Noname manuscript No.
(will be inserted by the editor)

Domain Consistency with Forbidden Values

Yves Deville, Pascal Van Hentenryck and Jean-Baptiste Mairy

the date of receipt and acceptance should be inserted later

Abstract This paper presents a novel domain-consistency algorithm which does not maintain supports dynamically during propagation, but rather maintains forbidden values. It introduces the optimal NAC4 (negative AC4) algorithm based on this idea, as an instance of the generic algorithm AC5. The paper then 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. The combination of NAC4 and AC4, denoted byPNAC4, allows 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 $O(d)$, or conversely. The paper also presents a simple variant of AC3, denoted PNAC3. Both PNAC4 and PNAC3 are especially efficient on classes of constraints offering a $O(1)$ `getSupports` or `getForbidden` function. Experimental results show that, on these particular classes of constraints, the joint exploitation of supports and forbidden values outperforms the standard AC algorithms, and that the use of a specialized `getSupports` or `getForbidden` function enhances the efficiency of the algorithms, especially for PNAC3 which is very close to the efficiency of totally dedicated consistency algorithms.

1 Introduction

In constraint programming, propagation aims at reducing the search space without removing solutions. The propagation algorithm considers each constraint individually and terminates

Yves Deville, Jean-Baptiste Mairy
Université catholique de Louvain,
Institute of Information and Communication Technologies, Electronics and Applied Mathematics
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium
E-mail: {Yves.Deville,Jean-Baptiste.Mairy}@uclouvain.be

Pascal Van Hentenryck
Optimization Research Group, NICTA, Victoria Research Laboratory, Electrical and Electronic Engineering,
The University of Melbourne, VIC 3010, Australia
E-mail: pvh@nicta.com.au

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. Domain consistency is ideal because it is the best propagation that can be achieved when considering the constraints locally. Stronger forms of consistency require to consider pairs or sets of constraints simultaneously.

Many algorithms have been proposed for achieving domain consistency, such as AC3 [20], AC4 [23], AC6 [2] and AC2001 [6]. Consistency algorithms typically use the concept of *support*. For a binary constraint c 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, where $c(x/a, y/b)$ is the constraint c with x replaced by a and y by b. The optimal time complexity to achieve domain consistency for a CSP is $O(ed^2)$ for binary constraints and $O(erd^r)$ 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) [3]. 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 1.1 Consider the constraint $x = y \bmod 10$, with $D(x) = \{0..9\}$ and $D(y) = \{0..99\}$. The number of supports for x is linear (that is $O(\#D(y))$, where $\#A$ is the size of A). The propagation step of AC4 for this constraint is also linear as the time complexity of the propagation step of AC4 is proportional to the number of supports. However, the time complexity of the propagation step of other optimal AC algorithms such as AC6, AC7, or AC2001, is independent of the number of supports and remains quadratic.

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 [27] 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 a $O(ed)$ 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 in a constraint is quadratic in the size of the domain, but the number of forbidden values (also called conflict set) is linear?*

Example 1.2 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 $O(\#D(x) \cdot \#D(y))$ 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 ($O(\#D(y))$). The main question in this paper is if there is 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 $O(ed^2)$ time complexity for binary CSPs, but dynamically maintains the number of forbidden values

instead of supports. As AC4 and NAC4 are instances of the generic AC5 algorithm, they can be combined naturally in a single consistency algorithm PNAC4 that can exploit the constraint semantics to obtain higher efficiency.

- It demonstrates how PNAC4 can efficiently achieve domain consistency on logical combinations of constraints over the same variables.
- It shows that PNAC4 can easily be extended to provide efficient methods assessing the validity and the entailment of a constraint.
- It identifies classes of constraints for which domain consistency can be achieved in linear time with PNAC4.
- It presents PNAC3, a simple variant of AC3, that uses a `getSupports` or `getForbidden` function provided by the constraints. For classes of constraints with a $O(1)$ `getSupports` or `getForbidden` function, PNAC3 has a $O(ed^2)$ time complexity.
- It presents experimental results showing the benefits of PNAC4 and PNAC3 on these classes of constraints.

Related Work This paper is an extended version of [10]. The idea of using forbidden values is not new in CP: what is novel in this paper is that NAC4 maintains the number of forbidden values dynamically during the propagation. References [4,18,9,15,21] discuss 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 [7] showed that (x, a) has a support for a constraint $c(x, y)$ if the size of $D(y)$ is greater than 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 [21].

The consistency of a combination of constraints has been handled in different ways. Some approaches achieve domain consistency, which is NP-hard in general [1]. A domain-consistency algorithm, based on AC7, was proposed in [5] for the conjunction of constraints. Lhomme [19] 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 [26] (cardinality), [28] (constructive disjunction), or [1] which provides an algebra for combining constraints.

Overview The paper is structured as follows. Section 2 revisits the AC5 algorithm and shows how to specialize it to AC4. The NAC4 algorithm, another instance of AC5, is described in Section 3. Section 4 presents applications of the idea of maintaining forbidden values and/or supports dynamically, i.e., novel classes of constraints that admit a $O(ed)$ domain consistency algorithm, domain consistency on logical combinations of constraints, value validity, and constraint entailment. Section 5 presents PNAC3, a simple variant of AC3. Experimental results showing the benefits of PNAC4, the integration of AC4 and NAC4, and PNAC3 are provided in Section 5. Section 6 concludes the paper.

2 AC5

In consistency algorithms, one usually distinguishes constraint-based algorithms (also called coarse-grained) from value-based algorithms (also called fine-grained) [3]. In a constraint-based algorithm, such as AC3 and AC2001, the propagation is performed at the level of constraints. In a value-based algorithm, such as in AC4 and AC6, the propagation is performed at the level of values.

This section revisits the AC5 algorithm [27]. It is a generic value-based consistency algorithm that can be instantiated into AC4, AC6 and our new NAC4 algorithm. If $c(x, y)$ is a (binary) constraint, the set of variables $\{x, y\}$ is denoted by *Vars*(c).

Definition 2.1 (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 $D(x_i)$ is the set of possible *values* for variable x_i, 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*$(c) = \{x, y\}$, $a \in D(x), b \in D(y)$, and $c(x/a, y/b)$ 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, as well as removing a value from a domain. 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 2.2 Let c be a constraint with *Vars*$(c) = \{x, y\}$. The set of *inconsistent*, *consistent*, and *valid* values for x in c with respect to a set of values B are defined as follows:

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

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

Definition 2.3 (Domain Consistency) A constraint c over $\{x, y\}$ is *domain-consistent* wrt $D(X)$ if and only if (iff) *Inc*$(c, x) = \emptyset$ and *Inc*$(c, y) = \emptyset$. A binary 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 of specifications, Q_o and $D(X)_0$ represents the value of Q and $D(X)$ 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, the constraint c is not aware of the removal of value a from $D(x)$. The data structure of the constraint might consider that the value a is not removed from $D(x)$. It is thus 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.

