Partial (Neighbourhood) Singleton Arc Consistency for Constraint Satisfaction Problems [⋆]

Richard J. Wallace

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

Abstract. Algorithms based on singleton arc consistency (SAC) show considerable promise for improving backtrack search algorithms for constraint satisfaction problems (CSPs). The drawback is that even the most efficient of them is still comparatively expensive. Even when limited to preprocessing, they give overall improvement only when problems are quite difficult to solve with more typical procedures such as maintained arc consistency (MAC). The present work examines a form of partial SAC and neighbourhood SAC (NSAC) in which a subset of the variables in a CSP are chosen to be made SAC-consistent or neighbourhood-SAC-consistent. These consistencies are well-characterized in that algorithms have unique fixpoints and there are well-defined dominance relations. Heuristic strategies for choosing an effective subset of variables are described and tested, in particular a strategy of choosing by constraint weight after random probing. Experimental results justify the claim that these methods can be nearly as effective as full (N)SAC in terms of values discarded while significantly reducing the effort required.

1 Introduction

Singleton arc consistency (SAC) is a well-known enhancement of arc consistency (AC). The basic idea is to reduce the set of possible domain values associated with a variable to a singleton value a before establishing AC. Under these conditions, failure in the form of a domain wipeout somewhere in the problem implies that there is no solution containing a. Hence, it can be discarded without affecting the solution set for the problem. When this is done for each value in the problem, the resulting problem is singleton arc consistent [1].

Neighbourhood singleton arc consistency (NSAC) is a limited form of SAC-based reasoning in which only the subgraph based on the neighbourhood of the variable with the singleton domain is made arc consistent during the SAC phase [2]. This form of singleton arc consistency can sometimes be nearly as effective as SAC (with respect to values deleted and proving unsatisfiability) while requiring much less time. NSAC variants have also been devised and tested in which the neighbourhood in question is extendeded to the k-neighbourhood [3]. Thus, roughly speaking, 2-NSAC refers to singleton arc consistency based on the both the immediate neighbourhood of a variable and

[⋆] This is a slightly longer version of a paper accepted for the FLAIRS-31 conference.

the neighbours of the neighbours, while 3-NSAC adds the neighbours of the neighbours of the neighbours. (In this paper, “SAC-based reasoning” refers to these algorithms collectively, since they are all built around the basic idea of singleton arc consistency.)

Although these algorithms can significantly improve runtimes during subsequent search, even the most efficient algorithm of this type is still fairly expensive. When compared with the other candidates for enhanced local consistency, in particular algorithms based on restricted path consistency (RPC) [4], they are much more expensive, although they typically allow more values to be removed [5]. However, unlike RPC algorithms, (N)SAC algorithms can be readily extended to problems with n-ary constraints, including global constraints, which may not be readily transformed into binary problems. [3, 5]. This is sufficient motivation for trying to discover reduced forms of (N)SAC that are still reasonably effective while being more efficient.

In the present work, various forms of partial SAC and NSAC are introduced. In all cases, the basic idea is to select a small set of variables whose domains are made consistent in accordance with the specified form of consistency. When this is done in accordance with some simple rules, the result is a partial singleton arc consistency that is well-defined with regard to fixpoints and dominance relations.

There is an extraordinary variety of strategies that can be devised based on these new methods. So only a fraction of the possibilities can be considered in a short article. In this work, we will establish the foundations for the general approach and examine some ways that it can be utilized. We also show that if a ‘good’ subset of variables is chosen, the effectivess of these procedures in terms of reducing subsequent search can be almost as great as when the same problems are made fully (N)SAC.

Section 2 presents background concepts and definitions. Section 3 describes some algorithms that establish full SAC or k-NSAC. Section 4 introduces partial (N)SAC algorithms and their properties. Section 5 describes experimental methods and Section 6 gives experimental results. Concluding remarks are made in Section 7. Due to space limitations most proofs of propositions are omitted.

