Beyond Singleton Arc Consistency

M.R.C. van Dongen 1

Abstract. Shaving algorithms, like singleton arc consistency (SAC), are currently receiving much interest. They remove values which are not part of any solution. This paper proposes an efficient shaving algorithm for enforcing stronger forms of consistency than SAC. The algorithm is based on the notion of weak k-singleton arc consistency, which is equal to SAC if \( k = 1 \) but stronger if \( k > 1 \). This paper defines the notion, explains why it is useful, and presents an algorithm for enforcing it. The algorithm generalises Lecoutre and Cardon’s algorithm for establishing SAC. Used as pre-processor for MAC it improves the solution time for structured problems. When run standalone for \( k > 1 \), it frequently removes more values than SAC at a reasonable time. Our experimental results indicate that at the SAC phase transition, it removes many more values than SAC-1 for \( k = 16 \) in less time. For many problems from the literature the algorithm discovers lucky solutions. Frequently, it returns satisfiable CSPs which it proves inverse consistent if all values participate in a lucky solution.

1 Introduction

The notion of local consistency plays an important role in constraint satisfaction, and many such notions have been proposed so far. For the purpose of this paper we restrict our attention to binary Constraint Satisfaction Problems (CSPs).

A CSP having variables \( X \) is \((s, t)\)-consistent [6] if any consistent assignment to the \( s \) variables in \( S \) can be extended to some consistent assignment to the \( s + t \) variables in \( S \cup T \), for any sets \( S \subseteq X \) and \( T \subseteq X \backslash S \) such that \( |S| = s \) and \( |T| = t \). Local consistency notions are usually enforced by removing tuples and/or recording nogoods. As \( s \) increases enforcing \((s, t)\)-consistency becomes difficult because it requires identifying and recording \( O((S \cup T)^d_t) = O(n^t d^t) \) nogoods, where \( n \) is the number of variables in the CSP and \( d \) is the maximum domain size. For example, \((2, 1)\)-consistency, also known as path consistency [9], is in the most benign class beyond \( s = 1 \) but it is considered prohibitive in terms of space requirements. The space complexity issues arise with increasing \( s \) are the reason why practical local consistency algorithms keep \( s \) low. Usually, this means setting \( s \) to 1, which means enforcing consistency by removing values from the domains.

Shaving algorithms also enforce consistency by removing values from the domains. They fix variable-value assignments and remove values that inference deems inconsistent. They terminate if no values can be removed. A shaving algorithm, which is currently receiving much attention, is singleton arc consistency (SAC) [4; 5; 11; 1; 8].

Another computational complexity source of \((s, t)\)-consistency is the requirement that each consistent assignment to the \( s \) variables in \( S \) be extendable to a consistent assignment to the \( s + t \) variables in \( S \cup T \), for each set \( S \subseteq X \) having Cardinality \( s \) and any set \( T \subseteq X \backslash S \) having Cardinality \( t \). Relaxing this requirement by substituting some for any would make it considerably more easy to enforce the resulting consistency notion, which we call weak \((s, t)\)-consistency.

At first glance weak \((s, t)\)-consistency may seem too weak to do useful propagation. However, it has strong advantages:

1. We don’t have to find a consistent extending assignment to the variables of all sets \( T \) of size \( t \). This is especially useful if the problem is already \((s, t)\)-consistent or if \( s \) is small.
2. By cleverly selecting \( T \) we may still find inconsistencies. For example, if there is no consistent assignment to the variables in \( S \cup T \) for the first set \( T \) then the current assignment to the variables in \( S \) cannot be extended.
3. By increasing \( t \) we can enforce levels of consistency which, in a certain sense, are stronger and stronger.

This paper proposes to exploit these strengths, proposing an algorithm which switches between enforcing weak and full consistency, taking the best from both worlds. Given a consistent assignment to the variables in \( S \), the algorithms only seek a consistent assignment to \( S \cup T \) for a, cleverly chosen, first set \( T \), for which such consistent assignment is unlikely to exist. Should there be a consistent assignment then the current assignment to \( S \) is weakly consistent and otherwise it is inconsistent. If \( s = 1 \) then this allows us to prune the value that is currently assigned to the variable in \( S \).

