Preprocessing versus Search Processing 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. A perennial problem in hybrid backtrack CSP search is how much local consistency processing should be done to achieve the best efficiency. This can be divided into two separate questions: (1) how much work should be done before the actual search begins, i.e. during preprocessing?, and (2) how much of the same processing should be interleaved with search? At present there are two leading approaches to establishing stronger consistencies than the basic arc consistency maintenance that is done in most solvers. On the one hand there are various kinds singleton arc consistency that can be used; on the other there are several variants of restricted path consistency. To date these have not been compared directly. The present work attempts to do this for a variety of problems, and in so doing, it also provides an empirical evaluation of the preprocessing versus search processing issue. Comparisons are made using the domain/degree and domain/weighted degree variable ordering heuristics. In general, it appears that preprocessing with higher levels of consistency followed by hybrid-AC processing (i.e. MAC) gives the best results, especially when the weighted degree heuristic is used. For problems with n-ary constraints, this difference seems to be even more pronounced. In some cases, higher levels of consistency maintenance established during preprocessing leads to performance gains over MAC of several orders of magnitude.

1 Introduction

Many combinatorial problems can be represented as networks of overlapping relations. These include problems in such areas as scheduling, configuration, and transportation as well as logic puzzles and problems in computer vision. Many of these problems can be viewed as labelling problems in which there is a set of variables, each of which must be assigned a label, and k-tuples of labellings must satisfy the stipulated relations among subsets of variables. (Labels are usually called values, so that is the term that will be used from now on.) In most cases, such problems can be represented with a simple scheme known as the constraint satisfaction problem, or CSP [5, 8].

All effective complete algorithms for solving CSPs rely on depth-first backtrack search where some form of *local consistency* is performed after each new assignment (and sometimes after retracting an assignment as well). These are called hybrid search algorithms. Local consistency refers to consistency with respect to the values assigned to specified subsets of variables, while global consistency refers to consistency with

respect to all relations that have been specified simultaneously. The former is used in order to reduce the number of possible values that need to be considered, since if a value cannot meet the requirement of local consistency then it cannot meet global consistency requirements either. Given this state of affairs, research has focused on two areas where performance can be improved by using alternative algorithms for local consistency. These are,

1. The use of stronger forms of local consistency when a problem is "preprocessed" before beginning the actual backtrack search,
2. The use of stronger forms of local consistency for interleaving with search assignments.

(There is also the important question of how to make certain heuristic decisions, but here this topic is finessed by using two heuristics for variable ordering that are known to be very effective for a variety of problems.)

At this time, the most popular form of hybrid search employs *arc consistency* (AC) both for preprocessing the problem and for interleaved local consistency. This procedure is called maintained arc consistency (MAC) [6], since this is the degree of consistency that is maintained throughout search. Arc consistency simply refers to the fact that for every possible value of every variable in the problem and for every relation that that variable is involved in, there is at least one set of values that satisfies that relations and which includes that value.

During the past several years alternative forms of consistency have been proposed, for either preprocessing or search (or both), some of which are very promising. Generally speaking, two main sorts of strategies have been considered. Their selling point is that algorithms in both groups are provably superior to AC in that they always delete as many or more values under the same conditions. But at the same time they are always more expensive to compute. Hence, hybrids based on these algorithms have not in general proven to be superior to MAC in practice, although counter-claims have been made and some are still 'on the table' (e.g. [7]).

The first group includes algorithms based on *path consistency* (PC). PC extends arc consistency by considering values in pairs of variables and determines whether this 2-tuple can be extended in a consistent fashion to any other variable. Although full path consistency has proven to be much too expensive, various forms of "restricted" path consistency have been proposed that efficient enough to be competitive. These are described in the next section.

The second group includes algorithms based on *singleton arc consistency* (SAC). The basic idea behind these algorithms is to reduce the set of possible labels associated with a variable to a singleton before establishing arc consistency. When this is done for each value in the problem, the resulting problem is singleton arc consistent. In addition to SAC, there are other forms of singleton arc consistency based on the neighbourhood of a variable. These are called neighbourhood SAC (NSAC), and more generally k-neighbourhood SAC (k-NSAC, defined below).

