Arc Consistency in MAC: A New Perspective[⋆]

Chavalit Likitvivatanavong, Yuanlin Zhang,
James Bowen, and Eugene C. Freuder

Cork Constraint Computation Centre, University College Cork, Ireland
{chavalit, yzhang, j.bowen, e.freuder}@4c.ucc.ie

Abstract. AC refers to algorithms that enforce arc consistency on a constraint network while MAC refers to a backtracking search scheme where after each instantiation of a variable, arc consistency is maintained (or enforced) on the new network. In this paper, we use *mac* to denote *maintaining arc consistency* in MAC. In all existing studies, *mac* is simply taken as an associate of an AC algorithm. In this paper, we argue that it is worth taking *mac* as an entity separated from AC. Based on an observation that *mac* is invoked many times during a search, we propose three schemes to improve the efficiency of *mac*. First, the results of constraint checks are cached. Second, values remained in the auxiliary data structures used by sophisticated AC algorithms are better exploited. Third, we adopt a non-invariant ordering on domain values. Algorithms are also presented for these schemes. Their performances are discussed in terms of time complexity, space complexity, number of checks, and running time. In our experimental setting, we find that it is possible to design a *mac* algorithm which is simple to implement and runs faster as well as uses less space.

1 Introduction

AC refers to algorithms that enforce arc consistency on a constraint network while MAC refers to a backtracking search scheme where after each instantiation of a variable, arc consistency is maintained (or enforced) on the new network. MAC is regarded as one of the best search schemes not only by the researchers in the community, but also by the practitioners in the industry. It has been employed by many constraint solvers, for instance ILOG solver and CHOCO.

We will use *mac* to denote the *maintaining arc consistency* component of a MAC algorithm. It is taken for granted that to improve the efficiency of a MAC algorithm, one needs to focus on the AC algorithm employed by MAC; conversely, the effectiveness of AC algorithm is usually shown in the context of MAC. This reasoning works well and produces significant insight on the efficiency of AC algorithms. Unfortunately as a result, the larger context of MAC is completely ignored. In other words, AC is simply equated to *mac*. In this paper we argue that although *mac* is similar to AC, it deserves separate treatment.

⋆ This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075.

A number of AC algorithms have been designed since Waltz first proposed a scene filtering algorithm in [9]. Fundamentally, they fall into two classes: coarse-grained (e.g. AC3 [5], AC3.1 [10] and AC2001 [2]) and fine-grained (e.g. AC6 [1]). When a value is removed from a domain, algorithms in the former class will revise any affected domain with respect to the corresponding constraint, whereas algorithms in the latter class will revise only the affected values. There are algorithms in both classes which have optimal worst-case time complexity and perform well in empirical studies. Sabin and Freuder [7] proposed MAC in 1994 and it becomes so well accepted in the last decade by the community that to show the efficiency of a new AC algorithm, one has to test it in MAC. It has been shown that most efficiency improvements of AC algorithms could also lead to the improvements of MAC.

The most obvious difference between AC and *mac* is that an AC algorithm is executed only once, but due to backtracking and failed assignment of a variable, a *mac* algorithm may be executed many times (tens of thousands of times in a difficult random problem instance with 150 constraints and 50 variables, each of which has a domain with 30 values). Taking this massive number of executions into account, we propose three improvements for *mac*.

The fist idea involves caching the results of constraint checks. Unlike AC, for *mac* the same constraint checks may be repeated *enormous* amount times (in one of our experiments, for a problem that can be solved in 10 to 20 minutes, *mac* needs billions of checks whereas AC uses only millions.) Therefore, not only is it sensible to memorize the results for future use, it is *essential* for hard problems or the problems in which the cost of constraint checks is high.

Indeed, the number of constraint checks alone cannot be used to determine precisely the empirical performance of an algorithms due to the uncertainty on the cost of a constraint check. By using cache, we are able to deal with this uncertainty better in *mac*, making the impact of the cost of constraint checks more controllable. In our experiments, the performance of *mac*3 is significantly improved by using cache, even under reasonably cheap constraint checks. More details will be presented in Section 3.

