AI Communications 29 (2016) 249–268
DOI 10.3233/AIC-150696
IOS Press

Neighbourhood SAC: Extensions and new algorithms

Richard J. Wallace
Insight Centre for Data Analytics, Department of Computer Science, University College Cork, Cork, Ireland
E-mail: richard.wallace@insight-centre.org

Abstract. Neighbourhood singleton arc consistency (NSAC) is a type of singleton arc consistency (SAC) in which the subproblem formed by variables adjacent to a variable with a singleton domain is made arc consistent. This paper describes two extensions to neighbourhood SAC. The first is a generalization from NSAC to k-NSAC, where k is the maximum length of the shortest path between the singleton variable and any variable in the subgraph. The second is an extension of k-NSAC to problems with n-ary constraints, which retains the basic definition of a k-neighbourhood subgraph. To establish such consistencies a suite of algorithms is considered based on various SAC algorithms including SAC-1, SACQ, SAC-SDS, and SAC-3. In analyzing these different algorithms it was found useful to distinguish between "light-weight" and "heavy-weight" SAC algorithms, based on the complexity of data structures and procedures needed to carry out the task of establishing (N)SAC. Under this classification, SAC-1 and SACQ are light-weight; the other two (and SAC-2) are heavy-weight. It was found that only light-weight algorithms can be readily and effectively transformed into efficient NSAC algorithms. In contrast, because of their specialized procedures, it was necessary to modify heavy-weight algorithms significantly, which also compromised performance. Extensive experimental analysis shows that with a spectrum of neighbourhood consistencies and attendant algorithms, one can finesse the fundamental tradeoff between efficiency and effectiveness across a greater range of problems than with SAC and NSAC algorithms alone. This work serves to enlarge the scope of SAC-based consistency maintenance as well as defining the various niches that light-weight and heavy-weight algorithms are best suited for.

Keywords: Constraint satisfaction, arc consistency, singleton arc consistency

1. Introduction

Arc consistency (AC) is the best-known form of local consistency processing within the field of constraint solving. This is because it manages the tradeoff between processing effort and search space reduction in a very effective fashion. Singleton arc consistency (SAC) is a more powerful form of consistency, although it is still based on AC. As such it is often more effective than AC, either in removing values or proving unsatisfiability; however, it is much more expensive.

Neighbourhood SAC is a form of singleton arc consistency in which consistency with respect to single domain values is established only with respect to the neighbourhood of the variable with a singleton domain (the "focal variable"), i.e. the subgraph that includes the latter plus its neighbours in the original constraint graph. Although weaker than SAC, it is still a stronger form of consistency than AC [11]. As would be expected, this form of consistency can be established much more efficiently than full singleton arc consistency; at the same time, it often deletes many more values than simple AC, sometimes almost as many as SAC [11]. As a result, it has considerable potential as a specialized form of SAC-based consistency.

The present paper enlarges upon the original idea of neighbourhood SAC along with its attendant algorithms in two different ways. First, it extends the idea of neighbourhood SAC to k-neighbourhood SAC, where k is the maximum length of the shortest path from the focal variable to any variable in the subgraph. Within this new framework, neighbourhood SAC becomes 1-neighbourhood SAC. Second, it extends the notion of k-neighbourhood consistency from the binary to the more general non-binary case. In connection with this, issues have to be addressed concerning what to include in the k-neighbourhood subgraph, as will be explained.

A further contribution involves a new distinction among SAC-based algorithms. Algorithms like SAC-1 and SACQ that require only simple data structures and procedures can be called "light-weight", while al-

gorithms like SAC-2, SAC-SDS, and SAC-3, which require elaborate data structures and special procedures to maintain them, can be called "heavy-weight". Previous work has only considered light-weight algorithms in connection with NSAC [11]. As we will see, this distinction brings to light some important differences between SAC and NSAC with respect to the efficiency of different algorithmic strategies in establishing these forms of consistency.

The next section gives general background concepts and definitions. Section 3 is a discussion of SAC and NSAC and the algorithms used to establish these properties. It also shows how different SAC algorithms can be modified to perform NSAC. Section 4 introduces k-NSAC and Section 5 presents experimental comparisons related to the various forms of k-neighbourhood consistency and to the different algorithms that can be used to establish these properties. Section 6 describes extensions of these ideas to non-binary constraints and Section 7 presents experimental comparisons among the different levels of neighbourhood SAC (as well as SAC) for problems with n-ary constraints. Section 8 discusses these results and the benefits of performing k-neighbourhood SAC over full SAC.

2. Background concepts

A constraint satisfaction problem (CSP) is defined in the usual way, as a tuple, (X, D, C) where X are variables, D are domains such that D_i is associated with X_i, and C are constraints. A *solution* to a CSP is an assignment or mapping from variables to values that includes all variables and does not violate any constraint in C.

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} with X_i in its scope, there is at least one value b in the domain of X_j such that (a, b) satisfies that constraint. For non-binary, or n-ary, constraints generalized arc consistency (GAC) refers to the property that for every value a in the domain of variable X_i and for every constraint C_j with X_i in its scope, there is a valid tuple that includes a.

Singleton arc consistency, or SAC, is a 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, and a can be discarded. If this condition can be established for all values in problem P, then the problem is singleton arc consistent. (Obviously, SAC implies AC, but not vice versa.)

Neighbourhood SAC establishes SAC with respect to the neighbourhood of the variable whose domain is a singleton.

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. Variables belonging to X_N are called the neighbours of X_i.

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

In this 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.

The following proposition from [11] will be used in some of the proofs and discussion:

Proposition 1 (Neighbourhood Lemma)**.** *For any algorithm involving AC testing and no other form of consistency maintenance, if removal of value a 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.*

3. SAC and NSAC algorithms

3.1. Light-weight vs. heavy-weight algorithms for SAC

In examining NSAC and its extensions we will employ both light-weight and heavy-weight algorithms. As noted in the Introduction, SAC-1 [6] and SACQ [11] can be considered light-weight algorithms, while

SAC-2 [2], SAC-SDS [3] and SAC-3 [9] are heavyweight. In earlier work, NSACQ was found to be appreciably faster than NSAC-1, while SACQ was somewhat faster than SAC-1 for most problem classes [11]. Of the heavy-weight algorithms, SAC-3 was most efficient on most problem classes. For random problems and RLFAPs, SACQ was the most efficient overall; for other problem classes, the best heavyweight algorithm (SAC-3) was also the best overall [11].

In the present work, all but one of these SAC algorithms were modified to produce neighbourhood SAC algorithms. (In previous work, SAC-2 was found to be much slower than any other algorithm on all problem classes, so it is omitted here.) Before describing the modifications, the basic SAC algorithms will be briefly reviewed. More information, including further implementation details, can be found in [11].

SAC-1, the original SAC algorithm [6], uses an AC-1-style procedure. This means that all values in all domains are tested for singleton arc consistency in each major pass of the procedure, and this continues until no values are deleted. Whenever a value is deleted (because instantiating a variable to this value led to a wipeout), a full AC procedure is applied to the remaining problem before testing the next singleton value.

The SAC-SDS algorithm [3,4] is a modified form of the authors' "optimal" SAC algorithm, SAC-Opt. The key idea of SAC-SDS (and SAC-Opt) is to represent each SAC reduction separately; consequently there are $n \times d$ problem representations (where n is the number of variables and d 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, the value is deleted from the master problem and this problem is made arc consistent again. If this leads to failure, the problem is inconsistent; otherwise, all values that were deleted in making 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 resubjected to a SAC-test.

SAC-SDS also uses 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. They are used whenever the subproblem is taken off the pending list. In addition, if a SAC-test leads to failure, the subproblem involved can be taken 'off-line' to avoid unnecessary processing. 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 that subproblem cannot occur.

The SAC-3 algorithm [9] 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. the first test in a series. This strategy is carried out within the SAC-1 framework. That is, the major pass in which all existing assignments are tested for SAC is repeated until there is no change to the problem.

SACQ uses an AC-3 style procedure at the top-level instead of the AC-1 style that is often used with SAC algorithms. This means that there is a list (a queue) of variables, whose domains are considered in turn; in addition, if there is a SAC-based deletion of a value from the domain of X_i, then all variables not currently on the queue are put back on. Unlike other SAC (or NSAC) algorithms, there is no "AC phase" following a SAC-based value removal. The idea is that if a deletion from the domain of focal variable X_i has any further effects on the consistency of the network, then by putting all the variables back on the queue, this will be discovered by subsequent SAC tests (since SAC dominates AC). Pseudocode for this algorithm is shown in Fig. 1.

In this work, both NSAC and SAC algorithms were preceded by a step in which arc consistency was established (e.g., Fig. 1, line 2), although this is not required to establish either SAC or NSAC. This was done to rapidly rule out problems in which AC is sufficient to prove unsatisfiability. It also eliminates inconsistent values which are easily detected using a less expensive consistency algorithm.