To my knowledge, there has been no serious attempt as yet to compare these various approaches, nor to carefully test the alternative strategies of enhancing preprocessing alone with a stronger local consistency algorithm versus trying to enhance search itself with stronger consistencies. The purpose of the present work is to contribute to this

analysis. Here, we consider MAC as well as other hybrid algorithms, combined with different levels of preprocessing. One group involves SAC-based methods, either full SAC or some form of neighbourhood SAC [10, 11]. In most cases either full SAC or the most reduced form of neighbourhood SAC was tested. In what follows, an algorithm that maintains full SAC will be referred to as MSAC, while an algorithm that maintains NSAC will be referred to as MNSAC.

For path consistency algorithms, based on both the literature and my own experience, two forms of restricted path consistency (RPC) were chosen, called maxRPC and (r)RPC3 [9, 7].

Here, we consider both binary problems, where all of the algorithm can be used, and problems with n-ary constraints (both random and structured, including problems with global constraints). Since to my knowledge there are no PC-based algorithms for the latter types of problems, this part of the study will necessarily be restricted to SAC-based algorithms. Extensions of the latter to handle problems with n-ary constraints have been described in earlier papers [11, 12].

The next section gives general background concepts and definitions, including a discussion of NSAC and an algorithm for establishing it. Section 3 gives brief descriptions of the algorithms used in this work. Section 4 describes methods of testing. Sections 5 and 6 give the experimental results for random binary and structured problems, respectively. Section 7 gives the results for non-binary problems, including problems with global constraints. Section 8 gives conclusions. (Some of the present results are drawn from an RCRA 2014 workshop paper on SAC and NSAC algorithms, although they were not included in a subsequent journal article.)

2 Background Concepts

A constraint satisfaction problem (CSP) is defined as a tuple, (X, D, C). X is a set $X_1, X_2 \ldots X_n$ of variables. D is composed of sets of values (called "domains"), such that domain D_i is associated with variable X_i. C represents the set of relations that must be satisfied. Each member of C is a tuple of the form (Y, R) called a "constraint". For a given constraint C_i, Y_i is the set of variables associated with it, which is called the "scope" of that constraint, and R_i is the relation. As usual, the latter is a subset of a Cartesian product, in this case the product of the domains of Y, the variables in the scope.

Solving a constraint satisfaction problem means finding one or more *solutions*; these are assignments of values to variables, which form a mapping so that each variable in the problem has an assigned value and all constraints are "satisfied" (i.e. for all C_i, the mapping from variables in Y_i to values in their respective domains is an element in relation R_i).

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

Singleton arc consistency, or SAC, is a form of AC in which the just-mentioned value a, for example, is considered the sole representative of the domain of X_i. If AC can be established for the problem under this condition, then it may be possible to find a solution containing this value. On the other hand, if AC cannot be established then there can be no such solution, 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.)

Neighbourhood SAC (NSAC) establishes SAC with respect to the neighbourhood of the variable whose domain is a singleton. Specifically, for each value in each domain, the subgraph formed by all constraints whose scopes are included in the neighbourhood of that variable (including the singleton variable itself) is made arc consistent. The notion of neighbourhood can be extended to the idea of a k-neighbourhood, where k is the maximum shortest path from the singleton variable to any of its k-neighbours. Thus the original notion of neighbourhood becomes the 1-neighbourhood, while if we also include all the neighbours of these neighbours we obtain the 2-neighbourhood, etc.

In binary CSPs, path consistency (PC) refers to the property that given a viable tuple (a, b) between variables X_i and X_j, then for any third variable X_k in the problem, there is a value c in the domain of the latter that will support both a and b. Restricted path consistency (RPC) is a form of consistency in which the aforementioned PC property holds whenever a value a in the domain of X_i has only one support in the domain of X_j [1]. Max restricted path consistency (maxRPC) holds if for every value in every domain, there is at least one path consistent value among its supports in every adjacent constraint [3]. Recently a form of RPC based on an AC-3 style algorithm has been described, together with a variant with a further restriction [7]. In the latter case, (called restricted-restricted path consistency or rRPC3) when elements (here representing constraints between variables) have to be added back onto a queue for further processing, one only adds back arcs adjacent to the variable whose domain was reduced. One does not also add arcs between neighbouring variables, which is necessary to obtain full RPC.

