Domain Consistency with Forbidden Values

Yves Deville[1], Pascal Van Hentenryck[2]

[1] Université catholique de Louvain (yves.deville@uclouvain.be)
[2] Brown University (pvh@cs.brown.edu)

Abstract. This paper presents a novel domain-consistency algorithm which does not maintain supports dynamically during propagation, but rather maintain forbidden values. It introduces the optimal NAC4 (negative AC4) algorithm based on this idea. It further shows that maintaining forbidden values dynamically allows the generic algorithm AC5 to achieve domain consistency in time $O(ed)$ for classes of constraints in which the number of supports is $O(d^2)$ but the number of forbidden values is $O(d)$. The paper also shows how forbidden values and supports can be used jointly to achieve domain consistency on logical combinations of constraints and to compute validity as well as entailment of constraints. Experimental results show the benefits of the joint exploitation of supports and forbidden values.

1 Introduction

In constraint programming, propagation aims at reducing the search space without removing solutions. The propagation algorithm considers each constraint individually and terminates when no constraint can be used to reduce the domains of the variables. The ideal propagation for a constraint is *domain consistency*, also known as arc consistency: It removes from the domain of each variable all values that do not belong to a solution of the considered constraint. Many algorithms have been proposed for achieving domain consistency, such as AC3, AC4, AC6, AC7 and AC2001 (see [1]). Consistency algorithms typically use the concept of *support*. For a binary constraint x over variables x and y, a support for a pair (x, a), where a is a possible value for x, is a pair (y, b) such that $C(x/a, y/b)$ holds. The optimal time complexity to achieve domain consistency for a CSP is $O(e.d^2)$ for binary constraints and $O(e.r.d^r)$ for non-binary constraints (d is the size of the largest domain, e the number of constraints and r the largest arity of the constraints). An algorithm such as AC4 maintains all the supports for all pairs (x, a), while other algorithms (e.g., AC6) only maintain a single support and search for subsequent supports on demand. AC4 works in two steps. First, it computes all the supports for all the variable/value pairs in each constraint. Then, it propagates the removal of a value a from the domain of a variable x. An interesting property of the propagation step is that its time complexity is proportional to the total number of supports.

Example 1. Consider the constraint $x = y \bmod 10$, with $D(x) = \{0..9\}$ and $D(y) = \{0..99\}$, the size of the supports for x is linear ($O(\#D(y))$, where $\#A$ is the size of A). The propagation step of AC4 for this constraint is also linear, while it remains quadratic for other optimal AC algorithms such as AC6, AC7, or AC2001.

Of course, the initialization step of AC4, which computes all the supports, is $O(d^2)$ even if the number of supports is $O(d)$, since the algorithm has no knowledge of the semantics of the constraint. The generic AC5 algorithm [2] was designed to exploit the semantics of the constraints, and can then be used to generate the supports of such constraints in linear time, resulting in an $O(ed)$ complexity.

The scientific question addressed in this paper is the following: *Is it possible to design a domain-consistency algorithm running in time $O(ed)$ if the number of supports is quadratic, but the number of forbidden values (also called conflict set) is linear?*

Example 2. Consider the constraint $x \neq y \bmod 10$, with $D(x) = \{0..9\}$ and $D(y) = \{0..99\}$. The size of the supports for x is 900, hence a complexity of $O(\#D(x).\#D(y))$ in the propagation step of AC4. Using AC3, AC7 or AC2001 does not help to reduce this complexity. However, the size of the forbidden values is 100 ($O(\#D(y))$). Can we use an AC4-like algorithm that maintains the list of forbidden values instead of the supports to obtain an $O(ed)$ algorithm?

This paper answers this question positively and makes the following contributions:

- It proposes the NAC4 algorithm (Negative AC4) that achieves the optimal $O(e.d^2)$ time complexity for binary CSPs, but dynamically maintains the set of forbidden values instead of supports. It shows that both AC4 and NAC4 are instances of the generic AC5 algorithm: they can be combined naturally in a single constraint solver and AC5 can exploit the constraint semantics to obtain higher efficiency. NAC4 is also generalized for non-binary CSPs.
- It identifies classes of constraints for which domain consistency can be achieved in linear time.
- It demonstrates how the combination AC4/NAC4 can achieve domain consistency on logical combinations of constraints over the same variables.
- It shows that the combination of AC4/NAC4 can easily be extended to provide methods assessing the validity and the entailment of a constraint.
- It presents experimental results showing the benefits of the combination AC4/NAC4.

Related Work The idea of using forbidden values is not new in CP: what is novel in this paper is that NAC4 maintains the set of forbidden values dynamically during the propagation. References [3,4] use negative table constraints, where the table describes the set of forbidden tuples. The negative table is used to find the next support of a value by means of binary search; it is static however and not updated during propagation. Lecoutre [5] showed that (x, a) has a support for a constraint $c(x, y)$ if the size of $D(y)$ is strictly superior to the size of the initial conflict set of (x, a). This idea is integrated in a coarse-grained algorithm. Once again, the size of the conflict set is not updated during the computation. The same idea is also proposed as a support condition in [6].

The consistency of a combination of constraints has been handled in different ways. Some approaches achieve domain consistency, which is NP-hard in general. A domain-

consistency algorithm, based on AC7, was proposed in [7] for the conjunction of constraints. Lhomme [8] describes a domain-consistency algorithm for any combination of constraints. It focuses primarily on constraints given in extension. Forbidden tuples are used once again through a static negative table. Other approaches compute an approximation of domain consistency, such as in [9] (cardinality), [10] (constructive disjunction), or [11] which provides an algebra for combining constraints.