The second idea involves passing information from one execution of *mac* to another. Most AC algorithms uses auxiliary data structures to speed up the execution time in standalone preprocessing but they cannot be used effectively in *mac*. The main problem lies in the rigid invariant associated with this structure and the total ordering of variable domains, which require expensive maintenance when backtrack. In section 4, we will present a method that relax the invariant and better exploit this auxiliary data structures. This also lead to the third idea which involves adaptive domain ordering, to be presented in Section 5. Section 6 describes related works. The paper is concluded in Section 7.

These issues will be studied using coarse-grained algorithms, specifically AC3 and AC3.1. The reason lies in their simplicity, and in the case of AC3.1, its optimal worst-case time complexity. Both algorithm are also empirically efficient.

2 Preliminaries

In this section we review AC3, AC3.1, and MAC. We introduce *mac* and experimental settings used in next few sections.

Definition 1 (Binary Constraint Network) A *binary constraint network* is a triplet $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$ is a finite set of variables, $\mathcal{D} = \{D_V \mid V \in \mathcal{V}\}$ where D_V is a finite set of possible values for V, and $\mathcal{C}$ is a finite set of constraints such that each $C_{XY} \in \mathcal{C}$ is a subset of $D_X \times D_Y$ indicating the compatible pairs of values for X and Y, where $X \neq Y$. Deciding whether $(a, b) \in C_{XY}$ is called a *constraint check*; this is sometimes written as a boolean function $C_{XY}(a, b)$. If $C_{XY} \in \mathcal{C}$, then $C_{YX} = \{(y, x) \mid (x, y) \in C_{XY}\}$ is also in $\mathcal{C}$.

For $(a, b) \in C_{XY}$, b is called a *support* of a in Y. If a value $a \in D_X$ has no support in Y where $C_{XY} \in \mathcal{C}$ then a is *invalid*. A constraint C_{XY} is *arc consistent* if every value $a \in D_X$ has a support in D_Y. A constraint network is *arc consistent* if all constraints are arc consistent. If a constraint network is not arc consistent, it can be made so by removing invalid values. Such an algorithm is called Arc Consistency (AC) algorithm.

Throughout this paper, we use n, e, and d to denote the number of variables, the number of constraints, and the maximum domain size in the network. We use D_V^0 to distinguish the *original domain* of V, while D_V denotes the *current domain*, which may change in the course of search. Deciding whether a value is in the current domain is called a *domain check*. A value in D_V^0 is *alive* if it is in D_V; otherwise it is said to be *pruned*.

Function $\texttt{succ}(b,\ D_Y)$ is a successor function defined in the usual sense. head and tail denote the start and the end of a domain, of which they are not members. succ(head) gives the first value while succ of the last value returns tail; if the domain is empty then succ(head) = tail. Function cirSucc ("circular successor") is defined as follows: $\texttt{cirSucc}(x) = \text{head}$ if $\texttt{succ}(x) = \text{tail}$; otherwise $\texttt{cirSucc}(x) = \texttt{succ}(x)$

2.1 AC3 and AC3.1

AC3 and AC3.1 are two representatives of coarse-grained algorithms. A basic operation a constraint *revision*, that is, removing invalid values in order to make the network arc consistent. The results are then *propagated* to other connecting variables, entailing more revisions. The generic algorithm AC() for coarse-grained algorithms is listed below. We also list AC3 and AC3.1 whose pseudo-code include only procedures different from the generic algorithm. In both algorithms, we assume a total ordering on each domain.

The difference between AC3 and AC3.1 lies in the routine hasSupport. AC3 finds a support from scratch while AC3.1 finds it by using the support found in the previous revision as a starting point. AC3.1 uses a data structure $\text{last}(X, a, Y)$ to remember the last support of a in D_Y, where $a \in D_X$.

