Domaine Consistance et Valeurs Interdites

Yves Deville[1] **Pascal Van Hentenryck**[2] **Jean-Baptiste Mairy**[1]

[1] ICTEAM, UCLouvain, Belgique
[2] Brown University, USA
{Yves.Deville,Jean-Baptiste.Mairy}@uclouvain.be pvh@cs.brown.edu

Abstract

Cet article [1] présente un nouvel algorithme de domaine consistance qui ne maintient pas les supports dynamiquement lors de la propagation, mais plutôt maintient les valeurs interdites. Nous introduisons l'algorithme optimal NAC4 (AC4 négatif), basé sur cette idée. Nous montrons en outre que le maintien de valeurs interdites permet à l'algorithme générique AC5 d'assurer la domaine consistance en temps $O(ed)$ pour les catégories de contraintes dans lequel le nombre de supports est de $O(d^2)$, mais où le nombre de valeurs interdites est de $O(d)$. Le papier montre également comment les valeurs interdites et les supports peuvent être utilisés conjointement pour assurer la domaine consistance sur des combinaisons logiques de contraintes ainsi que pour calculer la validité et l'implication de contraintes. Les résultats expérimentaux montrent les avantages de l'exploitation communes des supports et des valeurs interdites.

1 Introduction

En programmation par contraintes, la propagation vise à réduire l'espace de recherche sans retirer de solutions. L'algorithme de propagation considère chaque contrainte individuellement et se termine lorsque aucune contrainte ne peut être utilisée pour réduire les domaines des variables. La propagation idéale pour une contrainte est la *consistance de domaine*, aussi connu sous la consistance d'arc. Elle supprime du domaine de chaque variable toutes les valeurs qui n'appartiennent pas à une solution de la contrainte considérée. De nombreux algorithmes ont été proposés pour assurer la consistance de domaine, tels que AC3, AC4, AC6, AC7 et AC2001 (voir [2]). Les algorithmes de consistance utilisent généralement la notion de *support*. Pour une contrainte binaire $c(x, y)$, un support pour une paire (x, a), où a est une valeur possible pour x, est une paire (y, b) tel que $c(a, b)$ est vraie. La complexité temporelle optimale pour assurer la consistance de domaine pour un CSP est de $O(ed^2)$ pour des contraintes binaires et $O(e.r.d^r)$ pour des contraintes non binaires (d est la taille du plus grand domaine, e le nombre de contraintes et r la plus grande arité des contraintes). Un algorithme tel que AC4 maintient tous les supports pour tous les couples (x, a), alors que d'autres algorithmes (par exemple, AC6) maintiennent un support unique et cherchent les supports suivants à la demande. L'algorithme AC4 fonctionne en deux étapes. Tout d'abord, il calcule tous les supports pour les paires variable/valeur de chaque contrainte. Puis, il propage la suppression d'une valeur a du domaine d'une variable x. Une propriété intéressante de l'étape de propagation est que sa complexité temporelle est proportionnel au nombre total de supports.

Example 1 *Considérons la contrainte* $x = y \bmod 10$, *avec* $D(x) = \{0..9\}$ *et* $D(y) = \{0..99\}$. *La taille de l'ensmeble des supports pour* x *est linéaire (en* $O(\#D(y))$*, où* $\#A$ *est la taille de* A*). L'étape de propagation deAC4 pour cette contrainte est aussi linéaire, alors qu'elle reste quadratique pour d'autres algorithmes AC optimaux, tels que AC6, AC7, ou AC2001.*

L'étape d'initialisation de AC4 calculant les supports est de complexité $O(d^2)$, même si le nombre de supports est de $O(d)$, car l'algorithme n'a pas connaissance de la sémantique de la contrainte. L'algorithme générique AC5 [16] a été conçu pour exploiter la sémantique des contraintes qui peut ensuite être utilisée pour produire les supports de ces contraintes en temps linéaire. Ceci permet d'obtenir un algorithme de consistance de domaine en $O(ed)$.

1. Cet article est une version française de [8].

La question scientifique abordée dans cet article est la suivante. *Est-il possible de concevoir un algorithme de domaine consistance de complexité temporelle $O(ed)$ lorsque le nombre de supports est quadratique, mais le nombre de valeurs interdites (également appelé ensemble des conflits) est linéaire ?*

Example 2 *Considérons la contrainte $x \neq y \bmod 10$, avec $D(x) = \{0..9\}$ et $D(y) = \{0..99\}$. La taille de l'ensemble des supports pour x est de 900, d'où une complexité $O(\#D(x).\#D(y))$ dans l'étape de propagation de AC4. L'utilisation de AC3, AC7 ou AC2001 ne contribue pas à réduire cette complexité. Toutefois, la taille de valeurs interdites est de 100 ($O(\#D(y))$). Peut-on utiliser un algorithme de type AC4 pour maintenir la liste des valeurs interdites à la place des supports pour obtenir un algorithme de complexity $O(ed)$?*

Cet article répond à cette question de façon positive et apporte les contributions suivantes.

