CAC: Un algorithme d'arc-consistance configurable, générique et adaptatif

Jean-Charles Régin
ILOG Sophia Antipolis,
Les Taissounières HB2,
1681 route des Dolines,
06560 Valbonne, France,
email: `regin@ilog.fr`

Résumé

Dans ce papier, nous présentons CAC, un nouvel algorithme établissant la consistance d'arc pour les contraintes binaires. Cet algorithme est configurable, générique et adaptatif. "CAC est configurable" signifie qu'en combinant certains de ses paramètres, on peut représenter n'importe quel algorithme existant : AC-3, AC-4, AC-6, AC-7, AC-2000, AC-2001, AC-8, AC-3_d, AC-3.2 and AC-3.3. CAC est générique, comme AC-5, car il est capable d'exploiter la sémantique des contraintes. CAC est adaptatif parce qu'il est capable de sélectionner à tout moment l'algorithme le plus efficace. Ce nouvel algorithme nous permet de proposer une nouvelle nomenclature des algorithmes d'arc consistance qui est basée sur les caractéristiques des algorithmes, par exemple les valeurs qui sont reconsidérées lorsqu'un domaine est modifié, où la manière dont un nouveau support est recherché. Cette nouvelle nomenclature montre que plusieurs nouvelles combinaisons sont possibles. Autrement dit, nous pouvons facilement combiner certaines des idées d'AC-3 avec certaines des idées d'AC-7 et certaines des idées d'AC-2001 avec certaines idées d'AC-6. Les avantages de notre approche sont mis en évidence expérimentalement.

1 Introduction

Dans ce papier, nous ne considérons que des contraintes binaires.

Depuis plus de vingt ans, de nombreux algorithmes établissant la consistance d'arc (algorithmes d'AC) ont été proposé : AC-3 [Mac77], AC-4 [MH86], AC-5 [VDT92], AC-6 [Bes94], AC-7, AC-Inference, AC-Identical [BFR99], AC-8 [CJ98], AC-2000 : [BR01], AC-2001 (alse denoted by AC-3.1) [BR01], AC-3_d [van02], AC-3.2 and AC-3.3 [LBH03]. Malheureusement, ces algorithmes sont décrits de façons différentes et leur comparaison n'est pas facile. Pour remédier à cela nous présentons CAC un nouvel algorithme qui est configurable, générique et adaptatif.

Configurable signifie que tous les algorithmes connus peuvent être représentés en combinant les paramètres de CAC. Cela a plusieurs avantages :

• on obtient un seul algorithme.

• cela montre clairement les différences entre tous les algorithmes, ce qui continue la discussion commencée par [BR01].

• de nouveaux algorithmes d'AC peuvent être facilement et rapidement dérivés de CAC, parce que certaines combinaisons de paramètres n'ont jamais été testées.

• une nouvelle nomenclature dérive naturellement de CAC. Cette nomenclature est bien plus explicite que celle couramment employée ("AC-" suivi par un nombre), parce que les algorithmes sont maintenant nommés en fonction de combinaisons de paramètres prédéfinis qu'ils utilisent. Par exemple AC-3 est renommé CAC-pvD-sD et AC-6 devient CAC-pvΔs-last-sD.

Générique signifie que CAC peut prendre en compte les spécificités de certaines contraintes binaires. Des algorithmes dédiés pour certaines contraintes peuvent être facilement écrits à partir de CAC, par exemple pour les contraintes fonctionnelles. On peut en ce sens comparer CAC et AC-5. Dans notre cas, l'aspect incrémental d'AC-5 est amélioré, puisque l'on peut utiliser les principes de n'importe quel algorithme connu lors du développement d'un algorithme particulier.

Adaptatif signifie que CAC est capable d'utiliser successivement des algorithmes différents, comme il est suggéré dans [BR01]. Par exemple, en fonction du meilleur algorithme pour une configuration donnée des domaines et des delta domaines, CAC pourra utiliser AC-2001 , puis AC-7 et enfin AC-2001. Ce point est particulièrement important, car nous pensons que **la programmation par contraintes pourra être fortement améliorée si un algorithme de filtrage est capable par lui-même de sélectionner à tout moment sa version la plus efficace, au lieu de demander à l'utilisateur de le faire à-priori.**

Les algorithmes d'AC procèdent en deux étapes. Premièrement, une phase d'initialisation est appelée. Cette phase consiste à rechercher un support (c'est-à-dire une valeur compatible) pour chaque valeur. Si une valeur n'a pas de support alors elle est supprimée. Ensuite, dans une deuxième étape, les conséquences de la disparition de valeurs sont étudiées : un nouveau support valide doit être recherché pour certaines valeurs. Dans ce papier, nous ne considérerons que la seconde étape qui est la plus importante.

Ce papier est organisé comme suit. Tout d'abord, nous rappelons certaines définitions de la programmation par contraintes. Puis, nous étudions tous les algorithmes d'AC existants, et nous identifions les différents concepts utilisés par ces algorithmes. Ensuite, nous détaillons l'algorithme CAC et proposons une nouvelle nomenclature. Nous prêterons une attention particulière à l'aspect adaptatif de CAC. Enfin, nous donnons certains résultats expérimentaux et concluons.

