International Journal of Innovative
Computing, Information and Control
Volume **8**, Number **6**, June **2012**

ICIC International ©2012 ISSN 1349-4198
pp. **3891–3906**

A FILTERING TECHNIQUE TO ACHIEVE 2-CONSISTENCY IN CONSTRAINT SATISFACTION PROBLEMS

Marlene Arangú, Miguel A. Salido and Federico Barber

Instituto de Automática e Informática Industrial
Universidad Politécnica de Valencia
Camino de Vera s/n. 46022. Valencia, Spain
{ marangu; msalido; fbarber }@dsic.upv.es

Received February 2011; revised June 2011

Abstract. *Arc-Consistency algorithms are the most commonly used filtering techniques to prune the search space in Constraint Satisfaction Problems (CSPs). 2-consistency is a similar technique that guarantees that any instantiation of a value to a variable can be consistently extended to any second variable. Thus, 2-consistency can be stronger than arc-consistency in binary CSPs. In this work we present a new algorithm to achieve 2-consistency called 2-C4. This algorithm is a reformulation of AC4 algorithm that is able to reduce unnecessary checking and prune more search space than AC4. The experimental results show that 2-C4 was able to prune more search space than arc-consistency algorithms in non-normalized instances. Furthermore, 2-C4 was more efficient than other 2-consistency algorithms presented in the literature.*
Keywords: Constraint satisfaction problems, Consistency techniques, 2-consistency

1. **Introduction.** Over the last years, many real problems have been modeled and solved using several complete and heuristic techniques: genetic algorithms, fuzzy logic, simulated annealing, evolutionary algorithms, constraint programming, etc. [14,19,22-24]. Large, complex and combinatorial problems can be modeled as Constraint Satisfaction Problems (CSPs) and solved using constraint programming techniques. Much effort has been spent to increase the efficiency of the constraint satisfaction algorithms: filtering, learning and distributed techniques, improved backtracking, use of efficient representations and heuristics, etc. This effort resulted in the design of constraint reasoning tools which were used to solve numerous real problems. Constraint filtering is based on the idea of using the constraints actively to prune the search space. By integrating systematic search algorithms with consistency techniques, it is possible to get more efficient algorithms such as forward checking and look-ahead algorithms.

The consistency-enforcing algorithm performs any partial solution of a small subnetwork that is extensible to a surrounding network. The number of possible combinations can be huge, while only very few are 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 [29]. There exist several levels of consistency [6,14] depending on the number of involved variables: node-consistency involves only one variable; arc-consistency involves two variables; path-consistency involves three variables; and k-consistency involves k variables. However, arc-consistency is the most commonly used consistency technique.

Proposing efficient algorithms for enforcing arc-consistency has always been considered as a central question in the constraint reasoning community [11]. It is well-known that the arc-consistency process consumes a large portion of the search time required to solve

the CSP [30]. Therefore, there are many arc-consistency algorithms such as: AC1, AC2, and AC3 [25]; AC4 [26]; AC5 [18,27]; AC6 [8]; AC7 [9]; AC8 [13]; AC2001, AC3.1 [31].

The concept of consistency was generalized to k-consistency by [16]. Thus, 2-consistency is related to constraints that involve two variables. Furthermore, many works on arc-consistency make the simplified assumptions that CSPs are binary and normalized (two different constraints do not involve exactly the same variables), because these notations are simpler and new concepts are easier to present. However, there is a *strange effect* of associating arc-consistency with binary and normalized CSPs [28]. It is related to the confusion between the notions of arc-consistency and 2-consistency. On binary CSPs, 2-consistency is at least as strong as arc-consistency. Only in binary and normalized CSPs, both arc-consistency and 2-consistency perform the same pruning.

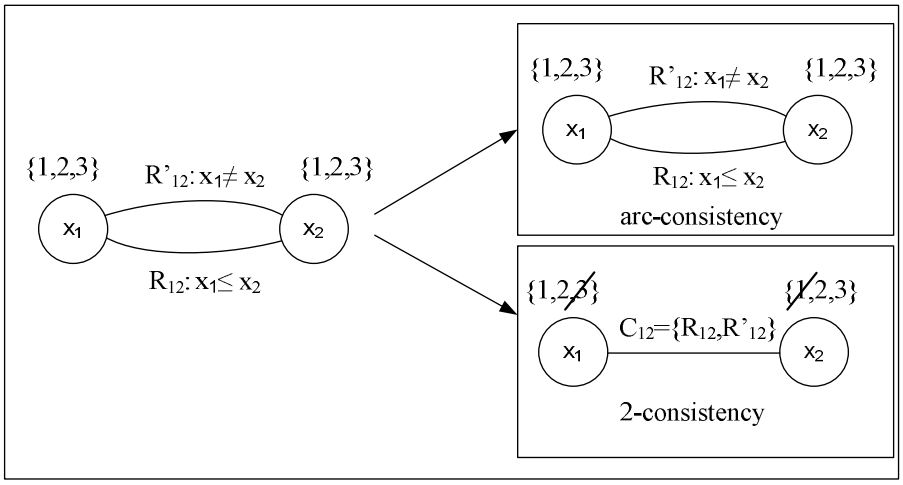

FIGURE 1. Example of binary and non-normalized CSP [28]

Figure 1 left shows a binary CSP with two variables X_1 and X_2, $D_1 = D_2 = \{1, 2, 3\}$ and two constraints $R_{12}(X_1 \leq X_2)$, $R'_{12}(X_1 \neq X_2)$. It can be observed that this CSP is arc-consistent due to the fact that every value of every variable has a support for constraints R_{12} and R'_{12}. In this case, arc-consistency does not prune any value of the domain of variables X_1 and X_2. However, (as authors say in [28]) this CSP is not 2-consistent because the instantiation $X_1 = 3$ cannot be extended to X_2 and the instantiation $X_2 = 1$ cannot be extended to X_1. Thus, Figure 1 right presents the resultant CSP filtered by arc-consistency and 2-consistency. It can be observed that 2-consistency is stronger than arc-consistency.

Our aim is focused on binary and non-normalized constraints. It is well-known that a non-normalized CSP can be transformed into a normalized one by using the intersection of valid tuples [3]. However, it is a time-consuming task mainly in problems with large domains [1]. Thus, our goal is to develop consistency techniques to reduce the search space in non-normalized CSPs in such a way the solution can be efficiently found.

In this work, we propose 2-C4, an algorithm, that reaches 2-consistency in binary and non-normalized CSPs. 2-C4 uses the structures of AC4, the constraints set of 2-C3 algorithm [1] and bidirectionally when it searches for supports. Thus, 2-C4 performs 2-consistency in an efficient way by saving the number of constraint checking and running time. It can obtain more prunes than other arc-consistency algorithms and it avoids the detected inefficiencies when the data structures of AC4 are updated. Thus, 2-C4 is more efficient than other 2-consistency algorithms mainly in inconsistent instances.

