A Study of Residual Supports in Arc Consistency

Christophe Lecoutre and **Fred Hemery**
CRIL−CNRS FRE 2499
Université d'Artois
Lens, France
{$lecoutre, hemery$}@cril.univ-artois.fr

Abstract

In an Arc Consistency (AC) algorithm, a residual support, or residue, is a support that has been stored during a previous execution of the procedure which determines if a value is supported by a constraint. The point is that a residue is not guaranteed to represent a lower bound of the smallest current support of a value. In this paper, we study the theoretical impact of exploiting residues with respect to the basic algorithm AC3. First, we prove that AC3rm (AC3 with multi-directional residues) is optimal for low and high constraint tightness. Second, we show that when AC has to be maintained during a backtracking search, MAC2001 presents, with respect to MAC3rm, an overhead in O(μed) per branch of the binary tree built by MAC, where μ denotes the number of refutations of the branch, e the number of constraints and d the greatest domain size of the constraint network. One consequence is that MAC3rm admits a better worst-case time complexity than MAC2001 for a branch involving μ refutations when either $\mu > d^2$ or $\mu > d$ and the tightness of any constraint is either low or high. Our experimental results clearly show that exploiting residues allows enhancing MAC and SAC algorithms.

1 Introduction

It is well-known that Arc Consistency (AC) plays a central role in solving instances of the Constraint Satisfaction Problem (CSP). Indeed, the MAC algorithm, i.e., the algorithm which maintains arc consistency during the search of a solution, is still considered as the most efficient generic approach to cope with large and hard problem instances. Furthermore, AC is at the heart of a stronger consistency called Singleton Arc Consistency (SAC) which has recently attracted a lot of attention (e.g., [Bessière and Debruyne, 2005; Lecoutre and Cardon, 2005]).

For more that two decades, many algorithms have been proposed to establish arc consistency. Today, the most referenced algorithms are AC3 [Mackworth, 1977] because of its simplicity and AC2001/3.1 [Bessière *et al.*, 2005] because of its optimality (while being not too complex). The worst-case time complexities of AC3 and AC2001 are respectively O(ed^3) and O(ed^2) where e denotes the number of constraints and d the greatest domain size. The interest of an optimal algorithm such as AC2001 resides in its robustness. It means that AC2001 does not suffer from some pathological cases as AC3 does. This situation occurs when the tightness of the constraints is high, as it is the case for the equality constraint (i.e. constraint of the form $X = Y$). Indeed, as naturally expected and demonstrated later, AC3 admits then a practical behaviour which is close to the worst-case, and the difference by a factor d between the two theoretical worst-case complexities becomes a reality.

In this paper, we are interested in residues for AC algorithms. A residue is a support that has been stored during a previous execution of the procedure which determines if a value is supported by a constraint. The point is that a residue is not guaranteed to represent a lower bound of the smallest current support of a value. The basic algorithm AC3 can be refined by exploiting residues as follows: before searching a support for a value from scratch, the validity of the residue associated with this value is checked. We then obtain an algorithm denoted AC3r, and when multi-directionality is exploited, an algorithm denoted AC3rm.

In fact, AC3r is an algorithm which can be advantageously replaced by AC2001 when AC must be established standalone on a given constraint network. However, when AC has to be maintained during search, MAC3r which corresponds to mac3.1residue [Likitvivatanavong *et al.*, 2004] becomes quite competitive. On the other hand, AC3rm is interesting of its own as it exploits *multi-directional* residues just like AC3.2 [Lecoutre *et al.*, 2003]. But, let us see the interest of exploiting residues.

First, we prove in this paper that AC3rm, contrary to AC3, admits an optimal behaviour when the tightness of the constraints is high. To illustrate this, let us consider the Domino problem introduced in [Bessière *et al.*, 2005]. All but one constraints of this problem correspond to equality constraints. The results that we obtain when running AC3, AC2001, AC3.2 and the new algorithm AC3rm on some instances of this problem are depicted in Table 1. The time in seconds (cpu) and the number of constraint checks (ccks) is given for each instance of the form $domino$-n-d where n corresponds to the number of variables and d the number of values in each domain. Clearly, AC3rm largely compensates the weakness of the basic AC3.

