# 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 $T \subseteq X \setminus S$ 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 $\mathcal{O}\left(\binom{n}{s} d^s\right) = \mathcal{O}\left(n^s\, d^s\right)$ 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 arising 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].

[1] Cork Constraint Computation Centre (4C). 4C is supported by Science Foundation Ireland under Grant 00/PI.1/C075.

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 $T \subseteq X \setminus S$ 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, $\mathcal{P}$, 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 $\mathcal{P}$ can be made arc consistent. SAC is equivalent to $k$-singleton 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 *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 $\langle \sigma, \rho \rangle$, where $\sigma = \langle x, y \rangle \in X^2$ 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 $\mathcal{P} \neq \bot$ 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 $D(x) \times \{ w \} \cap \rho \neq \emptyset$. 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 $2\,e/(n^2 - n)$. 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 $\{ x_1 = f(x_1), \ldots, x_k = f(x_k) \}$ 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 $v_1 \in D(x_1)$ or $k > 1$ and $\langle v_i, v_j \rangle \in \rho$ 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*. $\mathcal{P}|_f$ 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 $\mathcal{O}(n^s\, d^s)$ 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* [6] 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-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*

$$(\forall x_1 \in X\,)(\,\forall v_1 \in D(x_1)\,)(\,ac(\mathcal{P}|_{\{\,x_1 = v_1\,\}}) \neq \bot\,)\,.$$

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*

$$\begin{aligned}
&(\forall x_1 \in X\,)(\,\forall v_1 \in D(x_1)\,)(\forall \{\,x_2, \ldots, x_k\,\} \subseteq X \setminus \{\,x_1\,\}\,)\\
&\quad(\,\exists \langle\,v_2, \ldots, v_k\,\rangle \in D(x_2) \times \ldots \times D(x_k)\,)\\
&\qquad(\,ac(\mathcal{P}|_{\{\,x_1 = v_1, \ldots, x_k = v_k\,\}}) \neq \bot\,)\,.
\end{aligned}$$

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 $ac(\mathcal{P}|_{\{\,x_1 = v_1, \ldots, x_l = v_l\,\}}) \neq \bot$ implies $ac(\mathcal{P}|_{\{\,x_{i_1} = v_{i_1}, \ldots, x_{i_k} = v_{i_k}\,\}}) \neq \bot$, for any $\{\,x_{i_1}, \ldots, x_{i_k}\,\} \subseteq \{\,x_1, \ldots, x_l\,\}$. This generalises the SAC algorithms in [8], which use greedy search, exploiting that $ac(\mathcal{P}|_{\{\,x_1 = v_1, \ldots, x_l = v_l\,\}}) \neq \bot$ implies $ac(\mathcal{P}|_{\{\,x_{i_1} = v_{i_1}\,\}}) \neq \bot$, for any $\{\,x_{i_1}\,\} \subseteq \{\,x_1, \ldots, x_l\,\}$.

The algorithms are depicted in Figure 1. The outer **while** loop of *wksac* is executed while there are changes, while there is no inconsistency, and while no lucky solution has been found. (Removing the statement *solved* := *true* in *extendable* prohibits finding lucky solutions.) The second outer **while** loop selects the next variable, $x$. The inner-most **while** loop removes singleton inconsistent values. For any remaining value, $v$, *wksac* tries to extend the assignment $x = v$, to some $K$-SAC assignment, for some $K \geq k$ by executing *extendable*$(k - 1, \textit{wksac}, \textit{solved}, X \setminus \{\,x\,\})$. If this fails then $v$ is removed. The underlying arc consistency algorithm is *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 *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 *wksac* is equal to the space complexity of MAC plus the space required for storing the array *wksac*$[\,\cdot, \cdot\,]$. The space complexity of *wksac*$[\,\cdot, \cdot\,]$ is $\mathcal{O}(n\,d)$, which cannot exceed the space complexity of MAC. Therefore, the space complexity of *wksac* is equal to the space complexity of MAC. The outer loop of *wksac* is executed $\mathcal{O}(n\,d)$ times. For each of the $\mathcal{O}(n\,d)$ values, finding an extending $k$-SAC assignment takes $\mathcal{O}\left(d^{k-1}\,T\right)$ time, where $T$ is the time complexity of *ac*. For each $k$-SAC assignment it takes $\mathcal{O}((n-k)\,T)$ time for trying to find a lucky solution. Therefore, *wksac*'s time complexity is $\mathcal{O}\left(n^2\,d^{k+1}\,T + (n-k)\,n^2\,d^2\,T\right)$.

