AI Communications 28 (2015) 345–364
DOI 10.3233/AIC-140635
IOS Press

SAC and neighbourhood SAC

Richard J. Wallace
Cork Constraint Computation Centre and Department of Computer Science, University College Cork, Cork, Ireland
E-mail: r.wallace@4c.ucc.ie

Abstract. This paper introduces a new kind of local consistency based on the general idea of singleton arc consistency (SAC). This is a reduced form of SAC that only considers neighbourhoods of a variable with a singleton domain; hence, the name "neighbourhood SAC" (NSAC). Like AC and SAC, NSAC has a unique fixpoint, so that NSAC algorithms will produce the same result when applied to a problem regardless of the order in which problem elements are processed. Although NSAC is, of course, dominated by full SAC, on many problems these algorithms produce almost as much filtering with significantly less cost. NSAC can also be incorporated into full search, as a maintained neighbourhood SAC algorithm. The implementation of NSAC has also inspired two new SAC algorithms. One is a full SAC algorithm that is somewhat more efficient than the classical SAC-1 algorithm on many problems and is much easier to code than more advanced versions of SAC. The other is a partial SAC procedure that performs almost as much domain reduction as full SAC, while requiring much less time than full SAC algorithm on some problems. It is, therefore, a possible alternative to restricted SAC-1. These new algorithms are evaluated in experimental tests, together with SAC-1 and three well-known advanced SAC procedures, on a variety of problem classes.

Keywords: Constraint satisfaction, arc consistency, singleton arc consistency

1. Introduction

Singleton arc consistency (SAC) is a powerful enhancement of the best-known type of local consistency algorithm. In this variant of arc consistency, each value associated with a variable is considered as a singleton domain, and arc consistency is carried out under this assumption. Under these conditions, a failure, in the form of a domain wipeout, implies that there is no solution containing this value; hence it can be discarded.

Although SAC sometimes removes appreciably more values than ordinary arc consistency (AC), its drawback is that it is very expensive. It is therefore mostly used as a preprocessing step, but even then the overhead is appreciable. Hence, it is worthwhile to consider new variations on the classical procedure as well as approximations to full SAC that are more powerful than AC for the same reasons, but are less expensive than full SAC algorithms.

Several SAC algorithms have been proposed in the past. The original algorithm proposed by [4] used a multiple pass procedure in the manner of AC-1; hence, it is referred to as SAC-1. [1] proposed a version of SAC (SAC-2) that uses support counts in the manner of AC-4. More recently, [2] presented SAC-SDS, which is based on an optimal-worst-time SAC algorithm (SAC-Opt). Both SAC-SDS and SAC-Opt use multiple copies of the original problem in order to avoid starting from scratch on a single copy multiple times. [8] developed a greedy form of SAC called SAC-3. The idea behind this algorithm is to extend a singleton value to a singleton series (a "branch") until this gives an arc-inconsistent problem. With this strategy, values added to the branch after the first are checked against the problem reduced by the previous singleton values, which reduces the amount of consistency checking required.

The present work describes a new *form* of singleton arc consistency, called neighbourhood singleton arc consistency (NSAC), in which SAC is established in relation to the neighbourhood of each variable. This form of consistency can be established with much greater efficiency than SAC. Although dominated by SAC, in many cases establishing NSAC serves to remove nearly as many domain values as establishing full singleton arc consistency. Two basic strategies for establishing NSAC are described, one that uses the SAC-1 procedure, and one which uses a single queue of variables in AC-3 style, with appropriate queue additions to ensure that the requisite consistency is established.

The strategy used in the latter NSAC algorithm can also be used to establish full SAC, thus giving rise to a new SAC algorithm, referred to as SACQ. This algorithm avoids some of the redundancy involved in SAC-1 in much the same way that AC-3 avoids some of the redundancy of AC-1. In addition, a form of SACQ that approximates full SAC can be devised. In this procedure, called SACQ-adj, only the neighbours of a variable undergoing revision are added back to the queue.

These algorithms are compared with algorithms devised previously, both theoretically and experimentally. We show that SACQ is equivalent to classical SAC algorithms in its effects, and that NSAC is dominated by the neighbourhood inverse consistency algorithm (NIC). We then show that for random problems, SACQ and SACQ-adj both outperform SAC-1, as well as the more elaborate forms of SAC described above. In the case of SACQ-adj, there is an appreciable reduction of overall effort (reflected in CPU time) when compared with SACQ, although very little is given up in terms of values deleted. In this respect, it improves on the restricted SAC-1 algorithm proposed by [11], although the latter requires less time. Similar results are found for NSAC in comparison with NIC for random problems of the standard type, although for random problems with heterogeneous features, NIC is much more effective than NSAC in deleting values and is even better than SAC in this case.

SAC-1 and SACQ also give comparable results on a variety of structured problems, although again the latter is more efficient for some problem classes. In many cases, SAC-SDS and especially SAC-3 outperform these algorithms (SAC-2 never does). However, for both advanced algorithms there is evidence of scaling difficulties (severe for SAC-SDS), so that SACQ can be considered a viable choice as a general-purpose SAC algorithm.

The next section gives general background concepts and definitions. Section 3 describes the present implementation of SAC-1 as well as giving brief descriptions of the operation of the advanced SAC algorithms and their present implementations. Section 4 introduces the neighbourhood singleton arc consistency (NSAC) algorithms and describes their properties, while Section 5 discusses their relations with SAC and NIC. Section 6 describes a new full SAC algorithm (SACQ) that uses some of the same strategies and data structures as algorithm NSACQ. We also describe a queue-ordering heuristic that can be used with SACQ and the SACQ-adj approximation. Section 7 describes results of experiments on random problems, including a series with graded tightness that has been used in earlier studies of SAC. Section 8 describes experimental results for a variety of structured CSPs, including publicly available benchmarks as well as "random relop" problems devised by the author. Section 9 gives some preliminary results for a maintained neighbourhood SAC algorithm. Section 10 gives conclusions.

2. Background concepts

A constraint satisfaction problem (CSP) involves assigning values to a set of variables subject to restrictions on way that values can go together.

More formally, a CSP can be defined as a tuple, (X, D, C) where:

> X is a set of *variables*, $X_1, \dots, X_n$.
> D is a set of *domains*, D_i, where each D_i is a set of possible *values* (or labels) that can be assigned to variable X_i.
> C is a set of *constraints* among subsets of the variables belonging to X. Each C_i belonging to C consists of a relation R_i and a particular subset of the variables in X, *vars*(C_i), called the *scope* of the constraint. R_i is based on the Cartesian product of the values of the domains of the variables in the scope of C_i.
> A *solution* to a CSP is an assignment or mapping from variables to values, $A = \{(X_1, a), (X_2, b), \dots, (X_k, x)\}$, that includes all variables ($k = n$) and does not violate any constraint in C.

Specific classes of constraint satisfaction problems are often designated by the values of their main parameters; this is especially useful for random CSPs where CSPs with specific values can be generated. The typical designation is in the form $\langle n, d, p_1, p_2 \rangle$, where n is the number of variables, d is the (average) domain size, p_1 is the constraint graph density, and p_2 is the (average) tightness of the constraints. By "tightness" is meant the number of possible tuples in the Cartesian product of domain values associated with the constraint that are *not* allowed by the constraint's relation.

CSPs have an important monotonicity property in that inconsistency with respect to even one constraint implies inconsistency with respect to the entire problem. This has given rise to methods for filtering out values that cannot participate in a solution, based on local inconsistencies, i.e. inconsistencies with respect to subsets of constraints. By doing this, these algorithms establish well-defined forms of local consistency in a problem. The most widely studied and most widely used methods establish *arc consistency*.

In problems with binary constraints, *arc consistency* (AC) refers to the property that for every value a in the domain of variable X_i and for every constraint C_{ij} involving X_i there is at least one value b in the domain of X_j such that (a, b) satisfies that constraint. More formally, a problem P with binary constraints is arc consistent if $\forall X_i \in X$, $\forall a \in D_i$, $\forall C_{ij} \in C$ $\exists$ a value $b \in D_j$ such that (a, b) satisfies C_{ij}. A similar definition can be given for a problem with constraints whose arity is greater than two.

Singleton arc consistency, or SAC, is a particular form of AC in which the just-mentioned value a, for example, is considered the sole representative of the domain of X_i. If AC can be established for the problem under this condition, then it may be possible to find a solution containing this value. On the other hand, if AC cannot be established then there can be no such solution, since AC is a necessary condition for there to be a solution, and so a can be discarded. If this condition can be established for all values in problem P, then the problem is singleton arc consistent. More formally, a problem P is singleton arc consistent if $\forall X_i \in X$, $\forall a \in D_i$, the problem $P|_{X_i=a}$ is arc consistent. (Obviously, SAC implies AC, but not vice versa.) In what follows, we will sometimes refer to the variable whose domain is currently a singleton as the "focal variable".

Before introducing the next form of local consistency, we give a definition that will be useful here and elsewhere in the paper.

Definition 1. The *neighbourhood* of a variable X_i is the set $X_N \subseteq X$ of all variables in all constraints whose scope includes X_i, excluding X_i itself. More precisely, if the neighbourhood of X_i is designated $N(X_i)$, then

$$N(X_i) = \bigcup_{j: X_i \in \mathit{vars}(C_j)} \mathit{vars}(C_j) \setminus X_i.$$

Variables belonging to X_N are called the neighbours of X_i.

Neighbourhood inverse consistency, or NIC, is a form of local consistency in which each value in the domain of X_i can be extended to an assignment that includes the domains of all the variables in the neighbourhood of X_i [5]. If this condition can be established for all values in problem P, then the problem is neighbourhood inverse consistent. More formally, a problem P is neighbourhood inverse consistent if $\forall X_i \in X$, $\forall a \in D_i$, $\exists$ an assignment A to $X_N \cup X_i$ which includes the assignment of a to X_i, such that $\forall C_j$ where $\mathit{vars}(C_j) \subseteq X_N$, C_j is satisfied by A.

3. Existing SAC algorithms

Here, we describe the various algorithms that have been devised to establish singleton arc consistency for an entire problem. We also discuss significant implementation details.