2 Background Concepts

A constraint satisfaction problem (CSP) involves assigning values to a set of variables subject to restrictions on the way that values can go together. More formally, a CSP can be defined as a tuple, (X, D, C) where:

X is a set of *variables*, $X_1, \ldots, X_n$,

D is a set of *domains*, D_i, where each D_i is a set of possible *values* for variable X_i

C is a set of *constraints*. Each C_i belonging to C consists of a relation R_i and a particular subset of the variables in X, called the *scope* of the constraint. R_i is based on the Cartesian product of the values of the domains of the variables in the scope.

A *solution* to a CSP is an assignment or mapping from variables to values, $A = \{(X_1, a), (X_2, b), ..., (X_k, x)\}$, that includes all variables ($k = n$) 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 methods for filtering out values that cannot participate in a solution, based on local inconsistencies, i.e. inconsistencies with respect to subsets of constraints. By doing this, these algorithms establish well-defined forms of local consistency in a problem.

The most widely used methods establish *arc consistency*. In problems with binary constraints, arc consistency (AC) refers to the property that for every value a in the domain of variable X_i and for every constraint C_{ij} involving X_i there is at least one value b in the domain of X_j such that (a,b) satisfies that constraint. A similar definition can be given for constraints of higher arity.

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. (Here, X_i will be called the focal variable.) If AC cannot be established under this condition, then there can be no solution with this value, 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.

If for each value $a \in D_i$, where i is in $\{1 \ldots n\}$, arc consistency can be established in the subgraph based on that variable and its neighbours, then the problem is neighbourhood singleton arc consistent.

For k-neighbourhood SAC, 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 variables connected by a path of length k or less to the focal variable. For example, 2-NSAC is based on subgraphs that include all variables that can be reached from X_i by a path of length 1 or 2. Obviously, when k becomes large enough the subgraph includes all variables, and k-NSAC is equivalent to SAC.

3 Basic (N)SAC Algorithms

Due to space and time constraints, the work will focus on "light-weight" SAC and NSAC algorithms given their simplicity and scalability [6]. These include the original SAC-1 procedure in which SAC or NSAC is performed repeatedly until no values can be deleted [1]). Interestingly, in the present work where only a fraction of the variables need to be processed, the disadvantage of repeated consistency testing is greatly reduced.

In addition, some algorithms are based on (N)SACQ. This type of algorithm employs an AC-3 style of processing at the top-level to avoid (N)SAC-1's repeat loop. As with SAC-1, there is a list (a queue) of variables, whose domains are considered in turn. But in this case, if there is an (N)SAC-based deletion of a value, then any variables in the current set and not on the queue are put back on. For SACQ, the current set is all variables in the problem; for k-NSAC, it refers to any variable in the k-neighbourhood.

The idea behind this manoeuvre is that if the deletion has any effect it must affect its neighbours, and any effects elsewhere in the problem can only occur through effects on its neighbours.

Unlike other (N)SAC algorithms, with (N)SACQ algorithms there is no "AC phase" after 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). Detailed descriptions of these and other (N)SAC algorithms as well as references to earlier work can be found in [2, 3].

Although the worst-case complexity of (N)SACQ is no different from that of (N)SAC-1 [2], in practice, SACQ is usually somewhat faster than SAC-1. Undoubtedly because the queue is much smaller, NSACQ is much faster than NSAC-1 and similar results are found for k-NSAC with $k = 2$ or 3 [2, 3].

In this work, all (N)SAC algorithms were preceded by a step in which arc consistency was established. 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. For partial (N)SAC algorithms, this step is crucial for guaranteeing dominance over AC.

4 Partial (N)SAC Algorithms

4.1 Basic description of the procedures

In all cases, following selection of a subset of variables, a given procedure is carried out in a way that is based on the procedure followed when all variables are chosen. For example, when a partial version of (N)SAC-1 is used, then each variable in the set is made singleton arc consistent. If during this procedure, *any* values are deleted, then the entire process is repeated. Therefore, the procedure is repeated until no values are deleted.

Similarly, with partial SACQ, after any value removal, all variables in the selection set are put back on the queue. For partial NSACQ, all *neighbouring* variables in the set chosen are put back.

It should be emphasized that all partial (N)SAC algorithms described here still entail (N)SAC testing of each value in the domain of a variable that is tested. For k-NSAC this means that the full k-neighbourhood is made AC; for SAC, the entire constraint network is made AC. The only restriction is that only a subset of variables are subject to singleton arc consistency checking. In what follows the subset of variables chosen for some form of (N)SAC processing is called the "variable selection set" or simply the "selection set".