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(in k, in 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;
    od;
    change := false; vars := X;
    while consistent and ¬change and ¬solved and vars ≠ ∅ do
      Select and remove any variable x from vars;
      vals := { v ∈ D(x) : ¬wksac[ x, v ] };
      while vals ≠ ∅ 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 \ { x })
          undo ac(); unassign v; wksac[ x, v ] := true;
        else
          remove v from D(x);
          change := true; undo ac(); unassign v; consistent := ac();
        fi;
      od;
    od;
  od;
  return consistent;
od;

function extendable(in k, in/out wksac, in/out solved, in X) do
  local variables consistent, extendable, values;
  if X = ∅
    solved := true; /* Remove line if not used for solving. */
    return true;
  else
    select any x from X;
    values := D(x);
    extendable := false;
    while ¬extendable and values ≠ ∅ do
      select and remove any v from values;
      assign v to x;
      if ac() and extendable(k − 1, wksac, solved, X \ { x }) then
        wksac[ x, v ] := true; extendable := true;
      fi;
      undo ac(); unassign v;
    od;
    return extendable or k ≤ 0;
  fi;
od;
```

**Figure 1.** Algorithms for enforcing weak $k$-SAC.

The algorithms in Figure 1 do not represent efficient implementations. The following are suggestions for improvement.

The computation time usually improves if *extendable* first selects variables from the set *vars* used by *wksac*, and if it first selects $v \in D(x)$ such that $wksac[\,x, v\,] = false$.

Finally, note that a good variable ordering is essential. For example, given a CSP consisting of $k$ connected components, it makes no sense to select one variable from each component since *any* assignment to these variables is consistent, not allowing any pruning. The conflict directed variable order $dom/w_{deg}$ [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]. Remember that $k = 1$. If *extendable* has not been called yet then a dead-end immediately proves the assignment $x = v$ inconsistent. Otherwise, it immediately proves $x = v$ weakly $K$-SAC, for some $K \geq k$. Ties are broken lexically. Values for $x$ are selected preferring value $v$ such that $wksac[\,x, v\,] = false$, breaking ties using the order $svoh_2$ [10]. Other variable and value orders usually result in worse results.

For certain CSPs and $K > k > 0$, *wksac* may prune *more* for $k$ than for $K$. However, experimental evaluation of the algorithm indicates that this usually is not the case.

## 5 Experimental Results

To investigate their behaviour, we experimentally compared SAC-1, and *wksac* for $k \in \{\,1, 2, 4, 8, 16\,\}$. We (1) investigate their shaving capabilities for problems known from the literature, (2) investigate their behaviour for random problems, and (3) investigate their use as a preprocessor for MAC [12].

All algorithms were implemented in `C`. They were run on a `Linux` 2.8GHz PC, having 500Mb of RAM. The underlying arc consistency algorithm is AC-3. Some authors prefer optimal arc consistency algorithms, but we feel that AC-3 is a good general purpose algorithm. For example, the best solvers in the binary and overall categories of the First International CSP Solver Competition are based on AC-3 [13].

**Shaving Problems from the Literature** The algorithms have been applied to known problems, including forced random binary problems `frb`, RLFAP, modified RLFAP, attacking prime queen problems `qa1`, and job-shop instances `enddr1-10-by-5-10` and `enddr2-10-by-5-2` (they are called `js-1` and `js-2` in [8]). All problems are described in [2] and may be downloaded from `http://cpai.ucc.ie/`.

Table 1 lists the main results for the first experiment. It lists the problem instances, and for each instance: the number of variables, the number of values, the number of removed values, and the time this took for SAC-1 and *wksac* for $k \in \{\,1, 2, 4, 8, 16\,\}$. The satisfiable instances are indicated by a + in the column `sat`. An `i` in the column `del` indicates that the CSP has been made and proved inverse consistent. The column `min` (`max`) lists the minimum (maximum) Cardinalities of sets of lucky solutions, which were found in the process of enforcing weak 1-SAC. These sets are found when there are no more constraints among the future variables, in which case the number of solutions is equal to the product of the domain sizes. The column `count` lists the total number of such solutions. Due to the number of sets of lucky solutions and their size, it was impossible, for some problem classes, to compute the actual number of lucky solutions, in which case the entry for column `count` has a question mark. [8] also report the finding of lucky solutions. [8] report 0 lucky solutions for `qa-5`, `qa-6`, 1 for `scen5` and `enddr2-10-by-5-2`, 4 for `enddr1-10-by-5-10`, 10 for `graph14`, and 16 for `scen2`, which is less than the numbers reported here. Perhaps these differences are caused by differences in variable and value ordering.

