Interleaving Levels of Consistency Enforcement for Singleton Arc Consistency in CSPs, with a New Best (N)SAC Algorithm

Richard J. Wallace(✉)

Insight Centre for Data Analytics, Department of Computer Science, University College Cork, Cork, Ireland
richard.wallace@insight-centre.org

Abstract. A basic technique used in algorithms for constraint satisfaction problems (CSPs) is removing values that are locally inconsistent, since they cannot form part of a globally consistent solution. The best-known algorithms of this type establish arc consistency (AC), where every value has support in neighbouring domains. Here, we consider algorithms that use AC repeatedly under severe local assumptions to achieve higher overall levels of consistency. These algorithms establish (neighbourhood) singleton arc consistency ((N)SAC). Most of these use simple AC interleaved with the basic (N)SAC procedure. To date, however, this strategy of interleaving weaker and stronger forms of reasoning has not received much attention in and of itself. Moreover, one of the best (N)SAC algorithms (called (N)SACQ) does not use this method. This paper investigates the effects of interleaving and presents new methods based on this idea. We show that different kinds of problems vary greatly in their amenability to AC interleaving; while in most cases it is beneficial, with some algorithms and problem types it can be harmful. More significantly, when this feature is added to (N)SACQ algorithms, the latter's superiority to other (N)SAC algorithms becomes more consistent and decisive. We also consider an AC-4 based approach to interleaving as well as interleaving with stronger methods than AC.

1 Introduction

The constraint satisfaction problem is a basic form of representation for many important AI problems such as configuration, planning, and scheduling. It has also led to new approaches to solving problems in the field of combinatorial optimisation. The key idea is to discard elements in the search space that cannot be part of a solution by showing that they lead to inconsistencies within small parts of the problem. Often this can be done in polynomial time.

The simplest and best-known methods establish *arc consistency* (AC), that roughly means consistency with one's neighbours in the network of constraints. However, in recent years, considerable attention has been paid to specialized

© Springer Nature Switzerland AG 2021
M. Baldoni and S. Bandini (Eds.): AIxIA 2020, LNAI 12414, pp. 301–317, 2021.
https://doi.org/10.1007/978-3-030-77091-4_19

methods for ensuring stronger forms of consistency. The most important methods use either AC reasoning based on strongly reduced domains, e.g. singleton arc consistency (SAC) and neighbourhood SAC (NSAC) [2,3,11], or reduced forms of path consistency [9]. In the present paper only the former methods are discussed.

In some algorithms, stronger forms of consistency reasoning are interleaved with weaker forms. This includes those (N)SAC algorithms where each SAC- or NSAC-based value deletion is followed by an AC step to make the entire problem arc consistent before proceeding to the next (N)SAC-based step.

Until now, such interleaving has not been the focus of research in this area. Yet its properties make it a strategy of considerable interest. The most important property is that interleaving as it is normally used does not affect the level of consistency eventually achieved, even though this level is greater than that obtained by the interleaved algorithm. This means that overall efficiency can be improved without any decrement in effectiveness, where the latter refers to values deleted and more generally to the degree of problem simplification.

However, the degree to which such interleaving actually improves performance is not known. Since both SAC and NSAC dominate AC, interleaving is not necessary. Moreover, there are algorithms that do not use interleaving, in particular SACQ and NSACQ, that often perform as well or better than those that do. Given such results, it is important to learn more about interleaving, to determine the conditions under which it speeds up processing, and, conversely, whether there are conditions (problem features) where it should be avoided. In particular, at this time it is not known whether SACQ and NSACQ would be even more efficient if AC interleaving was added.

Interleaving in (N)SAC algorithms is a special case of using multiple propagators in constraint programming [4,8]. It also has some relation to adaptive propagation [1,7], although the latter uses heuristic methods to find a best level of consistency to apply to a problem or constraint rather than mixing propagators to achieve a given level of consistency more efficiently. In addition, previous work has involved only a limited amount of empirical investigation. In particular, we would like to know not only whether to use such methods, but where one should use them.

The present paper subjects this form of multiple propagation in the (N)SAC setting to closer examination. We find that AC interleaving is generally beneficial, although there are conditions where it is detrimental. We show that there is a simple measure that can be used to determine whether a given problem type will be amenable to such methods. We also show that interleaving enhances (N)SACQ algorithms, which in their original form do not use this method; with this improvement, SACQ clearly becomes the best SAC algorithm, while the dominance of NSACQ over other NSAC algorithms is made more decisive. We also consider other approaches to interleaving, specifically a procedure based on the AC-4 algorithm, and interleaving with higher forms of consistency.