$Instances$		$AC3$	$AC3rm$	$AC2001$	$AC3.2$
domino-100-100	cpu	1.81	0.16	0.23	0.18
	$ccks$	$18M$	$990K$	$1485K$	$990K$
domino-300-300	cpu	134	3.40	6.01	3.59
	$ccks$	$1377M$	$27M$	$40M$	$27M$
domino-500-500	cpu	951	15.0	21.4	15.2
	$ccks$	$10542M$	$125M$	$187M$	$125M$
domino-800-800	cpu	6144	60	87	59
	$ccks$	$68778M$	$511M$	$767M$	$511M$

Table 1: Establishing Arc Consistency on Domino instances

Next, we analyse the cost of managing data structures with respect to backtracking. On the one hand, it is easy to embed AC3rm in MAC and SAC algorithms as these algorithms do not require any maintenance of data structures during MAC search and SAC inference. On the other hand, embedding an optimal algorithm such as AC2001 entails an extra development effort, with, in addition, an overhead at the execution. For MAC2001, this overhead is O(μed) per branch of the binary tree built by MAC as we have to take into account the reinitialization of a structure (called $last$) which contains smallest found supports. Here, μ denotes the number of refutations of the branch, e denotes the number of constraints and d the greatest domain size.

2 Constraint Networks

A Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a set of n variables and $\mathscr{C}$ a set of e constraints. Each variable $x \in \mathscr{X}$ has an associated domain, denoted by *dom*(x), which contains the set of values allowed for x. Each constraint $C \in \mathscr{C}$ involves a subset of variables of $\mathscr{X}$, called scope and denoted *scp*(C), and has an associated relation, denoted *rel*(C), which contains the set of tuples allowed for the variables of its scope. The initial (resp. current) domain of a variable X is denoted dom$^{init}(X)$ (resp. dom(X)). For each r-ary constraint C such that scp$(C) = \{X_1, \ldots, X_r\}$, we have: rel$(C) \subseteq \prod_{i=1}^{r} dom^{init}(X_i)$ where $\prod$ denotes the Cartesian product. Also, for any element $t = (a_1, \ldots, a_r)$, called tuple, of $\prod_{i=1}^{r} dom^{init}(X_i)$, $t[X_i]$ denotes the value a_i. It is also important to note that, assuming a total order on domains, tuples can be ordered using a lexicographic order $\prec$. To simplify the presentation of some algorithms, we will use two special values $\bot$ and $\top$ such that any tuple t is such that $\bot \prec t \prec \top$.

Definition 1 *Let C be a r-ary constraint such that scp(C) = $\{X_1, \ldots, X_r\}$, a r-tuple t of $\prod_{i=1}^{r} dom^{init}(X_i)$ is said to be allowed by C iff $t \in rel(C)$, valid iff $\forall X_i \in$ scp(C), $t[X_i] \in$ dom(X_i), and a support in C iff it is allowed by C and valid.*

A tuple t will be said to be a support of (X_i, a) in C when t is a support in C such that $t[X_i] = a$. Determining if a tuple is allowed is called a constraint check. A solution to a CN is an assignment of values to all the variables such that all the constraints are satisfied. A CN is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given CN is satisfiable. A CSP instance is then defined by a CN, and solving it involves either finding one (or more) solution or determining its unsatisfiability. Arc Consistency (AC) remains the central property of CNs and establishing AC on a given network P involves removing all values that are not arc consistent.

Definition 2 *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A pair (X, a), with $X \in \mathscr{X}$ and $a \in dom(X)$, is arc consistent (AC) iff $\forall C \in \mathscr{C} \mid X \in$ scp(C), there exists a support of (X, a) in C. P is AC iff $\forall X \in \mathscr{X}$, dom($X$) $\neq \emptyset$ and $\forall a \in dom(X)$, (X, a) is AC.*

The following definitions will be useful later to analyze the worst-case time complexity of some algorithms.

Definition 3 *A cn-value is a triplet of the form (C,X,a) where $C \in \mathscr{C}$, $X \in$ scp(C) and $a \in dom(X)$.*

Definition 4 *Let (C,X,a) be a cn-value such that scp(C) = $\{X, Y\}$.*

- *The number of supports of (X, a) in C, denoted $s_{(C,X,a)}$, corresponds to the size of the set $\{b \in dom(Y) \mid (a, b) \in rel(C)\}$.*
- *The number of conflicts of (X, a) in C, denoted $c_{(C,X,a)}$, corresponds to the size of the set $\{b \in dom(Y) \mid (a, b) \notin rel(C)\}$.*

Note that the number of cn-values that can be built from a binary constraint network is O(ed). To sum up all evaluations of an expression $Expr(C, X, a)$ wrt all the cn-values of a given CN, we will write: $\sum_{C,X,a} Expr(C, X, a)$.