Definition 2.4 The *local view* of a domain $D(x)$ wrt a queue Q for a constraint c, denoted $D(x, Q, c)$, is defined as the set $D(x) \cup \{a | (c, x, a) \in Q\}$.

```
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

Example 2.1 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 $D(x, Q, c_1) = D(y, Q, c_1) = D(z, Q, c_1) = \{1, 2\}$. We also have $D(x, Q, c_2) = \{1, 2\}$, $D(y, Q, c_2) = \{1, 3\}$ and $D(z, Q, c_2) = \{1\}$.

The central method of the AC5 algorithm is the `valRemove` method, where the set $\triangle$ is the set of values no longer supported because of the removal of value b in $D(y)$. The set $\triangle$ is called the delta-set in the folklore of CP due the use of the letter $\triangle$ in the original AC5 description. 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., $(x, a) \in \mathit{Cons}(c, x, \{b\})$). However, we leave `valRemove` the possibility of achieving more pruning ($\triangle_2$), which is useful for monotonic constraints [27]. As usual in specifications, we assume that a correct implementation only affect variables and parameters as described in the postcondition.

The AC5 algorithm is depicted in Algorithm 1. We extend the usual C notations `X += a` and `X -= a` to add and remove an element a to and from a set X. The method `propagateQueueAC5` applies `valRemove` on each element of the queue until the queue is empty. The method `initAC5` initializes the queue. The method `post`(c,$\triangle$) computes the inconsistent values of the constraint c. If it removes values from some domains, only the already posted constraints are considered by the `enqueue` call. The constraints not yet posted are not considered in 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 NAC4 and to exploit the semantics of the constraints for achieving a better efficiency.

Proposition 2.1 *Assuming a correct implementation of* `post` *and* `valRemove`*, AC5 is correct wrt its specification.*

```
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 }
```

Algorithm 1: The AC5 Algorithm.

Proof Let $D(X)^* \subseteq D(X)_0$ be the largest domain such that $(X, D(X)^*, C)$ is domain consistent. We have $(X, D(X)^*, C)$ equivalent to $(X, D(X)_0, C)$. We thus have to show that AC5 computes the domain $D(X)^*$.

(1) $D(X)^* \subseteq D(X)$ is an invariant at lines 11 and 21. It is obvious as AC5 only removes inconsistent values of $D(X)$.

(2) If AC5 terminates, $(X, D(X), C)$ is domain consistent. Suppose it is *not* domain consistent after the execution. There must then exist $c \in C$ with $\mathit{Vars}(c) = \{x, y\}$ and $a \in D(x)$ such that $\forall b \in D(y) : \neg c(x/a, y/b)$. The value a must thus be supported by some elements in the initial domain $D(y)_0$, otherwise a would had been removed from $D(x)$ after post$(c, \triangle)$. Let $b_1, \ldots, b_m$ $(m > 0)$ be all these supports in $D(y)_0$. These supports are thus removed during the execution of AC5. The supports cannot be all removed before the execution of post$(c, \triangle)$, otherwise (x, a) would be in $\triangle$, and a removed from $D(x)$. Let $b'_1, \ldots, b'_{m'}$ $(m' > 0)$ be the remaining supports that are removed after the execution of post$(c, \triangle)$. The tuples (c, y, b'_i) $(1 \leq i \leq m')$ have been enqueued in Q. When the last of this tuples, denoted (c, y, b'_k), is dequeued, valRemove$(c, y, b'_k, \triangle)$ yields $(x, a) \in \triangle$ as a has no more support in $D(y, Q, c)$. Hence a is removed from $D(x)$. This is a contradiction.

(3) AC5 terminates as an element (c, x, a) can be put only once in Q. ∎

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

Proposition 2.2 *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 $O(ed^2)$. 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* $O(ed)$.

AC4 as an instance of AC5 We now present AC4 [22] as an instantiation of AC5 by giving the implementation of `post` and `valRemove` (Algorithm 2). The `postAC4` method uses the `getSupports` function of the constraint (Specification 2). The methods use a data structure *nbSupport* recording the number of supports of each value in the constraint. It also uses a data structure S recording the set of supports of each value in the constraint when the constraint is posted. The data structure S is thus not updated when domains are pruned. The data structures are initialized in `postAC4` and satisfies the following invariant at line 21 of AC5 (Algorithm 1).

Let $c \in C$ with *Vars*$(c) = \{x, y\}$,
and *Supp*(x, a, c) denote the set $\{b \in D(y, Q, c) | c(x/a, y/b)\}$:

$$
\begin{array}{ll}
(1.x) & \forall a \in D(x) : \textit{nbSupport}[x, a, c] = \#\textit{Supp}(x, a, c) \\
(2.x) & \forall a \in D(x) : \textit{nbSupport}[x, a, c] > 0 \\
(3.x) & \forall a \in D(x) : \textit{Supp}(x, a, c) \subseteq S[x, a, c] \\
(4.x) & \forall a \in D(x)\ \forall b \in S[x, a, c] : c(x/a, y/b)
\end{array}
$$

And similarly for y (denoted (1.y) to (4.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 $O(ed^2)$. In the following, the notation (1-2.x) denotes the invariants (1.x) and (2.x), and similarly for (1.x-y) and (1-2,x-y).

Proposition 2.3 *Let* $c \in C$ *with* *Vars*$(c) = \{x, y\}$. *Invariants (1-4.x-y) hold at line 21 of AC5.*

Proof Proof by induction. (Base case) The execution of `post`$(c, \triangle)$ establishes Invariants (1.x-y) and (3-4.x-y) with $D(x, Q, c) = D(x)$ and $D(y, Q, c) = D(y)$. The removal of the values in $\triangle$ from the domain ensures Invariants (2.x-y). Any subsequent removal of values in $D(x)$ or in $D(y)$ by the post of other constraints does not change $D(x, Q, c)$ nor $D(y, Q, c)$. The invariants thus hold when the execution reaches line 21 for the first time.
(Induction step) Suppose the invariants holds at line 21 of AC5 and `valRemove`$(c, y, b, \triangle)$ is executed because (c, y, b) has been removed from the queue. Before the execution of `valRemove`, Invariants (1-4.y), (2.x) and (4.x) hold, as well as (1.x) and (3.x) but for $D(y, Q, c) \cup \{b\}$. The execution of `valRemove`$(c, y, b, \triangle)$ and the following enqueue instructions do not affect Invariants (1-2,y) as well as Invariants (3-4,x-y) because S is not modified, $D(y, Q, c)$ has one less element (b) and $D(x, Q, c')$ is unchanged for all $c' \in C$. For Invariant (1.x), from the induction hypothesis, we have $\forall a \in D(x) : \textit{nbSupport}[x, a, c] = \#\{b' \in D(y, Q, c) \cup \{b\} | c(x/a, y/b')\}$. Hence we have to decrement *nbSupport*$[x, a, c]$ for all a in $D(x)$ for which $c(x/a, y/b)$ holds, that is for all $a \in D(x) \cap S[y, b, c]$ (by Invariants (3-4.y)). Invariant (2.x) is ensured after the removal of the values in $\triangle$ from the domain $D(x)$. ∎

Proposition 2.4 *AC4 is correct wrt its specification. Its time and space complexity are* $O(ed^2)$*; its time complexity is thus optimal.*

```
function getSupports(in c: Constraint;in x: Variable; \\
                     in a: Value) : Set of Values
// Pre: c ∈ C, Vars(c) = {x, y}
// Post: return {b ∈ D(y)|c(x/a, y/b)}
```

Specification 2: The `getSupports` function of a constraint