AC

```
AC()
  ACInitialize()
  $Q \leftarrow \{(X, Y) \mid C_{XY} \in \mathcal{C}\}$
  return propagate($Q$)
propagate($Q$)
  while $Q \neq \emptyset$ do
      select and delete an arc $(X, Y)$ from $Q$
      if revise($X, Y$) then
          if $D_X = \emptyset$ then return failure
          $Q \leftarrow Q \cup \{(W, X) \mid C_{WX} \in \mathcal{C}, W \neq Y\}$
  return success
revise($X, Y$)
  delete $\leftarrow$ false
  foreach $a \in D_X$ do
      if not hasSupport($X, a, Y$) then
          remove $a$ from $D_X$
          delete $\leftarrow$ true
  return delete
hasSupport($X, a, Y$) {}
ACInitialize() {}
```

AC3

```
hasSupport($X, a, Y$)
  $b \leftarrow$ head
  while $b \leftarrow$ succ($b, D_Y$) and $b \neq$ tail do
      if $C_{XY}(a, b)$ then return true
  return false
```

AC3.1

```
ACInitialize()
  foreach $C_{XY} \in \mathcal{C}$ and $a \in D_X$ do last($X, a, Y$) $\leftarrow$ head
hasSupport($X, a, Y$)
  $b \leftarrow$ last($X, a, Y$)
  if $b \in D_Y$ then return true
  while $b \leftarrow$ succ($b, D_Y$) and $b \neq$ tail do
      if $C_{XY}(a, b)$ then
          last($X, a, Y$) $\leftarrow b$
          return true
  return false
```

The worst-case time complexity of AC3 is $O(ed^3)$ [5], while the worst-case complexity of AC3.1 is optimal at $O(ed^2)$. AC3 does not use any auxiliary data structure, whereas the space complexity of the auxiliary data structure of AC3.1 is $O(ed)$.

One way to evaluate the empirical performance of the AC algorithms is to count the number of constraint checks they conduct. However, an algorithm that has fewer number of constraint checks may consume more CPU time than another one with more checks. In our experiment, we also count the number of domain checks whose cost become more significant as the cost of a constraint check goes down.

2.2 MAC

MAC [7] is a backtracking search scheme (see Fig. 1) for finding a solution of a constraint network. Under this scheme, the network is preprocessed by an AC algorithm. During search, arc consistency is maintained (or enforced) after each instantiation of a variable in order to prune the search space. This process of maintaining arc consistency is denoted by *mac*; in the figure, V denotes the most recent assigned variable. Obviously, a key component of *mac* is AC, i.e. *mac* subsumes AC. In this paper, *mac* based on AC3 is called mac3, and *mac* based on AC3.1 called mac3.1. A generic pseudo-code for *mac* is listed below.

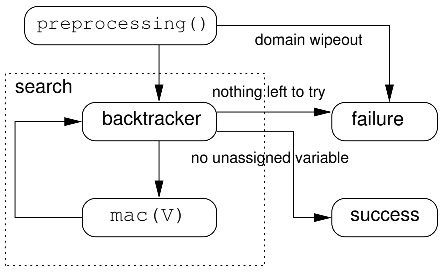

Fig. 1. MAC schema

The main difference between *mac* and AC is that AC is executed only once, but *mac* is executed whenever a variable is instantiated or when backtracking occurs. Given an AC algorithm, it is easy to design a *mac* algorithm based on it. For instance, mac3 is derived from AC3 and functions exactly like AC3. In the following sections we will add more functionalities that take advantage of the MAC environment.

mac

```
mac($V$)
  1 macInitialize()
  2 $Q \leftarrow \{(U,V) \mid C_{UV} \in \mathcal{C}\}$
  3 **return** propagate($Q$)
macInitialize() {}
```

2.3 Repeated Constraint Checks

During search in MAC, some constraint checked may be repeated even though an individual run of *mac* is optimal. We identify the following types of repeated constraint check during search:

- **positive repeat** A constraint check between a and b is performed, and as a result b supports a. Later in the search, during which period a and b remain in their respective domains at all time, (a, b) is checked again.
- **negative repeat** A constraint check between a and b is performed, and as a result b does *not* support a. Later in the search, during which period a and b remain in their respective domains at all time, (a, b) is checked again.