3 AC3rm

In this section, we introduce AC3rm, and we propose a detailed analysis of its complexity. It is important to remark that our algorithm is given in the general case (i.e. it can be applied to instances involving constraints of any arity). Hence, strictly speaking, its description corresponds to GAC3rm since for non binary constraints, one usually talks about Generalized Arc Consistency (GAC). However, to simplify, theoretical complexities will be given for binary instances. More precisely, for all theoretical results, we will consider given a binary CN $P = (\mathscr{X}, \mathscr{C})$ such that, to simplify and without any loss of generality, each domain exactly contains d values.

To establish (generalized) arc consistency on a given CN, $doAC$ (Algorithm 1) can be called. It returns $true$ when the given CN can be made arc-consistent and it is described in the context of a coarse-grained algorithm. Initially, all pairs (C, X), called arcs, are put in a set Q. Once Q has been initialized, each arc is revised in turn (line 4), and when a revision is effective (at least one value has been removed), the set Q has to be updated (line 6). A revision is performed by a call to the function $revise$, and entails removing values that have become inconsistent with respect to C. This function

Algorithm 1 doAC (P = $(\mathscr{X}, \mathscr{C})$: CN): Boolean

```
1: Q ← {(C, X) | C ∈ 𝒞 ∧ X ∈ scp(C)}
2: while Q ≠ ∅ do
3:     pick and delete (C, X) from Q
4:     if revise(C,X) then
5:         if dom(X) = ∅ then return false
6:         Q ← Q ∪ {(C', Y) | C' ≠ C, Y ≠ X, {X, Y} ⊆ scp(C')}
7: return true
```

Algorithm 2 revise(C : Constraint, X : Variable) : Boolean

```
1: nbElements ← | dom(X) |
2: for each a ∈ dom(X) do
3:    if supp[C, X, a] is valid then continue
4:    t ← seekSupport(C, X, a)
5:    if t = ⊤ then remove a from dom(X)
6:    else for each Y ∈ scp(C) do supp[C, Y, t[Y]] ← t
7: return nbElements ≠ | dom(X) |
```

Algorithm 3 seekSupport(C, X, a) : Tuple

```
1: t ← ⊥
2: while t ≠ ⊤ do
3:    if C(t) then return t
4:    t ← setNextTuple(C, X, a, t)
5: return ⊤
```

returns $true$ when the revision is effective. The algorithm is stopped when a domain wipe-out occurs (line 5) or the set Q becomes empty.

Following the principle used in AC3.2 [Lecoutre *et al.*, 2003], we propose a mechanism to partially benefit from (positive) multi-directionality. The idea is that, when a support t is found, it can be recorded for all values occurring in t. For example, let us consider a binary constraint C such that $scp(C) = \{X, Y\}$. If (a, b) is found in C when looking for a support of either (X, a) or (Y, b), in both cases, it can be recorded as being the last found support of (X, a) in C and the last found support of (Y, b) in C. In fact, one can simply record for any cn-value (C,X,a) the last found support of (X,a) in C. However, here, unlike AC2001, by exploiting multi-directionality, we cannot benefit anymore from unidirectionality. It means that, when the last found support is no more valid, one has to search for a new support from scratch. Indeed, by using multi-directionality, we have no guarantee that the last found support corresponds to the last smallest support. This new algorithm requires the introduction of a three-dimensional array, denoted *supp*. This data structure is used to store for any cn-value (C,X,a) the last found support of (X,a) in C. Initially, any element of the structure *supp* must be set to $\bot$. Each revision (see Algorithm 2) involves testing for any value the validity of the last found support (line 3) and if, it fails, a search for a new support is started from scratch (see Algorithm 3). It uses $setNextTuple$ which returns either the smallest valid tuple t' built from C such that $t \prec t'$ and $t'[X] = a$, or $\top$ if it does not exist. Without any loss of generality, we assume that any call to $setNextTuple$ is performed in constant time. Note that $C(t)$ must be understood as a constraint check and that $C(\bot)$ returns false. If this search succeeds, structures corresponding to last found supports are updated (line 6).