Many values in the job-shop problems are inverse consistent. At the First International CSP Solver Competition they were difficult to solve by MAC [13]. This may explain why it is difficult to enforce higher level of $k$-SAC using search: for $k \in \{\,8, 16\,\}$ it takes too much time. For the attacking prime

**Table 1.** Results of shaving problems from the literature using SAC-1 and *wksac* for different values of $k$

| | | | | SAC-1 | | lucky | | | $k=1$ | | $k=2$ | | $k=4$ | | $k=8$ | | $k=16$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | sat | vars | vals | del | time | count | min | max | del | time | del | time | del | time | del | time | del | time |
| frb30-15-1 | + | 30 | 450 | 0 | 0.04 | 0 | 0 | 0 | 0 | 0.10 | 0 | 0.10 | 0 | 0.09 | i 410 | 6.55 | i 410 | 0.54 |
| frb30-15-2 | + | 30 | 450 | 0 | 0.04 | 0 | 0 | 0 | 0 | 0.10 | 0 | 0.09 | 0 | 0.10 | i 413 | 7.02 | i 413 | 1.16 |
| frb35-17-1 | + | 35 | 595 | 0 | 0.05 | 0 | 0 | 0 | 0 | 0.14 | 0 | 0.14 | 0 | 0.14 | 0 | 1.72 | i 559 | 3.19 |
| frb35-17-2 | + | 35 | 595 | 0 | 0.06 | 0 | 0 | 0 | 0 | 0.14 | 0 | 0.14 | 0 | 0.14 | 0 | 0.90 | i 552 | 7.54 |
| frb40-19-1 | + | 40 | 760 | 0 | 0.08 | 0 | 0 | 0 | 0 | 0.21 | 0 | 0.21 | 0 | 0.21 | 0 | 0.51 | i 701 | 15.05 |
| frb40-19-2 | + | 40 | 760 | 0 | 0.07 | 0 | 0 | 0 | 0 | 0.22 | 0 | 0.21 | 0 | 0.21 | 0 | 0.52 | i 717 | 28.18 |
| scen-1 | + | 916 | 36200 | 0 | 20.90 | 42 | 1 | 1 | 0 | 6.46 | 0 | 6.47 | 0 | 6.47 | 0 | 6.48 | 0 | 6.48 |
| scen-2 | + | 200 | 8004 | 0 | 4.89 | 56 | 1 | 1 | i 0 | 1.28 | i 0 | 1.28 | i 0 | 1.29 | i 0 | 1.29 | i 0 | 1.28 |
| scen-5 | + | 400 | 15768 | 13814 | 0.97 | 46 | 1 | 1 | i 13814 | 0.28 | i 13814 | 0.28 | i 13814 | 0.28 | i 13814 | 0.28 | i 13814 | 0.28 |
| scen-11 | + | 680 | 26856 | 0 | 15.17 | 6 | 1 | 1 | 0 | 4.88 | 0 | 4.89 | 0 | 4.88 | 0 | 4.78 | 76 | 4803.89 |
| scen1-f8 | + | 916 | 29496 | 6704 | 6.29 | 17 | 1 | 1 | 6704 | 2.55 | 6704 | 2.55 | 6704 | 2.56 | 6704 | 2.55 | 6704 | 2.58 |
| scen2-f24 | + | 200 | 4025 | 0 | 0.57 | 25 | 1 | 1 | 0 | 0.27 | 0 | 0.27 | 0 | 0.27 | 0 | 0.32 | 0 | 0.42 |
| scen3-f10 | + | 400 | 12174 | 3726 | 2.20 | 22 | 1 | 1 | 3726 | 1.43 | 3726 | 1.43 | 3738 | 1.56 | 3738 | 1.54 | 3746 | 1.78 |
| scen6-w1 | + | 200 | 8020 | 1580 | 2.78 | 58 | 1 | 1 | 1580 | 0.78 | 1580 | 0.77 | 1616 | 0.76 | 1616 | 0.76 | 1616 | 0.80 |
| scen7-w1-f4 | + | 400 | 14568 | 6286 | 1.88 | 62 | 1 | 1 | 6286 | 1.10 | 6286 | 1.10 | 6318 | 0.90 | 6318 | 0.89 | 6326 | 1.15 |
| scen1-f9 | - | 916 | 28596 | 7628 | 5.24 | 0 | 0 | 0 | 7628 | 2.47 | 7628 | 2.47 | 7640 | 4.75 | 28596 | 2.46 | 28596 | 2.89 |