```
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 data structures
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] = getSupports(c, x, a) ;
12     nbSupport[x, a, c] = # S[x, a, c] ;
13     if (nbSupport[x, a, c]==0)
14       △ += (x, a) ;
15   }
16 }
17 valRemoveAC4(in c: Constraint;in y: Variable;in b: Value;
18              out △: Set of Values) {
19 // Pre: c ∈ C, Vars(c) = {x, y} , b ∉ D(y, Q, c)
20 // Post: △ = Inc(c, x, D(y, Q, c)) ∩ Cons(c, x, {b})
21   △ = ∅;
22   forall(a in S[y, b, c] : a in D(x)) {
23     nbSupport[x, a, c] -- ;
24     if (nbSupport[x, a, c]==0)
25       △ += (x, a) ;
26   }
27 }
```

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

Proof By Proposition 2.1, it is sufficient to prove the correctness of `postAC4` and `valRemoveAC4`. We also know that Invariants (1-4.x-y) hold at line 21 of AC5.

postAC4($x, \triangle$) is correct. Obvious.

To show that valRemoveAC4($c, y, b, \triangle$) is correct, we need that

$$\triangle = \{(x, a) | a \in D(x) \wedge c(x/a, y/b) \wedge \forall b' \in D(y, Q, c) : \neg c(x/a, y/b')\}.$$

When entering valRemoveAC4, all the invariants holds, with (1.x) and (3.x) holding for $D(y, Q, c) \cup \{b\}$. We have:

$$\begin{aligned} & a \in D(x) \wedge c(x/a, y/b) \wedge \forall b' \in D(y, Q, c) : \neg c(x/a, y/b') \\ \text{iff } & a \in D(x) \wedge a \in S[y, b, c] \wedge \forall b' \in D(y, Q, c) : \neg c(x/a, y/b')) \quad \text{(by (3-4.y))} \\ \text{iff } & a \in D(x) \wedge a \in S[y, b, c] \wedge \mathit{nbSupport}[x, a, c] = 1 \quad \text{(by (1.x) for } D(y, Q, c) \cup \{b\}). \end{aligned}$$

The set $\triangle$ thus contains the required values.

The time complexity of postAC4 is $O(d^2)$, assuming a time complexity of $O(d)$ of `getSupports`. The time complexity of valRemoveAC4 is $O(d)$. Hence by Proposition 2.2, the time complexity of AC4 is O(ed^2). The space complexity is obviously O(ed^2). ■

3 NAC4

NAC4 (Negative AC4), another instance of AC5, is based on forbidden values that are dynamically maintained during the propagation. A value b of variable y is a *forbidden value* for value a of x if $c(x/a, y/b)$ does not hold. By NAC4, we mean the AC5 algorithm with the `postNAC4` and `valRemoveNAC4` methods depicted in Algorithms 3 and 4. The `postNAC4` method uses the `getForbidden` function of the constraint (Specification 3).

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) using 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 $F[x, a, c]$ contains the set of forbidden values for (x, a) on c when posting the constraint, *nbForbidden*$[x, a, c]$ contains the number of forbidden values for (x, a) on c. The F data structure is static and is thus not updated during the propagation. The data structure *setOfSize*$[x, k, c]$ denotes the set of values b such that *nbForbidden*$[x, b, c] = k$, and *localSize*$[x, c]$ denotes the size of the local view of domain $D(x)$. The method `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 number of forbidden values and *setOfSize* for each $(x, a) \in F[y, b, c]$ with a in $D(x)$ (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 14–17).

The data structures satisfy the following invariant at line 21 of AC5 (Algorithm 1). Let $c \in C$ with *Vars*$(c) = \{x, y\}$, and *Forb*(x, a, c) denote the set $\{b \in D(y, Q, c) | \neg c(x/a, y/b)\}$:

$(5.x)$ $\forall a \in D(x) : \textit{nbForbidden}[x, a, c] = \#\textit{Forb}(x, a, c)$
$(6.x)$ $\forall a \in D(x) : \textit{nbForbidden}[x, a, c] < \#D(y, Q, c)$
$(7.x)$ $\forall a \in D(x) : \textit{Forb}(x, a, c) \subseteq F[x, a, c]$
$(8.x)$ $\forall a \in D(x)\ \forall b \in F[x, a, c] : \neg c(x/a, y/b)$
$(9.x)$ $\textit{setOfSize}[x, k, c] = \{a \in D(x, Q, c)) | \#\textit{Forb}(x, a, c) = k\} (0 \leq k \leq \#D(y, Q, c))$
$(10.x)$ $\textit{localSize}[x, c] = \#D(x, Q, c)$

and similarly for y. From these invariants, we have that *nbForbidden*$[x, a, c] \leq \#D(y, Q, c)$ before line 24 of AC5 and the value a must be removed from $D(x)$ if *nbForbidden*$[x, a, c] = \#D(y, Q, c)$. Hence, if $s = \textit{localSize}[y, c]$, 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 structures is $O(ed^2)$. 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 any 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 2.2, the overall time complexity of NAC4 is $O(ed^2)$, which has been shown to be the optimal complexity for achieving domain consistency on binary CSPs.

```
function getForbidden(in c: Constraint; in x: Variable;
                      in a: Value) : Set of Values
// Pre: c ∈ C, Vars(c) = {x, y}
// Post: return {b ∈ D(y)|¬c(x/a, y/b)}
```

Specification 3: The `getForbidden` function of a constraint

```
postNAC4(in c: Constraint;out △: Set of Values) {
// Pre: c ∈ C with Vars(c) = {x, y}
// Post: △ = Inc(c, x) ∪ Inc(c, y)
//       + initialization of the F, setOfSize and localSize data structures
  post_varNAC4(c,x,△1);
  post_varNAC4(c,y,△2);
  localSize[x, c] = #D(x);
  localSize[y, c] = #D(y);
  △ = △1 ∪ △2;
}
post_varNAC4(in c: Constraint;in x: Variable;out △: Set of Values) {
  △ = ∅;
  forall(k in 0..#D(y))
    setOfSize[x, k, c] = ∅;
  forall(a in D(x)){
    F[x, a, c] = getForbidden(c, x, a);
    k = #F[x, a, c];
    nbForbidden[x, a, c] = k ;
    setOfSize[x, k, c]) += a;
    if (nbForbidden[x, a, c]==#D(y))
      △ += (x, a) ;
  }
}
```

Algorithm 3: The `post` Algorithm for NAC4.

Example 3.1 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:

$$F[x, 1, c_1] = \{1, 2, 3\} \qquad F[y, 1, c_1] = \{1, 2\}$$
$$F[x, 2, c_1] = \{1, 3, 4\} \qquad F[y, 2, c_1] = \{1, 3\}$$
$$F[x, 3, c_1] = \{2, 5\} \qquad F[y, 3, c_1] = \{1, 2\}$$
$$F[y, 4, c_1] = \{2\}$$
$$F[y, 5, c_1] = \{3\}$$

$$\mathit{nbForbidden}[x, 1, c_1] = 3 \qquad \mathit{nbForbidden}[y, 1, c_1] = 2$$
$$\mathit{nbForbidden}[x, 2, c_1] = 3 \qquad \mathit{nbForbidden}[y, 2, c_1] = 2$$
$$\mathit{nbForbidden}[x, 3, c_1] = 2 \qquad \mathit{nbForbidden}[y, 3, c_1] = 2$$
$$\mathit{nbForbidden}[y, 4, c_1] = 1$$
$$\mathit{nbForbidden}[y, 5, c_1] = 1$$

```
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 = nbForbidden[y, b, c];
7    setOfSize[y, k, c] -= b;
8    forall(a in F[y, b, c] : a in D(x)){
9      nbForbidden[x, a, c] -- ;
10     k = nbForbidden[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

$$
\begin{array}{ll}
\mathit{setOfSize}[x,0,c_1]=\emptyset & \mathit{setOfSize}[y,0,c_1]=\emptyset \\
\mathit{setOfSize}[x,1,c_1]=\emptyset & \mathit{setOfSize}[y,1,c_1]=\{4,5\} \\
\mathit{setOfSize}[x,2,c_1]=\{3\} & \mathit{setOfSize}[y,2,c_1]=\{1,2,3\} \\
\mathit{setOfSize}[x,3,c_1]=\{1,2\} & \mathit{setOfSize}[y,3,c_1]=\emptyset \\
\mathit{setOfSize}[x,4,c_1]=\emptyset & \\
\mathit{setOfSize}[x,5,c_1]=\emptyset &
\end{array}
$$

$$
\mathit{localSize}[x,c_1]=3 \qquad \mathit{localSize}[y,c_1]=5
$$

`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 $D(y) = \{1,2,3\}$.