2 Preliminaries

Un **réseau de contraintes** $\mathcal{N}$ est défini par un ensemble de n **variables** $X = \{x_1, \ldots, x_n\}$, une ensemble de **domaines** $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ où $D(x_i)$ est un ensemble fini de **valeurs** possible pour la variable x_i, et un ensemble $\mathcal{C}$ de **contraintes** entre variables. Une **contrainte** C définie sur un ensemble ordonné de variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ est un sous-ensemble $T(C)$ du produit cartésien $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ qui spécifie les combinaisons **autorisées** de valeur pour les variables $x_{i_1}, \ldots, x_{i_r}$. Un élément

de $T(C)$ est appelé un **tuple** de $X(C)$ et r est **l'arité** de C. Une valeur a d'une variable x est notée (x, a). (x, a) est **valide** si $a \in D(x)$, et un tuple est **valide** si toutes les valeurs qu'il contient sont valides.
Etant donné C une contrainte, C est **consistante** si et seulement si il existe un tuple τ de $T(C)$ qui soit valide. Une valeur $a \in D(x)$ est **consistante avec** C si et seulement si $x \notin X(C)$ ou il existe un tuple valide τ de $T(C)$ avec $(x, a) \in \tau$. Une contrainte est **arc consistante** si et seulement si $\forall x_i \in X(C), D(x_i) \neq \emptyset$ et $\forall a \in D(x_i)$, a is consistante avec C.

Un **algorithme de filtrage** associé à une contrainte C est un algorithme qui supprime des valeurs qui ne sont pas consistantes avec C, et qui ne supprime pas de valeur consistante avec C. Si l'algorithme de filtrage supprime toutes les valeurs inconsistantes avec C, alors on dit qu'il réalise la consistance d'arc de C.

La **propagation** est le mécanisme qui consiste à appeler l'algorithme de filtrage associé aux contraintes impliquant une variable x chaque fois que le domaine de x est modifié.

L'ensemble des valeurs qui sont supprimées du domaine d'une variable x est appelé **le delta domaine** de x. Cet ensemble est noté $\Delta(x)$. Nous invitons le lecteur à consulter [VDT92] pour de plus amples informations à propos des delta domaines.

La fonction PROPAGATION de l'algorithme 1 est une implémentation possible du mécanisme de propagation. L'algorithme de filtrage associé à une contrainte C définie sur x et y corresponds à la fonction FILTER$(C, x, y, \Delta(y))$. Dans ce cas, l'algorithme de filtrage supprime des valeurs de $D(x)$ qui ne sont pas consistantes avec la contrainte en fonction du delta domaine de y. Nous supposerons dans un souci de clarté que cette fonction n'a pas d'effets de bord, autrement dit que le domaine de y n'est pas modifié lors de cet appel. Pour une contrainte C la fonction FILTER sera aussi appelée avec les paramètres $(C, y, x, \Delta(x))$. Nous supposerons également que la fonction RESET$(\Delta(y))$ est disponible. Cette fonction réinitialise $\Delta(y)$ à l'ensemble vide. L'algorithme que nous proposons est donné à titre d'exemple et d'autres algorithmes peuvent être envisagés. Notre présentation est conceptuellement proche de celle d'AC-5 [VDT92].

Algorithm 1: function PROPAGATION

```
PROPAGATION()
    while ∃y such that Δ(y) ≠ ∅ do
        pick y with Δ(y) ≠ ∅
        for each constraint C involving y do
            if ¬ FILTER(C, x, y, Δ(y)) then return false
        RESET(Δ(y))
    return true
```

3 Algorithmes d'AC

Nous considérerons, par la suite, une contrainte C définie sur les variables x et y pour laquelle on doit étudier les conséquences des modifications du domaine de y.

Definition 1 *Nous appelerons* **valeurs pendantes** *l'ensemble des valeurs* **valides** *d'une variable x pour lesquelles un support est cherché par un algorithme d'AC lorsque les conséquences des suppressions de valeurs de la variable y sont étudiées.*

Grâce à cette définition, les principes des algorithmes d'AC peuvent être facilement exprimés : **Rechercher un support valide pour chaque valeur pendante et supprimer celles qui n'en ont pas.**

L'algorithme 2 est une implémentation possible de ce principe. C'est aussi le coeur de l'algorithme CAC. Les fonctions SELECTPENDINGVALUEMODE et SELECTEXISTSUPPORTMODE peuvent être ignorées pour l'instant ; elles seront détaillées plus tard. Nous

Algorithm 2: Algorithme de filtrage CAC

```
FILTER(in C, x, y, Δ(y)) : boolean
    get the parameters of C : pvMode and sMode
    pvMode ← SELECTPENDINGVALUEMODE(C, x, y, Δ(y), pvMode)
    sMode ← SELECTEXISTSUPPORTMODE(C, x, y, Δ(y), sMode)
    (x, a) ← pvMode.FIRSTPENDINGVALUE(C, x, y, Δ(y))
    while (x, a) ≠ nil do
        if ¬ EXISTVALIDSUPPORT(C, x, a, y, sMode) then
1           REMOVEFROMDOMAIN(x, a)
            if D(x) = ∅ then return false
        (x, a) ← pvMode.NEXTPENDINGVALUE(C, x, y, Δ(y), a)
    return true
```

pouvons maintenant donner les principes des fonctions FIRSTPENDINGVALUE, NEXTPENDINGVALUE et EXISTVALIDSUPPORT pour chaque algorithme d'AC existant.

AC-3 : Les valeurs pendantes sont les valeurs de $D(x)$, et $\Delta(y)$ n'est pas utilisé. Toutes les valeurs de $D(x)$ sont donc considérées et la recherche d'un nouveau support valide pour (x, a) est faite en testant la compatibilité de (x, a) avec les valeurs de $D(y)$ jusqu'à en trouver une compatible. Il n'y a aucune mémorisation des précédents calculs, aussi le même test entre (x, a) et une valeur b de $D(y)$ peut être effectué plusieurs fois. La complexité en temps est en $O(d^3)$ par contrainte[1]. L'avantage de cette approche est que la complexité en espace est en $O(1)$ par contrainte.

