunct of the single disjunctive constraint does not remove any value that AC on the second disjunct removes. Thus the network is CC. However, if we apply arc consistency on $C \cup \{X_1 < X_2\}$ or on $C \cup \{X_2 < X_3\}$, we remove values $(X_1, 3)$ and $(X_3, 1)$ in both cases. Therefore, the network is not ECC. □

5.3. Singleton consistencies as disjunctive consistencies

We first propose a slight modification in the definition of a constraint network that will allow us to build a comparison between disjunctive consistencies and other known local consistencies. By definition, it is implicit that every variable is assigned a value in any solution to a constraint network. This implicit constraint could be made explicit without changing the semantics of the problem. For instance, adding the constraint $(X_i = a) \vee (X_i = b) \vee (X_i = c)$ does not change anything in a network where $D(X_i) = \{a, b, c\}$. We say that a constraint network $P_E = (X, D, C \cup C_{disj})$ is the *expansion* of a constraint network $P = (X, D, C)$[2] iff $C_{disj} = \{\bigvee_{v \in D(X_i)}(X_i = v) | i \in 1..n\}$. By construction, we have $sol(P) = sol(P_E)$.

We can now show that SAC can be reformulated in terms of the propagation of disjunctive constraints by extended-delayed-disjunction consistency.

Theorem 5. *Given a network $P = (X, D, C)$, P is SAC iff its expansion P_E is EDC.*

Proof. Suppose the network $P = (X, D, C)$ is not SAC. Suppose value $v \in D(X_i)$ is the first value removed by a SAC algorithm. Hence, $AC(P|_{X_i=v})$ produces a wipe out. Thus, the network $P_E = (X, D, C \cup C_{disj})$ which is the expansion of P is not EDC because C_{disj} contains a disjunctive constraint $\bigvee_{w \in D(X_i)}(X_i = w)$ such that there exists $w \in D(X_i)$, $w = v$ and $(X, D, C \cup \{X_i = v\})$ is arc inconsistent.

Suppose the expansion P_E of P is not EDC. Hence, there exists a disjunctive constraint $X_i = v_1 \vee \cdots \vee X_i = v_k$ in C_{disj}, there exists $j \in 1..k$, such that $(X, D, C \cup \{X_i = v_j\})$ is arc inconsistent. Thus (X_i, v_j) is not SAC whereas it belongs to D because P_E contains a disjunctive constraint with $(X_i = v_j)$ as a disjunct □

We can also show that 1-AC, a local consistency stronger than SAC proposed by Affane and Bennaceur in [3], can be reformulated in terms of the propagation of disjunctive constraints by extended-constructive-disjunction consistency.

Definition 5 *(1-AC [3]).* A constraint network $P = (X, D, C)$ is 1-AC iff P is SAC and for all $X_i \in X$, for all $a \in D(X_i)$, for all $X_j \in X$, there exists $b \in D(X_j)$ such that $(X_i, a) \in AC(P|_{X_j=b})$.

Theorem 6. *Given a network $P = (X, D, C)$, P is 1-AC iff its expansion P_E is ECC.*

Proof. Suppose the network $P = (X, D, C)$ is not 1-AC. Suppose value $v \in D(X_i)$ is the first value removed by an algorithm enforcing 1-AC. Hence, $AC(P|_{X_i=v})$ produces a wipe out or there exists X_j such that $\forall w \in D(X_j)$, $(X_i, v) \notin AC(P|_{X_j=w})$. If $AC(P|_{X_i=v})$ produces a wipe out, the disjunctive constraint $\bigvee_{u \in D(X_i)}(X_i = u)$ is not ECC in P_E because (X_i, v) is removed by AC on $C \cup \{(X_i = u)\}$ for any u in $D(X_i)$. If $AC(P|_{X_i=v})$ does not lead to a wipe out, we know that there exists X_j such that $\forall w \in D(X_j)$, $(X_i, v) \notin AC(P|_{X_j=w})$. Thus, the disjunctive constraint $\bigvee_{w \in D(X_j)}(X_j = w)$ is not ECC in P_E because (X_i, v) is removed by AC on $C \cup \{(X_j = w)\}$ for any w in $D(X_j)$.

[2] Observe that we only consider constraint networks P that do not contain any disjunctive constraint.

