

# Light-Weight versus Heavy-Weight Algorithms for SAC and Neighbourhood SAC

**Richard J. Wallace**

Insight Centre for Data Analytics

Department of Computer Science, University College Cork, Cork, Ireland

richard.wallace@insight-centre.org

## Abstract

This paper compares different forms of singleton arc consistency (SAC) algorithms with respect to: (i) their relative efficiency as a function of the increase in problem size and/or difficulty, (ii) the effectiveness with which they can be transformed into algorithms for establishing neighbourhood singleton arc consistency (NSAC). In such comparisons, it was found useful to distinguish two classes of SAC algorithm with the terms “light-weight” and “heavy-weight”. The difference turns on the complexity of data structures required; this complexity is much greater for the heavy-weight than the light-weight algorithms. The present work shows that this difference is reflected in scalability. In general, light-weight algorithms scale better than heavy-weight algorithms on both random and structured problems, although in the latter case the heavy-weight algorithm SAC-3 is able to compensate for its extra overhead. In addition, it is shown that modifying heavy-weight algorithms to produce NSAC algorithms is problematic because the logic underlying their improvements, while suited to SAC, requires major revisions when carried over to neighbourhood SAC. As a result, even for problems where heavy-weight SAC algorithms are competitive with light-weight algorithms, the efficiency of the corresponding NSAC algorithms is seriously compromised. In contrast, light-weight SAC algorithms can be easily converted to efficient NSAC algorithms. These results show that heavy-weight algorithms for SAC cannot be considered as unconditional improvements on light-weight algorithms, and more importantly that more attention should be paid to the basic tradeoff between reducing the number of dominant operations (constraint checks) and minimizing extra overhead in local consistency algorithms.

## Introduction

Singleton arc consistency (SAC) is a powerful enhancement of arc consistency, the best-known type of local consistency algorithm. In this variant, each value associated with a variable is considered as a singleton domain, and arc consistency is carried out under this assumption. Under these conditions, failure in the form of a domain wipeout implies that there is

no solution containing this value; hence it can be discarded without affecting the solution set for the problem.

The original SAC algorithm proposed by (Debruyne and Bessière 1997) performs SAC on each value of the problem, and repeats this process until no value is deleted. Since this is the same strategy used by AC-1, the algorithm is referred to as SAC-1. (Bartak and Erben 2004) proposed a version of SAC (SAC-2) that uses support counts in the manner of AC-4. More recently, (Bessière and Debruyne 2005) developed SAC-SDS, which is based on an optimal-time SAC algorithm (SAC-Opt). SAC-Opt uses  $n \times d$  copies of the original problem, each altered to give a different singleton domain. SAC-SDS retains some multiple information in the form of domains and queues for updating. It thereby removes some of the redundancy in problem representation in SAC-Opt, while still avoiding some of the redundancy in effort of SAC-1. (Lecoutre and Cardon 2005) 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. More recently (Wallace 2015) introduced a SAC algorithm called SACQ, which replaces the repeat loop of SAC-1 with a queue update after each value deletion.

In working with these algorithms it became clear that a useful distinction can be made between “light-weight” and “heavy-weight” SAC (and NSAC) algorithms. Light-weight algorithms like SAC-1 and SACQ require only simple data structures and procedures. In contrast, heavy-weight algorithms like SAC-2, SAC-SDS, and SAC-3 require elaborate data structures and special procedures to maintain them. While the former are much easier to code (an important consideration in actual practice), they fail to achieve optimal or near-optimal performance in terms of constraint checks. On the other hand, heavy-weight algorithms are not only much more difficult to code correctly, they are also space-inefficient and they involve extra processing to keep the data structures up to date. These differences bear on a fundamental tradeoff in this field, between the reduction in the number of dominant operations, here constraint checks, and the overhead incurred in maintaining the complex data struc-

tures needed to effect this reduction.