The rest of the work is organized as follows: Section 2 briefly summarizes the main definitions used to understand the rest of the paper and the main consistency algorithms as well. Section 3 explains the proposed 2-C4 algorithm. In Section 4, we evaluate empirically our approach in both random and benchmark instances. The evaluation was done in two different phases: (a) in a pre-process step, in which we show that 2-C4 outperforms other arc-consistency and 2-consistency algorithms and (b) in the search process, in which we show that 2-C4+Forward checking outperforms other consistency algorithms by using the same search technique. Finally, we present our conclusions.

2. CSP Preliminaries.

The basic idea of a CSP is to model the problem as a set of variables with finite domains 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 satisfying all the constraints. In general, this tasks is NP-Complete so filtering techniques are necessary to reduce the complexity in the search process.

2.1. Notations and definitions.

Following we present the standard notations and definitions presented in the literature.

Definition 2.1. *A Constraint Satisfaction Problem (CSP) is a triple* $P = \langle X, D, R\rangle$ *where:* X *is the finite set of variables* $\{X_1, X_2, \ldots, X_n\}$*;* D *is a set of domains* $D = D_1, D_2, \ldots, D_n$ *such that for each variable* $X_i \in X$ *there is a finite set of values that the variable can take;* R *is a finite set of constraints* $R = \{R_1, R_2, \ldots, R_m\}$ *which restrict the values that the variables can simultaneously take.*

Definition 2.2. *A constraint is binary if it only involves two variables. In this work the binary constraints are encoded by using intensional representation (by a function). We denote* $R_{ij} \equiv (R_{ij}, 1) \vee (R_{ij}, 3)$ *as the direct constraint defined over the variables* X_i *and* X_j *and* $R'_{ji} \equiv (R_{ij}, 2)$ *is the same constraint in the inverse direction over the variables* X_i *and* X_j *(inverse constraint).*

Definition 2.3. *An arithmetic constraint is a constraint in the form* $X_i \pm a$ *op* $X_j \pm b$*, where* $X_i, X_j \in X$*;* $a, b \in Z$ *and the operator op* $\in \{<, \leq, =, \neq, \geq, >\}$.

Definition 2.4. *A block of constraints* C_{ij} *is a set of binary constraints that involve the same variables* X_i *and* X_j*. Thus, we denote* $(C_{ij}, t) : t = \{1, 3\}$ *as the block of direct constraints defined over the variables* X_i *and* X_j *and* $(C'_{ji}, 2)$ *as the same block of constraints in the inverse direction over the variables* X_i *and* X_j *(block of inverse constraints).*

Definition 2.5. *An instantiation is a pair* $\langle X_i, a\rangle$ *that represents the assignment of the value* a *to the variable* X_i*, if* a *is in the domain of* X_i.

Definition 2.6. *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 2.7. *The number constraint checks of a constraint* R_{ij} *is the number of times a constraint* $R_{ij} \in R$ *is checked to reach arc-consistency.*

Definition 2.8. *Symmetry of the support: If a value* $b \in D_j$ *supports a value* $a \in D_i$*, then* a *supports* b *as well.*

Definition 2.9. *A CSP is normalized iff two different constraints do not involve exactly the same variables.*

Definition 2.10. *A CSP is binary iff all constraints* $R_{ij} \in R$ *are binary.*

Definition 2.11. *A* **value** $a \in D_i$ **is arc-consistent** *relative to* X_j*, iff there is 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 variables are arc-consistent, e.g., all the constraints* R_{ij} *and* R'_{ji} *are arc-consistent. (Note: here we are talking about full arc-consistency). A variable* X_i *is arc-inconsistent if its domain* D_i *does not contain any consistent value. If one (or more) variables is not arc-consistent then the CSP is called arc-inconsistent.*

Definition 2.12. *A* ***value*** $a \in D_i$ *is 2-consistent relative to* X_j*, iff there is a value* $b \in D_j$ *such that* $\langle X_i, a\rangle$ *and* $\langle X_j, b\rangle$ *satisfy all the constraints* R^k_{ij} *(*$\forall k : (\langle X_i = a\rangle, \langle X_j = b\rangle) \in R^k_{ij}$*). A* ***variable*** X_i *is 2-consistent relative to* X_j *iff all values in* D_i *are 2-consistent. A* ***CSP*** *is 2-consistent iff all variables are 2-consistent, e.g., any instantiation of a value to a variable can be consistently extended to a second variable.*

2.2. **Consistency algorithms.** Arc-consistency algorithms are a major component of many industrial and academic CSP solvers. Arc-consistency algorithms are based on the notion of a support proposed by [15]. These algorithms ensure that each value in the domain of each variable is supported by some value in the domain of a variable by which it is constrained.

We will conduct a brief description of some arc-consistency algorithms, presented in the literature:

- **AC1** [25] repeatedly revises all the domains and all constraints in order to remove unsupported values until no change occurs. Its inefficiency lies that in some prune causes a revision of all arcs (meanwhile only some of them will probably be affected).
- **AC2** [25] carries out the arc-consistency on a single loop through the nodes.
- **AC3** [25] repeatedly revises the domains in order to remove unsupported values. To avoid many useless calls to the *Revise* procedure, AC3 keeps all the constraints that do not guarantee arc-consistency in a queue. However, AC3 performs many ineffective checks.
- **AC4** [26] is the only algorithm that confirms the existence of a support by not identifying it throughout search. However, it stores all supports for each value in auxiliary data structures. It is an optimal algorithm. Its inefficiency lies in its spatial complexity and the necessity of maintaining huge data structures.
- **AC5** [18,27] allows a specialized arc-consistency for functional, anti-functional and monotonic classes of constraints. AC5 can be implemented on AC3 or AC4.
- **AC6** [8] maintains a data structure lighter than AC4. In fact, the idea in AC6 is not to count all supports that a value has on a constraint, but just to ensure that it has at least one.
- **AC6++** [11] improves AC6 by adding new structures and performing constraint check bidirectionally.
- **AC7** [9] improves on the idea of value support applied in AC4, AC6 and AC6++. The idea is that a value $a \in D_i$ supports a value $b \in D_j$ if and only if b also supports a. AC7 does not performs useless constraint checks.
- **AC8** [13] is based on supports but without recording any of them (it is like AC3 but propagations are made over values). AC8 records the references of the variables in a list and it maintains their status in an array.
- **AC2001/3.1** [10,31] follows the same framework than AC3, but it stores the smallest support for each value on each constraint.
- **AC3.2** and **AC3.3** [20] add partial and full bidirectionality, respectively. They are based on both AC2001/3.1 and AC7, respectively.

- **AC3r** and **AC3rm** [21] use the residues[1] of arc-consistency by verifying their validity before searching for a support. AC3rm exploits the multi-directionality (bidirectionality, for binary constraints).