The method `valRemoveNAC4`$(c_1, y, 4, \triangle)$ updates the following variables:

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

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

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

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

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

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

Proposition 3.1 *Let $c \in C$ with Vars$(c) = \{x, y\}$. Invariants (5-10.x-y) hold at line 21 of AC5.*

Proof Proof by induction. (Base case) Consider the execution of `post`$(c, \triangle)$. After the execution of lines 5 and 6 (`post_varNAC4`), Invariants (5.x-y) and (7-9.x-y) hold with $D(x, Q, c) = D(x)$ and $D(y, Q, c) = D(y)$. Lines 7–8 establish Invariants (10.x-y). The removal of $\triangle$ from $D(x)$ and $D(y)$ establishes Invariants (6.x-y) and does not affect the other Invariants. Any subsequent removal of values in $D(x)$ or in $D(y)$ by the post of other constraints does not change $D(x, Q, c)$ nor $D(y, Q, c)$. The invariants thus hold when the execution reaches line 21 for the first time.
(Induction step) Suppose the invariants hold at line 21 of AC5 and `valRemove`$(c, y, b, \triangle)$ is executed because (c, y, b) has been removed from the queue. Before the execution of `valRemove` Invariants (5-8.y), (8.x) and (10.x) hold and are not affected by this execution of `valRemove`. Invariants (5-7.x), (9,x-y) and (10.y) holds, but for $D(y, Q, c) \cup \{b\}$. Invariant (7.x) holds also for $D(y, Q, c)$ as the set F is unchanged. Lines 5-7 reestablish Invariants (9-10.y) and lines 8-13 reestablish Invariants (5.x) and (9.x). Since Invariant (6.x) holds for $D(y, Q, c) \cup \{b\}$, we have at Line 14 $\forall a \in D(x) : \textit{nbForbidden}[x, a, c] \leq \#D(y, Q, c)$. Invariant (6.x) is reestablished by the computation of $\triangle$ and the removal of these values from $D(x)$. Finally, the removal of $\triangle$ from $D(x)$ does not affect the set $D(x, Q, c')$ for all $c' \in C$, thanks to the enqueue operations. ∎

Proposition 3.2 *NAC4 is correct and its time and space complexity are $O(ed^2)$; i.e. its time complexity is optimal.*

Proof By Proposition 2.1, it is sufficient to prove the correctness of `postNAC4` and `valRemoveNAC4`, knowing that Invariants (5-10.x-y) hold at line 21 of AC5. The correctness of `postNAC4`$(x, \triangle)$ is straighforward.

At line 14 of `valRemove`, Invariants (5.x-y), (7-10,x-y) and (6.y) hold. Since *localSize*$[y, c]$ has been decremented as well as some *nbForbidden* counters, Invariant (6.x) becomes at Line 14

$$\forall a \in D(x) : a \in F[x, a, c] \Rightarrow \textit{nbForbidden}[x, a, c] < \#D(y, Q, c)$$

Hence, by (7-8.x)

$$\forall a \in D(x) : \textit{nbForbidden}[x, a, c] = \#D(y, Q, c) \Rightarrow c(x/a, y/b). \tag{3.1}$$

To prove the correctness of `valRemoveNAC4`$(c, y, b, \triangle)$, we need to show that at line 18

$$\triangle = \{(x, a) | a \in D(x) \wedge c(x/a, y/b) \wedge \forall b' \in D(y, Q, c) : \neg c(x/a, y/b')\}.$$

We have:

$$\begin{array}{l} a \in D(x) \wedge c(x/a, y/b) \wedge \forall b' \in D(y, Q, c) : \neg c(x/a, y/b')) \\ \text{iff } a \in D(x) \wedge c(x/a, y/b) \wedge \textit{nbForbidden}[x, a, c] = \#D(y, Q, c) \text{ (by (5.x))} \\ \text{iff } a \in D(x) \wedge \textit{nbForbidden}[x, a, c] = \#D(y, Q, c) \text{ (by (3.1)).} \end{array}$$

The construction of the set $\triangle$ in Lines 14-17 then fulfills the specification (by (9-10.x)).

The time complexity of postNAC4 is $O(d^2)$, assuming a time complexity of $O(d)$ of `getForbidden`. The time complexity of valRemoveNAC4 is $O(d)$. Hence by Proposition 2.2, the time complexity of NAC4 is (ed^2). The space complexity is obviously (ed^2). ■

4 Applications

As two instances of AC5, the AC4 and NAC4 algorithms can be combined, each constraint implementing its AC4 or NAC4 version of the `post` and `valRemove` methods. This combined algorithm, denoted PNAC4, is thus able to exploit for each constraint either its supports or its forbidden values.

We here review a variety of applications of PNAC4.

4.1 Sparse AC Constraints

A nice property of PNAC4 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 2.2.

Proposition 4.1 *If the* `getSupports` *or* `getForbidden` *function of each constraint of a binary CSP has a $O(K)$ time complexity, then the time and space complexity of PNAC4 is $O(edK)$.*

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

Definition 4.1 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 4.1 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 $(x + y) \bmod k = 0$, $x \bmod k = y \bmod k$ and $(x + y) \bmod k \neq 0$, $x \bmod k \neq y \bmod k$ which are sparse when k is $O(d)$.

Without modifying PNAC4, we can exploit the semantics of the constraints by using a specific `getSupports` method for positively sparse constraints and a specific `getForbidden` method for negatively sparse constraints with a $O(1)$ time complexity.

Proposition 4.2 *For positively and negatively sparse constraints with a $O(1)$* `getSupports` *or* `getForbidden`*, PNAC4 has a space and time complexity of $O(ed)$.*

4.2 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 PNAC4 with a complexity of $O(kd^2)$. The propagation step on this constraint to achieve domain consistency will then run in time $O(K)$, where K is the number of supports or forbidden values.

Example 4.2 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 4.1 depicts the rules to combine constraints and to compute a data structure S or F 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 (respectively F) data structure will be denoted by c_i^+ (respectively c_i^-). If the post method applies these rules on c, then the resulting algorithm achieves domain consistency. When comparing the posting of the combined constraint with the posting of its individual constraints, there is no time or space overhead as all the operations in Table 4.1 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 $O(kd)$.

Proposition 4.3 *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 4.1, then the resulting PNAC4 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 PNAC4 applied on $C \cup \{c\}$ is $O((e + k)d)$, with $e = \#C$.*

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

Table 4.1: Rules for Combining $c_1(x,y)$ and $c_2(x,y)$.

Notice that the above combination may also be useful when constraints effectively get the same (relevant) scope during search.

4.3 Validity and Entailment

The `isValid` and `isEntailed` methods (Specification 4) are useful for constraint combinators [19,1], reification [25,13] and in an *Ask & Tell* framework [24]. Such method can easily be implemented assuming a `getSupports` method of the constraints. A value (x,a) is valid for the constraint $c(x,y)$ if $\#\texttt{getSupports}(c,x,a) = \#D(y)$. A constraint is entailed if all the values of one of its variable are valid. The complexity of testing the entailment of a binary constraint is then $O(d^2)$.

PNAC4 can be extended in order to test entailment in $O(1)$. In AC4, a value (x,a) is detected to be valid in c if *nbSupport*$[x,a,c]$ 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 *nbForbidden*$[x,a,c] = 0$. The invariants of the data structures for both AC4 and NAC4 would then be (1-10.x-y). The theoretical complexity of PNAC4 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

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

for `post`, and

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

for `valRemove`.

Let us illustrate the use of entailment on reification. The idea of reification is to associate a {0,1} variable b to a constraint c as follows:

$$\mathit{reify}(b,c) \equiv b = 1 \Leftrightarrow c$$

The variable b can then be used in any arithmetic expression. Reification is often implicit, such as in the constraint $\sum(x_i = y) \leq k$. In an implementation of reification, such as described in [13], one needs the `isEntailed` method for the reified constraint c and its negation

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

Specification 4: The `isValid` and `isEntailed` Methods.

$\neg c$. When the domain of b is $\{0, 1\}$, `isEntailed(`c`)` allows to prune the value 0 from $D(b)$, while `isEntailed(`$\neg c$`)` allows to prune the value 1.

4.4 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 [1], where primitive constraints produce not only the inconsistent values but also the valid ones. Our extended PNAC4 can be used for combining constraints using the proposed algebra.

5 Specializing AC3 for Sparse Constraints

We showed that the PNAC4 algorithm has a $O(ed)$ space and time complexity for positively and negatively sparse constraints. We here present a variant of AC3, the classical constraint-based domain-consistency algorithm, that efficiently handles sparse constraints. The variant, denoted PNAC3, uses a `getSupports` or `getForbidden` function provided by the constraints. This variant is however useless for validity and entailment.

