Int. J. Appl. Math. Comput. Sci., 2011, Vol. 21, No. 4, 733–744
DOI: 10.2478/v10006-011-0058-2

A FINE–GRAINED ARC–CONSISTENCY ALGORITHM FOR NON–NORMALIZED CONSTRAINT SATISFACTION PROBLEMS

MARLENE ARANGÚ, MIGUEL A. SALIDO

Institute of Industrial Informatics and Automatics
Technical University of Valencia, Camino de Vera, s/n. 46020 Valencia, Spain
e-mail: {marangu,msalido}@dsic.upv.es

Constraint programming is a powerful software technology for solving numerous real-life problems. Many of these problems can be modeled as Constraint Satisfaction Problems (CSPs) and solved using constraint programming techniques. However, solving a CSP is NP-complete so filtering techniques to reduce the search space are still necessary. Arc-consistency algorithms are widely used to prune the search space. The concept of arc-consistency is bidirectional, i.e., it must be ensured in both directions of the constraint (direct and inverse constraints). Two of the most well-known and frequently used arc-consistency algorithms for filtering CSPs are AC3 and AC4. These algorithms repeatedly carry out revisions and require support checks for identifying and deleting all unsupported values from the domains. Nevertheless, many revisions are ineffective, i.e., they cannot delete any value and consume a lot of checks and time. In this paper, we present AC4-OP, an optimized version of AC4 that manages the binary and non-normalized constraints in only one direction, storing the inverse founded supports for their later evaluation. Thus, it reduces the propagation phase avoiding unnecessary or ineffective checking. The use of AC4-OP reduces the number of constraint checks by 50% while pruning the same search space as AC4. The evaluation section shows the improvement of AC4-OP over AC4, AC6 and AC7 in random and non-normalized instances.

Keywords: constraint satisfaction problems, filtering techniques, consistency algorithms.

1. Introduction

Over the last few years, many real problems have involved constraints that the solution must satisfy (Królikowski and Jerzy, 2001; Sikora, 2003). These problems come from different areas of computer sciences such as planning (Barták *et al.*, 2010), scheduling (Brdyś and Littler, 2002; Mesghouni *et al.*, 2004), control (Deng *et al.*, 2009), etc. To this end, many algorithms and heuristic techniques have been developed to manage these problems. Constraint programming is a software technology for description and effective solving of large and complex problems, particularly combinatorial problems in many areas of real life (Dechter, 2003; Barták, 1999).

The basic idea of a Constrained Satisfaction Problem (CSP) is to model the problem as a set of variables with finite domains (the values for the variables) and a set of constraints that impose a limitation on the values that a variable, or a combination of variables, may be assigned. The task is to find an assignment of values to the variables that satisfy all the constraints. In general, the tasks posed in the CSP paradigm are computationally intractable (NP-complete) so filtering techniques to simplify the search space are still necessary.

The consistency-enforcing algorithm performs any partial solution of a small sub-network that is extensible to a surrounding network. The number of possible combinations can be huge, while only very few may be consistent. By eliminating redundant values from the problem definition, the size of the solution space decreases. If any domain becomes empty as a result of reduction, then it is immediately known that the problem has no solution (Ruttkay, 1998).

In this paper, we focus our attention on arc-consistency. It is the basic propagation mechanism that is probably used in all solvers (Bessiere, 2006). Arc-consistency algorithms are based on the notion of a support. These algorithms ensure that each value in the domain of each variable is supported by some value in the domain of each variable by which it is constrained. Arc-consistency algorithms are a major component of many industrial and academic CSP solvers. Sitting at the heart of a CSP solver, arc-consistency algorithms consume a large portion of the time that is required to solve the input CSP (van Dongen *et al.*, 2008).

Proposing efficient algorithms for enforcing arc-consistency has always been considered a central question in the constraint reasoning community. Thus, there are many arc-consistency algorithms such as AC1, AC2, and AC3 (Mackworth, 1977), AC4 (Mohr and Henderson, 1986), AC5 (Perlin, 1992; Hentenryck *et al.*, 1992), AC6 (Bessiere and Cordier, 1993; Bessiere, 1994), AC7 (Bessiere *et al.*, 1999), AC8 (Chmeiss and Jegou, 1998), AC2001 (Bessiere *et al.*, 2005), AC2001-OP (Arangu *et al.*, 2010), and more. However, AC3 (Mackworth, 1977) and AC4 (Mohr and Henderson, 1986) are most widely used (Barták, 2005).

Algorithms that perform arc-consistency have focused their improvements on time-complexity and space-complexity. Main improvements have been achieved by changing the way of propagation: from arcs to values (i.e., changing the granularity: coarse-grained to fine-grained), appending new structures, performing bidirectional searches (AC7), changing the support search: searching for all supports (AC4) or searching for only the necessary supports (AC6, AC7, AC2001), improving the propagation (i.e., it performs propagation only when necessary, AC7 and AC-2001), etc. However, main arc-consistency techniques are focused on normalized CSPs, that is, those where any pair of variables can be restricted to no more than one constraint. Nevertheless, many real problems are modeled with constraints that involve the same set of variables (non-normalized CSPs), so it is necessary to develop filtering techniques to manage these resultant CSPs.

AC4 is the first fine-grained algorithm where the propagation is value oriented. AC4 is the only algorithm that confirms the existence of a support by not identifying it throughout search (Mehta, 2008). To do this, AC4 stores lots of information about all the supports for each value in the auxiliary data structures. The main reasons for studying AC4 are the following:

1. It has optimal time-complexity (Bessiere, 2006; Mohr and Henderson, 1986).
2. It is one of the most widely and frequently used algorithms for maintaining arc-consistency (Barták, 2005).

Despite maintaining huge data structures during search, we have detected inefficiencies when these data structures are updated:

(a) It is not necessary to check each constraint in both directions when the checking of the constraint in one direction can store all the information.

(b) There are propagations of the pruned tuples that are ineffective because the variables do not store any variable support.

(c) In non-normalized problems the way that these structures are updated changes.

We propose an optimized version of AC4 by means of the management of the constraints in only one direction rather than in both directions. Furthermore, our algorithm allows the processing of non-normalized CSPs. Thus, this paper is organized as follows. In the next section, we provide the necessary definitions to understand the rest of the paper. Then, we explain the AC4 algorithm in detail, and we present our AC4-OP algorithm. In the experimental results section, we evaluate AC4 and AC4-OP empirically and, finally, we present our conclusions.

1.1. Definitions.

By following the standard notation and definitions in the literature (Bessiere, 2006; Barták, 2001; Dechter, 2003), we summarize the basic definitions and notation used in this paper.

Definition 1. *Constraint Satisfaction Problems (CSPs)* are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. Formally a CSP is a triple $P = \langle X, D, R\rangle$, where X is a finite set of variables $\{X_1, X_2, \dots, X_n\}$, D is a set of domains $D = \{D_1, D_2, \dots, D_n\}$ such that, for each variable $X_i \in X$, there is a finite set of values that each variable can take, and R is a finite set of constraints $R = \{R_1, R_2, \dots, R_m\}$ which restrict the values that the variables can simultaneously take.