To summarize, the structure *supp* allows to record what we call *multi-directional* residues. Of course, it is possible to exploit simpler residues [Likitvivatanavong *et al.*, 2004], called here *uni-directional* residues, by not exploiting multi-directionality. We can then derive a new algorithm, denoted AC3r, by replacing line 6 of Algorithm 2 with:

else $supp[C, X, a] \leftarrow t$

However, with AC3r, when AC must be established standalone, rather than searching a new support from scratch when the residue is no more valid, it is more natural and more efficient to perform the search using the value of the residue as a resumption point. This is exactly what is done by AC2001. It means that, in practice, AC3r is interesting only when it is embedded in MAC [Likitvivatanavong *et al.*, 2004] or a SAC algorithm.

AC3rm has a space complexity of $O(ed)$ and a non-optimal worst-case time complexity of $O(ed^3)$. However, it is possible to refine this result as follows:

Proposition 1 *In AC3rm, the worst-case cumulated time complexity of $seekSupport$ for a cn-value (C,X,a) is O($cs + d$) with $c = c_{(C,X,a)}$ and $s = s_{(C,X,a)}$.*

Proof. The worst-case in terms of constraint checks is when: 1) only one value is removed from $\text{dom}^{init}(Y)$ between two calls to revise(C,X), 2) values of $\text{dom}^{init}(Y)$ are ordered in such a way that the c first values correspond to values which do not support a and the s last values correspond to values which support a, 3) the first s values removed from $\text{dom}^{init}(Y)$ systematically correspond to the last found supports recorded by AC3rm (until a domain wipe-out is encountered). For these $s + 1$ calls (note the initial call) to $seekSupport(C, X, a)$, we obtain $s * (c + 1) + c$ constraint checks. On the other hand, the number of other operations (validity checks and updates of the supp structure) in revise performed with respect to a is bounded by d. Then, we have a worst-case cumulated complexity in O($sc + s + c + d$) = O($cs + d$). □

What is interesting with AC3rm is that, even if this algorithm is not optimal, it is adapted to instances involving constraints of low or high tightness. Indeed, when the constraint tightness is low (more precisely, when c is O(1)) or high (when s is O(1)), the worst-case cumulated time complexity becomes $O(d)$, what is optimal. On the other hand, sc is maximized when $c = s = d/2$, what corresponds to a medium constraint tightness. However, AC3rm can also be expected to have a good (practical) behavior for medium constraint tightness since, on average (i.e. asymptotically), considering random constraints, 2 constraint checks are necessary to find a support when the tightness is 0.5. We can deduce the following result.

Proposition 2 *The worst-case time complexity of AC3rm is: O($ed^2 + \sum_{C,X,a} c_{(C,X,a)} * s_{(C,X,a)}$).*

Table 2 indicates the overall worst-case complexities[1] to establish arc consistency with algorithms AC3, AC3rm, AC2001 and AC3.2. It is also interesting to look at worst-case cumulated time complexities to seek successive supports for a given cn-value (C,X,a). Even if it has not been introduced earlier, it is easy to show that optimal algorithms admit a cumulated complexity in O(d). By observing Table 3, we do learn that AC3 and AC3rm are optimal when the tightness is low (i.e. c is O(1)), and that, unlike AC3, AC3rm is also optimal when the tightness is high (i.e. s is O(1)).

[1] Due to lack of space, we do not provide the detailed proof of the original complexities given for AC3.

	Space	Time
AC3	$O(e+nd)$	O($d * \sum_{C,X,a} c_{(C,X,a)} + \sum_{C,X,a} s_{(C,X,a)}$)
AC3rm	$O(ed)$	O($ed^2 + \sum_{C,X,a} c_{(C,X,a)} * s_{(C,X,a)}$)
AC2001	$O(ed)$	$O(ed^2)$
AC3.2	$O(ed)$	$O(ed^2)$

Table 2: Worst-case complexities to establish AC.

	Tightness			
	Any	Low	Medium	High
AC3	$O(cd+s)$	$O(d)$	$O(d^2)$	$O(d^2)$
AC3rm	$O(cs+d)$	$O(d)$	$O(d^2)$	$O(d)$
AC2001	$O(d)$	$O(d)$	$O(d)$	$O(d)$
AC3.2	$O(d)$	$O(d)$	$O(d)$	$O(d)$

Table 3: Cumulated worst-case time complexities to seek successive supports for a cn-value (C,X,a). We have $c + s = d$.