- Il propose l'algorithme NAC4 (AC4 négatif) qui réalise la consistance de domaine avce une complexity optimale de (ed^2) pour les CSP binaires. Cet algorithme maintient dynamiquement l'ensemble des valeurs interdites au lieu de maintenir les supports. Nous montrons que AC4 et NAC4 sont deux instances de l'algorithme AC5 générique. Ils peuvent être combinés naturellement dans un solveur de contraintes. AC5 peut exploiter la sémantique des contraintes pour obtenir une plus grande efficacité.
- Il identifie les catégories de contraintes pour lesquelles la consistance de domaine peut être obtenue en un temps linéaire.
- Il montre comment la combinaison AC4/NAC4 peut assurer la consistance de domaine sur des combinaisons logiques de contraintes portant sur les mêmes variables.
- Il montre que la combinaison de AC4/NAC4 peut être facilement étendue afin de permettre l'évaluation de la validité et de l'implication d'une contrainte.
- Il présente des résultats expérimentaux montrant les avantages de la combinaison AC4/NAC4.

Travaux reliés L'idée d'utiliser des valeurs interdites n'est pas nouvelle en CP. La nouveauté de ce papier est que NAC4 maintient l'ensemble des valeurs interdites dynamiquement lors de la propagation. Dans [3, 12, 7], les auteurs utilisent des contraintes tables négatives, où le tableau décrit l'ensemble de tuples interdits. La table négative est utilisée pour trouver le prochain support d'une variable/valeur au moyen d'une recherche binaire. La table est statique, et n'est pas mise à jour lors de la propagation. Lecoutre [5] a montré que (x, a) a un support pour une contrainte $c(x, y)$ si la taille de $D(y)$ est strictement supérieure à la taille de l'ensemble initial des conflits de (x, a). Cette idée est intégrée dans un algorithme de propagation. Une fois de plus, la taille de l'ensemble des conflits n'est pas mise à jour pendant le calcul. La même idée est également proposée comme condition de support dans [14].

La consistance d'un ensemble de contraintes a été traitée de différentes manières. Certaines approches assurent la consistance de domaine, qui est NP-difficile en général. Un algorithme de domaine consistance, fondée sur AC7, a été proposée dans [4] pour la conjonction de contraintes. Lhomme [13] décrit un algorithme de domaine consistance pour toute combinaison de contraintes. Il se concentre principalement sur des contraintes données en extension. Les tuples interdits sont utilisés ici aussi à travers une table négative statique. D'autres approches calculent une approximation de consistance de domaine, comme dans [15] (cardinalité), [17] (disjonction constructive), ou [1] qui propose une algèbre pour combiner des contraintes.

2 AC5

Cette section revisite l'algorithme générique AC5 [16] en le généralisant légèrement afin de couvrir AC4, AC6 et AC2001.

Definition 1 (CSP) *Un CSP binaire $(X, D(X), C)$ est composé d'un ensmeble de n* variables *$X = \{x_1, \ldots, x_n\}$, un ensmeble de* domaines *$D(X) = \{D(x_1), \ldots, D(x_n)\}$ où $D(x)$ est l'ensemble des* valeurs *possibles pour la variable x et un ensemble de* constraintes *binaires $C = \{c_1, \ldots, c_e\}$, avec $Vars(c_i) \subseteq X$ $(1 \leq i \leq e)$. On note $d = max_{1 \leq i \leq n}(\#D(x))$.*

Soit c une contrainte avec $Vars(c) = \{x, y\}$, $a \in D(x), b \in D(y)$. $c(x/a, y/b)$ ou $c(y/b, x/a)$ désigne la contrainte où les variables x et y ont été remplacées par les valeurs a et b. Nous supposons que tester $c(x/a, y/b)$ prend un temps $O(1)$. Si $c(x/a, y/b)$ est vrai, alors $(x/a, y/b)$ est appelé un support de c et (y, b) est un support de (x, a) pour c. Si $c(x/a, y/b)$ est faux, alors $(x/a, y/b)$ est appelé un conflit de c et (y, b) est un conflit (ou valeur interdite) de (x, a) pour c.

Definition 2 *Soit c une contrainte avec $Vars(c) = \{x, y\}$. L'ensemble des valeurs inconsistantes, consistantes et valides pour x dans c par rapport à un en-*

semble de valeurs B sont définis comme suit.

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

Nous utilisons $Inc(c,x)$ pour denoter $Inc(c,x,D(y))$, et de même pour les autres ensembles.

Definition 3 (Domaine Consistance) *Une contrainte c sur $\{x,y\}$ est* domaine consistante *par rapport à $D(X)$ ssi $Inc(c,x) = \emptyset$ et $Inc(c,y) = \emptyset$. Un CSP $(X, D(X), C)$ est* domaine consistant *ssi toutes ses contraintes sont domaine consistantes par rapport à $D(X)$.*

La Spécification 1 décrit les méthodes principales utilisées par AC5. L'algorithme utilise une file Q de triplets (c,x,a) indiquant que la domaine consistance de la contrainte c devra être reconsidérée car la valeur a a été retirée de $D(x)$. Quand une valeur d'un domaine est supprimée, la méthode `enqueue` met les informations nécessaires dans la file. Dans la postcondition, Q_0 représente la valeur de Q lors de l'appel. Le paramètre $C1$ nous permet de considérer un sous-ensemble de contraintes, nécessaire à l'initialisation. Tant que la file d'attente contient un triplet (c,x,a), il est souhaitable que l'algorithme puisse considérer que, selon la perspective de la contraine c, la valeur a est toujours dans $D(x)$. Ceci est formalisé ci-après.