2 Background

2.1 General Concepts

A constraint satisfaction problem (CSP) is defined as a tuple, (X, D, C) where X are variables, D are domains (of values) such that D_i is associated with X_i, and C are constraints that place restrictions on the values that can be assigned to their respective variables. 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 can be represented as (hyper)graphs, where nodes are the variables and (hyper)edges are constraints. This representation highlights the importance of graph parameters such as *density*, based on the number of contraints, and *tightness* of individual constraints, i.e. the number of possibilities not allowed. For example, in a constraint between two variables whose domains have ten values, a tightness of 0.9 means that ninety of the 100 possible value pairings are not consistent.

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 used methods establish arc consistency, as noted earlier. In problems with binary constraints, 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 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 tuple of values that includes a that satisfies that constraint.

Singleton arc consistency, or SAC, is a particular form of AC in which the just-mentioned value a, for example, is considered the sole value in the domain of X_i. If AC cannot be established in the reduced problem, then there can be no solution with value a assigned to X_i, since AC is a necessary condition for the existence of such a solution. So 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 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 these do not include variables outside the neighbourhood of X_i. Problem P is neighbourhood singleton arc consistent if each value in each of its domains is neighbourhood singleton arc consistent.

2.2 (N)SAC Algorithms

Since the initial description of SAC-1 [3], several different SAC and NSAC algorithms have been described. This paper will restrict itself to the three SAC algorithms and two NSAC algorithms that are the most efficient in practice [10,11]. The SAC algorithms are SAC-1, SAC-3, and SACQ. The NSAC algorithms are NSAC-1 and NSACQ.

All SAC algorithms proceed by setting a domain to a single value and then establishing arc consistency under that condition. This is done for every value in every domain; hence AC is performed repeatedly until no more values can be removed in this manner. SAC-1 accomplishes this by using a repeat loop and going through the entire set of current domains again and again until nothing is deleted.

SAC-3 [2,5] also uses a repeat loop for the same purpose. However, instead of testing each domain value without reference to the others, values in different domains are tested using the problem reduced by earlier tests. This continues until a failure occurs; however, only when the failure occurs at the beginning of such a sequence (called a "branch") can the value be discarded. Savings in time occurs because values subsequent to the first on a branch are tested with a reduced problem. (If arc consistency can be established under these more restrictive conditions, then it will hold in the unreduced problem.) In practice, this can result in considerable speedup.

Instead of a repeat loop, SACQ [11] uses a queue of variables to be tested, consisting of the entire variable set. If a domain value is discarded, then any variable not on the queue is put back. Unlike the other SAC algorithms, which perform AC on the full problem after each SAC-based deletion, SACQ eschews this step, relying only on the basic SAC strategy to remove values.

NSAC-1 and NSACQ are identical to SAC-1 and SACQ, respectively, except that following the reduction of a domain to a singleton, AC is only performed on the neighbourhood subgraph. They, therefore, establish the more restricted form of singleton arc consistency called neighbourhood SAC.

To make all this more concrete, consider the pseudocode in Fig. 5 below, for a type of NSAC algorithm. Line 8 shows the NSAC-based consistency step, which is carried out for each domain value. (The domain reduction step precedes this on line 7.) Line 12 shows the AC step, which is *interleaved* between repeated NSAC steps. Note that this action only occurs if the NSAC step fails (produces a wipeout). SAC algorithms interleave AC in the same way, but in this case SAC is established at each step (line 8) instead of NSAC.

3 To Interleave or Not: Some (N)SAC Variants

The main purpose of the present paper is to evaluate the usefulness of the AC step that typically follows a singleton-based deletion in (N)SAC algorithms. Since both SAC and NSAC dominate AC, it is possible to eliminate AC interleaving in SAC-1 and SAC-3 as well as NSAC-1. In this paper, these will be called SAC-1noac, SAC-3noac, and NSAC-1noac.

While it is possible to add an AC step to SACQ or NSACQ, there are some complications. Since these algorithms use a queue rather than a repeat loop, if AC is done in addition, then after every AC-based deletion, the algorithm must ensure that all neighbouring variables are on the queue in order to be equivalent to the other (N)SAC-based algorithms. For this reason, these algorithms will be called SACQacn and NSACQacn. (Note. In some tables acn is shortened to ac and noac to no.)

Proposition 1. Both SACQacn and NSACQacn achieve the same unique fixpoints as SAC-1 and NSAC-1, respectively.

Proof. We begin with the fact that the basic versions of (N)SACQ achieve the same fixpoint as the corresponding (N)SAC-1 algorithms [11]. By this token, if the basic (N)SACQ procedure is followed, then the same dependencies between discarded values will be discovered as in the NSAC phase of (N)SAC-1. Since in addition we perform AC after each (N)SAC-based deletion, this reduces the problem in the same way as in the (N)SAC-1 case. Finally, by the Neighbourhood Lemma [11], the only way that an AC-based deletion of a value in the domain of variable X_j can affect the singleton arc consistency of any value in the remaining problem is via neighbours of X_j. So if these are put back on the queue after every AC- as well as (N)SAC-based deletion, such dependencies will always be discovered. □

4 (N)SAC with and Without AC: Initial Experiments

Algorithms were implemented in Common Lisp, and experiments were run in the Xlispstat environment with a Unix OS on a Dell Poweredge 4600 machine (1.8 GHz). Cross-checks were made for all problems tested to confirm that each type of (N)SAC algorithm deleted the same number of values for problems not proven unsatisfiable, and that the same unsatisfiable problems were proven unsatisfiable by equivalent algorithms (i.e. by all SAC and all NSAC algorithms, respectively). In these experiments, variables and values were always chosen according to the lexical order of these elements.

