ACID : Disjonction constructive adaptative sur intervalles

Bertrand Neveu, Gilles Trombettoni

ACID : Disjonction constructive adaptative sur intervalles

Bertrand Neveu[1] **Gilles Trombettoni**[2]

[1] LIGM, Université Paris Est, France
[2] LIRMM, Université Montpellier 2, France
Bertrand.Neveu@enpc.fr Gilles.Trombettoni@lirmm.fr

Résumé

Un opérateur de filtrage nommé CID et une variante efficace 3BCID ont été proposés en 2007. Ces opérateurs calculent pour des CSP numériques traités avec des méthodes à intervalles une consistance partielle équivalente à Partition-1-AC pour les CSP à domaines finis. Les deux paramètres principaux de CID sont le nombre d'appels à la procédure principale et le nombre maximum de sous-intervalles traités par cette procédure. Cet opérateur 3BCID est efficace pour la résolution de CSP numériques, mais l'est moins en optimisation globale sous contraintes.

Cet article propose une variante adaptative de 3BCID. Le nombre de variables traitées est calculé automatiquement durant la recherche, les autres paramètres étant fixés de manière robuste. Sur un échantillon représentatif d'instances, ACID apparaît comme la meilleure approche en résolution et en optimisation. Elle a été choisie comme méthode de contraction par défaut du résolveur par intervalles Ibex.

Abstract

An operator called CID and an efficient variant 3BCID were proposed in 2007. For numerical CSPs handled by interval methods, these operators compute a partial consistency equivalent to Partition-1-AC for discrete CSPs. The main two parameters of CID are the number of times the main CID procedure is called and the maximum number of sub-intervals treated by the procedure. The 3BCID operator is state-of-the-art in numerical CSP solving, but not in constrained global optimization.

This paper proposes an adaptive variant of 3BCID. The number of variables handled is auto-adapted during the search, the other parameters are fixed and robust to modifications. On a representative sample of instances, ACID appears to be the best approach in solving and optimization, and has been added to the default strategies of the Ibex interval solver.

1 Disjonction constructive sur intervalles (CID)

Un opérateur de filtrage/contraction pour les *CSP numériques* appelé *Constructive Interval Disjunction* (CID) a été proposé dans [13]. Appliqué d'abord aux problèmes de satisfaction traités par les méthodes à intervalles, l'opérateur a été appliqué plus récemment en optimisation globale sous contraintes. Cet algorithme détient les performances de référence en résolution, mais est généralement surpassé en optimisation par les algorithmes plus simples de propagation de contraintes comme HC4. La principale contribution pratique de cet article est de montrer qu'une version auto-adaptative de CID peut devenir efficace à la fois en résolution et en optimisation, et ce, sans ajouter de paramètres utilisateurs.

1.1 Rognage

Le principe de rognage (*shaving* en anglais) est utilisé pour calculer *Singleton Arc Consistency* (SAC) en domaines finis [7] et la 3B-consistance sur les CSP continus [9]. Le rognage est également au cœur de l'algorithme SATZ [11] utilisé pour prouver la satisfiabilité d'une formule booléenne. Le rognage fonctionne de la manière suivante. Une valeur est temporairement affectée à une variable (les autres valeurs étant temporairement supprimées du domaine) et une consistance est calculée sur le sous-problème correspondant. Si une inconsistance est détectée, alors la valeur peut être supprimée définitivement du domaine de la variable. Sinon la valeur est maintenue dans le domaine.

Contrairement à l'arc-consistance, cette consistance n'est pas incrémentale [7]. En effet, le travail de la procédure de réfutation porte sur toutes les variables d'un

sous-problème afin de supprimer une seule valeur d'un domaine. C'est pourquoi calculer SAC en domaines finis fait appel à un algorithme de point-fixe où toutes les variables doivent être traitées à nouveau à chaque retrait d'une valeur [7]. La remarque tient aussi pour la version améliorée `SAC-Opt` [5].

Le même principe de rognage peut être suivi pour les CSP numériques (NCSP).

1.2 CSP numérique

Un NCSP est défini par un triplet $P = (X, [X], C)$, où X désigne un n-ensemble de variables numériques, à valeurs réelles dans un domaine $[X]$. On note $[x_i] = [\underline{x_i}, \overline{x_i}]$ le domaine/intervalle de la variable $x_i \in X$, où $\underline{x_i}$ et $\overline{x_i}$ sont des nombres flottants (permettant d'implanter les algorithmes sur un ordinateur). Une solution de P est un n-vecteur dans $[X]$ qui satisfait toutes les contraintes de C. Les contraintes C définies dans un NCSP sont numériques. Ce sont des équations et des inégalités comprenant des opérateurs mathématiques comme $+$, $\cdot$, $/$, exp, log, sin.

On appelle *boîte* (parallèle aux axes) un produit cartésien d'intervalles, comme le domaine $[X] = [x_1] \times ... \times [x_n]$. $w(x_i)$ dénote la *largeur* $\overline{x_i} - \underline{x_i}$ d'un intervalle $[x_i]$. La largeur d'une boîte est donnée par la largeur $\overline{x_m} - \underline{x_m}$ de sa plus grande dimension x_m. L'union de plusieurs boîtes n'est généralement pas une boîte si bien que l'on utilise plutôt un opérateur d'enveloppe (*Hull*) pour calculer la plus petite boîte comprenant toutes les boîtes traitées.