Algorithms that perform arc-consistency have focused their improvements on time-complexity and space-complexity (see Table 1). The main improvements have been achieved by: changing the way of propagation: from arcs to values, (i.e., changing the granularity: from coarse-grained to fine-grained); appending new structures (storing more information that is useful); performing bidirectional searches (AC6++, AC7, AC3.3); changing the support search: searching for all supports (AC4) or searching for only the necessary supports (AC3, AC6, AC7, AC8 and AC2001); improving the constraint checking (AC7 and AC2001); storing the first support found (AC3, AC3.2, AC3.3), the smallest support found (AC20013.1) or the residual support found (AC3r and AC3rm), etc.

TABLE 1. Spatial and temporal complexity of arc-consistency algorithms

Algorithm	*Spatial Complexity*	*Temporal Complexity*
AC1	$O(n^3d^3)$	$O(ned^3)$
AC2	$O(e)$	$O(ed^3)$
AC3	$O(e)$	$O(ed^3)$
AC4	$O(ed^2)$	$O(ed^2)$
AC5*	$O(ed)$	$O(ed)$
AC6	$O(ed)$	$O(ed^2)$
AC6++	$O(ed)$	$O(ed^2)$
AC7	$O(ed^2)$	$O(ed^2)$
AC8	$O(n)$	$O(ed^3)$
AC2000	$O(ed)$	$O(ed^3)$
AC2001/3.1	$O(ed)$	$O(ed^2)$
AC3.2	$O(ed)$	$O(ed^2)$
AC3.3	$O(ed)$	$O(ed^2)$
AC3r	$O(ed)$	$O(ed^3)$
AC3rm	$O(ed)$	$O(ed^3)$
Key: e = edges; d = size of largest domain; n = variables; * for functional and monotonic constraints		

2.3. **Search techniques.** Many search algorithms have been developed to solve CSPs [7,14]. They are classified in look-back algorithms or look-ahead algorithms, according to the approach that they implement. In the first case, they check for inconsistencies of the current variable taking into account the variables previously instantiated. In the second case, they check for inconsistencies of the future variables involved besides the current and past variables. In this work we focus on three well-known search techniques: Backtracking [12], Forward Checking [17] and Real Full Look Ahead.

Backtracking (BT) [12] is a look-back algorithm. It is the most common algorithm to perform systematic search and it is the fundamental basis of the search algorithms. BT algorithm requires a static order among the variables and among their values. BT algorithm expands incrementally a partial assignment, which specifies consistent values with previously assigned variables, toward an entire assignment. This process chooses a first variable. Then it chooses a value of its domain until it finds a value that is

[1]A residue is a support, not necessary the smallest, that has been stored during a previous execution of the procedure which determines if a value is supported by a constraint.

consistent with respect the values of previously assigned variables. If no value is found, the assignment of the previous variable should be undone and a new value of its domain is chosen.

Forward Checking (FC) [17] is one of the more common look-ahead algorithms. At each stage of the search, FC verifies the current mapping forward against all future values of the variables which are restricted to the current one. The values of future variables that are inconsistent with the current assignment are temporarily removed from their domains. If the domain of a future variable becomes empty, the instantiation of the current variable is undone and a new value is chosen. If no value is consistent, then a backtrack is performed. FC ensures that at each stage the partial solution is consistent with each future value of each variable. Thus, by checking forward, a dead-end can be identified before continuing with a branch of the search, that finally will be pruned. Thus, it is possible to prune at an earlier stage.

3. The 2-C4 Algorithm.

By analyzing AC4, it can be observed the following items:

1. whenever Initialization phase of AC4 evaluates each constraint, it stores the information about the values of variables in M and the supports in S and $Counter$ (see below);
2. the direct constraint R_{ij} and inverse constraint R'_{ij} share the same set of variables (X_i and X_j).
3. the definition of support is bidirectional (see Definition 2.8),
4. whenever a value $a \in D_i$ is removed, a propagation phase must be carried out $\langle X_i, a\rangle$.
5. in non-normalized problems there are probably several constraints R_{ij} between the same pair of variables X_i and X_j.

Due to 1 and 2, an inefficiency in AC4 is detected because some values for M, S and $Counter$ might be updated for $\langle X_j, b\rangle$ once a direct constraint R_{ij} 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 for several times (e.g., once for each value $a \in D_i$). If there is a support, an upgrade of S is performed. However, AC4 only upgrades this structure to the variable X_i and the the variable X_j is ignored (item 3). As item 4 indicates, if there is no support for the tuple $\langle X_i, a\rangle$ because ($total = 0$), a propagation in queue Q is performed for this tuple. However, the propagation is only carried out if the pruned value a may affect the consistency of one or more previously assessed variables. This information is stored in S. If S is empty due to the fact that the pruned value $a \in D_i$ does not support any other value, then AC4 ignores it and it generates an inefficient propagation. Finally, as item 5 indicates, different constraints R_{ij} can generate different number of supports.

To take into account all the above items, we have developed 2-C4. It is a fine grained algorithm that achieves 2-consistency in binary and non-normalized CSPs (see Algorithm 2). This algorithm deals with block of constraints as 2-C3 [1] and 2-C3OP [2] but it only requires to keep half of the block of constraints in the queue Q (as 2-C3OP). Furthermore, each block of constraints $C_{ij} \in C$ is evaluated only once (in the same way that AC4 evaluates each individual constraint $R_{ij} \in R$).

Thus, the general improvement of 2-C4 is focused on the bidirectional check; it bidirectionally stores the support values for each block of constraints C_ij; and it performs inference to avoid unnecessary checks. However, inference is done by using a new array called $suppInv$ which is shared by all the constraints.

In order to perform one constraint checking in C and to identify the relevant values needed to be re-examined, 2-C4 uses the same structures than AC4, but it adds $suppInv$ to store the supports of each value of variable. Thus, once the values $a \in D_i$ are updated

$Counter[X_i, a, X_j]$, values $b \in D_j$ can be pruned if $suppInv[b] = 0$. Thus, the inverse constraint R'_{ji} is not needed to evaluated.

The data structures required by 2-C4 are the following:

- **S** is a matrix $S[X_j, b]$ that contains a list of pairs $\langle X_i, a\rangle$ such that $\langle X_j, b\rangle$ supports them. It must be taken into account that the same pair $\langle X_i, a\rangle$ could appear more than once in 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.
- **M** is a matrix $M[X_i, a]$ that stores 1 if value $a \in D_i$ or stores 0 if value $a \notin D_i$ (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.
- **suppInv** is a vector whose size is the maximum size of all domains ($maxD$). It stores a value greater or equal to 1 when the value of X_j is supported.

The main 2-C4 algorithm has two phases: initialization of the data structures and propagation. The initialization phase is used to remember pairs of consistent variable values (matrix S); to count "supporting" values from variable domain (matrix $Counter$) and to remove those values that have not any support or to remember those values (matrix M and queue Q).