3.1. SAC-1 with AC-3

SAC-1 is the original SAC algorithm. It is simple and straightforward in contrast to other SAC algorithms, but it is still relatively efficient in comparison with them [1,2,8].

The basic scheme of our version of SAC-1 followed the original description in [4] exactly: an initial pass of AC, followed by a repeat loop which is carried out until no further values are deleted. In each step of the repeat loop, each value in each domain is tested for singleton arc consistency. If this test fails, then the value under consideration is deleted, and following this deletion, the resulting problem is made arc consistent.

For reference, pseudocode for SAC-1 is shown in Fig. 1.

The present version of SAC-1 uses AC-3 to perform consistency testing. This version of AC-3 uses the classical form of the queue composed of pairs of adjacent variables, each representing a directional test of support for values in one domain by values in the other. Assuming initial AC preprocessing and subsequent AC processing after each deletion, it is sufficient in the latter case to initialize the AC queue to pairs consisting of the variable with the reduced domain and each of its neighbours (relaxing the latter against the former). If a value in a neighbouring domain is deleted, then *all* of its neighbours (sans the variable responsible for the deletion) are added to the queue [9].

A restricted form of SAC-1 was described by [11]. In this incomplete form of SAC, there is only one pass

```
Procedure SAC-1
    OK ← AC($P$)
    Repeat                          /* if OK */
        Changed ← $false$
        Foreach $X_i \in X$
            Foreach $v_j \in dom(X_i)$
                $dom'(X_i) \leftarrow \{v_j\}$
                If AC($P'$) leads to a wipeout
                    $dom(X_i) \leftarrow dom(X_i)/v_j$
                    OK ← AC($P$)
                    Changed ← $true$
    Until Changed = false or not OK
```

Fig. 1. Pseudocode for SAC-1 (after [4]).

through the possible assignments. Here it will be designated SAC-1r. [11] showed that in a number of cases, SAC-1r deletes almost as many values as a full SAC algorithm.

3.2. Advanced SAC algorithms

Since the description of the first SAC algorithm, several other algorithms have been proposed for achieving this form of consistency. The motivation behind these is to limit the large number of repeated comparisons required by SAC-1. This has been done in various ingenious ways. Here, we describe the three most important algorithms: SAC-2, SAC-SDS, and SAC-3.

The SAC-2 algorithm [1] was inspired by the arc consistency algorithm AC-4 [10]. In both cases, the basic idea is to gather and store information about support during an initial processing phase (in the form of counters and "support lists"), in order to avoid performing redundant and irrelevant constraint checks in the course of establishing local consistency throughout the problem. SAC-2 begins by running AC-4. Then, analogous to AC-4, there is an SAC initialization phase followed by a SAC pruning phase.

Recall that during the initial phase of AC-4, any domain values with zero support with respect to some constraint are removed and also added to a special no-support list. Then, in the pruning phase, by decrementing counters and adding support lists to the no-support list whenever a counter reaches zero, other domain values are removed until the no-support list is empty, at which point the problem is arc consistent.

In the SAC-2 SAC initialization phase, SAC support lists are built along with a list of assignments that need to be checked, which corresponds to the no-support list of AC-4. SAC support lists are constructed by placing each value that did *not* fail during the SAC-test (in which that value was made the sole member of its domain) on the SAC support list of every value in the remainder of the problem. If, on the contrary, the SAC-test fails, then an AC4-style pruning phase is carried out; the only difference from AC-4 being that if a value's support goes to zero, in addition to it being put on the AC no-support list, all its SAC-supports are put on the SAC no-support list. Following all this, in the SAC pruning phase, each value on the SAC no-support list is tested for SAC. During this stage, if a value fails in a SAC test, this leads to the same procedures as during SAC-initialization: a bout of AC-4-pruning and addition of all assignments on the support list of this value to the SAC-no-support list.

The SAC-SDS algorithm [2,3] is a modified form of the authors' "optimal" SAC algorithm, SAC-Opt. The modifications are intended to produce a more practical algorithm by trading an increase in worst-case time complexity for a reduction in the high space complexity of the latter algorithm.

The key idea of SAC-SDS (and of SAC-Opt) is to represent each SAC reduction separately, so that there are $n \times d$ problem representations (where n is the number of variables and d is the maximum domain size), each with one domain D_i reduced to a singleton. These are the "subproblems"; in addition there is a "master problem". If a SAC-test in a subproblem fails, then the value is deleted from the master problem and this problem is made arc consistent. If this leads to failure, the problem is inconsistent; otherwise, all values that were deleted in order to make the problem arc consistent are collected in order to update any subproblems that still contain those values. Along with this activity, the main list of assignments (the "pending list") is updated, so that any subproblem with a domain reduction is re-subjected to a SAC-test.

The main difference between SAC-Opt and SAC-SDS is that certain data structures are not duplicated in the latter. For example, when AC-2001 is used for AC processing, then the Last data structure is not duplicated [2,3]. When AC-3 is used, however, there is no difference of this sort. In addition, in the original descriptions, for SAC-Opt all subproblems are created in an initialization step (cf. Algorithm 1 of [2]), while for SAC-SDS subproblems are created 'on the fly', i.e. when the relevant assignment is taken off the pending list (see next paragraph). Since this is really only a difference in efficiency of implementation, the present version of SAC-SDS can be considered an efficient version of SAC-Opt with AC-3. However, since the coding was done on the basis of the description of SAC-SDS in [2,3], it will be referred to under this name.

SAC-SDS and SAC-Opt also make use of queues (here called "copy queues"), one for each subproblem, composed of variables whose domains have been reduced. These are used to restrict SAC-based arc consistency in that subproblem, in that the AC-queue of the subproblem can be initialized to the neighbours of the variables in the copy queue. Copy queues themselves are initialized (at the beginning of the entire procedure) to the variable whose domain is a singleton. In addition, if a SAC-test leads to failure, the subproblem involved can be taken 'off-line' to avoid unnecessary processing. In the present implementation, an $n \times d$ array of flags was used to indicate whether a given subproblem was still active. Subproblems need only be created and processed when the relevant assignment is taken from the pending list; moreover, once a subproblem is 'off-line' it will not appear on the pending list again, so a spurious reinstatement of the problem cannot occur. (It is worth noting that this feature plus ensuring that redundant elements are not added to the pending list are critical for realizing the full efficiency of this algorithm; together they can reduce run time by an order of magnitude on some problems.)

The SAC-3 algorithm [8] uses a greedy strategy to eliminate some of the redundant checking done by SAC-1. The basic idea is to perform a set of SAC tests in a cumulative series, i.e. to perform SAC with a given domain reduced to a single value, and if that succeeds to perform SAC with *an additional* domain reduced to a singleton, and so forth until a SAC-test fails. (This series is called a "branch" in the original paper.) The gain occurs because successive tests are done on problems already reduced during earlier SAC tests in the same series. However, a value can only be deleted during a SAC test if it is an unconditional failure, i.e. if this is the first test in a series. This strategy is carried out within the SAC-1 framework. That is, successive phases in which all of the existing assignments are tested for SAC are repeated until there is no change to the problem.

In order to carry out the SAC-3 procedure efficiently, it is necessary to be able to select assignments for SAC testing within a branch so that variables are not repeated, since for a given assignment the domain is a singleton that must remain constant throughout the series. In the present implementation, this was done by representing the queue as an array of domain lists, each with an associated Boolean variable to indicate whether an assignment of that (CSP) variable is or is not currently on a branch. The branch itself was represented by an array indexed by the variable number, with the value stored in that cell; in addition a special cell was used to track the length of the current branch. As with SAC-1 and SAC-SDS, the present encoding of SAC-3 employs AC-3 as the basic AC algorithm.

4. Neighbourhood SAC algorithms (NSAC)

The main contribution of this paper is the description of a new form of local consistency, which is also a form of singleton arc consistency. It is called neighbourhood singleton arc consistency because it establishes SAC with respect to the neighbourhood of the variable whose domain is a singleton.

Definition 2. A problem P is *neighbourhood singleton arc consistent* with respect to value v in the domain of X_i, if when D_i (the domain of X_i) is restricted to v, the problem $P_N = (X_N \cup X_i, D_N \cup \{v\}, C_N)$ is arc consistent, where X_N is the neighbourhood of X_i, D_N is the set of domains of variables in X_N, and C_N is the set of all constraints whose scope is a subset of $X_N \cup X_i$.

In the last definition, note that C_N includes constraints among variables other than X_i, provided that these do not include variables outside the neighbourhood of X_i.

Definition 3. A problem P is neighbourhood singleton arc consistent (NSAC) if each value in each of its domains is neighbourhood singleton arc consistent.

In this paper, two basic algorithms are described that establish NSAC; these are referred to as NSAC-1 and NSACQ. The first, as its name implies, is simply a version of SAC-1, where consistency maintenance during each "SAC phase" (line 7 of Fig. 1) is restricted to the neighbourhood of the variable with the singleton domain. In other words, the line

If AC$\big(P'\big)$ leads to a wipeout

is replaced by

If AC$\big(X_i + \text{neighbours}(X_i)\big)$ leads to a wipeout

Thus, AC is established in the subgraph formed by a variable and its neighbours after restricting the domain of the former to a single value v. If this attempt to establish AC fails, then value v is removed and AC is reestablished for the entire problem. As with SAC-1, this process continues until there is a pass through all the domain values of P in which no value deletion occurs.

We also consider a variant of NSAC-1 in which AC following a SAC-based deletion is also restricted to the neighbourhood of the variable currently being tested for singleton arc consistency. We will refer to this version of NSAC-1 as NSAC-1ACr.