| scen2-f25 | - | 200 | 3918 | 106 | 0.58 | 0 | 0 | 0 | 106 | 0.35 | 106 | 0.36 | 110 | 0.40 | 3918 | 0.54 | 3918 | 1.50 |
| scen3-f11 | - | 400 | 11966 | 3934 | 1.59 | 0 | 0 | 0 | 3934 | 1.79 | 3934 | 1.79 | 3980 | 1.85 | 11966 | 1.36 | 11966 | 0.41 |
| scen6-w1-f2 | - | 200 | 7716 | 2082 | 2.34 | 0 | 0 | 0 | 2082 | 1.25 | 2082 | 1.25 | 7716 | 0.40 | 7716 | 0.18 | 7716 | 0.22 |
| scen6-w1-f3 | - | 200 | 7518 | 2474 | 1.59 | 0 | 0 | 0 | 2474 | 0.59 | 2474 | 0.59 | 7518 | 0.07 | 7518 | 0.07 | 7518 | 0.09 |
| scen11-f1 | - | 680 | 26524 | 332 | 13.94 | 0 | 0 | 0 | 332 | 4.47 | 332 | 4.47 | 332 | 4.39 | 26524 | 3097.10 | | |
| qa-5 | + | 26 | 631 | 9 | 0.16 | 0 | 0 | 0 | 9 | 0.10 | 9 | 0.10 | 9 | 0.10 | 12 | 0.45 | i 386 | 24.06 |
| qa-6 | + | 37 | 1302 | 48 | 2.19 | 0 | 0 | 0 | 48 | 0.78 | 48 | 0.80 | 48 | 0.81 | 67 | 1.86 | 127 | 1923.01 |
| qa-10 | + | 101 | 10015 | 373 | 311.92 | 0 | 0 | 0 | 373 | 93.91 | 373 | 94.64 | 393 | 440.62 | 416 | 306.66 | — | |
| enddr1-10-by-5-10 | + | 50 | 5760 | 0 | 16.07 | ? | 532 | 15.3e6 | 0 | 5.85 | 0 | 5.83 | 0 | 5.83 | — | | — | |
| enddr2-10-by-5-2 | + | 50 | 6315 | 0 | 25.95 | ? | 21 | 50.5e3 | 0 | 10.68 | 0 | 10.70 | 0 | 10.70 | — | | — | |

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 $k \geq 8$, 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 model B problems [7]. A model B class is typically denoted $\langle\, n, d, D, T\,\rangle$, 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 $t = 0.05\, i$, $2 \leq i \leq 18$, the average shaving time is over 50 random instances from $\langle\, 100, 20, 0.05, t\,\rangle$.

Figure 3 does not depict all data. For $T \leq 0.50$ all algorithms remove less than 0.06, and for $T \geq 0.8$ they remove about 1989.6 values on average. Figure 3 confirms that SAC and weak 1-SAC are equivalent. Comparing SAC-1 and `w1sac` in Figure 2, `w1sac` is better in time when $T$ is small and large. However, near the SAC complexity peek SAC-1 is about three times quicker than `w1sac`. 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 `w1sac` 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, `weak16sac` outperforms SAC-1 marginally in time and significantly in the number of removed values. At $T = 0.65$ `w16sac` removes about 114.2 values on average, whereas all other algorithms remove between 2 and 3 values on average. However, `w16sac` 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 `w1sac+mac` and `w8sac+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 $dom/w_{deg}$ [3] and the value ordering $svoh_2$ [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 `w1sac+mac` then they are also found by `w8sac+mac` and vice versa.