3.2. Altering SAC algorithms to perform NSAC

Four SAC algorithms were used as the basis for NSAC algorithms: SAC-1, SACQ, SAC-SDS and

```
        Function SACQ
1           Q ← X
2           OK ← AC(P)
3           While OK and not empty-Q
4               Select and remove X_i from Q
5               Changed ← false
6               For each v_j ∈ dom(X_i)
7                   dom'(X_i) ← {v_j}
8                   If AC(P') leads to wipeout
9                       Changed ← true
10                      dom(X_i) ← dom(X_i)/v_j
11                      If dom(X_i) == ∅
12                          OK ← false
13              If Changed == true
14                  Update Q to include all X_j ∈ X
15          Return OK
```

Fig. 1. Pseudocode for SACQ.

SAC-3. Descriptions of NSAC based on SAC-1 and SACQ were reported in [11]. Since heavy-weight SAC algorithms often outperform light-weight SAC algorithms [3,9,11], versions of these algorithms may also be effective in the context of neighbourhood SAC.

As shown in earlier work, modification of SAC-1 and SACQ to perform neighbourhood singleton arc consistency is straightforward. In both cases, SAC-based consistency testing is confined to the subgraph induced by the focal variable and its neighbours. For NSACQ, this means changing line 8 in Fig. 1 from

If AC(P′) leads to wipeout

to

If AC(X_i + neighbours(X_i)) leads to wipeout

and line 14 from

Update Q to include all $X_j \in X$

to

Update Q to include all neighbours of X_i.

For NSAC-1 only the first change is made.

To convert SAC-SDS to NSAC-SDS it was necessary to eliminate the copy-queues and to substitute a different means of updating subproblems. This is because carrying out AC with copy-queues based on values deleted from the master takes one beyond the neighbourhood subgraph; this may cause inconsistency and deletion of the singleton value even if the subgraph associated with it is consistent. Instead, if any values are removed from a subproblem it is put back on the pending list; then, when it is subsequently taken off the list, singleton arc consistency is re-established for the neighbourhood subgraph. A variant of this procedure was also tested in which a subproblem was *not* put back on the pending list even though values were deleted from it, if none of those values were in domains of variables in the neighbourhood subgraph represented by that subproblem. This had mixed results: in some cases this helped speed up the procedure appreciably, but in other cases the algorithm was slower (see below).

Pseudocode for NSAC-SDS is shown in Fig. 2. In this figure, the only difference from the corresponding SAC algorithm is the line where AC is performed on the subproblem. In this case, AC is only applied to the neighbourhood subgraph. Moreover, this is done without a subproblem copy-queue. In addition (and this is not shown in the figure), unlike the corresponding SAC algorithm, where the initial AC queue can be restricted to tests of neighbours against variables in the copy-queue, here all variables must be tested against any other variable in the subgraph if there is a constraint between them. (The reason for this is explained below.)

Pseudocode for NSAC-3 is shown in Fig. 3(a) and (b). For this algorithm, a similar procedure of testing all arcs in a subgraph on each AC pass had to be followed. If variable–value pairs are successively added to the NSAC-3 branch, it is possible to choose a pair whose subgraph is no longer arc consistent due to previous NSAC-based processing, but where supports can be found in all neighbouring domains for the singleton value. In this case (in contrast to SAC-3), if one restricted the initial arc consistency queue to arcs involving the focal variable, the inconsistency would not be detected.

A simple example of this is shown in Fig. 4, where two focal variables are indicated by filled circles and Roman numeral labels, and their neighbours are indicated by outline circles labeled with Arabic numerals. Suppose that (I, v_{Ik}) is added to the branch and NSAC carried out, that a value belonging to variable 3 is deleted, and that there is at least one value in the domain of variable 4 that required that value for support. Now suppose that there was no wipeout, and that the next pair added to the branch is (II, v_{IIl}). If we were carrying out full SAC, we would only need to check the domains of variables adjacent to II, since the previous

```
    Function NSAC-SDS
1       OK ← ACm(P)
2       If OK
3           Q ← {(X_i, v_ik)}
4           While OK and not empty-Q
5               Select and remove next pair (X_i, v_ik) from Q
6               If v_ik ∉ D_i continue
7               If ∄ subproblem for (X_i, v_ik)
8                   Set up subproblem
9                   sub-problem-flag(X_i, v_ik) ← true
10              consistent ← ACs(X_i ∪ neighbours(X_i))
11              If not consistent
12                  sub-problem-flag(X_i, v_ik) ← false
13                  dom(X_i) ← dom(X_i)/v_ik        ; in master
14                  consistent ← ACm(P)
15                  If consistent
16                      deletion-list ← values deleted from master problem
17                      For each subproblem (X_j, v_jk) for which sub-problem-flag is true
18                          Remove any domain values that are in the deletion list
19                          If there are such deletions
20                              Put (X_j, v_jk) back on Q
21                  Else
22                      OK ← false
23      Return OK
```

Fig. 2. Pseudocode for NSAC-SDS. "ACm" refers to AC performed on the master problem; "ACs" refers to AC performed on a subproblem.

step would have established arc consistency throughout the problem. But in the present case we would not have checked variable 4 against variable 3 in the previous step (since 4 is not in variable I's neighbourhood), and it is possible that the values of variables 3 and 4 are all supported by v_{III} in the domain of variable II. To avoid overlooking an inconsistency in this case, variables like 4 must be checked against adjacent neighbourhood variables like variable 3 and vice versa.

As already indicated, a similar situation holds for NSAC-SDS. In this case it is because values deleted in the master problem may occur anywhere in the subgraph, and copy-queues are no longer used. In this case, the same situation may occur as with NSAC-3, where values in domains of variables adjacent to the focal variable are all supported by the singleton value, but there are inconsistencies between neighbour domains.

Proposition 2 (Correctness of NSAC-SDS)**.** *For any CSP, NSAC-SDS establishes neighbourhood singleton arc consistency correctly if all constraints in the neighbourhood subgraph are examined during each NSAC test.*

Proof. *Soundness*. If a value is deleted, this means that testing for AC in a subproblem led to a wipeout or that reestablishing AC in the master caused a deletion. Subproblems are always set up based on the current domain values (line 8 of Fig. 2), and any existing subproblem is retested only if it contains values deleted from the master. Thus, it is impossible for a spurious wipeout to occur, leading to deletion of a value that is actually neighbourhood singleton arc consistent. (If a value is deleted because of an NSAC test, its subproblem is never readded to the queue; moreover, a domain check after taking an entry off the queue (line 6) prevents spurious subproblem testing.)

Completeness. The algorithm can be complete only if every subproblem that is ever associated with a neighbourhood-SAC inconsistent value is tested for neighbourhood SAC. Since all subproblems are put on the queue initially, all are tested at least once. We need only consider cases where a value that was consistent becomes inconsistent because other values were deleted, and this always occurs in connection

```
    Function NSAC-3
1       OK ← AC(P)
2       If OK
3           Repeat
4               Q ← {(X_i, v_ik)}
5               changeflag ← false
6               While OK and not empty-Q
7                   OK ← Buildbranch
8           Until not changeflag ∨ not OK
9       Return OK
```

(a)

```
    Function Buildbranch
1       branch ← ∅
2       remainingvars ← {X_i}
3       consistent ← true
4       While consistent and remainingvars ≠ ∅
5           Select and remove next pair (X_i, v_ik) from Q
6           dom'(X_i) ← {v_ik}
7           If AC(X_i ∪ neighbours(X_i)) leads to wipeout
8                   consistent ← false
9           If not consistent ∧ branch = ∅
10                  dom(X_i) ← dom(X_i)/v_ik
11                  consistent ← true
12                  changeflag ← true
13                  If AC(P) leads to wipeout
14                      consistent ← false
15                  Else
16                      Revise Q to eliminate deleted values
17          Else if not consistent
18              Put last (X_i, v_ik) back on Q
19              Return true
20          Else
21              Add (X_i, v_ik) to branch
22              Remove (X_i) from remainingvars
23      Return consistent
```

(b)

Fig. 3. (a) Pseudocode for NSAC-3 main function. (b) Pseudocode for NSAC-3 function Buildbranch.

with deletions in the master problem. Under these conditions, (i) if the subproblem already exists (i.e. it was created (Fig. 2, line 8) and tested), and it now has deleted values, it will be put back on the queue and retested (lines 17–20, "For each subproblem…"), (ii) if the subproblem hasn't yet been created, and its singleton value is still viable, then it will still be on the queue, and when the subproblem is created, deleted values will not be included.