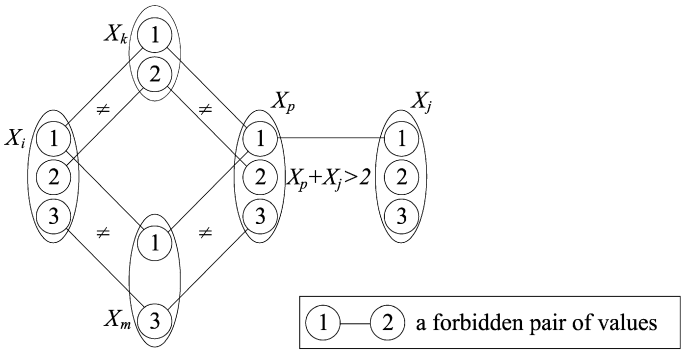

Fig. 2. Inconsistency graph of a constraint network where $(X_i, 1)$ and $(X_j, 1)$ are not mutually compatible or incompatible wrt SAC.

Suppose the expansion P_E of P is not ECC. Hence, there exists a disjunctive constraint $X_j = w_1 \vee \cdots \vee X_j = w_k$ in C_{disj}, there exists $i \in 1..n$, there exists $v \in D(X_i)$ such that v is removed from $(X, D, C \cup \{X_j = w_l\})$ by AC for all $l \in 1..k$. Thus, (X_i, v) is not 1-AC because there exists X_j such that $\forall w \in D(X_j), (X_i, v) \notin AC(P|_{X_j=w})$. □

6. Bidirectional SAC

In addition to the non-locality of supports observed in Section 3.2, SAC has a second irregular behavior. We characterize it in this section, and we propose a new local consistency, bidirectional SAC (BiSAC), which does not have this strange behavior. Like SAC, this local consistency can be applied indifferently to binary or non-binary constraints. We show that the new local consistency is stronger than SAC.

In local consistencies, usually, if a value (X_j, b) belongs to a set of values supporting a value (X_i, a), this set of values together with (X_i, a) also supports (X_j, b). This property was named *bidirectionality* of support in [7]. SAC does not have this property.

Observation 2. *Let* (X_i, a) *and* (X_j, b) *be values in* P. $(X_j, b) \in AC(P|_{X_i=a}) \not\Rightarrow (X_i, a) \in AC(P|_{X_j=b})$.

Proof. See Fig. 2 in which $(X_i, 1) \in AC(P|_{X_j=1})$ whereas $(X_j, 1) \notin AC(P|_{X_i=1})$. □

Observation 2 shows a weakness in the pruning capability of SAC. If (X_j, b) does not belong to $AC(P|_{X_i=a})$, we are guaranteed that (X_i, a) and (X_j, b) cannot belong to the same solution. So, the arc consistency test on $P|_{X_j=b}$ could safely be done on a network from which (X_i, a) has been removed. Then, there would be more chances of wipe out when testing (X_j, b). This leads to a slight modification of the definition of SAC. We obtain a stronger consistency level that we will compare to existing ones.

Definition 6 *(Bidirectional singleton arc consistency).* A constraint network $P = (X, D, C)$ is *bidirectional singleton arc consistent* (BiSAC) iff $\forall X_i \in X, \forall a \in D_i$, $AC(T_{ia}) \neq \emptyset$, where $T_{ia} = (X, D^T_{ia}, C)$, with $D^T_{ia}(X_j) = \{b \in D(X_j) \mid (X_i, a) \in AC(P|_{X_j=b})\}$. (So, $D^T_{ia}(X_i) = \{a\}$.)

Remark that D^T_{ia} is a kind of dual of D_{ia} of Section 4: $D_{ia}(X_j)$ contains all values for X_j arc consistent in $P|_{X_i=a}$ whereas $D^T_{ia}(X_j)$ contains all values b such that (X_i, a) is arc consistent in $P|_{X_j=b}$.