Recently a new *form* of singleton arc consistency was defined, called neighbourhood singleton arc consistency (NSAC) (Wallace 2015). NSAC is a limited form of SAC in which only the subgraph based on the neighbourhood of the variable with the singleton domain is made arc consistent during the SAC phase. This form of singleton arc consistency, while dominated by SAC, is nearly as effective on many problems (with respect to values deleted and proving unsatisfiability) while requiring much less time. Three NSAC algorithms were proposed that establish this form of consistency throughout a problem. All of them are based on SAC-1 or SACQ. But since NSAC algorithms are variations of SAC procedures, this is another area where it is important to compare light-weight and heavy-weight algorithms.

In the present work, we compare light-weight and heavy-weight algorithms with respect to their scaling properties when problem size and/or difficulty increases. We also present results for NSAC versions of some of these algorithms; here we find that the strategies involved in heavy-weight SAC do not carry over readily to the NSAC context, so that adjustments must be made that compromise performance. This is in marked contrast to the situation with light-weight SAC algorithms.

The next section gives general background concepts and definitions and brief descriptions of the basic SAC algorithms. Section 3 recounts results of experiments to test scaling properties of SAC algorithms on four types of problem. Section 4 describes how SAC algorithms can be converted to NSAC algorithms and gives some experimental comparisons of the latter. Section 5 presents conclusions.

## Background

### General concepts

A constraint satisfaction problem (CSP) is defined 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  $C$ .

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 algorithms 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 can establish well-defined forms of local consistency in a problem. The most widely studied 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  $C_{ij}$  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 particular form

of AC in which the just-mentioned value  $a$ , for example, is considered the sole representative of the domain of  $X_i$  (Debruyne and Bessière 1997). (In the present paper,  $X_i$  is called the “focal variable”.) 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  $a$  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.)

The following definition is given in order to clarify the description of the neighbourhood SAC algorithms.

**Definition 1.** The *neighbourhood* of a variable  $X_i$  is the set  $X_N \in X$  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  $X_i$ .

Neighbourhood SAC 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  $P_N = (X_N \cup X_i, C_N)$  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.

### SAC Algorithms

SAC-1, SAC-2, SAC-SDS, and SAC-3 are well-known and have been described in detail elsewhere (see original papers as well as descriptions in (Wallace 2015)). SACQ is described in (Wallace 2015). So only brief descriptions are given here.

SAC-1, the original SAC algorithm (Debruyne and Bessière 1997), 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. In addition, if a value is deleted (because instantiating a variable to this value led to a wipeout), then after removal a full AC procedure is applied to the remaining problem before continuing on to the next singleton value.

The SAC-SDS algorithm (Bessière and Debruyne 2005; 2008) 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  $d$  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 that 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.

SAC-SDS also makes 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. 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.

The SAC-3 algorithm (Lecoutre and Cardon 2005) 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.

SACQ uses an AC-3 style of processing at the top-level instead of the AC-1 style procedure 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 values that are 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. Since this algorithm is less familiar than the others, pseudocode is shown in Figure 1.

In this work, both SAC (and NSAC) algorithms were preceded by a step in which arc consistency was established (shown in Figure 1), although this is not required to establish either property. 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 arc consistency algorithm.

All algorithms were coded in Common Lisp. Care was taken to develop efficient data structures, especially for the heavy-weight algorithms. In some cases this required itera-

```

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

```

Figure 1: Pseudocode for SACQ.

tions over a number of months. (For more details see (Wallace 2015).) In addition, during testing correctness of implementation was evaluated by cross-checking the number of values deleted for all algorithms on all problems when a problem was not proven unsatisfiable during preprocessing.

## Experimental Analysis

The chief point of the experiments was to determine how well each algorithm scales as a function of problem size and/or difficulty. Here we will mainly consider problem size.

To assess differences among algorithms across a range of problems, both randomly generated problems and benchmarks were used. (There is one limitation in that all are binary CSPs.) The randomly generated problems were of two types: (1) homogeneous random CSPs, where neither domain nor constraint elements are ordered, the probability of a constraint between two variables is the same for each variable pair, and the probability that a particular tuple is part of a given constraint's relation is the same for all tuples that might appear, (2) random relop problems, where domain values are ordered and the constraints are relational operators, such as not-equals or greater-than, while as with the homogeneous problems, the likelihood of a constraint between any two variables is the same for all variable pairs. Two kinds of benchmark problems were used: radio frequency assignment problems and open-shop scheduling problems.