NSACQ differs from the NSAC-1 algorithms in that it also uses an AC-3 style of processing at the top-level. This means that there is a list (a queue) of variables, whose domains are considered in turn (just as in NSAC-1 and SAC-1); but in this case, if there is a SAC-based deletion of a value from the domain of X_i, then any neighbours of X_i that are not currently on the queue are put back on. Unlike the NSAC-1 algorithms, there is no "AC phase" following a SAC-based value

```
Procedure NSACQ
    Q ← $X$
    OK ← AC($P$)
    While OK and not empty-Q
        Select $X_i$ from Q
        Changed ← *false*
        Foreach $v_j \in dom(X_i)$
            $dom'(X_i) \leftarrow \{v_j\}$
            If AC($X_i$ + neighbours($X_i$)) leads to
            a wipeout
                Changed ← *true*
                $dom(X_i) \leftarrow dom(X_i)/v_j$
                If $dom(X_i) = \emptyset$
                    OK ← *false*
        If OK and Changed = *true*
            Update Q to include all neighbours
            of $X_i$
```

Fig. 2. Pseudocode for NSACQ.

removal (line 8 in Fig. 2). The idea behind this strategy is that if a deletion from the domain of focal variable X_i has any effect, it must affect the neighbours of X_i, and any effects elsewhere in the problem can only occur through effects on these neighbours.

We now establish some basic properties of NSAC and these NSAC algorithms.

Proposition 1. *NSAC-*1 *reaches a unique fixpoint.*

Proof. A problem P is neighbourhood SAC consistent if no value violates the NSAC condition. Since NSAC-1 continues to check every value until all values meet this criterion, it will achieve the same fixpoint regardless of the order in which tests are made. □

Next, we introduce a proposition that will be used in some of the subsequent proofs.

Proposition 2 (Neighbourhood Lemma). *For any algorithm involving AC testing and no other form of consistency maintenance, if removal of value v of D_i (the domain of X_i) has the removal of value z of D_j (the domain of X_j) as a consequence, then values must be removed from successive neighbourhoods beginning with the neighbourhood of X_i and ending with a neighbourhood of which X_j is a member.*

Proof. In AC testing, suppose that removal of v from D_i has no effect on values in any domain D_j of a neigbouring variable, i.e. a variable X_j for which there is a constraint C for which $X_i \in vars(C)$ and $X_j \in vars(C)$. Then it cannot affect values associated with any constraint that does not include X_i, since the domains associated with the subgraph based on $X - X_i$ are unchanged. For the same reason, if values are removed from neighbouring domains following removal of v, then if these removals do not affect any of their neighbouring domains, they cannot affect other parts of the problem. Since this argument can be continued ad infinitum along any path of successive deletions, regardless of cycles, the lemma holds. □

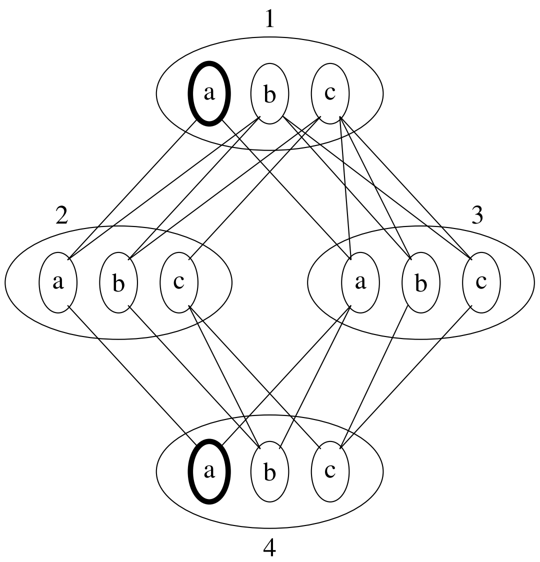

Fig. 3. Example showing that converse of Neighbourhood Lemma does not hold. In this and subsequent figures, lines between values in different domains indicate support, i.e. that the two values form a tuple that satisfies the constraint between their variables. Further details in text.

Note that the converse of the Neighbourhood Lemma does *not* hold, i.e. if value z in D_j depends on value v in D_i, then removal of v will not necessarily result in removal of z. An example is shown in Fig. 3. Here, if value a in the domain of variable 1 is removed, then a in the domain of variable 4 cannot form part of a solution. Yet, if successive neighbourhoods starting with variable 1 are tested for arc consistency, this will not be discovered, since all values in the domains of variables 2 and 3 are still supported, and some of these support a in variable 4.

Proposition 3. *NSAC-*1, *NSAC-*1*ACr, and NSACQ always reach the same fixpoint.*

Proof. We consider two cases: (1) values are only removed during the NSAC phase, (2) values are also removed during an AC phase that follows an NSAC-based removal. Obviously, any differences in the fixpoints reached by NSAC-1 and NSAC-1ACr will depend on ultimate effects of the AC phase of the algorithms, since if values are only removed during the NSAC phase, the behaviour of the two algorithms will be identical. It is also obvious that a difference in results between the two algorithms can only occur if a value is deleted during an AC phase of NSAC-1 but not during the corresponding AC phase of NSAC-1ACr.

But, because of repeated testing of the entire set of variables, even the limited form of AC in NSAC-1ACr will eventually examine the same patterns on non-support as the more extensive AC of NSAC-1, so they will both reach the same fixpoint. More precisely, if a value is removed by NSAC-1 that is not in the neighbourhood of the value removed by SAC testing, then by the Neighbourhood Lemma, this value will also be removed by NSAC-1ACr.

For NSAC-1, if value a in the domain of variable X_i is deleted during the NSAC phase of the algorithm, then the subsequent AC will only remove a neighbouring value if none of the remaining values in the domain of X_i support it. In this case, these values will also be removed by NSACQ when the domains of the neighbours of X_i are tested following the removal of a. This is guaranteed to occur because the algorithm ensures that all the neighbours are on the queue following such deletion. Moreover, this guarantee extends to values removed during the AC phase of NSAC-1 that are not in the neighbourhood of a, given the Neighbourhood Lemma. Hence, NSAC-1 and NSACQ will always reach the same fixpoint. □

The formula for worst-case complexity for NSAC-1 resembles the one for the classical SAC-1 algorithm [4]. In the present case the formula is O($e_{\text{sub}}n^2d^5$), because an ordinary AC-3 algorithm is used; in this case, the term for number of constraints in the entire problem (e) is replaced by a similar term for the number of constraints in the largest neighbourhood (e_{sub}). NSACQ does not appear to affect this formula, since the worst-case is bounded by the complexity of AC (O($e_{\text{sub}}d^3$)) times the number of assignments ($n \times d$), and the full series of SAC-tests can occur up to $n \times d$ times. However, it is to be expected (and it has been verified) that the average-case complexity is often better than that of NSAC-1.

5. Relations between NSAC and related algorithms

Here, we describe some important dominance relations between NSAC and other forms of local consistency.

Proposition 4. *SAC dominates NSAC. That is, if a value is removed by NSAC-k, then it will also be removed by SAC, but the converse does not hold.*

Proof. Clearly, any value removed during singleton AC by NSAC will also be removed by SAC, since the latter includes all the checking done by NSAC. A value deleted by SAC and not by NSAC can only occur outside the neighbourhood of focal variable X_i, either by domain wipeout during the SAC phase of the former or by domain reduction during an AC phase. In the latter case (AC phase reduction), deletion of values in neighbouring domains will also occur with NSAC because these values will be tested in subsequent singleton phases.

It is certainly possible for reductions in neighbouring domains to lead to a wipeout elsewhere in the network, so that full SAC can deduce that a value cannot participate in a solution while NSAC applied to the same variable and value for a network in the same state will not. In contrast to cases where values are deleted by simple AC (where the Neighbourhood Lemma ensures that they will also be deleted by NSAC), such failures will not necessarily be discovered during subsequent passes of the NSAC algorithm, because during these passes, arc consistency is not carried out under the conditions (reduction of the domain of X_i to a singleton) that led to this SAC-based wipeout. That such conditions actually occur can be shown by example (see Fig. 4). □

Since the proof of the next proposition involves the notion of a minimal unsatisfiable core (MUC), a definition is given.

Definition 4. Let $P = (X, D, C)$ be a CSP that is unsatisfiable. $P' = (X', D', C')$ is an unsatisfiable core of P, if $X' \subseteq X$, $D' \subseteq D$, $C' \subseteq C$ and P' is unsatisfiable. P' is a *minimal unsatisfiable core* if there is no unsatisfiable core of P' that is not identical to P'.

Proposition 5. *NIC dominates NSAC.*

Proof. If a value in the domain of variable X_i is removed by NSAC, then this is because it led to a wipeout in a neighbouring domain. It cannot, therefore, be part of any consistent assignment to that variable and the X_j variables in the neighbourhood of variable X_i, and NIC will, therefore, also delete this value.

To see that NIC can delete values that NSAC cannot, consider a neighbourhood singleton arc consistent

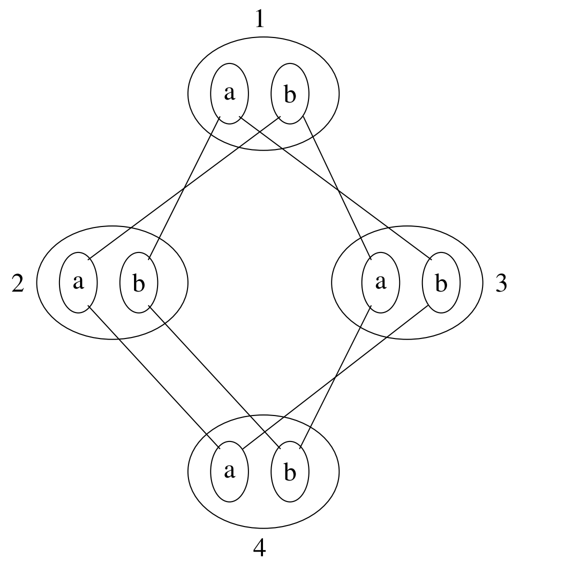

Fig. 4. Problem that is neighbourhood-SAC consistent but not SAC–consistent. Testing for SAC will show that neither value belonging to variable 3 is singleton arc consistent, while testing for NSAC will not delete either value.

network that is also a MUC. That this is possible can be shown by example (see Fig. 4).