Definition 4 *La vue locale d'un domaine $D(x)$ par rapport à une file Q pour une contrainte c est l'ensemble $D(x,Q,C) = D(x) \cup \{a | (c,x,a) \in Q\}$.*

Example 3 *Étant donné une file $Q = \{(c_1,y,2),(c_1,z,2),(c_2,y,3)\}$ et domaines $D(x) = \{1,2\}$, $D(y) = D(z) = \{1\}$, on a $D(x,Q,c_1) = D(y,Q,c_1) = D(z,Q,c_1) = \{1,2\}$.*

La méthode principale de l'algorithme AC5 est `valRemove`, où l'ensemble $\triangle$ (appelé ensemble delta dans le folklore de CP en raison de l'utilisation de la lettre $\triangle$ dans la description originale d'AC5) est l'ensemble des valeurs qui ne sont plus prises en charge en raison de la suppression de la valeur b dans $D(y)$. Dans cette spécification, b est une valeur qui n'est plus dans $D(y)$ et `valRemove` calcule les valeurs de (x,a) qui ne sont plus supportées en raison de la suppression de b de $D(y)$. Notons que les valeurs dans la file (pour la variable y) sont toujours considérées comme des supports potentiels etant donné que celles-ci n'ont pas encore été prises en compte dans cette contrainte. Nous limitons également notre attention aux valeurs qui ont b dans leur supports (par exemple, $(x,a) \in Cons(c,x,\{b\})$). Cependant, nous laissons à `valRemove` la possibilité de réaliser plus d'élagage ($\triangle_2$), ce qui est utile pour les contraintes monotones [16].

```
enqueue(in x: Variable;in a: Value;
        in C1: Set of Constraints;
        inout Q: Queue)
// Pre: x ∈ X, a ∉ D(x) et C1 ⊆ C
// Post: Q = Q0 ∪ {(c,x,a)|c ∈ C1, x ∈ Vars(c)}

post(in c: Constraint;out △: Set of Values)
// Pre: c ∈ C with Vars(c) = {x,y}
// Post: △ = Inc(c,x) ∪ Inc(c,y)
//   + initialisation de structures de données spécifiques

boolean valRemove(in c: Constraint;
                  in y: Variable; in b: Value;
                  out △: Set of Values)
// Pre: c ∈ C, Vars(c) = {x,y} , b ∉ D(y,Q,c)
// Post: △1 ⊆ △ ⊆ △2
//    avec △1 = Inc(c,x,D(y,Q,c)) ∩ Cons(c,x,{b})
//    et   △2 = Inc(c,x)
```

Specification 1 – Méthodes enqueue, post et valRemove pour AC5.

L'algorithme AC5 est décrit dans l'algorithme 1. La fonction `propagateQueueAC5` applique `valRemove` sur chaque élément de la file jusqu'à ce que la file soit vide. La fonction `initAC5` initialise la file. La Fonction `post(c, △)` calcule les valeurs inconsistantes de la contrainte c. Si elle supprime des valeurs de domaines, seules les contraintes déjà postées sont considérées par les appels de `enqueue`. Les contraintes qui ne sont pas encore postées ne sont pas concernées par cette suppression étant donné qu'elles vont utiliser le nouveau domaine des variables lors du post. L'appel de `post` généralement initialise certaines structures de données à utiliser dans `valRemove`. Avec une légère généralisation de la spécification de `post` et de `valRemove`, l'algorithme AC5 gère également les contraintes non binaires. AC5 est générique ; la mise en oeuvre de `post` et de `valRemove` est libre. Différentes contraintes peuvent avoir leur propre implémentation de ces fonctions. Cela permet a AC5 de combiner, dans un cadre unique, différents algorithmes tels que AC4, AC6, AC7 et AC2001 et d'exploiter la sémantique des contraintes pour atteindre une meilleure efficacité.

Proposition 1 *En supposant une implémentation correcte de* `post` *et de* `valRemove`*, AC5 est correct par rapport à sa spécification.*

Dans AC5, un élément (c,x,a) ne peut être mis qu'une seule fois dans la file. La taille de la file donc de $O(e.r.d)$ pour les CSP non binaires et $O(ed)$ pour les CSP binaires. Le nombre d'exécutions de `valRemove` est aussi borné par $O(e.r.d)$.

Proposition 2 *Pour les CSP binaires, si la com-*

```
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
4  //        (X, D(X)0, C)
5  //        et (X, D(X), C) is domain consistent
6    initAC5(Q);
7    propagateQueueAC5(Q);
8  }

9  initAC5(out Q){
10   Q =  ∅;
11   C1 =  ∅;
12   forall(c in C){
13     C1 += c;
14     post(c,△);
15     forall((x, a) in △){
16       D(x) -= a;
17       enqueue(x, a,C1,Q);
18     }
19   }
20 }

21 propagateQueueAC5(in Q){
22   while Q != ∅ {
23     select( (c, y, b) in Q) {
24       Q = Q - (c, y, b);
25       valRemove(c, y, b,△);
26       forall((x, a) in △){
27         D(x) -= a;
28         enqueue(x, a,C,Q);
29       }
30     }
31   }
32 }
```

