Enforcing Arc Consistency using Bitwise Operations

Christophe Lecoutre, Julien Vion

Constraint Programming Letters 2 (2008) 21–35 Submitted 3/2007; Published 1/2008

Enforcing Arc Consistency using Bitwise Operations

Christophe Lecoutre LECOUTRE@CRIL.FR
Julien Vion VION@CRIL.FR
CRIL (Centre de Recherche en Informatique de Lens),
CNRS FRE 2499,
rue de l'université, SP 16
62307 Lens cedex, France

Editor: M.R.C. van Dongen

Abstract

In this paper, we propose to exploit bitwise operations to speed up some important computations such as looking for a support of a value in a constraint, or determining if a value is substitutable by another one. Considering a computer equipped with a x-bit CPU, one can then expect an increase of the performance by a coefficient up to x (which may be important, since x is equal to 32 or 64 in many current central units). To show the interest of enforcing arc consistency using bitwise operations, we introduce a new variant of AC3, denoted by AC3^{bit}, which can be used when constraints are (or can be) represented in extension. This new algorithm when embedded in MAC, is approximately two times more efficient than AC3^{rm}. Note that AC3^{rm} is a variant of AC3 which exploits the concept of residual supports and has been shown to be faster than AC2001.

Keywords: Arc Consistency, Bitwise Operations

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, which maintains arc consistency during the search of a solution as described in (Sabin and Freuder, 1994), is still considered as the most efficient generic approach to cope with large and hard problem instances. Furthermore, AC is at the heart of stronger consistencies that have recently attracted some attention: singleton arc consistency (Bessiere and Debruyne, 2005; Lecoutre and Cardon, 2005), weak k-singleton arc consistency (van Dongen, 2006) and conservative dual consistency (Lecoutre et al., 2007).

For more than two decades, many algorithms have been proposed to establish arc consistency. They are usually classified as coarse-grained or fine-grained algorithms. Even if fine-grained algorithms are conceptually elegant, e.g. AC7 (Bessiere et al., 1999), coarse-grained algorithms are easier to implement while being competitive, and are thus more attractive. The basic coarse-grained algorithm is AC3 which has been introduced by Mackworth (1977). Its worst-case time complexity is $O(ed^3)$ where e denotes the number of constraints and d the greatest domain size.

Since its conception, AC3 has been the subject of many studies or developments. Wallace (1993) explained why AC3 was almost always more efficient than the optimal AC4 (Mohr and Henderson, 1986). Van Dongen (2002) proposed to equip AC3 with a double-support domain heuristic; some refinements of this method being described by Mehta and van Dongen (2004). Interestingly, the algorithm AC2001/3.1 proposed by Bessiere et al. (2005) corresponds to AC3 made optimal (its

worst-case time complexity is $O(ed^2)$) by the introduction of a structure that manages last found supports. Finally, Lecoutre et al. (2003) have introduced two additional extensions of AC3 by exploiting multi-directionality.

Recently, Lecoutre and Hemery (2007) studied the impact of exploiting residual supports (called residues, in short) which were introduced in (Lecoutre et al., 2003; Likitvivatanavong et al., 2004), and showed both theoretically and experimentally the advantage of embedding AC3^{rm} (AC3 exploiting multi-directional residues) in MAC or in an algorithm that enforces singleton arc consistency. More precisely, unlike AC2001, AC3^{rm} does not require the maintenance of data structures upon backtracking and, unlike AC3, AC3^{rm} does not suffer from known pathological cases. As a consequence, embedding AC3^{rm} is a very simple and efficient solution. As we can consider that AC3^{rm}, although not optimal in the worst case, behaves in an optimal way most often (at least for constraints whose tightness is high or low, or with supports uniformly dispersed), the opportunity to improve on it is a new challenge.

The O notation is the most usually used when presenting time (and space) complexities of algorithms. This corresponds to an asymptotic analysis, which is relevant to judge the practical efficiency of an algorithm, provided that the elements (terms and coefficients) discarded from the raw complexity expression are not too high. To illustrate this, let us consider a constraint network composed of n variables, the domain of each being composed of d values, and e binary *max-support* constraints[1]. A max-support constraint involving the variables X and Y is defined as follows: the maximum value in the domain of X supports all values in the domain of Y, and vice versa. Figure 1 depicts such a constraint.

