Complexity Analysis vs. Engineering Design in CSP Algorithms: Contravening Conventional Wisdom Again

Richard J. Wallace

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

Abstract. In this work we compare simple and 'advanced' algorithms that have been used to perform various functions in connection with constraint satisfaction problems (CSPs). These include algorithms for arc consistency, singleton arc consistency, arc consistency used with MAC, maxRPC algorithms, and algorithms for simple table reduction with non-binary constraints. In each case tested so far, we find tradeoffs between efficient implementations of basic operations like constraint checking and elaborations that allow a reduction in the number of these basic operations but add bookkeeping expenses. It is argued that such tradeoffs must be given more thorough consideration. This also suggests that there are interesting theoretical problems in this connection that have not yet received the attention they merit. Finally, the possibility that certain psychological biases have affected the analysis of algorithms in this area is considered.

1 Introduction

In the design of algorithms for constraint satisfaction, optimal time complexity has generally been taken as the gold standard for quality assessment. This has led to a history of attempts to improve various algorithms guided by the determination of what is optimal in each instance. This began with the development of optimal arc consistency algorithms [18]. Later, the same process occurred in connection with singleton arc consistency (SAC)[5], search with maintained arc consistency (MAC) [14, 17], restricted path consistency methods [1, 21], and generalized arc consistency for extensional constraints with simple table reduction (STR) [12].

From time to time there have been dissenting voices. A number of years ago, Wallace [22] questioned the then conventional wisdom that since AC-4 was optimal, it was, therefore, the algorithm of choice for establishing arc consistency. In the past decade van Dongen [20] questioned whether it was always better to avoid constraint checks while doing MAC in favour of other operations involving extra data structures. More recently, Wallace [23, 25] showed that advanced methods for SAC were often inefficient with larger problems in comparison with what he called "light-weight" methods such as SACQ and even SAC-1, despite the higher time complexities of the latter.

An immediate inspiration for the present work was the author's experience with SAC algorithms. He found that when he tried to implement the advanced algorithms, that were either optimal or nearly so by the usual standards, they turned out to be unwieldy, with poor scalability [23]. Even more telling has been the author's experience

with MAC-2001 as opposed to MAC-3, in connection with binary CSPs. Despite the fact that the former is often touted as an improvement over the latter, the present author found that his version ran about 50% slower than his version MAC-3. This was reminiscent of the findings of von Dongen [20] based on well-tuned C code; the latter author had found that MAC-2001 was 35% slower. After the experience of actually coding the algorithm, this came as no surprise. MAC-2001 uses a large array in which the last support found for each value and each constraint is stored. Constraint checking involves checking this array as well as evaluating actual tuples. Moreover, maintaining and restoring this data structure involves storing former last supports as well as indexes into the array, in order to replace a supporting value in the right cell of the array.

On the basis of these experiences, it seemed worthwhile to look more closely into the general problem of efficiency in these algorithms. Especially since other authors have reported results that contradict those described in the last paragraph. The present work is a collection of case studies involving the four algorithmic tasks mentioned in the first paragraph. In each case, it appears that there are significant tradeoffs between reducing the dominant operations (usually constraint checking) and increasing the amount of bookkeeping required to handle the more complex data structures that always seem necessary in these cases.

One conclusion from this work is that it is possible to use complexity analysis to obscure as well as to illuminate. It seems to me that the way forward is to carefully consider various implementations of these algorithms, in order to extract principles of good algorithm design. Obviously, this effort must be coordinated with complexity analysis. But when used in tandem, the result should be to clarify the latter.

The next section gives general background concepts and definitions. Section 3-6 discuss the various problems with purportedly optimal versions of AC, SAC, MAC, and STR, respectively. Section 7 discusses some lessons learned from this exercise, and suggests some ways to improve and systematize the design and analysis of algorithms in this field. Section 8 gives conclusions.

2 Background Concepts

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

In problems with binary constraints, arc consistency (AC) refers to the property that for every value a in the domain of variable X_i and for every constraint C_{ij} with X_i in its scope, there is at least one value b in the domain of X_j such that (a,b) satisfies that constraint. For non-binary, or n-ary, constraints generalized arc consistency (GAC) refers to the property that for every value a in the domain of variable X_i and for every constraint C_j with X_i in its scope, there is a valid tuple that includes a. One method for establishing GAC is called simple table reduction (STR) [19]. Instead of checking values against tuples to see if they are supported, this procedure builds up domains anew by considering viable tuples, since if a tuple is viable then each of its constituent values is supported.