In this section we will only consider random (binary) problems where the probabilities of a constraint between two variables as well as a given tuple belonging to a constraint relation are the same throughout the problem. This will allow us to make initial comparisons among algorithms and to analyze why a given variant is better under a given condition. (As we will see, different parameter classes do give different patterns of results.)

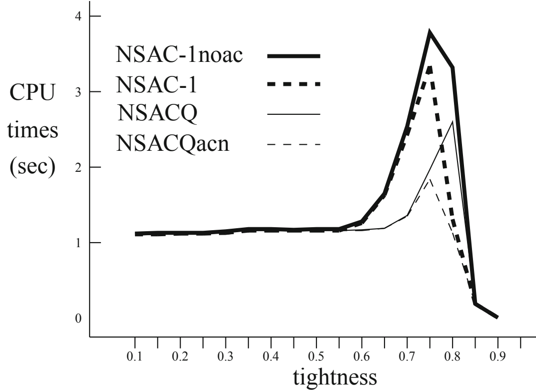

Fig. 1. Runtimes for four NSAC algorithms on random problems, <100, 20, .05, t> series. Note. In this and all other figures and tables, CPU times, ("runtimes") are for consistency (i.e. preprocessing) algorithms only. (Search times of course depend only on the level of consistency established, not the algorithm used to achieve this.)

We first look at a problem series that has been examined in the past [5,11]. These problems have 100 variables, domain size 20, and graph density 0.05. Constraint tightness is varied in steps of 0.05 from 0.1 to 0.9 inclusive; at each step 50 problems were tested. Another series of random problems was also tested. These had the same parameters as the first series except that the density was 0.25.

Figure 1 shows average runtimes for NSAC variants based on the first series, Fig. 2 for the second series. The first thing to note is that in the first series, both versions with AC interleaved outperform their simpler counterpart in some regions of the parameter space. On the other hand, for the second series, in the one range where the variants differ in performance (tightness 0.55 to 0.70), the non-interleaved versions outperform the corresponding versions with AC interleaving.

To understand these differences, first it should be noted that in the first series AC alone is sufficient to prove that problems at the two highest tightnesses are unsatisfiable, and for the second series this is true for the three highest tightnesses and almost true for the fourth (0.75, 47/50 proven unsatisfiable by AC). Hence, these cases are irrelevant for our purposes, since problems are proven unsatisfiable by the initial AC.

In the first series, NSAC can prove most problems unsatisfiable for tightness 0.8. For this tightness, AC interleaving reduces runtimes by a factor of 2, and this is the only case where this procedure makes a large difference. In this case, the interleaved AC sometimes deletes numerous values following an NSAC deletion, so many that in a number of cases wipeout occurs during the AC phase. Figure 3 shows NSAC and AC deletions for a problem that was not proved unsatisfiable during preprocessing. It illustrates how a single NSAC deletion can lead to numerous values deleted during the subsequent bout of AC. For lower tight-

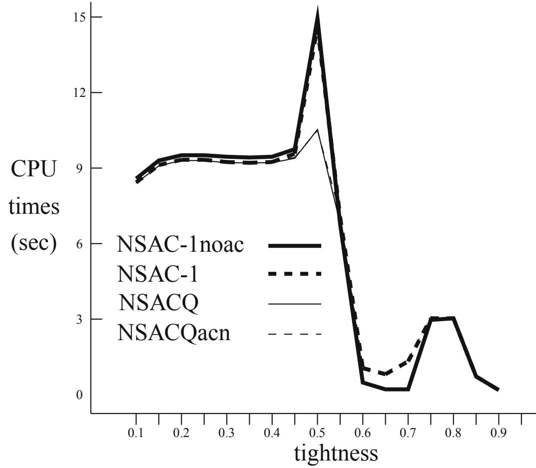

Fig. 2. Runtimes for four NSAC algorithms on random problems. <100, 20, .25, t> series

nesses, beginning with 0.75, very few values are deleted during the AC phase; here, runtimes are similar with or without interleaving.

In the second series (0.25 density), NSAC preprocessing proves all problems unsatisfiable over the range from 0.55 to 0.70, where a difference between NSAC variants is found. As with the first series, the interleaved AC deletes many values, but in this case NSAC reasoning alone produces a domain wipe-out with the first or second variable tested. Hence, while interleaved AC does lead to a wipe-out, sometimes after fewer singleton values have been tested, the process is slower, usually by a factor of 7–8, than with NSAC alone. Fortunately, the conditions under which this occurs seem to preclude long runtimes with or without interleaving, so this isn't a major drawback in itself.