Overall, and these results are typical for these problems, `sac+mac` performs worse in time and checks than `wksac+mac`. `Mac` performs better than `sac+mac` for some instances, especially some `graph` instances, otherwise the two algorithms are about equally efficient. Compared to `w1sac+mac` and `w8sac+mac` it is clear that `sac+mac` is worse in time and checks. Compared to `mac` the results are not so clear but overall `w1sac+mac` and `w8sac+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 `sac` alone is sufficient to detect the inconsistency. For example, for the queens-knight problems `qk1` [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

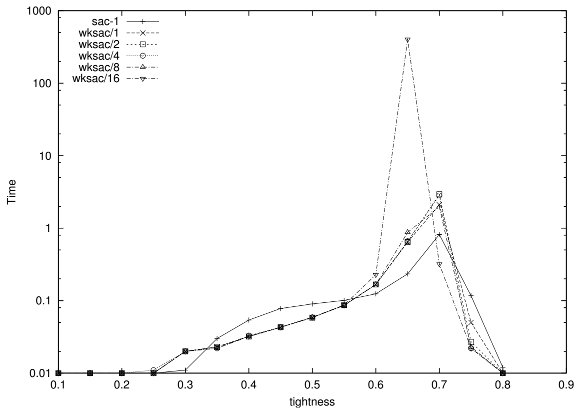


**Figure 2.** Shaving time for random B class $\langle 100, 20, 0.05, t \rangle$ for different algorithms, where $0.1 \leq T \leq 0.9$

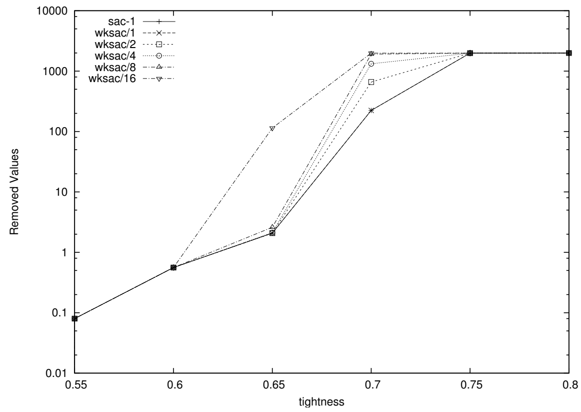


**Figure 3.** Number of shaved values for random B class $\langle 100, 20, 0.05, t \rangle$ for different algorithms, where $0.5 \leq T \leq 0.8$

satisfiable modified `rlfap` 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 amount of work, which seems possible only if, in addition, they make 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 `w8sac` than for `w1sac`.

**Table 2.** Problem solving capabilities of search algorithms.

| | | mac | | sac+mac | | w1sac+mac | | w8sac+mac | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | | time | checks | time | checks | time | checks | time | checks |
| scen2 | L | 0.15 | 2.525e6 | 3.62 | 78.121e6 | 0.16 | 2.511e6 | 0.16 | 2.511e6 |
| scen5 | L | 0.09 | 0.564e6 | 0.92 | 8.199e6 | 0.10 | 0.595e6 | 0.10 | 0.595e6 |
| scen11 | L | 0.61 | 9.296e6 | 11.49 | 246.425e6 | 0.53 | 8.545e6 | 0.53 | 8.545e6 |
| scen1-f8 | L | 0.39 | 4.810e6 | 5.11 | 86.210e6 | 0.41 | 4.987e6 | 0.41 | 4.987e6 |
| scen2-f24 | L | 0.03 | 0.723e6 | 0.49 | 10.923e6 | 0.05 | 0.961e6 | 0.05 | 0.949e6 |
| scen6-w1 | L | 0.04 | 0.573e6 | 2.24 | 33.830e6 | 0.04 | 0.572e6 | 0.04 | 0.572e6 |
| scen7-w1-f4 | L | 0.07 | 0.897e6 | 1.54 | 22.930e6 | 0.08 | 0.996e6 | 0.08 | 0.996e6 |
| scen1-f9 | - | 0.84 | 8.857e6 | 4.56 | 70.602e6 | 2.51 | 25.310e6 | 2.29 | 22.107e6 |
| scen2-f25 | - | 1.44 | 15.261e6 | 1.91 | 25.102e6 | 0.43 | 5.667e6 | 0.54 | 5.688e6 |
| scen3-f11 | - | 0.88 | 8.046e6 | 2.16 | 32.568e6 | 1.94 | 20.443e6 | 1.34 | 13.450e6 |
| scen6-w1-f2 | - | 2.09 | 25.712e6 | 2.08 | 28.949e6 | 1.11 | 13.898e6 | 0.15 | 1.908e6 |
| scen6-w1-f3 | - | 0.84 | 11.878e6 | 1.37 | 19.030e6 | 0.53 | 6.631e6 | 0.06 | 0.813e6 |
| scen6-w2 | - | 0.36 | 4.382e6 | 0.35 | 4.290e6 | 0.15 | 1.768e6 | 0.16 | 1.980e6 |
| scen7-w1-f5 | - | 0.11 | 1.235e6 | 0.08 | 1.032e6 | 0.06 | 0.731e6 | 0.06 | 0.731e6 |
| scen9-w1-f3 | - | 0.16 | 1.972e6 | 0.11 | 1.432e6 | 0.13 | 1.654e6 | 0.13 | 1.664e6 |
| scen10-w1-f3 | - | 0.16 | 1.972e6 | 0.11 | 1.432e6 | 0.13 | 1.654e6 | 0.13 | 1.664e6 |
| graph3 | L | 0.17 | 2.185e6 | 60.11 | 764.413e6 | 0.17 | 2.180e6 | 0.17 | 2.180e6 |
| graph10 | L | 0.72 | 8.501e6 | 368.56 | 198.230e6 | 0.72 | 8.491e6 | 0.72 | 8.491e6 |
| graph14 | L | 0.53 | 9.286e6 | 15.11 | 351.031e6 | 0.53 | 9.230e6 | 0.54 | 9.230e6 |
| qa-5 | + | 0.29 | 2.392e6 | 0.41 | 5.501e6 | 0.42 | 3.652e6 | 0.06 | 0.806e6 |
| qa-6 | + | 106.65 | 783.722e6 | 125.22 | 949.601e6 | 0.81 | 12.807e6 | 1.92 | 22.346e6 |