Termination. This is obvious, given that the number of domain values is finite. □

Proposition 3 (Correctness of NSAC-3)**.** *For any CSP, NSAC-3 establishes neighbourhood singleton arc consistency correctly if all constraints in the neighbourhood subgraph are examined during each NSAC test.*

In proving correctness in this case, we can base our arguments on the correctness of SAC-3 [9]. Sound-

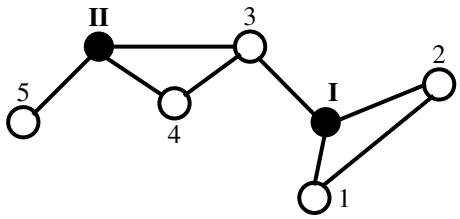

Fig. 4. Constraint graph to demonstrate potential incorrectness of NSAC-3 if all constraints in neighbourhood subgraph are not tested directly (because a wipeout may not be detected).

ness is ensured because, as with SAC-3, NSAC-3 does not remove values unless inconsistency is detected at the beginning of a branch. (The subsequent AC pass in this case is done just as in the SAC algorithm.) Regarding completeness: if a value passes the NSAC test in a problem reduced by previous NSAC tests, it will of course pass the test in the unreduced problem. Hence, using the SAC-3 strategy in this case will not overlook any neighbourhood-SAC inconsistent values. Given the basic SAC-1 style repeat loop, which ensures that neighbouring domains will be retested subsequent to a deletion, all neighbourhood-SAC inconsistent values will eventually be detected.

4. Extending neighbourhood SAC to k-neighbourhood SAC

In this paper we introduce a significant extension to the original concept of neighbourhood singleton arc consistency, which is based on an expansion of the neighbourhood concept to the more general one of "closure" or k-neighbourhood.

The idea behind k-neighbourhood SAC is straightforward: instead of restricting the variables in the subgraph to be made singleton consistent to neighbours of the focal variable, one extends the subgraph to include all those variables that are connected by a path of length k or less to the focal variable. A k-neighbourhood is a special case of the *closure* of a subgraph of a CSP as defined in [7].

Definition 4 (After [7])**.** Let S be a subset of variables of problem P and PS be the subproblem induced by those variables. The *closure* of PS is the subproblem induced by S and all variables that share a constraint with a variable in S.

Given this definition, we can define the k-neighbourhood recursively.

Definition 5. A 1-neighbourhood is a closure where S is the single variable X_i. Then given a k-neighbourhood ($k = 1, 2, \ldots$), the $k+1$-neighbourhood is the closure of the k-neighbourhood.

There are several reasons for investigating these forms of consistency:

- This form of extension continues to take advantage of the "neighbourhood character" of AC, as reflected in the Neighbourhood Lemma. That is, value deletions in the domain of variable X_i will lead to deletions in the domain of X_j only if deletions occur in successive neighbourhoods beginning with that of X_i and ending with a neighbourhood of which X_j is a member.
- Evaluating k-neighbourhood SAC algorithms for different values of k may allow us to discover forms of singleton arc consistency that are best able to handle the tradeoff between effort and effectiveness for a given class of problems, where effort is the time taken to achieve the given level of consistency and effectiveness is the number of values deleted and the proportion of unsatisfiable problems proven unsatisfiable during preprocessing.
- Following from the last-mentioned point, with a series of consistencies of this sort, there may be opportunities to tune the level of consistency, i.e. to learn the best level of consistency for a given class of problems or for a given problem instance or for a given state of search.

Proposition 4. *For any value of k, $k+1$-NSAC dominates k-NSAC, unless both subgraphs include all the variables in the problem. That is* (*except for the last-mentioned condition*)*, if a value is removed by k-NSAC then it will also be removed by $k+1$-NSAC, but the converse does not hold.*

Proof. The proof follows the scheme of the proof that full SAC dominates NSAC given in [11]. Thus, it is clear that any value removed during singleton AC by k-NSAC will also be removed by $k+1$-NSAC, since the latter includes all the checking done by the former. On the other hand, if a value is deleted by $k+1$-NSAC but not by k-NSAC, this can only occur when values are deleted outside the k-neighbourhood of some focal variable X_i, either by domain wipeout during the $k+1$-NSAC phase of the former or by domain reduction during an AC phase, if that is included as part of the proce-

dure. In case of AC phase reduction, deletion of values in neighbouring domains will also occur with k-NSAC because these values will be tested in subsequent k-NSAC phases. This is guaranteed because NSAC in any form dominates AC ([11]), and by the Neighbourhood Lemma, we are assured that any removal of a singleton value (which triggered the AC phase) can only lead to other value removals through a succession of removals in neighbouring domains, which are subsumed by a succession of k-neighbourhoods. On the other hand, it is possible for reductions in domains in a k-neighbourhood to imply a wipeout outside that neighbourhood, so that an algorithm that establishes $k+1$-consistency can deduce that a value cannot participate in a solution while k-NSAC applied to the same variable and value in the same state will not. Such failures will not necessarily be discovered during subsequent passes of the k-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 would lead to this SAC-based wipeout. That such conditions actually occur can be shown by example. Thus, the example given in Fig. 4 of [11] to show that full SAC dominates NSAC also applies to the 2- versus 1-NSAC comparison. It is obvious that the arrangement that gives rise to the difference in this example can be associated with variables in the $k+1$-neighbourhood of X_i, so that establishing k-neighbourhood consistency will not necessarily remove all values removed by $k+1$-neighbourhood consistency. □

The algorithms used in this paper to establish each successive level of neighbourhood consistency are essentially the same as the basic NSAC (i.e. 1-NSAC) algorithms. The only difference is that the variables that make up the set neighbour(X_i) are all those in the k-neighbourhood of X_i. In the present work we only consider algorithms for k-neighbourhood SAC where $1 \leqslant k \leqslant 3$.

5. Experimental tests of k-neighbourhood SAC algorithms

Experimental tests are necessary to compare the way in which the tradeoff between effectiveness and efficiency varies with different levels of SAC, and to compare the different SAC and k-NSAC algorithms. For these tests, homogeneous random, structured random and benchmark problems were used. Randomly generated problems allow systematic exploration of parameter spaces as well as sufficiently large samples. Benchmarks provide further tests of generality with less artificial problems.

In this work, algorithms were implemented in Common Lisp, and experiments were run in the XLisp-stat environment with a Unix OS on a Dell Poweredge 4600 machine (1.8 GHz). In all experiments, tests were run as a block, one immediately after the other, in order to minimize timing differences due to extraneous actions of the operating system such as those involving resource allocation and job swapping.

5.1. Random problems

The first tests used homogeneous random problems with the same parameters as those in [9], with the additional requirement that the constraint graphs be connected. This serves to relate the present experiments to earlier work on SAC algorithms and is, therefore, an appropriate place to begin an experimental analysis. These problems have $\langle n, d, p_1, p_2 \rangle$ parameter values $\langle 100, 20, 0.05, t \rangle$, where t varies from 0.10 to 0.90 in steps of 0.05. (n is the number of variables, d the domain size, p_1 the constraint graph density, and p_2 the constraint tightness.) There were 50 problems for each value of t.

For algorithms based on neighbourhood properties, it is useful to know the difference between the size of a k-neighbourhood and n, the number of variables associated with a pass of full SAC. For random CSPs a formula for the expected neighbourhood size can be derived;[1] however the derivation is not straightforward, and since it depends on random sampling assumptions, it will not necessarily hold for non-homogeneous random or structured problems. In addition, it is useful to assess variability along with central tendency values. For these reasons, in this paper means and standard deviations for k-neighbourhood size were derived empirically for each class of problem tested. When a problem set involved only a small number of problems, all problems were tested; for larger problem sets as in the present case, the first ten problems were tested. (In the present case, there were several sets of problems, but they all had the same graph density.) For each problem tested, the k-neighbourhood size for *all* variables in the problem was calculated, and from this a mean and standard deviation were derived.

[1] This was demonstrated to the author by one of the reviewers.

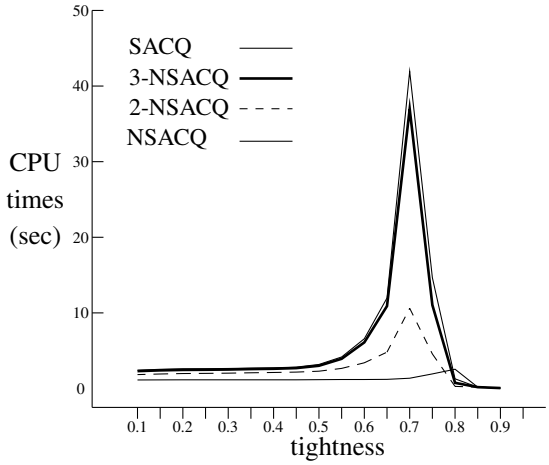

Fig. 5. Runtimes for full SACQ and k-NSACQ algorithms on random problems. (Highest curve is for SACQ, lowest for NSACQ.)