The *Initialize2C4* procedure initializes the required structures Q, S, M, $Counter$ and $suppInv$. Then, four loops are performed to select and revise the block of constraints $R_{ij} \in C_{ij}$ for each value $a \in D_i$ and for each value $b \in D_j$.

Due to the fact that the same pair of variables X_i, X_j may be involved in more than one constraint R_{ij} (non-normalized CSPs), the support counters may have previous stored values from previous constraint checking. Pruning is carried out according to the counters in each constraint. The supports counter of variable X_i ($total$) is initialized to 0 for each value $a \in D_i$. However, the support counter 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 1, step 5). When the value $b \in D_j$ supports the value $a \in D_i$, $suppInv[b]$ is increased (see Algorithm 1, step 21). During the loop of steps 7-29, the array is updated in order to be analyzed in step 32. Once all values of D_i have been processed, if a value b of D_j has no support ($suppInv[b] = 0$) then this value is pruned from the domain D_j. If $suppInv[b] > 0$ then b is supported and it is initialized to 0 (see Algorithm 1, step 31 to 37) for further use of the array.

To achieve 2-consistency, 2-C4 only computes a support if the instantiations $\langle X_i, a\rangle$ and $\langle X_j, b\rangle$ hold with all R_{ij} constraints in C_{ij}. This is completed by initializing the variable *supported* to 1 (as a flag). 2-C4 stops the revision of the set C_{ij} if the instantiations $\langle X_i, a\rangle$ and $\langle X_j, b\rangle$ do not hold a R_{ij}; changing the *supported* variable to 0 (see Algorithm 1, steps 13 to 16).

Furthermore, 2-C4 only propagates those tuples that are supported by another tuple (see steps 26-27 and 35-36 of Algorithm 1, and steps 11-12 of Algorithm 2). Thus, 2-C4 avoids inefficient propagations of tuples for Q, and it avoids inefficient checking for those tuples. The process is stopped if a domain remain empty (Algorithm 1, steps 30 and 38 and Algorithm 2, step 9) or 2-C4 returns a 2-consistent CSP'.

3.1. **Soundness and complexity of 2-C4.**

1. The algorithm 2-C4 is sound.

   Proof: By contradiction, let us suppose that a value $c \in D_i$ is removed for X_i but it has a support with all values of variables which X_i are restricted with. The value

Algorithm 1: Procedure Initialize2C4

```
   Data: A CSP, P = ⟨X, D, C⟩ where C is the set the of blocks of constraints C_ij
   Result: initial=true and P', Q, S, M, Counter or  initial=false and P' (which is 2-inconsistent).
   begin
1      Q ← {}
2      S[X_j, b] ← {}  /* ∀X_j ∈ X ∧ ∀b ∈ D_j */
3      M[X_i, a] = 1  /* ∀X_i ∈ X ∧ ∀a ∈ D_i */
4      Counter[X_i, a, X_j] = 0  /* ∀X_i, X_j ∈ X ∧ ∀a ∈ D_i */
5      suppInv[b] = 0  /* ∀b ∈ [1, maxD] */
6      for every set C_ij ∈ C do
7          for each a ∈ D_i do
8              total = 0
9              for each b ∈ D_j do
10                 R_ij= first R_ij ∈ C_ij
11                 supported = 1
12                 while supported = 1 ∧ R_ij ≠ NULL do
13                     if (⟨X_i, a⟩, ⟨X_j, b⟩) ∈ R_ij then
14                         R_ij= next R_ij ∈ C_ij
15                     else
16                         supported = 0
17                 if supported = 1 then
18                     total = total + 1
19                     Append(S[X_j, b], ⟨X_i, a⟩)
20                     Counter[X_j, b, X_i] = Counter[X_j, b, X_i] + 1
21                     suppInv[b] = suppInv[b] + 1
22                     Append(S[X_i, a], ⟨X_j, b⟩)
23             if total = 0 then
24                 remove a from D_i
25                 M[X_i, a] = 0
26                 if S[X_i, a] ≠ {} then
27                     Q ← Q ∪ ⟨X_i, a⟩
28             else
29                 Counter[X_i, a, X_j] = total
30         if D_i = φ then
               return initial = false
31         for each b ∈ D_j do
32             if suppInv[b] = 0 then
33                 remove b from D_j
34                 M[X_j, b] = 0
35                 if S[X_j, b] ≠ {} then
36                     Q ← Q ∪ ⟨X_j, b⟩
37             else
                   suppInv[b] = 0
38         if D_j = φ then
               return initial = false
39     return initial = true and Q, M, S, Counter
   end
```

$c \in D_i$ could have been removed in the Inizialize2C4 phase or in step 8 of 2-C4. Let's study both cases:

- If the value $c \in D_i$ was removed in the Inizialize2C4 phase, then it is removed in step 24 or in step 33.
  If it is removed in step 24 then a direct constraint is being analyzed and total=0 so that no value in X_j is supported of $c \in D_i$. *#contradiction*

Algorithm 2: Procedure 2C4

```
Data: A CSP, P = ⟨X, D, C⟩ where C is the set the of blocks of constraints C_ij /*C_ij involves direct
      constraints R_ij only*/
Result: true and P' (which is 2-consistent) or  false and P' (which is 2-inconsistent)
begin
1   Initialize2C4(P)
2   if initial = true then
3     while Q ≠ φ do
4       select and delete ⟨X_j, b⟩ from queue Q
5       for each ⟨X_i, a⟩ ∈ 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 ∧ M[X_i, a] = 1 then
8           remove a from D_i
9           if D_i = φ then
              return false
10          else
11            if S[X_i, a] ≠ {} then
12              Q ← Q ∪ ⟨X_i, a⟩
13          M[X_i, a] = 0
14    return true
15  else
      return false
end
```

If it is removed in step 33 then an inverse constraint is being analyzed $suppInv[c] = 0$ so that b is not a support of any value of a variable. *#contradiction*

- If the value $c \in D_i$ is removed in step 8 of 2-C4, then this is due to the fact that $Counter[X_i, c, X_j] = 0$, so that $\langle X_i = c\rangle$ has no supports for the variable X_j. *#contradiction*

So, every value $c \in D_i$ removed for X_i by 2-C4 has not support with all values of variables which X_i are restricted with, so this value will not take part of any solution.

2. The complexity of 2-C4 is $O(fd^2)$, where f is the number of blocks of constraints, e is the number of binary constraints store in the blocks ($e > f$), and d the domain size in the problem.

   Proof: The Inizialize2C4 phase has a temporal cost of $O(fd^2)$. Step 6 analyzes each block ($O(f)$). For each block, every value $a \in D_i$ is analyzed (step 7). Therefore, for each value $a \in D_i$, every value $b \in D_j$ is again analyzed (step 9). Thus, the current cost is $O(fdd) = O(fd^2)$. Furthermore 2-C4 has a loop in Q which is a queue that stores rejected values waiting further processing. The cardinality of Q is $|Q| = nd$ where n is the number of variables. If we assume that $n < fd$ then the complexity of 2-C4 is $O(fd^2 + nd) < O(fd^2 + fd^2) \equiv O(fd^2)$.