## Experiments with random problems

The first experiment was done with homogeneous random problems. Problems had either 50, 75 or 100 variables. Density and tightness were chosen to ensure that each set of problems was in a critical complexity region. The values of the standard parameters for each problem set were  $\langle 50, 10, 0.21, 0.43 \rangle$ ,  $\langle 75, 15, 0.15, 0.57 \rangle$ , and  $\langle 100, 20, 0.05, 0.70 \rangle$ , where the series inside the brackets indicates number of variables, domain size (kept constant), constraint graph density, and constraint tightness (kept constant) in that order. Each problem set had fifty problems. These and later experiments were run in the XLispstat environment with a Unix OS on a Dell Poweredge 4600 machine

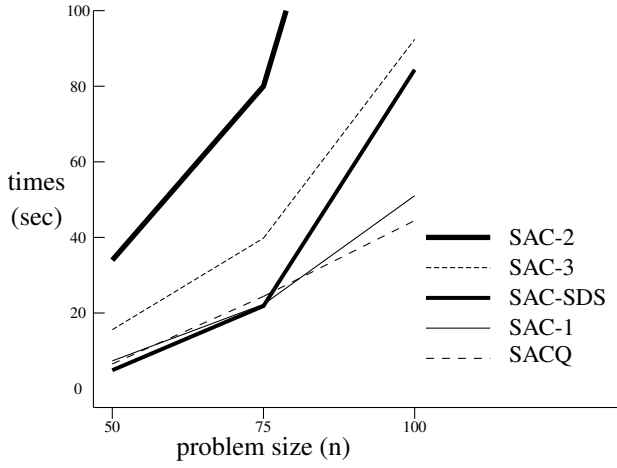


Figure 2: Mean runtimes for SAC algorithms on homogeneous random problems of increasing size. Last segment of SAC-2 curve only shown up to 100-sec limit.

(1.8 GHz). Also, in each experiment all algorithms were run sequentially, one immediately after the other, to minimize vagaries in timing that have been observed when similar runs are separated by days or weeks. The results of the first experiment are shown in Figure 2.

From the figure it can be seen that there is a definite divergence in efficiency in favour of the light-weight algorithms as problem size increases. The most spectacular increase in runtime is for SAC-2. In this case, the last point (for the 100-variable problems) could not be graphed without compressing the other curves, so it was omitted. (The value of the mean in this case was 402 sec.) For SAC-3 and SAC-SDS the increase was much less dramatic, but for the largest problems there was a marked divergence from the light-weight algorithms. For problems of this type, SAC-1 and SACQ had very similar average runtimes, although there is some indication of a divergence for the largest problems in favour of SACQ. (As expected, SAC-SDS showed a dramatic reduction in constraint checks, to about 50% of those generated by SAC-1.)

The second experiment was done with random relap problems. These problems were generated with equal proportions of greater-than-or-equal and not-equals constraints; the latter ensured that the problems were not tractable. Three sets of fifty problems were used with the following parameters: (i) 60-variable problems with domain size 15 and constraint graph density 0.32, (ii) 100-variable problems with domain size 20 and constraint graph density 0.27, (iii) 150-variable problems with domain size 20 and constraint graph density 0.21.

Results for this experiment are shown in Figure 3. Although it was possible to collect some data for SAC-2, runtimes were so much greater than for other algorithms that they are omitted from the graph. (For the 60-variable problems mean runtime was more than 3000 sec., while for the 100-variable problems was more than 30,000 sec.)

For the 60-variable problems runtimes were low for all

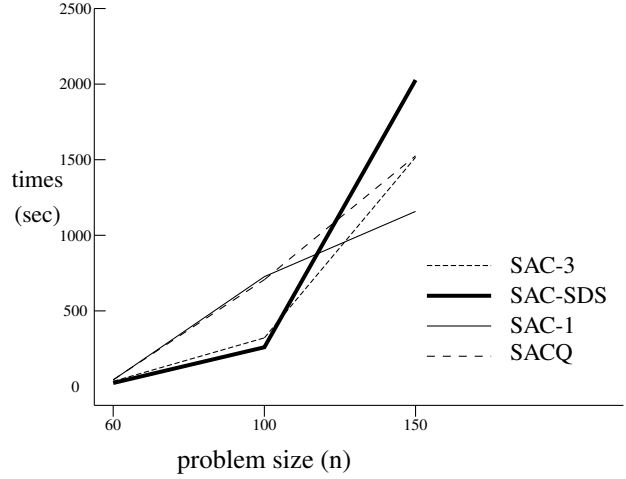


Figure 3: Runtimes for SAC algorithms on relap problems of increasing size.