The well-known AC3 algorithm is given in Algorithm 5. It uses a queue of pairs (x, c) stating that the domain consistency of constraint c should be reconsidered for the domain $D(x)$. When a value is removed from the domain of a variable, the domain consistency of the constraint involving this variable should be reconsidered for all the other variables involved in the constraint. The corresponding pairs are then added to the queue.

The method `propagate(`$x, c, \triangle$`)` (also called *revise*) assigns to $\triangle$ the values in $D(x)$ inconsistent with the constraint c. In Algorithm 6 we propose two variants of the standard implementation of `propagate`. The first variant uses the method `getSupports` and the second variant uses the method `getForbidden` to check whether a value is inconsistent. It is clear that these two implementations meet their specification. We denote by PNAC3 the AC3 algorithm that combines the two variants of the propagate method. As PNAC4, this combined algorithm is able to exploit the semantics of the constraints by using a specific `getSupports` method for positively sparse constraints and a specific `getForbidden` method for negatively sparse constraints with a $O(1)$ time complexity. For this class of constraints, the time complexity of PNAC3 then reduces to $O(ed^2)$, instead of $O(ed^3)$ for AC3.

```
1  AC3(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    Q = {(x, c)|c ∈ C, x ∈ Vars(c)};
6    while (Q != ∅) {
7      select and remove (x, c) from Q;
8      propagate(c, x, △);
9      if (△\neq ∅) {
10       forall((x, a) in △) D(x) -= a;
11       Q = Q ∪ {(x′, c′)|c′ ∈ C, c ≠ c′, x ≠ x′, x, x′ ∈ Vars(c′)};
12     }
13   }
14 }
```

Algorithm 5: The AC3 Algorithm.

```
1  propagateAC3S(in c: Constraint, in x: Variable, out △: Set of Values) {
2  // Pre: c ∈ C with Vars(c) = {x, y}
3  // Post: △ = {(x, a)|a ∈ D(x) ∧ ∀b ∈ D(y) : ¬c(a, b)}
4    △ = ∅;
5    forall(a in D(x))
6      if (getSupports(c, x, a) == ∅)  △ += (x, a) ;
7  }
8  propagateAC3F(in c: Constraint, in x: Variable, out △: Set of Values) {
9  // Pre: c ∈ C with Vars(c) = {x, y}
10 // Post: △ = {(x, a)|a ∈ D(x) ∧ ∀b ∈ D(y) : ¬c(a, b)}
11   △ = ∅;
12   forall(a in D(x))
13     if (# getForbidden(c, x, a) == #D(y))  △ += (x, a) ;
14 }
```

Algorithm 6: The `propagateAC3S` and `propagateAC3F` variants

Proposition 5.1 *For positively and negatively sparse constraints with a $O(1)$* `getSupports` *or* `getForbidden`*, PNAC3 has a time complexity of $O(ed^2)$.*

The PNAC3 algorithm could also be used to combine sparse constraints on the same variables, using the rules in 4.1. However, if the constraints are not sparse, PNAC3 is worse than AC3.

Proposition 5.2 *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 getSupports/getforbidden functions for c applies the rules of Table 4.1, then the resulting PNAC3 algorithm on $C \cup \{c\}$ achieves domain consistency. If the time complexity of the getSupports or getForbidden in $C \cup \{c_1, \ldots, c_k\}$ is $O(1)$, then the time and space complexity of PNAC3 applied on $C \cup \{c\}$ is $O((e + k)d^2)$, with $e = \#C$.*

Although it is possible to use a similar approach to specialize AC3rm and AC2001 to sparse constraints, the resulting algorithms are uninteresting as AC3rm and AC2001 cannot easily handle negative constraints. They are more complex than PNAC3 and are always less efficient by at least a factor of 2.

6 Experimental Results

This section illustrates the benefits of jointly exploiting supports and forbidden values with PNAC4 and PNAC3. We consider the positively and negatively sparse constraints $x = y \bmod k$, $x = |y - k|$, $x + y = k$, $|x - y| = k$, $(x + y) \bmod k = 0$, $x \bmod k = y \bmod k$ and their negative version, where k is a constant. For each constraint, we will consider the default $O(d)$ `getSupports` or `getForbidden` functions, as well as their $O(1)$ implementation exploiting the semantics of the constraints. They are straighforward to realize, as illustrated in Algorithm 7 for the constraint $|x-y| \neq k$. A specialized implementation of a `getSupports` or `getForbidden` function is easier to realize than a specialized propagator. For comparison, a specific propagator for the constraint $|x - y| \neq k$ is shown in Algorithm 8. Students often missed the second part of this propagator.

Hard problem instances involving the above constraints were built. Although these instances are not real problems, they involve realistic constraints used in constraint programming. We believe that solving hard CSPs with only sparse constraints is a realistic way of comparing the consistency algorithms. The order in which values are removed from the domains or the number of removals in the domain during the search are not specific in the generated benchmarks. Adding non sparse constraints would not provide any information on the comparison of PNAC4 and PNAC3 and with other consistency algorithms on sparse constraints. We show that PNAC3 is more efficient than PNAC4 and outperforms other AC algorithms on theses instances. We can therefore conclude that in a real CSP involving some sparse constraints, it will be beneficial to replace, for the sparse constraints, the consistency algorithm with PNAC3 or PNAC4. Since modern constraint-programming systems make it possible to combine various consistency algorithms within the same framework, PNAC3 and PNAC4 provides a direct way to speed up constraint programming for this class of constraints.

6.1 Implementation

PNAC4, PNAC3 and other state of the art consistency algorithms have been implemented on top of the Comet constraint programming language [11]. Comet uses an event-based mechanism to handle propagation. It offers a constraint-based propagation, used in AC3-like algorithms, and a value-based propagation, used in AC5-like algorithms. The implementation of the proposed algorithms in Comet only required the implementation of the `post` and `valRemove` methods. The event mechanism executes these methods when appropriate. One should however directly remove the elements from the domains. As usual when integrating consistency algorithms in a MAC search procedure, posting the constraints is done at the root of the search tree. The data structures, such as *nbSupport*, *nbForbidden* and *setOfSize* must be backtrackable. Comet offers trailable data structures that are maintained by the system. This considerably simplifies the implementation of propagators with auxiliary data structures. The use of backtrackable data structures does not influence the space complexity of the algorithms. Note that in AC4 and NAC4, the data structures S and F are not modified during the execution and are thus not backtracked.

```
1  getForbidden(in c: Constraint, in x: Variable; in a: Value) Set of Values {
2  // Pre: c ≡ |x − y| ≠ k
3  // Post: return {b ∈ D(y)||a − b| = k}
4    F = ∅;
5    if ((a − k) ∈ D(y))   F += (a − k);
6    if ((a + k) ∈ D(y))   F += (a + k);
7  }
```

Algorithm 7: The `getForbidden` function for the constraint $|x - y| \neq k$

```
1  propagate_Dist_Spec(in c: Constraint, in x: Variable, out △: Set of Values){
2  // Pre: c ≡ |x − y| ≠ k
3  // Post: △ = {(x, a)|a ∈ D(x) ∧ ∀b ∈ D(y) : ¬c(a, b)}
4    △ = ∅;
5    if (D(y) == {b}) {
6      if ((b + k) ∈ D(x))   △ += (x, b + k);
7      if ((b − k) ∈ D(x))   △ += (x, b − k);
8    }
9    if (D(y) == {b1, b2}) {
10     if (abs(b2 − b1)==2 ∗ k && (b1 + b2)div 2 ∈ D(x))   △ += (x, (b1 + b2)div 2);
11   }
12 }
```

Algorithm 8: A specific `propagate` method the constraint $|x - y| \neq k$

6.2 Benchmarks

When comparing PNAC4 and PNAC3 with other consistency algorithms, it is important to generate hard instances, avoiding trivially satisfiable or unsatisfiable instances, as well as instances solvable without search. These pathological cases do not correspond to real problems tackled by constraint programming. The generation of hard instances with sparse constraints is not necessarily a trivial task as positively sparse constraints are very restrictive while negatively sparse constraints are very permissive. The next paragraph shows a simple way to do so.