AC-4 : Dans AC-4 l'ensemble des tuples est précalculé et sauvegardé dans une structure de données que nous appelerons **table**. Cette table contient pour chaque valeur (x, a) un pointeur vers le prochain tuple contenant (x, a). La complexité d'AC-4 est donc en $O(d^2)$ pour une contrainte. Les valeurs pendantes sont pour chaque valeur $(y, b) \in \Delta(y)$ l'ensemble des valeurs (x, a) compatibles avec (y, b), c'est-à-dire telles que $((x, a)(y, b)) \in T(C)$. Il est important de remarquer qu'une valeur (x, a) peut-être considérée plusieurs fois comme étant pendante. La recherche d'un nouveau support valide est alors particulièrement efficace, puisque la fonction EXISTVALIDSUPPORT est implémentée en $O(1)$ en associant à chaque valeur (x, a) un compteur contenant le nombre de supports valides de (x, a) dans $D(y)$. Chaque fois que la fonction EXISTVALIDSUPPORT est appelée cela signifie que la valeur a perdu un support valide, donc le compteur est décrémenté et s'il devient égal à zéro alors cela signifie que la valeur n'a plus de support. AC-4 a été le premier algorithme dont la complexité en temps est en $O(d^2)$ par contrainte, autrement dit optimale. AC-4 ne refait jamais deux fois le même calcul, mais effectue systématiquement de nombreuses opérations pour éviter cela.

[1] Dans le papier nous exprimerons toujours les complexités par contrainte, car un réseau de contraintes peut impliquer divers types de contraintes. On retrouve les complexités habituelles en les multipliant par le nombre de contrainte binaires.

AC-5 : Cet algorithme est principalement un algorithme générique. Il a été inventé afin de pouvoir prendre en compte la sémantique des contraintes, c'est-à-dire les spécificités de leurs structures. Par exemple, la consistance d'arc des contraintes fonctionnelles peut être réalisée en $O(d)$ par contrainte. AC-5 donne la possibilité à l'utilisateur de spécialiser la fonction existValidSupport afin de pouvoir exploiter la structure des contraintes. La fonction filter et le mécanisme de propagation que nous avons donné sont proches des idées d'AC-5.

AC-6 : AC-6 a été une avancée majeure dans la compréhension des principes des algorithmes d'AC. AC-6 combine certaines idées d'AC-3 avec certaines idées d'AC-4. AC-6 utilise, comme AC-4, le delta domaine pour déterminer les valeurs pendantes, mais au lieu de considérer toutes les valeurs supportées par les valeurs de $\Delta(y)$, il exploite le fait que la connaissance d'un support est suffisante. AC-6 peut donc être vu comme un calcul paresseux des supports. Pour obtenir ce résultat AC-6 introduit une nouvelle structure de données qui est une variation d'une table : les **S-list**. Pour chaque valeur (y, b), la S-list associée à (y, b), notée S-list$[(y, b)]$, est la liste des valeurs qui sont actuellement supportées par (y, b). Contrairement à AC-4, dans AC-6 la connaissance d'un seul support est suffisante, aussi une valeur (x, a) est supportée par une et une seule valeur de $D(y)$, donc il y a une et une seule valeur de $D(y)$ qui contient (x, a) dans sa S-list. Les valeurs pendantes sont les valeurs valides qui sont contenues dans les S-list des valeurs de $\Delta(y)$, et, pour un $\Delta(y)$ donné, une valeur (x, a) ne peut être considérée qu'une seule fois comme étant pendante. La fonction existValidSupport est une amélioration de celle d'AC-3. En effet, la recherche d'un valeur compatible dans le domaine suit un ordre des valeurs et commence à partir du support qui vient de devenir invalide, c'est-à-dire la valeur du delta domaine contenant (x, a) dans sa S-list. Par contrainte, la complexité en espace d'AC-6 est en $O(d)$ et la complexité en temps est en $O(d^2)$.

AC-7 : C'est une amélioration d'AC-6. AC-7 exploite le fait que si (x, a) est supporté par (y, b) alors (y, b) est aussi supporté par (x, a). Aussi, lorsque l'on recherche un support valide pour (x, a), AC-7 propose de rechercher d'abord une valeur valide dans S-list$[(x, a)]$, et toute valeur non valide trouvée est supprimée de cette liste. On dit alors que l'on recherche un support par **inférence**. Cette idée va à l'encontre d'un invariant d'AC-6 : un support trouvé par inférence n'est plus nécessairement la dernière valeur testée dans $D(y)$. Aussi, AC-7 introduit explicitement la notion de **dernière valeur testée** en utilisant la donnée **last** associée à chaque valeur. AC-7 garantit la propriété suivante : si last$[(x, a)] = (y, b)$ alors il n'y a pas de support (y, a) de $D(y)$ avec $a < b$. Si aucun support n'est trouvé par inférence, alors AC-7 utilise une amélioration d'AC-6 pour trouver un support. Lorsque l'on veut savoir si (y, b) est un support pour (x, a) on peut immédiatement répondre par la négative si last$[(y, b)] > (x, a)$, parce que last$[(y, b)] > (x, a)$ signifie que (x, a) n'est pas un support de (y, b) et donc que (y, b) n'est pas non plus un support pour (x, a). Les deux propriétés sur lesquelles s'appuie AC-7 sont souvent appelées multi-directionnalité. Donc AC-7 économise certains tests de consistance réalisé par AC-6 tout en ayant les mêmes complexités.