3 Description of Algorithms Used

3.1 General features

The algorithms used in these studies were based on AC, SAC [4], neighbourhood SAC [10], maxRPC [3], RPC3 and rRPC3 [7]. Excepting AC, which was always used in preprocessing and search, each of the other algorithms was used either for preprocessing alone or for search as well.

In this paper an algorithm called NSACQ is used for neighbourhood SAC; in previous work, this was found to be the most efficient NSAC algorithm across a variety of problem classes [10, 11]. (The following description applies to binary CSPs, although it can be readily extended to the more general case.) NSACQ uses a list (a queue) of variables, whose domains are considered in turn; for a given variable, X_i and its domain D_i, each value is tested to see if arc consistency can be established in its neighbourhood. If

it cannnot, then the value is deleted, and in addition any neighbours of X_i that are not currently on the queue are put back on. Unlike most SAC (or NSAC) algorithms, there is no pure "AC phase" following a SAC-based value removal. The idea is that if a deletion from the domain of X_i has any effect, it must affect the neighbours of X_i, and any effects elsewhere in the problem can only occur through effects on these neighbours.

For this paper, the SAC algorithm was of the same form as the NSACQ algorithm. The only difference is that when the queue is revised all variables in the problem are put back on rather than just the neighbours.

In practice, NSAC is preceded by a step in which arc consistency is established, although because of dominance this is not required to establish neighbourhood arc consistency. This is done to rapidly rule out problems in which AC is sufficient to prove unsatisfiability. It also eliminates values which are easily detected using a less expensive arc consistency algorithm.

The RPC algorithms used were based on descriptions in the literature, although they also involved adjustments to pre-existing MAC code. maxRPC was based on [3] and [9], although AC and PC processing were more thoroughly interleaved than was apparently the case in the latter. The RPC3 and rRPC3 followed the description and pseudocode shown in [7] as closely as possible. They therefore use the find-two-supports strategy to detect cases where restricted path consistency must be enforced (as in [3]). I.e. if a second support cannot be found then, RPC must be checked.

Maintained neighbourhood SAC or NSAC (here called MSAC and MNSAC, respectively) uses the same basic strategy as the MAC algorithms. In an initial pre-search phase, SAC or neighbourhood SAC is established. Then after each decision, i.e. after each assignment, this level of consistency is reestablished for the entire problem. This means carrying out SAC or NSAC on every value in the domains of variables that have not yet been assigned a value at that point in search.

Search with MNSAC (or MSAC) can be made more efficient thanks to the following property. (Since the proof is straightforward, it is skipped for brevity.)

Proposition 1. Given previous processing that established neighbourhood SAC (resp. SAC) in the problem, if an AC step following a (single) new assignment does not delete any values, then neither will NSAC (resp. SAC).

Two heuristics were used for variable selection during search: minimum domain over forward-degree (dom/fwd) and minimum domain over weighted-degree (dom/wtg). In both cases forward-degree refers to the number of constraints with as yest unassigned variables. Weighted-degree refers to weights accumulated due to domain wipeouts (reduction to zero values) during interleaved local consistency processing during search [2].

3.2 Algorithms for problems with n-ary constraints

While arc consistency algorithms can be readily extended to problems with n-ary constraints, the same is not true for path consistency algorithms. Hence, in this section we only consider the former. Previous work has shown how to extend neighbourhood SAC algorithms to problems with n-ary constraints [11, 12]. However, the range of problems tested was rather restricted, and the results were not that impressive.

Previous work has also shown that for any k, there are two variants of k-neighbourhood SAC algorithms. This difference depends on how one handles the general case of two or more variables in the k-neighbourhood that share a constraint that also includes non-neighbouring variables. The simplest approach is to ignore such constraints, keeping to the basic requirement that only constraints contained within the neighbourhood are considered. A more extended form of the algorithm is to consider such constraints, but only with respect to the neighbouring variables; thus, the constraint is projected on the neighbourhood sub-tuple. In [12] these were called the extended and restricted forms, respectively, of k-NSAC, and this terminology will be followed here. So, I will refer to them as k-NSACext and k-NSAClim.

This work also left an important question open. While both the extended and restricted forms of NSAC form a hierarchy, where k-NSACx is dominated by k+1-NSACx (x $\in$ {lim, ext}) with respect to values deleted, it wasn't immediatedly clear how the two hierarchies were related. As it turns out, it is easy to prove that they form a single hierarchy.