2 AC5

This section revisits the generic AC5 algorithm [2], generalizing it slightly to accommodate AC4, AC6, and AC2001 as instantiations.

Definition 1 (CSP). *A binary CSP* $(X, D(X), C)$ *is composed of a set of* n variables $X = \{x_1, \ldots, x_n\}$*, a set of* domains $D(X) = \{D(x_1), \ldots, D(x_n)\}$ *where* $D(x)$ *is the set of possible* values *for variable* x*, and a set of binary* constraints $C = \{c_1, \ldots, c_e\}$*, with* $Vars(c_i) \subseteq X$ $(1 \leq i \leq e)$*. We denote* $d = max_{1 \leq i \leq n}(\#D(x))$.

Let c be a constraint with $Vars(c) = \{x, y\}$, $a \in D(x), b \in D(y)$. $c(x/a, y/b)$ or $c(y/b, x/a)$ denote the constraint where variables x and y have been replaced by the values a and b. We assume that testing $c(x/a, y/b)$ takes $O(1)$ time. If $c(x/a, y/b)$ holds, then $(x/a, y/b)$ is called a support of c and (y, b) is a support for (x, a) on c. If $c(x/a, y/b)$ does not hold, then $(x/a, y/b)$ is called a conflict of c and (y, b) is a conflict (or forbidden value) for (x, a) on c.

Definition 2. *Let* c *be a constraint with* $Vars(c) = \{x, y\}$*. The set of inconsistent, consistent, and valid values for* x *in* c *wrt a set of values* B *are defined as follows:*

$$Inc(c, x, B) = \{(x, a) |\ a \in D(x) \wedge \forall b \in B : \neg c(x/a, y/b)\}$$
$$Cons(c, x, B) = \{(x, a) |\ a \in D(x) \wedge \exists b \in B : c(x/a, y/b)\}$$
$$Valid(c, x, B) = \{(x, a) |\ a \in D(x) \wedge \forall b \in B : c(x/a, y/b)\}$$

We use $Inc(c, x)$ *to denote* $Inc(c, x, D(y))$ *and similarly for the other sets.*

Definition 3 (Domain Consistency). *A constraint* c *over* $\{x, y\}$ *is* domain-consistent *wrt* $D(X)$ *iff* $Inc(c, x) = \emptyset$ *and* $Inc(c, y) = \emptyset$*. A CSP* $(X, D(X), C)$ *is* domain-consistent *iff all its constraints are domain-consistent wrt* $D(X)$.

Specification 1 describes the principal methods used by AC5. The AC5 algorithm uses a queue Q of triplets (c, x, a) stating that the domain consistency of constraint c should be reconsidered because value a has been removed from $D(x)$. When a value is removed from a domain, the method `enqueue` puts the necessary information on the queue. In the postcondition, Q_o represents the value of Q at call time. The parameter $C1$ allows us to consider a subset of constraints, which will be necessary in the initialization. As long as (c, x, a) is in the queue, it is algorithmically desirable to consider that value a is still in $D(x)$ from the perspective of constraint c. This is captured by the following definition.

```
1  enqueue(in x: Variable;in a: Value;in C1: Set of Constraints;
2          inout Q: Queue)
3  // Pre: x ∈ X, a ∉ D(x) and C1 ⊆ C
4  // Post: Q = Q0 ∪ {(c, x, a)|c ∈ C1, x ∈ Vars(c)}
5
6  post(in c: Constraint;out △: Set of Values)
7  // Pre: c ∈ C with Vars(c) = {x, y}
8  // Post: △ = Inc(c, x) ∪ Inc(c, y)  + initialization of specific data structures
9
10 boolean valRemove(in c: Constraint;in y: Variable; in b: Value;
11                   out △: Set of Values)
12 // Pre: c ∈ C, Vars(c) = {x, y} , b ∉ D(y, Q, c)
13 // Post: △1 ⊆ △ ⊆ △2  with  △1 = Inc(c, x, D(y, Q, c)) ∩ Cons(c, x, {b})
14 //                      and  △2 = Inc(c, x)
```

Specification 1: The enqueue, post, and valRemove Methods for AC5

Definition 4. *The local view of a domain $D(x)$ wrt a queue Q for a constraint c is defined as $D(x, Q, c) = D(x) \cup \{a|(c, x, a) \in Q\}$.*

Example 3. Given a queue $Q = \{(c_1, y, 2), (c_1, z, 2), (c_2, y, 3)\}$ and domains $D(x) = \{1, 2\}$, $D(y) = D(z) = \{1\}$, then $D(x, Q, c_1) = D(y, Q, c_1) = D(z, Q, c_1) = \{1, 2\}$.

The central method of the AC5 algorithm is the `valRemove` method, where the set $\triangle$ (called the delta-set in the folklore of CP due the use of the letter $\triangle$ in the original AC5 description) is the set of values no longer supported because of the removal of value b in $D(y)$. In this specification, b is a value that is no longer in $D(y)$ and `valRemove` computes the values (x, a) no longer supported because of the removal of b from $D(y)$. Note that values in the queue (for variable y) are still considered in the potential supports as their removal has not yet been reflected in this constraint. We also restrict our attention to values that had the value b in their support (i.e., $(x, a) \in Cons(c, x, \{b\})$). However, we leave `valRemove` the possibility of achieving more pruning ($\triangle_2$), which is useful for monotonic constraints [2].