Now assume that there is an additional variable whose neighbourhood consists of this MUC, but whose values at this stage of processing are compatible with any value in any domain of the variables in the MUC. This enlarged network is, therefore, still neighbourhood singleton arc consistent, but it is not neighbourhood inverse consistent. □

Proposition 6. *NSAC dominates AC.*

Proof. That neighbourhood SAC can remove values not removed by AC can be shown by example (see Fig. 5); hence, AC does not dominate NSAC. That any value deleted by AC is also deleted by NSAC can be shown by an inductive argument which considers dependency chains of support of length r. Thus, for $r = 0$, any value deleted from the original problem by AC on the first pass, i.e. values whose removal does not depend on the removal of other values, will also be removed by NSAC, since it also tests every domain D_i (of variable X_i) against the domain of every other variable in problem P that shares a constraint with X_i. Supposing the condition is true for any chain of length k, then because any value for which $r = k + 1$ will be in the neighbourhood of the domain containing a value with a chain of length k, this value will also be tested following removal of the first value. □

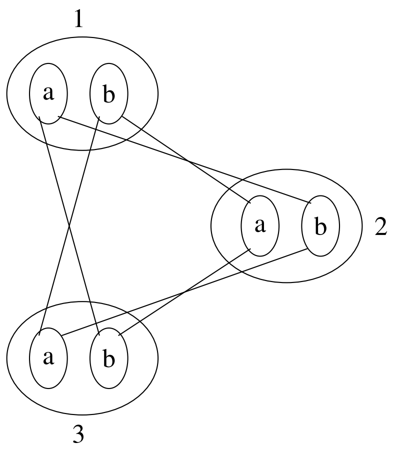

Fig. 5. Problem that is arc-consistent but not neighbourhood-SAC consistent.

It should be noted that, given Proposition 6, the initial AC in the pseudocode listed in Fig. 2 is not necessary. (Nor is it required if used with the SAC and NIC algorithms.) However, since AC is so easy to compute, and since it is sometimes possible to prove unsatisfiability in this fashion, it seems reasonable to follow this practice when establishing any of the more stringent forms of local consistency.

6. A new SAC algorithm

6.1. Basic algorithm

The strategy used by the NSACQ algorithm can also be used to produce a full SAC algorithm. This algorithm uses a queue like NSACQ, but instead of updating it with the neighbours of a variable whose domain was subject to reduction, all variables not on the queue are returned to it. Pseudocode is shown in Fig. 6. This algorithm is, therefore, similar to SAC-1 in that it resets itself after domain reductions, but it does not go through the entire set of variables and domain values before doing so. More importantly, resetting in this case means putting a subset of the variables at the end of the queue. Note that, in addition, this algorithm does *not* perform AC after a SAC-based domain reduction.

Proposition 7. *SACQ reaches the same fixpoint as SAC-*1.

Proof. If a value can be deleted without other values being deleted first, this will be discovered by SACQ as well as by SAC-1. If the deletion of a value depends

```
Procedure SACQ
    Q ← $X$
    OK ← AC($P$)
    While OK and not empty-Q
        Select $X_i$ from Q
        Changed ← *false*
        Foreach $v_j \in dom(X_i)$
            $dom'(X_i) \leftarrow \{v_j\}$
            If AC($P'$) leads to a wipeout
                Changed ← *true*
                $dom(X_i) \leftarrow dom(X_i)/v_j$
                If $dom(X_i) = \emptyset$
                    OK ← *false*
        If Changed = *true*
            Update Q to include all $X_j \in X$
```

Fig. 6. Pseudocode for SACQ.

on a previous deletion, there are the usual two cases to consider:

- The deletion depends only on previous SAC passes.
- The deletion depends on deletions that occurred during an AC phase that followed a SAC phase.

In the first case, it is obvious that SACQ will also make the deletion, since all variables are put back on the queue after each deletion. Now, any deletions during an AC phase must involve the neighbourhood of the focal variable, although they may include other domains as well. This can only occur if the neighbourhood value(s) deleted were not supported by any remaining value in the domain of the focal variable. But in this case, the neighbourhood values will also be deleted when another value of the focal variable is tested or when the neighbouring variable is made focal and these values are tested as singletons, either of which will happen in a subsequent SAC phase. But if that value is deleted, then, by the Neighbourhood Lemma, the same argument will apply to values in domains adjacent to that domain, and so forth. □

Clearly, the worst-case complexity of SACQ is no different from that of SAC-1. Although SACQ avoids some redundant testing, it also uses the SAC strategy to delete any values that SAC-1 deletes in its AC phase. As a result, it is not clear whether there will be a reduction in average-time complexity with SACQ.

6.2. *Heuristics and approximations*

One reason for considering a queue-based SAC algorithm is that this form of SAC may be more amenable to enhancements by ordering heuristics than the simple repetitive form of SAC represented by SAC-1 without resorting to elaborate data structures as with SAC-2, SAC-SDS or SAC-3. To this end SACQ was also tested with a queue ordered by the degree of the variable (descending degree order). The idea is that if a variable has a high degree, single values of its domain may be more likely to lead to inconsistencies, and this may be discoverable sooner during arc consistency testing.

In addition, a variant of SACQ, called SACQ-adj, was tested in which, instead of returning all variables to the queue when a domain is decremented, only the neighbours of the variable whose domain was affected are added back on the queue. The idea behind this is that if a domain is decremented, then inconsistencies will have to be propagated through that variable's neighbours to the rest of the problem. Note that AC with a singleton value is carried out as before, to the entire problem. In preliminary testing, it was found that while this procedure occasionally missed a value discovered by the full SAC algorithms, this was a surprisingly rare occurrence. Characterizing these cases remains an open problem.

The pseudocode for this restricted SAC procedure is identical to that shown in Fig. 6, except for the last line where

Update Q to include all $X_j \in X$

is replaced with

Update Q to include all $N(X_i)$.

Proposition 8. *SACQ dominates SACQ-adj.*

Proof. Clearly, any value removed by SACQ-adj will also be removed by SACQ, since the latter performs all the tests carried out by the former. When a value a of variable X_i is deleted by a full SAC algorithm, it is possible to have a value b in the domain of a variable outside the neighbourhood of X_i such that without a there would be no solution containing b. However, this dependency will not necessarily be found if only the neighbours of X_i are put back on the queue. (Recall that the converse of the Neighbourhood Lemma does not hold.) That such dependencies exist can be shown by example. Thus, in the example shown in Fig. 3, if a is removed from the domain of variable 1 while variable 4 is not in the queue, then a SAC algorithm will add that variable back to the queue and will discover that a in the domain of variable 4 is not singleton arc consistent. SACQ-adj will not add this variable, and if nothing else causes it to be added, this procedure will not discover this inconsistency. □

7. Experimental results – I

In the initial experimental tests, SAC and neighbourhood SAC algorithms were compared in the standard fashion, using homogeneous random problems of varying tightness (cf. [2,4,8]). Problems were sparse; since this test was mainly for comparison with earlier work, it was not felt necessary to test very dense problems as well, as in [2]. The initial experiments were followed by a smaller experiment with a single set of random problems with heterogeneous features. This demonstrated some interesting contrasts in the patterns of differences among algorithms when compared with homogeneous problems.

7.1. Notes on methods

Problems in the initial experiment had the parameters used by [8]: $\langle 100, 20, 0.05, t\rangle$, where t varied from 0.10 to 0.90 in steps of 0.05. As in that study, 50 problems were generated for each value of t. In the present case, however, problems were always generated with an initial spanning tree, in order to ensure that all problems were connected, and therefore truly of size 100. Following tests with these problems, a similar series of tests were carried out with problems having the same parameter values, including the same number of constraints, but without ensuring that they formed a single connected component. This was done to test the full SAC algorithms on problems with features that were identical to those reported in the literature.

The experiment on problems with more heterogeneous features was carried out using a single set of 100 problems. These were geometric problems [7] whose values had varying levels of support in the domains of variables whose nodes were adjacent in the constraint graph. They were, therefore, randomly generated problems without the extreme homogeneity of those that conform to standard models.

Geometric problems have constraint graphs which are clumpy. These problems are generated by selecting points at random within the unit square to represent variables, and then constraining pairs of variables if the Euclidean distance between their points is less than some criterion, called the “distance”. In the method used in the present work, if there was more than one connected component, separate components were connected via the closest variables to make a single connected graph. (That is, of all variable pairs (X_i, X_j) where X_i belongs to one component and X_j to the other, the pair with the smallest distance between such pairs is determined and a constraint is added between this pair.) In the present case, the value for the distance was 0.17. In addition, a target and range was used to ensure that only problems with 540 constraints (the target) $\pm$ 3 (the range) were included in the sample; this gives a density of about 0.076. Problems had 120 variables and the domain size was 20. The sample size was 100.

Variation in probability of support was established by a two-tier process: given a set of probability values, each representing a possible likelihood of support, select one with a certain probability, and then select supporting values in the adjacent domain according to the probability selected. In the present procedure, selection of a probability of support was done independently for each value and for each constraint. For these problems, two support probabilities were used: 0.3 and 0.7 (so corresponding tightnesses were 0.7 and 0.3, respectively). The first was chosen with a probability of 0.2, the second with a probability of 0.8. (Note that while probabilities in the first set are independent, those in the second set must sum to 1.)

In these experiments all of the new algorithms described above were tested along with previously described algorithms: SAC-1, restricted SAC-1 (SAC-1r), SAC-2, SAC-SDS, and SAC-3.

Implementations were written in Common Lisp, and experiments were run in the Xlispstat environment with a Unix OS on a Dell Poweredge 4600 machine (1.8 GHz).

In coding these algorithms, care was taken to develop efficient data structures and to avoid redundant processing. Correctness of implementation was tested by cross-checking the number of values deleted for *all* algorithms on *all* problems tested when a problem was not proven unsatisfiable by the preprocessing algorithm. (In the latter case, differences in the sequence of tests can result in different numbers of values deleted before a wipeout occurs.) Although in a few cases the domains themselves were examined to verify that exactly the same values had been deleted, this was not considered necessary given the large number of comparisons made of numbers of values deleted.

In each experiment all algorithms were run sequentially, one immediately after the other, to minimize vagaries of timing that occur over time (e.g. after several days), and are not under the control of the experimenter. (These are presumably due to changes in system configuration including resource allocation.) This meant that if an algorithm like SAC-1 was included in more than one experiment, it was run once in each experiment even if the problems were the same. In addition, in a given experiment algorithms were all run with the same process priority.