Proposition 2. For any value of k, k+1-NSAC dominates k-NSAC, unless both subgraphs include all the variables in the problem. This holds for both the extended and limited forms of k-NSAC. In addition, k-NSACext dominates k-NSAClim and k+1-NSAClim dominates k-NSACext. That is (except for the limiting condition just specified), if a value is removed by any form of k-NSAC then it will also be removed by any form that dominates it, but the converse does not hold.

Proof Sketch. Previously this relation was shown to hold among the different forms of k-NSAClim or k-NSACext, respectively, for $k = 1, 2, \ldots$. Obviously, for any given k, the extended form will dominate the limited form, since every constraint considered in the latter case is considered in the former but not vice versa. Now, consider the case of k-NSACext and k+1-NSAClim. Since the k+1 neighbourhood will include any constraint that is partly inside and partly outside the k-neighbourhood, then any restriction that occurs in the k form of NSAC because of this constraint will also hold in the k+1 form. □

4 Methods for Testing Search Algorithms

Tests of the different maintained consistency algorithms were made with a variety of test problems including both random CSPs and problems with various kinds of structure. 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).

4.1 Binary CSPs without definite structure

These were either homogeneous random problems or geometric problems with variable support, as described in [10]. Due to time constraints, in the latter case only variants of singleton arc consistency have been tested to date.

4.2 Binary CSPs with structure

These CSPs had constraints that were related to the magnitudes of the domain values. Problems of various types have been tested, two of which are reported here.

Problems of the first type were radio frequency allocation problems (RLFAPs). The values in the domains are numerical values ranging roughly between 15 and 800, although only a small subset of the range is included. These problems have numerical constraints of two types: $X_i - X_j > k$ and $X_i - X_j = k$ In other words, the difference (or distance) between certain pairs of variables must be either greater than some constant k, or equal to k. For equality constraints k is always 128; moreover, such constraints hold between successive variable pairs, i.e. between X_1 and X_2, between X_3 and X_4, etc.

Results presented here are based on problems called graph problems at the Université Artois website. [1] In one run four of these problems were used (called graph1-4 at the website). In another test, problems were generated by taking one of the benchmarks (graph1) and altering it by incrementing values of k in the constraint specifications. This allowed us to generate problems with the same structure but with increasing degrees of difficulty. A set of 25 problems based on the graph1 benchmark were generated with a base increment of 18; all had solutions. Search with MAC gave a mean number of search nodes of about 100,000.

Problems of the second type were random relop problems, where constraints were relational operators, and the constraint graph was random. In all cases some of the constraints were inequality constraints, which gives intractable problems. Again, due to time constraints, these experiments only involved variants of singleton arc consistency.

4.3 Nary CSPs with structure

There were three goals in these experiments:

- to run tests based on benchmark problems,
- to use problems with global constraints,
- to generate problem sets where global constraint parameters could be varied, and the number of problems could be arbitrarily large.

These goal were met by using benchmarks with binary constraints and adding global constraints to them. In doing this, one criterion was that the global constraints seemed to make sense in the context of the problem class. To this end, problem generators were built to add global constraints to an existing RLFAP. In addition to the number of constraints of each type to be added, the following parameters could be specified:

- Number of constraints of specified types.
- Range of constraint arities for each type.
- Values of parameters specific to a particular constraint, e.g. for at-least constraints, the proportion of values to be considered was specified by the user, while for among constraints, this and the proportion of possible domain values could be specified.

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

- The degree of acceptable overlap among the global constraints, specified as a range within the [0,1] interval.
- The benchmark(s) to use.
- The number of problems to generate from each benchmark.

In the experiment reported here, each problem had ten global constraints of the following types:

- 3 atmost constraints of arity 5
- 3 atleast constraints of arity 5
- 1 disjoint constraint of arity 10 (which includes the disjoint set)
- 3 among constraints of arity 5

Atmost constraints stipulate that at most k variables in the scope will have a specified value. Atleast constraints stipulate that at least k will have a specified value. Disjoint constraints stipulate a partition of two sub-scopes such that the two have no assignment in common. Among constraints stipulate that k assignments within the scope will be drawn from a specified set of possible values. In all cases, k was set to be $\leq 50\%$ of the scope. For among constraints the proportion of the union of domains of the scope variables to be used was also $\leq 50\%$.