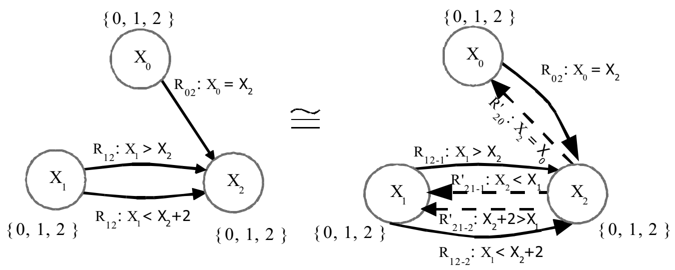
Definition 2. *Normalized CSP.* A CSP is normalized iff two different constraints in R do not involve exactly the same variables. A CSP is non-normalized if different constraints may involve exactly the same variables.

Definition 3. *Binary CSP*. A CSP is binary iff all constraints in R involve two variables.

In this paper, we limit our attention to binary and non-normalized CSPs. Thus, we can represent the CSP as a directed graph, where variables are represented as nodes and binary constraints correspond to directed arcs (see Fig. 1). To this end, we write

- R_{ij} as the direct constraint defined over the variables X_i and X_j (given by the user),
- R'_{ji} as the same constraint in the inverse direction over the variables X_i and X_j (inverse constraint)[1].

[1] In the work of Bessiere (2006) the inverse constraint is named transposition.



Initial values of R

R_{ij}	Constraint	direction
R_{02}	$X_0 = X_2$	direct
R'_{20}	$X_2 = X_0$	inverse
R_{12-1}	$X_1 > X_2$	direct
R'_{21-1}	$X_2 < X_1$	inverse
R_{12-2}	$X_1 < X_2 +2$	direct
R'_{21-2}	$X_2 +2 > X_1$	inverse

Fig. 1. Example of a binary and non-normalized CSP.

Figure 1 (left) shows a non-normalized and binary CSP with three variables X_0, X_1 and X_2, with domains $D_0 = D_1 = D_2 = \{0, 1, 2\}$ and three direct constraints

$$\begin{aligned} R_{02} &: X_0 = X_2, \\ R_{12-1} &: X_1 > X_2, \\ R_{12-2} &: X_1 < X_2 + 2. \end{aligned}$$

It can be observed that there are two different constraints R_{12-1} and R_{12-2} between the variables X_1 and X_2. Furthermore, it can be distinguished between $R_{12-1} : X_1 > X_2$ as a direct constraint and $R'_{21-1} : X_2 < X_1$ as the same constraint in the inverse direction (see Fig. 1, right).

Definition 4. *Instantiation* is a pair $\langle X_i, a\rangle$ that represents an assignment of the value a to the variable X_i, and a is in the domain of X_i.

Definition 5. *Satisfying a constraint.* A constraint R_{ij} is satisfied if the instantiation of $\langle X_i, a\rangle$ and $\langle X_j, b\rangle$ is legal for this constraint $(\langle X_i, a\rangle, \langle X_j, b\rangle) \in R_{ij}$.

Definition 6. *Arc-consistency.* A *value* $a \in D_i$ is arc-consistent with respect to a constraint R_{ij} iff there exists a value $b \in D_j$ such that $\langle X_i, a\rangle$ and $\langle X_j, b\rangle$ satisfy the constraint R_{ij}. A *variable* X_i *is arc-consistent* relative to X_j iff all values in D_i are arc-consistent. A *CSP is arc-consistent* iff all the variables are arc-consistent, e.g., all the arcs R_{ij} and R'_{ji} are arc-consistent. (Note that here we are talking about full arc-consistency.) A CSP is arc-inconsistent if one (or more) of the variables is not arc-consistent.

Definition 7. *Support value with respect to a constraint* R_{ij}. Given $a \in D_i$ and $b \in D_j$, if $\langle X_i, a\rangle$ and $\langle X_j, b\rangle$ satisfy the constraint R_{ij}, then b supports a. Thus, if the value $b \in D_j$ supports a value $a \in D_i$, then a supports b as well (the symmetry of the constraint).

Definition 8. *Number of constraint checkings.* (*checks*) is the number of times a given pair is checked with respect to some constraint $R_{ij} \in R$.

Definition 9. *Number of propagations.* (*Np*) is the number of times that a tuple $\langle X_i, a\rangle$ is added to a structure (i.e., queue, stack, set, etc.) for its re-evaluation.

2. Algorithm AC4

The algorithm AC4 is one of the most widely used algorithms for arc-consistency (see Algorithms 1 and 2 for non-normalized CSPs).[2] It was proposed to improve the time complexity of AC3 (Mohr and Henderson, 1986). It is the first algorithm in a category called *fine-grained algorithms* because it performs propagations at the level of values (Bessiere, 2006). AC4 stores the maximum amount of information in a preprocessing step in order to avoid having to redo the same constraint checking several times during the propagation of deletions. Moreover, AC4 has an optimal time complexity $O(ed^2)$, and it has an $O(ed^2)$ space complexity, where e is the number of binary constraints (edges) and d the domain size in the problem. Also, AC4 has influenced other arc-consistency algorithms like AC6 and AC7.

The procedure InitializeAC4 is very close to the original procedure of AC4. The only difference is focused on the matrix $Counter$, which must be increased with the variable *total* ($Counter = Counter + total$ instead of $Counter = total$).

In order to perform only once the constraint checking in R and to identify the relevant values that are needed to be re-examined, AC4 stores the following data structures:

- ***S*** is a matrix $S[X_j, b]$ that contains a list of pairs $\langle X_i, a\rangle$ that are supported by $\langle X_j, b\rangle$. The pair $\langle X_i, a\rangle$ may appear several times in the list kept in matrix S.

- ***Counters*** is a matrix $Counter[X_i, a, X_j]$ that contains the number of supports for the value $a \in D_i$ in the variable X_j.

[2]In order to remove ambiguity and improve efficiency, Algorithms 1 and 2 combine the encoding of Mohr and Henderson (1986), Tsang (1995), Barták (2001), Dechter (2003), and Bessiere (2006).