In some experiments a full search was carried out after preprocessing. Search was always done with an implementation of MAC-3 with d-way branching, using the minimum domain-over-forward-degree heuristic for variable ordering. For tests with heterogeneous random problems, search was cut off at one million search nodes.

In this and the next section, some differences between means were evaluated statistically using the Wilcoxon Signed Rank Test [6]. Given the marked differences in the patterns of results across experiments (e.g. in the problem sets tested in this section there were no consistent trends across problems, while in some cases in the next section (e.g. rlfapgraph and langford problems) there was a systematic increase in difficulty across problems), it seemed best to use a nonparametric test statistic like the Wilcoxon T, which only involves ordinal assumptions about the underlying distribution. This test requires paired observations (e.g. results for SAC-1 vs. SACQ on the same problem set) and is based on a ranking of differences between the paired scores, where each rank is given a sign corresponding to the sign of the difference. Positive and negative ranks are summed, and the smaller sum gives the Wilcoxon T statistic. Given the size of the samples ($N = 50$) in the experiments in this section a significance criterion of 0.01 was used. In order to avoid doing too many tests, order-of-magnitude differences were not tested, although given the vanishingly small values for p (the probability of no actual differences between algorithms) in most cases, the possibility of spuriously significant results is not a major issue in these experiments.

7.2. *Results of main experiments*

Comparisons for SAC algorithms are presented in two graphs. The first compares the new SAC algorithms with SAC-1 and SAC-1r. The second compares the more elaborate algorithms proposed in recent years with SAC-1.

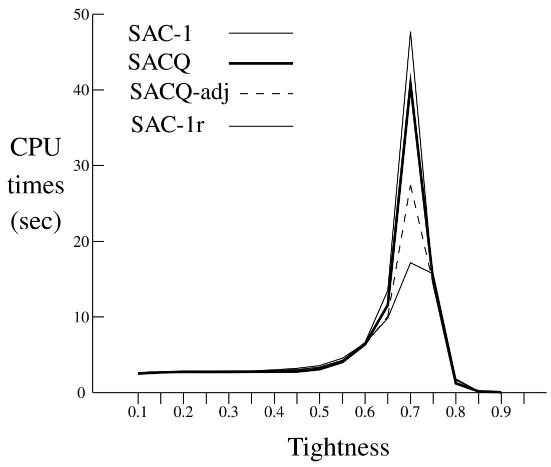

Fig. 7. CPU times for new SAC algorithms compared with SAC-1 and SAC-1r. Random problems (with connected constraint graphs) of varying tightness.

As shown in Fig. 7, SACQ was somewhat more efficient than SAC-1 on the most difficult problems, while SACQ with neighbours only added back to the variable queue was appreciably more efficient. As expected, in the region of greatest difficulty, SAC-1r was the fastest procedure.

In this and the next experiment, statistical comparisons were made for differences at peak tightness (0.7). In this experiment, all differences were statistically significant (T usually equalled 0 and was at most 14; $p \lll 0.01$).

The degree ordering heuristic described in Section 6 only had a noticeable effect for problems of tightness 0.75 and 0.80. For 0.7 there was a small improvement, which, surprisingly, was larger for SAC-1 than for SACQ in either of its forms. Since the results were so similar to the original results they are not shown here.

Figure 8 shows comparisons among the more advanced algorithms and SAC-1. In all cases the peak times for the advanced algorithms are appreciably greater than for SAC-1. SAC-2 is the slowest (the mean value for tightness $= 0.7$ [not shown in the graph] was 412 sec), SAC-3 is much faster, and SAC-SDS is faster still; however, the latter is still slower than SAC-1 for the most difficult problem classes.

In this experiment all comparisons with SAC-1 were statistically significant (T $\leqslant 6$; $p \lll 0.01$). In addition, a comparison of SAC-SDS and SAC-3 showed that the former was significantly faster, despite the fact that the two greatest differences were in favour of the latter algorithm (T $= 99$; $p \ll 0.01$).

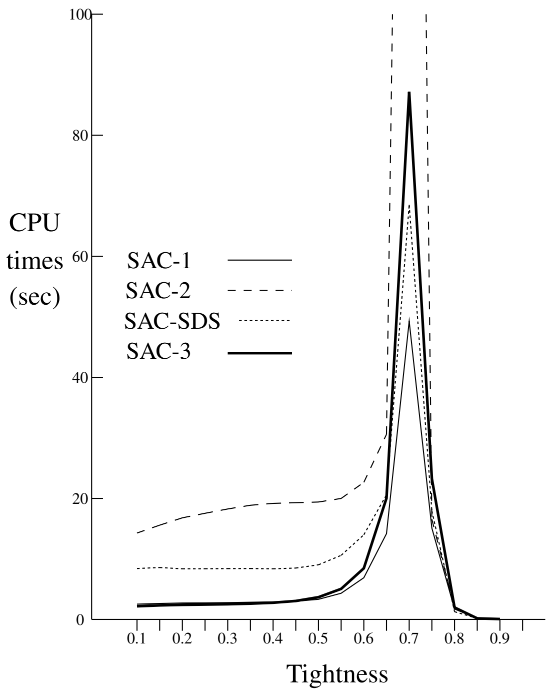

Fig. 8. CPU times for advanced SAC algorithms together with SAC-1. Same problems as in Fig. 7.

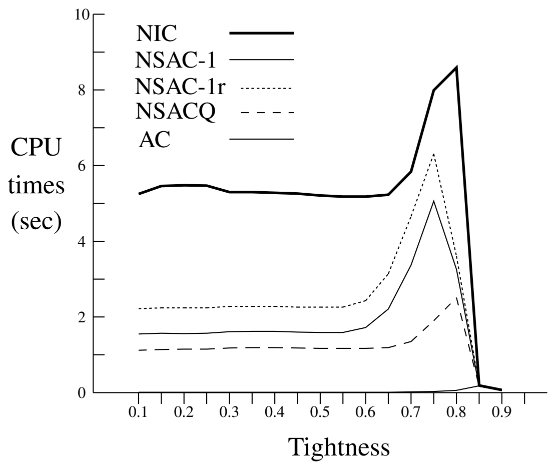

Fig. 9. CPU times for NIC, neighbourhood SAC (NSAC), and AC only. Same problems as in Fig. 7.

Table 1

Number of problems proved unsatisfiable with different forms of preprocessing

Algorithm type	Tightness					
	0.65	0.70	0.75	0.80	0.85	0.90
SAC	0	2	50	50	–	–
SAC-1r	0	0	50	50	–	–
SAC-adj	0	2	50	50	–	–
NIC	0	0	1	43	–	–
NSAC	0	0	1	43	–	–
AC	0	0	0	0	50	50

Notes: Maximum value in each case is 50. For problems with tightness < 0.65, no problems were shown to be unsatisfiable during preprocessing. In Tables 1–5 "algorithm type" refers to algorithms that produce a distinct type of consistency.

A similar pattern of results was found for the second set of problems whose constraint graphs were not fully connected. Curves for the full SAC algorithms were very similar to the curves in Figs 7 and 8. For example, for the hardest problems (tightness = 0.7), the mean run times for SAC-1, SACQ, SAC-2, SAC-SDS and SAC-3 were 53.6, 46.9, 551.9, 87.7 and 98.7 sec, respectively; this is the same ordering as with the connected problems with almost the same relative values.

In the tests of neighbourhood SAC algorithms, both NSAC algorithms were significantly faster than NIC, while the NSACQ version was distinctly faster than NSAC-1, the version based on SAC-1 (see Fig. 9). Interestingly, NSAC-1ACr was slower than NSAC-1.

In this case, since the peaks for different algorithms occurred at different tightnesses, statistical tests were made for three tightness values. For tightness 0.7 and 0.75 all differences were statistically significant (T = 0 for all three tests). For tightness 0.8, both comparisons with NSACQ were statistically significant ($T \leqslant 167.5$; $p < 0.01$).

Interpretation of these results is facilitated by considering the number of values deleted by each algorithm and especially by the number of problems of higher tightness that were proven unsatisfiable by preprocessing, shown in Table 1. There are several points of interest in this table. For the two highest tightness values, the initial AC pass is sufficient to prove unsatisfiability. Among other things, this ensures that CPU times will be almost identical for all algorithms in these cases. Interestingly, at the next lowest tightness value (0.8), none of the problems can be proven unsatisfiable with this method, even though all are in fact unsatisfiable. NIC and NSAC are almost as effective for proving unsatisfiability as full SAC when the tightness is 0.8; but at 0.75, the latter is still highly effective while the former are not. In addition, in all cases NSAC is just as effective as NIC for proving problems unsatisfiable. (In all cases where the number proven unsatisfiable is less than 50, the *same* problems are discovered by NIC and NSAC.) SAC-adj is more effective than SAC-1r in proving problems unsatisfiable with problems of intermediate tightness (here 0.7).

The number of values deleted by each algorithm is shown in Table 2. Note that the figures for SAC, NIC and NSAC are for deletions that are in addition to those deleted by the preliminary AC. Also, the frequency counts and means are for problems *not* proven unsatisfiable. For unsatisfiable problems, SAC-1 deletes many more values before proving unsatisfiability, but this does not necessarily translate into greater time, since SACQ performs more singleton AC steps.

These results show that SACQ with queue additions restricted to neighbours (SACQ-adj) is almost as effective as the full SAC algorithms in detecting values that cannot occur in a solution. The only differences were found for tightness = 0.70; here there were eight problems (out of 48) where 1–3 more values were deleted by full SAC. As with problems proven to be unsatisfiable, this algorithm is more effective than the one-pass SAC-1 procedure, because it is better able to detect singleton inconsistencies that depend on other singleton inconsistencies.

These results also show that a clear dominance of SAC over NIC and NSAC occurs primarily in regions of the problem space where problems are difficult. For example, for tightness = 0.75, AC + NIC or AC + NSAC deleted only 80 values (out of 2000) on average, while SAC proved all of these problem unsatisfiable. For problems with looser constraints, the few values detected by SAC are also detected by more limited assessment of singleton values.

In addition, the results show that NIC and NSAC are almost identical in their effects with these problems. In fact, there were only 2 problems (in the 0.7 and 0.75 tightness groups) where one or two inconsistent values were detected by NIC but not by NSAC.