Les NCSPs sont généralement résolus par une stratégie de type Brancher & Contracter à intervalles :

- **Brancher :** une variable x_i est choisie et son intervalle $[x_i]$ est divisé en deux sous-intervalles ; les deux sous-boîtes ainsi construites sont traitées, ce qui rend combinatoire l'ensemble du processus de résolution.
- **Contracter :** un processus de filtrage permet de contracter les intervalles (i.e., améliorer les bornes des intervalles) sans perte de solutions.

Le processus commence avec un domaine initial $[X]$ et s'arrête quand la largeur des feuilles/boîtes de l'arbre de recherche atteint une largeur inférieure à une précision donnée en entrée. Ces feuilles fournissent une approximation de toutes les solutions du NCSP.

Plusieurs algorithmes de contraction ont été proposés. Mentionnons l'algorithme de propagation de contraintes appelé `HC4` [3, 10], une implantation efficace de l'algorithme 2B [9] qui calcule une consistance locale optimale (enveloppe-consistance – *hull-consistency*) seulement si des hypothèses fortes sont réunies (en particulier, chaque variable doit apparaître au plus une fois dans une même contrainte). La procédure 2B-Revise travaille avec toutes les *fonctions de projection* d'une contrainte donnée. Informellement, une fonction de projection isole une occurrence donnée d'une variable dans l'expression de la contrainte. Par exemple, considérons la contrainte $x + y = z.x$; $x \leftarrow z.x - y$ est une fonction de projection (parmi d'autres) qui vise à réduire le domaine de la variable x. Evaluer la fonction de projection, en utilisant l'arithmétique des intervalles, sur le domaine $[x] \times [y] \times [z]$ (i.e., remplacer les occurrences de variable de la fonction de projection par leurs domaines et utiliser l'extension aux intervalles des opérateurs mathématiques impliqués) produit un intervalle image intersecté ensuite avec $[x]$. D'où une réduction potentielle du domaine. Une boucle de propagation proche de AC3 permet alors de propager les réductions obtenues pour un domaine de variable donné vers d'autres contraintes du système.

1.3 L'algorithme 3B

Des consistances plus fortes ont également été proposées. La 3B-consistance [9] est une consistance partielle similaire à SAC pour CSP quoique limitée aux bornes du domaine. Considérons les $2n$ sous-problèmes d'un NCSP donné où chaque intervalle $[x_i]$ ($i \in \{1..n\}$) est réduit à sa borne inférieure $\underline{x_i}$ (resp. borne supérieure $\overline{x_i}$). La 3B-consistance est vérifiée ssi chacun des $2n$ sous-problèmes est enveloppe-consistant.

En pratique, l'algorithme `3B`(w) subdivise les domaines en plusieurs sous-intervalles, également appelés tranches, de largeur w, largeur qui correspond à une précision : la 3B(w)-consistance est vérifiée ssi les tranches aux bornes de la boîte traitée ne peuvent pas être éliminées par `HC4`. Appelons `var3B` la procédure de l'algorithme de `3B` chargée de rogner un intervalle $[x_i]$. Le paramètre s_{3b} de `var3B` est un entier positif spécifiant un nombre de sous-intervalles : $w = w(x_i)/s_{3b}$ donne la largeur d'un sous-intervalle.

1.4 L'algorithme CID

La disjonction constructive sur intervalles (*Constructive Interval Disjunction* – `CID`) est une consistance plus forte que la 3B-consistance [13]. La CID-consistance est similaire à Partition-1-AC dans les CSP à domaines finis [4]. Partition-1-AC est strictement plus forte que SAC [4].

La procédure principale `varCID` traite une variable x_i. Les paramètres principaux de `varCID` sont x_i, un nombre s_{cid} de sous-intervalles (précision) et un algorithme de contraction *ctc*, comme `HC4`. $[x_i]$ est subdivisé en s_{cid} tranches de taille égale ; chaque sous-problème correspondant est traité par le contracteur *ctc* et on renvoie finalement l'enveloppe des différentes boîtes contractées, comme le détaille l'algorithme 1.

Procedure `VarCID` *(x_i, s_{cid}, (X, C, in-out $[X]$), ctc)*
  $[X]' \leftarrow$ *empty box*
  for $j \leftarrow 1$ *to* s_{cid} **do**
    /* La j^e sous-boîte de $[X]$ sur x_i est traitée : */
    `sliceBox` $\leftarrow$ `SubBox` (j, x_i, $[X]$)
    /* Calcul d'une consistance sur la sous-boîte : */
    `sliceBox'` $\leftarrow$ *ctc*(X, C, `sliceBox`)
    /* "Union" avec les sous-boîtes précédentes : */
    $[X]' \leftarrow$ `Hull`($[X]'$, `sliceBox'`)
  $[X] \leftarrow [X]'$

Algorithm 1: La procédure principale de l'opérateur `CID` pour le rognage du domaine d'une variable x_i.

Intuitivement, `CID` généralise `3B` puisque une sous-boîte éliminée par `var3B` serait aussi éliminée par `varCID`. De plus, contrairement à `var3B`, `varCID` peut aussi contracter $[X]$ sur *plusieurs* dimensions.

Notons que, dans l'implémentation effective, la boucle peut être interrompue plus tôt si $[X]'$ devient égale à la boîte initiale $[X]$ dans toutes les dimensions, exceptée x_i.