Remark that the complexities given for AC3rm also hold for AC3r. The advantage of AC3rm is the fact that as we always record the most recent found supports (by exploiting multi-directionality), there is a greater probability that a residue be valid. Finally, note that it should be possible to extend the AC-* framework [Régin, 2005] in order to include the concept of residues.

4 Maintaining arc consistency

In this section, we focus on maintaining arc consistency during search. More precisely, we study the impact, in terms of time and space, of embedding some AC algorithms in MAC. The MAC algorithm aims at solving a CSP instance and performs a depth-first search with backtracking while maintaining arc consistency. At each step of the search, a variable assignment is performed followed by a filtering process that corresponds to enforcing arc-consistency. MAC is based on a binary branching scheme. It means that, at each step of the search, a pair (X,a) is selected where X is an unassigned variable and a a value in dom(X), and two cases are considered: the first one corresponds to the assignment $X = a$ and the second one to the refutation $X \neq a$.

On the other hand, it is important to remark that all known AC algorithms (including AC3rm) are incremental. An arc-consistency algorithm is incremental if its worst-case time complexity is the same when it is applied one time on a given network P and when it is applied up to nd times on P where between two consecutive executions at least one value has been deleted. By exploiting incrementality, one can get the same complexity, in terms of constraint checks, for any branch of the search tree as for only one establishment of AC.

For AC3 and AC3rm, the (non optimal) worst-case time complexity for any branch of the search tree is guaranteed (by incrementality) even if, meanwhile, sub-trees have been explored and then backtracking has occurred. However, for optimal algorithms AC2001 and AC3.2, it is important to manage the data structure, denoted $last$, in order to restart search, after exploring a sub-tree, as if backtracking never occurred. In this paper, MAC2001 and MAC3.2 correspond to the algorithms that record the smallest supports that have been successively found all along the current branch. Note that it is at the price of a space complexity in O(min(n,d)ed) [van Dongen, 2004].

	Space	Time (per branch)
MAC3	$O(e+nd)$	O($ed^2 + d * \sum_{C,X,a} c_{(C,X,a)}$)
MAC3rm	$O(ed)$	O($ed^2 + \sum_{C,X,a} c_{(C,X,a)} * s_{(C,X,a)}$)
MAC2001	$O(min(n,d)ed)$	$O(ed(d+\mu))$
MAC3.2	$O(min(n,d)ed)$	$O(ed(d+\mu))$

Table 4: Worst-case complexities to run MAC. Time complexity is given for a branch involving μ refutations.

Proposition 3 *In MAC2001 and MAC3.2, the worst-case cumulated time complexity of reinitializing the structure $last$ is O(μed) for any branch involving μ refutations.*

Proof. For any refutation occurring in a branch, we need to restore the data structure $last$. In the worst-case, we have at most $e * 2 * d$ operations since for each cn-value (C,X,a), we have to reinitialize $last[C, X, a]$ to a stacked value (or, for variants, to $\perp$ or a new recomputed value). Hence, we obtain (μed). □

If $\mu = 0$, it means that a solution has been found without any backtracking. In this case, there is no need to restore the structure $last$ as the instance is solved. At the opposite, we know that the longest branch that can be built contains nd edges as follows: for each variable X, there are exactly $d - 1$ edges that correspond to refutations and only one edge that corresponds to an assignment. Then, we obtain a worst-case cumulated time complexities of reinitializing the structure $last$ in O(end^2) and although it is omitted here, we can also show that it is $\Omega(end^2)$.

One nice feature of AC3rm is that, when they are embedded in MAC, no initialization is necessary at each step since the principle of this algorithm is to record the last found support which does not systematically correspond to the last smallest one. In fact, it was reported in [Lecoutre *et al.*, 2003] that it is worthwhile to leave unchanged last found supports (using AC3.2) while backtracking, having the benefit of a so-called memorization effect. It means that a support found at a given depth of the search has the opportunity to be still valid at a weaker depth of the search (after backtracking). In other words, it is worthwhile to exploit residues during search. The importance of limiting in MAC the overhead of maintaining the data structures employed by the embedded AC algorithm was pointed out in [Likitvivatanavong *et al.*, 2004] (but no complexity result was given). In fact, MAC3r corresponds to the algorithm mac3.1residue introduced in [Likitvivatanavong *et al.*, 2004].