AC-Inference : Cet algorithme utilise les listes S-list d'AC-6 pour déterminer de la même façon les valeurs pendantes, mais la recherche d'un nouveau support est différente de celle d'AC-6. Pour chaque valeur (x, a) deux listes de valeurs sont utilisées : **P-**

list$[(x,a)]$ et **U-list**$[(x,a)]$. P-list$[(x,a)]$ contient certains supports de (x,a), alors que U-list$[(x,a)]$ contient les valeurs pour lesquelles on n'a jamais testé la compatibilité avec (x,a). Lorsque l'on cherche un support valide pour (x,a), AC-Inference teste tout d'abord s'il existe une valeur valide dans P-list$[(x,a)]$, et supprime de cette liste toute valeur non valide atteinte. Si aucune valeur valide n'est trouvée dans P-list$[(x,a)]$, alors les valeurs de U-list$[(x,a)]$ sont successivement considérées jusqu'à trouver un support valide. Toute valeur de U-list$[(x,a)]$ atteinte est supprimée de cette liste. Quand un nouveau support est trouvé, alors des règles d'inférence sont appliquées afin de déduire de nouveaux supports et les listes P-list$[(x,a)]$ et U-list$[(x,a)]$ sont modifiées en conséquence. Par exemple, si (y,b) est trouvé comme étant un support pour (x,a) alors il est inféré que (x,a) est un support pour (y,b). Plusieurs types de règles d'inférence peuvent être utilisées comme la commutativité ou la réflexivité (cf. [BFR99]). Les complexités en temps et en espace sont en $O(d^2)$ par contrainte.

AC-Identical : C'est une extension d'AC-Inférence qui exploite le fait qu'une même contrainte peut être définie sur différentes variables. Dans ce cas, n'importe quelle information obtenue pour une contrainte est inférée pour les autres contraintes semblables.

AC-2000 : C'est une modification d'AC-3. Les valeurs pendantes sont les valeurs de $D(x)$ qui ont un support dans $\Delta(y)$. Aucune donnée supplémentaire n'est utilisée, aussi il peut être coûteux de calculer l'ensemble des valeurs pendantes. C'est pourquoi AC-2000 propose de ne calculer cet ensemble pour les valeurs pendantes que si $|\Delta(y)| < 0.2|D(x)|$; sinon $D(x)$ est considéré. On peut donc dire que AC-2000 est le premier algorithme adaptatif.

AC-2001 : Cet algorithme est basé sur AC-3 et utilise la donnée last d'AC-6. Autrement dit, les valeurs pendantes sont les mêmes que celles d'AC-3 et la fonction EXISTVALIDSUPPORT est semblable à celle d'AC-6, excepté qu'elle vérifie si la valeur last est valide. Cet algorithme hérite de la complexité en espace d'AC-6, sans utiliser les listes S-list. On notera enfin que cette présentation d'AC-2001 est originale et plus simple que celle donnée dans [BR01].

AC-3.3 : AC-3.3 est une amélioration d'AC-2001 qui associe à toute valeur (x,a) un compteur correspondant à une borne inférieure de la taille de S-list$[(x,a)]$. L'algorithme n'utilise pas les listes S-list, mais ces compteurs à la place. Lors de la recherche d'un support, le compteur de (x,a) est tout d'abord testé, s'il est strictement positif alors on sait que la valeur admet un support valide. Ce support ne peut pas être identifié mais on sait qu'il existe. Si (y,b) est supprimée alors on décrémente le compteur de toutes les valeurs supportées par (y,b).

Nous ne considérerons pas AC-8 [CJ98], AC-3_d [van02], et AC-3.2 [LBH03], parce qu'ils améliorent AC-3 principalement en proposant de propager les contraintes selon un ordre spécifique, et ce n'est pas notre propos[2].

[2]En fait cela revient à déterminer comment les variables y ayant un $\Delta(y)$ non vide vont être sélectionnée dans la fonction PROPAGATE donnée au début de cet article.

Les algorithmes d'AC peuvent donc utiliser les structures de données suivantes :

Support : le support courant de (x, a) est noté support$[(x, a)]$.

Last : la dernière valeur testée de (x, a) pour une contrainte C est représentée par last$[(x, a)]$ qui est égal à une valeur de y ou nil.

S-List, P-List, U-list : ce sont des structures de données de listes. Pour n'importe quelle liste L, nous considérerons que les fonctions ADD$(L, (x, a))$ et REMOVE$(L, (x, a))$ sont disponibles. Ces fonctions ajoutent (x, a) à L et suppriment (x, a) de L. On supposera également que ces fonctions ainsi que la fonction retournant le nombre d'éléments d'une liste peuvent être implémentées en $O(1)$.

Compteurs de tuples : ils sont représentés par counter$[(x, a)]$ qui compte le nombre de tuples de $T(C)$ contenant (x, a) et qui sont valides.

Table : Une table est l'ensemble des tuples $T(C)$ munie de deux fonctions :
FIRSTTUPLE(C, y, b) qui retourne le premier tuple de $T(C)$ contenant (y, b),
NEXTTUPLE$(C, x, y, b, a))$ qui retourne le premier tuple de $T(C)$ contenant (y, b) et suivant le tuple $((x, a), (y, b))$.
Ces fonctions retournent nil s'il n'existe pas de tuples répondant aux critères spécifiés.

Nous proposons d'identifier les différents concepts utilisés par les algorithmes d'AC existant au lieu d'avoir une fonction par algorithme et un paramètre correspondant à chaque algorithme spécifique. En procédant ainsi nous obtiendrons un algorithme configurable à partir duquel tout algorithme d'AC pourra être obtenu en combinant certains paramètres, chacun d'eux correspondant à un concept.