Fifty problems with these specifications were generated from a single benchmark, the 200-variable RLFAP graph3 problem. (In its original form, this problem can be solved by MAC without backtracking.) In this sample there were 11 satisfiable and 39 unsatisfiable problems.

5 Results with Random Binary Problems

5.1 Homogeneous random problems

The results shown in Table 1, for the pure algorithms, are typical for this type of problem. When used for preprocessing, there was little or no change in the search space for these problems. At the same time, aside from maxRPC there was little or no change in overall runtimes.

For maintained algorithms, it was found that RPC variants were appreciably faster than algorithms based on neighbourhood SAC or full SAC. At the same time, they did effect a considerable reduction in the size of the search tree, although not as much as MNSAC. For rRPC, this was done with very little increase in time when compared with MAC.

MSAC reduced the search size quite dramatically. However, both it and MNSAC are considerably more expensive than either MAC or the RPC algorithms.

Table 1. Search Results for Homogeneous Random Problems

	dom/fwd		dom/wtdg	
algorithm	nodes	time	nodes	time
MAC	1621	4	1538	4
MAC/init xRPC	1621	12	1536	13
MAC/init RPC3	1621	4	1536	5
MAC/init rRPC3	1621	4	1542	5
MAC/init NSAC	1621	5	1540	5
MAC/init SAC	1621	5	1540	6
maxRPC	834	12	839	13
RPC3	973	11	968	11
rRPC3	983	5	983	5
MNSAC	751	38	775	37
MNSAC/init SAC	751	40	775	44
MSAC	83	51	83	56

Notes. <50,10,0.18,0.369> problems. Sample size 100. Mean search nodes and runtimes (sec).

When the weighted degree heuristic is used with RPC- or SAC-based algorithms, there is some question of which failures should be included in the weights, since for this implementation at least constraint failures could be due either to AC or to higher-order processing. For the results in the table, constraint weights were based on all failures.

5.2 Random problems with heterogeneous features

As noted before, these results are limited to SAC-based variants. Results for geometric problems with varying support are shown in Table 2. These are for the dom/fwd heuristic only; with weighted degree these problems become very easy, so that extra processing does not improve performance. It is possible that there are patterns of support for which effects can be shown, but this line of inquiry has not yet been pursued.

Table 2. Search Results for Random Geometric/ Varying-Support Problems

algorithm	search nodes	time	# solved
MAC	>70,105	668	292
MAC/init NSAC	>30,610	167	295
MAC/init SAC	>10,801	83	297
MNSACQ	>10,078	?	297
MNSACQ/init SAC	72	78	300
MSAC	69	171	300

Notes. 120-variable problems. Other parameters in text. dom/fwd-degree heuristic. 1-million node cutoff. Sample size 300. Mean search nodes and runtimes (sec).

These results show, first, that there are problems for which the extra processing required by MNSAC or MSAC can pay off both in number of nodes searched and overall runtime, i.e. there are problems for which the search-tree-size/runtime tradeoff can be finessed. Moreover, it is of interest that such results are found when some form of structure or non-randomness is introduced into the problem. At the same time, the results with various forms of MNSAC show that it is sometimes difficult to judge beforehand how much processing is required for substantial and reliable improvement.

6 Results with Structured Binary Problems

6.1 Radio frequency problems

Before reviewing the results, something needs to be said about the implementation of the algorithms. Since all of the RPC algorithms tested here used residues (last support found), an array was required to access this last support for a given value and constraint. Now, since RLFAPs have domains with about 40 values ranging from about 15 to 800, it is not practical (and often not feasible) to use the domain values themselves as indexes into last-arrays, as one can with other kinds of problems. In my implementation, an array was set up that allowed an index to be retrieved that was associated with a given domain value, which was the ordinal position of that value in the set of all possible domain values for that problem. This worked very efficiently during processing, adding little to the time required to retrieve a residue.

The first set of problems tested were the four benchmark problems from the graph series. Since these problems are so easy to solve, the only points of interest are the efficiency and effectiveness (values deleted) in the preprocessing step. The results are shown in Table 3. Times are for both preprocessing and search.