By taking into account Proposition 3 and Table 2, we obtain the results given in Table 4. It appears that, for the longest branch, when $\mu > d^2$, MAC3 and MAC3rm have a better worst-case time complexity than other MAC algorithms based on optimal AC algorithms since we know that, for any branch, due to incrementality, MAC3 and MAC3rm are O(ed^3). Also, if the tightness of any constraint is either low or high (more precisely, if for any cn-value (C,X,a), either $c_{(C,X,a)}$ is O(1) or $s_{(C,X,a)}$ is O(1)), then MAC3rm admits an optimal worst-case time complexity in O(ed^2) per

		MAC embedding			
		$AC3$	$AC3rm$	$AC2001$	$AC3.2$
Classes of random instances (mean results for 100 instances)					
⟨40-8-753-0.1⟩	cpu	22.68	22.96	34.38	33.35
	$ccks$	$81M$	$17M$	$24M$	$16M$
⟨40-11-414-0.2⟩	cpu	21.19	19.23	27.91	26.34
	$ccks$	$97M$	$22M$	$28M$	$19M$
⟨40-16-250-0.35⟩	cpu	21.86	18.31	25.18	23.38
	$ccks$	$121M$	$28M$	$33M$	$24M$
⟨40-25-180-0.5⟩	cpu	37.35	25.53	35.30	32.27
	$ccks$	$233M$	$56M$	$60M$	$45M$
⟨40-40-135-0.65⟩	cpu	37.62	26.62	35.98	34.45
	$ccks$	$344M$	$83M$	$82M$	$64M$
⟨40-80-103-0.8⟩	cpu	89.01	51.62	67.74	61.48
	$ccks$	$1072M$	$240M$	$225M$	$180M$
⟨40-180-84-0.9⟩	cpu	166.12	76.99	98.69	87.50
	$ccks$	$2540M$	$506M$	$479M$	$381M$
Academic instances					
ehi-85-12	cpu	394	377	557	511
	$ccks$	$642M$	$60M$	$190M$	$83M$
geo-50-20-19	cpu	194	157	278	263
	$ccks$	$1117M$	$244M$	$284M$	$199M$
qa-5	cpu	31.60	28.31	37.49	36.02
	$ccks$	$130M$	$36M$	$38M$	$27M$
qcp-819	cpu	139	143	215	208
	$ccks$	$116M$	$21M$	$41M$	$25M$
Real-world instances					
$fapp$01-0200-9	cpu	0.54	0.37	0.60	0.60
	$ccks$	$6905K$	$3080K$	$3018K$	$2778K$
js-$enddr$2-3	cpu	53.66	29.08	39.24	29.60
	$ccks$	$596M$	$104M$	$88M$	$48M$
$scen$-11	cpu	15.67	11.88	16.26	14.58
	$ccks$	$92M$	$18M$	$15M$	$10M$
$graph$-10	cpu	0.64	0.54	0.69	0.68
	$ccks$	$4842K$	$2216K$	$2228K$	$1925K$

Table 5: Cost of running MAC

branch. In this case, MAC3rm outperforms MAC2001 as soon as $\mu > d$. These observations suggest that MAC3rm should be very competitive.

5 Experiments

To compare the different algorithms mentioned in this paper, we have performed a vast experimentation (run on a PC Pentium IV 2.4GHz 512MB under Linux) with respect to random, academic and real-world problems. Performances have been measured in terms of the CPU time in seconds (cpu) and the number of constraint checks (ccks).

To start, we have tested MAC (equipped with dom/deg) by considering 7 classes of binary random instances situated at the phase transition of search. For each class $\langle n,d,e,t\rangle$, defined as usually, 100 instances have been generated. The tightness t denotes the probability that a pair of values is allowed by a relation. What is interesting here is that a significant sampling of domain sizes, densities and tightnesses is introduced. In Table 5, we can observe the results obtained with MAC embedding the various AC algorithms. As expected, the best embedded algorithms are AC3 and AC3rm when the tightness is low (here 0.1) and AC3rm when the tightness is high (here, 0.9). Also, AC3rm is the best when the tightness is medium (here 0.5) as expected on random instances. All these results are confirmed for some representative selected academic and real-world instances. Clearly, MAC3rm outperform all other MAC algorithms in terms of cpu while MAC3.2 is the best (although beaten on a few instances by