The AC5 algorithm is depicted in Algorithm 1. Function `propagateQueueAC5` applies `valRemove` on each element of the queue until the queue is empty. Function `initAC5` initializes the queue. Function `post`(c,$\triangle$) computes the inconsistent values of the constraint c. If it removes values in some domains, only the already posted constraints are considered by the `enqueue` call. The constraints not yet posted are not concerned by such removals as they will directly use the current domain of the variables upon posting. The `post` call typically initializes some data structures to be used in `valRemove`. With a slight generalization of the specifications of `post` and `valRemove`, the AC5 algorithm also handles non-binary constraints. AC5 is generic because the implementation of `post` and `valRemove` is left open. Different constraints may have their own implementation of these functions. This allows AC5 to combine, in a single framework, different algorithms such as AC4, AC6, AC7, and AC2001 and to exploit the semantics of the constraints for achieving a better efficiency.

```
1  AC5(in X, C, inout D(X)){
2  // Pre: (X, D(X), C) is a CSP
3  // Post: D(X) ⊆ D(X)0, (X, D(X), C) equivalent to (X, D(X)0, C)
4  //       (X, D(X), C) is domain consistent
5    initAC5(Q);
6    propagateQueueAC5(Q);
7  }

8  initAC5(out Q){
9    Q = ∅;
10   C1 = ∅;
11   forall(c in C){
12     C1 += c;
13     post(c,△);
14     forall((x,a) in △){
15       D(x) -= a;
16       enqueue(x,a,C1,Q);
17     }
18   }
19 }

20 propagateQueueAC5(in Q){
21   while Q != ∅ {
22     select( (c,y,b) in Q) {
23       Q = Q - (c,y,b);
24       valRemove(c,y,b,△);
25       forall((x,a) in △){
26         D(x) -= a;
27         enqueue(x,a,C,Q);
28       }
29     }
30   }
31 }
```

Algorithm 1: The AC5 Algorithm.

Proposition 1. *Assuming a correct implementation of* `post` *and* `valRemove`*, AC5 is correct wrt its specification.*

In AC5, an element (c, x, a) can be put in the queue only once. The size of the queue is thus $O(e.r.d)$ for non-binary CSPs and $O(e.d)$ for binary CSPs. The number of executions of `valRemove` is also bounded by $O(e.r.d)$.

Proposition 2. *For binary CSPs, if the time complexity of post is $O(d^2)$, and the time complexity of* `valRemove` *is $O(d)$, then the time complexity of AC5 is the optimal $O(e.d^2)$. If the time complexity of* `post` *is $O(d)$ and the amortized time complexity of all the executions of* `valRemove` *for each constraint is $O(d)$ (e.g., time complexity of* `valRemove` *is $O(\triangle)$), then the time and space complexity of AC5 is $O(e.d)$.*

We now present AC4 as an instantiation of AC5 by giving the implementation of `post` and `valRemove` (Algorithm 2). `valRemoveAC4` uses a data structure S to record the supports of each value in the different constraints. It is initialized in `postAC4` and satisfies the following invariant at line 21 of Algorithm 1 (AC5).

Let $c \in C$ with $Vars(c) = \{x, y\}$:

$$(1.x) \quad \forall a \in D(x, Q, c)) : S[x, a, c] = \{b \in D(y, Q, c) | c(x/a, y/b)\}$$
$$(2.x) \quad \forall a \in D(x) : S[x, a, c] \neq \emptyset$$

```
1  postAC4(in c: Constraint;out △: Set of Values) {
2  // Pre: c ∈ C with Vars(c) = {x, y}
3  // Post: △ = Inc(c, x) ∪ Inc(c, y) + initialization of the S data structure
4    post_varAC4(c, x, △1);
5    post_varAC4(c, y, △2);
6    △ = △1 ∪ △2;
7  }
8  post_varAC4(in c: Constraint;in x: Variable;out △: Set of Values) {
9    △ = ∅;
10   forall(a in D(x)){
11     S[x, a, c] = ∅;
12     forall(b in D(y) : c(x/a, y/b))
13       S[x, a, c] += b ;
14     if (S[x, a, c]==∅)
15       △ += (x, a) ;
16   }
17 }
18 valRemoveAC4(in c: Constraint;in y: Variable;in b: Value;
19              out △: Set of Values) {
20 // Pre: c ∈ C, Vars(c) = {x, y} , b ∉ D(y, Q, c)
21 // Post: △ = Inc(c, x, D(y, Q, c)) ∩ Cons(c, x, {b})
22   △ = ∅;
23   forall(a in S[y, b, c]) {
24     S[x, a, c] -= b ;
25     if (S[x, a, c]==∅ & a in D(x))
26       △ += (x, a) ;
27   }
28 }
```

Algorithm 2: The `post` and `valRemove` Methods for AC4

And similarly for y. This invariant ensures the correctness of `valRemoveAC4`. After calling `postAC4`, we also have $\sum_{a \in D(x)} \#S[x, a, c] = \sum_{b \in D(y)} \#S[y, b, c]$ which is $O(d^2)$. The size of the data structure is $O(e.d^2)$.

3 NAC4

NAC4 (Negative AC4), another instance of AC5, is based on forbidden values that are dynamically maintained during the propagation. By NAC4, we mean the AC5 algorithm with the `postNAC4` and `valRemoveNAC4` methods depicted in Algorithms 3 and 4.