From now on let \( s = 1 \). We apply the idea of switching between weak and full consistency to \( k\)-singleton arc consistency, which generalises SAC [4; 5; 11; 1; 8]. Here, a CSP, \( P \), is \( k\)-singleton arc consistent (weakly \( k\)-singleton arc consistent) if for each variable \( x \) and each value \( v \) in its domain, the assignment \( x := v \) can be extended by assigning values to each (some) selection of \( k - 1 \) other variables such that \( P \) can be made arc consistent. SAC is equivalent to \( k\)-singleton arc and weak \( k\)-singleton arc consistency if \( k = 1 \) but weaker if \( k > 1 \). Switching between weak and full \( k\)-singleton arc consistency allows us, in a reasonable time, to enforce stronger levels of consistency, which go beyond SAC. Using an algorithm which enforces weak \( k\)-singleton arc consistency as a pre-processor for MAC [12] allows the solution of CSPS which cannot be solved by other known algorithms in a reasonable amount of time. Our algorithm is inspired by Lecoutre and Cardon’s algorithm for enforcing SAC [8]. Like theirs it frequently detects lucky solutions [8] (solutions discovered while enforcing consistency), making search unnecessary for certain

---

1 Cork Constraint Computation Centre (4C). 4C is supported by Science Foundation Ireland under Grant 00/PI.1/C075.
satisfiable problems. Going beyond SAC, our algorithm detects certain unsatisfiable problems without search, including problems that can be made SAC. A problem is inverse consistent if all values participate in some solution. Our algorithms make and prove many problems inverse consistent, including (un)modified radio link frequency assignment problems, and forced random binary problems. Sometimes this is done more quickly than it takes SAC-1 to make these problems SAC.

We start by recalling definitions of constraint satisfaction, by recalling existing notions of consistency, and by introducing new consistency notions. This includes weak k-singleton arc consistency. Next we describe an algorithm for enforcing it. Finally, we present experimental results and conclusions.

2 Constraint Satisfaction

A binary constraint satisfaction problem (CSP), \( \mathcal{P} \), comprises a set of \( n \) variables \( X \), a finite domain \( D(x) \) for each \( x \in X \), and a set of \( e \) binary constraints. The maximum domain size is \( d \). Each constraint is a pair \( (\sigma, \rho) \), where \( \sigma = (x, y) \in X^2 \) is the scope, and \( \rho \subseteq D(x) \times D(y) \) is the relation of the constraint. Without loss of generality we assume that \( x \neq y \) for any scope \( \langle x, y \rangle \). We write \( \mathcal{P} \models \varphi \) if \( \mathcal{P} \) has no empty domains.

Let \( \langle \langle x, y \rangle, \rho \rangle \) be a constraint. Then \( v \in D(x) \) and \( w \in D(y) \) are arc consistent if \( \{ v \} \times D(y) \cap \rho \neq \emptyset \) and \( D(x) \times \{ w \} \cap \rho = \emptyset \). The arc consistent equivalent of \( \mathcal{P} \), denoted \( ac(\mathcal{P}) \), is obtained from \( \mathcal{P} \) by repeatedly removing all arc inconsistent values (and, if needed, adjusting constraint relations).

The density of \( \mathcal{P} \) is defined \( 2c/(n^2 - n) \). The tightness of constraint \( \langle \langle x, y \rangle, \rho \rangle \) is defined \( |\rho|/|D(x) \times D(y)| \).

An assignment is a partial function with domain \( X \). A k-assignment assigns values to \( k \) variables (only). By abuse of notation we write \( \{ x_i = f(x_i), \ldots, x_k = f(x_k) \} \) for k-assignment \( f \). Let \( f = \{ x_1 = v_1, \ldots, x_k = v_k \} \) be a k-assignment. We call \( f \) consistent if \( k = 1 \) and \( v_i \in D(x_i) \) for any \( i \in \{ 1, \ldots, k \} \) or \( k > 1 \) and \( \{ v_i, v_j \} \in \rho \) for each constraint \( \langle \langle x_i, x_j \rangle, \rho \rangle \) such that \( 1 \leq i, j \leq k \). A consistent n-assignment is a solution. \( \mathcal{P} \models \varphi \) is obtained from \( \mathcal{P} \) by substituting \( D(x_i) \cap \{ v_i \} \) for \( D(x_i) \) for all \( i \) such that \( 1 \leq i \leq k \).