4 Valeurs pendantes

Trouver efficacement un petit ensemble de valeur pendantes est difficile parce que les ensemble de valeurs pendantes combinent deux différentes notions en même temps : validité et support. Aussi, de nombreux ensembles de valeurs pendants ont été considérés. Nous en avons identifié quatre :

1. Les valeurs de $D(x)$ (comme dans AC-3, AC-2001, AC-3.3). Cet ensemble est désigné par pvD.
2. Les valeurs valides supportées par une valeur de $\Delta(y)$, autrement dit les valeur valides des listes S-list de $\Delta(y)$. Cet ensemble est utilisé par AC-6, AC-7, AC-Inference, AC-Identical. Il est désigné par pvΔs.
3. Les valeurs qui appartiennent à un tuple contenant une valeur de $\Delta(y)$. Une valeur étant considérée comme pendante autant de fois qu'elle apparait dans un tel tuple. AC-4 utilise cet ensemble, qui est désigné par pvΔt.
4. Les valeurs de $D(x)$ compatibles avec au moins une valeur de $\Delta(y)$, comme dans AC-2000. Cet ensemble est désigné par pvΔc.

Comme notre but est d'obtenir un algorithme générique, nous proposons de définir un cinquième type : pvG qui représente n'importe quelle fonction définie par l'utilisateur.

Algorithm 3: Sélection de valeurs pendantes en fonction de $pvMode$ (b est une donnée interne.)

$pvMode =$ pvD

FIRSTPENDINGVALUE$(C, x, y, \Delta(y))$: return FIRST$(D(x))$
NEXTPENDINGVALUE$(C, x, y, \Delta(y), a)$: return NEXT$(D(x), a)$

$pvMode =$ pvΔs

FIRSTPENDINGVALUE$(C, x, y, \Delta(y))$: value
  $b \leftarrow$FIRST$(\Delta(y))$
  return TRAVERSES-LIST$(C, x, y, \Delta(y))$

NEXTPENDINGVALUE$(C, x, y, \Delta(y), a)$: value
  return TRAVERSES-LIST$(C, x, y, \Delta(y))$

TRAVERSES-LIST$(C, x, y, \Delta(y))$: value
  while $(y, b) \neq nil$ **do**
    $(x, a) \leftarrow$SEEKVALIDSUPPORTEDVALUE(C, y, b)
    if $(x, a) \neq nil$ **then** return (x, a)
    $b \leftarrow$NEXT$(\Delta(y), b)$
  return nil

$pvMode =$ pvΔt

FIRSTPENDINGVALUE$(C, x, y, \Delta(y))$: value
  $b \leftarrow$FIRST$(\Delta(y))$
  $(x, a) \leftarrow$FIRSTTUPLE(C, y, b)
  return TRAVERSETUPLE$(C, x, y, \Delta(y), a)$

NEXTPENDINGVALUE$(C, x, y, \Delta(y), a)$: value
  $a \leftarrow$NEXTTUPLE(C, y, b, a)
  return TRAVERSETUPLE$(C, x, y, \Delta(y), a)$

TRAVERSETUPLE$(C, x, y, \Delta(y), a)$: C-value
  while $(y, b) \neq nil$ **do**
    while $(x, a) \neq nil$ **do**
      if $a \in D(x)$ **then** return (x, a)
      $(x, a) \leftarrow$NEXTTUPLE(C, x, y, b, a)
    $b \leftarrow$NEXT$(\Delta(y), \delta a)$
    $(x, a) \leftarrow$FIRSTTUPLE$(C, x, y, \delta a)$
  return nil

$pvMode =$ pvΔc

FIRSTPENDINGVALUE$(C, y, \Delta(y))$: value
  $a \leftarrow$FIRST$(D(x))$
  return SEEKCOMPATIBLE$(C, x, y, \Delta(y), a)$

NEXTPENDINGVALUE$(C, y, \Delta(y), a)$: value
  $a \leftarrow$NEXT$(D(x), a)$
  return SEEKCOMPATIBLE$(C, x, y, \Delta(y), a)$

SEEKCOMPATIBLE$(C, x, y, \Delta(y), a)$: value
  while $(x, a) \neq nil$ **do**
    for *each* $b \in \Delta(y)$ **do**
      if $((x, a), (y, b)) \in T(C)$ **then** return (x, a)
    $(x, a) \leftarrow$NEXT$(D(x), a)$
  return nil

$pvMode =$ pvG : example of generic function : < constraint

FIRSTPENDINGVALUE$(C, y, \Delta(y))$: value
  return NEXT$(D(x), \max(D(y)) - 1)$

NEXTPENDINGVALUE$(C, y, \Delta(y), a)$: return NEXT$(D(x), a)$

Par exemple, pour $x < y$ seules les modifications de la valeur maximum de $D(y)$ peuvent entrainer des suppressions de valeurs. Aussi, les valeurs pendantes sont les valeurs de $D(x)$ qui sont plus grandes que le maximum de $D(y)$.

L'algorithme 3 est une implémentation possible du calcul des valeurs pendantes. Selon l'ensemble recherché de valeurs pendantes, l'algorithme traverse un ensemble particulier. On notera que certaines fonctions requièrent une donnée interne (une donnée dont la valeur est sauvée entre chaque appel). Nous supposerons que sont disponibles les fonctions FIRST($D(x)$) qui retourne la première valeur de $D(x)$ et NEXT($D(x), a$) qui retourne la première valeur de $D(x)$ plus grande que a et *nil* s'il n'y en a pas. La fonction SEEKVALIDSUPPORTEDVALUE(C, x, a) retourne une valeur supportée valide qui appartient à S-list[(y, b)].