other algorithms, although average time for SAC-SDS was half that for the light-weight algorithms (about 20 versus 40 seconds), while the mean runtime for SAC-3 was half-way between these values. For the 100-variable problems, both heavy-weight and both light-weight algorithms performed similarly, but the heavy-weight algorithms were more than twice as fast as the light-weight algorithms on average. (SAC-SDS was somewhat faster than SAC-3.) However, for the largest problems, this difference tended to reverse, so that SAC-1 was now the fastest algorithm on average and SAC-SDS the slowest. In this case the mean runtimes for SACQ and SAC-3 were almost the same. The difference between SAC-3's performance in this experiment (and later ones) and on experiments with homogeneous random problems is due to the fact that SAC-3 can take account of the redundancy in ordered problems, and this compensates for the extra overhead.

### Experiments with benchmark problems

Radio frequency assignment (RLFAP) problems were obtained from the site maintained at Université Artois<sup>1</sup>. These were the modified RLFAPs called graph problems. Of the seven problems in this set only the four with solutions were used. These will be designated Graph1, Graph2, Graph3 and Graph4. (The remaining problems are extremely easy so they are not useful for comparisons of this sort.)

RLFAP problems have domains composed of integer values, which include only a small subset of the values between the smallest and largest values. The constraints are "distance constraints" of the form  $|X_1 - X_2| > k$  or  $|X_1 - X_2| = k$ , with the meaning that the absolute difference between variables  $X_1$  and  $X_2$  must be greater than a constant  $k$  or equal to it, respectively.

The Graph1 and Graph3 problems have 200 variables; the others 400. Despite this, each successively numbered problem is more difficult than the previous ones. This is due to

<sup>1</sup><http://www.cril.univ-artois.fr/lecoutre/benchmarks.html>

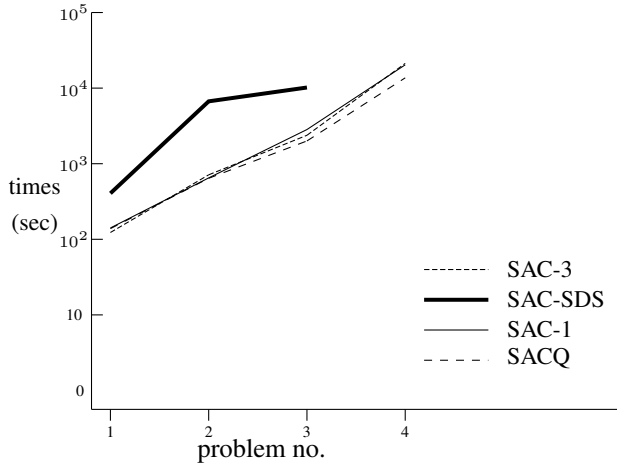


Figure 4: Runtimes for SAC algorithms on RLFAP problems of increasing size and/or difficulty. Missing data point SAC-SDS is due to inability to finish the run (time  $> 10^5$  sec). Note log scale on ordinate.

the fact that a few domains in the Graph1 and Graph2 problems are severely reduced in size, making them less difficult to process than Graph3 and Graph4.

Results of tests with these problems are shown in Figure 4. For these moderately large problems it was not possible to complete any runs with SAC-2. SAC-SDS was also highly inefficient, and this inefficiency increased as problem size and difficulty increased, to the point where the run with hardest problem could not be completed. Another point worth noting is that the inflection for SAC-SDS is different from the other algorithms; this undoubtedly reflects the fact that Problem 2 in this series is much larger than Problem 3 although it is basically easier to solve. In this case the space inefficiency of SAC-SDS is also reflected in the runtime.