4. **Experimental Results.** In this section, we evaluate the behavior of 2-C4 with well-known arc-consistency algorithms used in the CSP community: AC3 [25], AC2001/3.1 [10,31]; AC4 [26][2]; AC6 [8], AC7 [9] and 2-consistency algorithms: 2-C3OPL [4] and AC3NH [3].

The algorithms AC4-NN and 2-C4 look for all the supports of each value, while other algorithms (AC3, AC2001/3.1, AC6, AC7, AC3NH and 2-C3OPL.) merely seek for a single support. For this reason it is not appropriate to compare the efficiency of consistency

[2]We use the non-normalized version of AC4 named AC4-NN in [5]

techniques with different scopes. However, it is important to determine that the proposed algorithm achieves the same number of prunes in consistent instances with other 2-consistency algorithms.

Determining which algorithms are superior to others remains difficult. Algorithms often are compared by observing its performance on benchmark problems or on suites of random instances generated from a simple or uniform distribution.

On the one hand, the advantage of using benchmark problems is that if they are an interesting problem (to someone), then information about which algorithm works well on these problems is also interesting. However, although an algorithm outperforms to any other algorithm in its application to a concrete benchmark problem, it is difficult to extrapolate this feature to general problems. On the other hand, an advantage of using random problems is that there are many of them, and researchers can design carefully controlled experiments and report averages and other statistics. However, a drawback of random problems is that they may not reflect real life situations.

For consistency algorithms, in pre-process step (before search), the measures of efficiency are: the running time, the number of prunes and the number of constraint checks. For search algorithms the measure of efficiency was the running time. All algorithms were written in C. The experiments were conducted on a PC Intel Core 2 Quad (2.83 GHz processor and 3 GB RAM).

4.1. **Random instances.** The experiments performed on random and non-normalized instances were characterized by the 5-tuple $\langle n, d, m, c, p\rangle$, where n was the number of variables, d the domain size, m the number of binary constraints, c the maximum number of constraints in each block, and p the percentage of non-normalized constraints. All domains are ordered in ascendent form. 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 $b, c \in N$. The problems were randomly generated by modifying these parameters. Due to the fact that we are working on 100% of non-normalized instances, we assume that any pair of variables can be restricted to no less than 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, in all instances, we set four of the parameters and varied the other one in order to assess the algorithm's performance when this parameter is increased. Performance was measured in terms of running time (time), the number of constraint checks (checks), the number of support found and the number of prunes. The running time include both *inizialization time* and *propagation time* for the algorithms: AC6, AC7, AC4-NN y 2-C4.

Table 2 shows the number of constraint checks, running time, prunes and the number of found supports in random, consistent and non-normalized instances, where the number of constraints was increased from 100 to 800; the number of variables, the domain size, the number of constraints in each block and the percentage of non-normalized constraint were set at 50,20,4,100, respectively: $\langle 50, 20, m, 4, 100\rangle$. Considering algorithms that find all supports, the results show that the number of constraint checks and running time were lower in 2-C4 than AC4-NN. The number of prunes obtained by 2-C4 was greater than AC4-NN (about 85% more). This is due to the fact that AC4-NN analyzed each constraint individually meanwhile 2-C4 studied the block of constraints (see Example of Figure 1). Furthermore, the number of found supports was lower in 2-C4 than in AC4-NN because 2-C4 reached 2-consistency and it pruned all supports that did not reach 2-consistency. Considering the algorithms that only find one support, 2-C4 was more efficient because it carried out more pruning than other algorithms of arc-consistency (AC3, AC6 and AC7).

TABLE 2. Results of consistency techniques on random, consistent and non-normalized instances: $\langle 50, 20, m, 4, 100\rangle$

		arc-consistency				2-consistency		
m		**AC3**	**AC6**	**AC7**	**AC4-NN**	**AC3NH**	**2-C3OPL**	**2-C4**
100	checks	7.82×10^{09}	7.82×10^{03}	4.25×10^{03}	7.59×10^{04}	8.99×10^{10}	7.18×10^{09}	1.75×10^{04}
	time (ms)	0.1	0.1	47	8	31	0.1	0.1
	Prunes	3	3	0	3	36	36	36
	SuppFound	1	1	1	5.94×10^{10}	1	1	5.50×10^{09}
200	checks	1.59×10^{10}	1.56×10^{04}	8.05×10^{03}	1.52×10^{05}	1.82×10^{11}	1.43×10^{10}	3.50×10^{04}
	time (ms)	0.1	0.1	195	31	62	0.1	0.1
	Prunes	6	6	0	6	70	70	70
	SuppFound	1	1	1	1.18×10^{11}	1	1	1.09×10^{10}
300	checks	2.36×10^{10}	2.34×10^{04}	1.21×10^{04}	2.27×10^{05}	2.72×10^{11}	2.08×10^{10}	5.19×10^{04}
	time (ms)	0.1	0.1	439	56	100	0.1	8
	Prunes	6	6	0	6	70	70	70
	SuppFound	1	1	1	1.78×10^{11}	1	1	1.64×10^{10}
400	checks	3.19×10^{10}	3.12×10^{04}	1.61×10^{04}	3.03×10^{05}	3.65×10^{11}	2.77×10^{10}	6.91×10^{04}
	time (ms)	0.1	0.1	776	86	145	0.1	16
	Prunes	9	9	0	9	80	80	80
	SuppFound	1	1	1	2.37×10^{11}	1	1	2.18×10^{10}
500	checks	3.96×10^{10}	3.90×10^{04}	2.01×10^{04}	3.79×10^{05}	4.55×10^{11}	3.42×10^{10}	8.59×10^{04}
	time (ms)	0.1	15	1207	108	184	16	16
	Prunes	9	9	0	9	80	80	80
	SuppFound	1	1	1	2.96×10^{11}	1	1	2.72×10^{10}
600	checks	4.81×10^{10}	4.68×10^{04}	2.42×10^{04}	4.55×10^{05}	5.51×10^{11}	4.11×10^{10}	1.03×10^{05}
	time (ms)	0.1	16	1722	122	220	16	23
	Prunes	12	12	0	12	90	90	90
	SuppFound	1	1	1	3.54×10^{11}	1	1	3.23×10^{10}
700	checks	5.59×10^{10}	5.45×10^{04}	2.82×10^{04}	5.30×10^{05}	6.41×10^{11}	4.76×10^{10}	1.20×10^{05}
	time (ms)	15	18	2328	120	245	15	16
	Prunes	12	12	0	12	90	90	90
	SuppFound	1	1	1	4.14×10^{11}	1	1	3.79×10^{10}
800	checks	6.46×10^{10}	6.24×10^{04}	3.22×10^{04}	6.06×10^{05}	7.39×10^{11}	5.44×10^{10}	1.37×10^{05}
	time (ms)	15	20	3033	140	282	16	18
	Prunes	15	15	0	15	100	100	100
	SuppFound	1	1	1	4.72×10^{11}	1	1	4.29×10^{10}

Note that AC7 performed fewer constraint checks than the other algorithms, its running time was higher but its pruning was lower. This is due to the fact that its *Last* matrix is wrong in non-normalized instances. Also, when 2-C4 was compared with the other 2-consistency algorithms AC3NH and 2-C3OPL, it kept a good performance because 2-C4 found much more supports than both AC3NH and 2-C3OPL in an efficient way.