Algorithm 1 – Algorithme AC5.

plexité temporelle de `post` *est de* $O(d^2)$ *et la complexité de* `valRemove` *est de* $O(d)$*, alors la complexité temporelle de AC5 est optimale et est de* $O(e.d^2)$*. Si la complexité temporelle de* `post` *est de* $O(d)$ *et la complexité amortie de toutes les exécutions de* `valRemove` *pour une contrainte est de* $O(d)$ *(par exemple, la complexité temporelle de* `valRemove` *est* $O(\triangle)$*), alors la complexité spatiale et temporelle de AC5 est* $O(ed)$*.*

Nous présentons l'algorithme AC4 comme une instanciation de AC5 en donnant le code des méthodes `post` et `valRemove` (Algorithmes 2). La méthode `valRemoveAC4` utilise une structure de données S pour enregistrer les supports de chaque valeur dans les différentes contraintes. Elle est initialisée par `postAC4` et satisfait l'invariant suivant à la ligne 21 de l'algorithme 1 (AC5).

```
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)
4  //       + initialisation de la structure de données S
5    post_varAC4(c, x, △1);
6    post_varAC4(c, y, △2);
7    △ = △1 ∪ △2;
8  }
9  post_varAC4(in c: Constraint;in x: Variable;
10             out △: Set of Values) {
11   △ = ∅;
12   forall(a in D(x)){
13     S[x, a, c] = ∅;
14     forall(b in D(y) : c(x/a, y/b))
15       S[x, a, c] += b ;
16     if (S[x, a, c]==∅)
17       △ += (x, a) ;
18   }
19 }
20 valRemoveAC4(in c: Constraint;in y: Variable;
21          in b: Value; out △: Set of Values) {
22 // Pre: c ∈ C, Vars(c) = {x, y} , b ∉ D(y, Q, c)
23 // Post: △ = Inc(c, x, D(y, Q, c)) ∩ Cons(c, x, {b})
24   △ = ∅;
25   forall(a in S[y, b, c]) {
26     S[x, a, c] -= b ;
27     if (S[x, a, c]==∅ & a in D(x))
28       △ += (x, a) ;
29   }
30 }
```

Algorithm 2 – Méthodes `post` et `valRemove` de AC4.

Soit $c \in C$ avec $Vars(c) = \{x, y\}$:

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

Et de manière similaire pour y. Cet invariant assure l'exactitude de `valRemoveAC4`. Après l'appel de `postAC4`, nous avons $\sum_{a \in D(x)} \#S[x, a, c] = \sum_{b \in D(y)} \#S[y, b, c]$ qui est $O(d^2)$. La taille de la structure de donnée est de $O(e.d^2)$.

3 NAC4

NAC4 (AC4 Négatif) est une nouvelle instance de AC5 qui est basée sur les valeurs interdites. Celles-ci sont dynamiquement maintenues lors de la propagation. Par NAC4, nous désignons l'algorithme AC5 dont les méthodes `postNAC4` et `valRemoveNAC4` sont définies comme décrites dans les Algorithmes 3 et 4.

NAC4 utilise une structure de données F pour enregis-

```
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  //          + initialisation des structures de données F,
5  //            setOfSize et localSize
6    post_varNAC4(c,x,△1);
7    post_varNAC4(c,y,△2);
8    localSize[x, c] = #D(x);
9    localSize[y, c] = #D(y);
10   △ = △1 ∪ △2;
11 }
12 post_varNAC4(in c: Constraint;in x: Variable;
13               out △: Set of Values) {
14   △ = ∅;
15   forall(k in 0..#D(y))
16     setOfSize[x, k, c] = ∅;
17   forall(a in D(x)){
18     F[x, a, c] = ∅;
19     forall(b in D(y) : ¬c(x/a, y/b))
20       F[x, a, c] += b ;
21     k = #F[x, a, c];
22     setOfSize[x, k, c]) += a;
23     if (k==#D(y))
24       △ += (x, a) ;
25   }
26 }
```

Algorithm 3 – La méthode `post` de NAC4.

trer les valeurs interdites pour chaque valeur dans les différentes contraintes. Pour une contrainte c, définie sur x, y, l'idée de base est que la valeur a devrait être retirée de $D(x)$ dès que l'ensemble des valeurs interdites pour (x, a) et l'ensemble $D(y)$ sont les mêmes. Ce test peut être réalisé efficacement (1) en raisonnant sur la taille de l'ensemble des valeurs interdites pour (x, a) et de l'ensemble $D(y)$, (2) en utilisant une structure de données triant par taille les ensmebles de valeurs interdites, et (3) en enregistrant la taille de la vue locale des domaines. Ces structures de données sont initialisées dans `postNAC4` et mises à jour dans `valRemoveNAC4`. La structure de données $F[x, a, c]$ enregistre l'ensemble des valeurs interdites pour (x, a) et c ; $setOfSize[x, k, c]$ désigne l'ensemble des valeurs b telles que $\#F[x, b, c] = k$ et $localSize[x, c]$ désigne la taille de la vue locale du domaine $D(x)$. La méthode `valRemoveNAC4` met tout d'abord à jour la taille de la vue locale de $D(y)$ et supprime b de la structure de données $setOfSize$ pour la variable y (lignes 5 à 7). Il met ensuite à jour l'ensemble des valeurs interdites ainsi que $setOfSize$ pour chaque paire $(x, a) \in F[y, b, c]$ (lignes 8 à 13). Enfin, il supprime les valeurs qui ne sont plus supportées, c'est-à-dire, les valeurs de $setOfSize[x, s, c] \cap D(x)$, où s est la taille locale de $D(y)$ (lignes 14 à 18).

```
valRemoveNAC4(in c: Constraint;in y: Variable;
          in b: Value,out △: Set of Values) {
// Pre: c ∈ C, Vars(c) = {x, y} , b ∉ D(y, Q, c)
// Post: △ = Inc(c, x, D(y, Q, c)) ∩ Cons(c, x, {b})
  localSize[y, c]-- ;
  k = #F[y, b, c];
  setOfSize[y, k, c] -= b;
  forall(a in F[y, b, c]){
    F[x, a, c] -= b ;
    k = #F[x, a, c];
    setOfSize[x, k + 1, c] -= a;
    setOfSize[x, k, c] += a;
  }
  △ = ∅;
  s = localSize[y, c];
  forall(a in setOfSize[x, s, c] : a in D(x))
    △ += (x, a);
}
```