SAC-3 as about as efficient as SAC-1 on these problems, but both are less efficient than SACQ. These differences become clear for difficult problems that are also large. Thus, runtimes for Problem No. 4 were 20,336, 21,244 and 13,664 sec for SAC-1, SAC-3 and SACQ, respectively. Since SAC-1 begins to diverge from SACQ on the most difficult problems, this indicates that here the queue-based strategy scales better than the repeat-loop strategy.

Scheduling problems were taken from two of the Taillard series (Taillard 1993). These were the taillard-4-100 and taillard-5-100 problems, where the time window is set to the best-known value. These problems have solutions, but since the time window restrictions are tight, they are relatively difficult to solve. For these problems, constraints prevent two operations that require the same resource from overlapping; specifically, they are disjunctive relations of the form,  $X_i + k_1 \leq X_j \vee X_j + k_2 \leq X_i$ . The os-taillard-4 problems have 16 variables with 100-200 values per domain; os-taillard-5 problems have 25 variables with 200-300 values per domain.

Results of tests with these problems are shown in Figure 5. (SAC-2 is not included in the graph since the mean

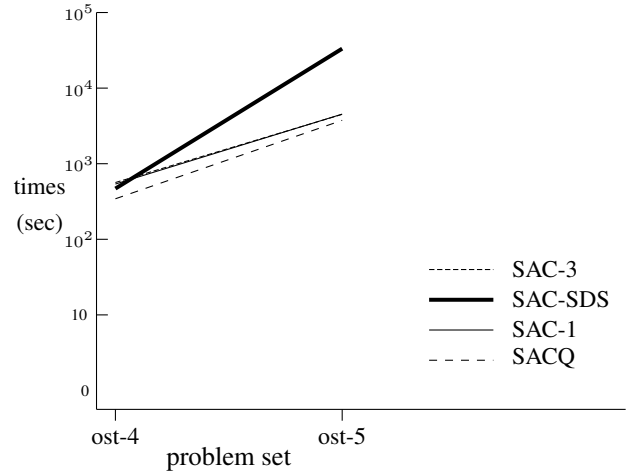


Figure 5: Runtimes for SAC algorithms on os-taillard problems of increasing size. Note log scale on ordinate.

runtime with os-taillard-4 problems was over  $10^4$  sec; tests with os-taillard-5 problems were not attempted.) Times for the smaller os-taillard-4-100 problems were similar (mean runtimes were 538, 344, 564 and 469 seconds for SAC-1, SACQ, SAC-3 and SAC-SDS, respectively). However, with larger problems, SAC-SDS was much slower than the other algorithms; in this case SAC-1 and SAC-3 had similar runtimes while SACQ was somewhat faster.

### Converting SAC to NSAC Algorithms

Converting either light-weight SAC algorithm to an algorithm for establishing neighbourhood singleton arc consistency is straightforward. In both cases all that is necessary is that SAC-based consistency testing be replaced with a SAC-based test limited to the neighbourhood of the focal variable. For NSACQ this means replacing the line (in Figure 1)

If  $AC(P')$  leads to wipeout

with the line

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

and the line

Update Q to include all variables of P

with

Update Q to include all neighbours of  $X_i$

On the other hand, converting heavy-weight algorithms requires some significant changes. (In this case no attempt was made to convert SAC-2 because it proved to be so inefficient on the experiments reported in the last section.)

To convert SAC-SDS to NSAC-SDS it was necessary to eliminate the copy-queues and to substitute a different means of updating the subproblems. The reason is that performing AC on the basis of copy-queues (which include all variables whose domains have been reduced since the last test of that subproblem) takes one beyond the neighbourhood subgraph. This may cause the singleton value to be discarded even if the subgraph is consistent. Instead, after

deleting values from a subproblem and putting the subproblem back on the pending list, SAC-based arc consistency was re-established for the neighbourhood subgraph. In doing this care was taken to test all arcs in the neighbourhood subgraph since it could not be assumed that simply testing those adjacent to the focal variable would eliminate all possible values. One can make this assumption with the full SAC algorithm, since neighbours of *all* affected variables are checked.