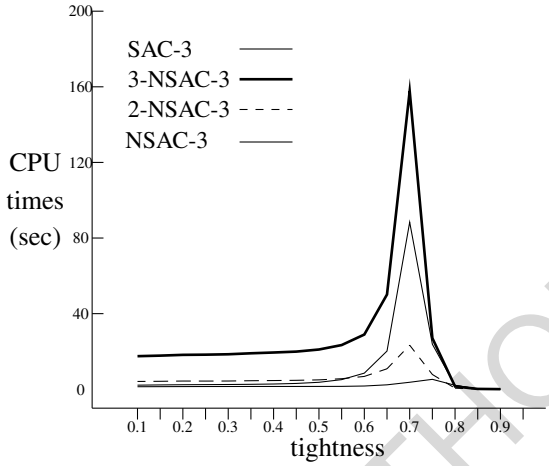

Fig. 6. Runtimes for full SAC-3 and k-NSAC-3 algorithms on random problems. (Second highest curve is for SAC-3, lowest for NSAC-3.)

For the present problems, the mean size of a 2-neighbourhood subgraph in a single problem varied between 26 and 27 variables across the set of sampled problems. For 3-neighbourhood subgraphs the mean varied between 69 and 73. Corresponding standard deviations varied between 8 and 11 ($k = 2$) and 14 and 17 ($k = 3$) across the ten problems. Thus, for these problems both 2-NSAC and 3-NSAC can be distinguished from SAC with respect to the number of variables included in a singleton test.

Figure 5 shows runtimes across the spectrum of tightnesses for (N)SACQ algorithms; each curve is for a different algorithm producing a different level of consistency. Figures 6 and 7 show corresponding results for (N)SAC-3 and (N)SAC-SDS algorithms, respectively. (In these and other figures, runtimes are for pre-

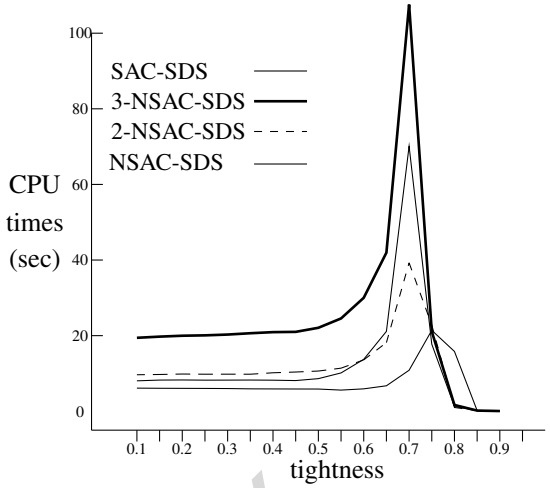

Fig. 7. Runtimes for full SAC-SDS and k-NSAC-SDS algorithms on random problems. (Second highest curve is for SAC-SDS, lowest for NSAC-SDS.)

processing only.) Results for SAC algorithms correspond to earlier work; note that in this case there is little redundancy for SAC-3 to take advantage of.

For algorithms based on SACQ the results are straightforward: as k increases, thus increasing the k-neighbourhood size, runtimes increase. However, the increase is non-linear in k, so that for 2-neighbourhood SAC times are fairly close to NSAC, while for 3-neighbourhood SAC times are greatly elevated, although for the most difficult problems the average is still less than for SACQ. Algorithms based on SAC-1 gave the same pattern of results as SACQ, although runtimes were slightly longer.

For algorithms based on SAC-3, the results are different (Fig. 6). 2-NSAC-3 behaves similarly to 2-NSACQ, although the times are greater. (Note the difference in scale on the ordinates of the two graphs.) However, runtimes for 3-NSAC-3 are much greater than those for 2-NSAC-3, and even greater than for SAC-3. We conclude that the changes that were made to the basic SAC-3 algorithm to enable it to perform k-NSAC correctly seriously compromise its efficiency as k increases.

For algorithms based on SAC-SDS, results are qualitatively similar to those for (N)SAC-3, although the differences are not as pronounced (Fig. 7). Moreover, both NSAC-SDS and 2-NSAC-SDS are inefficient in comparison to the corresponding algorithms based on SAC-1 or SACQ. (NSAC-SDS in the modified form gave somewhat faster runtimes for 1- and 2-NSAC and slower times for 3-NSAC.)

Table 1 shows the relative effectiveness of SAC and k-NSAC algorithms across the part of the tightness

Table 1
Random problems proved unsatisfiable by preprocessing algorithms according to level of consistency achieved

Level of consistency	Problem tightness				
	0.70	0.75	0.80	0.85	0.90
SAC	2	50	50	50	50
3-NSAC	2	50	50	50	50
2-NSAC	0	50	50	50	50
NSAC	0	1	43	50	50
AC	0	0	0	50	50

Notes: Number unsatisfiable in each case is 50. (There were no unsatisfiable problems with tightness < 0.70.)

spectrum where problems have no solutions. Here, the measure of effectiveness is the number of problems proved unsatisfiable. Note that for the two highest tightness values, unsatisfiability is proved in the initial AC pass (which is why the runtime curves converge in Figs 5–7). For the problem set at tightness $= 0.8$, NSAC is clearly more effective than AC, while at 0.75, a higher level of consistency is necessary. This corresponds to the fact that in the three graphs, the peak for NSAC occurs at tightness 0.75–0.8, while for higher levels of consistency, the peak is at 0.7. In addition, it should be noted that 3-NSAC is always as effective as full SAC on these problems.

In summary, for this type of problem 2-NSAC algorithms seem best able to handle the tradeoff between efficiency and effectiveness. In addition, algorithms derived from light-weight SAC procedures are more efficient than heavy-weight algorithms. 3-NSAC is always as effective as the full SAC algorithm, but it is more efficient than SAC only when it is based on a light-weight procedure.

5.2. Structured problems

SAC and k-NSAC algorithms were also tested on structured problems of different types. These included three types of benchmarks obtained from the site maintained at Université Artois.[2] These were RLFAP-graph problems with solutions, blackhole 4–7 problems and os-taillard scheduling problems.

RLFAP-graph problems had either 200 or 400 variables with constraints of the form $|X_i - X_j| > k$ or $|X_i - X_j| = k$ (hereafter called "distance" constraints). For each problem the mean size of the 1-neighbourhood subgraphs was 12. For 2-neighbourhood graphs, means fell between 42 and 50 for different problems. For 3-neighbourhood graphs, the means for the 200-variable problems were 126 and 131 and for the 400-variable problems 178 and 180. Standard deviations varied between 2 and 6 for the 1-, 14 and 20 for 2- and 28 and 53 for 3-neighbourhood graphs. Blackhole problems had 112 variables. All problems had the same means for 1-, 2- and 3-neighbourhood subgraph size: 23.5, 59.4, and 92.6, respectively. Corresponding standard deviations were 22, 32 and 22.

The taillard problems used were the os-taillard-4 problems. Os-taillard-4 problems have 16 variables with domain size ranging from 99 to 187 values. Each variable is part of two cliques of size four that represent precedence and resource constraints. All constraints are disjunctive constraints of the form

$$X_i + k_i \leqslant X_j \vee X_j + k_j \leqslant X_i.$$

There are three subgroups of problems: the 4-95 problems have no solutions, while 4-100 and 4-105 problems have solutions. Since problems in the latter subgroup are easily solved they were not used for testing. 1-neighbourhood graph sizes were always 7; for 2- and 3- they were 16. (Obviously, there is no advantage to using 2- or 3-NSAC algorithms on these problems instead of SAC; nonetheless, these problems proved useful for comparing different k-NSAC algorithms.)

Algorithms were also tested on two kinds of randomly generated problems with intensional constraints. The first type ("relop" problems) had binary constraints based on relational operators, e.g. $X_i \geqslant X_j$ or $X_i = X_j$. The second type ("distance" problems) had absolute distance constraints of the form $|X_i - X_j| \otimes k$, where $\otimes$ is a relational operator. For both types, two different problem sets were used that had either sparse or moderately dense constraint graphs. For the sparse graphs, the mean size for 2- and 3-neighbourhood subgraphs was less than the number of variables, while for the dense graphs the 2-neighbourhood subgraph mean was nearly as large as the problem size, and the 3-neighbourhood subgraphs always included the entire problem.

Dense relop problems had 100 variables, domain size 20, graph density 0.26, with $\geqslant$ and $\neq$ constraints in equal proportions. Dense distance problems had 50 variables, domain size 20, graph density 0.23, while the distance constant k had an expected value of 3 and deviations from this occurred with diminishing probability in either direction. The relational operator was always $>$. All relop problems had solutions, while the distance problems included satisfiable and unsatisfiable instances.

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

For both the sparse relop and sparse distance problems, the same density was used as with the homogeneous random problems (0.05), which ensured that both the 2- and 3-neighbourhood subgraphs were smaller than the entire problem. (Since the statistics for subgraph sizes were therefore proportional to those given earlier, they are omitted.) Sparse relop problems had 200 variables, domain size 4 and $\neq$ and $\geqslant$ constraints in proportions 0.7 and 0.3, respectively. None of these problems had solutions. (For relop problems with these or similar parameter values, problems with solutions were always trivially easy.) Sparse distance problems had 150 variables and domain size 12. Constraints were based on $\neq$, $\geqslant$ or $=$ operators in the proportions 0.6, 0.3 and 0.1, respectively. For each constraint, the expected value for the distance constant k was 4, but it could be 3 or 5, 2 or 6, etc. with diminishing likelihood. These problems all had solutions.