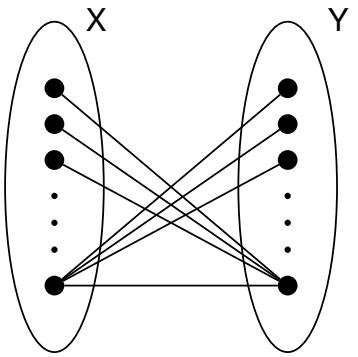

Figure 1: A *max-support* constraint. An edge represents an allowed tuple.

If we enforce AC on this network using AC3 or AC2001, we can establish that exactly $2e.(d^2 - d + 1)$ constraint checks are necessary to prove that the network is arc consistent. Here we are assuming we are looking for support in the opposite domain by considering the values from top to bottom. Now, consider (without any loss of generality) that we benefit for each domain from a binary representation of its current state, i.e. a bit is associated with each value of the domain and indicates if the value is present or not. The domain of a variable thus consists of a bit vector. Assume similarly that constraints are represented by giving the binary representation (vector) of all allowed and forbidden values of any triplet (C, X, a), where a is a value belonging to the domain

1. It is interesting to note that, even if they are initially absent from a given CN, such constraints may dynamically "appear" during search and propagation (when considering reduced domains).

of the variable X and C a constraint involving X. When looking for a support of (C, X, a), we can then simply apply a bitwise operation on these two vectors (as described later).

Considering that bit vectors are equivalent to an array of words (natural data units of the computer architecture), each elementary bitwise operation between two words represents, with respect to our illustration, x constraint checks, x being the word size (usually, 32 or 64). Hence, we have just established that, for the example introduced above, we need up to x times less operations when enforcing AC using this principle (we will call it AC3^{bit}) than classical AC3, AC2001 and AC3^{rm} algorithms. Table 1 gives some experimental results that we have obtained for this problem (each instance is described under the form n-d-e) on a 64-bit processor. Here, $\#ops$ denotes either the number of bitwise operations performed by AC3^{bit} or the number of constraint checks performed by AC3, AC3^{rm} and AC2001. As expected, we can observe that AC3^{bit} is about 60 times more efficient although AC3^{bit} and AC3 both are $O(ed^3)$.

$Instances$		$AC3$	$AC3^{rm}$	$AC2001$	$AC3^{bit}$
250-50-5000	cpu	1.58	1.56	1.61	0.05
	$\#ops$	$24.5M$	$24.3M$	$24.5M$	$0.5M$
250-100-5000	cpu	6.17	6.15	6.26	0.10
	$\#ops$	$99.0M$	$98.5M$	$99.0M$	$2.0M$
500-50-10000	cpu	3.11	3.11	3.21	0.11
	$\#ops$	$49.0M$	$48.5M$	$49.0M$	$1.0M$
500-100-10000	cpu	12.29	12.27	12.48	0.19
	$\#ops$	$198.0M$	$197.0M$	$198.0M$	$4.0M$

Table 1: Establishing Arc Consistency on *max-supports* instances

The idea of exploiting bitwise operations to speed up computations is not new. In particular, McGregor (1979) indicated that bit vectors can be used to represent domains and sets of supports as described above. Similar optimizations were already mentioned by Ullmann (1976). Also, bit parallel forward checking has been addressed in (Haralick and Elliott, 1980; Nudel, 1983). The modest contribution of this paper is to provide a precise description of how bitwise operators can be exploited to enforce arc consistency and to show, from a vast experimentation, that this approach is really the most efficient one.

The paper is organized as follows. First, we introduce constraint networks and arc consistency. Then, we show how to represent domains and constraints in binary, and how to exploit bitwise operators on them. Next, we propose a new generic algorithm AC3^{bit} which can be seen as a simple optimization of AC3. Finally, after presenting the results of an experimentation we have conducted, we conclude.

2. Constraint Networks and Arc Consistency

A Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of n variables and $\mathscr{C}$ a finite set of e constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted $dom(X)$, which contains the finite set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves an ordered 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. From now on, we will only consider binary constraints, i.e. constraints involving exactly two variables.

The initial domain of a variable X is denoted $dom^{init}(X)$ whereas the current domain of X is denoted $dom(X)$. For any binary constraint C such that $scp(C) = \{X, Y\}$, we have:

$$rel(C) \subseteq dom^{init}(X) \times dom^{init}(Y)$$

where $\times$ denotes the Cartesian product. A value $a \in dom^{init}(X)$ will often be denoted by (X, a). We will consider that each domain is ordered.