2.4 Experimental Settings

We use a backtracking algorithm that dynamically picks a variable with minimum domain to be instantiated first. Since both AC and *mac* involve frequent operations on a variable domain, we design a domain as a *random-accessed doubly linked-list*. Under this implementation, *the following operations take constant time*: (1) deleting a value, (2) checking whether a value is in the current domain (3) given a value, returning the next value in the current domain.

In the experiments reported in this paper we compare a number of algorithms against AC3, so to be fair we try to reduce as much as possible the amount of time needed to perform constraint checks, which dominates the overall running time of AC3. To this end, we use explicit constraint storage in our implementation. However, given wide-ranging applications of CSPs, it is better to make use of a function call that determines whether a given tuple is allowed by a constraint. As a result, the total cost of a constraint check in our implementation is the overhead of a function call plus the cost for memory lookup.

We use random problems to compare the experimental performance of our proposals, based on model B [3]. It is parameterized as (n, d, e, tightness) where tightness is the number of tuples *disallowed* by a constraint. We control the difficulty of the generated instances by varying the tightness. Results are averaged over 10 different instances. Since we observed a large variance on easy problems, all statistic on easy problems are averaged over three batches of executions, each containing 10 instances.

The algorithms are written in C++ and compiled with g++. Running time of specific routines are profiled using gprof. The experimental platform is Linux

2.4.20 running on a Dell PowerEdge 4600 which has two Intel Xeon 2.80GHz CPU's and 4GB of RAM.

3 Caching Constraint Checks

Our proposal relies the fact that *mac* will be called many times and thus the consistency of the same tuple may be checked repeatedly. Given the uncertainty of the cost of constraint checks, it is worth recording the result so that the next time the same check is requested it will be answered with little cost.

For simplicity, we cache the result of every possible constraint check. In a binary constraint network, the size of the cache is $O(ed^2)$ because there are e constraints and for each constraint there are at most d^2 tuples.

An immediate consequence of this approach is that the performance of *mac* with cache would be significantly improved even if each constraint check is moderately expensive. Another benefit of this approach is that we can now assume the constraint check is reasonably cheap because the cost of the management of the cache and the cost of the initial $O(ed^2)$ raw constraint checks are amortized over a large amount of repeated checks.

The cache idea is tested by using AC3. The reason for this choice of algorithm is that the physical CPU time of AC3 is good but the number of its constraint checks is usually several times of those of more sophisticated algorithms. Using cache may benefit AC3 the most. The new algorithm is named mac3cache and listed below.

```
mac3cache

preprocessing()
  1 AC3()
ACInitialize()
  2 foreach $C_{XY} \in \mathcal{C}$ and $a \in D_X$ and $b \in D_Y$ do  cache($C_{XY}, a, b$) $\leftarrow$ nil
hasSupport($X, a, Y$)
  3 $b \leftarrow$ head
  4 while $b \leftarrow$ succ($b, D_Y$) and $b \neq$ tail do
  5     if cache($C_{XY}, a, b$) = nil then
  6         cache($C_{XY}, a, b$) $\leftarrow C_{XY}(a, b)$
  7     if cache($C_{XY}, a, b$) then  return true
  8 return false
```

We design the following experiments in order to benchmark mac3 and mac3cache. The main purpose is to test how the cache performs in the case where a constraint check is extremely cheap (whose cost is the cost of function call + the cost of memory lookup, as mentioned in Section 2.4).

In implementing the cache, we try to make its access as fast as possible. When a constraint C_{XY} is revised, we first locate the cache area for C_{XY}, and before looking for a support for $a \in D_X$, we locate the cache area that stores the relationship between a and values in D_Y. In this way, when checking a against a value $b \in D_Y$, the value of b is used directly as an index to the cached content.

	mac3	mac3cache	MAC3	MAC3cache
P1 (easy problems)	0.56s	0.38s	0.62s	0.48s
P2 (hard problems)	382s	293s	474s	396s

Table 1. mac3 vs mac3cache. P1=$(50, 30, 150, 560)$. P2=$(50, 30, 150, 580)$. The number of constraint checks for P1 and P2 are 11.0M and 5.35B respectively.