Since for the most part the curves occlude each other, the significant portion of the data for SAC variants is shown in Tables 1 and 2. For the sparser problem series (Table 1), the only place where there are clear differences is for tightness = 0.75. For this tightness, all problems are proven unsatisfiable by SAC. With interleaved AC, during the AC phase very few values are deleted initially. But after several variables have been tested, there is an upsurge of AC-based deletions leading to wipe-out. Note, however, that runtime differences were only found for SAC-1 and SACQ.

For the denser problem series (Table 2), interleaving with SAC-1 or SACQ outperforms non-interleaving at two and possibly three tightness values (0.55 and 0.60 and possibly 0.50). In the first two cases, all problems are proven unsatisfiable by SAC, and, again, with interleaving the same eventual upsurge in AC-based deletions is found as in the first series. (For tightness 0.50, no problem was proven unsatisfiable by SAC.) It should also be noted that with SAC, differences due to interleaving are proportionally much smaller than with NSAC.

Table 1. Times for different forms of SAC on random problems at 5 tightness levels (<100, 20, .05, t> Series)

t	SACQ	SACQac	SAC-1	SAC-1no	SAC-3	SAC-3no
0.60	6.7	6.5	7.0	7.0	8.6	8.6
0.65	12.2	11.8	14.3	14.3	20.3	20.4
0.70	42.8	41.3	49.6	49.7	89.2	89.2
0.75	15.0	12.4	13.0	15.1	24.2	25.0
0.80	1.3	1.3	1.3	1.3	2.1	1.7

Means of 50 probs. Times in sec. "t" tightness.

With SAC-3, interleaving actually slows down the algorithm at one tightness value. (A possible reason for this will be discussed in a later section where tests with structured problems give similar results.)

What these results show is that for problems of this sort, interleaving only improves efficiency over a small part of the range of tightness values. The basic rule of thumb is that, if constraints are tight enough, then AC alone is likely to remove some values. This is observed in the initial AC. Afterwards, if NSAC removes more values, this increases the tightness in the neighbourhood of these values, and AC can again be effective.

Table 2. Times for different forms of SAC on random problems at 5 tightness levels (<100, 20, .25, t> Series)

t	SACQ	SACQac	SAC-1	SAC-1no	SAC-3	SAC-3no
0.50	131.5	128.5	159.4	160.2	893.1	906.3
0.55	162.5	156.5	154.0	163.0	252.9	225.0
0.60	20.9	17.1	16.9	20.9	18.7	22.4
0.65	8.0	8.3	8.2	8.1	9.7	8.6
0.70	3.9	4.6	4.5	4.0	6.4	4.2

See Table 1 for notes.

For random problems like these, this rule of thumb can be expressed in a simple formula, $expected\ support \cdot p(support) < 1$, where "expected support" is the number of supporting values for the value in question (i.e. those consistent with it) across a given constraint, and probability of support refers to the likelihood that those values are supported by some value across some other constraint. The basic idea is that interleaving will work when there are few supporting values, and the latter themselves do not have much support.

A simple statistic has proven useful for predicting the effectiveness of interleaving. This is the number of AC deletions per bout, where the number of bouts is equal to the number of (N)SAC deletions. For example, in the first problem series, at tightness 0.8 the average ratio of AC to NSAC deletions (here called the

"bout ratio") was 1.58 for the seven problems not proven unsatisfiable, while for tightness 0.75, where there was only a slight difference in favour of interleaving, the ratio was 0.19, and for lower tightnesses the ratio was 0.

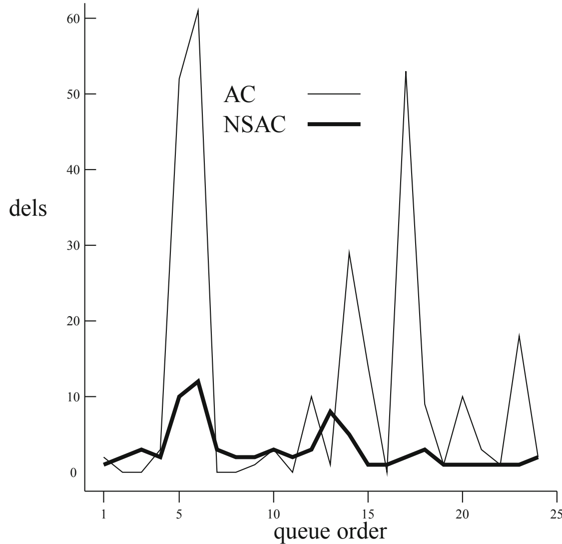

Fig. 3. Number of deletions for successive variables in queue. Problem from <100, 20, .05, 0.80> set, where interleaving lowers runtimes appreciably (Bout ratio for problem = 3.79. Points in graph represent one to several bouts, depending on how many values of the variable being tested were deleted by NSAC.)