6.1. Algorithm for BiSAC

Algorithm 2 is a brute-force algorithm for enforcing BiSAC. The loop in lines 3–10 checks BiSAC for each value in turn. To know whether a value (X_i, a) is BiSAC, we need to build the domain D^T of the values (X_j, b) such that (X_i, a) belongs to $AC(P|_{X_j=b})$ (lines 5–6). Then, the test for BiSAC on (X_i, a) is performed in line 7. In case of failure, (X_i, a) is removed from the domain (line 8) and the Boolean is updated (line 10). Because there are no data structures to store already computed information, the process is embedded in a main loop (line 1) that is executed until no changes occur.

```
function BiSAC1 (inout P: Problem): Boolean;
1   repeat
2     CHANGE ← false;
3     foreach (X_i, a) ∈ D do
4       D^T ← ∅;
5       foreach (X_j, b) ∈ D do
6         if (X_i, a) ∈ AC(P|_{X_j=b}) then D^T ← D^T ∪ {(X_j, b)};
7       if AC(X, D^T, C) = ∅
8         D(X_i) ← D(X_i) \ {a};
9         if D(X_i) = ∅ then return false;
10        CHANGE ← true;
    until not CHANGE;
    return true;
```

Algorithm 2. A brute-force algorithm for BiSAC.

Theorem 7. *BiSAC1 is a correct algorithm for enforcing BiSAC.*

Proof. *Soundness.* Suppose some BiSAC values are removed by BiSAC1. Let (X_i, a) be the first BiSAC value removed. It is necessarily removed in line 8. This means that (X, D^T, C) was found arc inconsistent in line 7. Now, by construction (lines 5–6), $(X, D^T, C) = T_{ia} \cap D$, where T_{ia} is as defined in Definition 6. Hence, if (X_i, a) was BiSAC, there are some values that are BiSAC but which were no longer in D when (X_i, a) has been unsuccessfully checked for BiSAC. This contradicts the assumption that (X_i, a) was the first BiSAC value removed. Therefore, BiSAC1 is sound.

Completeness. The main loop finishes when the loop in line 3 has been executed on all values remaining in D without any deletion. Thus, for every value (X_i, a) in D, the problem T_{ia} as defined in Definition 6 has been tested not to be arc inconsistent (line 7). Therefore, D is BiSAC. □

The complexity of BiSAC1 depends on the complexity of the AC algorithm used. Calls to arc consistency are performed in lines 6 and 7. The AC call in line 6 is executed at most nd times for each call to the loop of line 5. That loop of line 5 and the AC call of line 7 are executed at most nd times for each call to the loop of line 3. This last loop is executed at most for each domain change, that is, nd times. Thus, the total time complexity of BiSAC1 is $nd \cdot nd \cdot (nd + 1)$ times the complexity of the arc consistency algorithm used. For binary networks, this gives a complexity in $O(en^3d^5)$ if an optimal AC algorithm is used. As for the space complexity, there is no data structures other than those of the arc consistency algorithm used.

BiSAC1 is of course non-optimal. There are many ways to improve it. For instance, if we accept to increase the space complexity to $O(n^2d^2)$, we can store and maintain the domain D^T for each value in D, and avoid the arc consistency test in line 6. Therefore, time complexity decreases by a factor nd, becoming $O(en^2d^4)$ for networks of binary constraints. We could also use a technique close to that in *SAC-Opt* to enforce BiSAC at a lower cost.

6.2. Comparing BiSAC to other local consistencies

BiSAC is a direct application of Observation 2. It can seem similar to 1-AC. Nevertheless, they are different, as shown in the following theorem.

Theorem 8. *BiSAC is strictly stronger than 1-AC.*

Proof. Let (X_i, a) removed by 1-AC. (X_i, a) is then such that $\exists X_j \mid \forall b \in D(X_j), (X_i, a) \notin AC(P|_{X_j=b})$. BiSAC will test $AC(T_{ia})$ with $D^T_{ia}(X_j) = \emptyset$. Therefore, (X_i, a) is removed by BiSAC and BiSAC is stronger than 1-AC. Strictness is shown in Fig. 3. $(X_i, 1)$ is removed by BiSAC and not by 1-AC. □

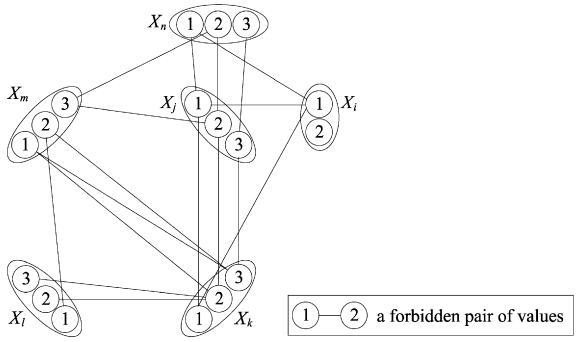

Fig. 3. Inconsistency graph of a constraint network which is 1-AC but on which $(X_i, 1)$ is not BiSAC.