Algorithm 4 – La méthode valRemove de NAC4.

Les structures de données satisfont l'invariant suivant à la ligne 21 de l'Algorithme 1 (AC5). Soit $c \in C$ avec $Vars(c) = \{x, y\}$:

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

et de manière similaire pour y. A partir de ces invariants, nous avons que $F[x, a, c] \subseteq D(y, Q, c)$ est vrai à la ligne 24 et que la valeur a doit être supprimée de $D(x)$ si $F[x, a, c] = D(y, Q, c)$. Donc, si $s = localSize[y, c]$, l'algorithme doit retirer les valeurs de $setOfSize[x, s, c]$ du domaine de $D(x)$. Ces invariants assurent l'exactitude de `valRemoveNAC4`.

La taille des structure de données est de $O(e.d^2)$. Dans l'élagage de la méthode postNAC4, la vue locale de la taille du domaine $D(y)$ est $\#D(y)$ vu que la file ne contient pas d'éléments de la forme $(x, ., c)$. La complexité de `postNAC4` est $O(d^2)$ et de la complexité de `valRemoveNAC4` est de $O(d)$. Ainsi, par la propriété 2, la complexité globale de NAC4 est de $O(ed^2)$, qui est la complexité optimale pour assurer la consistance de domaine sur les CSP binaires.

Example 4 *NAC4 est illustré sur le CSP suivant :* $c_1(x, y) = \{(1, 4), (1, 5), (2, 2), (2, 5), (3, 1), (3, 3), (3, 4)\}$, $c_2 : y \neq 4$, $c_3 : y \neq 5$, $D(x) = \{1, 2, 3\}$ *et* $D(y) = \{1, 2, 3, 4, 5\}$. *L'execution de* `postNAC4`$(c_1, \triangle)$ *produit* $\triangle = \emptyset$ *et remplit les structures de données*

comme suit :

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

$$setOfSize[x,1,c_1]=\emptyset \qquad setOfSize[y,1,c_1]=\{4,5\}$$
$$setOfSize[x,2,c_1]=\{3\} \qquad setOfSize[y,2,c_1]=\{1,2,3\}$$
$$setOfSize[x,3,c_1]=\{1,2\} \qquad setOfSize[y,3,c_1]=\emptyset$$
$$setOfSize[x,4,c_1]=\emptyset$$
$$setOfSize[x,5,c_1]=\emptyset$$
$$localSize[x,c_1] \quad =3 \qquad localSize[y,c_1] \quad =5$$

`postNAC4`*$(c_2,\triangle)$ retourne $\triangle = \{(y,4)\}$ et* `postNAC4`*$(c_3,\triangle)$ produit $\triangle = \{(y,5)\}$, ce qui donne $Q=\{(c_1,y,4),(c_1,y,5)\}$, $D(x)=\{1,2,3\}$, et $D(y)=\{1,2,3\}$. La methode* `valRemoveNAC4`*$(c_1,y,4,\triangle)$ met à jour les variables suivantes :*

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

Etant donné que $setOfSize[x,4,c_1]=\emptyset$, on a $\triangle=\emptyset$, $Q=\{(c_1,y,5)\}$, $D(x)=\{1,2,3\}$, et $D(y)=\{1,2,3\}$. `valRemoveNAC4`*$(c_1,y,5,\triangle)$ met à jour les variables suivantes :*

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

Etant donné que $setOfSize[x,3,c_1]=\{1\}$, on a $\triangle=\{(x,1)\}$, $Q=\{(c_1,x,1)\}$, $D(x)=\{2,3\}$, et $D(y)=\{1,2,3\}$. `valRemoveNAC4`*$(c_1,x,1,\triangle)$ met à jour les variables suivantes :*

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

Les domaines deviennent finalement $D(x)=\{2,3\}$ et $D(y)=\{1,2,3\}$.

Proposition 3 *Soit $c \in C$ sur les variables $\{x,y\}$. Les Invariants (3-6.x-y) sont satisfaits a la ligne 21 de AC5.*

Proposition 4 *NAC4 est correct et sa complexité spaciale et temporelle est de $O(e.d^2)$.*

4 Applications

Nous allons maintenant examiner une série d'applications liées au principe du maintien dynamique des valeurs interdites.