Since preprocessing is done in order to speed up subsequent search (if the latter cannot be avoided altogether), it was of interest to examine the effects of different forms of AC filtering on subsequent search effort. The results of this analysis are shown in Table 3.

There are several important features of these results. Search efficiency after SACQ-adj showed little or no decrement in comparison with full SAC. The only difference was found for tightness 0.70 where there was a difference of a few nodes for a few problems. This

Table 2

Number of values deleted with different forms of preprocessing

Algorithm type	Tightness									
	0.60		0.65		0.70		0.75		0.80	
	prb	del	prb	del	prb	del	prb	del	prb	del
SAC	4	1.3	22	1.4	48	21.0	–	–	–	–
SAC-1r	4	1.3	22	1.4	48	17.0	–	–	–	–
SACQ-adj	4	1.3	22	1.4	48	20.8	–	–	–	–
NIC	4	1.2	20	1.3	50	6.1	49	49.4	7	294.4
NSAC	4	1.2	20	1.3	50	6.1	49	49.3	7	294.4
AC	14	1.1	34	2.0	50	6.5	50	30.6	50	143.8

Notes: Means (and numbers of problems) based on problems not proven unsatisfiable for which there were deletions. For SAC, NIC and NSAC, entries for deletions do *not* include initial AC.

Table 3

Search effort after different forms of preprocessing

Algorithm type	Tightness													
	0.50		0.55		0.60		0.65		0.70		0.75		0.80	
	prb	srch	prb	srch	prb	srch	prb	srch	prb	srch	prb	srch	prb	srch
SAC	50	100	50	100.3	50	100.5	50	9948.1	48	2238.2	–	–	–	–
SAC-1r	50	100	50	100.3	50	105.5	50	9948.1	50	2147.8	–	–	–	–
SACQ-adj	50	100	50	100.3	50	100.5	50	9948.1	48	2238.5	–	–	–	–
NIC	50	100	50	100.3	50	100.5	50	9948.1	50	2554	49	33.6	7	8.9
NSAC	50	100	50	100.3	50	100.5	50	9948.1	50	2554	49	33.6	7	8.9
Satisfiable	50		50		50		50		0		0		0	

Notes: Means (and numbers of problems) based on problems not proven unsatisfiable by preprocessing. Means are search nodes. Number of problems that were satisfiable is shown on last line.

was also true for SAC-1r, although search was required for the two problems proven unsatisfiable by the other SAC methods. Using NIC or NSAC only led to noticeable decrements for tightness 0.70. (Recall however, that they were much less effective than SAC algorithms for proving unsatisfiability when problems had slightly higher tightness (0.75 and 0.80).) Search after NSAC was always as efficient as after NIC. (Recall that the capacity to prove unsatisfiability was also identical on these problems.)

For these problems, the usual MAC search gave results that were identical or only slightly worse than after preprocessing with SAC, NIC, or NSAC. Differences were restricted to tightness of 0.70 or greater, the major difference occurring at tightness 0.70.

7.3. *Results for heterogeneous problems*

The results of the experiments on geometric problems with varying degrees of support are shown in Tables 4–5. (Results with more advanced SAC algorithms were similar to those found with homogeneous random problems (all were slower overall than SAC-1, which in turn was slower than SACQ), so they are not reported here. In addition, NSACQ was again found to be faster than the other NSAC algorithms (by a factor of two), so only times for this algorithm are reported here.) For purposes of analysis, the problem set was divided into four groups for each algorithm:

(1) not satisfiable, proved during preprocessing,
(2) not satisfiable, proved during subsequent search,
(3) satisfiable,
(4) search reached 10^6 node cutoff before finishing.

For the first category, only frequencies (i.e. number of problems falling in that category for each algorithm) and times are given, because the number of values deleted is not a deterministic quantity (and, of course, there is no search tree). Categories 2 and 3 include these statistics along with the mean number of values deleted from a problem by the preprocessing algorithm and the mean size of the search tree. For the last category, only the number of problems is given, since the same cutoff was used throughout. Note that the times for unsatisfiable problems with search pertain to only one problem for the SAC and NIC algorithms. Also, number of values removed pertains to removals *after* an initial pass of AC, which removed 39 values on average.

Several points are of interest. Firstly, for these problems, SACQ-adj is as effective as the full SAC algorithms (SAC-1 and SACQ), while being faster overall. In contrast, SAC-1r proves fewer problems unsatisfiable, as well as removing fewer values. This has definite effects on the subsequent search, which is more extensive both for unsatisfiable and satisfiable problems in comparison with full SAC. Secondly, for these problems, SACQ is appreciably faster than SAC-1. Thirdly, for these problems, NIC removes appreciably more

Table 4

Algorithm performance on heterogeneous problems I. Frequencies, mean removals and search nodes

Algorithm type	nsol-pre	nosol			Solution			Cutoff
	#	#	del	nodes	#	del	nodes	#
SAC	43	1	168	378	55	224	528	1
SAC-1r	16	27	187	670	55	154	1507	2
SACQ-adj	43	1	168	378	55	224	528	1
NIC	42	1	259	71	56	275	299	1
NSAC	35	8	211	819	55	158	1456	2

Notes: # is number of problems in category, other values are category means. "del" is number of values removed, "nodes" is search nodes.

Table 5

Algorithm performance on heterogeneous problems II. Search times

Algorithm type	no-sol-preproc	No-solution	Solution
SAC1	71	133	152
SACQ	47	55	97
SAC-adj	47	69	75
NIC	10	41	42
NSAC	7	17	16

Note: Mean times (sec).

values than does NSAC, and this has a beneficial effect on search. NIC also proves more problems unsatisfiable. Nonetheless, NSAC(Q) in combination with MAC is faster overall than NIC. In addition, for these problems, NIC is more effective than SAC with respect to values removed and somewhat more effective with respect to the reduction in search nodes.

Due to differences in sample size and problems included in different conditions for different algorithms, it was not possible to evaluate all these differences statistically. However, it could be shown that the difference between SACQ and SAC-1 is statistically significant both for cases where unsatisfiability was proven by SAC and for problems with solutions (T $= 0$ ($N = 43$) and T $= 3$ ($N = 55$), respectively; $p \lll 0.01$ for both comparisons).

8. Experimental results – II

In this section performance of the different SAC and NSAC algorithms is compared for various structured problems. This includes four types of benchmark problems downloaded from the website of C. Lecoutre.[1] In addition, tests were made with "random relop" problems. With problems of this type, density can be varied in order to produce a graded series of problems analogous to the graded series of random problems, and it is possible to produce samples of equal size at each separate point in the range.

In these experiments, due to time constraints, only SAC and NSAC algorithms were tested. This is in line with the main focus of this part of the work, which was to determine the generality of differences found for SAC and NSAC algorithms with random problems.

8.1. Notes on methods

The problems used in the first set of experiments are shown in Table 6, along with the number of instances and, if known, whether they are satisfiable or unsatisfiable. Rlfap-sub problems are the "subinstance" class at the benchmark website. Rlfap-graph are the four mid-size (200–400 variable) problems in the graphs category that are satisfiable. (There are also three unsatisfiable problems in this size range, but since these can be proven unsatisfiable by arc consistency, they were not useful for the present work.) The langford-2 and langford-3 problems and the blackhole-4-4, -4-7, and -4-13 problems are the sets available at the benchmark website. Both sets of langford problems contain 24 instances, but for each set the largest problems could not be used because they exceeded the memory bound of the lisp system during processing by SAC-2. All of the blackhole problems were used that were available. Jobshop problems were the e0ddr1 series originally created by N. Sadeh [13]. (Since problems from the other series in this group gave similar results, one series of this type was deemed sufficient.) Information on these problems, including subcategories used and numbers of problems is given in Table 6. Some of these (rlfap and jobshop) have been used in earlier work on SAC [8].

Table 6

Classes of structured problem used in this study

Problem class	No. of instances	sat/unsat
rlfap-sub	9	unsat
rlfap-graph	4	sat
langford-2	23	both?
langford-3	21	both?
blackhole-4-4	10	unsat
blackhole-4-7	20	?
blackhole-4-13	7	?
jobshop e0ddr1	10	sat
relop ($\geqslant, \neq$)	50	sat

Rlfap problems have binary distance constraints, where the difference between the values of two variables must either be greater than some value k (where k is specific to that constraint) or (in a small number of cases) equal to k. Langford problems include an n-ary all-different constraint; this is represented by a set of binary inequality constraints. In addition there are $n/2$ distance constraints based on the equality relation, where k takes on values between 2 and approximately $n/2$. Blackhole problems have non-random extensional constraints. The jobshop problems have two kinds of constraints. In both cases the form of the constraint is $V1 + k \leqslant V2$; most are disjunctive ($V1 + k_1 \leqslant V2 \vee V2 + k_2 \leqslant V1$), but about a third are non-disjunctive.

Successive problems in the rlfapgraph and langford series are progressively more difficult. For other benchmark classes, problems are more homogeneous with respect to difficulty, and there are no discernible trends for successive problems.

In these tests, random relop problems were 100-variable binary CSPs with $\geqslant$ and $\neq$ constraints in equal proportions. Not-equals constraints ensured that this class of problems is intractable in the worst case. These

[1] http://www.cril.univ-artois.fr/lecoutre/benchmarks.html.

problems had domain sizes of 20 and a graph density of 0.26 (1291 constraints); in all cases the constraint graph was connected.

In addition to these problems, a relop series was tested in an experimental design corresponding to the initial experiment with random problems described in the previous section. These problems had 60 variables and domain sizes of 20. Half the constraints were $>$ and half were $\neq$ constraints. A graded series was produced by varying the graph density from 0.10 to 0.70 in steps of 0.05. As in the earlier experiment with random problems, the number of problems generated for each density value was 50.

8.2. *Results*

Tables 7 and 8 give (arithmetic) mean runtimes for the SAC and NSAC algorithms, respectively, for the different classes of problems shown in Table 6. (Note that for these jobshop problems, SAC does not delete any values.)