Definition 1 *Let C be a binary constraint such that $scp(C) = \{X, Y\}$, a pair of values $t = ((X, a), (Y, b))$ is said to be:*

- *allowed by C iff $(a, b) \in rel(C)$,*
- *valid iff $a \in dom(X) \wedge b \in dom(Y)$,*
- *a support in C iff it is allowed by C and valid.*

A tuple t is a support of (X, a) in C if t is a support in C such that the value assigned to X in t is a. Determining if a tuple is allowed or not is called a constraint check. A solution to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied. A constraint network 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 constraint network is satisfiable. A CSP instance is then defined by a constraint network, and solving it involves either finding at least one solution or determining its unsatisfiability. Arc Consistency (AC) remains the central property of (binary) constraint networks and establishing AC on a given network 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.*

We will use the following notion of cn-value when presenting some algorithms.

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

3. Binary Representation

In this section, we provide some details about the binary representation of domains and constraints. We consider that bit vectors are represented under the form of an array of words (natural data units of the computer architecture). Indeed, some programming languages do not provide the possibility of using bit vectors as data structures. Besides, as we will see, it is more efficient to perform some computations based on bitwise operators, using arrays of words rather than bit vectors.

Without any loss of generality, we will consider here that the computer is equipped with a 64-bit processor. It means for example that the declaration of arrays in the Java language would be $long[]$ since one $long$ corresponds to 64 bits.

3.1 Representing Domains

When a copying mechanism (Schulte, 1999) is used to manage domains during a backtracking search, one can associate a single bit with any value of each domain. More precisely, a bit can be associated with the index (starting at 0) of any value of a domain. When this bit is set to 1 (respectively 0), it means that the corresponding value is present in the domain (respectively absent from it). Using an array of words, one can then compactly represent domains. We will call such arrays the binary representation of domains. For any variable X, the space complexity is then $\Theta(|dom(X)|)$, which is optimal.

Another mechanism used in many current CP systems is called trailing. A precise description about how to represent domains using this mechanism can be found in (Lecoutre and Szymanek, 2006), following elements introduced by van Hentenryck et al. (1992). The space complexity of this representation is also $\Theta(|dom(X)|)$ for any variable X, and the time complexity of all elementary operations (determining if a value is present, removing a value, adding a value, etc.) is $O(1)$. In this context, adding and maintaining the structures for the binary representation of domains do not modify worst-case space and time complexities, as shown below.

To represent domains, we keep the structures presented in (Lecoutre and Szymanek, 2006) and introduce an additional two-dimensional array called $bitDom$ that associates with any variable X the binary representation $bitDom[X]$ of $dom(X)$, and:

- when adding (or restoring) the i^{th} value in $dom(X)$, the only operation required on the structure $bitDom$ is the following:

$$bitDom[X][i \text{ div } 64] \leftarrow bitDom[X][i \text{ div } 64] \text{ OR } masks1[i \text{ mod } 64]$$

- when removing the i^{th} value in $dom(X)$, the only operation required on structure $bitDom$ is the following:

$$bitDom[X][i \text{ div } 64] \leftarrow bitDom[X][i \text{ div } 64] \text{ AND } masks0[i \text{ mod } 64]$$

Here, div denotes the integer division, mod the remainder operator, OR the bitwise operator that performs a logical OR operation on each pair of corresponding bits and AND the bitwise operator that performs a logical AND operation on each pair of corresponding bits. The structure $masks1$ (resp. $masks0$) is a predefined array of 64 words that contains in its i^{th} square a value that represents a sequence of 64 bits which are all set to 0 (resp. 1) except for the i^{th} one.

3.2 Representing Constraints

In this paper, we will only consider binary constraints. A binary constraint can be represented in extension using a two-dimensional array of Booleans or a list of tuples, or in intention using a predicate expression.

Here, to represent constraints, we introduce a two-dimensional array called $bitSup$. More precisely, for each cn-value (C, X, a), $bitSup[C, X, a]$ represents the binary representation of the (initial) supports of (X, a) in C. To simplify the presentation and without any loss of generality, we can assume that indexes and values match (i.e. the i^{th} value of the domain of any variable is equal to i). If C is such that $scp(C) = \{X, Y\}$, then $(a, b) \in rel(C)$ iff the b^{th} bit in $bitSup[C, X, a]$ is 1.