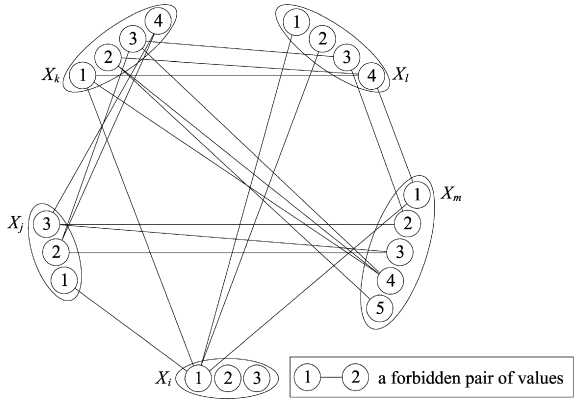

Fig. 4. Inconsistency graph of a constraint network which is BiSAC but on which strong path consistency removes $(X_i, 1)$.

Theorem 8 gives a lower bound to the level of consistency ensured by BiSAC. The following theorem shows that strong path consistency (denoted by StrongPC) is an upper bound. Strong path consistency is the property of having both arc and path consistency [16].

Theorem 9. *StrongPC is strictly stronger than BiSAC.*

Proof. Let us prove that StrongPC is stronger than BiSAC. Consider a BiSAC inconsistent value (X_i, a) in a constraint network P. According to the definition of BiSAC, the propagation of the deletion from $P|_{X_i=a}$ of the set S of all the values (X_j, b) such that $(X_i, a) \notin AC(P|_{X_j=b})$ leads AC to wipe out a domain $D(X_k)$. In [22], McGregor shows that a network P is path consistent iff there does not exist any pair of values $((X_i, a), (X_j, b))$, allowed by P, such that (X_i, a) is arc inconsistent in $P|_{X_j=b}$. So, all the pairs $((X_i, a), (X_j, b))$ such that $(X_j, b) \in S$ are path inconsistent. Therefore, once these pairs have been removed from P, AC on $P|_{X_i=a}$ will remove all those values (X_j, b) such that $(X_i, a) \notin AC(P|_{X_j=b})$ and will wipe out the same domain $D(X_k)$ as BiSAC. Thus, for all $c \in D(X_k)$, the pair of values $((X_i, a), (X_k, c))$ is not path consistent. As a result, applying AC on the path consistent closure of P will remove (X_i, a), which has no support on X_k. So, (X_i, a) is not StrongPC. Strictness is shown in Fig. 4. $(X_i, 1)$ is removed by StrongPC and not by BiSAC. □

In [13] we gave a complete picture of the relations between existing local consistencies, from AC to StrongPC. This picture can now be completed by our new results. Here is the summary of the relations between BiSAC and the other local consistencies:

$$\mathit{StrongPC} \succ \mathit{BiSAC} \succ \text{1-}\mathit{AC} \succ \mathit{SAC}.$$

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

7. Conclusion

We have proved what is the optimal worst-case time complexity of enforcing SAC on a network of binary constraints. We have presented *SAC-Opt*, the first SAC algorithm that has an optimal time complexity when enforced on binary constraints. (Optimality is conjectured for non-binary constraints.) We have shown how SAC (and its extension to 1-AC) is related to techniques for propagating disjunctive constraints. Furthermore, some observations we made on the characteristics of SAC led us to propose BiSAC, a new local consistency stronger than SAC. We proposed a simple algorithm enforcing BiSAC and we theoretically compared BiSAC to existing local consistencies.

This comprehensive theoretical analysis of SAC opens new directions of research. On the one hand, *SAC-Opt* is a new type of algorithm for SAC. We could look for other algorithms based on this scheme, such as *SAC-SDS* already proposed in [6]. On the other hand, BiSAC is a new local consistency that remains to be fully studied, by proposing better algorithms to enforce it, and by analyzing its effectiveness to prune values.