Algorithm 1: Procedure InitializeAC4.

```
Data: $P = \langle X, D, R\rangle$ /*$R$ involves direct and inverse constraints*/
Result: $initial$ =true and $P'$, $Q$, $S$, $M$, $Counter$ OR
        $initial$ =false and $P'$ (which is arc-inconsistent).
begin
1   $Q \leftarrow \{\}$
2   $S[X_j, b] \leftarrow \{\}$ $/*\forall X_j \in X \wedge \forall b \in D_j */$
3   $M[X_i, a] = 1$ $/*\forall X_i \in X \wedge \forall a \in D_i */$
4   $Counter[X_i, a, X_j] = 0$ $/*\forall X_i, X_j \in X, i \neq j \wedge \forall a \in D_i */$
5   for every arc $R_{ij} \in R$ do
6       for each $a \in D_i$ do
7           $total = 0$
8           for each $b \in D_j$ do
9               if $(\langle X_i, a\rangle, \langle X_j, b\rangle) \in R_{ij}$ then
10                  $total = total + 1$
11                  $Append(S[X_j, b], \langle X_i, a\rangle)$
12          if $total = 0$ then
13              remove $a$ from $D_i$
14              if $D_i = \phi$ then
                    return $initial$ = false
15              $Q \leftarrow Q \cup \langle X_i, a\rangle$
16              $M[X_i, a] = 0$
17          else
                $Counter[X_i, a, X_j] = Counter[X_i, a, X_j] + total$
    return $initial$ = true and $Q$, $M$, $S$, $Counter$
end
```

Algorithm 2: Procedure AC4.

```
Data: A CSP $P = \langle X, D, R\rangle$
Result: true and $P'$ (which is arc-consistent) or false and $P'$ (which
        is arc-inconsistent)
begin
1   InitilizeAC4(P)
2   if $initial = true$ then
3       while $Q \neq \phi$ do
4           select and delete $\langle X_j, b\rangle$ from queue $Q$
5           for each $\langle X_i, a\rangle \in S[X_j, b]$ do
6               $Counter[X_i, a, X_j] = Counter[X_i, a, X_j] - 1$
7               if $Counter[X_i, a, X_j] = 0 \wedge M[X_i, a] = 1$ then
8                   remove $a$ from $D_i$
9                   if $D_i = \phi$ then
                        return false
10                  $Q \leftarrow Q \cup \langle X_i, a\rangle$
11                  $M[X_i, a] = 0$
12      return true
13  else
        return false
end
```

- ***M*** is a matrix $M[X_i, a]$ that stores 1 if $a \in D_i$ or stores 0 if $a \notin D_i$ (indicating that $\langle X_i, a\rangle$ has been deleted).

- ***Q*** is a queue that stores pairs $\langle X_i, a\rangle$ (rejected values) awaiting further processing.

The main algorithm AC4 (see Algorithm 2) has two phases: initialization of the data structures (Step 1 calls to Algorithm 1) and propagation (Steps 3–13). Algorithm 1 is committed to initializing the above structures. These initializations are used to remember pairs of consistent values of variables (matrix S), to count "supporting" values from the domain of the variable (matrix $Counter$), to remove those values that do not have any support and to remember them (matrix M and queue Q).

The number of supporting values found, for each value $a \in D_i$, is initially stored in a local variable called *total*. If there is no supporting value for a, (i.e., $total = 0$), then a is removed from D_i and both matrix M and queue Q are updated (see Algorithm 1, Steps 12–16). Otherwise, the matrix $Counter$ is increased with the total value (see Algorithm 1, Step 17). In this step of the algorithm, the matrix $Counter$ stores the number of supports of each value for normalized and non-normalized constraints.

Once Algorithm 1 has finished, Algorithm 2 begins the propagation phase. This process propagates the consequences of the removal of values. Thus, the pair ⟨variable, value⟩ stored in Q is selected and revised until no change occurs (Q is empty) or the domain of a variable remains empty. In the first case, the algorithm ensures that all values in the domains are consistent with all constraints, while, in the second case the algorithm returns that the problem has no solution.

For the example presented in Fig. 1 (right), R stores the constraints in both directions (direct constraints and inverse constraints). Table 1 shows how the procedure `InitializeAC4` evaluates each constraint. The tuples that must be re-evaluated ($\langle X_1, 0\rangle$ and $\langle X_2, 2\rangle$) were added to the queue Q (See Table 1, Loops 3 and 4). The values for M, S and *Counter* (after the initialization phase) are shown in the column *Ini* of Table 3 (left), Table 3 (right) and Table 4, respectively.

After `InitilizationAC4`, the propagation is carried out with the tuples previously stored in Q. Table 2 shows this process. It can be observed that Loop 1 is an inefficient propagation since $\langle X_1, 0\rangle$ does not support any variable because $S[X_1, 0]$ in Table 3 (right) has an empty value in the column Ini.

In the previous example, AC4 performs 3 prunes of domain values, carries out 41 constraint checkings (*Cc*) with Algorithm 1, and carries out 3 propagations (*Np*) in Q with Algorithm 2, to achieve arc-consistency.

3. Algorithm AC4-OP

As we have pointed out above, the following properties can be observed:

(i) Whenever the procedure `InitializationAC4` evaluates a constraint, it stores the information regarding the values of variables, in M and regarding supports in S and $Counter$.

(ii) The direct constraint R_{ij} and the inverse constraint R'_{ji} share the same set of variables (X_i and X_j).

Table 3. Changes in matrix M (left) and matrix S (right) after both the initialize phase (Ini) and the propagation phase (Prop) by the AC4 procedure for the example in Fig. 1.

$M[var, val]$	**Start**	Ini	Prop
$M[X_0, 0]$	1		
$M[X_0, 1]$	1		
$M[X_0, 2]$	1		0
$M[X_1, 0]$	1	0	
$M[X_1, 1]$	1		
$M[X_1, 2]$	1		
$M[X_2, 0]$	1		
$M[X_2, 1]$	1		
$M[X_2, 2]$	1	0	

$S[X_j, b]$	**Start**	Ini
$S[X_0, 0]$	$\{\}$	$\{\langle X_2, 0\rangle\}$
$S[X_0, 1]$	$\{\}$	$\{\langle X_2, 1\rangle\}$
$S[X_0, 2]$	$\{\}$	$\{\langle X_2, 2\rangle\}$
$S[X_1, 0]$	$\{\}$	
$S[X_1, 1]$	$\{\}$	$\{\langle X_2, 0\rangle, \langle X_2, 0\rangle, \langle X_2, 1\rangle\}$
$S[X_1, 2]$	$\{\}$	$\{\langle X_2, 0\rangle, \langle X_2, 1\rangle, \langle X_2, 1\rangle\}$
$S[X_2, 0]$	$\{\}$	$\{\langle X_0, 0\rangle, \langle X_1, 1\rangle, \langle X_1, 2\rangle, \langle X_1, 1\rangle\}$
$S[X_2, 1]$	$\{\}$	$\{\langle X_0, 1\rangle, \langle X_1, 2\rangle, \langle X_1, 1\rangle, \langle X_1, 2\rangle\}$
$S[X_2, 2]$	$\{\}$	$\{\langle X_0, 2\rangle\}$

Table 4. Changes in the matrix $Counter$ after both the initialize phase (Ini) and the propagation phase (Prop) by the procedure AC4 for the example in Fig. 1.