NAC4 uses a data structure F to record the forbidden values for each value in the different constraints. For a constraint c over x, y, the basic idea is that the value a should be removed from $D(x)$ as soon as the set of forbidden values for (x, a) and the set $D(y)$ are the same. This check can be performed efficiently by (1) reasoning about the sizes of the set of forbidden values for (x, a) and the set $D(y)$, (2) us-

```
1  postNAC4(in c: Constraint;out △: Set of Values) {
2  // Pre: c ∈ C with Vars(c) = {x, y}
3  // Post: △ = Inc(c, x) ∪ Inc(c, y)
4  //       + initialization of the F, setOfSize and localSize data structures
5    post_varNAC4(c, x, △1);
6    post_varNAC4(c, y, △2);
7    localSize[x, c] = #D(x);
8    localSize[y, c] = #D(y);
9    △ = △1 ∪ △2;
10 }
11 post_varNAC4(in c: Constraint;in x: Variable;out △: Set of Values) {
12   △ = ∅;
13   forall(k in 0..#D(y))
14     setOfSize[x, k, c] = ∅;
15   forall(a in D(x)){
16     F[x, a, c] = ∅;
17     forall(b in D(y) : ¬c(x/a, y/b))
18       F[x, a, c] += b ;
19     k = #F[x, a, c];
20     setOfSize[x, k, c]) += a;
21     if (k==#D(y))
22       △ += (x, a) ;
23   }
24 }
```

Algorithm 3: The `post` Algorithm for NAC4.

ing a data structure sorting the conflict sets by size, and (3) recording the size of the local view of the domains. These data structures are initialized in `postNAC4` and updated in `valRemoveNAC4`. The data structure $F[x, a, c]$ denotes the set of forbidden values for (x, a) and c, $setOfSize[x, k, c]$ denotes the set of values b such that $\#F[x, b, c] = k$, and $localSize[x, c]$ denotes the size of the local view of domain $D(x)$. `valRemoveNAC4` first updates the size of the local view of $D(y)$ and removes b from the $setOfSize$ data structure for variable y (lines 5–7). It then updates the set of forbidden values and $setOfSize$ for each pair $(x, a) \in F[y, b, c]$ (lines 8–13). Finally, it removes the values which are no longer supported, i.e., those values in $setOfSize[x, s, c] \cap D(x)$, where s is the local size of $D(y)$ (lines 14–18).

The data structures satisfy the following invariant at line 21 of Algorithm 1 (AC5). Let $c \in C$ with $Vars(c) = \{x, y\}$:

$(3.x)$ $\forall a \in D(x, Q, c) : F[x, a, c] = \{b \in D(y, Q, c) | \neg c(x/a, y/b)\}$
$(4.x)$ $\forall a \in D(x) : F[x, a, c] \subset D(y, Q, c)$
$(5.x)$ $setOfSize[x, k, c] = \{a \in D(x, Q, c)) | \#F[x, a, c] = k\} (0 \leq k \leq \#D(y, Q, c))$
$(6.x)$ $localSize[x, c] = \#D(x, Q, c)$

and similarly for y. From these invariants, we have that $F[x, a, c] \subseteq D(y, Q, c)$ at line 24 and the value a must be removed from $D(x)$ if $F[x, a, c] = D(y, Q, c)$. Hence, if

```
1  valRemoveNAC4(in c: Constraint;in y: Variable;in b: Value,
2                out △: Set of Values) {
3  // Pre: c ∈ C, Vars(c) = {x, y} , b ∉ D(y, Q, c)
4  // Post: △ = Inc(c, x, D(y, Q, c)) ∩ Cons(c, x, {b})
5    localSize[y, c]-- ;
6    k = #F[y, b, c];
7    setOfSize[y, k, c] -= b;
8    forall(a in F[y, b, c]){
9      F[x, a, c] -= b ;
10     k = #F[x, a, c];
11     setOfSize[x, k + 1, c] -= a;
12     setOfSize[x, k, c] += a;
13   }
14   △ = ∅;
15   s = localSize[y, c];
16   forall(a in setOfSize[x, s, c] : a in D(x))
17     △ += (x, a);
18 }
```

Algorithm 4: The valRemove method for NAC4

$s = localSize[y, c]$, the algorithm must remove the values in $setOfSize[x, s, c]$ from $D(x)$. These invariants ensure the correctness of `valRemoveNAC4`.

The size of the data structure is $O(e.d^2)$. In the pruning of the postNAC4 method, the local view of the size of domain $D(y)$ is $\#D(y)$ since the queue does not contain element of the form $(x, ., c)$. The complexity of `postNAC4` is $O(d^2)$ and the complexity of `valRemoveNAC4` is $O(d)$. Hence, by Property 2, the overall time complexity of NAC4 is $O(e.d^2)$, which has been shown to be the optimal complexity for achieving domain consistency on binary CSPs.

Example 4. We illustrate NAC4 on the following CSP:
$c_1(x, y) = \{(1,4),(1,5),(2,2),(2,5),(3,1),(3,3),(3,4)\}$, $c_2 : y \neq 4$, $c_3 : y \neq 5$, $D(x) = \{1,2,3\}$, and $D(y) = \{1,2,3,4,5\}$. The execution of `postNAC4`$(c_1, \triangle)$ yields $\triangle = \emptyset$ and fills the data structures as follows:

$$
\begin{array}{llll}
F[x,1,c_1] & = \{1,2,3\} & F[y,1,c_1] & = \{1,2\} \\
F[x,2,c_1] & = \{1,3,4\} & F[y,2,c_1] & = \{1,3\} \\
F[x,3,c_1] & = \{2,5\} & F[y,3,c_1] & = \{1,2\} \\
 & & F[y,4,c_1] & = \{2\} \\
 & & F[y,5,c_1] & = \{3\} \\
setOfSize[x,1,c_1] & = \emptyset & setOfSize[y,1,c_1] & = \{4,5\} \\
setOfSize[x,2,c_1] & = \{3\} & setOfSize[y,2,c_1] & = \{1,2,3\} \\
setOfSize[x,3,c_1] & = \{1,2\} & setOfSize[y,3,c_1] & = \emptyset \\
setOfSize[x,4,c_1] & = \emptyset & & \\
setOfSize[x,5,c_1] & = \emptyset & & \\
localSize[x,c_1] & = 3 & localSize[y,c_1] & = 5
\end{array}
$$