Algorithm 4: Fonction SEEKVALIDSUPPORTEDVALUE

```
SEEKVALIDSUPPORTEDVALUE(C, x, a) : value
    for each value (y, b) ∈ S-list[(x, a)] do
        if b ∈ D(y) then return (y, b)
        REMOVE(S-list[(x, a)], y, b)
    return nil
```

5 Existence d'un support valide

Cette fonction différencie aussi les algorithmes d'AC existants. Presque chaque algorithme utilise une méthode différente. Nous avons identifié huit méthodes pour déterminer s'il existe un support valide pour (x, a) :

1. Tester dans le domaine à partir du début (AC-3, AC-2000).
2. Tester si la dernière valeur testée est encore valide et sinon tester dans le domaine en ne considérant que les valeurs plus grandes qu'elle (AC-2001).
3. Tester dans le domaine à partir de la dernière valeur testée (AC-6).
4. Tester si un support peut être trouvé dans S-list[(x, a)], ensuite tester dans le domaine à partir de la dernière valeur testée. Lors de ce test utiliser la notion de dernière valeur testée pour éviter certains tests de compatibilité (AC-7).
5. Tester s'il existe un support valide dans P-list[(x, a)], s'il n'y en a pas tester la compatibilité des valeur valides de U-list[(x, a)]. Quand certains tests de compatibilité sont effectués en déduire le résultat d'autres tests de compatibilité et mettre à jour les listes U-list et P-list (AC-Inference, AC-identical).
6. Décrémenter le compteur contenant le nombre de supports valides et tester si ce compteur est strictement positif (AC-4).
7. Utiliser une fonction spécifique dédiée pour la contrainte considérée.
8. Utiliser un compteur mémorisant une borne inférieure du nombre d'élément contenus dans la liste S-list[(x, a)], et tester dans le domaine à partir de la dernière valeur testée (AC-3.3).

Ce dernier point mérite une attention particulière. La complexité en temps de l'utilisation d'un compteur du nombre d'éléments contenu dans une liste est la même que celle de la gestion d'une liste. De plus, AC-3.3 implique que les compteurs soient immédiatement mis à jour lorsqu'une valeur est supprimée, ce qui n'est pas le cas d'AC-7, et l'approche

paresseuse d'AC-7 pour maintenir la consistence des listes S-list a été prouvée plus efficace [Rég95]. C'est pourquoi nous préférons maintenir explicitement les listes S-list plutôt que d'utiliser un compteur. On pourrait être tenté d'objecter qu'un compteur ne calculant pas exactement la taille des listes S-list n'a pas besoin d'être remis à jour lors du backtrack, mais on trouvera dans [Rég95] une implémentation détaillé des algorithmes de type AC-6 ou AC-7 qui montre qu'il n'est pas difficile de faire les mises à jour simplement et efficacement.

Nous proposons de considérer les paramètres suivants :

• last : la recherche d'un support valide commence à partir de la dernière valeur testée. Cette valeur est aussi utilisée pour éviter des tests négatifs (AC-6, AC-7, AC-Inference, AC-Identical, AC-2001, AC-3.3).

• inf : lors de la recherche d'un support valide, on recherche d'abord un support valide dans les listes S-list (AC-7, AC-3.3).

• slist : ce paramètre signifie que les S-list sont utilisées. Il est impliqué par inf et pvΔs (AC-6, AC-7, AC-Inference, AC-Identical, AC-3.3).

• sD : La recherche d'un support valide est faite en testant la compatibilité entre (x, a) et les valeurs de $D(y)$ (AC-3, AC-6, AC-7, AC-2000, AC-2001, AC-3.3.)

• sC : la recherche d'un support valide consiste à décrémenter le compteur de tuples valides et à tester si ce compteur est strictement positif (AC-4).

• sT : la recherche d'un support valide est faite en testant la validité des valeurs de P-list$[(x, a)]$ et en testant s'il existe une valeur dans U-list$[(x, a)]$ qui soit valide et compatible avec (x, a) (AC-Inference, AC-Identical.)

• sGen : la recherche d'un support valide est définie par une fonction fournie par l'utilisateur et dédiée à la contrainte considérée. Nous présentons un exemple pour la contrainte $x < y$.

Algorithm 5: Fonction existValidSupport

```
existValidSupport(C, x, a, y, sMode) : boolean
    if slist then remove(S-list[support[(x, a)]], (x, a))
    (y, b) ← nil
    if last then if last[(x, a)] ∈ D(y) then (y, b) ← last[(x, a)])
    if inf and (y, b) = nil then
        (y, b) ← seekValidSupportedValue(C, x, a)
    if (y, b) = nil then (y, b) ← sMode.seekSupport(C, x, a, y)
    if slist then if (y, b) ≠ nil then add(S-list[(y, b)], (x, a))
    if (y, b) ≠ nil then support[(x, a)] ← (y, b)
    return ((y, b) ≠ nil)
```

A partir de ces paramètres nous pouvons maintenant proposer un code pour la fonction existValidSupport de l'algorithme CAC (cf. Algorithme 5). Des instantiations possibles de la fonction seekSupport sont donnés dans l'algorithme 6. Un exemple est aussi proposé pour la contrainte $<$.