| enddr1-10-by-5-10 | L | 879.89 | 546.428e6 | 891.26 | 974.780e6 | 0.13 | 4.405e6 | 0.13 | 4.405e6 |
| enddr2-10-by-5-2 | L | 117.47 | 3925.611e6 | 134.53 | 321.808e6 | 0.20 | 7.304e6 | 0.20 | 7.304e6 |

## 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. Like SAC-3 and SAC-3+ the algorithm sometimes finds lucky solutions. If it does it usually find many. SAC cannot solve unsatisfiable instances which can be made SAC. However, by enforcing higher degrees of weak SAC some of these instances can be proved inconsistent within a reasonable time. When used as a preprocessor for MAC the algorithm compares favourably to existing algorithms. By increasing the level of weak SAC we are able to solve difficult problems, including `scen11-f1`, which had not been solved before.

## REFERENCES


[1] C. Bessière and R. Debruyne, 'Optimal and suboptimal singleton arc consistency algorithms', in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pp. 54–59, (2005).

[2] F. Boussemart, F. Hemery, and C. Lecoutre, 'Description and representation of the problems selected for the first international constraint satisfaction problem solver competition', in *Proceedings of the Second International Workshop on Constraint Propagation And Implementation*, volume II, pp. 7–26, (2005).

[3] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs, 'Boosting systematic search by weighting constraints.', in *Proceedings of the Sixteenth Eureopean Conference on Artificial Intelligence*, eds., Ramon López de Mántaras and Lorenza Saitta, pp. 146–150, (2004).

[4] R. Debruyne and C. Bessière, 'Some practicable filtering techniques for the constraint satisfaction problem', in *Procceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, ed., M.E. Pollack, pp. 412–417, (1997).

[5] R. Debruyne and C. Bessière, 'Domain filtering consistencies', *Journal of Artificial Intelligence Research*, **14**, 205–230, (2001).

[6] E.C. Freuder, 'A sufficient condition for backtrack-bounded search', *Journal of the ACM*, **32**(4), 755–761, (1985).

[7] I.P. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh, 'Random constraint satisfaction: Flaws and structure', *Journal of Constraints*, **6**(4), 345–372, (2001).

[8] C. Lecoutre and S. Cardon, 'A greedy approach to establish singleton arc consistency', in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, (2005).

[9] A.K. Mackworth, 'Consistency in networks of relations', *Artificial Intelligence*, **8**, 99–118, (1977).

[10] D. Mehta and M.R.C. van Dongen, 'Static value ordering heuristics for constraint satisfaction problems', in *Proceedings of the Second International Workshop on Constraint Propagation And Implementation*, ed., M.R.C. van Dongen, pp. 49–62, (2005).

[11] P. Prosser, K. Stergiou, and T. Walsh, 'Singleton consistencies', in *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming, CP'2000*, ed., R. Dechter, pp. 353–368, (2000).

[12] D. Sabin and E.C. Freuder, 'Contradicting conventional wisdom in constraint satisfaction', in *Proceedings of the Eleventh European Conference on Artificial Intelligence*, ed., A.G. Cohn, pp. 125–129. John Wiley and Sons, (1994).

[13] *Proceedings of the Second International Workshop on Constraint Propagation And Implementation*, ed., M.R.C. van Dongen, volume II, 2005.