Table 2 shows the mean number of values deleted for sets of problems with solutions. In five of the six problem classes, NSAC deleted more values than AC, and in three cases the difference was considerable. In each of these three cases, the difference between NSAC and 2-NSAC was also large. In two cases where the 3-neighbourhood graphs were distinctly different in size from the full constraint graph (RLFAP and sparse distance problems), there was a further increase in number of values deleted when 3-NSAC was established, and little further increase when full SAC was carried out. (Recall that for the scheduling and dense relop and distance problems, 3-NSAC was very similar or identical to full SAC, and for the scheduling problems this was also true for 2-NSAC.)

Two problem sets had no solutions: the os-taillard-95 problems and the sparse relop problems. For these scheduling problems, AC alone was unable to prove any of the problems unsatisfiable, while NSAC was able to prove this for 9 of the 10 problems. (The tenth problem could not be proven unsatisfiable even with SAC.) For these relop problems, neither AC nor NSAC was able to prove unsatisfiability, 2-NSAC proved 13 problems unsatisfiable, and these were also the only problems proved unsatisfiable by 3-NSAC and SAC.

Table 2
Effectiveness of different forms of NSAC on benchmark and random structured problems (values deleted during preprocessing)

Probs	#	AC	NSAC	2NSAC	3NSAC	SAC
				benchmark		
RLFAP-graph	4	279	778	989	1,021	1,038
Blackhole	20	280	280	280	280	280
Os-taillard-100	10	117	979	1,666	1,666	1,666
				random structured		
Relop-dense	100	0	556	761	761	761
Distance-dense	55	8.2	12.2	14.3	14.3	14.3
Distance-sparse	100	215	238	272	333	337

Notes: # is number of problems in each set. Entries are number of values deleted per problem. (All problems had solutions.)

Tables 3 and 4 give run times for different SAC algorithms and k-NSAC algorithms derived from them. The results show quite clearly that k-NSAC algorithms based on heavy-weight SAC algorithms perform poorly when compared with those based on light-weight SAC algorithms. In fact, there is no problem class where this is not true. In addition, when there is no difference between the expected size of the k-neighbourhood and the number of variables in the problem, performance of light-weight k-NSAC algorithms is comparable to their SAC counterparts. In contrast, under these conditions k-NSAC algorithms based on heavy-weight strategies give much worse performance than the corresponding SAC algorithms. On a minor note, it was found that for a few problem sets the modified form of NSAC-SDS was much better than the more straightforward version, although for other problem sets there was little difference.

Collectively, the results for structured problems show that for many kinds of problem, any level of neighbourhood singleton arc consistency deletes considerably more values than AC. As with random problems there is usually a diminishing rate of return for enforcing higher levels of singleton arc consistency. Typically, there is a clear increase in effectiveness on going from 1-NSAC to 2-NSAC. More rarely, there is also a definite improvement with 3-NSAC compared to 2-NSAC. Again, in no case is there a clear benefit from using full SAC as opposed to some form of NSAC.

At the same time, the difference between 1-NSACQ and 2-NSACQ with respect to efficiency (mean runtime) is also usually marked. And the difference between 2- and 3-NSAC is even greater. In most cases, therefore, if an improvement in effectiveness over AC is desired, then the best choice would appear to be either an NSAC or a 2-NSAC algorithm.

So far, the effects on subsequent search have not been emphasized. Since any form of SAC-based reasoning is expensive, it will only pay off if there is a lengthy search phase that can be reduced or avoided. With NP-complete problems, search time can usually

Table 3
Preprocessing runtimes for different SAC and NSAC algorithms with benchmark problems

Probs	AC	NSAC	2NSAC	3NSAC	SAC
			sac1-based		
RLFAP-graph	1	745	1,377	3,767	5,780
Blackhole	1	29	35	46	49
Taillard-4-95	0.2	112	163	157	160
Taillard-4-100	0.2	175	470	455	461
			sacq-based		
RLFAP-graph	–	418	984	2,572	4,047
Blackhole	–	29	34	45	47
Taillard-4-95	–	20	71	72	75
Taillard-4-100	–	64	292	288	301
			sac3-based		
RLFAP-graph	–	909	2,210	12,347	6,183
Blackhole	–	114	160	406	32
Taillard-4-95	–	32	147	211	176
Taillard-4-100	–	85	720	1,063	523
			sac-sds-based		
RLFAP-graph	–	*	*	*	*
Blackhole	–	63	101	622	44
Taillard-4-95	–	314	373	380	124
Taillard-4-100	–	339	3,699	4,133	341
			sac-sdsm-based		
RLFAP-graph	–	3,951	23,274	*	–
Blackhole	–	59	102	428	–
Taillard-4-95	–	141	327	617	–
Taillard-4-100	–	170	2,828	5,924	–

Notes: Same runs as in Table 2. Entries are times in s. "*" indicates runs omitted or aborted after several days. "–" indicates redundant entries in the table.

be increased dramatically by increasing problem size. With this in mind, sets of problems with the same characteristics as those used in the previous experiments were generated and tested. There were three types of problem: (i) relop problems with 300 variables, domains of size 5 and a density of 0.05, the same as with the smaller sparse problems; also as before, constraints were based on $\neq$ and $\geqslant$ relations, in the ratio of 70 : 30, (ii) distance problems with 300 variables, domains of size 13, density of 0.05, and constraints based on $\neq$, $\geqslant$ and $=$ relations in the proportions 0.6, 0.35, 0.05, with a constant k value of 4, (iii) os-taillard-5-95 problems; these have 25 variables and larger domains than the taillard-4 problems; otherwise their structure is identical. There were 50 problems in sets (i) and (ii), and ten problems in set (iii). In each set all problems were unsatisfiable. These problems were tested with k-NSAC algorithms based on SACQ, as well as SACQ itself, since these algorithms gave good results overall in previous experiments. Scheduling problems were tested using a one million node limit on search. In all cases search was done with MAC-3 and the domain/weighted-degree variable ordering heuristic [5].

Results are shown in Table 5. Clearly, for larger problems of widely varying type, some form of SAC-based preprocessing does allow greater efficiency overall when compared to AC. In addition, it was typically the case that establishing consistency equal to or greater than 2-NSAC gave greater improvement than establishing NSAC.[3]

[3]The greater search time for SAC for distance problems was due to a timing vagary of unknown origin. (A corresponding difference occurred for preprocessing, but because most problems could be proven unsatisfiable quickly, the difference in means was less.) New runs of 3-NSAC and SAC on these problems gave search times of 566 and 596 s, respectively, which seems within the margin of error.

Table 4
Preprocessing runtimes for different SAC and NSAC algorithms with random structured problems

Probs	AC	NSAC	2NSAC	3NSAC	SAC
			sac1-based		
Relop-dense	0.1	56	391	356	388
Relop-sparse	0.0	5.3	8.4	15.8	15.8
Distance-dense	0.0	1.9	9.8	10.1	10.0
Distance-sparse	0.1	3.8	9.5	29.6	36.9
			sacq-based		
Relop-dense	–	42	418	336	429
Relop-sparse	–	4.1	13.7	14.5	13.1
Distance-dense	–	1.4	11.1	8.5	8.6
Distance-sparse	–	2.9	9.9	29.5	38.3
			sac3-based		
Relop-dense	–	63	1,184	1,289	287
Relop-sparse	–	5.8	15.1	82.6	23.2
Distance-dense	–	2.2	34.8	43.4	13.4
Distance-sparse	–	4.7	18.9	102.2	39.9
			sac-sds-based		
Relop-dense	–	170	434	677	230
Relop-sparse	–	9.4	26.1	82.2	26.9
Distance-dense	–	2.1	25.6	40.6	7.2
Distance-sparse	–	9.2	33.2	106.8	56.5
			sac-sdsm-based		
Relop-dense	–	69	550	754	–
Relop-sparse	–	5.0	18.7	110.5	–
Distance-dense	–	2.0	25.4	38.8	–
Distance-sparse	–	5.5	24.3	129.9	–

Notes: Same runs as in Table 2. Entries are times in s. "–" indicates redundant entries in the table.

Table 5
Search nodes and run times for k-NSAC and SAC on larger problems

Prob & measure	AC	NSACQ	2NSACQ	3NSACQ	SACQ
Relop-nodes	89,036	59,355	48,746	47,943	46,943
Relop-preproc	0.1	58	76	108	91
Relop-search	2,606	1,778	1,535	1,644	1,433
Distance-nodes	23,005	14,407	9,682	9,158	9,150
Distance-preproc	0.2	117	271	224	248
Distance-search	1,778	944	749	742	955
Os-5-95-nodes	>410 K	>164 K	48,420	–	–
Os-5-95-preproc	0.5	1,065	5,119	–	–
Os-5-95-search	>20,756	>5,404	8,709	–	–

Notes: Means for different problem sets. Preprocessing and search times in s. ">" indicates that the node-limit was reached for at least one problem. "–" indicates conditions not tested, since they are equivalent to 2-NSAC.