Contraintes AC éparses Les instances AC4 et NAC4 de AC5 peuvent être combinées ; chaque contrainte mettant en oeuvre sa version AC4 ou NAC4 des méthodes `post` et `valRemove` valRemove. Ceci sera noté AC5(AC4,NAC4). Une propriété intéressante de AC5(AC4,NAC4) est que la complexité amortie de toutes les exécutions de `valRemoveAC4` ou `valRemoveNAC4` pour une contrainte est limitée par le nombre d'éléments dans la structure de données S ou F de cette contrainte. On obtient alors la spécialisation suivante de la proposition 2.

Proposition 5 *Si une version spécialisée de* `postAC4` *ou* `postNAC4` *exploitant la sémantique de la contrainte s'exécute en un temps $O(K)$ pour chaque contrainte d'un CSP binaire, alors la complexité spaciale et temporelle de AC5(AC4,NAC4) est $O(e.K)$.*

Comme cas particulier, si S ou F peut être rempli en $O(d)$, un algorithme de domaine consistance s'exécute en un temps $O(ed)$, tel que formalisé par la classe contraintes suivante.

Definition 5 *Une contrainte c avec $Vars(c)=\{x,y\}$ est* positivement éparse *par rapport à un domaine D si et seulement si $\#\{(a,b)\in D^2 | c(x/a,y/b)\}$ est $O(\#D)$. La contrainte c est* négativement éparse *par rapport à un domaine D si et seulement si $\neg c$ est* positivement éparse *par rapport D.*

Example 5 *Des exemples de contraintes positivement et négativement éparses sont les contraintes bijectives ($x+y=k$, où k est une constante), les contraintes anti-bijectives ($x+y\neq k$), les contraintes fonctionnelles ($x=|y-k|$ ou $x=y \bmod k$), les contraintes anti-fonctionnelles ($x\neq|y-k|$ ou $x\neq y \bmod k$), mais comprennent aussi des des contraintes (anti-)fonctionnelles telles que$|x-y|=k$ et $|x-y|\neq k$. On peut aussi considérer les contraintes de congruence, tels que $(x+y) \bmod k=0$ et $(x+y) \bmod k\neq 0$ qui sont éparses lorsque k est $O(d)$.*

Grâce à la généricité d'AC5, nous pouvons exploiter la sémantique des contraintes spécifiques dans la méthode `postAC4` des contraintes positivement éparses et de `postNAC4` pour les contraintes négativement éparses afin de remplir la structure de données S ou F en $O(d)$ et obtenir ainsi une complexité temporelle de $O(d)$.

Proposition 6 *Pour des contraintes positivement et*

négativement éparses, la complexité spaciale et temporelle est de $O(ed)$.

Combinaison de constraintes sur des mêmes variables Considérons maintenant une contrainte c sur $\{x, y\}$ définie comme une combinaison booléenne de contraintes $\{c_1, \ldots, c_k\}$ portant sur les mêmes variables. Supposons pour simplifier que le nombre de connecteurs logiques est bornée par k. La contrainte c peut être postée en AC5(AC4,NAC4) avec une complexité de $O(k.d^2)$. L'étape de propagation sur cette contrainte sera alors réalisée en temps $O(K)$ pour atteindre la consistance de domaine, où K est le nombre de supports.

Example 6 *Considérons la contrainte $c \equiv (c_1 \wedge c_2) \vee (c_3 \wedge c_4)$ où $c_1 \equiv x \neq |y-2|$, $c_2 \equiv y - 1 \neq x \bmod 2$, $c_3 \equiv x = |y-1|$, $c_4 \equiv |x-2| = y$, avec $D(x) = \{0, 1\}$ et $D(y) = \{1, 2\}$. Chaque contrainte c_i est domaine consistante, mais ni $c_1 \wedge c_2$ ni $c_3 \wedge c_4$ ne l'est. Une application de AC4 (ou NAC4) sur c permettra de détecter une inconsistance.*

Dans certains cas, comme dans l'exemple ci-dessus, il est possible de parvenir à une meilleure complexité en exploitant conjointement les supports et les valeurs interdites. L'idée centrale est que chaque contrainte c_i doit utiliser les supports ou les valeurs interdites en fonction de sa sémantique. Ensuite, les contraintes individuelles sont combinées par des opérateurs logiques qui utilisent les supports et les valeurs interdites afin de calculer ses propres supports ou valeurs interdites, et cela de façon récursive. Le Tableau 1 présente les règles permettant de combiner des contraintes et de calculer la structure de données S ou F pour les variables x et y, selon la structure de données maintenue dans les sous-expressions. Les règles sont données pour la variable x, mais sont similaires pour y. Une contrainte c_i qui utilise la structure de données S (resp. F) sera notée c_i^+ (resp. c_i^-). Si la méthode post applique ces règles à la contrainte c, alors l'algorithme réalise ensuite la consistance de domaine. Il n'y a pas coût temporel ou spatial supplémentaire étant donné que toutes les opérations du tableau 1 peuvent être effectuées en temps $s_1 + s_2$, où s_i est la taille de la structure de données (S ou F) pour c_i. Comme cas particulier, si la complexité temporelle pour poster chaque c_i est de $O(d)$ (e.g. contrainte fonctionnelle ou anti-fonctionnelle), la complexité spatiale et temporelle pour poster la contrainte c est de $O(k.d)$.