Table 3. Preprocessing Efficiency and Effectiveness with Radio Frequency Problems

algorithm	grph1		grph2		grph3		grph4	
	time	rem	time	rem	time	rem	time	rem
AC	2	0	15	0	3	340	15	776
maxRPC	7	0	28	0	23	790	63	1614
RPC3	2	0	22	0	3	380	22	1040
rRPC3	2	0	22	0	3	380	22	1040
NSAC	64	0	402	0	162	1064	1062	2046
2-NSAC	110	0	517	0	468	1220	2770	2730
3-NSAC	133	0	584	0	1395	1240	7964	2844
SAC	137	0	636	0	1920	1274	13520	2876

Notes. RLFAP graph problems. Times (sec) and values deleted.

Obviously, the RPC algorithms are much more efficient than the (N)SAC algorithms; however the latter are able to remove more values. Note also that there is a

similar but less dramatic tradeoff when we compare RPC3 with maxRPC. We, therefore, have an interesting tradeoff: for easy problems such as these, any RPC algorithm is preferable, but for much harder problems the greater effectiveness of SAC and NSAC algorithms may pay off by reducing search. Note also that with respect to the number of values deleted, the same dramatic differences hold even for NSAC.

A similar pattern of results was found for the small sample of hard RLFAPs. In this case preprocessing with the RPC algorithms did not remove any values, while both NSAC and SAC removed an average of 24 values. Overall times when MAC with weighted degree was used to find solutions were about 590 seconds on average for the RPC algorithms and 640 for the SAC-based algorithms.

6.2 Relop problems

Again, in this section only SAC-based strategies are discussed. The first set of problems tested had 100 variables, with domain size equal to 20; graph density was 0.25. Search results for the different algorithms are shown in Table 4. It can be seen that for the dom/fwd heuristic, these problems are quite difficult for MAC, and in this case using NSAC or SAC reduces the size of the search tree by three orders of magnitude. In this case runtimes also improve by an order of magnitude. It can also be seen that NSAC, used either as a preprocessing algorithm or as part of a maintained consistency algorithm, gives the best runtimes.

At the same time, with a more effective variable ordering heuristic, the performance of MAC is again improved to the extent that it becomes the most efficient algorithm.

Table 4. Search Results for Random Relop Problems

	dom/fwd		dom/wtdg	
algorithm	nodes	time	nodes	time
MAC	676,442	5948	656	11
MAC/init NSAC	6637	103	262	50
MAC/init SAC	732	698	–	–
MNSAC	265	129	106	198
MNSAC/init SAC	143	731	–	–
MSAC	101	503	100	1041

Notes. 100-variable relop problems based on $\geq$ and $\neq$ constraints. Sample size 100. Mean search nodes and runtimes (sec).

With larger problems of this type search with different forms of singleton arc consistency can result in marked improvements in performance even when using the dom/wdg heuristic. In this experiment, 150-variable problems were tested, again with a 50:50 mix of $\neq$ and $\geq$ constraints in which constraints and constraint-type were selected randomly. The domain size was always 20. Nine different densities were used, ranging from 0.1 to 0.5 in steps of 0.05. This spans a range that includes relatively easy problems with solutions through the critical complexity region and far enough into the unsatisfiable region that problems become easier to run to completion. The sample size at each density was 50.

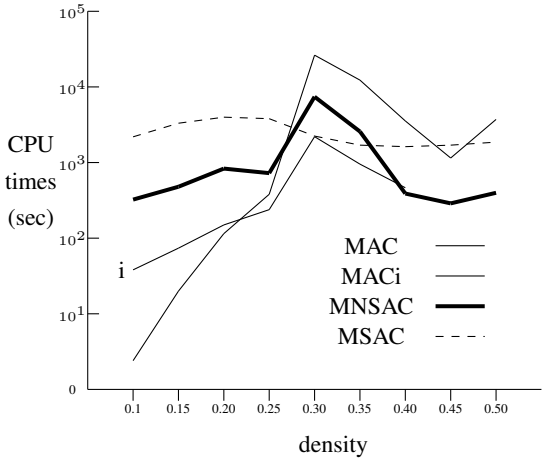

Fig. 1. Runtimes for 150-variable relop problems. Log scale on ordinate. MACi refers to MAC with initial NSAC; the curve is marked with an "i" on the left.