The instances were randomly generated with the parameters n (the number of variables), d (the size of the domain), p (the percentage of positive constraints between variables), and q (the percentage of negative constraints between variables). First, variables are partitioned into as many clusters as possible, each cluster having at least 7 variables. Second, inside each cluster, a tree is randomly built with random positive constraints. Next, constraints are randomly added within each cluster in order to meet the required percentage p and q. If constraints with the same scope are generated, these are kept as separated constraints. Finally, the clusters are randomly connected with negative constraints to form a tree between the clusters. In the random generation of a constraint, the parameter k is always chosen to satisfy a given random initial solution, ensuring the consistency of the CSP. The tree structure with positive constraints within each cluster ensures a high coupling between the variables of a cluster. The additional constraints within the cluster will induce propagation. The tree structure between the clusters ensures a loose coupling between the clusters, inducing a large search tree.

We generated benchmarks with $n = 30$ variables, a domain size $d \in \{500, 700, \ldots, 2100\}$, $p = 40\%$ positive constraints, and a percentage of negative constraints $q \in \{0, 10, 20, 30, 40\}$. We also generated larger benchmarks with $n \in \{30, 50, 100, 150\}$, $d = 1500$, $p = 40\%$ and $q = 30\%$. For 100 variables, we thus have 1980 positive constraints and 1485 negative constraints. We generated 200 instances per test, that is a total of 9,600 instances. Each CSP has been solved in Comet using different consistency algorithms: (1) AC4 for each constraint, (2) NAC4 for each constraint, (3) the combination PNAC4 using AC4 for positive constraints and NAC4 for negative constraints and (4) the combination PNAC4* which is similar to PNAC4 but uses specialized `getSupports` or `getForbidden` functions of the constraints exploiting their semantics. For sparse constraints, the specialized `getSupports` or `getForbidden` functions have a complexity of $O(1)$, while they have a complexity of $O(d)$ when using the generic implementation of these functions in PNAC4. We then consider (5) PNAC3*, which is PNAC3 using the $O(1)$ specialized `getSupports` or `getForbidden` functions. We do not consider PNAC3 using the $O(d)$ generic implementation of these functions as this is obviously less efficient than the standard AC3 as it does not stop as soon as a support is found. We also reimplemented state of the art consistency algorithms (6) AC3, (7) AC3$_{rm}$ [16, 17], (8) AC2001, and (9) AC6. To complete the comparison, we also implemented (10) a set of dedicated consistency algorithms, one per constraint type and denoted by Spec. All the reported execution times are expressed in seconds, and are the average of the 200 instances. The %best is computed on the total time. We also report the post time and the search time. The post time includes all the induced propagations when posting the constraints, that is the time to achieve domain consistency at the root of the search tree.

When appropriate, the number of constraint checks and the number of calls to the propagation methods are reported. The experiments were performed on a single core of a machine with an Intel Xeon 2.53GHz with 23GB memory, shared by 8 cores.

6.3 PNAC4 vs AC4 and NAC4

	AC4	**NAC4**	**PNAC4**	**PNAC4***
Total time (sec.)	27.36	69.93	2.47	1.12
Post time	13.04	21.63	1.73	0.38
Search time	14.32	48.3	0.74	0.74
% best	2 445%	6 252%	221%	100%

Table 6.1: Comparing AC4, NAC4 and PNAC4 ($n = 30$, $d = 500$, $p = 40\%$, $q = 30\%$, 200 instances, timeout of 180 sec.). NAC4 timeouts on 26 instances.

Our first experiments compare PNAC4 and PNAC4* with AC4 and NAC4 (Table 6.1). As expected, PNAC4 and PNAC4* are much more efficient than AC4 and NAC4. Maintaining a data structure counting the supports in AC4 (respectively forbidden values in NAC4) is very heavy for negatively (respectively positively) sparse constraints. It also illustrates the interest of specialized `getSupports` or `getForbidden` functions of the constraints, as used in PNAC4*, reducing the execution time by a factor of 2. In these benchmarks, the domain size of a variable after posting all the constraints reduces from 500 elements to an average of 107 elements.

6.4 PNAC3 vs PNAC4 vs State of the Art

	AC3	**AC3rm**	**AC2001**	**AC6**	**PNAC4**	**PNAC4***	**PNAC3***	**SPEC**
Total time (sec.)	37.59	17.51	32.31	40.39	21.38	9.22	1.25	0.97
Post time	22.16	6.88	10.22	9.97	13.24	1.15	0.33	0.26
Search time	15.43	10.63	22.09	30.42	8.14	8.07	0.92	0.71
% best	3872%	1804%	3328%	4161%	2202%	949%	128%	100%
Nb call propag.	5.5 k	5.5 k	5.5 k	2.4 M	2.6 M	2.6 M	5.5 k	5.5 k
Nb constr. checks	180 M	48 M	63 M	58 M	112 M	186 k	4.7 M	3.9 M

Table 6.2: Comparing PNAC4 and PNAC3 with state of the art algorithms ($n = 30$, $d = 1500$, $p = 40\%$, $q = 30\%$, 200 instances, 1k=10^3 and 1M=10^6).

Our next experiments compare PNAC4, PNAC4* and PNAC3* with the state of the art AC3, AC3$_{rm}$, AC2001 and AC6 standard consistency algorithms. We here considered harder instances (with $d = 1500$). The results are reported in Table 6.2.

In these experiments, AC3$_{rm}$ gives slightly better results than PNAC4. AC3$_{rm}$ is a variant of AC3 that keeps the last visited support of a value. When checking if a value is supported, AC3$_{rm}$ needs to find a support only if the last visited support is not in the domain anymore. This reduces the number of constraint checks from 180×10^6 (AC3) to 48×10^6 (AC3$_{rm}$). However, when the constraints offer a linear `getSupports` or `getForbidden` functions, what is straighforward for the considered constraints, we then obtain PNAC4* which is twice as fast as AC3$_{rm}$. The optimal AC2001 is better than AC3, but does not compete with PNAC4*. The cost of maintaining the first support of a value is too high in AC2001 compared to AC3. Similarly, the cost of maintaining data structures in the optimal AC6 is also too high. Exploiting a linear `getSupports` or `getForbidden` functions for the constraints in PNAC3* yields a very efficient algorithm.

These benchmarks also underlines the differences between a constraint based approach (AC3, AC3$_{rm}$, AC2001 and PNAC3*) and a value-based approach (AC6, PNAC4 and PNAC4*) in the number of calls to the propagate methods (called valRemove in the generic AC5 algorithm). A value-based approach has about 500 times more calls, but this is balanced by the efficiency of the valRemove method that only concentrates on the impact of a single removed value. In these benchmarks, the domain size of a variable after posting all the constraints reduces from 1500 elements to an average of 322 elements. Exploiting the semantics of the constraints allows us to reduce the number of constraint checks. PNAC4* performs however about 25 times less constraint checks than PNAC3* as PNAC4* stores the supports or forbidden values. This advantage of PNAC4* could have an impact for constraints with a high cost constraint check.

Finally, we compare the consistency algorithms with Spec, a set of dedicated consistency algorithms, one per constraint type. If a dedicated algorithm for $x + y \neq k$ is straighforward, it is more difficult for a constraint such as $x \neq y \bmod k$. Spec is more efficient than PNAC3*, but with only a factor of 1.3. The programming effort to obtain PNAC3* is very small (implementing the `getSupports` or `getForbidden` function) compared to the writing of specialized consistency algorithms, from scratch, exploiting the semantics of the constraints.