For problems like these, we can also infer whether AC interleaving is likely to be effective from the results of the initial AC: if numerous values are deleted at this time, then interleaving AC with NSAC or SAC is likely to be effective as well. To determine if this rule has general application, we must look at a variety of problem classes.

5 (N)SAC with and Without AC: More Extended Tests

More extended tests were done using various problem types including randomly generated structured problems, benchmarks, and benchmarks with added global constraints. All problems used in these tests had solutions; hence, both AC and NSAC always ran to completion without generating a wipeout. In addition, problems were chosen where (N)SAC deleted a large number of values on top of the initial AC.

The following problem classes were used:

- Relop problems had 150 variables, domain size 20, with constraint graph density of 0.30. Half the constraints were of the form $X_i \geq X_j$ and half were inequality constraints.
- Distance problems had 150 variables, domain size 12, and constraint graph density = 0.0307. Constraints were of the form $|X_i - X_j| \otimes k$, where $\otimes$ was either $<>$ (60%), $\geq$ (30%), or = (10%). The value for k varied from 1 to 8, with a mean value of 4.
- RLFAP-graph problems were benchmarks, with 200 or 400 variables; these problems also have distance constraints where $\otimes$ is either = or $>$ (for the former k is always 128; for the latter it varies widely). (It may be noted that these were drawn from an original set of 7, where four had solutions, two of which did not give deletions with any form of SAC-based preprocessing.)
- Driverlog problems were benchmarks, which are CSP representations of a well-known transportation problem. All but the smallest problem were used. The number of variables varies from 272 to 408. For the smallest problem, domain sizes range from two to eight; for the largest the range is 2–11. Constraints are binary table constraints, many very loose.
- Open shop problems were from the Taillard series. The ones used here are the Taillard-4-100 set. Constraints are disjunctive of the form $X_i + k_i \leq X_j \bigvee X_j + k_j \leq X_i$.
- The RLFAP-occurrence problems were based on the RLFAP-graph3 benchmark. In this case ten percent of the k values of $>$ distance constraints were altered (by randomly incrementing or decrementing them) to make the base problem more difficult. Then various forms of occurrence constraints were added: three atmost, three atleast, and three among. In addition, one disjoint constraint was also included. For occurrence constraints, arity varied between four and six; the disjoint constraint always had arity 10 based on two mutually exclusive sets of five variables. Each occurrence constraint could affect a maximum of 75% of the variables in its scope. The among constraints could involve up to 50% of the possible values. There was no overlap in the scopes of constraints of the same type; between types up to 50% of the scope could overlap.
- Configuration problems were derived from an original benchmark refrigerator configuration problem ("esvs") composed of table constraints with arities ranging from two to five. To make these problems, some constraints were tightened, and in some cases the problem was doubled in size by duplicating the constraint patterns.
- Golomb ruler problems were benchmarks obtained from a website formerly maintained at the Université Artois. Constraints had arities 2 or 3. Since NSAC only deleted a few more values than AC, tests were restricted to SAC for these problems.

Results are shown in Table 3 for NSAC and Table 4 for SAC. In addition to number of values deleted by AC and by NSAC following AC and overall runtimes, Table 3 shows the proportional changes in runtime, when interleaving is used versus no interleaving based on the formula:

$$reduct = \frac{basic - interleave}{basic}$$

(In Table 4, only the proportional changes in runtimeare shown.) Results for the algorithm whose best variant also gave the best performance overall are shown in boldface. Changes that resulted in runtime differences that were statistically significant at the 0.01 level (paired comparison t-test, two-tailed) are underlined.

As Table 3 indicates, the effectiveness of interleaving with neighbourhood SAC algorithms varied significantly for different problem types. The data for bout ratios show that this was because problems varied widely in their amenability to AC interleaving.

As in earlier work, it was found that NSACQ (in either form) was usually the more efficient algorithm. In cases where interleaving was effective, both NSAC algorithms showed improvement; as a result NSACQ maintained its superiority, and in some cases the difference became even greater. The one exception to this pattern was the RLFAP-occurrence problem set, where the basic NSACQ algorithm was markedly inferior to NSAC-1 (205 versus 102 sec per problem). However, with interleaving it became the most efficient overall (84 sec). Another finding was that the intermittent deletion of larger numbers of values, as shown in Fig. 3, occurred with all types of problems in which interleaving was effective.

With full SAC algorithms, the pattern of effectiveness of interleaving across problem types was similar to NSAC (Table 4). With these more powerful consistency algorithms all problem classes showed some amenability to interleaved AC in terms of values deleted. However, as with NSAC, very small bout ratios were associated with increases in runtime when interleaving was used.