4.2 Properties of the algorithms

It can be shown that for any subset of variables chosen, if these procedures are followed in the manner to be described, then the procedure is associated with a unique fixpoint. Hence, each procedure produces a well-defined result. This is easiest to show for (N)SAC-1 algorithms.

Proposition 1. If the basic (N)SAC-1 procedure is followed, then an algorithm that establishes a partial version of SAC or k-NSAC for a given variable selection set will always achieve the same fixpoint.

Proof Sketch. We begin with the fact that full SAC-1 or k-NSAC-1 achieves a unique fixpoint [2, 3]. Now we follow the same procedure using only a selection set that is a proper subset of the full set of variables. For AC-1 procedures it is obvious that the same logic holds as for the full algorithms, since one examines every (selection set) variable and its values again and again until no values are deleted. This procedure will uncover any dependency between values discarded regardless of the order in which variables are tested just as it does for any algorithm that uses AC-1 on the full variable set. In addition, since in keeping with the basic (N)SAC-1 procedure, AC is re-established for the entire problem after each (N)SAC-based deletion, any differences that might accrue because of undetected arc-inconsistent values are avoided. □

Note that in the version of the algorithm described above, since AC is carried out before (N)SAC, and after every (N)SAC-based deletion, partial (N)SAC based on any selection set will dominate AC.

It is fairly obvious that if partial (N)SACQ follows precisely the same procedure as full (N)SACQ, then it will not always reach the same fixpoint as partial (N)SAC-1 based on the same selection set. This is because the former will not detect arc-inconsistencies outside the selection set, while the latter will. Moreover, even if an AC step is introduced every time a value is deleted because of (N)SAC-based reasoning, this will not guarantee that all values deleted by the corresponding partial NSAC-1 algorithm will be deleted, since full AC can uncover inconsistencies outside the neighbourhoods of the variables remaining on the queue without deleting the values in question. However, results can be obtained that correspond to those of the equivalent partial NSAC-1 procedure if an additional step is carried out. This is indicated by the next proposition.

Proposition 2. If the basic (N)SACQ procedure is followed, and in addition, (i) the full problem is made arc consistent after each (N)SAC-based deletion, (ii) after each AC-based deletion, any neighbouring variables that are also in the selection set are put back on the queue, then for a given selection set the algorithm will achieve the same fixpoint as the corresponding (N)SAC-1 procedure.

Proof Sketch. Again, we begin with the fact that the full versions of (N)SACQ achieve the same fixpoint as the corresponding (N)SAC-1 algorithms [2]. By this token, if we follow the basic (N)SACQ procedure using a selection set that is a proper subset of X the set of variables, then the same dependencies between discarded values will be discovered as with (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 the only way that an AC-based deletion of a value in the domain of a variable X_i can affect the singleton arc consistency of a value of a variable in the selection set is via neighbours of X_i that are also in the selection set. 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. □

Whether this extended form of partial NSACQ will outperform or underperform the NSAC-1 version must be evaluated experimentally.

Given these varying relations between the two strategies, the following definition is useful.

Definition 2. An algorithm that deletes the same values as the corresponding (N)SAC-1 algorithm will be called SAC-1 equivalent.

Cases where SAC-1 equivalence actually involves k-NSAC-1 equivalence will be clear from context. Note that all full (N)SAC algorithms described in the literature are SAC-1 equivalent, so the definition is only useful for partial (N)SAC.

Conclusions for other (N)SAC algorithms will be presented informally. All (N)SAC-3 algorithms follow a SAC-based deletion with full AC and use a repeat loop that runs until quiescence just like (N)SAC-1 algorithms do. The main differences are that the ordering of SAC tests is constrained by branch-building requirements (no two values from the same variable domain on one branch) and that some SAC tests are carried out on a problem whose domains are reduced by previous SAC tests. Thus, partial versions of the former are always SAC-1 equivalent. Since SAC-SDS also does a full AC following any SAC-based deletion, and since all AC-based deletions result in the variable/value queue being updated, then partial versions of this algorithm are SAC-1 equivalent. Since k-NSAC-SDS retains these features and performs a full NSAC test for each variable/value pair on the queue, partial versions of these algorithms are also SAC-1 equivalent. The same arguments can be made for partial SAC-2 algorithms and possibly for NSAC-2.