3 Consistency

Definition 1 ((s,t)-consistency). A CSP with variables \( X \) is \((s,t)\)-consistent if for any variables \( S = \{ x_1, \ldots, x_s \} \subseteq X \) and any variables \( \{ x_{s+1}, \ldots, x_{s+t} \} \subseteq X \setminus S \), any consistent \( s\)-assignment to \( x_1, \ldots, x_s \) is extendable to some consistent \((s+t)\)-assignment to \( x_1, \ldots, x_{s+t} \).

Enforcing \((s,t)\)-consistency may require processing (almost) all assignments to all combinations of \( s \) variables and all assignments to all combinations of \( t \) additional variables. As \( s \) and \( t \) become large, enforcing \((s,t)\)-consistency usually results in a large average running time and (generally) requires \( \mathcal{O}(n^d) \) space for recording nogoods. The following relaxes \((s,t)\)-consistency by substituting some for the second occurrence of any in the definition of \((s,t)\)-consistency.

Definition 2 (Weak \((s,t)\)-Consistency). A CSP with variables \( X \) is weakly \((s,t)\)-consistent if for any variables \( S = \{ x_1, \ldots, x_s \} \subseteq X \) and some variables \( \{ x_{s+1}, \ldots, x_{s+t} \} \subseteq X \setminus S \), any consistent \( s\)-assignment to \( x_1, \ldots, x_s \) is extendable to some consistent \((s+t)\)-assignment to \( x_1, \ldots, x_{s+t} \).

A CSP is inverse \( k\)-consistent if it is \((1, k - 1)\)-consistent. Inverse consistency does not require additional constraints and can be enforced by shaving. A CSP is inverse consistent if it is inverse \( n\)-consistent. It is weakly inverse \( k\)-consistent if it is weakly \((1, k - 1)\)-consistent. Then (weak) inverse \( k\)-consistency implies (weak) inverse \( k\)-consistency if \( K \geq k \).

A CSP is called \((k,S)\)-consistent if it is \((k - 1, 1)\)-consistent and it is called arc consistent if it is \( 2\)-consistent. The following formally defines singleton arc consistency.

Definition 3 (Singleton Arc Consistency). A CSP, \( \mathcal{P} \), with variables \( X \) is called singleton arc consistent (SAC) if

\[
(\forall x_1 \in X)(\forall v_i \in D(x_1)) \left( ac(\mathcal{P}) \models \{ x_1 = v_i \} \right) \]

The following seems a natural generalisation of SAC.

Definition 4 (k-Singleton Arc Consistency). A CSP \( \mathcal{P} \) with variables \( X \) is called k-singleton arc consistent if

\[
(\forall x_1 \in X)(\forall v_i \in D(x_1)) \left( \forall \{ x_2, \ldots, x_k \} \subseteq X \setminus \{ x_1 \} \right)
\]

\[
(\exists \{ v_2, \ldots, v_k \} \in D(x_2) \times \cdots \times D(x_k)) \left( ac(\mathcal{P}) \models \{ x_1 = v_i \} \right) \]

We define weak k-singleton arc consistency (weak k-SAC) by substituting an existential quantifier for the last universal quantifier in Definition 4. Then weak 1-SAC is equivalent to 1-SAC and SAC, and (weak) \( k\)-SAC implies (weak) inverse \( k\)-consistency and (weak) \( k\)-SAC if \( K \geq k \).