`var3BCID` est une variante (hybride) opérationelle de `varCID`.

1. Comme `var3B`, elle cherche d'abord à éliminer des sous-intervalles aux bornes de $[x_i]$, sous-intervalles de largeur $w = w(x_i)/s_{3b}$ chacun. On conserve les boîtes gauche $[X_l]$ et droite $[X_r]$ qui ne sont pas exclues par le contracteur *ctc* (s'il y en a).
2. Ensuite, la boîte restante $[X]'$ est traitée par `varCID` qui subdivise $[X]'$ en s_{cid} sous-boîtes. Les sous-boîtes sont contractés par *ctc*, leur enveloppe (hull) donnant $[X_{cid}]$.
3. Finalement, on renvoie l'enveloppe de $[X_l]$, $[X_r]$ et $[X_{cid}]$.

Le fonctionnement de `var3BCID` est illustré par la figure 1.

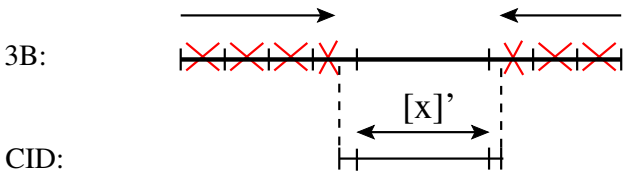

FIGURE 1 – Fonctionnement de la procédure `var3BCID`. La valeur 10 est choisie pour le paramètre s_{3b} et la valeur 1 est choisie pour s_{cid}.

`var3BCID` repose sur la volonté de gérer différentes largeurs de domaines (précisions) pour s_{3b} et s_{cid}. En effet, le meilleur choix pour s_{3b} se situe généralement dans la fourchette $\{5..20\}$ tandis que s_{cid} doit être quasiment toujours limité à 1 ou 2 (ce qui entraîne une enveloppe finale de 3 ou 4 sous-boîtes). La raison en est que le coût effectif en temps de la partie rognage est plus faible que celui de la disjonction constructive de domaine. En effet, si aucun sous-intervalle d'un domaine n'est éliminé par `var3B`, alors seulement deux appels à *ctc* sont réalisés, un pour chaque borne de l'intervalle traité ; alors que lorsque `varCID` est appliqué, le sous-contracteur est souvent appliqué s_{cid} fois.

La procédure `var3BCID` a été étudiée et expérimentée de manière approfondie dans le passé. Le nombre et l'ordre dans lequel les appels à `var3BCID` sont réalisés est une question plus difficile étudiée dans cet article.

2 CID adaptatif : apprendre le nombre de variables traitées

Comme dans `SAC` ou `3B`, un point quasi-fixe en terme de contraction peut être atteint par `3BCID` (ou `CID`) en appelant `var3BCID` dans deux boucles imbriquées. Une boucle intérieure appelle `var3BCID` sur chaque variable x_i. Une boucle extérieure appelle la boucle intérieure jusqu'à ce qu'aucun intervalle ne soit contracté plus que d'une certaine précision/largeur (atteignant ainsi un point quasi-fixe). Appelons `3BCID-fp` (*fixed-point*) cette version "historique".

Deux raisons nous ont amenés à changer radicalement cette gestion. D'abord, comme dit plus haut, `var3BCID` peut contracter la boîte traitée dans plusieurs dimensions. Un avantage significatif est que le point quasi-fixe en terme de contraction peut ainsi être atteint en un petit nombre d'appels à `var3BCID`. Sur la plupart des instances en satisfaction ou optimisation, il s'avère que le point quasi-fixe est obtenu en moins de n appels. Dans ce cas, `3BCID` est clairement trop coûteux. Ensuite, le principe derrière `varCID` est proche d'un point de choix dans un arbre de recherche. La différence est que l'on calcule une enveloppe des sous-boîtes obtenues après contraction (par *ctc*). C'est pourquoi une idée est d'utiliser une heuristique standard de branchement pour sélectionner la prochaine variable à "varcider". (Nous écrirons par la suite qu'une variable est *varcidée* quand la procédure `var3BCID` est appelée sur cette variable pour contracter la boîte courante.)

Pour résumer, l'idée pour rendre `3BCID` bien plus efficace en pratique est de remplacer les deux boucles imbriquées par une seule boucle appelant *numVarCID* fois la procédure `var3BCID` et d'utiliser une variante efficace de l'heuristique de branchement basée sur la fonction *Smear* pour sélectionner les variables à varcider (heuristique appelée `SmearSumRel` dans [12]). Informellement, la fonction Smear favorise

les variables avec un large domaine et un fort impact sur les contraintes – en mesurant les dérivées partielles (sur intervalles).

Une première idée est de fixer *numVarCID* au nombre n de variables. Appelons `3BCID-n` cette version. Elle produit de bons résultats en satisfaction mais est dominé par la propagation de contraintes pure en optimisation. Comme dit ci-dessus, cette approche est trop coûteuse quand la valeur optimale de *numVarCID* est inférieure à n (ce qui est souvent le cas en optimisation), mais peut aussi avoir un impact négatif sur les performances si un effort plus grand pouvait apporter un filtrage significativement plus grand.