Table 3 shows the number of constraint checks, running time, prunes and number of found supports in random, consistent and non-normalized instances, where the number of variables was increased from 50 to 190 and the number of constraints, the domain size, the number of constraints in each block and the percentage of non-normalized constraint were set at 20, 700, 4 and 100, respectively: $\langle n, 20, 700, 4, 100\rangle$. Again, those algorithms that reach 2-consistency performed more pruning that arc-consistency algorithms. The 2-C4 made fewer checks than 2-C3OPL but 2-C3OPL was faster than 2-C4. It is due to that 2-C4 founded all support and 2-C3OPL only found one support, and these instances had few propagations. Also AC6 spent less time, but made less pruning than 2-C4.

Table 4 shows the number of constraint checks and running time in inconsistent instances, where the number of constraints was increased from 100 to 800; the number of variables; the domain size, the number of non-normalized constraint and the percentage of non-normalized constraints were set at 100, 100, 4 and 100 respectively: $\langle 100, 100, m, 4, 100\rangle$. The tightness of the problems is 60% in average. The results show that 2-C4 was able to detect inconsistency in a more efficient way that the 2-consistency

TABLE 3. Results of consistency techniques on random, consistent and non-normalized instances: $\langle n, 20, 700, 4, 100\rangle$

		arc-consistency				2-consistency		
n		**AC3**	**AC6**	**AC7**	**AC4-NN**	**AC3NH**	**2-C3OPL**	**2-C4**
50	checks	5.59×10^{10}	5.45×10^{4}	2.82×10^{4}	5.30×10^{5}	6.41×10^{11}	4.76×10^{10}	1.20×10^{5}
	time (ms)	15	0.1	2437	151	249	12	16
	Prunes	12	12	0	12	90	90	90
	SuppFound	1	1	1	4.14×10^{11}	1	1	3.78×10^{10}
70	checks	5.51×10^{10}	5.45×10^{4}	2.82×10^{4}	5.30×10^{5}	6.35×10^{11}	4.74×10^{10}	1.20×10^{5}
	time (ms)	0.1	0.1	2465	161	247	0.1	16
	Prunes	9	9	0	9	100	100	100
	SuppFound	1	1	1	4.15×10^{11}	1	1	3.82×10^{10}
90	checks	5.46×10^{10}	5.43×10^{4}	2.82×10^{4}	5.30×10^{5}	6.31×10^{11}	4.73×10^{10}	1.20×10^{5}
	time (ms)	0.1	0.1	2480	172	249	0.1	16
	Prunes	6	6	0	6	110	110	110
	SuppFound	1	1	1	4.15×10^{11}	1	1	3.84×10^{10}
110	checks	5.46×10^{10}	5.44×10^{4}	2.82×10^{4}	5.30×10^{5}	6.31×10^{11}	4.75×10^{10}	1.20×10^{5}
	time (ms)	0.1	0.1	2428	127	241	14	30
	Prunes	6	6	0	6	130	130	130
	SuppFound	1	1	1	4.15×10^{11}	1	1	3.84×10^{10}
130	checks	5.46×10^{10}	5.44×10^{4}	2.82×10^{4}	5.30×10^{5}	6.31×10^{11}	4.77×10^{10}	1.20×10^{5}
	time (ms)	0.1	0.1	2500	139	248	16	31
	Prunes	6	6	0	6	150	150	150
	SuppFound	1	1	1	4.15×10^{11}	1	1	3.84×10^{10}
150	checks	5.46×10^{10}	5.44×10^{4}	2.82×10^{4}	5.30×10^{5}	6.31×10^{11}	4.79×10^{10}	1.20×10^{5}
	time (ms)	0.1	3	2547	158	247	16	31
	Prunes	6	6	0	6	170	170	170
	SuppFound	1	1	1	4.15×10^{11}	1	1	3.84×10^{10}
170	checks	5.46×10^{10}	5.44×10^{4}	2.82×10^{4}	5.30×10^{5}	6.31×10^{11}	4.81×10^{10}	1.21×10^{5}
	time (ms)	0.1	8	2603	195	248	24	32
	Prunes	6	6	0	6	190	190	190
	SuppFound	1	1	1	4.15×10^{11}	1	1	3.84×10^{10}
190	checks	5.43×10^{10}	5.43×10^{4}	2.82×10^{4}	5.30×10^{5}	6.29×10^{11}	4.78×10^{10}	1.20×10^{5}
	time (ms)	0.1	15	2642	244	250	31	47
	Prunes	3	3	0	3	186	186	186
	SuppFound	1	1	1	4.16×10^{11}	1	1	3.85×10^{10}

TABLE 4. Results of consistency techniques on random, inconsistent and non-normalized instances: $\langle 50, 20, m, 4, 100\rangle$

		arc-consistency				2-consistency	
m		**AC3**	**AC6**	**AC7**	**AC4-NN**	**2-C3OPL**	**2-C4**
100	checks	4.84×10^{10}	3.00×10^{4}	1.45×10^{4}	7.06×10^{5}	2.65×10^{10}	3.54×10^{4}
	time (ms)	0.1	26	27	0.1	0.1	4
200	checks	4.98×10^{10}	4.17×10^{4}	1.41×10^{4}	1.05×10^{5}	3.20×10^{10}	5.26×10^{4}
	time (ms)	0.1	22	27	10	0.1	16
300	checks	4.49×10^{10}	4.46×10^{4}	1.42×10^{4}	1.15×10^{5}	3.41×10^{10}	5.75×10^{4}
	time (ms)	0.1	4	36	16	16	16
400	checks	4.35×10^{10}	4.35×10^{4}	1.41×10^{4}	1.11×10^{5}	3.34×10^{10}	5.57×10^{4}
	time (ms)	12	0.1	41	16	16	16
500	checks	4.29×10^{10}	4.29×10^{4}	1.39×10^{4}	1.10×10^{5}	3.30×10^{10}	5.50×10^{4}
	time (ms)	16	0.1	48	16	16	16
600	checks	4.29×10^{10}	4.29×10^{4}	1.40×10^{4}	1.10×10^{5}	3.29×10^{10}	5.51×10^{4}
	time (ms)	16	16	52	16	30	18
700	checks	4.48×10^{10}	4.48×10^{4}	1.40×10^{4}	1.15×10^{5}	3.45×10^{10}	5.77×10^{4}
	time (ms)	28	15	62	16	31	15
800	checks	4.30×10^{10}	4.30×10^{4}	1.38×10^{4}	1.10×10^{5}	3.31×10^{10}	5.53×10^{4}
	time (ms)	16	0.1	48	0.3	30	16

algorithm 2-C3OPL, mainly in large instances. The number of constraint checks was better in 2-C4 that AC3 and AC4-NN, but it maintained a similar behavior with the rest of algorithms.