If the constraints are initially given to the solver in extensional form, then, building the $bitSup$ array does not present any particular difficulty. On the other hand, if the constraints are given

in intention, then all constraints checks have to be initially performed (by evaluating a predicate) in order to build $bitSup$. Assuming that each constraint check is performed in constant time, it represents an initial overhead of $\Theta(ed^2)$. However, for similar predicates and similar signatures of constraints (i.e. similar Cartesian products built from the domains associated with the variables involved in the constraints), sub-arrays of $bitSup$ can be shared, potentially saving a large amount of space and initial constraint checks.

The worst-case space complexity of the binary representation of constraints is $\Theta(ed^2)$ whereas the best-case space complexity is $\Theta(d^2)$, which corresponds to sharing the same binary representation between all constraints. The worst-case rather corresponds to unstructured (random) instances whereas the best-case to structured (academic or real-world) instances which usually involve similar constraints.

3.3 Exploiting Binary Representations

We can now exploit the binary representations of domains and constraints to efficiently achieve some computations by using bitwise operators. We illustrate our purpose in three different contexts. Note that for any array t, $t[1]$ will denote its first element and $t.length$ its size.

First, the following sequence of instructions can be used to determine whether the domain of a variable X is a subset of the domain of another variable Y (such that $|dom^{init}(X)| = |dom^{init}(Y)|$):

foreach $i \in \{1, \ldots, bitDom[X].length\}$ **do**
  if $(bitDom[X][i]$ ***OR*** $bitDom[Y][i]) \neq bitDom[Y][i]$ **then**
    return *false*
return *true*

This kind of computation can be interesting, for example, when implementing a symmetry breaking method by dominance detection, e.g. (Focacci and Milano, 2001; Fahle et al., 2001). In that case, we can compare the current domain of a variable with one that was recorded earlier, potentially from the same variable. It can then be useful to efficiently determine if one state is dominated by another one.

Second, the following sequence of instructions can be used to determine if a value (X, a) is neighborhood-substitutable by a value (X, b) with respect to a constraint C (involving X):

foreach $i \in \{1, \ldots, bitDom[X].length\}$ **do**
  if $(bitSup[C, X, a][i]$ ***OR*** $bitSup[C, X, b][i]) \neq bitSup[C, X, b][i]$ **then**
    return *false*
return *true*

Neighborhood substitutability has been introduced by Freuder (1991) and is defined as follows: given a variable X, two values a and b in $dom(X)$ and a constraint C, (X, a) is neighborhood-substitutable by (X, b) w.r.t. C iff the set of supports of a for X in C is a subset (or equal to) of the set of supports of b for X in C. The code presented above can be useful in practice to reduce the search space by eliminating neighborhood-substitutable values (e.g. see Bellicha et al. (1994); Cooper (1997)).

Finally, the following sequence of instructions can be used to determine if a value (X, a) admits at least one support in a constraint C (involving X and a second variable Y):

foreach $i \in \{1, \ldots, bitDom[Y].length\}$ **do**
  if $(bitSup[C, X, a][i]$ *AND* $bitDom[Y][i]) \neq ZERO$ **then**
    return *true*
return *false*

Note that $ZERO$ denotes a word defined as a sequence of bits all set to 0. This way of seeking a support was initially mentioned by McGregor (1979).

Interestingly enough, for all operations described above, it is sometimes possible to return a Boolean answer even if all elements of the domains have not been iterated. For example, for all three computations described above, it is possible to obtain a result at the first use of a bitwise operator (i.e. for $i = 1$). Certainly, this seems natural but one should be aware that using bit vectors to perform a bitwise operation, and then compare the result with another bit vector can be quite more expensive.

4. A Simple Optimization of AC3

In this section, we show how to simply adapt the algorithm AC3 in order to exploit bitwise operators. The new algorithm, denoted AC3^{bit}, is expected to save a large amount of operations (constraint checks) and consequently, CPU time.

To establish arc consistency on a given CN, we call the function $enforceAC$ (Algorithm 1). 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, and when a revision is effective (at least one value has been removed), the set Q has to be updated. A revision is performed by a call to the function $revise$ specific to the chosen coarse-grained arc consistency algorithm, and entails removing values that have become inconsistent with respect to C. This function returns $true$ when the revision is effective. The algorithm is stopped when the set Q becomes empty.