Table 3. Effect of interleaving NSAC and AC for various problem classes

probs	n	removals		NSAC-1			NSACQ			btratio
		iAC	NSAC	rt-no	rt-yes	reduct	rt-no	rt-yes	reduct	
Relop	41	0	1702	231.4	183.1	21	194.9	161.7	**17**	1.12
Distance	33	215	23	6.4	4.3	28	2.7	2.4	**9**	2.58
RLFAP-graph	2	558	997	1166.6	944.6	16	552.1	496.8	**6**	1.25
Driverlog	5	55	51	163.0	167.0	−2	93.9	97.0	**−3**	0
Open shop	10	117	862	73.0	83.6	−14	65.6	71.8	**−9**	0
RLFAP-occur	50	346	5452	269.5	101.9	62	204.9	84.2	**59**	3.81
Configuration	9	46	28	0.1	0.1	−3	0.1	0.1	**−18**	0.04

n is number of problems in group. removals is values deleted. iAC is initial AC. NSAC is NSAC after iAC. rt-no and rt-yes are runtimes without and with inter-leaving. Reduct is percent time reduction due to interleaving. btratio is bout ratio. All values except those under n are group means.

Some anomalous results were found with SAC-3. For this algorithm interleaving sometimes had untoward effects with respect to runtime that were greater than for the other two algorithms, and in one case (RLFAP-occ) this occurred

in spite of the large bout ratio. Presumably there are interactions between interleaving and the branch strategy, perhaps because the latter entails a different order of value testing (since only one value per domain can appear on a branch). (Configuration problems were not tested with this algorithm because it wasn't clear how to combine branch-building with simple table reduction.)

Table 4. Effect of interleaving SAC and AC for various problem classes

probs	n	removals		SAC1	SAC3	SACQ	btratio
		iAC	SAC	Reduct	Reduct	Reduct	
Relop	41	0	2204	30	**37**	26	0.91
Distance	33	215	139	10	9	**10**	1.36
RLFAP-grph	2	558	1517	−1	−1	**4**	1.16
Driverlog	5	55	437	10	1	**3**	0.64
Open shop	10	117	1549	−9	−32	**−3**	0.14
RLFAP-occ	50	346	6042	**31**	−27	**27**	5.04
Configurat	9	46	58	−5	*	**−11**	0.18
Golomb-3	10	221	473	3	1	**4**	0.55

* denotes combination not tested. See Table 3 for further notes.

Turning to specific problem classes, for the relop problems AC by itself does not delete any values. However, since these constraints are highly structured, one cannot infer from this that interleaving will be ineffective. (This shows that the initial AC rule suggested earlier does not hold in all cases.) In fact, for $>$ or $\geq$ constraints the amount of support for a given value ranges from d or $d-1$ down to 1 or 0. Hence, there are always values with little or no support. Moreover, this type of constraint has a 'progressive' property in that, if values with the least amount of support are deleted, then the values with minimal extra support become as poorly supported as the values that were deleted. Because of these features interleaving was in fact beneficial; as the bout ratios show, slightly more values were deleted by interleaved AC when NSAC was used, and slightly less with SAC. For problems of this type, the variability of this ratio across individual problems was quite small (range $\approx$0.3).

In contrast to most problem classes, bout patterns for distance problems were quite variable. For 17 of the original 50 problems, NSAC did not delete any more values than AC; hence, these were not included in the table. For the remaining 33 problems, the bout ratio varied from 0 to 13.0. Overall, however, interleaving was effective, more so for NSAC than for SAC.

For RLFAPs, all SAC- or NSAC-based deletions were followed by at least one AC-based deletion. (This is due to the equality constraints that affect successive pairs of variables.) Only a very few times in the series did interleaving lead to a large number of deletions. For RLFAP-occurrence problems, the singleton deletion pattern naturally also occurred; in addition, there were more bouts where large numbers of deletions occurred in the AC phase.

Golomb ruler problems showed a different pattern of AC deletions. In each case, AC deletions only occurred within the first 7-17 SAC deletions depending on the problem (out of a total of 44-339). In each case the first AC deletion occurred after the second SAC-based deletion, after which there was a pattern in which the greatest number of AC deletions occurred after the third SAC deletion, the second greatest after the fifth, and so forth, the pattern becoming clearer with larger problems where the series was longer.

Together, these results show that having "structure" does not in itself alter basic propagation effects, in particular the intermittency of large numbers of deletions, or the deductions that can be made regarding relative efficiency derived on this basis (reflected in the bout ratios).

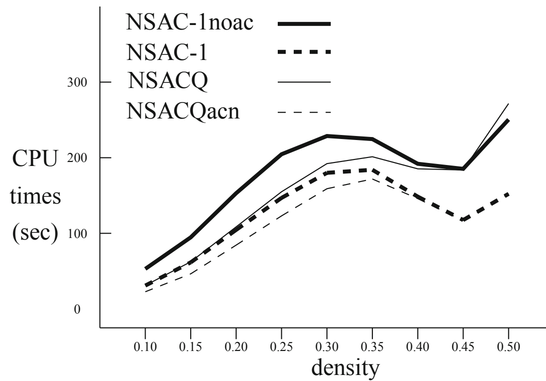

Fig. 4. Runtimes for four NSAC algorithms on random relop problems of varying density.