Le but d'un algorithme `CID` adaptatif (`ACID`) est précisément de calculer dynamiquement pendant la recherche la valeur du paramètre *numVarCID*. Plusieurs politiques d'auto-adaptation ont été testées et nous décrivons trois versions intéressantes. Toutes ces politiques mesurent le gain en taille de l'espace de recherche après chaque appel à `var3BCID`. Ils mesurent un *ratio de contraction* d'une boîte $[X]^b$ par rapport à une boîte $[X]^a$ comme un gain moyen relatif dans toutes les dimensions :

$$\texttt{gainRatio}([X]^b,[X]^a) = \frac{1}{n}\sum_{i=1}^{n}(1-\frac{w(x_i^b)}{w(x_i^a)})$$

2.1 ACID0 : auto-adapter numVarCID pendant la recherche

La première version `ACID0` adapte le nombre de variables rognées dynamiquement à chaque nœud de l'arbre de recherche. Les variables sont tout d'abord triées selon leur impact, calculé par la même formule que pour l'heuristique de branchement `SmearSumRel`. Les variables sont ensuite rognées jusqu'à ce que le ratio de contraction cumulée pendant les nv derniers appels à `var3BCID` devienne inférieur à *ctratio*. Cet algorithme a donc 2 paramètres nv and *ctratio* qui se sont avérés assez difficiles à régler. Les expérimentations ont montré qu'on pouvait fixer *ctratio* à 0.001 que nv dépendait du nombre de variables n du problème. Fixer nv avec la formule $nv = max(3, \frac{n}{4})$ a donné les meilleurs résultats. Les résultats expérimentaux ne sont pas mauvais, mais cette approche ne permet pas à `numVarCID` d'atteindre 0, c.-à-d. de n'appeler que la propagation de contraintes. C'est un inconvénient majeur quand une simple propagation est la méthode la plus efficace.

2.2 ACID1 : succession de phases d'apprentissage et d'exploitation

Une approche plus sophistiquée pallie cet inconvénient. `ACID1` alterne des phases d'apprentissage et d'exploitation pour l'auto-adaptation de la valeur *numVarCID*. Selon le numéro du nœud courant, l'algorithme est dans une phase d'apprentissage ou d'exploitation. Le comportement de `ACID1`, décrit par l'algorithme 2, est le suivant :

- Les variables sont triées selon leur impact (mesuré par la formule de l'heuristique `SmearSumRel`).
- Pendant une phase d'apprentissage (pendant *learnLength* nœuds), nous analysons l'évolution du ratio de contraction d'un appel à `var3BCID` au suivant et enregistrons le nombre *kvarCID* de rognages nécessaire pour obtenir l'essentiel du filtrage.
  Nous n'atteignons pas nécessairement le point-fixe en terme de filtrage à chaque nœud, et nous nous limitons à un nombre $2.numVarCID$ de variables rognées (avec un minimum égal à 2). Dans la première phase d'apprentissage, nous rognons n variables, c'est-à-dire que nous effectuons le premier appel à `ACID1` avec *numVarCID* $= 0.5\,n$.
  Pour le nœud courant, la fonction `lastSignificantGain` renvoie un nombre *kvarCID* de variables rognées, le $kvarCID^{eme}$ appel ayant donné la dernière contraction significative. Après cet appel à `var3BCID`, le gain sur la boîte courante produit par un appel à `var3BCID` et calculé par la formule `gainRatio`, n'excède jamais un ratio donné, appelé *ctratio*. Cette analyse commence par la dernière variable rognée. (Pour la lisibilité du pseudo-code, nous avons omis les paramètres de la procédure `var3BCID`, c.-à-.d. s_{3b}, s_{cid}, les contraintes C et le contracteur *ctc*.)
- Pendant la phase d'exploitation suivant une phase d'apprentissage, la moyenne des différentes valeurs *kvarCID* (obtenues dans les nœuds de la phase d'apprentissage) fournit la nouvelle valeur de *numVarCID*. Cette valeur sera utilisée pendant toute la phase d'exploitation. On notera que cette valeur peut au plus être le double de celle de la précédente phase d'exploitation, mais peut aussi significativement baisser.

Tous les `cycleLength` nœuds dans l'arbre de recherche, les 2 phases sont réappelées.

De nombreuses variantes de ce schéma ont été testées. En particulier, il est apparu contre-productif d'avoir une seule phase d'apprentissage et donc d'apprendre *numVarCID* une seule fois, ou au contraire de mémoriser les calculs d'une phase d'apprentissage à la suivante.

Nous avons fixé expérimentalement les 3 paramètres de la procédure `ACID1`, à savoir `learnLength`, `cycleLength` et `ctratio`, respectivement à 50, 1000 et

```
Procedure ACID1 (X, n, in-out [X], in-out call, in-out numVarCID)
    learnLength ← 50
    cycleLength ← 1000
    ctratio ← 0.002
    /* Tri des variables selon leur impact */
    X ← smearSumRelSort (X)
    if call % cycleLength ≤ learnLength then
        /* Phase d'apprentissage */
        nvarCID ← max(2, 2 . numVarCID)
        for i from 1 to nvarCID do
            [X]^old ← [X]
            var3BCID (X[i%n], [X], ...)
            ctcGains[i] ← gainRatio( [X], [X]^old)
        kvarCID[call] ← lastSignificantGain (ctcGains, ctratio, nvarCID)
        if call % cycleLength = learnLength then
            /* Fin de la phase d'apprentissage */
            numVarCID ← average (kvarCID[])
    else
        /* Phase d'exploitation */
        if numVarCID > 0 then
            for i from 1 to numVarCID do
                var3BCID (X[i % n], [X], ...)
    call ← call + 1
```

Algorithm 2: Algorithm ACID1

```
Function lastSignificantGain(ctcGains, ctratio, nvarCID)
    for i from nvarCID downto 1 do
        if (ctcGains[i] > ctratio) then
            return i
    return 0
```

0.002. ACID1 est donc devenue une procédure sans paramètres. Avec ces valeurs, le surcoût des phases d'apprentissage (pendant lesquelles la valeur précédente de *numVarCID* est doublée) reste faible.