Table 5. Mean Search Nodes for 150-Variable Relop Problems

algorithm	density					
	0.10	0.20	0.30	0.35	0.40	0.50
MAC	236	3460	798293	391134	138115	37813
MNSAC	159	217	3071	404	4	0
MSAC	150	150	134	2	0	0
MACiNSAC	179	1473	67388	13594	234	0
MACiSAC	163	505	5575	591	0	0

Notes. Mean search nodes for separate sets of 50 problems.

The results are shown in Figure 1. For densities 0.1-0.25 all problems had solutions, and here MAC was the most efficient. But for the next density (41/50 problems had solutions), MAC was appreciably less efficient than either of the other two, and this was also true for density = 0.35 (no problems had solutions). For the very hardest problems (densities 0.3 and 0.35) MSAC was in fact the most efficient; however, outside this range it was much less efficient than the other two algorithms.

Table 5 shows that as problems of this type become more difficult, there are drastic differences in search tree size for MAC as opposed to SAC-based algorithms. It also shows that most unsatisfiable problems could be proven unsatisfiable during preprocessing when either SAC or NSAC was used.

However, it was found that by using NSAC for preprocessing search could be reduced enough to make this combination competitive with MNSAC or MSAC. Preprocessing with SAC was also tested, but while it led to a greater reduction in search nodes (Table 4), overall runtime increased by a factor of two to ten depending on the problem class.

7 Results for Problems with n-Ary Constraints

7.1 Random problems

Results for such problems were reported earlier in [11], so they will be described briefly here. In the cases tested, both SAC and higher levels of NSAC were often able to prove unsatisfiability when AC could not, thereby avoiding search. In other cases, the reduction in search effort was not large so overall there was only a small reduction in mean search nodes, while the times tended to be greater than with AC alone.

7.2 RLFAPs with global constraints

In the test runs, MGAC-3 was used to search for a solution. The minimum domain over forward degree heuristic was used for variable selection. Because of time limitations and the difficulty of these problems for some procedures, a 250,000 node limit was imposed

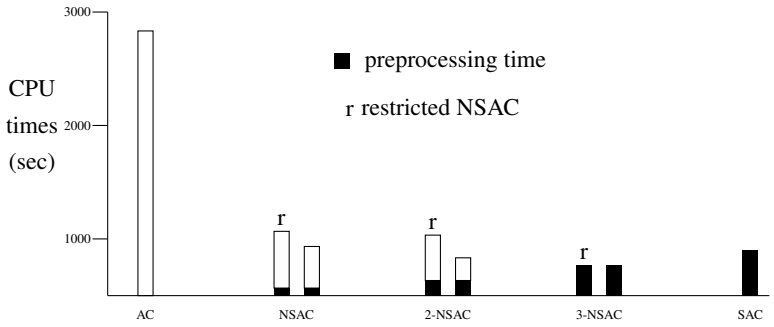

Fig. 2. Overall runtimes for preprocessing plus search with MGAC-3 on RLFAPs with global constraints. 0.25 M node limit.

Figure 2 shows mean runtimes across all 50 problems (including those for which the algorithm reached the node limit). Table 6 shows the number of unsatisfiable problems that could be proved unsatisfiable using the various kinds of preprocessing. Figure 3 shows the mean search nodes for the satisfiable subset. (Both forms of 3-NSAC and SAC gave means of 273 nodes for these problems, which is not visible in the graph.)

Table 6. RLFAPs with Global Constraints: Number Proved Unsat by Preprocessing

algorithm	#	algorithm	#
AC	11	2-NSAC-e	37
NSAC-r	28	3-NSAC-r	39
NSAC-e	30	3-NSAC-e	39
2-NSAC-r	37	SAC	39

Notes. Maximum = 39.

These results show that for problems with these characteristics, MGAC alone is not able to solve more than a fraction of them (16 out of 50). In contrast, most or all problems could be solved when MGAC was preceded by some form of SAC-based reasoning. However, only when preprocessing was carried out with 3-NSAC or full SAC was it possible to solve all of the problems. (Note that 3-NSAC was appreciable faster than full SAC, as shown in Figure 2.)