For NSAC-3, a similar procedure for testing all arcs in a subgraph had to be followed for the same reason. That is, as variable-value pairs are successively added to an 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, if one restricts the initial arc consistency queue to arcs involving the focal variable, this form of inconsistency is not detected. In contrast, in the SAC-3 algorithm the entire problem is made arc consistent at each step, so this condition doesn't occur.

**Table 1. Results for Different Forms of NSAC with Random and Structured Problems**

probs	NSAC-1	NSACQ	NSAC-SDS	NSAC-3
random	4	2	3	58
rlfap	745	418	3951	9098
relop	56	42	69	63
sched	511	200	719	260

Notes. Mean runtimes in sec.

Versions of NSAC based on each SAC algorithm except SAC-2 were tested with the four types of problem on which SAC algorithms had been tested. Each problem set was one of those tested earlier: random problems were the fifty 100-variable problems, relap problems were the fifty 100-variable problems, RLFAPs were the four rlfap-graph problems, and scheduling problems were the os-taillard-4-100 set. Results are shown in Table 1.

From these results, it is clear that NSAC algorithms based on heavy-weight SAC methods are generally inferior to those based on light-weight algorithms, sometimes dramatically so. This is true even for problems for which the heavy-weight SAC algorithms are either much more efficient than light-weight algorithms (relop problems), or in the case of SAC-3, equally efficient (RLFAPs)

## Conclusions

This work introduces a distinction between light-weight and heavy-weight SAC algorithms. That this distinction may be of critical importance is shown by results of tests with problems of increasing size and/or difficulty. For the most part, heavy-weight SAC algorithms do not scale as well as light-weight SAC algorithms. This highlights what may be the basic problem in this field, the tradeoff between reducing the dominant operation of constraint checks and contending with increased overhead due to the more complex data structures that are necessary to effect this reduction.

At the same time, it must be recognized that there is a fairly consistent ordering of efficiency among heavy-weight SAC algorithms, especially when dealing with structured

problems. With such problems SAC-3 is always the most efficient, and except for some very large problems, its efficiency often matches or exceeds that of the light-weight algorithms. Nonetheless, the limited data collected so far suggests that scaling problems will appear whenever problem size exceeds some mid-sized range, roughly from 150 to 200 variables. Next is SAC-SDS, which is always shows loss of efficiency relative to light-weight algorithms as problems scale up. Finally, SAC-2 is always much slower than SAC-SDS or any other SAC algorithm, and this difference can be observed even with small problems.

In this work it was also found that in addition to difficulties involving overall efficiency, heavy-weight algorithms are not well-suited for establishing neighbourhood singleton arc consistency. This is the case even with problems where the original SAC algorithm is much better than any of its light-weight counterparts.

All this shows that we cannot consider the developments in this area of algorithmics as a simple 'monotonic' progression to better and better algorithms. Instead, sooner or later we will have to face the fact that the tradeoffs entailed by such developments make this area problematic with respect to algorithmic efficiency.

## References

- Bartak, R., and Erben, R. 2004. A new algorithm for singleton arc consistency. In *Proc. 17th Internat. FLAIRS Conference*. Vol. 1, 257–262.
- Bessière, C., and Debruyne, R. 2005. Optimal and suboptimal singleton arc consistency algorithms. In *Proc. 19th Internat. Joint Conf. on Artif. Intell. – IJCAI'05*, 54–59.
- Bessière, C., and Debruyne, R. 2008. Theoretical analysis of singleton arc consistency and its extensions. *Artificial Intelligence* 172:29–41.
- Debruyne, R., and Bessière, C. 1997. Some practicable filtering techniques for the constraint satisfaction problem. In *Proc. 15th Internat. Joint Conf. on Artif. Intell. – IJCAI'97*. Vol. 1, 412–417. Morgan Kaufmann.
- Lecoutre, C., and Cardon, S. 2005. A greedy approach to establish singleton arc consistency. In *Proc. 19th Internat. Joint Conf. on Artif. Intell. – IJCAI'05*, 199–204. Professional Book Center.
- Taillard, E. 1993. Benchmarks for basic scheduling problems. *European Journal of Operational Research* 64:278–285.
- Wallace, R. J. 2015. SAC and neighbourhood SAC. *AI Communications* 28(2):345–364.