2.3 ACID2 : Prise en compte de la profondeur dans l'arbre de recherche

Une critique peut être formulée envers ACID1 : la moyenne kvarCID est faite avec des valeurs obtenues à différentes profondeurs de l'arbre de recherche. Cet inconvénient est partiellement corrigé par les phases d'apprentissage successives de ACID1, où chaque phase correspond à une partie de l'arbre de recherche.

Pour aller plus loin, nous avons conçu un raffinement de ACID1 où chaque phase d'apprentissage règle plusieurs valeurs différentes suivant la largeur de la boîte étudiée. Une valeur correspond à un ordre de grandeur dans la largeur de la boîte. Par exemple, nous déterminons une valeur *numVarCID* pour les boîtes ayant une largeur comprise entre 1 et 0.1, une autre pour les boîtes avec une largeur entre 0.1 et 0.01, etc. Finalement, cette approche, appelée ACID2, a donné en général des résultats similaires à ceux de ACID1 et est apparu moins robuste. En effet, à certains niveaux de largeur de boîte, le réglage a été effectué sur quelques nœuds seulement, ce qui le rend peu significatif.

3 Expérimentations

Tous les algorithmes ont été implantés dans la bibliothèque logicielle de résolution par intervalles en C++ Ibex (Interval Based EXplorer) [6]. Les expérimentations ont été effectuées sur la même machine (Intel X86 3GHz). Nous avons testé les algorithmes sur le résolution de NCSP carrés et sur des problèmes d'optimisation globale sous contraintes. Résoudre un NCSP consiste à trouver toutes les solutions d'un système bien contraint de n équations non linéaires portant sur n variables réelles bornées. L'optimisation globale consiste à trouver le minimum global d'une fonction à n variables, sous des contraintes (équations et inégalités), la fonction objectif et/ou les contraintes étant non convexes.

3.1 Expérimentations en satisfaction de contraintes

Nous avons sélectionné dans le banc d'essai COPRIN [1] tous les systèmes qui pouvaient être résolus par un des algorithmes testés dans un temps compris entre 2 s et 3600 s. La limite de temps a été fixée à 10000 s. La précision requise sur une solution est 10^{-8}. Certains de ces problèmes peuvent avoir une taille (nombre de variables) à fixer. Dans ce cas, nous avons choisi la plus grande taille pouvant être résolue par un des algorithmes en moins d'une heure.

Nous avons comparé notre méthode ACID et ses variantes avec les techniques de filtrage bien connues : une simple propagation de contraintes HC4, 3BCID-n (voir partie 2) et 3BCID-fp (point fixe) pour laquelle on lance une nouvelle itération sur toutes les variables quand un domaine est réduit de plus d'1%. À chaque nœud de l'arbre de recherche, nous avons utilisé la séquence suivante de contracteurs : HC4, *shaving*, Interval-Newton [8] et X-Newton [2]. *shaving* dénote une variante de ACID, 3BCID-n, 3BCID-fp, ou rien si HC4 seul est testé.

1. www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html

TABLE 1 – Résultats obtenus par ACID1 en résolution de CSP continus. Pour chaque problème, sont indiqués son nombre de variables et les résultats obtenus par ACID1 : le temps de calcul, le nombre de branchements, le nombre moyen de variables rognées (déterminé dynamiquement par ACID1). Nous indiquons aussi la meilleure et la plus mauvaise méthode parmi ACID1, HC4, 3BCID-fp, et 3BCID-n, le rapport en temps de calcul entre ACID1 et la meilleure méthode et entre ACID1 et la pire méthode.