		$SAC-1$ embedding			
		$AC3$	$AC3rm$	$AC2001$	$AC3.2$
Academic instances					
$domino$-300-300	cpu	446.32	9.56	14.40	9.67
	$ccks$	$1376M$	$26M$	$40M$	$26M$
$domino$-500-100	cpu	4.37	0.53	0.71	0.57
	$ccks$	$88M$	$4950K$	$7425K$	$4950K$
geo-50-20-19	cpu	1.18	0.74	1.49	1.24
	$ccks$	$9525K$	$1165K$	$2671K$	$1157K$
qa-5	cpu	1.22	0.89	1.91	1.34
	$ccks$	$10M$	$3104K$	$5001K$	$3085K$
Real-world instances					
$fapp$01-0200-9	cpu	637	158	312	192
	$ccks$	$10047M$	$905M$	$1795M$	$904M$
js-$enddr$2-3	cpu	58.95	12.35	24.66	14.38
	$ccks$	$980M$	$54M$	$128M$	$55M$
$graph$-10	cpu	980	439	836	581
	$ccks$	$12036M$	$1307M$	$2467M$	$1303M$
$scen$-11	cpu	44.89	21.07	56.03	53.21
	$ccks$	$479M$	$33M$	$52M$	$37M$

Table 6: Cost of establishing SAC-1

MAC3rm) in terms of constraint checks. Interestingly, in an overall analysis, we can remark that MAC3rm and MAC2001 roughly perform the same number of constraint checks. As, on the other hand, MAC3rm do not require any data structure to be maintained, it explains why it is the quickest approach. These results confirm the results obtained for MAC3r (mac3.1residue) in [Likitvivatanavong *et al.*, 2004] which has a behaviour similar to MAC3rm (due to lack of space, results for MAC3r are not presented).

We have then embedded AC algorithms in SAC-1. In Table 6, one can observe that, for domino instances which involve constraints of high tightness, AC3rm clearly shows its superiority to AC3. For real-world instances, the gap between AC3rm and the other algorithms increases. For example, SAC-1 embedding AC3rm is about 3 times more efficient than SAC-1 embedding AC2001 on $scen11$ and 4 times more efficient than SAC-1 embedding AC3 on $fapp$01-0200-9.

6 Residues for Non Binary Constraints

One can wonder what is the behaviour of an algorithm that exploits residues when applied to non binary instances. First, it is important to remark that seeking a support of a cn-value from scratch requires iterating O(d^{r-1}) tuples in the worst-case for a constraint of arity r. We then obtain a worst-case cumulated time complexity of seeking a support of a given cn-value in O(r^2d^r) for GAC3 and O(rd^{r-1}) for GAC2001 [Bessière *et al.*, 2005] since we consider that a constraint check is O(r) and since there are O(rd) potential calls to the specific $seekSupport$ function. Then, we can observe that there is a difference by a factor dr. It means that the difference between the two algorithms grows linearly with r.

On the other hand, if we assume that $c > 0$ and $s > 0$ respectively denote the number of forbidden and allowed tuples of the constraint, then we obtain, by generalizing our results of Section 3, a complexity in O(crd^{r-1}) for GAC3 and in O(cs) for GAC3rm. We can then deduce that the worst-case cumulated time complexity of seeking a support is O($min(c, rd).rd^{r-1}$) for GAC3 and O($min(csr, r^2d^r)$) for GAC3rm. If $c = O(1)$ or $s = O(1)$, we obtain O(rd^{r-1}) for GAC3rm as $c + s = d^{r-1}$, that is to say optimality. However,

		MGAC embedding			
Instance		GAC3	GAC3r	GAC3rm	GAC2001
Random instances (mean results for 10 instances - constraints are of arity 6)					
⟨20-6-36-0.55⟩	*cpu*	13.1	8.7	8.0	10.2
	ccks	$12M$	$6481K$	$4997K$	$6825K$
⟨20-8-24-0.75⟩	*cpu*	51.7	31.8	27.7	34.6
	ccks	$48M$	$26M$	$20M$	$26M$
⟨20-10-14-0.95⟩	*cpu*	220	135	102	135
	ccks	$249M$	$151M$	$108M$	$149M$
⟨20-20-15-0.99⟩	*cpu*	869	489	301	351
	ccks	$2255M$	$1289M$	$785M$	$887M$
Structured instances					
tsp-20-366	*cpu*	387	242	243	266
	ccks	$607M$	370	$364M$	$387M$
gr-44-9-*a*3	*cpu*	73.1	37.2	38.4	56.3
	ccks	$166M$	$44M$	$41M$	$74M$
gr-44-10-*a*3	*cpu*	2945	1401	1465	2129
	ccks	$6819M$	$1513M$	$1527M$	$2914M$
series-14	*cpu*	233	218	217	312
	ccks	$1135M$	$531M$	$490M$	$618M$
renault	*cpu*	25.0	25.4	16.2	25.2
	ccks	$68M$	$66M$	$42M$	$66M$

Table 7: Cost of running MGAC

in practice, the likelihood of having small (bounded) values of c or s when dealing with non binary constraints is weak.