Proposition 7 *Étant donné un ensemble de contraintes binaires C et une contrainte binaire c exprimée comme une combinaison logique des contraintes $c_1, \ldots, c_k$ avec $Vars(c) = Vars(c_i)$ $(1 \leq i \leq k)$, si la méthode* `post` *pour c applique les règles de la table 1, alors l'algorithm AC5(AC4,NAC4) sur $C \cup \{c_1, \ldots, c_k\}$ réalise la domaine consistance. Si la complexité temporelle des méthodes* `post` *des contraintes de $C \cup \{c_1, \ldots, c_k\}$ est de $O(d)$, alors la complexité temporelle et spatiale de AC5(AC4,NAC4) appliqué sur $C \cup \{c\}$ est $O((e+k).d)$, avec $e = \#C$.*

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

TABLE 1 – Règles pour combiner $c_1(x, y)$ et $c_2(x, y)$.

Validité et implication AC5(AC4,NAC4) peut être étendu pour supporter les méthodes `isValid` et `isEntailed` (Spécifications 2). Avec AC4, une variable/valeur (x, a) est détectée comme étant valide dans c si la taille de $S[x, a, c]$ est $\#D(y, Q, c)$. AC4 doit pour cela maintenir les structures $setOfSize$ et $localSize$ structures de données de NAC4. Avec NAC4, une variable/valeur (x, a) est détectée comme étant valide dans c si $F[x, a, c]$ est vide. L'invariant de la structure de données pour AC4 et NAC4 serait alors (1-6.x-y). La complexité théorique de AC5(AC4,NAC4) est inchangée, tandis que la complexité pratique est à peu près doublée. AC4 et NAC4 garderaient ainsi le nombre de valeurs valides pour chaque contrainte c. Si les valeurs valides pour x dans c atteint $\#D(x, Q, C)$, alors la contrainte est détectés toujours vraie (en supposant que les domaines soient non vides). On pourrait aussi étendre facilement les méthodes `post` et `valRemove` à des méthodes `post`(c,$\triangle^-$,$\triangle^+$) et `valRemove`(c,y,b,$\triangle^-$,$\triangle^+$) où l'argument supplémentaire $\triangle^+$ retourne l'ensemble des nouvelles valeurs valides, défini comme

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

pour `post`, et

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

pour `valRemove`. Ces algorithmes étendus de domaine consistance sont utiles pour des combinateurs de contraintes, la réification ainsi que dans un cadre *Ask & Tell*.

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

Specification 2 – Les méthodes `isValid` et `isEntailed`.

Combinaison de contraintes sur des variables différentes Assurer la consistance de domaine sur une combinaison de contraintes (binaires) portant sur des variables différentes est un problème NP-difficile. Une approximation de consistance de domaine peut être réalisée en utilisant le cadre proposé dans [1], où les contraintes primitives produisent non seulement des valeurs incompatibles, mais aussi les valeurs valides. Notre algorithme AC5(AC4,NAC4) étendu peut être utilisé pour combiner les contraintes en utilisant l'algèbre proposée dans [1]

5 Résultats experimentaux

Cette section illustre les avantages d'une exploitation conjointe des supports et des valeurs interdites. Nous avons évalué AC4, NAC4, et leur combinaison sur des CSPs impliquant lees contraintes positivement et négativement éparses $x = y \bmod k$, $x = |y-k|$, $x+y = k$, $|x - y| = k$, $(x + y) \bmod k = 0$ et leurs versions négatives, où k est une constante. Trois séries de 20 CSPs ont été générés. Le premier ensemble contient uniquement des contraintes positives (CPOS), le second uniquement des contraintes négatives (CNEG), et le troisième des contraintes positives et négatives (cPosNeg). Les résultats sont présentés dans la Table 2. Le nom de Cneg_50_200_10 signifie que chaque CSP a 50 variables avec un domaine {0..199}, et qu'il y a 10% de contraintes entre toutes les paires de variables distinctes. Les paramètres ont été choisis pour éviter les CSPs trivialement satisfaisables ou trivialement inconsistants. Les contraintes ont été choisies au hasard en utilisant une distribution uniforme. Les valeurs k sont également déterminées en utilisant une distribution uniforme. Chaque CSP a été résolu en Comet [9] en utilisant quatre algorithmes de consistance différents : (1) AC4 pour chaque contrainte, (2) NAC4 pour chaque contrainte, (3) la combinaison AC5(AC4,NAC4) en utilisant AC4 pour les contraintes positives et NAC4 pour les contraintes négatives et (4) la combinaison AC5(AC4*,NAC4*) qui est semblable à AC5(AC4,NAC4), mais utilise des méthodes spécialisées `post` (linéaires) exploitant la sémantique des contraintes. Pour un CSP qui ne contient que des contraintes positives, AC5(AC4,NAC4) se réduit à AC4 et, pour un CSP qui ne contient que des contraintes négatives, AC5(AC4,NAC4) se réduit à NAC4. Nous donnons le temps d'exécution moyen (en secondes), ainsi que le pourcentage de CSPs consistants dans chaque ensemble de données. L'inconsistance d'un CSP est toujours détectée dans le noeud racine de l'arbre de recherche. Pour les CSPs consistants, la recherche est arrêtée après 1000 noeuds. Les expériences ont été réalisées sur un seul coeur d'un ordinateur avec processeur Intel Core Duo cadencé à 2,8 GHz et avec 4 Go de mémoire.