	#var	ACID1	ACID1	ACID1	meilleur	pire	ratio en temps	ratio en temps
		temps	#nœuds	#varcids			ACID1/meilleur	ACID1/pire
Bellido	9	3.45	518	5	ACID1	HC4	1	0.89
Brown-7	7	396	540730	4.5	ACID1	HC4	1	0.82
Brent-10	10	17.6	3104	9	ACID1	HC4	1	0.14
Butcher8a	8	981	204632	9	3BCID-n	HC4	1.03	0.49
Butcher8b	8	388	93600	10.8	ACID1	HC4	1	0.31
Design	9	29.2	5330	11	3BCID-n	HC4	1.07	0.37
Dietmaier	12	926	82364	26.3	ACID1	HC4	1	0.19
Directkin	11	32.7	2322	7	ACID1	3BCID-fp	1	0.84
Disc.integralf2-16	32	592	58464	0.4	HC4	3BCID-fp	1.02	0.52
Eco-12	11	3156	297116	12	ACID1	HC4	1	0.32
Fredtest	6	25.2	11480	0.8	HC4	3BCID-fp	1.04	0.91
Fourbar	4	437	183848	0.1	ACID1	3BCID-n	1	0.85
Geneig	6	178	83958	2.9	HC4	3BCID-fp	1.02	0.82
Hayes	7	3.96	1532	7.5	3BCID-n	HC4	1.14	0.77
I5	10	15.9	3168	11.5	ACID1	HC4	1	0.13
Katsura-25	26	691	5396	10.4	ACID1	3BCID-fp	1	0.67
Pramanik	3	23.1	23696	0.2	ACID1	HC4	1	0.69
Reactors-42	42	1285	23966	134	3BCID-fp	HC4	1.07	0.13
Reactors2-30	30	1220	38136	90	3BCID-n	HC4	1.14	0.12
Synthesis	33	356	7256	53.8	3BCID-fp	HC4	1.15	0.25
Trigexp2-23	23	2530	227136	39.4	3BCID-fp	HC4	1.26	0.25
Trigo1-18	18	2625	37756	6.1	ACID1	3BCID-fp	1	0.8
Trigo1sp-35	36	2657	70524	2.4	ACID1	3BCID-fp	1	0.41
Virasoro	8	1592	266394	0.6	3BCID-n	3BCID-fp	1.08	0.28
Yamamura1-16	16	2008	68284	0.4	3BCID-n	HC4	1.02	0.86
Yamamura1sp-500	501	1401	146	144	ACID1	HC4	1	0.14

Pour chaque problème, nous utilisons la meilleure heuristique de bissection parmi deux variantes de la fonction Smear [12]). Le paramètre principal *ctratio* de ACID1 et ACID2, mesurant une stagnation dans le contraction quand les variables sont rognées, a été fixé à 0.002. Les paramètres de var3BCID, s_{3b} and s_{cid}, ont été fixés à leur valeur par défaut, respectivement 10 et 1, proposées dans [13]. Des expérimentations sur les instances sélectionnées ont confirmé que ces valeurs étaient pertinentes et robustes. En particulier, fixer s_{3b} à 10 donne de meilleurs résultats que des valeurs plus petites ($s_{3b} = 5$) ou plus grandes ($s_{3b} = 20$). Pour 21 des 26 instances, $s_{3b} = 20$ a donné de plus mauvais résultats. Comme on peut le voir sur le tableau 1, ACID1 apparaît souvent comme le meilleur, ou proche du meilleur. Sur seulement 4 problèmes sur 26, il est plus de 10% plus lent que le meilleur. Le nombre de variables rognées a été réglé près de 0 dans les problèmes où HC4 suffisait, et à plus que le nombre de variables dans les problèmes où 3BCID-fp est apparue être la meilleure méthode.

Sur la partie gauche du tableau 2, on a résumé les résultats obtenus par les 3 variantes de ACID et leurs concurrents.

ACID1 est le seul qui a pu résoudre les 26 problèmes en 1 heure, tandis qu'HC4 n'a pu résoudre que 21 problèmes en 10000 s. Les gains en temps de calcul obtenus par ACID1 sur ses concurrents sont assez significatifs (voir la ligne *gain max*), tandis que ses pertes restent faibles. ACID0 avec ses deux paramètres a été plus difficile à régler, et les expérimentations n'ont pas montré d'intérêt à utiliser l'algorithme plus complexe ACID2. ACID1 obtient de meilleurs gains par rapport à 3BCID-n en temps total qu'en moyenne parce que les plus grands gains ont été obtenus sur des instances difficiles avec un grand nombre de variables. Sur la partie

TABLE 2 – NCSP : Gains en temps de calcul. Sont indiqués : le nombre de problèmes résolus en 3600 s et en 10000 s et différentes statistiques sur le temps CPU, ratio entre ACID1 et chaque concurrent C_i (un par colonne) : la moyenne, le maximum, le minimum et l'écart type de ce rapport $\frac{acid1\ time}{C_i\ time}$

	ACID1	HC4	3BCID-fp	3BCID-n	ACID0	ACID2	ACID1	3BCID-fp	3BCID-n
							¬ XN	¬ XN	¬ XN
#pbs résolus< 3600	26	20	23	24	25	24	20	16	20
#pbs résolus< 10000	26	21	26	26	26	26	22	21	22
Gain moyen	1	0.7	0.83	0.92	0.96	0.91	1	0.78	1.02
Gain maximum	1	0.13	0.26	0.58	0.45	0.48	1	0.18	0.38
Perte maximum	1	1.04	1.26	1.14	1.23	1.05	1	2.00	1.78
Ecart type gains	0	0.32	0.23	0.15	0.15	0.19	0	0.34	0.28
Temps total	23594	>72192	37494	27996	26380	30428	29075	50181	31273
Gain total	1		0.63	0.84	0.89	0.78	1	0.58	0.93

droite du tableau, on indique les rapports de temps de résolution obtenus quand on enlève X-Newton de la séquence des contracteurs (4 problèmes n'ont pas pu être résolus en 10000 s). La seule variante d'ACID étudiée est ACID1. ACID1 et 3BCID-n obtiennent globalement des résultats similaires, meilleurs que 3BCID-fp, mais avec un plus grand écart type qu'avec X-Newton, car le rognage prend une part plus importante dans la contraction.