From the experiment, we see that, although the constraint check is very cheap already, by using cache we are able to speed up mac3 by 30% for hard problems. This result implies that mac3 is very sensitive to the cost of constraint checks.

The idea of caching results of constraint checks is applicable to *mac* derived from most AC algorithms, with the exception of AC4, which builds complex data structures during its initialization phase and does not need to do more constraint checks afterward.

4 Exploiting the Residues

AC3 is one of the algorithms that simply revise a constraint without using any historical memory. Getting mac3 from from AC3 is straightforward. For algorithms that use auxiliary data structures however, there are two conventional methods. In this section we focus only on AC3.1.

The first approach involves re-initializing the structure and establishing supports from scratch every time the algorithm is invoked, in the same fashion as mac3. Even though the value remained in the auxiliary structure is not exploited to its full potential, this approache is widely used due to its simplicity and low overhead.

The second approach takes advantage of the value in the last structure carried over from the previous execution. Since the search for support proceeds only in one direction, in order for the algorithm to be correct we need to record all the past supports during search, so that we can start from the exact same state when backtrack. We will call this algorithm mac3.1. It is observed that the worst-case space complexity is not the same as AC3.1, which is $O(ed^2)$, but a larger $O(ed \min(n, d)$ [8]. A trace of the algorithm is given in the following example; it shows that mac3.1 cannot avoid both positive and negative repeat.

Example 1. Assume there are two neighboring variables X and Y, where $a \in D_X$, $D_Y = \{x, y, z, u, v\}$ ordered from left to right, and $C_{XY} = \{(a, y), (a, u)\}$. During an execution of mac3.1 (a, y) is checked and last(X,a,Y) becomes $[y]$ (the structure last is a now stack in which the top is the left-most value). Later, suppose that y is removed from D_Y; thus a new support for a must be found.

Consequently, (a, z) and (a, u) are checked and last(X,a,Y) becomes $[u, y]$. Now suppose that backtracking occurs and y is restored. As a result last(X,a,Y) is rolled back to its previous state of $[y]$. Suppose the process repeats: y is again removed and a new support for a must be found. (a, z) and (a, u) would be checked more than once.

4.1 Flexible Domain Search

The major problem with mac3.1 lie in its overhead in maintaining the auxiliary structure. This is necessary because the search always proceeds from the last support found. We can avoid this cost simply by restarting the search from the beginning again; however, this comes at the expense of optimality, since a constraint may be revised many times. We call this algorithm mac3.1residue. Note that in the pseudo-code we change the term from last to support to indicate that no invariant on its position is assumed.

Figure 2(i) shows the search for support from Example 1: suppose that the last support of a is y and that it is no longer available. The search would restart from the beginning until the next support (u) is reached. If u is later deleted then the entire domain must be searched. Note that this is conceptually the same as having a *circular* domain, although in practice it is easier to restart the search and have a normal domain.

mac3.1residue takes $O(ed^3)$ constraint checks in the worst-case while occupying $O(ed)$ space. It can avoid positive repeat but not negative repeat.

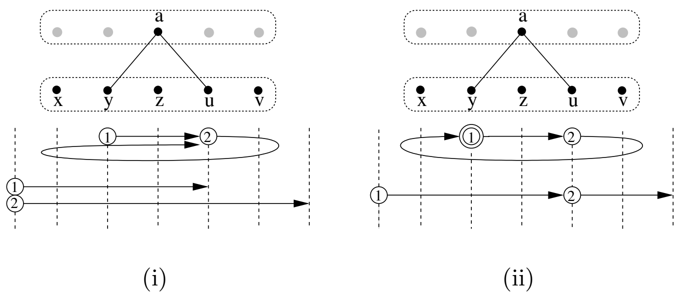

Fig. 2. Searching for support of a.