`postNAC4`($c_2, \triangle$) returns $\triangle = \{(y, 4)\}$ and `postNAC4`($c_3, \triangle$) yields $\triangle = \{(y, 5)\}$, giving $Q = \{(c_1, y, 4), (c_1, y, 5)\}$, $D(x) = \{1, 2, 3\}$, and $D(y) = \{1, 2, 3\}$. Method `valRemoveNAC4`($c_1, y, 4, \triangle$) updates the following variables:

$$\begin{array}{ll} F[x, 2, c_1] & = \{1, 3\} \\ setOfSize[x, 3, c_1] & = \{1\} \\ setOfSize[x, 2, c_1] & = \{2, 3\} \\ localSize[y, c_1] & = 4 \\ setOfSize[y, 1, c_1] & = \{5\} \end{array}$$

Since $setOfSize[x, 4, c_1] = \emptyset$, $\triangle = \emptyset$, $Q = \{(c_1, y, 5)\}$, $D(x) = \{1, 2, 3\}$, and $D(y) = \{1, 2, 3\}$. `valRemoveNAC4`($c_1, y, 5, \triangle$) updates the following variables:

$$\begin{array}{ll} F[x, 3, c_1] & = \{2\} \\ setOfSize[x, 2, c_1] & = \{2\} \\ setOfSize[x, 1, c_1] & = \{3\} \\ localSize[y, c_1] & = 3 \\ setOfSize[y, 1, c_1] & = \emptyset \end{array}$$

Since $setOfSize[x, 3, c_1] = \{1\}$, $\triangle = \{(x, 1)\}$, $Q = \{(c_1, x, 1)\}$, $D(x) = \{2, 3\}$, and $D(y) = \{1, 2, 3\}$. `valRemoveNAC4`($c_1, x, 1, \triangle$) updates the following variables:

$$\begin{array}{ll} F[y, 1, c_1] & = \{2\} \\ F[y, 2, c_1] & = \{3\} \\ F[y, 3, c_1] & = \{2\} \\ setOfSize[y, 2, c_1] & = \emptyset \\ setOfSize[y, 1, c_1] & = \{1, 2, 3\} \\ localSize[x, c_1] & = 2 \\ setOfSize[x, 3, c_1] & = \emptyset \end{array}$$

The domains are finally $D(x) = \{2, 3\}$ and $D(y) = \{1, 2, 3\}$.

Proposition 3. *Let $c \in C$ over $\{x, y\}$. Invariants (3-6.x-y) hold at line 21 of AC5.*

Proposition 4. *NAC4 is correct and its time and space complexity is $O(e.d^2)$.*

4 Applications

We now review a variety of applications of the principles of maintaining forbidden values.

Sparse AC Constraints As two instances of AC5, AC4 and NAC4 can be combined, each constraint implementing its AC4 or NAC4 version of the `post` and `valRemove` methods. This will be denoted AC5(AC4,NAC4). A nice property of AC5(AC4,NAC4) is that the amortized complexity of all the executions of `valRemoveAC4` or `valRemoveNAC4` for a constraint is bounded by the number of elements in the data structure S or F in this constraint. We then obtain the following specialization of Proposition 2.

Proposition 5. *If a specialization of* `postAC4` *or* `postNAC4` *exploiting the constraint semantics runs in time $O(K)$ for each constraint of a binary CSP, then the time and space complexity of AC5(AC4,NAC4) is $O(e.K)$.*

As a particular case, if S or F can be filled in $O(d)$, a domain-consistency algorithm runs in time $O(e.d)$, as formalized by the following class of constraints.

Definition 5. *A constraint c with $Vars(c) = \{x, y\}$ is* positively sparse *wrt a domain D iff $\#\{(a, b) \in D^2 | c(x/a, y/b)\}$ is $O(\#D)$. The constraint c is* negatively sparse *wrt D iff $\neg c$ is* positively sparse *wrt D.*

Example 5. Examples of positively and negatively sparse constraints are bijective constraints ($x + y = k$, where k is a constant), anti-bijective constraints ($x + y \neq k$), functional constraints ($x = |y - k|$ or $x = y \bmod k$), anti-functional constraints ($x \neq |y - k|$ or $x \neq y \bmod k$), but also include non (anti-)functional constraints such as $|x - y| = k$ and $|x - y| \neq k$. One can also consider congruence constraints, such as $(x + y) \bmod k = 0$ and $(x + y) \bmod k \neq 0$ which are sparse when k is $O(d)$.

Thanks to the genericity of AC5, we can exploit the semantics of the constraints in a specific `postAC4` method for positively sparse constraints and a specific `postNAC4` method for negatively sparse constraints to fill the data structure S or F in $O(d)$ and obtain a time complexity of $O(d)$.

Proposition 6. *For positively and negatively sparse constraints, AC5(AC4,NAC4) can run within a space and time complexity of $O(e.d)$.*

Combining Constraints on the same Variables Consider now a constraint c over $\{x, y\}$ defined as a boolean combination of constraints $\{c_1, \ldots, c_k\}$ on the same variables and assume for simplicity that the number of logical connectors is bounded by k. The constraint c can be posted in AC5(AC4,NAC4) with a complexity of $O(k.d^2)$. The propagation step on this constraint to achieve domain consistency will then run in time $O(K)$, where K is the number of supports.