3.2 Expérimentations en optimisation globale sous contraintes

Nous avons sélectionné dans la série 1 du banc d'essais Coconut d'optimisation globale sous contraintes [2] les 40 instances qu'ACID ou un concurrent peut résoudre en un temps compris entre 2 s and 3600 s. La limite en temps de calcul a été fixée à 3600 s. Nous avons utilisé IbexOpt, la stratégie d' Ibex qui réalise un algorithme de Branch & Bound en meilleur d'abord. Le protocole expérimental est le même que pour la résolution des NCSP, sauf que nous n'avons pas utilisé Interval-Newton, qui n'est implanté que pour les systèmes carrés.

Pour chaque instance, nous avons utilisé la meilleure heuristique de bissection (la même pour toutes les méthodes) parmi largestFirst, roundRobin et des variantes de la fonction *Smear*. La précision requise sur l'objectif est 10^{-8}. Chaque équation est relâchée en 2 inégalités avec une précision égale à 10^{-8}.

Le tableau 3 a les mêmes colonnes que le tableau 1, avec en plus une colonne indiquant le nombre de contraintes de l'instance.

Pour ce qui concerne la programmation par contraintes dans IbexOpt, HC4 correspond à l'état de l'art et 3BCID est rarement utile [3]. C'est pourquoi nous présentons dans l'avant dernière colonne une comparaison entre ACID1 et HC4. Le nombre de variables rognées a en effet été réglé par ACID1 à une valeur comprise entre 0 et le nombre de variables. De nouveau, on peut remarquer qu'ACID1 est robuste et est le meilleur, ou au plus 10% plus lent que le meilleur, pour 34 des 40 instances. Le tableau 4 montre que nous avons obtenu un gain moyen de 10% sur HC4. C'est significatif car la contraction due aux contraintes représente seulement une partie de la stratégie IbexOpt [12] (les relaxations linéaires et la recherche de points faisables sont d'autres algorithmes appartenant à la stratégie par défaut de IbexOpt qui ne sont pas étudiées dans cet article). ACID0 rogne un minimum de 3 variables, ce qui est souvent trop. ACID2 obtient des résultats légèrement moins bons qu'ACID1, ce qui rend en pratique ce raffinement non prometteur.

4 Conclusion

Nous avons présenté dans cet article une version adaptative de l'opérateur de contraction 3BCID utilisé par des méthodes à intervalles, opérateur proche de partition-1-AC dans les CSP en domaines finis. La meilleure variante de cet opérateur adaptatif, appelé ACID1 dans l'article, alterne des phases d'apprentissage et d'exploitation pour adapter le nombre de variables traitées. Ces variables sont sélectionnées par une heuristique de branchement efficace et tous les autres paramètres sont fixés et robustes aux modifications.

2. www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html

3. En fait, l'algorithme récent de propagation de contraintes Mohc [1] est meilleur que HC4. Mohc n'est pas encore réimplanté en Ibex 2.0. Cependant, 3BCID(Mohc) montre en gros les mêmes gains par rapport à Mohc que 3BCID(HC4) par rapport à HC4...

`ACID1` n'ajoute aucun paramètre aux stratégies de résolution et d'optimisation. Il produit les meilleurs résultats en moyenne. Pour chaque instance traitée, il est le meilleur ou proche du meilleur, même en présence des autres contracteurs (`Interval-Newton`, `X-Newton`). C'est pourquoi nous l'avons ajouté aux stratégies par défaut de résolution et d'optimisation d'`Ibex`.

Références

[1] I. Araya, G. Trombettoni, and B. Neveu. Exploiting Monotonicity in Interval Constraint Propagation. In *Proc. AAAI*, pages 9–14, 2010.

[2] I. Araya, G. Trombettoni, and B. Neveu. A Contractor Based on Convex Interval Taylor. In *Proc. CPAIOR*, volume 7298 of *LNCS*, pages 1–16. Springer, 2012.

[3] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proc. ICLP*, volume 5649 of *LNCS*, pages 230–244. Springer, 1999.

[4] H. Bennaceur and M.-S. Affane. Partition-k-AC : An Efficient Filtering Technique Combining Domain Partition and Arc Consistency. In *Proc. CP*, volume 2239 of *LNCS*, pages 560–564. Springer, 2001.

[5] C. Bessiere and R. Debruyne. Optimal and Suboptimal Singleton Arc Consistency Algorithms. In *Proc. IJCAI*, pages 54–59, 2005.

[6] G. Chabert and L. Jaulin. Contractor Programming. *Artificial Intelligence*, 173 :1079–1100, 2009.

[7] R. Debruyne and C. Bessiere. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proc. IJCAI*, pages 412–417, 1997.

[8] E. Hansen. *Global Optimization using Interval Analysis*. Marcel Dekker inc., 1992.

[9] O. Lhomme. Consistency Techniques for Numeric CSPs. In *Proc. IJCAI*, pages 232–238, 1993.

[10] F. Messine. *Méthodes d'optimisation globale basées sur l'analyse d'intervalle pour la résolution des problèmes avec contraintes*. PhD thesis, LIMA-IRIT-ENSEEIHT-INPT, Toulouse, 1997.

[11] C. Min Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proc. IJCAI*, pages 366–371, 1997.

[12] G. Trombettoni, I. Araya, B. Neveu, and G. Chabert. Inner Regions and Interval Linearizations for Global Optimization. In *Proc. AAAI*, pages 99–104, 2011.

[13] G. Trombettoni and G. Chabert. Constructive Interval Disjunction. In *Proc. CP*, volume 4741 of *LNCS*, pages 635–650. Springer, 2007.