Since they are correct albeit incomplete in some respects, either of the simpler NSACQ procedure will delete more values than can be done with AC alone. In addition, each of the more elaborate partial NSACQ procedures is well defined in the sense of leading to a unique fixpoint. Moreover:

Proposition 3. If the basic (N)SACQ procedure is followed for a selection set that is a proper subset of X (without the extensions described above), then the procedure will reach a unique fixpoint.

Proof Sketch. In this procedure, value deletions can only occur if a value is (N)SAC inconsistent or if this level of consistency depended on a value that has been deleted. In the former case, inconsistency will be discovered directly since all values in the domains of the variable selection set are tested. In the latter case any such dependencies will lead to value deletion because of the neighbourhood lemma and because following any (N)SAC-based deletion, all the neighbours in the selection set are put back on the queue for re-testing. □

This proposition has a significant corollary:

Corollary to Proposition 3. Using the basic (N)SACQ procedure, if the selection set consists of variables none of which is in the k-neighbourhood of any other, then a well-defined partial k-NSAC can be established in a single pass over the selection set.

The next proposition follows immediately from the previous discussion.

Proposition 4. Partial (N)SAC-1 equivalent procedures dominate simple pNSACQ procedures, as defined in Proposition 3, when both are based on the same selection sets.

That is, any value eliminated by the simple NSACQ procedure will be eliminated by the corresponding NSAC-1 equivalent procedure, while the converse does not hold.

The next proposition shows that given any set of variables, then for any set that includes the first set, the set of values deleted after application of some form of SAC or NSAC will always include the values deleted using the smaller set of variables.

Proposition 5. Given problem P with variables X and two selection sets S_1 and S_2, where $S_1, S_2 \subseteq X$ and $S_1 \subseteq S_2$, then for a partial (N)SAC algorithm establishing a given level of SAC or k-NSAC consistency, if a value is deleted when S_1 is tested, it will also be deleted when S_2 is tested, but the converse does not hold.

Proof. This follows immediately from fact that every variable, value pair tested in the smaller set for the given level of consistency is also tested in the superset but not vice versa. Note that the same principle holds for both SAC-1 equivalent and simple NSACQ forms of partial (N)SAC. $\square$

Corollary. For a given level of (neighbourhood) singleton arc consistency, the sets of deleted values associated with each possible variable selection set form a partial order based on set inclusion.

These results raise a number of questions concerning the relative effectiveness (number of values deleted) and efficiency (run time) of the various possible procedures. But before turning to experimental tests, the important issue of choosing a good selection set of size k will be discussed.

4.3 Heuristics for choosing selection sets

In choosing a selection set our ultimate goal is to minimize overall runtime, and this means finessing the tradeoff between preprocessing and search times. Specifically, we want to reduce preprocessing time with respect to the full form of (N)SAC without increasing search time excessively.

The key question is where in the problem will deleting values lead to a maximum reduction of search effort, reflected in the number of search nodes. Since search is maximally reduced when the fail-firstness is maximized [7], it would seem best to choose variables that participate in significant bottlenecks, and these are necessarily variables of higher degree in the problem.

Unfortunately, if we choose high degree variables for our selection set, then establishing (N)SAC-based local consistency is likely to take longer, so we are faced with another tradeoff.

Another issue is the size of the selection set. Presumably the ‘optimal’ size k, i.e. the size that finesses the tradeoff between efficiency (run time) and effectness (number of values deleted) most adequately, will vary for different problems even within one problem class. It may also vary depending on the selection strategy used.

Given all these considerations, it would seem most useful at the outset to establish some empirical relations between features of the selection sets and preprocessing and search times as well as collecting data to establish which strategies are the best. In this analysis, a random selection (repeated over each problem in the sample) of size

$k < n$ was used as a reference. Then, heuristic strategies were compared based on the following criteria:

- The k variables of highest degree.
- The k highest degree variables forming a connected subgraph.
- The k highest degree variables such that there are no neighbouring variables among them
- The k variables of highest constraint weight, as established by random probing.