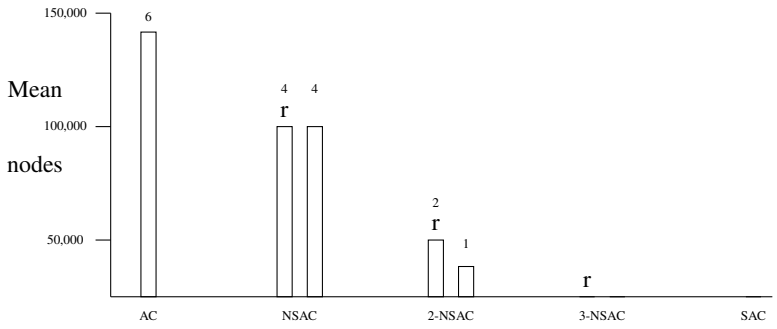

Fig. 3. RLFAPS with global constraints. Search nodes for 11 satisfiable problems using MGAC-3 after different types of preprocessing. Numbers above columns are cases where the 0.25 M node limit was reached.

Another significant finding was that the restricted form of k-NSAC was nearly always as good as the extended form. Implementing the latter, while not much more difficult, does involve revising code at a more intimate level than may be possible when one is adding this algorithm to an existing solver. Clearly, however, this is not necessary in most cases in order to obtain the benefits of the given level of consistency.

8 Conclusions

Although these results are preliminary, several patterns can be discerned that may be more general. For both random and structured problems, RPC-based algorithms are faster than SAC-based, sometimes appreciably so. At the same time, SAC-based methods are able to delete more domain values, thus effecting a greater reduction in the size of the search tree. Clearly, then, if the problems are fairly easy, RPC methods will be more effective overall, but for very difficult problems, SAC-based methods may be superior (although this has not yet been demonstrated clearly for binary problems).

Together, these results show that while MAC is a very powerful general-purpose algorithm, there are cases in which it is insufficient. This was found both for moderately large relop problems as well as problems with global constraints. It is, therefore, significant that these cases can sometimes be solved without inordinate expenditure when stronger forms of consistency are employed.

At this point, it appears that the best use of more powerful forms of consistency is during preprocessing. This was found in particular for the hard relop problems, and results consistent with this were also found for hard RLFAPs as well as RLFAPs that

included global constraints. However, more evidence is required before one can be confident that this is a general state of affairs.

Another noteworthy finding was that the variable ordering heuristic dom/wdg was often able to compensate for MAC's deficiencies in deleting values in comparison with the SAC-based methods. In many cases this occurred to such a degree that MAC, while being orders of magnitude less efficient than other methods when used in combination with a simpler, non-adaptive heuristic, again became the most efficient method. This shows that in discussing the benefits of lesser or greater degrees of consistency maintenance one must take account of the varying effectiveness of consistency strategies in relation to the current state of the problem. Nonetheless, use of this heuristic did not always result in adequate performance (e.g. hard relop problems), so that significant benefits still accrued from extra preprocessing.

References

1. P. Berlandier. Improving domain filtering using restricted path consistency. In *Conference on Artificial Intelligence for Applications - CAIA-95*, pp. 32–37.
2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Sixteenth European Conference on Artificial Intelligence-ECAI'04*, pp. 146–150.
3. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Principles and Practice of Constraint Programming-CP'97. LNCS No. 1330*, pp. 312–326.
4. R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Fifteenth International Joint Conference on Artifcial Intelligence – IJCAI'97. Vol. 1*, pp. 412–417.
5. R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
6. D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Principles and Pract. of Constraint Programming - PPCP'94*, LNCS No. 874, pp. 10–20.
7. K. Stergiou. Restricted path consistency revisited. In *Principles and Practice of Constraint Programming - CP 2015. LNCS. No. 9255*, pp. 419–428.
8. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
9. J. Vion and R. Debruyne. Light algorithms for maintaining max-RPC during search. In *Eighth Symposium on Abstraction, Reformulation, and Approximation - SARA2009*, pp. 167–174.
10. R. J. Wallace. SAC and neighbourhood SAC. *AI Communications*, 28(2):345–364, 2015.
11. R. J. Wallace. Neighbourhood SAC: Extensions and new algorithms. *AI Communications*, 29:249–268, 2016.
12. R. J. Wallace. Neighbourhood SAC for constraint satisfaction problems with non-binary constraints. In *Twenty-Ninth International FLAIRS Conference - FLAIRS-29*, pp. 162–165, 2016.