TABLE 3 – Résultats obtenus par ACID1 en optimisation

	#var	#ctr	ACID1	ACID1	ACID1	meilleur	pire	Temps	Temps	Temps
			temps	#nœuds	#vcids			ACID1/meilleur	ACID1/HC4	ACID1/pire
Ex2_1_7	20	10	8.75	465	3	HC4	3BCID-fp	1.03	1.03	0.7
Ex2_1_8	24	10	6.18	200	0	HC4	3BCID-fp	1.06	1.06	0.91
Ex2_1_9	10	1	10.1	1922	0.75	HC4	3BCID-fp	1.04	1.04	0.9
Ex5_4_4	27	19	915	23213	0.8	ACID1	3BCID-n	1	0.96	0.91
Ex6_1_1	8	6	60.8	13071	8.9	HC4	3BCID-fp	1.21	1.21	0.73
Ex6_1_3	12	9	297	29154	11.7	HC4	3BCID-fp	1.19	1.19	0.63
Ex6_1_4	6	4	1.99	505	6	ACID1	3BCID-fp	1	0.97	0.8
Ex6_2_6	3	1	107	46687	0	HC4	3BCID-fp	1.02	1.02	0.74
Ex6_2_8	3	1	48.2	21793	0.1	HC4	3BCID-fp	1.01	1.01	0.72
Ex6_2_9	4	2	51.9	19517	0.1	HC4	3BCID-fp	1.02	1.02	0.72
Ex6_2_10	6	3	2248	569816	0	ACID1	3BCID-fp	1	0.99	0.64
Ex6_2_11	3	1	29.3	13853	0.3	HC4	3BCID-fp	1.05	1.05	0.73
Ex6_2_12	4	2	21.6	7855	0.1	HC4	3BCID-fp	1.02	1.02	0.8
Ex7_2_3	8	6	19.4	4596	4.4	3BCID-n	HC4	1.07	0.17	0.17
Ex7_2_4	8	4	36.8	5606	4.2	3BCID-fp	HC4	1.04	0.66	0.66
Ex7_2_8	8	4	38.0	6792	4.1	3BCID-n	HC4	1.09	0.71	0.71
Ex7_2_9	10	7	78.0	14280	9.3	3BCID-n	HC4	1.07	0.48	0.48
Ex7_3_4	12	17	2.95	366	3	3BCID-n	3BCID-fp	1.23	0.99	0.89
Ex7_3_5	13	15	4.59	894	6	3BCID-n	HC4	1.05	0.38	0.38
Ex8_4_4	17	12	1738	46082	0.9	ACID1	3BCID-fp	1	0.99	0.87
Ex8_4_5	15	11	772	25454	4.8	HC4	3BCID-fp	1.03	1.03	0.75
Ex8_5_1	6	5	9.67	2138	2.75	ACID1	3BCID-fp	1	0.84	0.82
Ex8_5_2	6	4	32.5	5693	0.8	ACID1	3BCID-fp	1	0.9	0.87
Ex8_5_6	6	4	32.4	10790	1.8	HC4	3BCID-fp	1.02	1.02	0.76
Ex14_1_7	10	17	665	95891	3.3	3BCID-n	HC4	1.03	0.61	0.61
Ex14_2_3	6	9	2.01	360	2	HC4	3BCID-fp	1.17	1.17	0.69
Ex14_2_7	6	9	49.9	5527	0	HC4	3BCID-n	1.47	1.47	0.48
alkyl	14	7	3.95	714	4	HC4	3BCID-fp	1.2	1.2	0.91
bearing	13	12	11.6	1098	13	3BCID-n	HC4	1.01	0.53	0.53
hhfair	28	25	26.6	3151	10	3BCID-n	HC4	1.12	0.58	0.58
himmel16	18	21	188	21227	15.5	3BCID-n	3BCID-fp	1.1	0.94	0.88
house	8	8	62.8	27195	3.25	HC4	3BCID-fp	1.09	1.09	0.79
hydro	30	24	609	32933	0	ACID1	3BCID-fp	1	0.88	0.78
immun	21	7	4.17	1317	2.5	ACID1	3BCID-fp	1	0.55	0.28
launch	38	28	107	2516	21	ACID1	3BCID-n	1	0.79	0.43
linear	24	20	751	27665	0.25	ACID1	3BCID-n	1	0.98	0.65
meanvar	7	2	2.43	370	2	HC4	3BCID-fp	1.04	1.04	0.84
process	10	7	2.61	611	8	HC4	3BCID-fp	1.08	1.08	0.77
ramsey	31	22	164	4658	4.3	ACID1	3BCID-fp	1	0.85	0.68
srcpm	38	27	160	6908	0.5	ACID1	3BCID-fp	1	0.62	0.33

TABLE 4 – Problèmes d'optimisation : rapport des gains en temps de résolution : temps ACID1/temps xxx

	ACID1	HC4	3BCID-fp	3BCID-n	ACID0	ACID2
# pbs résolus	40	40	40	40	40	40
Gain moyen	1	0.9	0.77	0.88	0.91	0.97
Gain maximum	1	0.17	0.28	0.35	0.62	0.28
Perte maximum	1	1.47	1.04	1.23	1.18	1.19
Ecart type gains	0	0.25	0.16	0.18	0.12	0.14
Temps total	9380	10289	12950	11884	11201	9646
Gain total	1	0.91	0.72	0.79	0.84	0.97