4.1 AC3

For AC3 (Mackworth, 1977), each revision is performed by a call to the function $revise(C, X)$, depicted in Algorithm 2. This function iteratively calls, for any value $a \in dom(X)$, the function $seekSupportAC3$ which determines from scratch whether or not there exists a support of (X, a) in C. If no such support exists, the value (X, a) can be removed. The principle used in $seekSupportAC3$ (see Algorithm 3) is to iterate the list of current values of $dom(Y)$ in order to find a support. Note that $(a, b) \in rel(C)$ must be understood as a constraint check.

AC3 has a non-optimal worst-case time complexity of $O(ed^3)$ (Mackworth and Freuder, 1985). However, as shown by Lecoutre and Hemery (2007), it is possible to refine this result by focusing on the cumulated cost of seeking successive supports of a value (X, a) in a constraint C.

4.2 AC3^{bit}

For the algorithm we propose, AC3^{bit}, each revision is also performed by a call to the function $revise(C, X)$, depicted in Algorithm 2. However, instead of calling $seekSupportAC3$, we use the function $seekSupportAC3^{bit}$ (see Algorithm 4). Given the binary representation $bitDom[Y]$ of

Algorithm 1: enforceAC (P = $(\mathscr{X}, \mathscr{C})$: Constraint Network) : 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', X') | C' ∈ 𝒞, C' ≠ C ∧ scp(C') = {X, X'}}
7 return true
```

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

```
1 nbElements ← |dom(X)|
2 foreach a ∈ dom(X) do
3     if ¬seekSupport(C, X, a) then remove a from dom(X)
4 return nbElements ≠ |dom(X)|
```

Algorithm 3: seekSupportAC3(C, X, a): Boolean

```
1 Let Y be the variable such that scp(C) = {X, Y}
2 foreach b ∈ dom(Y) do
3     if (a, b) ∈ rel(C) then return true
4 return false
```

Algorithm 4: seekSupportAC3^{bit}(C, X, a): Boolean

```
1 Let Y be the variable such that scp(C) = {X, Y}
2 foreach i ∈ {1, ..., bitDom[Y].length} do
3     if (bitSup[C, X, a][i] AND bitDom[Y][i]) ≠ ZERO then return true
4 return false
```

Algorithm 5: seekSupportAC3^{bit+rm}(C, X, a): Boolean