Acknowledgements

We would like to thank Thierry Petit and Bruno Zanuttini for helpful comments on this work.

References

[1] D. Applegate, R. Bixby, V. Chvátal, W. Cook, On the solution of traveling salesman problems, in: International Congress of Mathematicians, Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung (1998) 645–656.
[2] R. Barták, R. Erben, A new algorithm for singleton arc consistency, in: Proceedings FLAIRS'04, Miami Beach, FL, AAAI Press, 2004.
[3] H. Bennaceur, M.S. Affane, Partition-k-AC: an efficient filtering technique combining domain partition and arc consistency, in: Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP'01), Paphos, Cyprus, in: LNCS, vol. 2239, Springer-Verlag, 2001, pp. 560–564, Short paper.
[4] P. Berlandier, Improving domain filtering using restricted path consistency, in: Proceedings IEEE Conference on Artificial Intelligence and Applications (CAIA'95), 1995.
[5] C. Bessiere, Arc-consistency and arc-consistency again, Artificial Intelligence 65 (1994) 179–190.
[6] C. Bessiere, R. Debruyne, Optimal and suboptimal singleton arc consistency algorithms, in: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05), Edinburgh, Scotland, 2005, pp. 54–59.
[7] C. Bessiere, E.C. Freuder, J.C. Régin, Using constraint metaknowledge to reduce arc consistency computation, Artificial Intelligence 107 (1999) 125–148.
[8] C. Bessiere, J.C. Régin, Arc consistency for general constraint networks: preliminary results, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), Nagoya, Japan, 1997, pp. 398–404.
[9] C. Bessiere, J.C. Régin, R.H.C. Yap, Y. Zhang, An optimal coarse-grained arc consistency algorithm, Artificial Intelligence 165 (2005) 165–185.
[10] M.C. Cooper, An optimal k-consistency algorithm, Artificial Intelligence 41 (1989/90) 89–95.
[11] R. Debruyne, C. Bessiere, From restricted path consistency to max-restricted path consistency, in: Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97), Linz, Austria, in: LNCS, vol. 1330, Springer-Verlag, 1997, pp. 312–326.
[12] R. Debruyne, C. Bessiere, Some practicable filtering techniques for the constraint satisfaction problem, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), Nagoya, Japan, 1997, pp. 412–417.
[13] R. Debruyne, C. Bessiere, Domain filtering consistencies, Journal of Artificial Intelligence Research 14 (2001) 205–230.
[14] J.W. Freeman, Improvements to propositional satisfiability search algorithms, PhD thesis, University of Pennsylvania, Philadelphia PA, 1995.
[15] E.C. Freuder, Synthesizing constraint expressions, Communications of the ACM 21 (11) (November 1978) 958–966.
[16] E.C. Freuder, A sufficient condition for backtrack-free search, Journal of the ACM 29 (1) (January 1982) 24–32.
[17] C. Lecoutre, S. Cardon, A greedy approach to establish singleton rc consistency, in: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05), Edinburgh, Scotland, 2005, pp. 199–204.
[18] O. Lhomme, Consistency techniques for numeric csps, in: Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93), Chambéry, France, 1993, pp. 232–238.
[19] O. Lhomme, Efficient filtering algorithm for disjunction of constraints, in: Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03), Kinsale, Ireland, in: LNCS, vol. 2833, Springer-Verlag, 2003, pp. 904–908.
[20] C.M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), Nagoya, Japan, 1997, pp. 366–371.
[21] A.K. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1977) 99–118.
[22] J.J. McGregor, Relational consistency algorithms and their application in finding subgraph and graph isomorphism, Information Science 19 (1979) 229–250.
[23] R. Mohr, T.C. Henderson, Arc and path consistency revisited, Artificial Intelligence 28 (1986) 225–233.
[24] U. Montanari, Networks of constraints: Fundamental properties and applications to picture processing, Information Science 7 (1974) 95–132.

[25] P. Prosser, K. Stergiou, T. Walsh, Singleton consistencies, in: Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP'00), Singapore, in: LNCS, vol. 1894, Springer-Verlag, 2000, pp. 353–368.
[26] P. Van Hentenryck, V.A. Saraswat, Y. Deville, Design, implementation, and evaluation of the constraint language cc(FD), Journal of Logic Programming 37 (1–3) (1998) 139–164.