Example 6. Consider the constraint $c \equiv (c_1 \wedge c_2) \vee (c_3 \wedge c_4)$ where $c_1 \equiv x \neq |y - 2|$, $c_2 \equiv y - 1 \neq x \bmod 2$, $c_3 \equiv x = |y - 1|$, $c_4 \equiv |x - 2| = y$, with $D(x) = \{0, 1\}$ and $D(y) = \{1, 2\}$. Each constraint c_i is domain-consistent, but neither $c_1 \wedge c_2$ nor $c_3 \wedge c_4$ are. Applying AC4 (or NAC4) on c will detect an inconsistency.

In some cases, such as in the above example, it is possible to achieve a better complexity by exploiting both supports and forbidden values. They key idea is that each constraint c_i should use either supports or forbidden values depending on its semantics. Then the individual constraints are combined through logical operators which use the supports and forbidden values to compute their own supports or forbidden values recursively. Table 1 depicts the rules to combine constraints and to compute a data structure S or F

$c^- \equiv \neg c_1^+$	$F[x,a,c] = S[x,a,c_1]$
$c^+ \equiv \neg c_1^-$	$S[x,a,c] = F[x,a,c_1]$
$c^+ \equiv c_1^+ \wedge c_2^+$	$S[x,a,c] = S[x,a,c_1] \cap S[x,a,c_2]$
$c^- \equiv c_1^- \wedge c_2^-$	$F[x,a,c] = F[x,a,c_1] \cup F[x,a,c_2]$
$c^+ \equiv c_1^+ \wedge c_2^-$	$S[x,a,c] = S[x,a,c_1] \setminus F[x,a,c_2]$
$c^+ \equiv c_1^+ \vee c_2^+$	$S[x,a,c] = S[x,a,c_1] \cup S[x,a,c_2]$
$c^- \equiv c_1^- \vee c_2^-$	$F[x,a,c] = F[x,a,c_1] \cap F[x,a,c_2]$
$c^- \equiv c_1^+ \vee c_2^-$	$F[x,a,c] = F[x,a,c_2] \setminus S[x,a,c_1]$

Table 1: Rules for Combining $c_1(x,y)$ and $c_2(x,y)$.

for the variables x and y according to the data structure maintained in the subexpressions. The rules are given for variable x but are similar for y. A constraint c_i using an S (resp. F) data structure will be denoted c_i^+ (resp. c_i^-). If the post method applies these rules on c, then the resulting algorithm achieves domain consistency. There is no time or space overhead as all the operations in Table 1 can be performed in time $s_1 + s_2$, where s_i is the size of the data structure (S or F) for c_i. As a particular case, if the time complexity to post each c_i is $O(d)$ (e.g. functional or anti-functional constraint), the time and space complexity of the post constraint for c is $O(k.d)$.

Proposition 7. *Given a set of binary constraints C and a binary constraint c expressed as logic combination of constraints $c_1, \ldots, c_k$ with $Vars(c) = Vars(c_i)$ $(1 \leq i \leq k)$, if the post method for c applies the rules of Table 1, then the resulting AC5(AC4,NAC4) algorithm on $C \cup \{c\}$ achieves domain consistency. If the time complexity of the post methods of the constraints in $C \cup \{c_1, \ldots, c_k\}$ is $O(d)$, then the time and space complexity of AC5(AC4,NAC4) applied on $C \cup \{c\}$ is $O((e+k).d)$, with $e = \#C$.*

Validity and Entailment AC5(AC4,NAC4) can be extended to support the `isValid` and `isEntailed` methods (Specification 2). In AC4, a value (x,a) is detected to be valid in c if the size of $S[x,a,c]$ is $\#D(y,Q,c)$. AC4 should then maintain the $setOfSize$ and $localSize$ data structures of NAC4. In NAC4, a value (x,a) is detected to be valid in c if $F[x,a,c]$ is empty. The invariant of the data structures for both AC4 and NAC4 would then be (1-6.x-y). The theoretical complexity of AC5(AC4,NAC4) is unchanged, while the practical complexity is roughly doubled. AC4 and NAC4 would keep the number of valid values for each constraint c. If the valid values for x in c reaches $\#D(x,Q,c)$, then the constraint is known to be entailed (assuming the domains are non empty). We could also easily extend the `post` and `valRemove` methods to `post`(c,$\triangle^-$,$\triangle^+$) and `valRemove`(c,y,b,$\triangle^-$,$\triangle^+$), where the extra argument $\triangle^+$ returns the set of new valid values, defined as

$$\triangle^+ = Valid(c,x) \cup Valid(c,y)$$

for `post`, and

$$\triangle^+ = Valid(c,x,D(y,Q,c)) \cap Inc(c,y,\{b\})$$

for `valRemove`. These extended domain consistency algorithms are useful for constraint combinators, reification and in an *Ask & Tell* framework.

```
1 Boolean isValid(in c: Constraint, in x: Variable, in a: Value)
2 // Pre: c ∈ C, Vars(c) = {x, y}, a ∈ D(x), D(y) ≠ ∅
3 // Post: return true iff (x, a) ∈ Valid(c, x, D(y, Q, c))
4 Boolean isEntailed(in c: Constraint)
5 // Pre: c ∈ C with Vars(c) = {x, y}, D(x) ≠ ∅, D(y) ≠ ∅
6 // Post: return true iff ∀a ∈ D(x) : (x, a) ∈ Valid(c, x, D(y, Q, c))
```

Specification 2: The `isValid` and `isEntailed` Methods.