Four of the ten scheduling problems could not be solved with AC alone within the specified limit. One of these could not be solved following NSAC either, but all were solved when search followed 2-NSAC. This shows that marked differences can occur when AC versus SAC-based strategies are used for preprocessing on problems of practical importance. It also shows that a k-neighbourhood SAC algorithm with $k > 1$ can outperform the basic NSAC algorithm on some scheduling problems.

6. Extension to non-binary constraints

Neighbourhood SAC can be extended to n-ary constraints in a way that, in some cases at least, is straightforward to deal with in practice. As far as the author is aware, this is not as easily done with other forms of higher-order consistency such as maxRPC [1].

However, there are some difficulties in defining NSAC under these conditions so as to specify a form of neighbourhood SAC commensurate with the binary case. Ultimately, this suggests that an extension like this can be done in various ways. Here, one approach is presented that is consistent with the variable-oriented approach to neighbourhoods taken in this work.

To maintain the idea of a k-neighbourhood in the form already defined, domain reduction via constraints must be restricted to the domains of neighbouring variables. This implies that there can be non-binary constraints which contain non-neighbouring variables whose domains should *not* be reduced during a bout of neighbourhood consistency checking. To cover such cases, we can revise Definition 2 slightly because $X_N \cup X_i$ no longer includes all the variables in the union of the scopes of C_N. This can be handled with a single change in the original definition, indicated by italics.

Definition 2′. For problems with non-binary constraints, 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 $P_N = (X_N \cup \{X_i\}, C_N)$ is arc consistent, where X_N is the neighbourhood of X_i and C_N is the set of all constraints whose scope *includes at least two members of the set* $X_N \cup \{X_i\}$.

Note that under this definition any non-binary constraint that includes the focal variable X_i will also include *all* of the other variables in that constraint, since by definition all are in the neighbourhood of X_i. It is only when non-binary constraints include two or more neighbours of X_i without including X_i itself that there is a difference from the binary case. This is shown in Fig. 8.

From this figure it should be clear that the present definition of the neighbourhood subgraph is a reasonable extension of the binary case. In the binary case, possible reductions in the domain of a non-neighbouring variable adjacent to one in the neighbourhood are ignored during neighbourhood SAC processing because this is not a case of two variables in the neighbourhood constraining each other. Similarly,

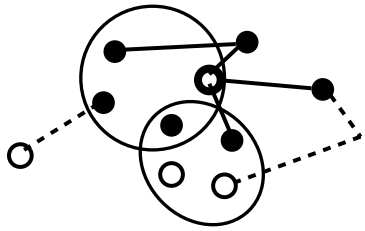

Fig. 8. A neighbourhood subgraph with binary and non-binary constraints. The focal variable is the circle with the thick outline, neighbouring variables are filled circles, non-neighbours outline circles. Binary constraints are shown as lines between the small circles, non-binary as a large outline circle or ellipse. Two constraints excluded from the neighbourhood subgraph are shown as dashed lines.

in the n-ary case we do not reduce domains of variables with respect to a non-binary constraint when these variables are not in the neighbourhood of the focal variable. The difference is that in the binary case we can ignore the constraint in question altogether. But in the non-binary case even if there are non-neighbourhood variables whose domains should not be reduced, there still may be neighbourhood variables that constrain each other. One way to deal with this is to project the constraint onto the set of neighbourhood variables.

Another option is to simply include any constraint involving two or more neighbourhood variables. This is an option that perhaps should be explored. However, in terms of the present definition of a k-neighbourhood, it leads to conceptual difficulties, since variables that are not in the k-neighbourhood will now be treated as if they were. Specifically, if a wipeout occurs in the domain of one of these variables, then the singleton value will be deleted. Although this is may produce a valid kind of consistency, it is not clear that the resulting procedure should be called "neighbourhood SAC".

But if the constraint cannot be readily decomposed, which may happen with certain global constraints depending on how they are implemented, then it may be necessary to follow the rule that whenever two neighbouring variables are within the scope of some n-ary constraint ($n > 2$), then this constraint will be included in the subgraph. (As noted above, this idea is left for future research.)

On the other hand, at this point it is not clear that there are any cases where the present approach cannot be applied in principle. Consider, for example, the element constraint, $element(y, z, x_1, \ldots, x_n)$, where y is an integer variable. The constraint stipulates that z must be equal to the yth variable in the x_i ordering, i.e. $z = x_y$. Now, in the context of establishing NSAC, one could have various scenarios, which can be classified

into five groups: (i) z and y are in the neighbourhood, while no x_i is, (ii) some subset of x_i's are in the neighbourhood, but z and y are not, (iii) and (iv) either y or z together with a subset of x_i's are in the neighbourhood, (v) both y or z and a subset of x_i's are in the neighbourhood. In each case, a fairly straightforward test can be made. For example, in case (i), values in the domain of z can be tested against x_i domains associated with current values of y (if none of the latter contain a given z value, then it can be deleted from the domain of z), and values in the domain of y can be tested by checking associated x_i domains against the z domain (if for a given y value, none of the z values are in the domain of x_y, then the value of y can be deleted).

Another example is the max constraint, max(y, $x_1, \ldots, x_k$), which specifies that y must equal the maximum value assigned to any x_i, $1 \leqslant i \leqslant k$. Now, suppose y and a set of one or more x_i are in the neighbourhood of the focal variable. Call the latter subset S' and the full set of x_i S. In this case, any value of y that is greater than any maximum or less than any minimum value of the variables in S can be deleted. Conversely, any value in any $x_i \in S'$ greater than the maximum value of y can be deleted; and values less than the minimum value of y can be deleted if there are no values equal to any value of y in any domain of $x_i \notin S'$. Similar reasoning can be applied in cases where a subset S' is in the neighbourhood, while y is not.

In all such cases, it is possible to use an STR-like approach, by rebuilding domains with values for which a viable tuple can be found. This approach was used with the sum constraint in the context of primes problems; the results are found in the following experimental section.

For table constraints, the NSACQ algorithm can be extended in a straightforward way to handle the non-binary case. In this case a GAC algorithm is used instead of a binary AC algorithm, and non-binary constraints are projected onto neighbouring variables if these form a proper subset of the scope. In the code, which uses an STR procedure [10], this could be accomplished with an additional check that a given domain was in the k-neighbourhood subgraph. For other types of constraint network, any method that ensures the same kind of GAC restriction can be used to handle such cases.

The next proposition shows that extensions like these can achieve the level of consistency required.

Proposition 5. *The NSACQ algorithm, with extensions to handle projections of neighbourhood variables onto non-binary constraints, achieves neighbourhood singleton arc consistency as defined in Definition 2′ for any CSP with non-binary constraints.*

Proof. As in the binary case, the extended form of NSACQ tests every value initially, and all neighbouring variables are put on the queue following removal of a singleton value. In addition, for each singleton value NSAC is carried out in accordance with Definition 2′. In the special case of non-binary constraints that include variables outside the neighbourhood, the proper level of consistency is achieved if, for every value a in the domain of a neighbourhood variable that is in the scope of this constraint, there is a constraint tuple whose projection onto the neighbourhood variables that belong to this constraint includes a. This is precisely what the restricted GAC procedure accomplishes. As in the binary case, any value in any variable outside the neighbourhood will not have become neighbourhood inconsistent by virtue of removal of a singleton value, since the neighbourhood of that variable is unchanged. If it does eventually become inconsistent due to an earlier deletion, this will be because a neighbouring domain was changed, and in this case the variable in question will have been put back on the queue. □

Other NSAC algorithms can be shown to achieve the same level of consistency for non-binary problems, but for brevity these arguments are not included in this paper. Naturally, the concept of full SAC can be carried over to the non-binary case without revision.

7. Experimental tests of k-NSAC and SAC algorithms on problems with non-binary constraints

Tests were made with GAC-based versions of the SACQ and k-NSACQ algorithms, using homogeneous random and structured benchmark problems.

7.1. Random problems

Random CSPs were generated by a program written by the author in which arities, their proportions, and their satisfiabilities could be specified independently, along with the mean degree of a variable. The program uses a probability-of-inclusion strategy to decide on the inclusion of each tuple in each constraint. For the present problems, the number of tuples in a con-

straint was precisely the expected value, i.e. this number was equal to the user-specified 'probability' value times the total number of possible tuples. For example, if the user specified a probability of inclusion of 0.3 and the total number of tuples was 10^4, the number of valid tuples was always 3,000. Tuple sets with this property were derived by generating tuple sets repeatedly until the desired number was obtained (i.e. by random generation with filtering). In all cases the constraint graph was connected.