```
1 Let Y be the variable such that scp(C) = {X, Y}
2 i ← residue[C, X, a]
3 if (bitSup[C, X, a][i] AND bitDom[Y][i]) ≠ ZERO then return true
4 foreach i ∈ {1, ..., bitDom[Y].length} do
5     if (bitSup[C, X, a][i] AND bitDom[Y][i]) ≠ ZERO then
6         residue[C, X, a] ← i
7         return true
8 return false
```

$dom(Y)$ and the binary representation $bitSup[C, X, a]$ of the (initial) supports of (X, a) wrt C, we just have to execute the code presented in Section 3.3.

Proposition 4 *The worst-case time complexity of AC3^{bit} is $O(ed^3)$.*

The proof is immediate. Interestingly, one can make the following observation that indicates that in practice, AC3^{bit} can be far more efficient than the other AC3-based variants. It suffices to consider the illustration given in introduction.

Observation 1 *The number of bitwise operations performed by AC3^{bit} can be up to x times less than the number of constraint checks performed by AC3, AC2001 and AC3^{rm}, where x is the word size of the computer.*

5. Experiments

To show the interest of the algorithm introduced in this paper (and more generally, the practical interest of dealing with bitwise operations), we have performed a vast experimentation (ran on a computer equipped with a 2.4GHz i686 Intel CPU, 512MiB of RAM and Sun JRE 5.0 for Linux) with respect to random, academic and real-world problems[2]. Performances[3] have been measured in terms of the CPU time in seconds (cpu) and the amount of memory in mebibytes (mem).

We have implemented the different arc consistency algorithms AC3, AC2001, AC3^{rm} and AC3^{bit} in our platform Abscon. We have compared them by using the algorithm that maintains arc consistency during the search of a solution (MAC). All AC algorithms benefit from the *support condition* mechanism corresponding to Proposition 1 of (Boussemart et al., 2004b) and Equation 1 of (Mehta and van Dongen, 2005a). It allows us to avoid some useless revisions and constraint checks. For search, the variable ordering heuristic was $dom/wdeg$ (Boussemart et al., 2004a), and the value ordering heuristic $min\text{-}conflicts$ (a static variant as presented by Mehta and van Dongen (2005b)). We did not use any restart policy.

To start, we have considered 7 classes of binary random instances, generated using Model D and situated at the phase transition of search (it means that about half of the instances are satisfiable). For each class $\langle n, d, e, t\rangle$, the number of variables n has been set to 40, the domain size d set between 8 and 180, the number of constraints e between 753 and 84 (and, so the density between 0.1 and 0.96) and the tightness t, which here denotes the probability that a relation forbids a pair of values, between 0.1 and 0.9. The first class $\langle 40, 8, 753, 0.1\rangle$ corresponds to dense instances involving constraints of low tightness whereas the seventh one $\langle 40, 180, 84, 0.9\rangle$ corresponds to sparse instances involving constraints of high tightness. In Table 2, one can observe that even for small domains (e.g. $d = 8$), MAC3^{bit} is the fastest algorithm. Interestingly, MAC3^{bit} is 2 to 4 times faster than MAC2001 and 1.5 to 3 times faster than MAC3^{rm}. For this first experiment, we also provide the number of constraints checks ($\#ccks$) and validity checks ($\#vcks$). However, for MAC3^{bit}, note that $\#ccks$ corresponds to the number of bitwise operations.

The good behavior of MAC3^{bit} is confirmed on different series of structured instances. Indeed, in Table 3, we can see that, once again, MAC3^{bit} outperforms the other algorithms. This is particularly true for the job-shop instances of series $enddr1$ and $enddr2$. This can be explained by the fact

2. `http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html`
3. In our experimentation, all constraint checks are performed in constant time and are as cheap as possible since constraints are represented in extension using arrays.

		MAC embedding			
		$AC2001$	$AC3$	$AC3^{rm}$	$AC3^{bit}$
$\langle 40, 8, 753, 0.1\rangle$	cpu	13.8	9.8	10.4	7.7
	mem	11	9.5	10	9.5
	$\#ccks$	$13M$	$15M$	$8.7M$	$33M$
	$\#vcks$	$2.7M$	0	$14M$	0
$\langle 40, 11, 414, 0.2\rangle$	cpu	19.6	15.0	14.5	10.0
	mem	8.8	8.0	8.4	8.0
	$\#ccks$	$30M$	$41M$	$21M$	$63M$
	$\#vcks$	$14M$	0	$35M$	0
$\langle 40, 16, 250, 0.35\rangle$	cpu	21.6	18.5	16.1	9.7
	mem	8.5	7.9	8.2	7.9
	$\#ccks$	$48M$	$80M$	$34M$	$78M$
	$\#vcks$	$35M$	0	$58M$	0
$\langle 40, 25, 180, 0.5\rangle$	cpu	28.9	27.8	21.2	11.5
	mem	8.4	7.9	8.2	7.9
	$\#ccks$	$89M$	$169M$	$63M$	$112M$
	$\#vcks$	$70M$	0	$100M$	0
$\langle 40, 40, 135, 0.65\rangle$	cpu	21.1	22.0	15.4	7.8
	mem	8.5	8.0	8.2	8.1
	$\#ccks$	$92M$	$183M$	$68M$	$88M$
	$\#vcks$	$59M$	0	$81M$	0
$\langle 40, 80, 103, 0.8\rangle$	cpu	16.6	19.5	12.2	5.0
	mem	10	9.5	9.8	9.6