$Counter[X_i, a, X_j]$	**Start**	Ini	Prop		$Counter[X_i, a, X_j]$	**Start**	Ini	Prop
$Counter[X_0, 0, X_2]$	0	1			$Counter[X_1, 0, X_2]$	0	0	
$Counter[X_0, 1, X_2]$	0	1			$Counter[X_1, 1, X_2]$	0	3	
$Counter[X_0, 2, X_2]$	0	1	0		$Counter[X_1, 2, X_2]$	0	3	
$Counter[X_0, 0, X_1]$	0				$Counter[X_2, 0, X_0]$	0	1	
$Counter[X_0, 1, X_1]$	0				$Counter[X_2, 1, X_0]$	0	1	
$Counter[X_0, 2, X_1]$	0				$Counter[X_2, 2, X_0]$	0	1	0
$Counter[X_1, 0, X_0]$	0				$Counter[X_2, 0, X_1]$	0	3	
$Counter[X_1, 1, X_0]$	0				$Counter[X_2, 1, X_1]$	0	3	
$Counter[X_1, 2, X_0]$	0				$Counter[X_2, 2, X_1]$	0	0	

Table 1. Loops carried out by InitializeAC4 for the example shown in Fig. 1.

Loop	Constraint R_{ij}	val a	val b	var total	Prune X_i	Add Q
1	$R_{02} : X_0 = X_2$	0	0	1		
			1	1		
			2	1		
		1	0	0		
			1	1		
			2	1		
		2	0	0		
			1	0		
			2	1		
2	$R'_{20} : X_2 = X_0$	0	0	1		
			1	1		
			2	1		
		1	0	0		
			1	1		
			2	1		
		2	0	0		
			1	0		
			2	1		
3	$R_{12} : X_1 > X_2$	0	0	0		
			1	0		
			2	0	$X_1 = 0$	$\langle X_1, 0\rangle$
		1	0	1		
			1	1		
			2	1		
		2	0	1		
			1	2		
			2	2		
4	$R'_{21} : X_2 < X_1$	0	1	1		
			2	2		
		1	1	0		
			2	1		
		2	1	0		
			2	0	$X_2 = 2$	$\langle X_2, 2\rangle$
5	$R_{12} : X_1 < X_2 + 2$	1	0	1		
			1	2		
		2	0	0		
			1	1		
6	$R'_{21} : X_2 + 2 > X_1$	0	1	1		
			2	1		
		1	1	1		
			2	2		

Table 2. Loops carried out by AC4 (only propagation) for the example in Fig. 1.

Loop	tuple $\langle X_j, b\rangle$	Supports $S[X_j, b]$	Counter $[X_i, a, X_j]$	Prune X_i	Add Q $\langle X_i, a\rangle$
1	$\langle X_1, 0\rangle$	$\{\}$			
2	$\langle X_2, 2\rangle$	$\{\langle X_0, 2\rangle\}$	0	$X_0 = 2$	$\{\langle X_0, 2\rangle\}$
3	$\langle X_0, 2\rangle$	$\{\langle X_2, 2\rangle\}$	0		
		$\langle X_1, 2\rangle$	1		

(iii) By Definition 8 (the symmetry of the constraint), the support is bidirectional.

Due to (i) and (ii), there is an inefficiency in the algorithm AC4 because some values for M, S and $Counter$ might be updated for $\langle X_j, b\rangle$ when a direct constraint is evaluated. At this point, only the values that might be pruned in X_j (if any) are lost because the internal loop is executed several times (e.g., once for each value $a \in D_i$, see Algorithm 1, Steps 8–11). Due to the symmetry of the constraint (Property (iii)), if there is no support ($total = 0$), then Steps 12–16 of Algorithm 1 are performed. However, AC4 only upgrades variable X_i and variable X_j is ignored.

AC4-OP uses the same structures that AC4, but it adds a new array, named $suppInv$, to store the supports of each value of a variable. Thus, the size of $suppInv$ is the maximum size of all domains ($maxD$).

Thus, once the revision of the values $a \in D_i$ is updated ($Counter$) for X_j, all values $b \in D_j$ for which $suppInv[b] = 0$ can be pruned. Thus, it is not needed to evaluate the inverse constraint R'_{ji}.

Therefore, we propose AC4-OP (see Algorithms 3 and 4), an algorithm that takes into account these three ideas to evaluates only direct constraints. Thus, in Algorithm 3 (Steps 4 and 5), both $Counter[X_j, b, X_i]$ and $suppInv[b]$ are initialized. The algorithm adds a new support for $Counter[X_j, b, X_i]$ in Step 13 and it counts a new support for $suppInv[b]$ in Step 14. Finally, the algorithm prunes inconsistent values of D_j in Steps 24–32.

Since the same pair of variables X_i, X_j may be involved in more than one constraint R_{ij} (in non-normalized CSPs), the counters of supports may have previously stored values. Pruning is carried out according to the counters in each constraint. The counter of supports of variable X_i ($total$) is initialized to 0 for each value $a \in D_i$. However, the counter of supports of variable X_j (inverse supports) must be split in two different counters: $Counter[X_j, b, X_i]$ and $suppInv[b]$. The array $suppInv$ stores the number of supports for each value of X_j. This array is initialized to zero (see Algorithm 3, Step 5). If value $b \in D_j$ supports the value $a \in D_i$, $suppInv[b]$ will be increased (see Algorithm 3, Step 14). During the loop of Steps 9–15, this array is updated in order to be analyzed later (in Step 25). Upon completion of processing all values of D_i, if a value b of D_j has no support ($suppInv[b] = 0$), then this value is pruned from the domain D_j. However, if $suppInv[b] > 0$, then b is supported and it is initialized to 0 (see Algorithm 3, Step 32) for further use of this array.

Furthermore, AC4-OP only propagates those tuples that are supported by another one (see Steps 20, 30 of Algorithm 3, and Step 12 of Algorithm 4). Thus, AC4-OP avoids inefficient propagations of tuples for Q, and it avoids inefficient constraint checking of those tuples.

AC4-OP is also valid for normalized CSPs, but some code lines and structures are unnecessary. The array $suppInv$ can be removed and Steps 5, 14 and 32 of Algorithm 3 are not necessary. Thus, in Step 25 of Algorithm 3, $suppInv[b] = 0$ must be changed by $Counter[X_j, b, X_i] = 0$.

Tables 5 and 6 show the initializations and propagations performed by AC4-OP for Example 1. The structures S, M, and $Counter$ for AC4-OP are the same as AC4, but AC4-OP only performs 19 constraint checks for the same problem (47% less than AC4) and 3 propagations (50% less than AC4) while the same pruning is obtained (6 values for this problem).