4.2 Optimal Worst-Case Flexible Domain Search

To make mac3.1residue optimal in the worst-case, we mark the first value that is checked when the constraint is revised for the first time in order to tell the search to stop as soon as this point is reached. This value, called start in the mac3.1residue

```
preprocessing()
  1 AC3.1()
hasSupport($X, a, Y$)
  2 $b \leftarrow$ support($X, a, Y$)
  3 if $b \in D_Y$ then return true
  4 $b \leftarrow$ head
  5 while $b \leftarrow$ succ($b, D_Y$) and $b \neq$ tail do
  6     if $C_{XY}(a, b)$ then
  7         support($X, a, Y$) $\leftarrow b$
  8         return true
  9 return false
```

pseudo-code, is initialized each time *mac* is called and it does not need to be reset when backtrack.

Figure 2(ii) shows how the search for support progresses. As in the case of mac3.1residue, we can implement this as a circular domain or starting out from the beginning, as shown in the bottom of Figure 2(ii). We choose the circular domain implementation because the supports found are more robust. Indeed, during search, *the deeper the level in which a support is found, the more robust it is* (also observed in [4].) This is due to fact a support found at a deeper level usually stays on in the current domain even after backtrack.

We call this algorithm mac3.1resOpt. It uses $O(ed)$ space in the worst-case. Like mac3.1residue, it can avoid positive repeat but not negative repeat.

mac3.1resOpt

```
preprocessing()
  1 AC3.1()
macInitialization()
  2 foreach $C_{XY} \in \mathcal{C}$ and $a \in D_X$ and $b \in D_Y$ do start($X, a, Y$) $\leftarrow$ support($X, a, Y$)
hasSupport($X, a, Y$)
  3 $b \leftarrow$ support($X, a, Y$)
  4 if $b \in D_Y$ then return true
  5 while $b \leftarrow$ cirSucc($b, D_Y^0$) and $b \in D_Y$ and $b \neq$ start($X, a, Y$) do
  6     if $C_{XY}(a, b)$ then
  7         support($X, a, Y$) $\leftarrow b$
  8         return true
  9 return false
```

The pseudo-code for mac3.1resOpt is presented in such a way that the idea is made as clear as possible; this should not be taken as the real implementation, especially the routine `macInitialization()`, in which each component of the structure start is reset. In fact we initialize start in a lazy way. Moreover, we only iterate through values in the current domain, rather than the original domain shown in line 6 of `hasSupport`. Using current domain involves more complex terminating condition since value of $\mathsf{start}(X, a, Y)$ may not be in the current domain, thus rendering the condition $b \neq \mathsf{start}(X, a, Y)$ incorrect.

4.3 Experimental Results

The empirical performance of mac3.1, mac3.1residue, and mac3.1resOpt are shown in the following table. The running time for MAC includes that of its corresponding *mac*, e.g. for P1 the running time for MAC3.1 = 0.75s, 0.66s of which is the time taken up by mac3.1. There are no extra checks in MAC beyond *mac*.

	P1 (easy problems)		P2(hard problems)		P3(over-constrained)	
	#checks	time	#checks	time	#checks	time
mac3	11.0M	0.56s	5.35B	398s	10.36B	713s
mac3.1	7.3M	0.66s	3.46B	519s	6.38B	1185s
mac3.1resOpt	2.9M	0.55s	1.43B	472s	2.69B	668s
mac3.1residue	7.1M	0.46s	3.30B	310s	6.14B	465s
MAC3	-	0.62s	-	493s	-	848s
MAC3.1	-	0.75s	-	615s	-	1314s
MAC3.1resOpt	-	0.65s	-	561s	-	796s
MAC3.1residue	-	0.56s	-	400s	-	590s

Table 2. Performance of various algorithms. #checks = #constraint checks + #domain checks. P1 = $(50, 30, 150, 560)$. P2 = $(50, 30, 150, 580)$. The time for mac3.1 includes that for restoring the auxiliary data structure `last`.