4 A Weak \( k\)-SAC Algorithm

This section presents our algorithms, which use greedy search to establish a weakly \( k\)-SAC equivalent of the input CSP \( \mathcal{P} \). They exploit that \( ac(\mathcal{P}) \models \{ x_1 = v_1, \ldots, x_k = v_k \} \) implies \( ac(\mathcal{P}) \models \{ x_1 = v_1, \ldots, x_k = v_k \} \) for any \( \{ x_1, \ldots, x_k \} \subseteq \{ x_1, \ldots, x_i \} \). This generalises the SAC algorithms in \([8]\), which use greedy search, exploiting that \( ac(\mathcal{P}) \models \{ x_1 = v_1, \ldots, x_n = v_n \} \) implies \( ac(\mathcal{P}) \models \{ x_1 = v_1, \ldots, x_n = v_n \} \) for any \( \{ x_1, \ldots, x_i \} \subseteq \{ x_1, \ldots, x_i \} \).

The algorithms are depicted in Figure 1. The outer \textbf{while} loop of \texttt{wksac} is executed while there are changes, while there is no inconsistency, and while no lucky solution has been found. (Removing the statement \texttt{solved := true in extendable} prohibits finding lucky solutions.) The second outer \textbf{while} loop selects the next variable, \( x \). The inner-most \textbf{while} loop removes singleton inconsistent values. For any remaining value, \( v \), \texttt{wksac} tries to extend the assignment \( x = v \), to some \( K\)-SAC assignment, for some \( K \geq k \) by executing \texttt{extendable}(\( k - 1 \), \texttt{wksac}, \( X \setminus \{ x \} \)). If this fails then \( x \) is removed. The underlying arc consistency algorithm is \texttt{ac}.

Like SAC3 and SAC3+ [\(8\)] extendable also searches for assignments of length greater than \( k \). This allows the discovery of lucky solutions [\(8\)] as part of consistency processing. If it finds a \( k\)-SAC assignment then \texttt{extendable} allows no digressions when trying to find an extending \( K\)-SAC assignment, for \( K > k \). This can be generalised to more digressions.

The space complexity of \texttt{wksac} is equal to the space complexity of \texttt{MAC} plus the space required for storing the array \texttt{wksac}[\( \cdot \cdot \cdot \)]. The space complexity of \texttt{wksac}[\( \cdot \cdot \cdot \)] is \( \mathcal{O}(n^d) \), which cannot exceed the space complexity of \texttt{MAC}. Therefore, the space complexity of \texttt{wksac} is equal to the space complexity of \texttt{MAC}. The outer loop of \texttt{wksac} is executed \( \mathcal{O}(n^d) \) times. For
each of the $O(n\,d)$ values, finding an extending $k$-SAC assignment takes $O(d^{k-1}\,T)$ time, where $T$ is the time complexity of $ac$. For each $k$-SAC assignment it takes $O((n-k)\,T)$ time for trying to find a lucky solution. Therefore, $ac$'s time complexity is $O(n^3\,d^{k+2}\,T + (n-k)\,n^2\,d^2\,T)$.

Termination and correctness proofs are straightforward. The following two propositions provide the basis for a correctness proof. Proofs are omitted due to space restrictions.

**Proposition 1.** If extendable($k-1$, $wksac$, solved, $X \setminus \{x\}$) succeeds then the assignment $x = v$ extends to a $k$-SAC assignment, for some $K \geq k$. Otherwise $x = v$ is inconsistent.

**Proposition 2.** If $wksac(k, X)$ succeeds then it computes a solution or some weakly $k$-SAC equivalent of the input CSP. If it fails then the input CSP is unsatisfiable.

---

**function** $wksac(k, \text{in} \ \text{X})$ do

local variables consistent, change, solved;
consistent := ac(); change := true; solved := false;
while consistent and change and ~solved do
foreach value $v$ in the domain of each variable $x$ do
wksac[$x$,$v$] := false;
enddo;
change := false; vars := X;
while consistent and ~change and ~solved and vars $\neq \emptyset$ do
Select and remove any variable $x$ from vars;
vals := \{$v \in D(x) : wksac[x, v]$\};
while vals $\neq \emptyset$ and consistent and ~change do
Select and remove any $v$ from vals;
assign $v$ to $x$;
if ac() and extendable($k-1$, $wksac$, solved, $X \setminus \{x\}$) then
wksac[$x$,$v$] := true;
else
remove $v$ from D($x$);
change := true; undo ac(); unassign $v$; consistent := ac();
fi;
fi;
od;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
do;
domeasub [3] is good for our algorithms. This heuristic uses the current degrees and the numbers of previously failed revisions, selecting a variable that quickly leads to a dead-end [3].
queen problems $wksac$ is removing more values as $k$ increases, but for $k = 16$ it needs too much time.