Combining Constraints on Different Variables Achieving domain consistency on a combination of (binary) constraints on different variables is an NP-hard problem. An approximation of domain consistency can be achieved by using the framework proposed in [11], where primitive constraints produce not only the inconsistent values but also the valid ones. Our extended AC5(AC4,NAC4) can be used for combining constraints using the proposed algebra.

5 GNAC4: NAC4 for non-Binary Constraints

This section extends NAC4 to non-binary constraints. It is specified by methods `postGNAC4` and `valRemoveGNAC4` specified in Algorithms 5 and 6. A tuple or vector $(v_1, \ldots, v_n)$ is denoted by $\mathbf{v}$ and $\mathbf{v}[x_i]$ denotes the value v_i. We denote $D(X)_{x_i=a}$ the set of tuples $\mathbf{v}$ in $D(X)$ with $\mathbf{v}[x_i] = a$. Let $Y = \{x_1, \ldots, x_k\} \subseteq X$. The set of tuples in $D(x_1) \times \ldots \times D(x_k)$ is denoted $D(Y)$.

Definition 6. *Let c be a constraint with $x, y \in Vars(c)$, and $D(X) \subseteq B(X)$*

$$Inc(c, x, B(X)) = \{(x, a) |\ a \in D(x) \wedge \forall \mathbf{v} \in B(Vars(c))_{x=a} : \neg c(\mathbf{v})\}$$
$$Cons(c, x, y, b) = \{(x, a) |\ a \in D(x) \wedge \exists \mathbf{v} : \mathbf{v}[x] = a \wedge \mathbf{v}[y] = b \wedge c(\mathbf{v})\}.$$

We define $Inc(c, B(X)) = \bigcup_{x \in Vars(c)} Inc(c, x, B(X))$ and similarly for $Cons(c, y, b)$.

The data structure F is generalized and satisfies the following invariant at line 21 of Algorithm 1 (AC5). Let $c \in C$ with $x \in Vars(c)$:

$$(3'.x) \quad \forall a \in D(x, Q, c) : F[x, a, c] = \{\mathbf{v} \in D(Vars(c), Q, c)_{x=a} | \neg c(\mathbf{v})\}$$
$$(4'.x) \quad \forall a \in D(x) : F[x, a, c] \subset D(Vars(c), Q, c)_{x=a}$$
$$(5'.x) \quad setOfSize[x, k, c] = \{a \in D(x, Q, c)) | \#F[x, a, c] = k\}$$
$$(6'.x) \quad localSize[x, c] = \#D(x, Q, c)$$

and similarly for the other variables of c. The time complexity of `postGNAC4` is $O(r.d^r)$. The time complexity of `valRemoveNGAC4` is $O(r.d^{r-1})$. However, an amortized analysis of GAC4 shows that each element in F can only be removed once, hence a global complexity of $O(e.r.d^r)$ for all the executions of `valRemoveGNAC4`. The complexity of GNAC4 is thus the optimal $O(e.r.d^r)$.

```
1  postGNAC4(in c: Constraint;out △: Set of Values) {
2  // Pre: c ∈ C
3  // Post: △ = Inc(c)
4  //       + initialization of the F, setOfSize and localSize data structures
5    forall(x in Vars(c), k in 1..#D(Vars(c) \ {x}) )
6      setOfSize[x,k,c] = ∅;
7    forall(x in Vars(c), a in D(x))  F[x,a,c] = ∅ ;
8    forall(v in D(Vars(c)): ¬c(v))
9      forall(x in Vars(c))
10       F[x,v[x],c] += v;
11   △ = ∅;
12   forall(x in Vars(c), a in D(x)){
13     k = #F[x,a,c];
14     setOfSize[x,k,c]) += a;
15     if (k==#D(Vars(c) \ {x}))
16       △ += (x,a) ;
17   }
18 }
```

Algorithm 5: The `post` Method for GNAC4

6 Experimental Results

This section illustrates the benefits of jointly exploiting supports and forbidden values. We evaluated AC4, NAC4, and their combination on CSPs involving the positively and negatively sparse constraints $x = y \bmod k$, $x = |y - k|$, $x + y = k$, $|x - y| = k$, $(x + y) \bmod k = 0$ and their negative version, where k is a constant. Three sets of 20 CSPs were generated: a set with only positive constraints (cPos), a set with only negative constraint (cNeg), and a set with positive and negative constraints (cPosNeg). The results are presented in Table 2. The name cNeg_50_200_10 means that each CSP has 50 variables with a domain $\{0..199\}$, and 10% of constraints between all the pairs of distinct variables. The settings were chosen to avoid trivially consistent or trivially inconsistent CSPs. The constraints were randomly chosen using a uniform distribution. The values k are also determined using a uniform distribution. Each CSP has been solved in Comet using four different consistency algorithms: (1) AC4 for each constraint, (2) NAC4 for each constraint, (3) the combination AC5(AC4,NAC4) using AC4 for positive constraints and NAC4 for negative constraints and (4) the combination AC5(AC4*,NAC*) which is similar to AC5(AC4,NAC4) but uses specialized (linear) post methods exploiting the semantics of the constraints. For CSPs with only positive constraints, AC5(AC4,NAC4) reduces to AC4 and, for CSPs with only negative constraints, AC5(AC4,NAC4) reduces to NAC4. The average execution time (in seconds) is reported. We also report the percentage of consistent CSPs in each data set. The inconsistency of the CSPs was always detected in the root node of the search tree. For consistent CSPs, the search is terminated after 1000 fail nodes. The experiments were performed on a single core of a machine with an Intel Core Duo at 2.8GHz with 4GB memory.

```
1  valRemoveGNAC4(in c: Constraint;in y: Variable;in b: Value,
2                 out △: Set of Values) {
3  // Pre: c ∈ C, y ∈ Vars(c) , b ∉ D(y, Q, c)
4  // Post: △ = Inc(c, D(X, Q, c)) ∩ Cons(c, y, b)
5    localSize[y, c]-- ;
6    k = #F[y, b, c];
7    setOfSize[y, k, c]) -= b;
8    forall(v in F[y, b, c]){
9      forall(x in Vars(c) \ {y}){
10       a = v[x];
11       F[x, a, c] -= v ;
12       k = #F[x, a, c];
13       setOfSize[x, k + 1, c]) -= a; setOfSize[x, k, c]) += a;
14     }
15   }
16   △ = ∅;
17   s = ∏_{x∈Vars(c)} localSize[x, c];
18   forall(x in Vars(c) \ {y}){
19     s1 = s/localSize[x, c];
20     forall(a in setOfSize[x, s1, c] : a in D(x))
21       △ += (x, a);
22   }
23 }
```