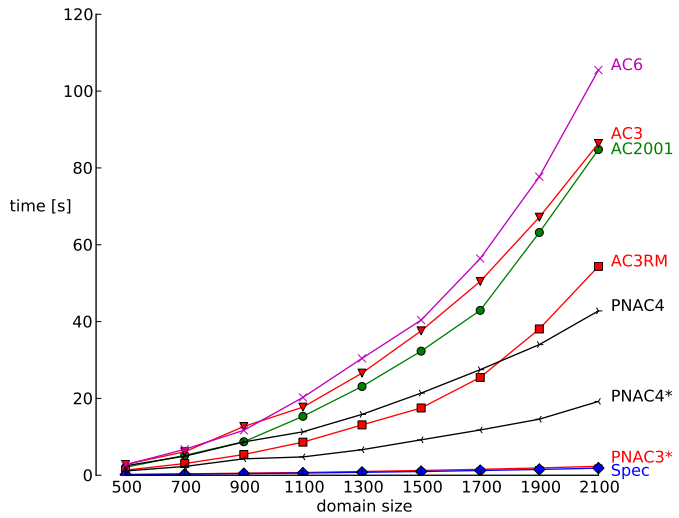


Fig. 6.1: Influence of the domain size ($n = 30$, $d \in \{500, 700, \ldots, 2100\}$, $p = 40\%$, $q = 30\%$, 200 instances).

Influence of the domain size. Figure 6.1 shows the results for different domain sizes, ranging from 500 to 2,100 elements. It illustrates that the advantages of PNAC3* and PNAC4* over state of the art consistency algorithms increases with the size of the domains. It also shows that for larger domains, PNAC4 is better that AC3$_{rm}$. The relative speed ups between PNAC4* and PNAC4, between PNAC4* and PNAC3* and between Spec and PNAC3* are stable.

Influence of the number of variables Figures 6.2 show the results when the number of variables is increased. PNAC4 and AC3$_{rm}$ have a similar performance, as well as AC3 and AC2001. The relative speed ups between PNAC4* and PNAC4 and between Spec and PNAC3* are stable.

Influence of the percentage of negative constraints Figure 6.3 shows the results for different percentages of negatively sparse constraints. When the percentage of negative constraints is low, PNAC4 is better than AC3$_{rm}$, but this is inverted when the percentage increases. PNAC4* is always better than AC3$_{rm}$, and PNAC3* is always better than PNAC4*. PNAC3* and Spec are very similar.

Pairwise comparison Figure 6.4 presents different pairwise comparisons of consistency algorithms, based on the results presented in Table 6.2. In these graphs, the coordinates of a point are the execution times of an instance on the two considered algorithms. The execution times of the 200 instances are reported. A large majority of points on one side of the diagonal illustrates the superiority of the algorithm on the other.

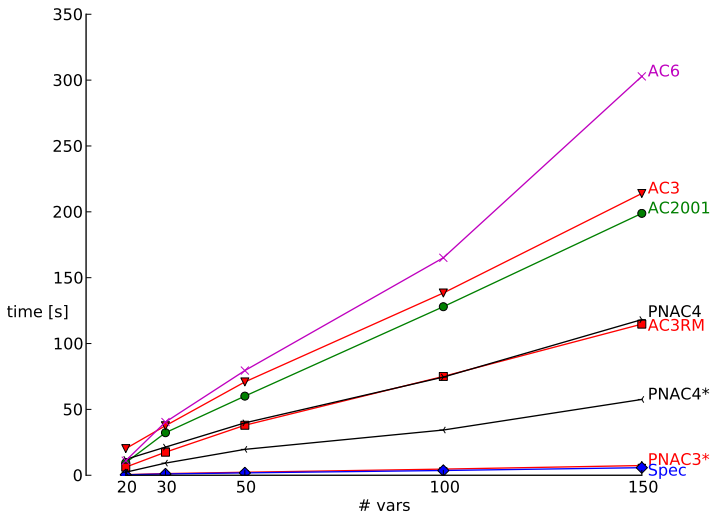


Fig. 6.2: Influence of the number of variables ($n \in \{20, 30, 50, 100, 150\}$, $d = 1500$, $p = 40\%$, $q = 30\%$, 200 instances).

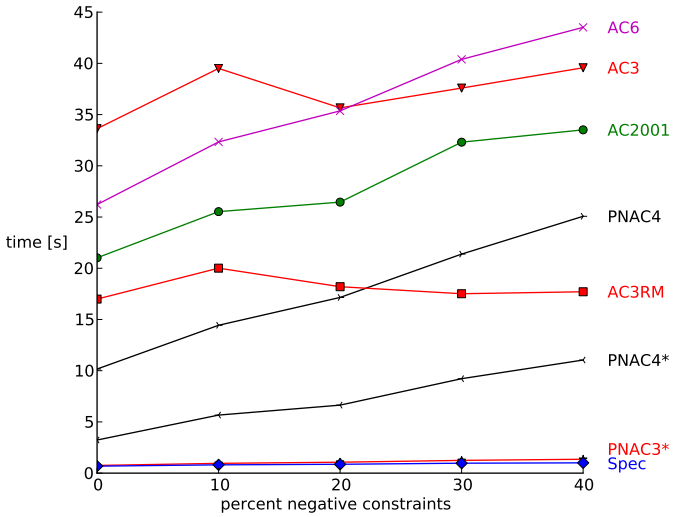


Fig. 6.3: Influence of the percentage of negative constraints ($n = 30$, $d = 1500$, $p = 40\%$, $q \in \{0, 10, 20, 30, 40\}$, 200 instances).

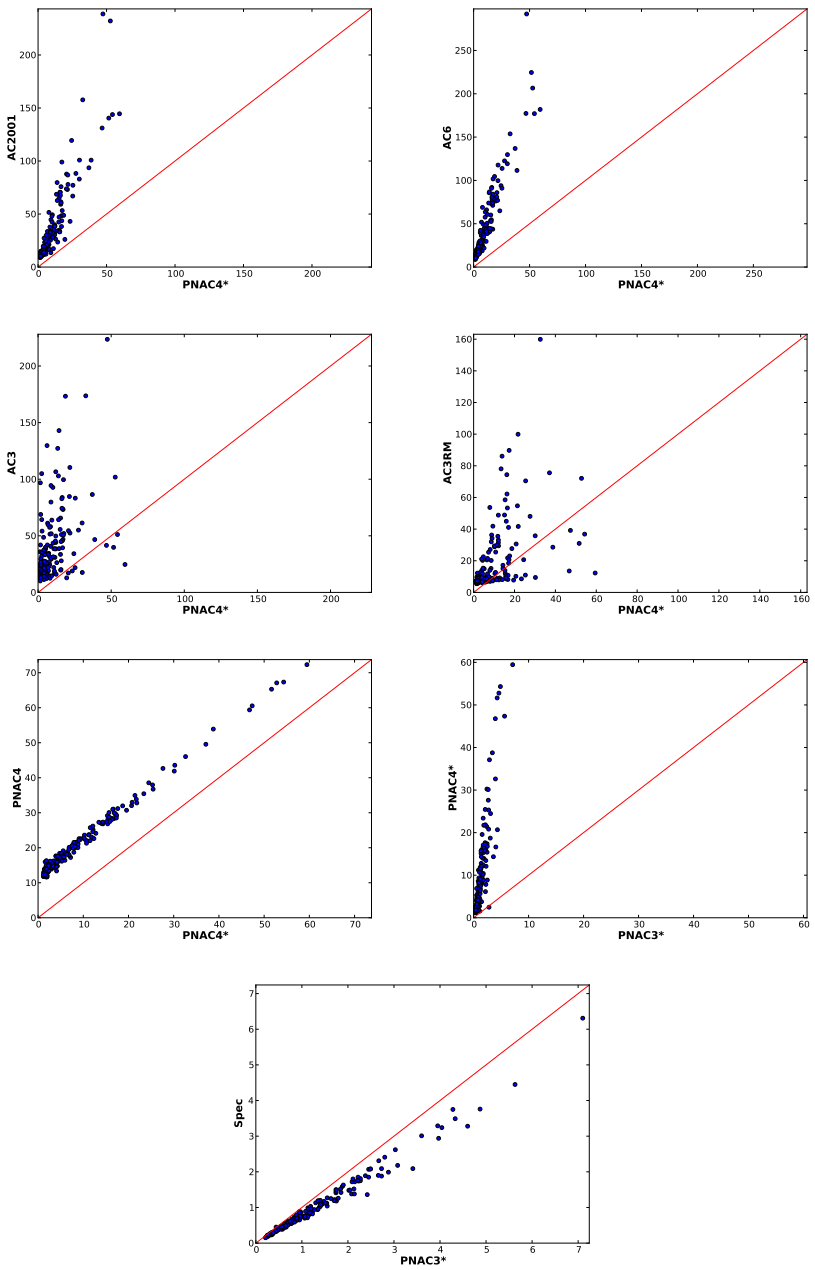


Fig. 6.4: Execution time of each instance for two different algorithms ($n = 30$, $d = 1500$, $p = 40\%$, $q = 30$, 200 instances).