4.2. **Benchmark problems: The pigeon problem.** The pigeon problem[3] is a well-known insoluble problem. The problem is to put n pigeons into $n-1$ holes. However, every hole admits only a single pigeon. The problem can be formulated as a CSP with n variables corresponding to the n pigeons, and every variable has $n-1$ values corresponding to the holes. Each variable is constrained with the rest of variables of the problem. Thus, all constraints are binary and all variables have the same domain size. There are two types of pigeon problems: normalized and no-normalized, so we choose the latter one. The original no-normalized instances of these benchmarks have two constraints between each pair of variables: $\forall\ i < j : X_i \leq X_j$ and $X_i \neq X_j$. (See Table 5)

TABLE 5. Data of Pigeon problems benchmarks

Instance	**Variables**	**Domains**	**Constraints**	**Predicates**	**% of No-normalized**
10	10	0..8	90	$P1 : X \leq Y$ $P1 : X \neq Y$	100%
20	20	0..18	380		
30	30	0..28	870		
40	40	0..38	1560		
50	50	0..48	2450		

TABLE 6. Constraint checks, number of prunes and time in benchmarks instances of the Pigeon problem

inst.		***AC3***	***AC2001/3.1***	***AC6***	***AC7***	***AC4-NN***	***AC3NH***	***2-C3OPL***	***2-C4***
10	Checks	3.33×10^3	9.00×10^2	3.33×10^3	1.39×10^3	1.45×10^4	1.85×10^4	2.06×10^3	4.20×10^1
	Time(ms)	0.1	0.1	0.1	31	0.1	0.1	0.1	0.1
	Prunes	0	0	0	0	0	53	53	53
20	Checks	4.73×10^4	7.60×10^3	4.73×10^4	1.34×10^4	2.74×10^5	5.77×10^5	3.62×10^4	1.87×10^2
	Time(ms)	0.1	0.1	15	797	46	109	15	15
	Prunes	0	0	0	0	0	208	208	208
30	Checks	2.27×10^5	2.61×10^4	2.27×10^5	4.82×10^4	1.46×10^6	4.29×10^6	1.89×10^5	4.32×10^2
	Time(ms)	15	0.1	16	6861	203	1281	47	78
	Prunes	0	0	0	0	0	463	463	463
40	Checks	7.01×10^5	6.24×10^4	7.01×10^5	1.17×10^5	4.74×10^6	1.78×10^7	6.09×10^5	7.77×10^2
	Time(ms)	16	15	62	34731	656	7640	172	219
	Prunes	0	0	0	0	0	818	818	818
50	Checks	1.68×10^6	1.22×10^5	1.68×10^6	2.33×10^5	1.17×10^7	5.41×10^7	1.50×10^6	1.22×10^3
	Time(ms)	31	16	140	123740	1609	30563	578	547
	Prunes	0	0	0	0	0	1273	1273	1273

Table 6 shows the results of arc-consistency and 2-consistency techniques on different instances of the Pigeon problem. The combinations of variables, domains and constraints used in this evaluation are presented in Table 5. Thus, arc-consistency was achieved out by using AC3, AC2001/3.1, AC6, AC7 and AC4-NN, meanwhile 2-consistency was achieved by using AC3NH, 2-C3OPL and 2-C4. It must be taken into account that all these instances have no solution, but arc-consistency techniques did not detected the inconsistency. However, 2-consistency algorithms detect the inconsistency in all by performing prunes. None of the arc-consistency algorithms carried out prunes and only some of them performed propagations (AC7 and AC4-NN). These last algorithms were more

[3]Pigeon Problem Benchmarks are available in http://www.cril.univ-artois.fr/CPAI08/

inefficient than 2-C4. In all these instances 2-C4 performed fewer checks than the other 2-consistency algorithms, although 2-C3OPL was faster than 2-C4 in 50% of the instances.

TABLE 7. Instances for the pigeon problems extended

Instance	Variables	Domains	Constraints	Predicates	% of no-normalized
30	30	0..40	870		
40	40	0..45	1560		
50	50	0..55	2450		
60	60	0..65	3540	$P1 : X \leq Y$	100%
70	70	0..75	4830	$P1 : X \neq Y$	
80	80	0..85	6320		
90	90	0..95	8010		
100	100	0..120	9900		

We extended the number of holes (the domain size) to these benchmarks in order to generate consistent instances with at least one solution (see Table 7). On these new instances, we carried out a consistency technique and a search technique (Forward-Checking) [17] for finding a solution to the resultant CSPs. The results are showed in Figure 2. The algorithms AC4-NN and AC3NH could only handle up to 80 instance. Again, all arc-consistency algorithms could not made any pruning meanwhile 2-consistency algorithms could prune more values (over 3 orders of magnitude, in average). Thus 2-C4 was more efficient than the other arc-consistency and 2-consistency techniques mainly in waslarger instances. On average, 2-C4 was 30% faster than 2-C3OPL. Summarizing, 2-C4 is an efficient technique for solving large instances. The fact that 2-C4 founds all support was useful during the search process.

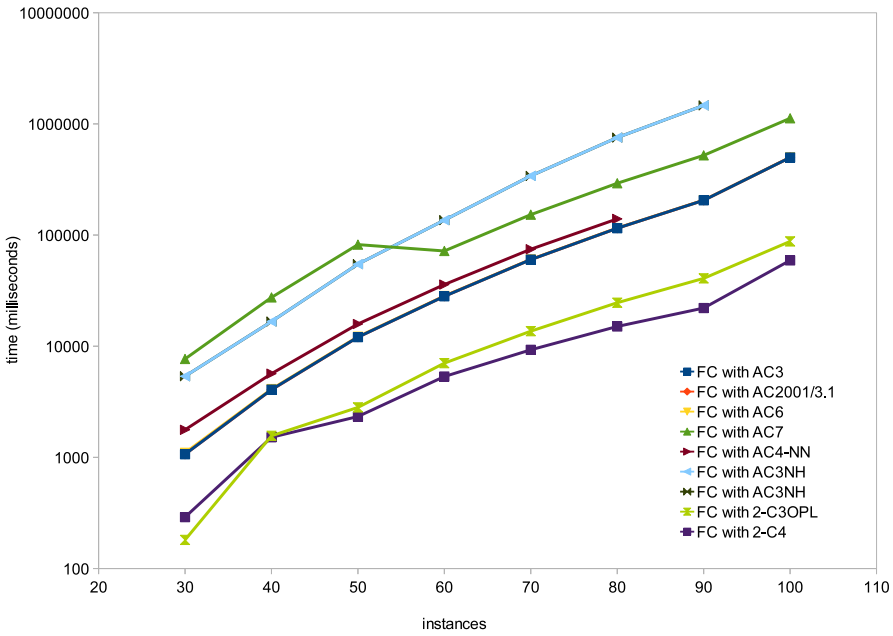

FIGURE 2. Forward checking plus consistency techniques for finding a solution in different instances of the pigeon problem showed in Table 7