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.

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 [3]. 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 [11].

Maintained arc consistency (MAC) is a complete algorithm for CSP search, that for any CSP either returns a solution or proves that the problem is unsatisfiable. It is an example of a hybrid algorithm that interleaves depth-first backtrack search with local consistency processing. Specifically, after a preprocessing step that establishes AC, MAC assigns variables in accordance with the basic backtracking algorithm, and after each assignment it re-establishes arc consistency. (There is also a variant that conducts search in a binary fashion, in which if an assignment is retracted, AC is done on a version of the problem with that value removed; here, we will limit the discussion to the variant where each value of a domain is tried in turn.)

All algorithms tested in this paper were coded in Common Lisp. Experiments were run in the XLispstat environment with a Unix OS on a Dell Poweredge 4600 machine (1.8 GHz).

3 The Case of AC

This is probably the earliest example of the phenomenon of interest in this paper. When AC-4 was first described in 1986, the authors were able to show that its $O(ed^2)$ time complexity, was optimal for establishing arc consistency. After that, many people in the field assumed that it had pre-empted earlier AC algorithms including AC-3. Hence, trying to improve the performance of AC-3 was pointless. However, in 1993 the present author was able to show with a simple statistical model that AC-3 outperformed AC-4 over a fairly wide range of contraint tightnesses in various problems [22]. There were two basic reasons for this:

1. AC-4 requires a first phase in which its data structures are initialized. To do this, it must carry out a complete evaluation of arc consistency for all values in the problem. Hence, it always requires a full ed^2 number of constraint checks.
2. AC-3's worst case complexity holds only for extreme cases in which each domain is reduced O(d) times, which requires that in each case all of its adjacent variables (minus one) are put back on the queue. In the aforementioned paper, the author

showed that even small perturbations of the worst-case conditions were sufficient to create problems where AC-3 was again superior to AC-4.

At about this time a new AC algorithm was proposed, called AC-6 [4]. Despite its cleverness, it didn't really provide a major performance boost, and it was difficult to incorporate into a hybrid search algorithm. As a result, it wasn't taken up by the community at large. (Another algorithm, AC-7, tries to avoid redundant constraint checks by maintaining dual supports. But it, too, is unwieldy and because of space and time limitations it will not be discussed in this paper.)

Faced with this situation, various workers in the field eventually decided to have another go at improving on AC-3. Interestingly, two independent groups of researchers hit upon the same trick at almost the same time. Therefore, around the year 2000 two papers appeared that described a variant of AC-3 that had optimal time complexity, called AC3.1 or AC2001 [9].

The basic idea behind the improvement was to store the first support found. Then, if a value had to be checked for support again in the course of consistency checking, one could retrieve the value checked before, and provided it was still viable, one did not have to perform more checks to find support. One therefore had to add a new data structure in which the last support found was stored. For this purpose, an array can be declared, but it must be large enough to hold a support for each value and constraint in the problem. So, just as with AC-4, elaborate data structures were used to reduce theoretical time complexity.

However, here as elsewhere there are tradeoffs, in this case in regard to the time required to carry out the procedure and establish arc consistency, Here, the basic tradeoff is associated with the assumption of current viability. To assess this, one must do some sort of test. Depending on the representation, the test can be done very quickly; however, the same is true for constraint checking, the operation one is trying to avoid. As a result, we are once again in the situation where it's not entirely clear whether we have made a genuine advance that will be significant in practice. (The use of this elaboration in connection with search is described in the next section.)

4 The Case of MAC

The last decade has seen two significant proposals for improving MAC-3 called MAC-2001 and MAC with residues (here called MAC-$x$res, where x is either 3 or 01[1]. MAC-2001 uses AC-2001 instead of AC-3. MAC-$x$res is also based on the idea of storing values that have been found to support a given domain value across a constraint. The key innovation is not to try to restore the last-support array to a previous condition. Instead, if the last-support is not in the current domain of the adjacent variable then the algorithm proceeds to carry out constraint checks. The last-supports are called residues, which gives the algorithm its name.

It's worth noting that the basic rationale for saving the last support is that one might have to perform lots of constraint checks otherwise. However, this doesn't take into

[1] Obviously, other AC algorithms than AC-2001 or AC-3 can be used with residues. In this paper I will confine the discussion to these two forms.