We clearly see that PNAC4* is always better than AC2001 and AC6. PNAC4* is better than AC3 on almost all instances (95.5%), and is better than AC3$_{rm}$ on a large majority of instances (85%). The simplicity of AC3 makes it better on some instances. The difference between PNAC4* and PNAC4 is close to a constant value on all the instances. This constant value comes from the linear `getSupports` or `getForbidden` functions used only once in the post method. It is also clear that PNAC3* is always much better than PNAC4*. The (slight) improvement of Spec compared to PNAC3* is close to the same factor on each instance. Finally, we note that the variance of the execution times of PNAC4* is much smaller than of AC2001, AC6, AC3 and AC3$_{rm}$. We have the same observation between PNAC3* and PNAC4*.

Summary PNAC3* and PNAC4* only requires the addition of a linear `getSupports` or `getForbidden` function to the constraints. This is a simple task that does not require any understanding of the consistency algorithms. On sparse constraints, PNAC3* and PNAC4* are better that all state of the art consistency algorithms; PNAC3* outperforms PNAC4* and is close to the efficiency of specialized consistency algorithms. The advantage of PNAC3* and PNAC4* is confirmed when increasing the size of the domains as well as the number of variables. PNAC3* and PNAC4* also offer a smaller variance of the execution time.

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 NAC4 algorithm achieves 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 $O(d)$. As AC4 and NAC4 are two instances of the generic AC5 consistency algorithm, they can be combined in a single PNAC4 algorithm. The paper showed how forbidden values and supports can be used jointly to achieve domain consistency on logical combinations of constraints and to efficiently compute validity and entailment of constraints. The PNAC4 algorithm can exploit the constraint semantics of a large class of constraints, called sparse constraints, to obtain higher efficiency. We also presented a variant of AC3, denoted PNAC3; that uses a `getSupports` or `getForbidden` function provided by the constraints. It is especially efficient on sparse constraints, as a $O(1)$ `getSupports` or `getForbidden` function can easily be written. The PNAC3 approach is however useless for the validity and entailment of constraints.

Experimental results showed that on the class of sparse constraints with large domains, PNAC4 is competitive with the best state of the art consistency algorithms, even without exploiting the semantics of the constraints. When exploiting the semantics of the constraints through a specialized $O(1)$ `getSupports` or `getForbidden` function, PNAC4 is twice more efficient than the best state of the art algorithm. When using these specialized functions, PNAC3 is much more efficient than the other algorithms. It is almost as efficient than dedicated consistency algorithms for each constraints. This suggests that PNAC3 and PNAC4 could be used for the propagation of such constraints in CSPs modeling real problems.

Future work includes the combination of several consistency algorithms, based on the semantics of constraints, extension and experimentations on non-binary constraints and extension of NAC4 to handle negative tables represented in a compact way such as in [12, 14, 8].

Acknowledgments

The authors want to thank the anonymous reviewers for their valuable comments. The third author is supported as a Research Assistant by the Belgian FNRS (National Fund for Scientific Research). 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). NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Bacchus, F., Walsh, T.: Propagating logical combinations of constraints. In: IJCAI, pp. 35–40 (2005)
2. Bessière, C.: Arc-consistency and arc-consistency again. Artif. Intell. **65**(1), 179–190 (1994)
3. Bessière, C.: Constraint propagation. In: F. Rossi, P.v. Beek, T. Walsh (eds.) Handbook of Constraint Programming. Elsevier Science Inc., New York, NY, USA (2006)
4. Bessière, C., Régin, J.C.: Arc consistency for general constraint networks: Preliminary results. In: IJCAI, pp. 398–404 (1997)
5. Bessière, C., Régin, J.C.: Local consistency on conjunctions of constraints. In: ECAI-98, proceedings Workshop on Non Binary Constraints, pp. 53–60 (1998)
6. Bessière, C., Régin, J.C., Yap, R.H.C., Zhang, Y.: An optimal coarse-grained arc consistency algorithm. Artif. Intell. **165**(2), 165–185 (2005)
7. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Support inference for generic filtering. In: CP, pp. 721–725 (2004)
8. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on ad hoc r-ary constraints. In: CP'08, pp. 509–523. Springer-Verlag (2008)
9. Cheng, K.C.K., Yap, R.H.C.: An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints **15**(2), 149–204 (2010)
10. Deville, Y., Van Hentenryck, P.: Domain consistency with forbidden values. In: D. Cohen (ed.) Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings, *Lecture Notes in Computer Science*, vol. 6308, pp. 191–205. Springer (2010)
11. Dynadec: Comet (2011). Www.dynadec.com
12. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: AAAI'07, pp. 191–197 (2007)
13. Jefferson, C., Moore, N.C.A., Nightingale, P., Petrie, K.E.: Implementing logical connectives in constraint programming. Artif. Intell. **174**, 1407–1429 (2010). DOI http://dx.doi.org/10.1016/j.artint.2010.07.001. URL http://dx.doi.org/10.1016/j.artint.2010.07.001
14. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: CP'07, pp. 379–393. Springer-Verlag (2007)
15. Lecoutre, C.: Constraint Networks: Techniques and Algorithms. ISTE/Wiley (2009)
16. Lecoutre, C., Hemery, F.: A study of residual supports in arc consistency. In: Proceedings of the 20th international joint conference on Artifical intelligence, pp. 125–130. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007). URL http://dl.acm.org/citation.cfm?id=1625275.1625294
17. Lecoutre, C., Likitvivatanavong, C., Shannon, S., Yap, R., Zhang, Y.: Maintaining arc consistency with multiple residues. Constraint Programming Letters(CPL) **2**, 3–19 (2008)
18. Lecoutre, C., Szymanek, R.: Generalized arc consistency for positive table constraints. In: Proceedings of CP 06, pp. 284–298 (2006)
19. Lhomme, O.: Arc-consistency filtering algorithms for logical combinations of constraints. In: CPAIOR, pp. 209–224 (2004)
20. Mackworth, A.K.: Consistency in networks of relations. Artif. Intell. **8**(1), 99–118 (1977)
21. Mehta, D., van Dongen, M.R.C.: Reducing checks and revisions in coarse-grained MAC algorithms. In: IJCAI, pp. 236–241 (2005)
22. Mohr, R., Henderson, T.: Arc and Path Consistency Revisited. Artificial Intelligence **28**, 225–233 (1986)
23. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artif. Intell. **28**(2), 225–233 (1986)

24. Saraswat, V.: Concurrent Constraint Programming. MIT Press, Cambridge, Mass. (1993)
25. Schulte, C.: Programming deep concurrent constraint combinators. In: Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages, PADL '00, pp. 215–229. Springer-Verlag, London, UK (2000). URL http://dl.acm.org/citation.cfm?id=645770.667779
26. Van Hentenryck, P., Deville, Y.: The cardinality operator: A new logical connective for constraint logic programming. In: ICLP, pp. 745–759 (1991)
27. Van Hentenryck, P., Deville, Y., Teng, C.M.: A generic arc-consistency algorithm and its specializations. Artif. Intell. **57**(2-3), 291–321 (1992)
28. Van Hentenryck, P., Saraswat, V.A., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(fd). In: A. Podelski (ed.) Constraint Programming: Basics and Trends, pp. 293–316. Springer (1994)