For randomly generated problems, one can vary problem parameters systematically in order to compare performance across the problem space. This was done with relop problems in an experiment where density was varied from 0.10 to 0.50 in steps of 0.05. (Fifty problems were tested at each density. Those for density 0.3 include the problems used in Table 3.) Results are shown in Fig. 4. At densities >0.3 all problems were unsatisfiable, while <0.3 all were satisfiable. At density 0.35, NSAC proved one problem unsatisfiable; at density 0.4, NSAC proved 32 problems unsatisfiable; at higher densities all 50 problems could be proven unsatisfiable.

Figure 4 shows that carrying out NSAC is fairly expensive. However, for some of these problem sets NSAC can reduce search times by a much larger amount. It also shows that interleaving is always more efficient and that for some problems, it can reduce runtimes by about 100 s per problem (a reduction of about 40%).

6 Comparisons with an AC4-Style Interleaving Algorithm

As we have seen, by testing a few problems in a class, it is sometimes possible to determine whether AC interleaving is likely to have benefits. In addition, it was thought that it might be possible to finesse the problem to a degree by using a more efficient form of interleaving. To this end, an algorithm based on AC-4 data structures was devised. (Here, we will only consider the binary form of this algorithm.)

To understand the algorithm, the reader should recall that AC-4 has two phases. In phase 1, all value combinations are checked, and data structures representing support sets are set up; these include lists of values supported by each value for each constraint (support sets), counters that tally the number of supports for a value across each constraint, and an array of binary values to indicate whether a value in a given domain is still viable. Phase 2 begins with the values found to have no support across some constraint (the “badlist”) and uses these to decrement counters for each member of the support set associated with each bad value. This continues, with new values being added to the badlist if one of their counters goes to zero, until the badlist is empty or a wipeout has occurred. (See [6] for details.)

```
   Procedure NSAC-1AC4
1    OK ← AC4(P)
3    Repeat                          /* if OK */
4      Changed ← false
5      Foreach X_i ∈ X
6        Foreach v_j ∈ dom(X_i)
7          dom'(X_i) ← {v_j}
8          If AC3(X_i+neighbours(X_i)) leads to wipeout
9            dom(X_i) ← dom(X_i)/v_j
10           Set entry for v_j in mark array to false
11             and set badlist to ((X_i, v_j))
12           OK ← AC4phase2(P)
13           Changed ← true
14   Until Changed == false or not OK
```

Fig. 5. Pseudocode for NSAC-1 incorporating AC-4.

When combined with NSAC, AC-4 is used for the initial AC pass. During subsequent (N)SAC processing, the support count system continues to be used whenever SAC-based processing proves that a value can be discarded. Hence, the phase 1 set up is done only once. In devising this procedure, the assumption was that the main cost of AC-4 involves the setting up of data structures in phase 1. For SAC or NSAC algorithms, the original cost may therefore be amortized through repeated use of the efficient phase 2 procedure.

The present algorithm uses AC-3 for (N)SAC-based reasoning, and uses phase 2 only for AC interleaving. Figure 5 gives pseudocode for the algorithm when used with NSAC-1.

Proposition 2. The NSAC-AC4 algorithm given in Fig. 5 is correct, complete and terminates.

Proof Sketch. Since the algorithm has only been coded in its binary form, we will restrict our arguments to this class of problems. Here, we assume the soundness of the basic (N)SAC procedures. Since AC-4 establishes complete sets of supports, then for each value that is found to be (N)SAC-inconsistent, all counters associated with adjacent values will be decremented properly. The same is true for variables adjacent to the latter, etc.; this follows from the correctness of the AC-4 procedure. Hence, by the correctness and completeness of AC-4, after an (N)SAC-based deletion all values deleted will have become arc-inconsistent and all arc-inconsistent values will be deleted. Hence, after each instance of (N)SAC-based deletion, the network will be made arc consistent as required. □

Unfortunately, the assumption about the efficiency of AC-4 phase 2 turned out to be false, at least for the present implementation. In practice, the present algorithm typically runs an order of magnitude slower than the algorithms based on AC-3. In fact, this is likely to be a general problem since constantly updating a large number of entries is bound to take time, but this is required for the correctness of the algorithm.