	$\#ccks$	$106M$	$226M$	$80M$	$81M$
	$\#vcks$	$48M$	0	$62M$	0
$\langle 40, 180, 84, 0.9\rangle$	cpu	24.3	36.6	18.4	6.7
	mem	15	14	14	14
	$\#ccks$	$256M$	$629M$	$199M$	$157M$
	$\#vcks$	$76M$	0	$93M$	0

Table 2: Mean results on random instances; 100 instances per class, cpu time given in seconds and mem(ory) in MiB.

that the average domain size for these instances is about 120 values, which means that on a 64-bit processor, only two main operations are required when seeking a support.

Finally, we present the results obtained on some hard academic and real-world instances. The interest of using AC3^{bit} clearly appears on an instance such as $knights$-50-25. What is also interesting to observe is that the gap between AC3^{bit} and the other algorithms increases with the difficulty of the instances of the series $scen11$-fX. Indeed, whereas all algorithms behave similarly w.r.t. the easy instance $scen11$-$f10$, AC3^{bit} is twice faster than the other AC algorithms w.r.t. the more difficult instance $scen11$-$f4$. The trend clearly appears when looking at results obtained for the intermediate instances $scen11$-$f8$ and $scen11$-$f6$.

What about residues? At this point, one can wonder if there is still an interest of exploiting residues for binary instances. Indeed, for domains up to 300 values, checking if a cn-value admits a support requires less than 5 operations (on a 64-bit architecture). That was the case for most of the series/instances presented above, and consequently, AC3^{bit} was always faster than AC3^{rm}. However, when domains become larger, it can become penalizing to exploit bitwise operations alone. This is why we propose to combine them with residues. The principle is the following: whenever a support is detected, its position in the binary representation of the constraint is recorded. Introducing a three-dimensional array $residue$ of integers (all set to 0 initially), we can then use

		MAC embedding			
		$AC2001$	$AC3$	$AC3^{rm}$	$AC3^{bit}$
blackHole-4-4 (10 instances)	cpu	1.46	1.37	1.35	0.91
	mem	8.6	7.9	8.7	7.9
driver (7 instances)	cpu	3.89	2.99	3.14	2.75
	mem	35	24	56	24
ehi-85 (100 instances)	cpu	1.75	0.92	1.12	0.71
	mem	30	19	38	19
ehi-90 (100 instances)	cpu	1.73	0.91	1.11	0.72
	mem	31	20	39	20
jobshop enddr1 (10 instances)	cpu	1616	1694	1218	453
	mem	14	13	14	13
jobshop enddr2 (6 instances)	cpu	1734	2818	1491	568
	mem	15	14	15	14
geom (100 instances)	cpu	12.4	10.8	8.9	5.8
	mem	11	10	11	10
hanoi (5 instances)	cpu	1.00	1.16	1.11	0.50
	mem	13	11	12	12
qwh-20 (10 instances)	cpu	266	183	242	153
	mem	33	21	44	21

Table 3: Mean results on series of structured instances; cpu time given in seconds and mem(ory) in MiB.

		MAC embedding			
		$AC2001$	$AC3$	$AC3^{rm}$	$AC3^{bit}$
Academic instances					
knights-50-9	cpu	85	1148	109	36
	mem	27	23	23	23
knights-50-25	cpu	> 1200	> 1200	> 1200	211
	mem				28
pigeons-11	cpu	54.6	53.4	57.4	43.5
	mem	21	21	21	21
pigeons-12	cpu	656	547	591	484
	mem	21	21	21	21
queenAttacking-6	cpu	123	125	128	79
	mem	21	21	25	21
queenAttacking-7	cpu	407	436	381	263
	mem	25	22	25	22
Real-world instances					
e0ddr2-10-by-5-1	cpu	257	316	177	68
	mem	23	23	23	23
enddr2-10-by-5-1	cpu	178	263	143	61
	mem	23	23	23	23
scen11-f10	cpu	5.0	5.4	5.7	5.5
	mem	33	29	45	29
scen11-f8	cpu	11.4	11.1	11.5	9.0
	mem	33	29	45	29
scen11-f6	cpu	81.7	75.6	74.7	47.3
	mem	33	29	45	29
scen11-f4	cpu	1250	1233	1106	670
	mem	33	29	45	29

Table 4: Results on hard structured instances ; cpu time given in seconds and mem(ory) in MiB.

Algorithm 5. When looking for a support, the residual position is first checked (line 3), and when one is found, its position is recorded (line 6).

To illustrate the importance of combining bitwise operations with residues when domains are large, we show in Table 5 the results obtained on instances of the Domino problem. This problem has been introduced in Zhang and Yap (2001) to emphasize the sub-optimality of AC3. Each instance, denoted $domino$-n-d, corresponds to an undirected constraint graph with a cycle. More precisely, n denotes the number of variables, the domains of which are $\{1, \ldots, d\}$, and there exists $n-1$ equality constraints $X_i = X_{i+1}$ ($\forall i \in \{1, \ldots, n-1\}$) and a trigger constraint $(X_1 = X_n + 1 \wedge X_1 < d) \vee (X_1 = X_n \wedge X_1 = d)$. For the most difficult instance, where domains contain 3000 values, AC3^{bit+rm} is about 5 times more efficient than AC3^{bit} and AC3^{rm}, and 9 times more efficient than AC2001.

$Instances$		$AC2001$	$AC3$	$AC3^{rm}$	$AC3^{bit}$	$AC3^{bit+rm}$
domino-500-500	cpu	12.7	403	9.4	4.3	3.7
	mem	$27M$	$23M$	$27M$	23	23
domino-800-800	cpu	48.4	2,437	34.5	13.4	8.7
	mem	$49M$	$33M$	$41M$	$33M$	$33M$
domino-1000-1000	cpu	89.5	5,911	62.4	25.1	14.3
	mem	$66M$	$42M$	$54M$	$42M$	$46M$