5. **Conclusions.** In this paper, we propose an filtering technique to reduce the solution space in a constraint satisfaction problem. To this end, we have presented a 2-consistency algorithm called 2-C4. It deals with binary and non-normalized constraints. 2-C4 is an optimized and reformulated version of AC4 that improves the efficiency of previous versions by reducing the number of propagations, the number of constraint checks and the running time. Furthermore, the number of supports generated by 2-C4 was smaller than AC4 due to the fact that it improves the pruning process. In the evaluation section, 2-C4 and other consistency techniques were evaluated and compared in both random and benchmark instances. The results show that 2-C4 was more efficient in both consistent and inconsistent instances, despite it carried out a full search process for finding all supports.

Acknowledgment. This work has been partially supported by the research project TIN2010- 20976-C02-01 (Min. de Educacion y Ciencia, Spain-FEDER).

REFERENCES

[1] M. Arangu, M. Salido and F. Barber, 2-c3: From arc-consistency to 2-consistency, *SARA*, 2009.
[2] M. Arangu, M. Salido and F. Barber, 2-c3op: An improved version of 2-consistency, *ICTAI*, pp.344-348, 2009.
[3] M. Arangu, M. Salido and F. Barber, Normalizando csp no-normalizados: Un enfoque híbrido, *CAEPIA Proc. of Workshop on Planning, Scheduling and Constraint Satisfaction*, 2009.
[4] M. Arangu, M. Salido and F. Barber, A filtering technique for the railway scheduling problem, *COPLAS: ICAPS 2010 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, pp.68-78, 2010.
[5] M. Arangu, M. Salido and F. Barber, Filtering techniques for non-normalized constraint satisfaction problems, *Technical Report*, DSIC-II/11/10.UPV, 2010.
[6] R. Bartak, Theory and practice of constraint propagation, *Proc. of the 3rd Workshop on Constraint Programming in Decision and Control*, 2001.
[7] R. Bartak, Constraint propagation and backtracking-based search, *First International Summer School on CP*, 2005.
[8] C. Bessiere and M. Cordier, Arc-consistency and arc-consistency again, *Proc. of the AAAI*, Washington, USA, pp.108-113, 1993.
[9] C. Bessiere, E. C. Freuder and J. C. Régin, Using constraint metaknowledge to reduce arc consistency computation, *Artificial Intelligence*, vol.107, pp.125-148, 1999.
[10] C. Bessiere, J. C. Régin, R. Yap and Y. Zhang, An optimal coarse-grained arcconsistency algorithm, *Artificial Intelligence*, vol.165, pp.165-185, 2005.
[11] C. Bessiere and J.-C. Regin, An arc-consistency algorithm optimal in the number of constraint checks, *ICTAI*, pp.397-403, 1994.
[12] J. R. Bitner and E. M. Reingold, Backtrack programming techniques, *Commun. ACM*, vol.18, no.11, pp.651-656, 1975.
[13] A. Chmeiss and P. Jegou, Efficient path-consistency propagation, *International Journal on Artificial Intelligence Tools*, vol.7, pp.121-142, 1998.
[14] R. Dechter, *Constraint Processing*, Morgan Kaufmann, 2003.
[15] R. Fikes, Ref-arf: A system for solving problems stated as procedures, *Artificial Intelligence*, vol.1, 1970.
[16] E. Freuder, Synthesizing constraint expressions, *Communications of the ACM*, vol.21, pp.958-966, 1978.
[17] R. Haralick and G. Elliot, Increasing tree efficiency for constraint satisfaction problems, *Artificial Intelligence*, vol.14, pp.263-314, 1980.
[18] P. Van Hentenryck, Y. Deville and C. M. Teng, A generic arc-consistency algorithm and its specializations, *Artificial Intelligence*, vol.57, pp.291-321, 1992.
[19] G. Kim, S. S. Kim, I.-H. Kim, D. H. Kim, V. Mani and J.-K. Moon, An efficient simulated annealing with a valid solution mechanism for tdma brodcast scheduling problem, *International Journal of Innovative Computing, Information and Control*, vol.7, no.3, pp.1181-1191, 2011.
[20] C. Lecoutre, F. Boussemart and F. Hemery, Exploiting multidirectionality in coarse-grained arc consistency algorithms, *Proc. CP 2003*, pp.480-494, 2003.

[21] C. Lecoutre and F. Hemery, A study of residual supports in arc consistency, *Proc. IJCAI 2007*, pp.125-130, 2007.
[22] F.-T. Lin and T.-R. Tsai, A two-stage genetic algorithm for solving the transportation problem with fuzzy demands and fuzzy supplies, *International Journal of Innovative Computing Information and Control*, vol.5, no.12(B), pp.4775-4785, 2009.
[23] S.-F. Lin and Y.-C. Cheng, Two-strategy reinforcement evolutionary algorithm using datamining based crossover strategy with tsk-type fuzzy controllers, *International Journal of Innovative Computing, Information and Control*, vol.6, no.9, pp.3863-4198, 2010.
[24] C. Liu, New evolutionary algorithm for multi-objetive constrained optimization, *ICIC Express Letters*, vol.2, no.4, pp.339-344, 2008.
[25] A. K. Mackworth, Consistency in networks of relations, *Artificial Intelligence*, vol.8, pp.99-118, 1977.
[26] R. Mohr and T. C. Henderson, Arc and path consistency revised, *Artificial Intelligence*, vol.28, pp.225-233, 1986.
[27] M. Perlin, Arc consistency for factorable relations, *Artificial Intelligence*, vol.53, pp.329-342, 1992.
[28] F. Rossi, P. Van Beek and T. Walsh, *Handbook of Constraint Programming*, Elsevier Science & Technology, 2008.
[29] Z. Ruttkay, Constraint satisfaction – A survey, *CWI Quarterly*, vol.11, no.2-3, pp.123-162, 1998.
[30] M. R. C. Van Dongen, A. B. Dieker and A. Sapozhnikov, The expected value and the variance of the checks required by revision algorithms, *CP Letters*, vol.2, pp.55-77, 2008.
[31] Y. Zhang and R. H. C. Yap, Making AC3 an optimal algorithm, *IJCAI-2001*, 2001.