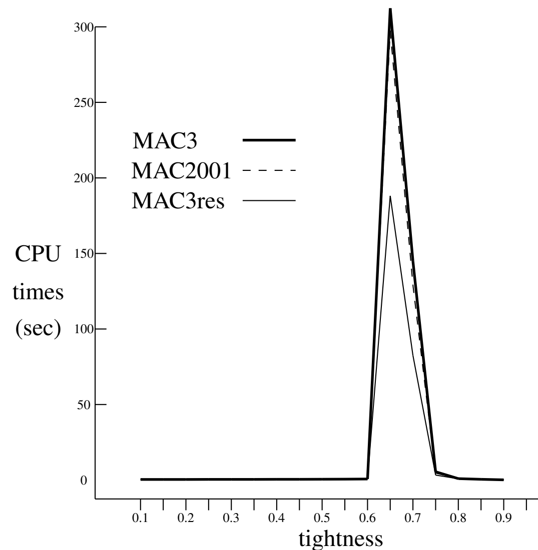

Fig. 1. Runtimes for three versions of MAC on random problems (constraint relations as lists).

account the possibility that the first support may be found early enough in the course of constraint checking that the extra work required to save last values, check them, and to maintain a last array may not be worth the trouble, especially if constraint checking can be done with great efficiency.

The problem with MAC-2001 is due to the fact, that unlike AC-2001, the last array needs to be maintained and restored constantly during search (specifically, whenever an assignment is retracted and a new value tried). Because the last array has three dimensions even when constraints are binary, these indexes must be stored in order to perform the necessary up-keep. Hence, storage and retrieval of indexes along with the actual replacement of last-values involves a massive amount of overhead.

Nonetheless, early in the last decade it was claimed, and the claim was generally accepted that MAC-2001 was an improvement over MAC-3. The one dissenting voice that I know of at that time (about ten years ago) was Marc van Dongen, who found that with an efficient version of MAC-3, the reduction in constraint checks afforded by MAC-2001 was not sufficient to overcome the ‘overhead problem’ [20]. As noted in the Introduction, this is consistent with my experience.

Nonetheless, there are published results that show that MAC-2001 is often better overall than MAC-3 [14]. So what is going on? Recently, I realized that a key factor in all of this may be the efficiency of constraint checking. In my implementation and in van Dongen’s, constraint checking for table constraints is carried out with incident arrays. In my case, I have a hash table of array addresses, which are accessed using the integer variable labels. (It is set up so that there are no collisions.) Then, the integer labels are used as indexes to a cell in the array whose value is TRUE or FALSE. Hence, constraint checking is very fast, i.e. very cheap.

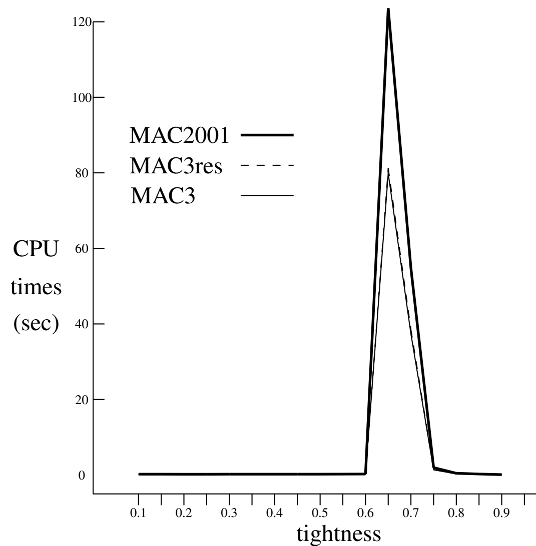

Fig. 2. Runtimes for three versions of MAC on random problems (constraint relations as incident arrays). Curves for MAC3 and MAC3res nearly coincide.

In order to evaluate this aspect of the algorithm, I implemented constraint checking in a different way. In this new version, the constraint tuples are kept in a list, so the data structure is a list of two-element lists. Constraint checking is done by composing the two labels into a list and then using the Lisp member function to determine if that list is in the constraint relations. Needless to say, this is much slower than the other method.

Using these two forms of constraint checking, the three algorithms, MAC-3, MAC-2001 and MAC-3res, were compared. The problems were homogeneous random binary CSPs with parameters $<100,20,0.05,t>$, where 100 is the number of variables, 20 the domain size, 0.05 the density of the constraint graph, while the constraint tightness t varied in steps of 0.05 between 0.1 and 0.9.

The results are shown in Figures 3 and 4. Note that when constraint checking is very efficient, MAC-3 easily outperforms MAC-2001, and there is almost no difference between MAC-3 and MAC-3res. On the other hand, when constraint checking is very inefficient, one finds the usual pattern of results reported in the literature: MAC-2001 is somewhat better than MAC-3, while there is a marked reduction in time with MAC-3res.