domino-2000-2000	cpu	678	$> 5h$	443	289	91
	mem	$210M$		$156M$	$117M$	$132M$
domino-3000-3000	cpu	2,349	$> 5h$	1,564	1,274	278
	mem	$454M$		$322M$	$240M$	$275M$

Table 5: Establishing Arc Consistency on Domino instances

6. Conclusion

In this paper, we have introduced a precise description of the exploitation of bitwise operations to improve the basic arc consistency algorithm AC3. The result is a new algorithm, denoted AC3^{bit}, which appears to be approximately twice more efficient than AC3^{rm}, an algorithm shown itself to be faster than the optimal AC2001. We have also shown how to combine bitwise operations with residues, which happens to be quite useful when domains become large (more than 300 values). We do believe that, for solving binary instances, when constraints are given in extension or can be efficiently converted into extension, the generic algorithm MAC, embedding AC3^{bit}/AC3^{bit+rm} is the most efficient approach. One reason is that, like MAC3^{rm}, no maintenance of data structures is required upon backtracking by MAC3^{bit}/MAC3^{bit+rm},

Finally, note that MAC3^{bit}/MAC3^{bit+rm} is the algorithm used by the solver $Abscon109$ that has participated to the second international competition of CSP solvers[4]. More precisely, it was used for binary instances involving constraints in extension and constraints in intention that could be converted efficiently into extension. For example, all (constraints of all) instances of the Radio Link Frequency Assignment Problem (RLFAP) were converted in less than 0.5 second. The good results that have been obtained by our Java-written Abscon solver during this competition indirectly confirm the results of this paper.

4. http://www.cril.univ-artois.fr/CPAI06

References

A. Bellicha, C. Capelle, M. Habib, T. Kokény, and M.C. Vilarem. CSP techniques using partial orders on domain values. In *Proceedings of ECAI'94 workshop on constraint satisfaction issues raised by practical applications*, 1994.

C. Bessiere and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.

C. Bessiere, E.C. Freuder, and J. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

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

F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004a.

F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Support inference for generic filtering. In *Proceedings of CP'04*, pages 721–725, 2004b.

M.C. Cooper. Fundamental properties of neighbourhood substitution in constraint satisfaction problems. *Artificial Intelligence*, 90:1–24, 1997.

T. Fahle, S. Schamberger, and M. Sellman. Symmetry breaking. In *Proceedings of CP'01*, pages 93–107, 2001.

F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of CP'01*, pages 77–92, 2001.

E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI'91*, pages 227–233, 1991.

R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

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

C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.

C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.

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

C. Lecoutre, S. Cardon, and J. Vion. Conservative dual consistency. In *Proceedings of AAAI'07*, pages 237–242, 2007.

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.

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

A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.

D. Mehta and M.R.C. van Dongen. Reducing checks and revisions in coarse-grained MAC algorithms. In *Proceedings of IJCAI'05*, pages 236–241, 2005a.

D. Mehta and M.R.C. van Dongen. Two new lightweight arc consistency algorithms. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 109–123, 2004.

D. Mehta and M.R.C. van Dongen. Static value ordering heuristics for constraint satisfaction problems. In *Proceedings of CPAI'05 workshop held with CP'05*, pages 49–62, 2005b.

R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

B.A. Nudel. Consistent-labeling problems and their algorithms: expected-complexities and theory based heuristics. *Artificial Intelligence*, 21(1-2):135–178, 1983.

D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.

C. Schulte. Comparing trailing and copying for constraint programming. In *Proceedings of ICLP'99*, pages 275–289, 1999.

J.R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

M.R.C. van Dongen. Beyond singleton arc consistency. In *Proceedings of ECAI'06*, pages 163–167, 2006.

M.R.C. van Dongen. $AC3_d$ an efficient arc consistency algorithm with a low space complexity. In *Proceedings of CP'02*, pages 755–760, 2002.

R.J. Wallace. Why AC3 is almost always better than AC4 for establishing arc consistency in CSPs. In *Proceedings of IJCAI'93*, pages 239–245, 1993.

Y. Zhang and R.H.C. Yap. Making AC3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321, 2001.