For these structured problems the new algorithms are about as efficient as SAC-1 for $k = 1$, if not better. All unsatisfiable modified $rlfap$ instances (the unsatisfiable unmodified instances are proved inconsistent by arc consistency) are proved inconsistent by $wksac$ for $k > 8$, whereas SAC-1 fails to prove some of these instances inconsistent. Note that instance scen11-f1 is very difficult and, to the best of our knowledge, has not been classified; days of MAC search could not solve it. However, enforcing weak 8-SAC proves it inconsistent within an hour. Finally, some satisfiable problems (they are not all listed in Table 1) are made and proved inverse consistent within a few seconds. Here a problem is proved inverse consistent if all values participate to some lucky solution. For example, scen2 is proved to be already inverse consistent, $graph2$, scen3, scen4, and scen5 are made and proved inverse consistent by making them weak $k$-SAC ($k \geq 1$), and all fifteen instances from the classes $frb=30-15$, $frb=35-17$, and $frb=40-19$ are made and proved inverse consistent by making them weakly $k$-SAC for $k = 8$ or $k = 16$.

**Shaving Random Problems**

We now study the behaviour of the algorithms for random B problems [7]. A model B class is typically denoted $(n, d, D, T)$, where $n$ is the number of variables, $d$ is the uniform domain size, $D$ is the density, and $T$ is the uniform tightness. Results are presented for the same class of problems as presented in [1; 8]. For each tightness $t = 0.05i$, $2 \leq i \leq 18$, the average shaving time is over 50 random instances from $(100, 20, 0.05, t)$.

Figure 3 does not depict all data. For $T \leq 0.50$ all algorithms remove less than 0.06, and for $T > 0.8$ they remove about 1989.6 values on average. Figure 3 confirms that SAC and weak 1-SAC are equivalent. Comparing SAC-1 and $visac$ in Figure 2, $visac$ is better in time when $T$ is small and large. However, near the SAC complexity peak SAC-1 is about three times quicker than $visac$. The majority of the problems in that region are unsatisfiable and most values are SAC. Typically, SAC-1 will find that a value is SAC and stop. The extra work put in by $visac$ in trying to find a lucky solution will fail at a shallow level. This work cannot do pruning and does not lead to many values that were not known to be SAC.

The weak $k$-SAC algorithms only behave differently near the SAC phase transition. The higher $k$ the more values are removed. At $T = 0.7$, the nearest point to the SAC phase transition, it is observed that weak16sac outperforms SAC-1 marginally in time and significantly in the number of removed values. At $T = 0.65$ w1sac removes about 114.2 values on average, whereas all other algorithms remove between 2 and 3 values on average. However, w1sac spends (much) more time. Clearly the algorithms cannot be compared in time at $T = 0.65$.

**Search**

We now compare the algorithms as a preprocessor for MAC. The first solver is MAC, denoted mac, the second is $sac+mac$, which is SAC-1 followed by MAC, and the third and fourth solvers are $visac+mac$ and $wksac+mac$, i.e. $wksac$ for enforcing weak $k$-singleton arc consistency followed by MAC, for $k \in \{1, 8\}$. All solvers used the variable ordering dom/wdom [3] and the value ordering $svbb2$ [10], breaking ties lexically. The support counters [10] for the value ordering are initialised after establishing initial arc consistency. Should $wksac$ prune more values, then they are also initialised before search.

Table 2 lists the results. The column sat denotes the satisfiability of the instances. A + and an L indicates satisfiable instances, the L indicating the discovery of lucky solutions. For all problems, if lucky solutions are found by $visac+mac$ then they are also found by $wksac+mac$ and vice versa.