Now, I'm not claiming that other workers in the field employ a constraint checking process that is as inefficient as my list processing procedure. These results are meant to be a demonstration only. However, they are consistent with the pattern of results reported elsewhere, and as far as I can tell with the "conventional wisdom" of the field. In contrast, with very efficient constraint checking, the results (at least for MAC-3 and MAC-2001) are consistent with those reported in [20].

Table 1 shows the number of constraint checks and the runtime for the three algorithms, for the most difficult problems (t = 0.65). Obviously, the newer forms of MAC

are enormously effective in reducing constraint checks (and MAC-3res is in fact better than the "optimal" MAC-2001, as others have noted [14, 17]). Nonetheless, they are still equalled or outperformed by MAC-3.

Table 1. Times and Constraint Checks for MAC Algorithms on Random Problems with t=0.65 (Hardest Set)

algorithm	ccks	runtime
MAC-3	34,145,738	79.9
MAC-2001	20,741,750	123.6
MAC-3res	10,521,173	81.2

Notes. Constraint-checking with incident arrays. Means for 50 problems. Times in seconds.

5 The Case of SAC

In 1997 Debruyne and Bessière [11] proposed a new form of consistency that they called singleton arc consistency (SAC). In this form, every value in every domain can form an arc-consistent problem even if it is the only value in that domain. In order to establish this state, an AC algorithm is run repeatedly, each time with one domain fixed to a single value. If AC fails under these conditions, then that value cannot be part of any solution, so it can be discarded. This process continues until it has been established that every remaining value can support an arc-consistent problem.

What is intriguing about SAC is that it is still AC-based, which means that it rests on a very efficient underlying procedure. At the same time, it can remove many more values than an AC algorithm applied to the same problem. Of course, the number of repetitions of AC required makes a SAC algorithm very expensive overall, so there is a significant tradeoff problem between effectiveness and efficiency.

In the aforementioned paper, the authors described an algorithm with a simple repeat loop, now called SAC-1. In this algorithm, the full procedure (i.e. a complete run of SAC over all values in all domains) is repeated until no more values are deleted. It seemed likely that this could be improved upon, and in fact a series of algorithms have been proposed that are purported to give better performance, including an "optimal" SAC algorithm. These will be described briefly.

The SAC-2 algorithm [2] was inspired by the arc consistency algorithm AC-4 [18]. In both cases, the basic idea is to gather and store information about support during an initial processing phase (in the form of counters and "support lists"), to avoid performing redundant and irrelevant constraint checks while establishing local consistency throughout the problem. SAC-2 begins by running AC-4. Then, analogous to AC-4, there is a SAC initialization phase followed by a SAC pruning phase.

Recall that during the initial phase of AC-4, any domain values with zero support with respect to some constraint are removed and also added to a special no-support list. Then, in the pruning phase, by decrementing counters and adding support lists to the no-support list whenever a counter reaches zero, other domain values are removed until the no-support list is empty, at which point the problem is arc consistent.

In the SAC-2 SAC initialization phase, SAC support lists are built along with a list of assignments that need to be checked, which corresponds to the no-support list of AC-4. SAC support lists are constructed by placing each value that did *not* fail during the SAC-test (in which that value was made the sole member of its domain) on the SAC support list of every value in the remainder of the problem. If, on the contrary, the SAC-test fails, then an AC4-style pruning phase is carried out; the only difference from AC-4 is that if a value's support goes to zero, in addition to it being put on the AC no-support list, all its SAC-supports are put on the SAC no-support list. Following all this, in the SAC pruning phase, each value on the SAC no-support list is tested for SAC. During this stage, if a value fails in a SAC test, this leads to the same procedures as during SAC-initialization: AC-4-pruning and addition of all assignments on the support list of this value to the SAC-no-support list.

The SAC-SDS algorithm [6, 7] is a modified form of the authors' "optimal" SAC algorithm, SAC-Opt. The key idea of SAC-SDS (and SAC-Opt) is to represent each SAC reduction separately; consequently there are $n \times d$ problem representations (where n is the number of variables and d is the maximum domain size), each with one domain D_i reduced to a singleton. These are the "subproblems"; in addition there is a "master problem". If a SAC-test in a subproblem fails, then the value is deleted from the master problem and that problem is made arc consistent. If this leads to failure, the problem is inconsistent; otherwise, all values that were deleted in order to make the problem arc consistent are collected in order to update any subproblems that still contain those values. Along with this activity, the main list of assignments (the "pending list") is updated, so that any subproblem with a domain reduction is re-subjected to a SAC-test.