Algorithm 6: Fonctions recherchant un support valide

$sMode =$ sD

```
SEEKSUPPORT(C, x, a, y) : value
    b ← FIRST(D(y))
    if last then
        b ←NEXT(D(y),last[(x, a)])
        while b ≠ nil do
            if last[(y, b)] ≤ (x, a) and ((x, a), (y, b)) ∈ T(C) then
                last[(x, a)] ← (y, b)
                return (y, b)
            b ←NEXT(D(y), b)
    else
        while b ≠ nil do
            if ((x, a), (y, b)) ∈ T(C) then return (y, b)
            b ←NEXT(D(y), b)
    return nil
```

$sMode =$ sC

```
SEEKSUPPORT(C, x, a, y) : value
    counter[(x, a)] ← counter[(x, a)] − 1
    if counter[(x, a)] = 0 then return nil
    else return (y,FIRST(D(y)))
```

$sMode =$ sT

```
SEEKSUPPORT(C, x, a, y) : value
    for each (y, b) ∈ P-list[(x, a)] do
        REMOVE(P-list[(x, a)], (y, b))
        if b ∈ D(y) then return (y, b)
    for each (y, b) ∈ U-list[(x, a)] do
        REMOVE(U-list[(x, a)], (y, b))
        REMOVE(U-list[(y, b)], (x, a))
        if ((x, a), (y, b)) ∈ T(C) then
            ADD(P-list[(y, b)], (x, a))
            if b ∈ D(y) then return (y, b)
    return nil
```

$sMode =$ sGen example of generic function : < constraint

```
SEEKSUPPORT(C, x, a, y) : return false
```

6 Analyse des différentes méthodes

Le problème principal des algorithmes d'AC est de prendre en compte deux notions différentes : validité et support. Il est difficile de manipuler ces deux notions en même temps. Aussi les algorithmes privilégient habituellement une des deux notions au détriment de l'autre :

• Lors de la construction de l'ensemble des valeurs pendantes, les algorithmes de type pvD ignorent totalement la notion de support. Les autres algorithmes essaient de combiner les deux notions : les algorithmes de type pvDc considèrent d'abord la validité, tandis que les algorithmes de type pvΔt considèrent tous les supports, et les algorithmes de type pvΔs traversent les valeurs supportées pour tester leur validité.

• Lors de la recherche d'un nouveau support, les algorithmes de type sD considèrent les valeurs valides et vérifient si ce sont des supports, alors que les algorithmes de type sT traversent les supports et vérifient leur validité.

7 Nomenclature

A partir des différent concepts que nous avons identifiés nous proposons une nouvelle nomenclature des algorithms d'AC. Jusqu'à présent les algorithmes étaient nommés en utilisant le prefixe "AC-" suivi d'un nombre ou d'une date. Il est clairement impossible de comprendre les spécificités des algorithmes à partir de leur nom. Seuls AC-Inference et AC-Identical ont essayé d'exprimer par leurs noms quelques idées de l'algorithme.

La nomenclature que nous proposons utilise CAC comme préfixe qui signifie Configurable Arc Consistency algorithm en anglais. La combinaison de paramètres définissant un algorithme est ensuite ajoutée au préfixe CAC. Par exemple, CAC-pvD-sD signifie que les valeurs pendantes sont les valeurs de $D(x)$ et qu'un support valide est recherché dans le domaine en testant la compatibilité entre valeurs. C'est la description exacte d'AC-3. Pour les parties adaptatives de l'algorithme on utilisera le symbole "/" afin de différencier les possibilités : AC-2000, par exemple, est renommé CAC-pvΔc/pvD-sD. Nous pouvons ainsi décrire tous les algorithmes existants :

name	new name
AC-3	CAC-pvD-sD
AC-4	CAC-pvΔt-sC
AC-6	CAC-pvΔs-last-sD
AC-7	CAC-pvΔs-last-inf-sD
AC-Inference or AC-Identical	CAC-pvΔs-sT
AC-2000	CAC-pvΔc/pvD-sD
AC-2001	CAC-pvD-last-sD
AC-3.3	CAC-pvD-last-inf-sD

On remarquera CAC-pvΔs-last-sD est une amélioration légère d'AC-6 puisque certains tests négatifs sont évités en utilisant la donnée last lors du test de compatibilité entre valeurs.

8 Algorithme adaptatif

L'avantage d'un algorithme adaptatif est d'éviter certains cas pathologiques de chaque algorithme. Une propriété permettant de différencier exactement AC-2001 et AC-6 en fonction du nombre d'opérations réalisé par chaque algorithme a été donné par [BR01] :

Property 1 *Le nombre de valeurs qui sont considérées pour trouver les valeurs pendantes dans :*

- *un algorithme de type* $\underline{\mathsf{pvD}}$ *est* $\#(pvD) = |D(x)|$.
- *un algorithme de type* $\underline{\mathsf{pv\Delta s}}$ *est*

$$\#(pv\Delta s) = |\Delta(y)| + \sum_{b \in \Delta(y)} |S\text{-}list[(y,b)]|$$

Ces deux nombres sont suffisants pour différencier AC-2001 et AC-6 parce qu'ils utilisent tous les deux le même algorithme pour trouver un support valide. Cela est clairement montré par leurs nouveaux noms qui sont respectivement CAC-pvD-last-sD et CAC-pvΔs-last-sD. Donc, en considérant la méthode qui pour calculer les valeurs pendantes, étudie le plus petit nombre de valeur nous pouvons définir un algorithme qui sera meilleur que n'importe lequel des deux. Nous pouvons utiliser d'abord un algorithme de type $\underline{\mathsf{pv\Delta s}}$ puis un algorithme de type $\underline{\mathsf{pvD}}$ et réciproquement.