7 Other Kinds of Interleaving

Interleaving using NSAC was also tested, where SAC is the basic algorithm. The simplest combination is to use NSAC initially and then run the SAC algorithm. With the SACQacn algorithm, this has resulted in improvements of up to 30% (e.g. with RLFAPs), although in one case it led to a 10% decrement (with driverlog problems). The key factor seems to be the effectiveness of NSAC relative to SAC; if the former algorithm is almost as effective, then a noticeable speedup can be obtained. (For example, for the graph3 RLFAP included in Tables 3 and 4, NSAC deletes 1064 values, SAC 1274, so that after an initial NSAC run, there are only 210 values left to delete.) But if SAC is much more effective than NSAC, then interleaving with the latter can increase overall runtime.

Note that in the present implementation AC is run first as before, then NSAC, then SAC. Thus, a cascade of consistency maintenance algorithms is applied, beginning with the weakest.

To date, no pattern of actually interleaving with NSAC has yielded further benefits. Further experiments showed that with the same problems, the earlier that interleaving with NSAC was done, the more effective it was. When two interleavings are allowed with the same problems (after one- and two-thirds of the SAC deletions), the runtime increases to about what it is with SAC alone, and with more interleavings performance is worse.

Another strategy that was tried is based on observation of the pattern of deletions by (N)SAC and AC. It was noted that in most cases large numbers of values were deleted by AC following a series of SAC-based deletions from the same domain. However, to date, interleaving with NSAC under these circumstances did not confer any further benefit.

8 Conclusions

This paper explores a little studied topic in the field of constraint satisfaction. Although the basic method has been used for many years (with SAC algorithms other than SACQ), heretofore it has not been the subject of analysis in its own right. One purpose of the present paper is to call attention to what may be a significant topic for further research.

By employing interleaving in a somewhat novel context, in combination with the queue-based strategy used in SACQ and NSACQ, it has been possible to produce the best algorithms proposed to date for SAC and neighbourhood SAC. Since arc consistency can be extended to generalized arc consistency in a straightforward way, these algorithms can be used with constraints of any arity; to date, the improvements demonstrated here apply to n-ary problems as much as to problems with only binary constraints. This work also serves to confirm the general superiority of AC-3 to AC-4 techniques.

Since for all problem classes interleaving AC was only intermittently effective, this suggests that this procedure could be used only intermittently to achieve even better performance. However, in this case one runs the risk of using SAC or NSAC to delete a value that could have been deleted with AC. Since some problem types are not amenable to AC interleaving, a better strategy may be to make such interleaving optional, using it only for problems where one can expect it to be effective.

From the present experiments, it appears that interleaving with more powerful algorithms than AC only works when the interleaved algorithm is itself effective on the same problem and when it is used early in the SAC process, e.g. when it is used before running SAC. However, this field is still wide open. In addition, there are many interesting tradeoffs that should be explored further, such as those related to the costs of the interleaved and base algorithm, and to their relative effectiveness.

Acknowledgements. I thank the anonymous reviewers for their close reading and apposite comments, which definitely improved the quality of the paper. This work was done using facilities supported by Science Foundation Ireland.

References

1. Balafrej, A., Bessière, C., Bouyakhf, E., Trombettoni, G.: Adaptive singleton-based consistencies. In: Twenty-Eighth AAAI Conference on Artificial Intelligence - AAAI 2014, pp. 2601–2607. AAAI (2014)

2. Bessière, C., Cardon, S., Debruyne, R., Lecoutre, C.: Efficient algorithms for singleton arc consistency. Constraints **16**, 25–53 (2011)
3. Debruyne, R., Bessière, C.: Some practicable filtering techniques for the constraint satisfaction problem. In: Fifteenth International Joint Conference on Artifcial Intelligence - IJCAI 1997, vol. 1. pp. 412–417. Morgan Kaufmann (1997)
4. Granvilliers, L., Monfroy, E.: Implementing constraint propagation by composition of reductions. In: Proceedings Nineteenth International Conference on Logic Programming. LNCS No. 2916. pp. 300–314. Springer (2003)
5. Lecoutre, C., Cardon, S.: A greedy approach to establish singleton arc consistency. In: Fourteenth International Joint Conference on Artificial Intelligence - IJCAI 2005, pp. 199–204. Professional Book Center (2005)
6. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artif. Intell. **28**, 225–233 (1986)
7. Paparrizou, A., Stergiou, K.: Evaluating simple fully automated heuristics for adaptive constraint propagation. In: Twenty-Fourth International Conference on Tools with Artificial Intelligence - ICTAI 2012, pp. 880–885 (2012)
8. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. ACM Trans. Program. Lang. Syst. **31**(1), 1–43 (2008)
9. Stergiou, K.: Restricted path consistency revisited. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 419–428. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_30
10. Wallace, R.J.: Light-weight versus heavy-weight algorithms for SAC and neighbourhood SAC. In: Russell, I., Eberle, W. (eds.) Twenty-Eighth International Florida Artificial Intelligence Research Society Conference - FLAIRS-28, pp. 91–96. AAAI Press (2015)
11. Wallace, R.J.: SAC and neighbourhood SAC. AI Commun. **28**, 345–364 (2015)