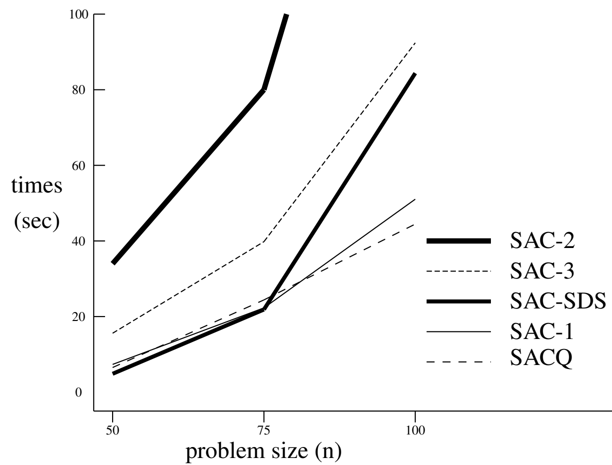

Fig. 3. Mean runtimes for SAC algorithms on homogeneous random problems of increasing size. Last segment of SAC-2 curve only shown up to 100-sec limit.

SAC-SDS also makes use of queues (here called "copy-queues"), one for each subproblem, composed of variables whose domains have been reduced. These are used to restrict SAC-based arc consistency in that subproblem, in that the AC-queue of the subproblem can be initialized to the neighbours of the variables in the copy queue. Copy queues themselves are initialized (at the beginning of the entire procedure) to the variable whose domain is a singleton. In addition, if a SAC-test leads to failure, the subproblem involved can be taken 'off-line' to avoid unnecessary processing. Subproblems need only be created and processed when the relevant assignment is taken from the pending list; moreover, once a subproblem is 'off-line' it will not appear on the pending list again, so a spurious reinstatement of the problem cannot occur.

The SAC-3 algorithm [13] uses a greedy strategy to eliminate some redundant checking done by SAC-1. The basic idea is to perform SAC tests in a cumulative series, i.e. to perform SAC with a given domain reduced to a single value, and if that succeeds to perform SAC with *an additional* domain reduced to a singleton, and so forth until a SAC-test fails. (This series is called a "branch" in the original paper.) The gain occurs because successive tests are done on problems already reduced during earlier SAC tests in the same series. However, a value can only be deleted during a SAC test if it is an unconditional failure , i.e. if this is the first test in a series. This strategy is carried out within the SAC-1 framework: successive phases in which all of the existing assignments are tested for SAC are repeated until there is no change to the problem.

Recently, another SAC algorithm was proposed in [24] called SACQ. SACQ uses an AC-3 style of processing at the top-level instead of the AC-1 style procedure that is often used with SAC algorithms. This means that there is a list (a queue) of variables, whose domains are considered in turn; in addition, if there is a SAC-based deletion of a value from the domain of X_i, then all values that are not currently on the queue are put back on. Unlike the other SAC algorithms, there is no "AC phase" following a SAC-based value removal.

SAC-1 and SACQ have been termed "light-weight" as opposed to "heavy-weight" SAC algorithms [23]. The former use relatively simple data structures, while the latter require elaborate data structures, as the descriptions above show. The question is, how well do these different kinds of approach scale as problems become larger? Here, I present some results taken from that paper.

The results of one experiment are shown in Figure 1. In this case, the problems were homogeneous random CSPs. Problems had either 50, 75 or 100 variables. Each point in the graphs is a mean of fifty problems.

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

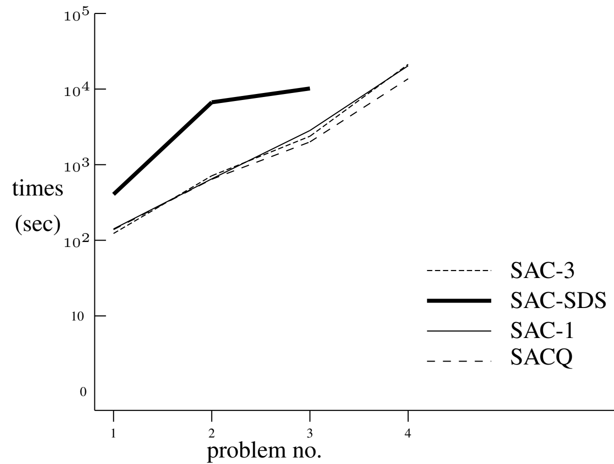

Fig. 4. Runtimes for SAC algorithms on RLFAP problems of increasing size and/or difficulty. Missing data point SAC-SDS is due to inability to finish the run (time $> 10^5$ sec). Note log scale on ordinate.