From the table, it is not surprising to see both mac3.1residue and mac3.1resOpt do better than mac3.1 in all three categories, due to the absence of overhead in maintaining the auxiliary data structure. However, we observe two unexpected results. The first is that the non-optimal algorithm mac3.1residue conducts even fewer checks than the optimal mac3.1; it is worth emphasizing that optimality is a property of a single execution of arc consistency algorithm, whereas the data gathered in this experiments is accumulated over the entire course of solving a problem. The other is that mac3.1resOpt uses less than half the number of checks performed by mac3.1. This can be explained by the robustness of the residue.

mac3.1 is the slowest algorithm in the table. There are a few possible reasons. One is that the most frequent operations like value accessing and constraint checks are reasonably cheap (and thus the saving on checks does not compensate the cost). Another is that when we maintain its auxiliary data structures, we use a library class `stack`. The implementation could be made more efficient. There may exist more efficient ways to maintain `support` incrementally. However, the running time is very unlikely to be lower than those of *mac*using residues because of the larger number of checks.

mac3.1resOpt runs much slower than mac3.1residue although it conducts significantly less number of checks. The code at the innermost loop of our implementation of mac3.1resOpt is several times longer than that of mac3.1residue as a result of complex terminating condition remarked previously. The experiments show that the saving of checks does not compensate for the cost of the extra instructions in mac3.1resOpt. On the other hand, mac3.1residue is faster than mac3 because it takes advantage of the `last` residue without incurring the heavy overhead associated with mac3.1. In fact, the code of routine `hasSupport` for mac3.1residue is only a few more instructions than that of mac3.

5 Adaptive Domain Ordering

Some AC algorithms like AC3.1 demand an ordering on the values of the domains of variables. The key idea behind is to find a support of a value a of D_X with respect to a constraint C_{XY} by going through the values of the domain D_Y only once. For this purpose, an ordering of the values is needed so that we never make the same constraint check twice when looking for a support for a during the execution of AC3.1. One implementation of AC3.1 in *mac* could assume a total ordering on the values of the domain of variables during the *whole* search procedure. Since the values in a domain are removed in an arbitrary order, it may be time-consuming to keep to the order when these values are restored.

As mentioned before, we implement a domain as a doubly linked list. When we put back a removed value back to a domain, to keep the correct ordering of the values, we need to go through the list and insert it in the correct position.

In this section, we use mac3.1resOpt as the basic algorithm. Our observation is that to keep the optimality of mac3.1resOpt, it is only necessary to guarantee the ordering of values in a domain in a single execution of *mac*. It implies that when we restore a domain at backtracking or failure of assignment, we can simply append the pruned values to the linked list. The additional benefit is that all negative repeat can be avoided.

We test the idea on random problems. In the following table, mac-order is mac3.1resOpt where the ordering of values are kept during the whole search process by using `restoreDomain-order`, and mac-tail is mac3.1resOpt in which pruned values are appended to the end of the domains by `restoreDomain-tail`.

Even though appending the removed values to the end of a domain is 20% better in term of runing time than inserting the removed values to a domain according to the ordering, the performance of mac-tail is, surprisingly, inferior to that of mac-order. One explanation is that mac-order is optimized over the entire period of search, which can not be done for mac-tail since the ordering on domain values is different from one execution to another.

6 Related Works

In boolean satisfiability problem, the time needed to find the clause suitable for unit propagation is recognized to be the most expensive part. In [6] the authors

	P1 (easy problems)		P2(hard problems)	
	#checks	time	#checks	time
restoreDomain-order	-	0.10s	-	92s
restoreDomain-tail	-	0.06s	-	77s
mac-order	2.88M	0.55s	1.43B	494s
mac-tail	2.87M	0.70s	1.42B	525s
MAC-order	-	0.65s	-	588s
MAC-tail	-	0.77s	-	605s

Table 3. Performance of adaptive domain algorithm.

suggest that the solver keep track of two literals (called *watched literals*) so that a single unvalued literal could be detected quickly.

Watched literals and the variants of *mac* presented here have some important features in common. First, the search direction is not fixed. Second, nothing needs to be restored upon backtracking.