Algorithm 3: Procedure `InitializeAC4OP`.

```
     Data: $P = \langle X, D, R \rangle$  /*$R$ involves direct constraints*/
     Result: initial=true and $P'$, $Q$, $S$, $M$, $Counter$ or initial=false
             and $P'$ (which is arc-inconsistent).
     begin
 1       $Q \leftarrow \{\}$
 2       $S[X_j, b] \leftarrow \{\}$  $/ * \forall X_j \in X \wedge \forall b \in D_j * /$
 3       $M[X_i, a] = 1$  $/ * \forall X_i \in X \wedge \forall a \in D_i * /$
 4       $Counter[X_i, a, X_j] = 0$ $/ * \forall X_i, X_j \in X \wedge \forall a \in D_i * /$
 5       $suppInv[b] = 0$  $/ * \forall b \in [1, maxD] * /$
 6       for every arc $R_{ij} \in R$ do
 7           for each $a \in D_i$ do
 8               $total = 0$
 9               for each $b \in D_j$ do
10                   if $(\langle X_i, a \rangle, \langle X_j, b \rangle) \in R_{ij}$ then
11                       $total = total + 1$
12                       $Append(S[X_j, b], \langle X_i, a \rangle)$
13                       $Counter[X_j, b, X_i] =$
                         $Counter[X_j, b, X_i] + 1$
14                       $suppInv[b] = suppInv[b] + 1$
15                       $Append(S[X_i, a], \langle X_j, b \rangle)$
16               if $total = 0$ then
17                   remove $a$ from $D_i$
18                   if $D_i = \phi$ then
                         return $initial$ = false
19                   else
                         if $S[X_i, a] \neq \{\}$ then
20                           $Q \leftarrow Q \cup \langle X_i, a \rangle$
21                   $M[X_i, a] = 0$
22               else
23                   $Counter[X_i, a, X_j] =$
                     $Counter[X_i, a, X_j] + total$
24           for each $b \in D_j$ do
25               if $suppInv[b] = 0$ then
26                   remove $b$ from $D_j$
27                   if $D_j = \phi$ then
28                       return $initial$ = false
29                       else
                             if $S[X_j, b] \neq \{\}$ then
30                               $Q \leftarrow Q \cup \langle X_j, b \rangle$
31                   $M[X_j, b] = 0$
32               else
                     $suppInv[b] = 0$
33       return $initial$ = true and $Q$, $M$, $S$, $Counter$
     end
```

3.1. Correctness of AC4-OP.

The algorithm AC4-OP is correct.

Proof. In order to obtain a contradiction, suppose that a value $a \in D_i$ is removed for X_i but it has a support with values of variables which X_i is restricted with. The value $a \in D_i$ could have been removed in the `InitializeAC4OP` phase of or in the procedure AC4-OP (Step 8). Investigate the following cases:

- If the value $a \in D_i$ is removed in the InizializeAC4OP phase, then it is removed in Step 17 or in Step 26 when each arc $R_{ij} \in R$ is analyzed.

  If it is removed in Step 17 then a direct constraint is being analyzed and total = 0 so that no value $b \in D_j$ is a support of $a \in D_i$. $\#contradiction$

  If it is removed in Step 26, then an inverse constraint is being analyzed, $suppInv[b] = 0$, so that b is not a support of any value of a variable. $\#contradiction$

- If the value $a \in D_i$ is removed in Step 8 of the procedure AC4-OP, then this is due to the fact that $Counter[X_i, a, X_j] = 0$, so that $\langle X_i = a \rangle$ has no supports for the variable X_j. $\#contradiction$

■

Thus, every value $a \in D_i$ removed for X_i by AC4-OP has no support with all values of variables which X_i is restricted with, and therefore this value will not take part in any solution.

4. Experimental results

In this section, we present some results to empirically demonstrate the practical efficiency of AC4-OP. Furthermore, we have implemented the most efficient version of AC4 with the improvement shown above in order to directly manage non-normalized CSPs.

Both algorithms AC4 and AC4-OP look for all the supports of each value, while other algorithms (AC3,

Table 5. Loops carried out by InitializeAC4-OP for the example in Fig. 1.

Loop 1. $R_{02} : X_0 = X_2$

D_0 a	D_2 b	Total	suppInv $[j]$	Counter $[X_i, a, X_j]$	Prune X_i	Counter $[X_j, b, X_i]$	Prune X_j	Add Q
	0	1	1			1		
0	1	1	0			0		
	2	1	0	1		0		
	0	0	1			1		
1	1	1	1			1		
	2	1	0	1		0		
	0	0	1			1		
2	1	0	1			1		
	2	1	1	1		1		
	0		0			1		
	1		0			1		
	2		0			1		

Loop 2. $R_{12} : X_1 > X_2$

D_0 a	D_2 b	Total	suppInv $[j]$	Counter $[X_i, a, X_j]$	Prune X_i	Counter $[X_j, b, X_i]$	Prune X_j	Add Q
	0	0	0			0		
0	1	0	0			0		
	2	0	0	0		0	$X_1 = 0$	
	0	1	1			1		
1	1	1	0			0		
	2	1	0	1		0		
	0	1	2			2		
2	1	2	1			1		
	2	2	0	2		0		
	0		0			2		
	1		0			1		
	2		0			0	$X_2 = 2$	$\langle X_2, 2\rangle$

Loop 3. $R_{12} : X_1 < X_2 + 2$

D_0 a	D_2 b	Total	suppInv $[j]$	Counter $[X_i, a, X_j]$	Prune X_i	Counter $[X_j, b, X_i]$	Prune X_j	Add Q
1	0	1	1			3		
	1	2	1	3		2	$X_1 = 0$	
	0	0	1			3		
2	1	1	2	3		3		
	0		0			3		
	1		0			3		

Table 6. Loops carried out by AC4-OP (only propagation) for the example in Fig. 1.

Loop	Tuple $\langle X_j, b\rangle$	Supports $S[X_j, b]$	Counter $[X_i, a, X_j]$	Prune X_i	Add Q $\langle X_i, a\rangle$
1	$\langle X_2, 2\rangle$	$\langle X_0, 2\rangle$	0	$X_0 = 2$	$\langle X_0, 2\rangle$
2	$\langle X_0, 2\rangle$	$\langle X_2, 2\rangle$	0		

AC2001/3.1, AC6, AC7, etc.) merely seek for just one support. Although it is not appropriate to compare the efficiencies of arc-consistency techniques which have different scopes, we compare the behavior of our proposed algorithm AC4-OP with that of AC4, and both AC6 (Rossi *et al.*, 2008) and AC7 (Bessiere *et al.*, 1999) algorithms in non-normalized and random instances and benchmark problems. We select the AC6 and AC7 algorithms because they are fine grained, they have the same phases (initialization and propagation) and therefore AC7 adds bidirectionality to AC6 (like AC4-OP does to AC4). All algorithms were written in C. The experiments were conducted on a PC Intel Core 2 Q9550 (2.83 GHz processor and 3 GB RAM).