Results for a further set of tests are shown in Figure 2. These were Radio Link Frequency Allocation Problems (RLFAPs) taken from a set of standard benchmarks. (At the site these are called graph problems.) These four problems all had solutions. Problems 1 and 3 had 200 variables, problems 2 and 4 400. Problems 3 and 4 were versions of 1 and 2, that were made more difficult by reducing the size of a few domains.

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

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

From these experiments, I conclude that the light-weight algorithms scale much more adequately than the heavy-weight algorithms despite the higher time complexity of the former. This is true even for SAC-1 whose simple repeat-loop procedure is guaranteed to result in considerable inefficiencies with regard to constraint checking. Obviously, in this case the tradeoff between constraint checking and the extra bookkeeping that comes with more elaborate data structures is critical for overall performance. Again, a key factor may be the efficiency of constraint checking in my implementation.

6 The Case of MaxRPC

Path consistency (PC) was one of the earliest forms of local consistency investigated, having first been defined in the min-1970s by Montanari. The problem has always been that it is too inefficient to be used except as a preprocessing step, and in addition in contrast to AC it can alter the constraint graph.

About 20 years ago an interesting variation on path consistency was suggested by Berlandier [3], which he called restricted path consistency (RPC). In this variant, PC is only enforced when a value has a single support in an adjacent domain. Unfortunately, this means that one has to count supports, and unless one is using AC-4 this is an extra burden added to search effort.

A few years later, Debruyne and Bessière [10] suggested a way of finessing this problem: always look for two supports instead of one. Whenever a second support for a given value cannot be found, then this value must be tested for path consistency. On the other hand, if two supports are found, then one doesn't have to do this test.

In addition, these authors suggested a new form of RPC that they called maxRPC. In this case, one always establishes that a value has at least one support that is also path consistent; if no such support can be found, then the value can be discarded. This has been perhaps the most widely studied form of path consistency over the past two decades.

Subsequent to the introduction of residues in connection with MAC, Vion and Debruyne [21] introduced their use in connection with maxRPC. However, just as with SAC support-lists for SAC-2, using PC residues with an RPC algorithm entails humongous data structures and constant, complex checking and updating. So, again, it isn't clear that saving and checking residues of this sort will result in a more efficient algorithm.

In the present work, I look at three different forms of maxRPC.

- A version that doesn't use residues at all.
- A version that only saves AC residues.
- A version that saves both AC and PC residues.

In coding up these variants, I came up against a further issue. In the pseudocode given in [21], consistency processing has an AC phase followed by a PC phase. But to be complete, path consistency processing cannot be done independently of AC. The reason for this is that when a value is deleted because it violates the path consistency requirement, then the problem is no longer necessarily arc consistent. In this case, my version differs from theirs in that AC and PC processing are interleaved throughout.

Table 2. Times for Max-RPC Algorithms with/out Residues on Random Problems

	tightness				
algorithm	0.60	0.65	0.70	0.75	0.80
AC	0.02	0.02	0.02	0.02	0.05
maxRPC no residues	0.04	0.04	0.05	0.09	0.22
maxRPC AC residues	0.05	0.06	0.07	0.10	0.23
maxRPC AC,PC resid	0.08	0.08	0.09	0.13	0.32

Notes. Preprocessing times, in sec. 50 problems at each tightness value.

Some comparative results are shown in Table 2 for random problems with 100 variables, domain size 20, density about 0.03 and varying levels of tightness. Overall, these are impressive times, only a little slower than AC. (In contrast, the fastest form of SAC-based algorithm, NSACQ, requires about 1.5 sec on average with these problems.) The slower times for tightness = 0.80 occur because in this cases maxRPC (unlike AC) is able to prove 43 of the 50 problems unsatisfiable, thus avoiding search. Obviously, this involved more processing (reflected in the number of values deleted) since more elements had to be added to the AC queue.

However, the pertinent fact here is that saving residues doesn't save time, at least not with the present implementations. Moreover, when PC residues are used, the times increase, which must be due to the extra bookkeeping involved.