Malheureusement, il est difficle de calculer rapidement $\#(pv\Delta s)$. En effet, la somme considère indépendamment toute valeur du delta domaine. Cependant, on a $\#(pv\Delta s) \geq 2|\Delta(y)|$, et $|\Delta(y)|$ peut être maintenu incrémentalement, aussi nous pouvons supposé que cette valeur est disponible en $O(1)$. L'algorithme 7 est une implémentation possible des fonctions sélectionnant les modes $sMode$ et $pvMode$ qui sont utilisés par l'algorithme 2 (AC-2000 est pris en compte dans cette fonction).

Algorithm 7: Sélection de pvMode et sMode

```
selectPendingValueMode(C,x,y,Δ(y),pvMode) :pvMode
    if pvMode=pvΔc/pvD then
        if |Δ(y)| < 0.2|D(x)| then return pvΔc
        return pvD
    if pvMode=pvD/pvΔs then
        if |D(x)| < 2.|Δ(y)| then return pvD
        if |D(x)|<|Δ(y)|+Σ_{b∈Δ(y)} |S-list[(y,b)]| then return pvD
        return pvΔs
    return pvMode
selectExistSupportMode(C, x, a, sMode) : sMode
    if sMode=sD/sT then
        if |D(x)| < |P-list[(x,a)]| then return sD
        return sT
    return sMode
```

Passer d'un algorithme à un autre peut entrainer certains problèmes, parce que les algorithmes de types différents n'utilisent pas les mêmes structures de données. Lorsque l'on passe d'un algorithme utilisant une certaine structure de données vers un algorithme ne l'utilisant pas, nous avons deux possibilités : soit la structure de données est mise à jour après un changement d'algorithme, soit elle est systématiquement mise à jour même si elle n'est pas utilisée. Pour les listes S-list, le coût de leur maintenance est $O(1)$ par suppression ou ajout, aussi la seconde solution est la plus simple. Pour les liste U-list

et P-list il n'y a pas de problème puisqu'elle n'ont pas besoin d'être mises à jour (cf. [Rég95]).

Nous avons vu qu'il était possible de changer la méthode qui calcule les valeurs pendantes. Deux autres possibilités sont : l'utilisation ou pas du paramètre inf et le changement de type sD vers sT et inversement. Il est beaucoup plus compliqué d'exhiber dans ce cas un critère de sélection entre ces deux types d'algorithmes, parce que les listes sont modifiés par les paramètres inf et sT. Aussi, à un moment donné les tailles des listes peuvent être en faveur d'un type d'algorithme puis devenir fortement en défaveur de ce type après application de l'algorithme. Après diverses expériences il apparaît que le passage d'un algorithme de type inf vers un algorithme n'utilisant pas inf ne change pas grand chose, et il apparaît qu'il est intéressant de passer de sD vers sT et réciproquement. Si la taille du domaine est plus petite que la taille de la liste P-list alors un algorithme de type sD est préféré, sinon on choisira plutôt un algorithme de type sT (cf. algorithme 7).

9 Résultats expérimentaux

Nous proposons de comparer les versions MAC des algorithmes sur le benchmark bien connu des problèmes d'affectation de fréquences (RLFAP). Nous remerçions à ce propos le Centre Electronique de l'Armement. Nous donnons des résultats uniquement pour les instances SCEN#1, SCEN#11, SCEN#8 parce que ces résultats sont tout à fait représentatifs. Aucun ordre spécifique de traitement des contraintes n'est considéré (tous les algorithmes considèrent les mêmes ordres). Pour chaque algorithme nous donnons le rapport entre le temps pris par l'algorithme considéré et celui pris par le meilleur algorithme.

pvΔt	•											
pvΔc			•									
pvΔs						•	•	•	•	•	•	•
pvD		•	•	•	•			•	•		•	•
last				•	•	•	•	•	•		•	•
inf					•		•		•			
sD		•	•	•	•	•	•	•	•			•
sC	•											
sT										•	•	•
#1	19.4	4.2	3.7	2.8	2.4	3.7	3.7	3.2	3.2	1.5	1.1	1
#11	15	7	6.8	4	3.3	2.6	2.5	1.8	1.8	1.7	1.1	1
#8	59.3	68.2	67.3	50	48	21	19	4	3	4	1.1	1

Ces résultats montrent clairement que les versions adaptatives de CAC donnent de bien meilleurs résultats que celles qui ne le sont pas. Il apparaît également que les algorithmes de type sT sont plus performants que les autres.

10 Conclusion

Nous avons présenté CAC un nouvel algorithme d'AC. CAC est configurable, générique et adaptatif. CAC peut représenter tous les algorithmes existants. Nous avons clairement

identifié les concepts les plus importants des algorithmes d'AC, et nous avons étudié de nouvelles combinaisons de concepts qui donnent de bons résultats en pratique.

Références

[Bes94] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1) :179–190, 1994.

[BFR99] C. Bessière, E.C. Freuder, and J-C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1) :125–148, 1999.

[BR01] C. Bessière and J-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, Seattle, WA, USA, 2001.

[CJ98] A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2) :79–89, 1998.

[LBH03] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionnality in coarse-grained arc consistency algorithm. In *Proceedings CP'03*, pages 480–494, Cork, Ireland, 2003.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8 :99–118, 1977.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28 :225–233, 1986.

[Rég95] J-C. Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.* PhD thesis, Université de Montpellier II, 1995.

[van02] M.R. van Dongen. Ac-3d an efficient arc-consistency algorithm with a low space-complexity. In *Proceedings CP'02*, pages 755–760, Ithaca, NY, USA, 2002.

[VDT92] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57 :291–321, 1992.