Two kinds of random problem were used in these tests. The first had constraints of arities 2, 3, 4 and 5 in the following proportions: 0.5, 0.3, 0.15 and 0.05. The number of variables was 100, and the mean degree was either 3 or 3.5. (These formed two separate series of problems.) The domain size was always eight. For each problem, the same tightness value was used across all arities. Tightness was varied across problem sets to produce a series. Specific tightness values were 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.725, 0.75, 0.775, 0.8 and 0.9. For each tightness value 50 problems were generated and tested. Problems with the tightest constraints had no solutions, while for tightness < 0.7 all problems had solutions. Problems with solutions occurred at higher tightnesses when the mean degree was 3 (2, 13, 47 and 49 for tightnesses 0.8, 0.775, 0.75 and 0.725) than when it was 3.5 (0, 0, 2, 43 for the same tightnesses).

Problems of the second kind had 100 variables, domain size 8 and mean degree 5. Ninety-three percent of the constraints were binary, and seven percent were quarternary (4-ary). These specifications gave problems with 234 binary and seven 4-ary constraints. For this series, tightness for binary constraints was fixed at 0.52, while the tightness of the 4-ary constraints was varied in different problem sets from 0.1 to 0.9 in steps of 0.1. Again, for each combination of parameter values 50 problems were generated and tested.

Mean neighbourhood sizes were 6 ($k = 1$), 27 ($k = 2$) and 68 ($k = 3$) for the 100 variable problems with arities 2–5 and mean degree = 3. Standard deviations were either half ($k = 1, 2$) or one quarter ($k = 3$) of these values. With mean degree = 3.5, the corresponding means were 7, 34, and 80. For binary/4-ary problems, means were 6.5, 32, and 80.

Figure 9 shows preprocessing times for algorithms that establish different levels of neighbourhood SAC consistency, as well as full SAC. As with random binary problems, 2-NSAC does not result in a large increase in runtime, while 3-NSAC and SAC incur much larger overheads. (Of course, all these algorithms are much slower than AC, which always gave a mean $\leqslant$ 0.8 s.) The pronounced asymmetry in the curves is an exaggeration of the asymmetry observed with binary problems (cf. Figs 5–7), and is due to the fact that here GAC is called repeatedly rather than AC. In addition, the looseness of the constraints means that STR is less efficient.

Table 6 shows the number of problems proved unsatisfiable when each level of consistency was established for the first problem series. For these problems, there were cases where establishing a higher level of consistency proved more problems unsatisfiable up through 3-NSAC. In fact, for these problems 3-NSAC was suf-

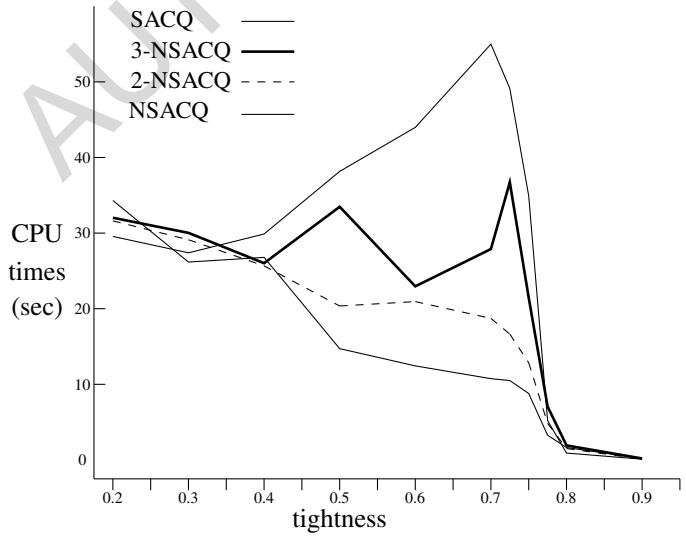

Fig. 9. Runtimes for full SACQ and k-NSACQ algorithms on random problems with constraints of arity 2–5 (mean degree 3). (Highest curve is for SACQ, lowest for NSACQ.)

Table 6
Random n-ary problems proved unsatisfiable by preprocessing algorithms (arity 2–5 problems)

Algorithm	Problem tightness				
	0.725	0.75	0.775	0.80	0.90
SAC	1	3	37	48	50
3-NSAC	1	3	37	48	50
2-NSAC	1	3	33	47	50
NSAC	1	1	28	41	50
AC	1	1	26	37	50
Total unsat.	1	3	37	48	50

Notes: 100-variable problems with arities 2–5, mean degree 3. Number of problems in each case is 50. Total unsatisfiable is shown in last row. (There were no unsatisfiable problems with tightness $\leqslant 0.70$.)

Table 7
Random n-ary problems proved unsatisfiable by preprocessing algorithms (arity 2, 4 problems)

Algorithm	Problem tightness								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
SAC	1	5	1	3	2	2	4	5	9
3-NSAC	1	5	1	3	2	2	4	4	9
2-NSAC	0	0	0	0	1	1	0	0	0
NSAC	0	0	0	0	0	0	0	0	0
AC	0	0	0	0	0	0	0	0	0
Total unsat.	2	7	1	5	12	9	7	32	47

Notes: 100-variable problems with arities 2 and 4. Number of problems in each case is 50. Total unsatisfiable shown in last row.

ficient to prove all unsatisfiable problems unsatisfiable. Similar results were found for problems with mean degree 3.5. For example, for tightness 0.75, the number proved unsatisfiable was 16 for AC, 30 for NSAC, 40 for 2-NSAC, and 48 for 3-NSAC and SAC.

For these (degree 3) problems, at higher tightness levels, values deleted for problems with solutions also showed marked differences related to the level of consistency established. For example, for tightness 0.775, mean values deleted per problem with solutions was 201 for AC, 253 for NSAC, 330 for 2-NSAC, 364 for 3-NSAC, and 369 for SAC. For tightness < 0.70 almost no values were deleted and there was almost no distinction in this respect between AC and higher consistency levels.

Corresponding results for problems proved unsatisfiable by consistency preprocessing in the binary/4-ary series are shown in Table 7. Here, a striking result is that no problem could be proven unsatisfiable by NSAC and only two of 122 by 2-NSAC. In contrast, 3-NSAC proved 31 problems unsatisfiable, one less than full SAC. In most cases, unsatisfiability could not be detected even with full SAC.

Another difference from the previous problem series is that aside from problems proven unsatisfiable, relatively few values were deleted per problem, even by SAC. Thus, AC deleted an average of between 13 and 14 values for problems *not* proven unsatisfiable, in different problem sets, while the corresponding range was 29–38 for SAC. Hence, the chief benefit offered by higher levels of consistency checking was discovering unsatisfiable problems during preprocessing. The runtime trends were similar to those of the other n-ary series, greatly increasing for problems with looser constraints (e.g. from 20 ($t = 0.9$) to 138 ($t = 0.1$) s for SACQ). However, in this case there was no evidence of a complexity peak.

7.2. Structured problems

For tests with structured problems, two kinds of benchmark were used. All problems of both types had solutions. The first type were configuration problems used in previous work [8]. (These were originally obtained from a configuration problem website that can no longer be located.) These problems had 20 or 40 variables with constraints of arity 2–5. In addition to the original problem, esvs-config ($n = 20$), variations were constructed by adding constraints (esvs-config.h1 and .h2; $n = 20$), tightening some constraints (esvs-cfghard; $n = 20$) and doubling the problem whilst varying one binary constraint, which reduced the number of solutions drastically (esvs-cfgdouble; $n = 40$). For these problems, which have table constraints, the STR algorithm was used for filtering.

Problems of the second type were primes problems from the -10 and -15 series, obtained from the website at Université Artois. All problems had 100 variables. Domain sizes were 28 for the primes-10 problems and 48 for the primes-15 problems. The number of constraints in different problems varied between 20 and 80. Constraints in these problems all have the form:

$$k_1 X_i + k_2 X_j + \cdots + k_n X_p = K,$$

where K and all k_i are integer values. The number of terms in the sum and therefore the constraint arity varies within and across problems.

These problems are of interest for the present work because constraints of this type are not decomposable unless one uses auxiliary variables, so projecting onto a subset may be difficult. In the present experiments a simple filtering algorithm was used, which examined each variable in turn, and for each value systematically

examined values for the other variables until a full tuple was found that satisfied the constraint or all tuples had been examined. In the latter event the original value can be discarded. In the case where only a subset of the variables were k-neighbours of the focal variable, only the domains of variables in this subset were actually updated. Although a recursive version of this algorithm was written that could handle constraints of any arity, because of time constraints, only results for problems having constraints between arity 2 and 5 are included (2-1, 2-3 and 3-1 problems). This gives two sets of twelve problems.