7 The Case of STR

Over the past five years, several improvements to the simple table reduction (STR) algorithm of Ullmann [19] have been suggested. The first was STR2 [12]; this algorithm avoids some constraint checks by excluding certain variables, but this involves membership tests, and therefore it does not permit a straightforward run through the variables in a constraint while checking a tuple. STR2w [15] attempts to improve best-case behavior by means of watched values. This avoids rebuilding domains in the fashion of STR and STR2, but involves extra data structures that must be maintained during search. Finally, STR3 [16] uses a radically different approach based entirely on watched values, but like some SAC algorithms this involves humongous data structures, which must be maintained during search. In addition, both STR2w and STR3 assume that the problem has already been made GAC-consistent, so preprocessing must be done with a different algorithm.

To date, I have coded STR2 and STR2w and subjected them to limited tests; so far I have not seen any great advantage, but it is too early to draw definite conclusions. Because of the complexity of the implementation, I have not finished coding STR3. However, at this point it is clear that again there are issues with regard to extra bookkeeping in the advanced methods that may compromise overall efficiency.

8 On Strategies for Algorithm Improvement

Needless to say, the point of the previous demonstrations is not to question time complexity analysis. Instead, the question is how to use such analyses to illuminate rather than obfuscate.

One way of looking at the problem, which is perhaps clearest in the case of MAC, is that one needs to consider additive versus multiplicative features of the algorithm. Additive components include both setting up basic data structures and, if search is involved, the initial preprocessing step. For example, in the MAC case study described above, the extra work in setting up an efficient structure for constraint checking has a tremendous effect on performance. And it doesn't figure at all in the time complexity analysis. But the significant fact is that this procedure changes the character of performance Most significantly, during the subsequent search it changes the relation between time to perform a constraint check and time to perform alternative tasks. Although this may still not affect asymptotic time-complexity, it will affect actual performance regardless of how large the problems are because the coefficient of proportionality is smaller.

Indeed, what often happens – as clearly shown with SAC algorithms – is that as problem size grows, the cost of bookkeeping increases more rapidly than the cost of basic operations like constraint checking, provided that the latter are carried out with reasonable efficiency.

Another important theme that can be discerned in this work (and which to my knowledge has not received the attention it merits) is the implicit versus explicit use of information by an algorithmic procedure. One of the most effective general strategies in designing algorithms seems to be: to design procedures that take advantage of information that is implicit in the way the problem is represented, the way that data structures have been set up, or even in the basic features of machine memory. Perhaps the best example I know of is the design of the basic backtracking algorithm. This is reflected in its description as "implicit enumeration". In this case one does not need any explicit information, such as counters or the like, to ensure that the algorithm covers the entire search space. Instead, this occurs automatically given the monotone character of unsatisfiability and the testing of variables and values in an orderly fashion.

It is also observable in comparisons between coarse-grained and fine-grained consistency algorithms. For example, in the coarse-grained AC-3 supports are not represented explicitly, while they are in AC-4. This leads to a very interesting tradeoff in that search time can be greatly reduced if supports are available (MAC-4), but the time to do this setting up can sometimes outweigh the gain during search even for fairly large problems (see [8]).

There is a third theme, which wasn't dealt with specifically in these demonstrations, but is worth mentioning here. This is the generality (or range of applicability) of algorithmic strategies. In the case of the SAC algorithms, it was found that the advanced methods could not be generalized to perform neighbourhood SAC without compromising some of their basic features (see [23, 25]. This is because they involve SAC-based assumptions, especially the assumption that the entire problem has been made arc consistent at each step, which do not hold in the case of neighbourhood SAC. As a result, it was found that alterations had to be made to ensure that they achieved the full level of consistency, and these alterations greatly compromised performance. In contrast the

simpler, 'light-weight' methods could be used without compromising their basic procedures.

9 Concluding remarks: On algorithms and narratives

Most papers on algorithmic efficiency in this field follow a fairly set strategy when they attempt to demonstrate the superiority of Algorithm X over Algorithm Y. First, complexity results are given; then tests are run using various problem classes. When, as usually happens, the experimental results are consistent with the complexity analysis, the author concludes that the superiority of Algorithm X has been demonstrated – both theoretically and empirically.

However, this appearance of consistency may sometimes be deceiving. Time complexity differences are, after all, asymptotic differences, and they therefore don't guarantee that a particular experiment will follow the asymptotic pattern. In fact, if one is off-loading operations in order to reduce what is normally the dominant operation (here, checking a tuple against the constraint relation), and the number of these operations also increases with increasing search effort, then it is not necessary for any experiment to match the analysis no matter how large or difficult the problems are.

Moreover, in this situation there is always a chance that confirmation bias is influencing the experimental results. This may have been the case for the alleged improvements to MAC-3, since as I have argued, they seem to depend on how efficiently one does constraint checking.