For each problem class, statistical comparisons were made between (1) the SAC algorithm with the best average time and the one with the worst, excluding SAC-2, (2) the former algorithm and the one with the second-best mean. If the first comparison was statistically significant, the mean for the superior algorithm is shown in bold in the tables; if the second was, the mean is underlined. (Although these conventions could not be used for the rlfapgraph problems because of the small sample size (even T $=$ 0 is not statistically significant), differences between SACQ and the other two algorithms were large and consistent, while differences between the latter was not.) In addition comparisons were made between the two approximation procedures. Here, statistically significant differences are indicated by boldface. Other statistical tests (not shown in the tables) were made between SAC-1 and SACQ, in order to evaluate the latter as an alternative to the standard SAC algorithm.

Table 7 shows that in most cases SAC-3 gives the fastest run-times and that the in these cases dif-

Table 7

Runtimes for different SAC procedures for different classes of structured problems

Problems	SAC-1	SACQ	SAC-2	SAC-SDS	SAC-3	SAC-1r	SACQ-adj
rlfap-sub	1.11	**0.54**	19.4	0.77	1.15	1.14	**0.54**
rlfap-graph	5985	4116	–	–	6110	1555	3107
langford-2	507	502	3111	3377	**154**	498	**484**
langford-3	581	562	6265	1263	**218**	**573**	788
blackhole-4-4	0.56	0.56	3.33	0.58	**0.53**	0.56	**0.55**
blackhole-4-7	44.6	44.5	774	43.4	**33.9**	**44.3**	44.7
blackhole-4-13	1170	1178	44,849	1957	**893**	1174	1188
relop-geneq	727	707	38,224	**260**	322	**116**	364
jobshop	184	255	6992	178	**129**	**182**	190

Notes: Values are mean times in sec. Runs on rlfap-graph problem #4 could not be completed for SAC-2 and SAC-SDS. Here and in Table 8 times in boldface indicate that algorithm was significantly better than other (N)SAC algorithms. See text for further details.

Table 8

Runtimes for different NSAC algorithms for different classes of structured problems

Problems	NSAC-1	NSAC-1ACr	NSACQ	AC
rlfap-sub	1.10	**0.47**	0.49	0.16
rlfap-graph	771	679	**414**	1.00
langford-2	410	**386**	416	0.11
langford-3	543	558	554	0.25
blackhole-4-4	0.46	0.49	0.42	0.14
blackhole-4-7	27.2	29.9	26.3	1.07
blackhole-4-13	803	899	**795**	15.2
relop-geneq	60.8	47.2	**41.4**	0.09
jobshop	62.2	53.9	56.9	0.15

Notes: Values are mean times in sec. Times for arc consistency are included for fuller comparison and evaluation of Table 9 results.

ferences with the second- and fourth-best algorithms are always statistically significant. SAC-SDS gave the fastest times for the relop problems, while SACQ was fastest for the rlfap problems. SAC-2 was always markedly inferior, giving times that were sometimes an order of magnitude greater than those for the other algorithms.

SAC-1 and SACQ gave roughly comparable results overall. SACQ was decisively better on the rlfap problems, while SAC-1 was definitely superior on the jobshop problems. The differences in means for the langford-3 and relop problems, which favoured SACQ, were also statistically significant, while the difference for the blackhole-4-13, which favoured SAC-1, was statistically significant.

The results with the largest problems (rlfap-graph) suggest that SAC-SDS does not scale well in practice for number of variables. There is some evidence from the same problems for similar effects with SAC-3, but to a much smaller degree.

For rlfap-graph and the relop problems, SAC-1r was appreciably faster than SACQ-adj. In other cases the two were similar, although there were some differences that were consistent enough to give statistical significance.

Table 8 uses the same conventions used in Table 7, in this case to evaluate the best average time against the second- and third-best times. NSACQ was definitely faster than NSAC-1 in some cases (see Table 8), and was never less efficient. In some cases it was also significantly faster than NSAC-1ACr. In contrast to the random problems, NSAC-1ACr was sometimes faster than NSAC-1, but except for the rlfap-sub problems, the proportional differences were small.

Table 9 shows mean values removed with different forms of consistency and consistency approximation. With some problem classes (langford and blackhole) there is little or no improvement over AC with any of the stronger forms of consistency. (This is also true for the jobshop problems, where no algorithm deleted any values.) However, for langford-3 problems, AC only proved the first problem unsatisfiable, while all of the SAC, NSAC and restricted SAC algorithms proved the first four problems unsatisfiable. In addition, SACQ-adj required fewer value deletions to do this; the mean was 78.3 versus 104 for the other procedures.

For other problem classes, however, the difference is marked. In these cases, therefore, there is an important tradeoff between time and amount of filtering. We also see that SACQ-adj does as well as full SAC. (The one difference in the table is for unsatisfiable problems; in this case SACQ-adj was able to prove unsatisfiability after markedly fewer deletions than the other SAC procedures.) NSAC generally deletes fewer values than SAC, although appreciably more than AC when there are differences among the procedures.

For the relop problems, these differences had a marked effect on subsequent search. (*Note*: Search was not attempted for the langford and blackhole problems. For the rlfap-graph problems, search was backtrack-free in all cases.) The mean numbers of search nodes were: following AC, $>$ 438,717 (in this case search was cut off for one problem after ten million nodes), following SAC, 1148, and following NSAC, 9757.

The results for SAC algorithms on the graded series of relop problems are shown in Fig. 10. Here, again, SACQ is somewhat better than SAC-1 on the harder problem classes. In this case, SAC-3 is roughly comparable to SACQ, being faster for the problems with

Table 9

Number of values removed by different procedures for different classes of structured problems

Problems	AC	SAC	NSAC	SAC-1r	SACQ-adj
rlfap-sub	470.7	1145.6	1145.6	1145.6	497.1
rlfap-graph	279	1037.5	777.5	1012	1037.5
langford-2(np)	336.4	337.6	337.6	337.6	337.6
langford-3(np)	812.4	812.4	812.4	812.4	812.4
blackhole-4-4	290	290	290	290	290
blackhole-4-7	280	280	280	280	280
blackhole-4-13	793	793	793	793	793
relop-geneq	0	758.1	554.8	536.8	758.1

Notes: Values are mean numbers removed. "(np)" indicates that these are the problems *not* proven unsatisfiable by *any* preprocessing algorithm.

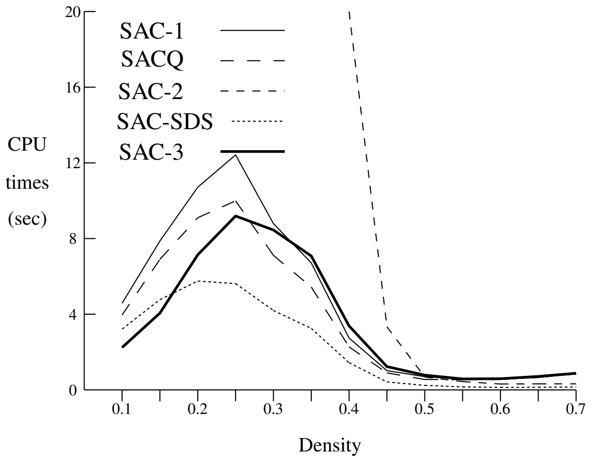

Fig. 10. CPU times for SAC algorithms. Random relop problems (50% greater-than, 50% not-equal constraints) of varying density.

the sparser constraint graphs and somewhat slower for problems with denser constraint graphs. Interestingly, SAC-SDS is appreciably faster than either of these algorithms except for the densest problems. SAC-2 is again much slower than the other algorithms on all but the densest problems. (For this algorithm, mean times for densities 0.1–0.35 ranged between 69 and 180 sec.) For these problems, NSAC algorithms gave mean times that were always less than 1.75 sec.

To evaluate these results statistically, peak mean values were compared as with the random problems; here, this occurred at density $= 0.25$. The differences between SAC-SDS and the other algorithms were highly significant statistically. In addition, the differences between SAC-1 and SAC-3 and between SAC-1 and SACQ were statistically significant at well beyond the 0.01 level.

9. A hybrid neighbourhood SAC algorithm

One reason for considering NSAC algorithms is that these consistency algorithms are efficient enough to be used in combination with backtrack search, where they would be expected to restrict search more effectively than AC procedures. Obviously, they can be used in a variety of ways ranging from a forward-checking version, in which NSAC is only applied to the immediate neighbourhood of the current variable, to a full maintained-AC version. To date only the latter, "maintained neighbourhood singleton arc consistency" (MNSAC) algorithm has been coded and tested.

The present version uses NSACQ, which in turn uses AC-3 for AC maintenance; hence it will be referred to as MNSACQ-3. Like the other algorithms discussed in this paper, it was coded in lisp. Since the general structure of the algorithm is exactly the same as the classical MAC-3 algorithm [12], it will not be described in detail here. In the present version, the NSACQ procedure follows the procedure shown in Fig. 2 exactly; thus, there is an initial AC pass, and if this does not fail, then the remainder of the NSACQ procedure is carried out as shown.

MNSACQ-3 was compared with MAC-3 on a few problem sets. The first was a set of 100 50-variable random CSPs used in previous work (e.g. [15]), with parameters $\langle 50, 10, 0.184, 0.369\rangle$. The variable ordering heuristic was minimum domain/forward-degree; values were chosen lexically. On these problems MAC-3 required an average of 1621 search nodes, and the CPU time was 4 sec. MNSACQ-3 solved these problems in an average of 751 nodes, with a CPU time of 37 sec.

In addition, these algorithms were tested on the heterogeneous problems described above. Here, the difference in nodes was much greater; in many cases, the CPU time was also in favour of MNSACQ-3, since the reduction in search was in orders of magnitude. However, MNSACQ-3 failed to solve one problem because it had a much larger search tree; search was terminated after several hours. MAC-3 also failed to solve this problem, but it reached the million-node cutoff after two hours.

From these preliminary results, I draw the tentative conclusion that maintained neighbourhood SAC probably cannot serve as a robust, 'all-purpose' algorithm like MAC or forward-checking. However, it is interesting that it is a viable competitor in some cases, which is not generally true of algorithms that try to establish higher levels of consistency during search. Further work should focus on the possibility of using this form of consistency maintenance more selectively *during search*, since these results show that it does restrict search much more effectively than AC in many cases. It is also possible that MNSAC will prove to be more effective than MAC on specialized classes of problems.