7 Conclusions

In this paper we propose studying *mac* and AC algorithms separately. Given the observation that *mac* will be executed many times, we have presented a few strategies that improve the efficiency of *mac* algorithms. They are analyzed in terms of time complexity, space complexity, number of checks (including constraint checks and domain checks), and running time.

First we study *mac* that caches the results of constraint checks. This proposal applies to most *mac* except those derived from AC4. On the one hand, the cache makes the cost of a constraint check almost constant over long period. Space permitting, most *mac* algorithms could be equipped with a cache. This would make the comparison of the performance of different *mac* algorithms more meaningful. The space complexity of the cache is the same as that of *mac* derived from an optimal AC algorithm. One the other hand, the cache also speeds up the running time of *mac*. In our experiment on hard problems, mac3cache saves 20% to 30% running time even under cheap constraint checks. On hard problems, mac3cache is the fastest among all algorithms reported in this paper. Given sufficient RAM, mac3cache is a good choice given its efficiency and simplicity.

For *mac* based on AC algorithms that use auxiliary data structures, we propose two schemes that reuse the residual value from previous execution: one keeps the worst-case running time optimal while the other concerns only about simplicity. Specifically, we present algorithm mac3.1resOpt using the former scheme and mac3.1residue using the latter. The conventional *mac* based on AC3.1 takes $O(ed\min(n,d))$ worst-case space complexity while mac3.1resOpt and mac3.1residue takes only $O(ed)$. Moreover, both algorithms do not maintain the last structure, thus avoiding the costly overhead suffered by mac3.1. An unexpected discovery through our experiment is that the robustness of residue play a major role in the efficiency of *mac*. The number of checks performed by mac3.1residue and mac3.1resOpt are significantly lower than those of mac3 and mac3.1 respectively (mac3.1residue performs even fewer checks than mac3.1). It is worth emphasizing that although mac3.1residue has a worst-case complexity of $O(ed^3)$, it is the fastest among *mac* algorithms that have $O(ed)$ space complexity. Moreover, mac3.1residue is just as easy to implement as mac3. In retrospect, mac3.1residue can also be understood as a *mac* algorithm that uses a much smaller cache, recording only a single support for each value with respect to affiliated constraints.

We could apply the same idea to non-binary constraint networks. AC3.2 [4] generalizes AC3.1 to address non-binary constraints and takes advantage of positive multi-directionality by setting the current support found, which is a tuple for non-binary CSPs, as an *external support* for all other values in that tuple. This requires an extra storage apart from last because if it were to be used for this purpose then some values could be overlooked due to the fixed direction and range for support search. By using circular domain last can be used to store external support as well. It remains to be seen how this approach compares to AC3.2 and AC3.3, which counts all external supports and requires maintenance.

Finally we introduce adaptive domain ordering for *mac*, which in theory can avoid all negative repeat. In practice however, our experiment shows that *mac* that uses adaptive domain is worse off than that one that uses ordinary domain.

The lesson learned here is that consistency processing in the context of search deserved to be studied separately from its standalone version.

References

1. Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *Proceedings of AAAI-93*, pages 108–113, Washington, DC, USA, 1993.
2. Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI-01*, pages 309–315, Seattle, Washington, 2001.
3. Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. Random constraint satisfaction: Flaws and structure. *Constraints*, 6(4):345–372, 2001.
4. Christophe Lecoutre, Frédéric Boussemart, and Fred Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP-03*, pages 480–494, Kinsale, Ireland, 2003.
5. Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
6. M. Moskewisz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *The 39th Design Automation Conference*, 2001.
7. Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of ECAI-94*, pages 125–129, Amsterdam, The Netherlands, 1994.
8. M. R. C. van Dongen. To avoid repeating checks does not always save time. In *Proceedings of AICS'2003: The 14th Irish Artificial Intelligence and Cognitive Science*, Dublin, Ireland, 2003.

9. D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report MAC-AI-TR-271, MIT, Cambridge, MA, 1972.
10. Yuanlin Zhang and Roland H. C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of IJCAI-01*, pages 316–321, Seattle, Washington, 2001.