In these cases, we may be dealing with the general problem of distinguishing between carrying out a causal analysis versus building a narrative. Apparently because of deep-seated biases in the human thinking process, the tendency for narrative-building to creep into one's arguments is an ever-present problem. In particular, the *narrative closure* that is obtained with the usual presentation may blind one to any limitations and alternative explanations for the results.

Perhaps I don't have to say that, in sharp contrast to some fashionable contemporary exponents of science practice, I *do not* wish to see narratives take the place of careful causal analysis. What is necessary, perhaps, is the cultivation of a greater sensitivity to the possibility of such problems arising (even in an apparently cut-and-dry field like algorithmics!) and less complacency regarding the results of any particular experiment.

References

1. T. Balafoutis, A. Paparrizou, K. Stergiou, and T. Walsh. New algorithms for max restricted path consistency. *Constraints*, 16:372–406, 2011.
2. R. Bartak and R. Erben. A new algorithm for singleton arc consistency. In *Proc. Seventeenth International Florida Artificial Intelligence Research Society Conference - FLAIRS-17*, volume 1, pages 257–262, 2004.
3. P. Berlandier. Improving domain filtering using restricted path consistency. In *Proc. Conference on Artificial Intelligence for Applications - CAIA-95*, pages 32–37, 1995.
4. C. Bessière. Arc consistency and arc consistency again. *Artificial Intelligence*, 65:179–190, 1994.

5. C. Bessière, S. Cardon, R. Debruyne, and C. Lecoutre. Efficient algorithms for singleton arc consistency. *Constraints*, 16:25–53, 2011.
6. C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proc. Nineteenth International Joint Conference on Artificial Intelligence – IJCAI'05*, pages 54–59, 2005.
7. C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency and its extensions. *Artificial Intelligence*, 172:29–41, 2008.
8. C. Bessière, E. C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In *Fourteenth International Joint Conference on Artificial Intelligence - IJCAI'95*, pages 592–598, 1995.
9. C. Bessière, J.-C. Régin, R. H. C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165:165–185, 2005.
10. 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*, pages 312–326. Springer, 1997.
11. R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proc. Fifteenth International Joint Conference on Artifcial Intelligence – IJCAI'97. Vol. 1*, pages 412–417. Morgan Kaufmann, 1997.
12. C. Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints*, 16:341–371, 2011.
13. C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proc. Fourteenth International Joint Conference on Artificial Intelligence – IJCAI'05*, pages 199–204. Professional Book Center, 2005.
14. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proc. Twentieth International Joint Conference on Artificial Intelligence – IJCAI'07*, pages 125–130, 2007.
15. C. Lecoutre, C. Likitvivatanavong, and R. Yap. Improving the lower bound of simple tabular reduction. *Constraints*, 20(1):100–108, 2015.
16. C. Lecoutre, C. Likitvivatanavong, and R. Yap. A path-optimal filtering algorithm for table constraints. *Artificial Intelligence*, 220:1–27, 2015.
17. C. Likitvivatanavong, Y. Zhang, S. Shannon, J. Bowen, and E. C. Freuder. Arc consistency during search. In *Proc. Twentieth International Joint Conference on Artificial Intelligence – IJCAI'07*, pages 137–142, 2007.
18. R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
19. J. R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Sciences*, 177:3639–3678, 2007.
20. M. R. C. van Dongen. Saving support-checks does not always save time. *Artificial Intelligence Review*, 21:317–334, 2004.
21. J. Vion and R. Debruyne. Light algorithms for maintaining max-RPC during search. In *Proc. Eighth Symposium on Abstraction, Reformulation, and Approximation - SARA2009*, pages 167–174, 2009.
22. R. J. Wallace. Why AC-3 is almost always better than AC-4 for establishing consistency in CSPs. In *Proc. Thirteenth International Joint Conference on Artificial Intelligence - IJCAI'93*, volume 1, pages 239–245, 1993.
23. R. J. Wallace. Light-weight versus heavy-weight algorithms for SAC and neighbourhood SAC. In I. Russell and W. Eberle, editors, *Twenty-Eighth International Florida Artificial Intelligence Research Society Conference - FLAIRS-28*, pages 91–96. AAAI Press, 2015.
24. R. J. Wallace. SAC and neighbourhood SAC. *AI Communications*, 28:345–364, 2015.
25. R. J. Wallace. Neighbourhood SAC: Extensions and new algorithms. *AI Communications*, 29:249–268, 2016.