The last heuristic in the list incorporates the method of random probing that has been used with weighted degree heuristics [8] for hybrid backtrack search [9, 7]. In the original context, random probing allows a search algorithm to utilize the powerful strategy of choosing variables by their constraint weights from the very beginning of search. Here, the same strategy is used to select variables that are more likely to be problem bottlenecks on the assumption that SAC-based reasoning is more likely to find values that can be discarded. As in the search context, constraint weights can be considered to enhance effects obtained by choosing variables of high degree. Although this technique is much more elaborate than the original constraint weighting strategy, it is still fairly efficient even for a moderately large number of probes.

In extensive testing it was found that selection sets based on the k highest degree variables were significantly more effective than a random selection of k variables, usually by about 40%. More elaborate high-degree heuristics (second and third in the list above) did no better and sometimes did worse. However, with selection based on random probing, a further improvement of about 10% was obtained. Hence, in the main experiments only high degree and high degree-weight were used to find the selection set.

In addition, it was found that a value of k equal to a sizeable fraction of the variables in the problem had to be used to delete an appreciable number of values. With this in mind, a selection set size of 25 was used in the main experiments.

5 Experimental Methods

In the main experiments algorithms were tested on two types of structured problem. (Time constraints did not permit further types to be tested.) Problems of the first type were randomly generated binary CSPs with relational constraints of the form $X_i \; op \; X_j$. For half the constraints *op* was $\geq$; for half it was $\neq$. These problems were relatively difficult for MAC alone, but could be solved fairly efficiently when MAC was preceded by NSAC. Problems had 150 variables, domain size 20, and degree = 0.25. All problems had solutions. Here, the algorithms were partial NSAC-1 and the three forms of partial NSACQ described earlier. For full NSAC, the NSACQ algorithm was always used since it is clearly the best such algorithm [2, 3]. The MAC algorithm was MAC-3 (see [10] for reasons to choose this over later elaborations).

Problems of the second type were Radio Link Frequency Allocation Problems (RLFAPs). All were derived from the graph3 benchmark at the Université Artois website [1].

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

This problem has 200 variables, with domain sizes between 6 and 44 inclusive. (This problem has solutions and can be solved without backtracking or even retraction after AC preprocessing alone.) Alterations were done using the following procedure. For each problem, ten percent of the distance constraints in the graph3 problem of the form $|X_i - X_j| > k$ were chosen at random. For each constraint selected, an equiprobable decision was made to either increment or decrement the value of k. Then, starting with a base increment/decrement of 5, this value was either accepted with a probability of 1/2 or, if not, then the absolute value was increased by 1 and the same decision made, etc., until a value had been accepted. The k value for that constraint was then altered by that amount. Four thousand problems were generated in this way, and the 50 hardest problems with solutions and 50 without were selected for experimentation.

All algorithms were implemented in Lisp (using Xlispstat), and experiments were run under a Unix OS on a Dell Poweredge 4600 machine (1.8 GHz). All solutions obtained were checked for correctness. In all experiments, the minimum domain/weighted-degree heuristic was used during search to choose the next variable to assign a value. Due to time and space limitations, experiments are restricted to simple neighbourhood SAC, because of its efficiency and interesting local properties, which the present methods might be expected to enhance.

6 Experimental Results

6.1 Results for random relop problems

Table 1 shows results for the highest-degree heuristic, Table 2 for the highest-constraint-weight heuristic based on random probing. In earlier runs with the highest-degree heuristic, anomalies were found in the search results that can be ascribed to the pattern of failures causing domain reduction when using partial NSAC. This in turn weights constraints in a biased way, which can throw off the weighted degree heuristic used during search. Hence, in these tests constraint weights were not incremented during preprocessing. Although this problem does not apply to probe-based weights, in order to make meaningful comparisons the same convention was followed for the results in Table 2.