Algorithm 6: The `valRemove` Method for GNAC4

	% Consist.	AC4	NAC4	AC5(AC4,NAC4)	AC5(AC4*,NAC4*)
cPos_10_200_01	55%	0.466	19.198	-	0.181
cNeg_50_200_10	100%	27.596	2.381	-	2.027
cPosNeg_50_200_05	53%	21.690	76.073	1.700	1.454

Table 2: Comparing AC4, NAC4, AC5(AC4,NAC4) and AC5(AC4*/NAC4*).

For positively sparse constraints, AC4 is much more efficient (speedup of 41) than NAC4, while NAC4 is much more efficient than AC4 (speedup of 11.5) on negatively sparse constraints. This shows the interest of NAC4. Using a specialized post constraint leads to a speedup of 2.57 for AC4 and 1.17 for NAC4. For CSPs combining positively and negatively sparse constraints, NAC4 is 3.5 times slower than AC4, which is is explained by the more complicated data structures maintained by NAC4. This last set of CSPs shows the interest of a generic algorithm allowing the combination of different algorithms such as AC4 and NAC4. The speedup of AC5(AC4,NAC4) compared to AC4 is 12.7. This speedup increases to 14.9 when using specialized post methods in AC5(AC4*,NAC4*).

7 Conclusion

This paper proposed the optimal domain-consistency algorithm NAC4 which is not based on supports but dynamically maintains forbidden values during propagation. The ideas behind NAC4 can be combined within the AC5 algorithm with the techniques

used in AC4, AC6, and AC2001 for exploiting the semantics of constraints and obtaining greater efficiency. In particular, forbidden values allow AC5 to achieve domain consistency in time $O(ed)$ for classes of constraints in which the number of supports is $O(d^2)$ but the number of forbidden values is $O(d)$. The paper also shows how forbidden values and supports can be used jointly to achieve domain consistency on logical combinations of constraints and to compute validity and entailment of constraints. Experimental results show that the combination of supports and forbidden values can bring significant computational benefits in arc-consistency algorithms.

Future work includes the comparison of AC5(AC4/NAC4) with other AC algorithms, experimental evaluation on other benchmarks, including non-binary CSP instances and extension of NAC4 to handle negative tables represented in a compact way such as in [12,13,14].

Acknowledgment Many thanks to Jean-Noël Monette for his help in the experimental setting. This research is partially supported by the Interuniversity Attraction Poles Programme (Belgian State, Belgian Science Policy) and the FRFC project 2.4504.10 of the Belgian FNRS (National Fund for Scientific Research).

References

1. Bessiere, C.: Constraint propagation. In Rossi, F., Beek, P.v., Walsh, T., eds.: Handbook of Constraint Programming. Elsevier Science Inc., New York, NY, USA (2006)
2. Van Hentenryck, P., Deville, Y., Teng, C.M.: A generic arc-consistency algorithm and its specializations. Artif. Intell. **57**(2-3) (1992) 291–321
3. Bessière, C., Régin, J.C.: Arc consistency for general constraint networks: Preliminary results. In: IJCAI. (1997) 398–404
4. Lecoutre, C.: Constraint Networks: Techniques and Algorithms. ISTE/Wiley (2009)
5. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Support inference for generic filtering. In: CP. (2004) 721–725
6. Mehta, D., van Dongen, M.R.C.: Reducing checks and revisions in coarse-grained MAC algorithms. In: IJCAI. (2005) 236–241
7. Bessière, C., Régin, J.C.: Local consistency on conjunctions of constraints. In: ECAI-98, proceedings Workshop on Non Binary Constraints. (1998) 53–60
8. Lhomme, O.: Arc-consistency filtering algorithms for logical combinations of constraints. In: CPAIOR. (2004) 209–224
9. Van Hentenryck, P., Deville, Y.: The cardinality operator: A new logical connective for constraint logic programming. In: ICLP. (1991) 745–759
10. Van Hentenryck, P., Saraswat, V.A., Deville, Y.: Design, Implementation, and Evaluation of the Constraint Language cc(FD). In: Constraint Programming: Basics and Trends. Springer (1994) 293–316
11. Bacchus, F., Walsh, T.: Propagating logical combinations of constraints. In: IJCAI. (2005) 35–40
12. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: AAAI'07. (2007) 191–197
13. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: CP'07, Springer-Verlag (2007) 379–393
14. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on ad hoc r-ary constraints. In: CP'08, Springer-Verlag (2008) 509–523