10. Conclusions

Although singleton arc consistency is often a more powerful filtering algorithm than ordinary arc consistency, its increased effectiveness comes with considerable cost. A standard approach to this type of problem is to discover procedures, or even better, forms of consistency that are not as strong as a given type but which are adequate for many purposes and which allow more efficient algorithms to be devised. In this paper we present results of this type: (i) a reduced form of consistency which is still a kind of SAC, with algorithms that achieve it, (ii) new algorithms for achieving full SAC or approximations to it.

Neighbourhood SAC is a new form of consistency with some appealing properties. It is characterized by a well-defined fixpoint. In addition, it often produces impressive results (number of values deleted) in a fraction of the time that SAC requires. This is true both for random and for some structured problems.

There are also *a priori* reasons for considering NSAC algorithms. NSAC combines the pruning power of SAC with the neighbourhood aspect that is inherent in AC processing, as first shown by AC-2/3 [9,16]. This allows it to be reasonably effective while at the same time avoiding the massive overhead that comes with full SAC procedures. These features also made it possible to devise a new hybrid algorithm, maintained neighbourhood SAC, that was sometimes competitive with MAC, even with a simple initial implementation.

With regard to full SAC, improvements to the original SAC algorithm proposed in earlier work involve fairly complicated procedures and elaborate data structures [1,2,8]. In contrast, the actual procedure – and hence the code – for the SAC algorithms proposed here is very simple. Yet these algorithms often improve on the original SAC-1 algorithm in its present implementation.

This work also gives abundant evidence that the extra processing required by the advanced forms of SAC can impair overall performance, sometimes severely. In contemplating the algorithms in detail, it is not too surprising that they are sometimes less efficient. Each of these algorithms employs elaborate data structures, which in the case of SAC-2 and SAC-SDS can grow to an enormous size. Thus, for SAC-2, the SAC-support lists involve putting a value proven to be singleton-consistent on the list of every other value in the reduced problem; these lists are, therefore, enormous. SAC-SDS requires separate queues for each copy of the original and a large queue for the basic SAC tests. In addition, there is special updating needed to ensure that each copy has correct domains (procedure "updateSubProblems" in [2]). Although, subsequent processing of a copy is rendered very efficient, since consistency testing need only consider these changes, this is to some degree countered by the updating. SAC-3 also requires special bookkeeping for updating (described in Section 2). Despite what appeared to be efficient data structures, this was not always enough to counter the updating requirements.

On the other hand, SAC-3 is clearly able to take advantage of the redundancy inherent in problems with clearly defined structure. As a result, it was often (but not always) the fastest full SAC algorithm (see Table 7). Nonetheless, there was evidence that for larger problems, its advantages could be lessened by the requisite bookkeeping.

These results raise questions regarding the adequacy of worst-case time complexity analysis when reductions in what is taken to be the dominant operation are 'off-loaded' onto other operations that may take as much or more time. This problem has already been noted in the case of AC-2001 in relation to AC-3 (see [14]), but it is much more pronounced in the present case. It seems clear that up until now there has not been sufficient consideration of the context in which such analyses are made.

To highlight the problem, consider first the case of internal comparison sorting algorithms, e.g. quicksort vs. bubble sort. Suppose moves is taken to be the dominant operation; in this case because of the properties of random access memory, the larger moves made by quicksort can be done in the same amount of time as the small moves made by bubble sort. In this case, an analysis of worst-case time complexity based on this operation should be reflected in performance because both algorithms are based on the same dominant operation, which can be done in essentially the same fashion.

However, this is typically not the case with consistency algorithms, where a reduction in consistency checks is obtained only by incorporating other operations, which are not taken into account in the analysis of time-complexity. The present empirical tests, which involved implementations in the same language, with sharing of code, in particular the basic AC procedure, among the different algorithms, make it clear that worst-case analyses of this sort can be very misleading with regard to overall efficiency. One can cavil about the use of lisp or speculate about lack of tuning in the implementation (although I believe that the present implementations are reasonably well tuned). However, I think that this is evading a serious issue that is important to address: how does one evaluate complex algorithms in terms of their overall efficiency, when new operations are added in some cases but not others?

One of the appealing features of the original SAC-1 algorithm is that it is easy to add to existing implementations (R. Bartak, personal communication), unlike the more elaborate SAC algorithms. Given the simplicity of the new SAC algorithms (SACQ and SACQ-adj), they should also have this feature; this enhances their potential importance for practical solvers.

Still another issue encountered in the experiments with structured problems is that two of these algorithms (SAC-2 and SAC-SDS) required updating with each new class of problems tested. For SAC-SDS, the problem was related to subproblem indexing, which was based on the variable and value number. Thus for rlfap problems, since domains are not sequential and sometimes have large maximum values, a separate data structure was used to specify the location of a value in its domain. Analogous problems were encountered in SAC-2, where various arrays had to be indexed by domain values. In contrast, with the new SAC (and NSAC) algorithms, the same code could be used once the AC functions were updated to handle the specific kinds of constraints in the new problems. This was also true for SAC-3, which despite its complexity was quite 'robust' in this regard.

The present work also shows that a simple form of partial SAC is almost as effective as full SAC while requiring much less effort. To my knowledge the only previous examination of partial SAC is the one-pass SAC procedure called "restricted SAC" [11], which was also tested here. Although the latter procedure runs faster than SACQ-adj (as expected), it is usually not as effective in deleting values. This allows an interesting tradeoff between time reduction and closeness in effect to full SAC algorithms.

In contrast to NSAC, where there is a well-defined form of consistency associated with the procedure, with SACQ-adj, as well as restricted SAC, this is not the case. Although this difference may not be of great practical importance, it does make it more difficult to characterise the latter procedures. It also makes it difficult to draw inferences concerning the level of consistency obtained, which might be of considerable importance for specific classes of problems.

To summarize the results with respect to overall runtimes: SAC-3 is generally best for structured problems, provided they are not too large. SACQ was the best SAC algorithm for certain problem classes, including radio frequency problems and various kinds of random problem. Results for the former suggest that it scales well as problem size increases. For neighbourhood SAC algorithms, NSACQ was best overall. All of these algorithms are much more expensive than simple arc consistency, but for some problem classes the payoff can be considerable.

The present work shows that there are a variety of interesting and potentially significant forms of reduced or partial singleton arc consistency. In the future, it will be of interest to see whether other selective SAC algorithms can be devised, perhaps based on other well-characterised features of constraint satisfaction problems. At the same time, it may be difficult to find combinations of algorithmic features that are as felicitous as the combination of SAC and neighbourhood consistency, because of the neighbourhood properties of arc consistency that have been highlighted in this paper.

Acknowledgements

NSACQ was derived from an algorithm for enforcing neighbourhood inverse consistency (NIC) implemented by D. Grimes. This implementation was also the basis for the NIC algorithm used in the present experiments (with the changes mentioned in the text). The author would also like to acknowledge the help of the anonymous reviewers, whose comments greatly aided the development of the formal arguments, and were responsible for the addition of Proposition 6 to the paper as well as statistical tests.

References

[1] R. Bartak and R. Erben, A new algorithm for singleton arc consistency, in: *Proc. 17th Internat. FLAIRS Conf.*, Vol. 1, 2004, pp. 257–262.

[2] C. Bessière and R. Debruyne, Optimal and suboptimal singleton arc consistency algorithms, in: *Proc. 19th Internat. Joint Conf. on Artif. Intell. – IJCAI'05*, 2005, pp. 54–59.

[3] C. Bessière and R. Debruyne, Theoretical analysis of singleton arc consistency and its extensions, *Artificial Intelligence* **172** (2008), 29–41.

[4] R. Debruyne and C. Bessière, Some practicable filtering techniques for the constraint satisfaction problem, in: *Proc. 15th Internat. Joint Conf. on Artif. Intell. – IJCAI'97*, Vol. 1, 1997, pp. 412–417.

[5] E.C. Freuder and C.D. Elfe, Neighborhood inverse consistency preprocessing, in: *Proc. 13th Nat. Conf. of the Amer. Assoc. Artif. Intell. – AAAAI'96*, Vol. 1, AAAI/MIT, 1996, pp. 202–208.

[6] W.L. Hays, *Statistics for the Social Sciences*, 2nd edn, Holt, Rinehart & Winston, 1973.

[7] D.S. Johnson, C.R. Aragon, L.A. McGeoch and C. Shevron, Optimization by simulated annealing: an experimental evaluation. Part II. Graph coloring and number partitioning, *Opns. Res.* **39** (1991), 378–406.

[8] C. Lecoutre and S. Cardon, A greedy approach to establish singleton arc consistency, in: *Proc. 19th Internat. Joint Conf. on Artif. Intell. – IJCAI'05*, 2005, pp. 199–204.

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

[10] R. Mohr and T.C. Henderson, Arc and path consistency revisited, *Artificial Intelligence* **28** (1986), 225–233.

[11] P. Prosser, K. Stergiou and T. Walsh, Singleton consistencies, in: *Principles and Practice of Constraint Programming – CP 2000*, R. Dechter, ed., LNCS, Vol. 1894, Springer, 2000, pp. 353–368.

[12] D. Sabin and E. Freuder, Contradicting conventional wisdom in constraint satisfaction, in: *Proc. Eleventh European Conf. on Artif. Intell. – ECAI'94*, Wiley, 1994, pp. 125–129.

[13] N. Sadeh and M.S. Fox, Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem, *Artificial Intelligence* **86** (1996), 1–41.

[14] M.R.C. van Dongen, Saving support-checks does not always save time, *Artificial Intelligence Review* **21** (2004), 317–334.

[15] R.J. Wallace and D. Grimes, Experimental studies of variable selection strategies based on constraint weights, *Journal of Algorithms: Algorithms in Cognition, Informatics and Logic* **63** (2008), 114–129.

[16] D. Waltz, Understanding line drawings of scenes with shadows, in: *The Psychology of Computer Vision*, P.H. Winston, ed., McGraw-Hill, 1975, pp. 19–91 (Chapter 2).