Table 1. Preprocessing and Search with Relop Problems: Hightest-Degree Heuristic

algorithm	del	nodes	time-pre	time-srch
full NSACQ	1321	6,255	159	120
pNSAC-1	868	10,553	24	233
pNSACQ0	141	8,450	32	181
pNSACQ-ac	868	12,258	21	280
pNSACQ-acn	868	10,553	21	231

Notes. Means for 50 problems. k = 25. pNASCQ0 is the simple version of pNSACQ; -ac includes an AC pass after deletion; -acn is the SAC-1 equivalent version.

For the highest-degree heuristic, while the partial NSAC algorithms are much faster than full NSAC, there is a considerable fall-off in search efficiency so total runtimes are similar. (Node counts are similar to MAC alone, which required 10K nodes.) In addition, partial NSACQ seems always to be slower in its simplest form, while the SAC-1 equivalent form does not clearly outperform partial NSAC-1.

Table 2. Preprocessing and Search with Relop Problems: Hightest-Constraint-Weight Heuristic

algorithm	del	nodes	time-pre	time-srch
pNSAC-1	934	4,274	21	82
pNSACQ0	166	2,702	28	52
pNSACQ-ac	916	4,839	17	93
pNSACQ-acn	933	2,889	20	59

Notes. Means for 50 problems. $k = 25$.

In contrast, when the highest-constraint-weight heuristic is used to obtain a selection set, the results are even better than full NSAC, so that search times are reduced along with NSAC-preprocessing times. Against these results is of course the cost of probing, which is not shown in the table. The mean time for each probe was about 3 sec (50 failures), so 100 probes required 300 sec on average. (Note that with this heuristic, the actual selection set can vary, which may account for the small difference in deletions between pNSAC-1 and pNSACQ-acn in Table 2.)

With these results, an obvious question is: can we get comparable results with fewer and shorter probes? To answer this, a test run was made with 50 probes, each with 20 failures. Time per probe was about 2 seconds. For comparison with the previous experiment with random probing, under one condition probe weights were not used at the beginning of search. Because the normal (and sensible) procedure would be to use them, this variation was also run. In these tests only pNSAC-1 and pNSACQ were used.

Table 3. Relops: Search w/o Probe Weights

algorithm	del	nodes	time-pre	time-sch
	probe weights not used in search			
pNSAC-1	906	4,493	21	85
pNSACQ0	163	4,225	30	86
	probe weights used in search			
pNSAC-1	920	1,914	21	30
pNSACQ0	161	2,217	29	39

Notes. Means for 50 problems. Highest-constraint-weight heuristic, $k = 25$.

Results are shown in Table 3. Two results stand out:

- One can obtain the benefits shown in the previous experiment with much more limited probing.
- Using probe weights from preprocessing enhances search efficiency beyond effects of effective consistency preprocessing.

Because of lower costs and additional enhancements, the mean total time per problem was reduced to 150 or 170 sec, depending on the algorithm. This is a clear improvement over results using the high-degree heuristic as well as full NSACQ. Also, search time was reduced by a factor of six to eight in comparison with the high-degree heuristic.

6.2 Results for Radio Frequency problems

In these experiments only partial NSAC-1 was used. For the highest-constraint-weight heuristic, the probing schedule was 50 probes with 20 failures per probe. For a fuller comparison, results for MAC alone and MAC following random probing using the same schedules are also shown (Table 4).

The results are similar to those in the previous section. With the highest-degree heuristic, many fewer values are deleted in comparison with full NSAC, and there is a considerable fall-off in search efficiency. Both effects are greatly ameliorated with the highest-constraint-weight heuristic. Interestingly, with this form of preprocessing it was possible to prove unsatisfiability in almost as many cases as with full NSAC. As a result, total runtime (including time for probing) is considerably reduced in comparison with MAC alone (although in this case the full NSAC algorithm does even better).

Table 4. Preprocessing and Search with
Radio Frequency Allocation Problems

algorithm	del	prove insol	nodes	time (NS)AC	time srch
		problems with solutions			
MAC only	343	–	5,564	0	104
MAC RP	343	–	1,504	0*	25
NSACQ	4549	–	618	188	5
pNSAC1/dg	820	–	2,025	25	32
pNSAC1/pb	1232	–	1,119	19*	17
		problems without solutions			
MAC only	341	0	8,040	0	244
MAC RP	341	0	4,351	0*	137
NSACQ	2085	40	556	71	10
pNSAC1/dg	1229	13	2,367	20	67
pNSAC1/pb	1592	37	877	7*	24

Notes. Means for 50 problems. k = 25. "*" additional time of 100 sec required for 50 probes.