4.1. Random instances. A random CSP instance is characterized by the quadruple $\langle n, d, m, b\rangle$, where n is the number of variables, d the domain size, m the number of binary constraints and b the maximum number of non-normalized constraints between two variables. The constraints are in the form $b \pm X_i \ op\ c \pm X_j$, where $X_i, X_j \in X$, $op \in \{<, \leq, \neq, >, \geq\}$ and $b, c \in \mathbb{N}$. The problems were randomly generated by modifying these parameters. Since we are working on non-normalized instances, we assume that any pair of variables is restricted by at least two constraints.

We generated two classes of random and non-normalized instances: consistent and inconsistent instances. In both types of instances, all the variables maintained the same size domain. We evaluated 50 test cases for each type of problem. Thus, we set two of the parameters and varied the other in order to assess the algorithm's performance when this parameter was increased. Performance was measured in terms of running time in milliseconds (time), the number of constraint checks (checks) and

Table 7. Results of consistency techniques AC6, AC7, AC4 and AC4-OP on random, consistent and non-normalized instances: $\langle n, 100, 700, 4\rangle$.

Instance n		AC6	AC7	AC4	AC4-OP
50	time [ms]	262	12566	3554	3234
	checks	8.39×10^5	1.40×10^5	1.27×10^7	6.37×10^6
	prunes	4.40×10^1	0	4.40×10^1	4.40×10^1
	supports	50	50	9.85×10^6	9.85×10^6
70	time [ms]	293	12660	3645	3757
	checks	8.38×10^5	1.40×10^5	1.27×10^7	6.37×10^6
	prunes	3.30×10^1	0	3.30×10^1	3.30×10^1
	supports	70	70	9.87×10^6	9.87×10^6
90	time [ms]	332	12765	4276	4284
	checks	8.37×10^5	1.40×10^5	1.27×10^7	6.37×10^6
	prunes	2.20×10^1	0	2.20×10^1	2.20×10^1
	supports	90	90	9.88×10^6	9.88×10^6
110	time [ms]	501	12352	3099	3009
	checks	8.37×10^5	1.40×10^5	1.27×10^7	6.37×10^6
	prunes	2.20×10^1	0	2.20×10^1	2.20×10^1
	supports	110	110	9.88×10^6	9.88×10^6
130	time [ms]	568	12725	3744	3503
	checks	8.37×10^5	1.40×10^5	1.27×10^7	6.37×10^6
	prunes	2.20×10^1	0	2.20×10^1	2.20×10^1
	supports	130	130	9.88×10^6	9.88×10^6
150	time [ms]	620	12930	4172	3764
	checks	8.37×10^5	1.40×10^5	1.27×10^7	6.37×10^6
	prunes	2.20×10^1	0	2.20×10^1	2.20×10^1
	supports	150	150	9.88×10^6	9.88×10^6

Algorithm 4: Procedure AC4-OP.

```
    Data: A CSP, $P = \langle X, D, R\rangle$
    Result: true and $P'$ (which is arc-consistent) or false and $P'$ (which
            is arc-inconsistent)
    begin
1       InitilizeAC4OP(P)
2       if $initial = true$ then
3           while $Q \neq \phi$ do
4               select and delete $\langle X_j, b\rangle$ from queue $Q$
5               for each $\langle X_i, a\rangle \in S[X_j, b]$ do
6                   $Counter[X_i, a, X_j] =$
                    $Counter[X_i, a, X_j] - 1$
7                   if
                    $Counter[X_i, a, X_j] = 0 \wedge M[X_i, a] = 1$
                    then
8                       remove $a$ from $D_i$
9                       if $D_i = \phi$ then
                            return false
10                      else
11                          if $S[X_i, a] \neq \{\}$ then
12                              $Q \leftarrow Q \cup \langle X_i, a\rangle$
13                      $M[X_i, a] = 0$
14          return true
15      else
            return false
    end
```

the number of prunes (prunes). The running time include both *inizialization time* and *propagation time*.

Table 7 shows the running time, the number of constraint checks and the amount of prunes and supports in consistent instances, where the number of variables was increased from 50 to 150 and the domain size was set at 100, the number of constraints was set at 700 and the maximum number of non-normalized constraints between two variables was set at 4: $\langle n, 100, 700, 4\rangle$. The average tightness of the instances was 27%. The results show that

- The number of constraint checks and running time were lower in AC4-OP than in AC4 in all cases with an average of 50% and 6%, respectively.
- The average of prune values was of 2.48×10^7 for AC6, AC4 and AC4-OP and 0 for AC7.
- The running time was lower in AC6 than in the rest of the algorithms (AC7, AC4 and AC4-OP) in all cases. This is due to the following:
  1. The low tightness of the instances make each prune value perform propagations (it is an advantage from AC6 over AC7, AC4 and AC4-OP by its light structures).
  2. AC4 and AC4-OP looked for all supports and AC6 looked for only one support.
- The number of constraint checks was lower in AC7 than in the rest of the algorithms (AC6, AC4 and AC4-OP in all cases), but AC7 performs no pruning.
- AC7 had a worse runtime than AC4, AC4-OP and AC6.

Thus, AC6 is a good choice for non-normalized, consistent and underconstrained problems.

Table 8 shows the running time, the number of constraint checks and the amount of prunes in inconsistent instances, where the number of variables was increased

Table 8. Results of consistency techniques AC6, AC7, AC4 and AC4-OP on random, inconsistent and non-normalized instances: $\langle n, 100, 700, 4\rangle$.

Instance n		AC6	AC7	AC4	AC4-OP
90	time [ms]	8890	1197	2263	1907
	checks	3.75×10^6	5.61×10^5	1.10×10^7	5.52×10^6
	prunes	6.84×10^3	3.92×10^3	7.26×10^3	7.26×10^3
110	time [ms]	11422	1812	1921	1549
	checks	3.91×10^6	7.14×10^5	1.16×10^7	5.81×10^6
	prunes	7.51×10^3	4.97×10^3	8.61×10^3	8.61×10^3
130	time [ms]	18990	2996	2164	1720
	checks	4.20×10^6	8.40×10^5	1.21×10^7	6.05×10^6
	prunes	8.39×10^3	5.96×10^3	9.22×10^3	9.22×10^3
150	time [ms]	15638	3858	2212	1756
	checks	4.20×10^6	9.80×10^5	1.24×10^7	6.21×10^6
	prunes	9.31×10^3	8.23×10^3	1.07×10^4	1.07×10^4
170	time [ms]	18766	7905	2350	1850
	checks	4.38×10^6	1.17×10^6	1.24×10^7	6.35×10^6
	prunes	1.04×10^4	1.06×10^4	1.09×10^4	1.09×10^4
190	time [ms]	20808	9639	2506	1905
	checks	4.410×10^6	1.30×10^6	1.28×10^7	6.43×10^6
	prunes	1.15×10^4	9.62×10^3	1.26×10^4	1.26×10^4

Table 9. Results of consistency techniques AC6, AC7, AC4 and AC4-OP on random, inconsistent and non-normalized instances: $\langle 100, 100, m, 4\rangle$.