We have performed a preliminary experimentation by maintaining GAC algorithms during search on series of random non binary instances. Here, we chose constraints of arity 6 and studied the behaviour of the algorithm for a tightness $t \in \{0.55, 0, 75, 0.95, 0.99\}$. For small values of t, we observed (as in the binary case) that the difference between all algorithms was limited. On these random instances, one can observe in Table 7 that GAC3rm and GAC3.2 are the most efficient embedded algorithms. Of course, when the tightness is high, GAC3 is penalized and GAC3r is less efficient then GAC3rm as exploiting multi-directionality pays off. On non binary structured instances of the 2005 CSP solver competition, one can see the good behaviour of GAC3r and GAC3rm.

7 Conclusion

In this paper, we have introduced some theoretical results about the use of residual supports, or residues, in Arc Consistency algorithms. The concept of residue has been introduced under its multi-directional form in [Lecoutre *et al.*, 2003] and under its uni-directional form in [Likitvivatanavong *et al.*, 2004]. We have first proved that the basic algorithm AC3 which is optimal for low constraint tightness, also becomes optimal for high constraint tightness when it is extended to exploit uni-directional or multi-directional residues. Furthermore, these extensions to AC3, respectively called AC3r and AC3rm, can be expected to have a good (practical) behavior for medium tightness as asymptotically, for random constraints, 2 constraint checks are necessary to find a support when the tightness is 0.5. Then, we have shown that MAC3rm admit a better worst-case time complexity than MAC2001 for a branch of the binary search tree when either $\mu > d^2$ or $\mu > d$ and the tightness of any constraint is low or high, with μ denoting the number of refutations of the branch.

On the practical side, we have run a vast experimentation including MAC and SAC-1 algorithms on binary and non binary instances. The results that we have obtained clearly show the interest of exploiting residues as AC3rm (embedded in MAC or SAC-1) were almost always the quickest algorithms (only beaten by AC3.2 on some non binary instances). It confirms for MAC3r (mac3.1residue) the results presented in [Likitvivatanavong *et al.*, 2004]. In terms of constraint checks, it appears that AC3rm is quite close to AC2001 (but usually beaten by AC3.2). We also noted that AC3rm was more robust than AC3r on non binary instances and constraints of high tightness.

Finally, residues can be seen as a lazy structure related to the concept of watched literals [Moskewicz *et al.*, 2001]. In both cases, no maintenance of data structures upon backtracking is required. We also want to emphasize that implementing (G)AC3rm (and embedding it in MAC or SAC) is a quite easy task. It should be compared with the intricacy of fine-grained algorithms which requires a clever use of data structures, in particular when applied to non binary instances. The simplicity of AC3rm offers another scientific advantage: the easy reproducibility of the experimentation by other researchers.

Acknowledgments

We would like to thank Christian Bessière and Romuald Debruyne for their useful comments. This paper has been supported by the CNRS and the ANR "Planevo" project.

References

[Bessière and Debruyne, 2005] C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proc. of IJCAI'05*, pages 54–59, 2005.

[Bessière *et al.*, 2005] C. Bessière, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[Lecoutre and Cardon, 2005] C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05*, pages 199–204, 2005.

[Lecoutre *et al.*, 2003] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.

[Likitvivatanavong *et al.*, 2004] C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Arc consistency in MAC: a new perspective. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 93–107, 2004.

[Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Moskewicz *et al.*, 2001] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC'01*, pages 530–535, 2001.

[Régin, 2005] J.C. Régin. AC-*: a configurable, generic and adaptive arc consistency algorithm. In *Proceedings of CP'05*, pages 505–519, 2005.

[van Dongen, 2004] M.R.C. van Dongen. Saving support-checks does not always save time. *Artificial Intelligence Review*, 21(3-4):317–334, 2004.