Note that because NSAC preprocessing was able to prove unsatisfiability in many problems, the mean number of search nodes is for a different problem set than MAC. For full NSAC, mean nodes for the ten problems that required search with both algorithms, was 2781; for MAC alone it was 7640; for MAC with random probing it was 7721. Corresponding figures (13 problems) for partial NSAC with probing were 4383, 9801, and 10,555. The higher figures for random probing without NSAC were due to a few cases of greatly increased search effort compared to either of the others, although probing does improve on MAC with wdg in the typical case. It is also of interest that seven of the hardest problems for MAC ($>$ 10,000 search nodes) were proven unsatisfiable by both NSAC and pNSAC.

7 Conclusions

Partial (N)SAC has the potential to greatly enlarge the scope of application of SAC-based reasoning by allowing much larger problems to be handled with these methods. It can therefore overcome the tradeoff between effectiveness and efficiency that is a problem for SAC-based methods. The present experiments show that this approach can be effective, although care has to be taken when choosing the subset of variables for greater consistency processing.

An important discovery in this work is that random probing and partial SAC-based reasoning together form a powerful combination for improving search. This may happen for the following reasons. Random probing allows one to locate the chief bottlenecks of the problem and, to use Carla Gomes' felicitous phrase, thereby serves to "unlock the combinatorics of the problem". When SAC-based reasoning is applied to the variables located by probing, this removes extraneous values and reduces the branching factor right at the top of the search tree, thus avoiding many unproductive choices.

A related finding is that if one can locate these bottleneck variables, then one does not need a very large selection set to significantly improve the efficiency of subsequent search. This means that these methods are more likely to scale up than full (N)SAC.

This work opens up a large field for further exploration. The same methods can of course be used with any form of k-NSAC or SAC. As noted earlier, SAC-based reasoning can be applied to problems with n-ary constraints [3, 11], although it may be necessary to develop specialized heuristics for choosing the selection set. Finally, the efficiency of these new procedures may allow them to be used effectively during search as well as preprocessing.

References

1. Debruyne, R., Bessière, C.: Some practicable filtering techniques for the constraint satisfaction problem. In: Fifteenth International Joint Conference on Artifcial Intelligence – IJCAI'97. Vol. 1, Morgan Kaufmann (1997) 412–417
2. Wallace, R.J.: SAC and neighbourhood SAC. AI Communications **28** (2015) 345–364
3. Wallace, R.J.: Neighbourhood SAC: Extensions and new algorithms. AI Communications **29** (2016) 249–268
4. Berlandier, P.: Improving domain filtering using restricted path consistency. In: Conference on Artificial Intelligence for Applications - CAIA-95. (1995) 32–37
5. Wallace, R.J.: Preprocessing versus search processing for constraint satisfaction problems. In Bistarelli, S., Formisano, A., Maratea, M., eds.: 23rd RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion. (2016)
6. 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, AAAI Press (2015) 91–96

7. Wallace, R.J., Grimes, D.: Experimental studies of variable selection strategies based on constraint weights. Journal of Algorithms: Algorithms in Cognition, Informatics and Logic **63** (2008) 114–129
8. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proc. Sixteenth European Conference on Artificial Intelligence-ECAI'04, IOS (2004) 146–150
9. Grimes, D., Wallace, R.J.: Learning to identify global bottlenecks in constraint satisfaction search. In: Proc. Twentieth International FLAIRS Conference, AAAI Press (2007) 592–598
10. Wallace, R.J.: Complexity analysis vs. engineering design in CSP algorithms: Contravening conventional wisdom again. In Bistarelli, S., Formisano, A., Maratea, M., eds.: 23rd RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion. (2016)
11. Wallace, R.J.: Neighbourhood SAC for constraint satisfaction problems with non-binary constraints. In Markov, Z., Russell, I., eds.: Twenty-Ninth International Florida Artificial Intelligence Research Society Conference - FLAIRS-29, AAAI Press (2016) 162–165