Instance m		AC6	AC7	AC4	AC4-OP
200	time [ms]	6414	1365	607	500
	checks	1.59×10^6	6.17×10^5	3.9×10^6	1.95×10^6
	prunes	6.65×10^3	5.22×10^3	6.02×10^3	6.02×10^3
400	time [ms]	5808	416	1165	956
	checks	2.52×10^6	5.75×10^5	7.31×10^6	3.66×10^6
	prunes	5.92×10^3	3.11×10^3	6.82×10^3	6.82×10^3
600	time [ms]	6873	1930	1779	1328
	checks	3.38×10^6	6.51×10^5	1.01×10^7	5.07×10^6
	prunes	6.33×10^3	4.49×10^3	7.44×10^3	7.44×10^3
800	time [ms]	7197	2139	2225	1798
	checks	4.12×10^6	6.58×10^5	1.24×10^7	6.22×10^6
	prunes	7.08×10^3	463×10^3	8.05×10^3	8.05×10^3

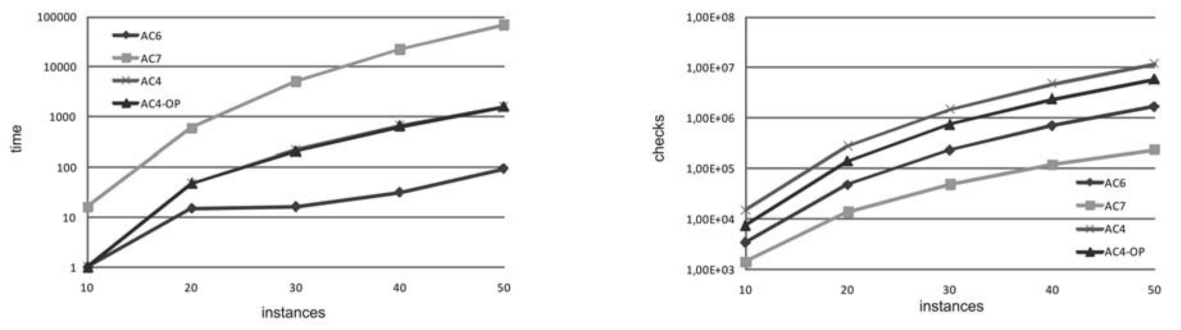
from 90 to 190, the domain size was set to 100, the number of constraints was set at 700 and the maximum number of non-normalized constraints between two variables was set at 4, i.e., $\langle n, 100, 700, 4\rangle$. The average tightness of the problems was 60%. The results show that AC4-OP had a lower running time than the rest of the algorithms (AC6, AC7 and AC4) in almost all cases, because AC4-OP employs bidirectionality and searches all supports. Thus AC4-OP is a good choice for this type of problem. Furthermore, both AC4-OP and AC4 performed more prunes than both AC6 and AC7. Again, AC7 performed fewer checks than AC6, AC4 and AC4-OP, but the behavior was worse as the number of variables increased. AC6 had poor results regarding these instances, i.e., with $n = 190$, AC6 spent 20808 milliseconds whereas AC7 spent 9639 milliseconds (the bidirectionally of AC7 was a good choice), and AC4-OP spent 1905 milliseconds whereas AC4 spent 2506 milliseconds (the bidirectionally of AC4-OP was the best choice). AC4-OP performed fewer checks and propagations than AC4.

Table 9 shows the running time, the number of constraint checks and the amount of prunes in inconsistent instances, where the number of constraints was increased from 200 to 800, the number of variables was set to 100, the domain size was set to 100 and the maximum number of non-normalized constraints between two variables was set at 4: $\langle 100, 100, m, 4\rangle$. The average tightness of the problems was 60%. As in the above table, the results were similar. They show that AC4-OP had a better behavior time than the rest of the algorithms (AC4-OP was faster than AC6 (21%), AC7 (80%) and AC4 (24%)). Moreover, AC4-OP performed fewer checks than AC4. In this case, the number of constraint checks increased as the number of constraints increased since the random instances maintained the same number of variables but the number of constraints increased. Thus, the random instances remained tightest.

Table 10 shows the running time, the number of constraint checks and the amount of prunes in inconsistent instances, where the domain size was increased from 50 to 300, the number of variables was set to 200, the number of constraints was set to 500 and the maximum num-

Table 10. Results of consistency techniques AC6, AC7, AC4 and AC4-OP on random, inconsistent and non-normalized instances: $\langle 200, d, 500, 4\rangle$.

Instance d		AC6	AC7	AC4	AC4-OP
50	time [ms]	1996	2131	426	350
	checks	8.17×10^5	3.33×10^5	2.28×10^6	1.14×10^6
	prunes	5.69×10^3	5.34×10^3	5.98×10^3	5.98×10^3
100	time [ms]	3357	8578	1683	1214
	checks	3.11×10^6	1.35×10^6	9.55×10^6	4.78×10^6
	prunes	1.10×10^4	1.15×10^4	1.20×10^4	1.20×10^4
150	time [ms]	48538	14713	3884	2789
	checks	8.05×10^6	2.91×10^6	2.18×10^7	1.09×10^7
	prunes	1.71×10^4	1.51×10^4	1.73×10^4	1.73×10^4
200	time [ms]	137219	38297	6145	5118
	checks	1.56×10^7	5.47×10^6	3.91×10^7	1.95×10^7
	prunes	2.16×10^4	2.21×10^4	1.99×10^4	1.99×10^4
300	time [ms]	369632	31779	14818	13142
	checks	3.78×10^7	1.05×10^7	8.87×10^7	4.43×10^7
	prunes	3.51×10^4	1.91×10^4	3.75×10^4	3.75×10^4



(a) (b)

Fig. 2. Behavior of AC6, AC7, AC4 and AC4-OP on different instances of the pigeon problem: runtime [ms] (a), number of constraint checks (b).

ber of non-normalized constraints between two variables was set at 4: $\langle 200, d, 500, 4\rangle$. In all instances the domain values were randomly generated and the average tightness of the instances was 60%. The results show that AC4-OP maintained a lower running time than the rest of the algorithms (AC4-OP was faster than AC4 (14%) , AC6 (95%) and AC7 (75%)). Improvements of both AC4 and AC4-OP are observed compared with AC6 and AC7 since AC6 and AC7 had to perform a domain ordering before the consistency process. An ascending order of domains is mandatory for both AC6 and AC7 but not for both AC4 or AC4-OP. The domain ordering was carried out by using the quicksort algorithm. It adds $O(nd \log d)$ complexity to AC6 and AC7, where n is the number of variables and d is the domain size of the problem.

4.2. Benchmarks: The pigeon problem.

The pigeon problem[3] is a well-known insolvable example. The problem is to put n pigeons into $n-1$ holes. However, each hole admits only a single pigeon. The problem can be formulated as a CSP with n variables corresponding to the n pigeons, and each variable has $n-1$ values corresponding to the holes. Each variable is constrained with the rest of variables in the problem. Thus, all constraints are binary and all variables have the same domain size. There are two types of pigeon problems: normalized and non-normalized, and we choose the latter. The original non-normalized instances of these benchmarks have two constraints between each pair of variables: $\forall i < j : X_i \leq X_j$ and $X_i \neq X_j$.