For the configuration problems all neighbourhood subgraph sizes were well below values for n, the number of variables. For the original problem, means were 4.3, 5.9 and 7.5 for 1-, 2- and 3-neighbourhood subgraphs, respectively. For the cfg-double problems the corresponding means were 5.2, 11.1 and 18.6. Means for the other problems fell between these values. Standard deviations varied from 1.5 to 1.8 for 1-neighbourhood subgraphs; for 2- and 3-neighbourhood subgraphs they varied from 3 to 6.

Neighbourhood subgraph sizes were also different from n for primes problems. These varied greatly depending on the number of constraints and their arity. Thus, mean subgraph sizes were 1.8 ($k = 1$), 2.3 ($k = 2$) and 2.5 ($k = 3$) for the primes-xx-20-2-1 problem and 8.9, 44.6 and 83.9 for the primes-xx-80-2-3 problem. Corresponding means for other problems fell between these two extremes. In each case the standard deviation was an appreciable fraction of the mean, varying between $1/4$ and 1. Means and standard deviations were identical for primes-10 and primes-15 problems with the same arities and number of constraints.

Results for configuration problems are shown in Table 8. There were clear differences in effectiveness for the harder problems, especially between AC and 1-NSAC. 2-NSAC deleted a few more values than NSAC with very little difference in time, while higher consistencies gave no further benefit.

Results for primes-10 problems are shown in Table 9. (A very similar pattern of results was found for the primes-15 problems although corresponding values were greater both for times and deletions.) For all problems, NSAC was much more effective than AC, while no further deletions were obtained by ensuring higher levels of k-NSAC. A further finding was that the effect of differences in arity was much more pronounced for AC than for NSAC. Differences in cost, i.e. in runtime, depended on both the number of constraints in the problem and the arities. In some cases a large number of additional values could be deleted with almost no extra cost; however, in two cases (20-2-3 and 20-3-1 problems) the difference in cost was enormous.

8. Conclusions

Neighbourhood SAC algorithms expand the scope of SAC-based strategies by allowing more limited yet still well-defined forms of SAC to be achieved. The present work demonstrates that the basic idea of neighbourhood SAC can itself be extended. Thus, an entire series of local consistencies can be defined that has an obvious dominance ordering with respect to values deleted. This allows the user to choose among different levels of consistency using the same basic strategy (and algorithms) to balance the conflicting goals of effectiveness and efficiency. The experimental results show that benefits can be gained through such strategies. Moreover, the results show clearly that both maximum effectiveness *and* the balance between effectiveness and efficiency are associated with different levels of consistency for different problem types. Gaining a better understanding of these relations is an important task for future research.

Neighbourhood SAC algorithms have special properties that potentially enhance their usefulness. In the first place, these algorithms take advantage of the statistical properties of a constraint network vis-à-vis a

Table 8
Results for different forms of NSAC with configuration problems

Problem	AC		1-NSAC		2-NSAC		3-NSAC		SAC	
	t	del	t	del	t	del	t	del	t	del
Esvs-config	0.00	0	0.04	0	0.05	0	0.08	0	0.07	0
Esvs-config.h1	0.00	0	0.05	4	0.06	4	0.09	4	0.09	4
Esvs-config.h2	0.00	1	0.03	11	0.04	11	0.05	11	0.04	11
Esvs-cfghard	0.01	3	0.05	60	0.07	72	0.08	72	0.08	72
Esvs-cfgdouble	0.01	16	0.09	126	0.11	144	0.14	144	0.15	144

Notes: t is time (s). del is number of values deleted per problem.

Table 9
Results for different forms of NSAC(Q) with primes-10 problems

Problem	AC		1-NSAC		2-NSAC		3-NSAC		SAC	
	t	del	*t*	del	*t*	del	*t*	del	*t*	del
20-2-1	0.2	505	1.0	804	1.0	804	1.0	804	1.2	804
20-2-3	43.5	52	720.8	493	841.9	493	867.3	493	885.1	493
20-3-1	3.1	0	59.5	484	83.3	484	95.9	484	170.1	484
40-2-1	0.3	789	0.9	1,584	0.9	1,584	0.9	1,584	1.0	1,584
40-2-3	50.0	372	106.4	1,561	239.3	1,561	314.2	1,561	341.6	1,561
40-3-1	4.4	0	5.1	1,831	5.2	1,831	5.3	1,831	5.2	1,831
60-2-1	0.3	1,148	0.6	2,196	0.6	2,196	0.6	2,196	0.7	2,196
60-2-3	53.8	483	53.2	2,478	54.7	2,478	53.4	2,478	54.3	2,478
60-3-1	4.1	0	4.3	2,341	4.4	2,341	4.6	2,341	4.4	2,341
80-2-1	0.3	1,423	0.5	2,455	0.5	2,455	0.6	2,455	0.6	2,455
80-2-3	47.6	858	47.4	2,644	48.9	2,644	47.9	2,644	48.6	2,644
80-3-1	4.0	0	4.1	2,536	4.2	2,536	4.5	2,536	4.2	2,536

Notes: *t* is time (s). del is number of values deleted per problem.

singleton domain. Under these conditions, likelihood of support will be lowest for the neighbours of the focal variable, next-lowest for their neighbours, and so forth. For example, consider a problem with domain size 10 and tightness 0.5. If a domain is reduced to a single value, then the probability that it supports a value in a neighbouring domain is 0.5, and the expected number of AC-based deletions in the latter domain is five. The probability that none of these five values supports a given value in *its* neighbour is 0.5^5 or $\frac{1}{32}$. Consequently, relatively few values in the neighbours-of-neighbours part of the subgraph should be deleted. This argument is consistent with the experimental results; however, the latter showed that establishing 2-NSAC and even 3-NSAC can give definite benefit compared to NSAC for some problem classes. On the other hand, it seems unlikely that neighbourhood SAC will be useful for $k > 3$.

Another potential benefit of using neighbourhood SAC instead of full SAC is due to the fact that a k-NSAC algorithm will automatically vary the size of the subproblem that is taken into account depending on the degree of the core variables. Thus, for a focal variable with a high degree, a larger subproblem will be considered, other things being equal, than for a focal variable with a low degree. Since the former variable is potentially more significant with respect to basic problem difficulty and is more likely to have values that can be deleted, the overall effect is to balance the tradeoff between effectiveness and efficiency more adequately than with a full SAC algorithm. Moreover, the fact that this is inherent in the k-NSAC strategy means that this beneficial effect is achieved without adding special data structures and procedures, which would necessarily incur extra processing overhead.

This work has introduced a distinction between light-weight and heavy-weight SAC algorithms. Although this is only a rough empirical classification, it appears to highlight some important differences among these algorithms regarding the tradeoff between efficiency and effectiveness and the range of effective application of light-weight versus heavy-weight strategies.

In particular, before this work was done, one could presume that because some heavy-weight SAC algorithms outperform light-weight algorithms on many problems [3,9,11], the same strategies could be used to make better NSAC algorithms. The present experiments show that this is not true; in fact, heavy-weight algorithms are clearly not well-suited for establishing neighbourhood SAC in any of its forms.

For all problem classes tested, some level of k-NSAC was just as effective or nearly as effective as full SAC. At the same time, for algorithms based on the light-weight SAC strategy, there was often an improvement in efficiency. This means that choosing to achieve some level of NSAC may generally be preferable to achieving full SAC if one's basic concern is handling the tradeoff between effectiveness and efficiency.

The present work has also shown one way to extend neighbourhood SAC algorithms to handle problems with n-ary constraints. In this case there are issues of interpretation, which can lead to possible variants. The approach described here is conceptually sound,

and it can be used with table constraints in a straightforward way, as well as some other types of constraint. More work is needed to determine whether other variants of closure-based consistency are reasonable alternatives, especially when the projection of a constraint onto a subset of its variables is not straightforward.

Acknowledgements

The author thanks the anonymous reviewers, whose comments were of great help in preparing this paper.

References

[1] T. Balafoutis, A. Paparrizou, K. Stergiou and T. Walsh, New algorithms for max restricted path consistency, *Constraints* **16** (2011), 372–406.

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

[3] 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.

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

[5] F. Boussemart, F. Hemery, C. Lecoutre and L. Sais, Boosting systematic search by weighting constraints, in: *Proc. Sixteenth European Conference on Artificial Intelligence – ECAI'04*, IOS, 2004, pp. 146–150.

[6] 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, Morgan Kaufmann, 1997, pp. 412–417.

[7] E.C. Freuder, Dispensable instantiations in constraint satisfaction problems, in: *Tenth International Workshop on Constraint Modelling and Reformulation – ModRef 2011*, 2011.

[8] E.C. Freuder, R. Heffernan, N. Wilson and R.J. Wallace, Lexicographically-ordered constraint satisfaction problems, *Constraints* **15** (2010), 1–28.

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

[10] J.R. Ullmann, Partition search for non-binary constraint satisfaction, *Information Sciences* **177** (2007), 3639–3678.

[11] R.J. Wallace, SAC and neighbourhood SAC, *AI Communications* **28**(2) (2015), 345–364.