Pour les contraintes positivement éparses, AC4 est beaucoup plus efficace (facteur d'accélération de 41) que NAC4, tandis que NAC4 est beaucoup plus efficace que AC4 (facteur d'accélération de 11,5) pour les contraintes négativement éparses. Ceci montre l'intérêt de NAC4. L'utilisation d'une contrainte post spécialisée conduit à un facteur d'accélération de 2,57 pour AC4 et de 1,17 pour NAC4. Pour les CSPs combinant des contraintes positivement et négativement éparses, NAC4 est 3,5 fois plus lent que AC4, ce qui s'explique par les structures de données plus complexes à maintenir pour NAC4. Cette dernière série de CSPs montre l'intérêt d'un algorithme générique permettant la combinaison de différents algorithmes tels que AC4 et NAC4. Le facteur d'accélération de AC5(AC4,NAC4) par rapport à AC4 est de 12,7. Il augmente jusqu'à 14,9 lorsque l'on utilise des méthodes post spécialisée dans AC5(AC4*,NAC4*).

6 Conclusion

Ce papier propose l'algorithme optimal NAC4 pour la domaine consistance ; cet algorithme n'utilise pas les supports, mais maintient dynamiquement les valeurs interdites lors de la propagation. Les principes de NAC4 peuvent être combinés au sein de l'algorithme générique AC5 avec les techniques utilisées dans AC4, AC6, et AC2001 pour exploiter la sémantique des contraintes et obtenir ainsi une plus grande efficacité. En particulier, les valeurs interdites permettent à AC5 d'obtenir la consistance de domaine en un temps $O(ed)$ pour les catégories de contraintes dans lesquelles le nombre de supports est de $O(d^2)$, mais le nombre de

	% 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 – Comparaison de AC4, NAC4, AC5(AC4,NAC4) et AC5(AC4*/NAC4*).

valeurs interdites est de $O(d)$. Cet article montre également comment les valeurs interdites et les supports peuvent être utilisés conjointement pour assurer la consistance de domaine sur des combinaisons logiques de contraintes et de calculer la validité et l'implication de contraintes. Les résultats expérimentaux montrent que la combinaison de supports et de valeurs interdites permet de considérablement réduire le coût de calcul de la consistance de domaine pour certaines classes de contraintes.

Les travaux futurs visent à comparer AC5(AC4/NAC4) avec d'autres algorithmes de domaine consistance, à réaliser une évaluation expérimentale sur d'autres jeux de tests, y compris des instances non-binaires ainsi qu'à étendre NAC4 pour gérer des tables négativess représentées de manière compacte, telle que décrits dans [10, 11, 6, 7].

Remerciements

Cette recherche est partiellement financée par le programme des Pôles d'Attraction Interuniversitaires (Politique scientifique belge), ainsi que par le projet FRFC 2.4504.10 du Fond National belge de la Recherche Scientifique.

Références

[1] Fahiem Bacchus and Toby Walsh. Propagating logical combinations of constraints. In *IJCAI*, pages 35–40, 2005.

[2] Christian Bessiere. Constraint propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*. Elsevier Science Inc., New York, NY, USA, 2006.

[3] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks : Preliminary results. In *IJCAI*, pages 398–404, 1997.

[4] Christian Bessière and Jean-Charles Régin. Local consistency on conjunctions of constraints. In *ECAI-98, proceedings Workshop on Non Binary Constraints*, pages 53–60, 1998.

[5] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Support inference for generic filtering. In *CP*, pages 721–725, 2004.

[6] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *CP'08*, pages 509–523. Springer-Verlag, 2008.

[7] Kenil C. K. Cheng and Roland H. C. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2) :149–204, 2010.

[8] Yves Devilel and Pascal Van Hentenryck. Domain consistency and forbidden values. In *CP*, pages 191–205, 2010.

[9] Dynadec. Comet user manuel. In *www.dynadec.be*, 2010.

[10] Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *AAAI'07*, pages 191–197, 2007.

[11] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *CP'07*, pages 379–393. Springer-Verlag, 2007.

[12] Christophe Lecoutre. *Constraint Networks : Techniques and Algorithms*. ISTE/Wiley, 2009.

[13] Olivier Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *CPAIOR*, pages 209–224, 2004.

[14] Deepak Mehta and Marc R. C. van Dongen. Reducing checks and revisions in coarse-grained MAC algorithms. In *IJCAI*, pages 236–241, 2005.

[15] Pascal Van Hentenryck and Yves Deville. The cardinality operator : A new logical connective for constraint logic programming. In *ICLP*, pages 745–759, 1991.

[16] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artif. Intell.*, 57(2-3) :291–321, 1992.

[17] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. *Constraint Programming : Basics and Trends*, chapter Design, Implementation, and Evaluation of the Constraint Language cc(FD), pages 293–316. Springer, 1994.