Table 11. Instances of the pigeon problem.

Instance	Variables	Domains	Constraints
10	10	1..9	90
20	20	1..19	380
30	30	1..29	870
40	40	1..39	1560
50	50	1..49	2450

[3]Some pigeon problem benchmarks are available at `http://www.cril.univ-artois.fr/CPAI08/`.

Figure 2 shows the behavior of AC6, AC7, AC4 and AC4-OP on different instances of the pigeon problem. The combinations of variables, domains and constraints are shown in Table 11. The results show that all instances are arc-consistent, although the problem is not solved. However, this issue is not detected by any arc-consistency algorithm. In this problem AC4-OP was faster than AC7, and AC4-OP, meanwhile, AC6 had better behavior (see Fig. 2 (a)). The runtime between AC4 and AC4-OP was similar because the tightness of the problems is low. Figure 2 (b) shows the number of constraint checks carried out by the algorithms. AC4-OP performs fewer constraints checks than AC4.

5. Conclusions

Constraint programming is a powerful technology for solving many real-life problems modeled as constraint satisfaction problems. Since solving a CSP is NP-complete, filtering techniques are widely used to prune the search space of CSPs. AC4 is one of the most well-known arc-consistency algorithms. In this paper, we present a reformulated version of AC4, called AC4-OP, for binary and non-normalized CSPs. This algorithm improves the efficiency of previous versions by reducing the number of constraint checks. This algorithm prunes the same search space as AC4, but its efficiency is provided by both the initialization strategy and the propagation strategy. AC4-OP checks the binary constraints in only one direction and it only propagates when it is necessary. Thus, it avoids unnecessary checking. In the evaluation section, it can be observed that AC4-OP is competitive in random instances. It had better behavior than AC6, AC7 and AC4 in non-consistent instances. In consistent instances, AC4-OP was competitive and stored all supports so that they could be used during the search process.

Acknowledgment

This work has been partially supported by the research projects TIN2010-20976-C02-01 (Ministry of Science and Innovation, Spain) and P19/08 (Ministry of Development, Spain, FEDER).

References

Arangú, M., Salido, M. A. and Barber, F. (2010). AC2001-OP: An arc-consistency algorithm for constraint satisfaction problems, *23rd International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2010, Córdoba, Spain*, pp. 219–228.

Barták, R. (1999). Constraint programming: In pursuit of the Holy Grail, *Proceedings of the Week of Doctoral Students (WDS99), Prague, Czech Republic*, Part IV, pp. 555–561.

Barták, R. (2001). Theory and practice of constraint propagation, *Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control, Gliwice, Poland*, pp. 7–14.

Barták, R. (2005). Constraint propagation and backtracking-based search, *First International Summer School on Constraint Programming, Acquafredda di Maratea, Italy*, pp. 1–43.

Barták, R., Salido, M.A. and Rossi, F. (2010). Constraint satisfaction techniques in planning and scheduling, *Journal of Intelligent Manufacturing* **21**: 5–15.

Bessiere, C. (1994). Arc-consistency and arc-consistency again, *Artificial Intelligence* **65**: 179–190.

Bessiere, C. (2006). Constraint propagation, *Technical report*, CNRS/University of Montpellier, Montpellier.

Bessiere, C. and Cordier, M. (1993). Arc-consistency and arc-consistency again, *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93), Washington, DC, USA*, pp. 108–113.

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

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

Brdyś, M.A. and Littler, J.J. (2002). Fuzzy logic gain scheduling for non-linear servo tracking, *International Journal of Applied Mathematics and Computer Science* **12**(2): 209–219.

Chmeiss, A. and Jegou, P. (1998). Efficient path-consistency propagation, *International Journal on Artificial Intelligence Tools* **7**: 121–142.

Dechter, R. (2003). *Constraint Processing*, Morgan Kaufmann, San Francisco, CA.

Deng, J., Becerra, V.M. and Stobart, R. (2009). Input constraints handling in an MPC/feedback linearization scheme, *International Journal of Applied Mathematics and Computer Science* **19**(2): 219–232, DOI: 10.2478/v10006-009-0018-2.

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

Królikowski, A. and Jerzy, D. (2001). Self-tuning generalized predictive control with input constraints, *International Journal of Applied Mathematics and Computer Science* **11**(2): 459–479.

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

Mehta, D. (2008). Reducing checks and revisions in the coarse-grained arc consistency algorithms, *Constraint Programming Letters* **2**: 37–53.

Mesghouni, K., Hammadi, S. and Borne, P. (2004). Evolutionary algorithms for job-shop scheduling, *International Journal of Applied Mathematics and Computer Science* **14**(1): 91–103.

Mohr, R. and Henderson, T. (1986). Arc and path consistency revised, *Artificial Intelligence* **28**: 225–233.

Perlin, M. (1992). Arc consistency for factorable relations, *Artificial Intelligence* **53**: 329–342.

Rossi, F., Van Beek, P. and Walsh, T. (2008). *Handbook of Constraint Programming*, Elsevier Science and Technology, Amsterdam.

Ruttkay, Z. (1998). Constraint satisfaction—A survey, *CWI Quarterly* **11**(2&3): 123–162.

Sikora, B. (2003). On the constrained controllability of dynamical systems with multiple delays in the state, *International Journal of Applied Mathematics and Computer Science* **13**(4): 469–479.

Tsang, E. (1995). *Foundations of Constraint Satisfaction*, Academic Press, London/San Diego, CA .

van Dongen, M., Dieker, A. and Sapozhnikov, A. (2008). The expected value and the variance of the checks required by revision algorithms, *Constraint Programming Letters* **2**: 55–77.

Marlene Arangú received a B.Sc. degree in computer science from Lisandro Alvarado Central Western University, Venezuela. She is an associate professor of computer science in the Department of Quantitative Techniques at the same university. She has authored/co-authored various journal articles and conference papers. She currently pursues her Ph.D. under the supervision of Prof. Miguel A. Salido. Her research interest is focused on filtering techniques for constraint satisfaction, planning and scheduling.

Miguel A. Salido received a B.Sc. degree in mathematics from the University of Valencia, Spain, in 1996, and a Ph.D. degree in computer science from the Technical University of Valencia, Spain, in 2002. He has been an associate/agregate professor of computer science in the Department of Information Systems and Computation at the Technical University of Valencia since 2005. He has participated in several national and international projects. He has authored/co-authored various journal articles, book chapters and conference papers. His research activity is focused on artificial intelligence, particularly, constraint satisfaction, planning and scheduling, and the application of these technique to real life problems.

Received: 13 September 2010
Revised: 22 March 2011