Overall, and these results are typical for these problems, $sac+mac$ performs worse in time and checks than $visac+mac$. MAC performs better than $sac+mac$ for some instances, especially some $graph$ instances, otherwise the two algorithms are about equally efficient. Compared to $visac+mac$ and $wksac+mac$ it is clear that $sac+mac$ is worse in time and checks. Compared to MAC the results are not so clear but overall $visac+mac$ and $wksac+mac$ are better. The only exceptions are for "easy" problems, for which MAC is easy. For these problems there is no need to establish more consistency before search and this results in slightly more solution time. We have also observed this for unsatisfiable problem instances where MAC alone is sufficient to detect the inconsistency. For example, for the queens-knight problems $q1k$ [2] SAC-1 is more efficient than the weak SAC algorithms. However, these problems are relatively easy and do not require much time, even with $wksac$. For the problems that are difficult for MAC and $sac+mac$ the two need a considerable amount of time more than $wksac$.

It is recalled that it turned out impossible to compute the weak $k$-SAC equivalent of the job-shop instances for $k = 8$ and $k = 16$. However, when the algorithms are used to find solutions, they perform much better and find lucky solutions. Lucky solutions are also found for all satisfiable $rlfap$, and all
satisfiable modified r1fap instances, including instances from these classes for which no results are presented in Table 1. It is interesting to note that if lucky solutions are found, it takes the same amount of checks for $k = 8$ and $k = 16$. This may indicate that both algorithms carry out exactly the same decisions about value and variable ordering. If this is true, then it is probably because these problems are loose, making any arc consistent 1-assignment, easily extendable to an arc consistent k-assignment for $k \geq 8$, which makes it impossible to prune more for v8sac than for visac.

Table 2. Problem solving capabilities of search algorithms.

<table>
<thead>
<tr>
<th>Problem</th>
<th>SAC</th>
<th>MAC</th>
<th>Heuristics</th>
<th>WKSAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>scen1-f8</td>
<td>-0.36</td>
<td>0.38</td>
<td>0.38</td>
<td>-0.36</td>
</tr>
<tr>
<td>scen1-f9</td>
<td>0.31</td>
<td>1.23</td>
<td>1.23</td>
<td>0.31</td>
</tr>
<tr>
<td>scen2-f24</td>
<td>-0.16</td>
<td>1.97</td>
<td>1.97</td>
<td>-0.16</td>
</tr>
<tr>
<td>scen2-f25</td>
<td>-0.16</td>
<td>1.97</td>
<td>1.97</td>
<td>-0.16</td>
</tr>
<tr>
<td>scen5</td>
<td>0.13</td>
<td>1.52</td>
<td>1.52</td>
<td>0.13</td>
</tr>
<tr>
<td>scen6-w1</td>
<td>-0.84</td>
<td>1.74</td>
<td>1.74</td>
<td>-0.84</td>
</tr>
<tr>
<td>scen6-w1-f2</td>
<td>0.04</td>
<td>0.45</td>
<td>0.45</td>
<td>0.04</td>
</tr>
<tr>
<td>scen6-w1-f3</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen6-w2</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w1-f4</td>
<td>0.04</td>
<td>0.45</td>
<td>0.45</td>
<td>0.04</td>
</tr>
<tr>
<td>scen7-w1-f5</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w1-f6</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w2-f1</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w2-f2</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w2-f3</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w2-f4</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w2-f5</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w2-f6</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w3-f1</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w3-f2</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w3-f3</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w3-f4</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w3-f5</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
<tr>
<td>scen7-w3-f6</td>
<td>0.03</td>
<td>0.45</td>
<td>0.45</td>
<td>0.03</td>
</tr>
</tbody>
</table>

6 Conclusions

This paper introduces k-singleton consistency (k-SAC) and weak k-SAC, which are generalisations of SAC and inverse k-consistency. Weak k-SAC is equal to SAC if $k = 1$ but stronger if $k > 1$. A weak k-SAC algorithm is presented, which uses greedy search. At the SAC phase transition, it removes many more values than SAC-1 for $k = 16$ using less time.

Figure 2. Shaving time for random B class $(100, 20, 0.05, t)$ for different algorithms, where $0.1 \leq T \leq 0.9$.

Figure 3. Number of shaved values for random B class $(100, 20, 0.05, t)$ for different algorithms, where $0.5 \leq T \leq 0.8$.

REFERENCES


