# Un algorithme arborescent de contraction forte pour le CSP numérique

**Olivier Sans** **Remi Coletta** **Gilles Trombettoni**
LIRMM, Université de Montpellier, 161 rue Ada, 34095 Montpellier, France
{prenom.nom}@lirmm.fr

**Résumé**

Nous proposons dans cet article un nouvel opérateur de contraction forte pour le CSP numérique. Cet opérateur, nommé TEC (*Tree for Enforcing Consistency*), construit un sous-arbre d'exploration avant d'effectuer l'union (l'enveloppe) des espaces de recherche associés aux feuilles de ce sous-arbre. TEC construit le sous-arbre en largeur d'abord par un processus de bissection et de contraction. TEC généralise à plusieurs dimensions le principe de la disjonction constructive utilisée par l'opérateur CID existant. A la différence des opérateurs classiques de contraction forte, tels que CID ou 3B, qui considèrent toutes les variables du problème, TEC concentre son travail sur un sous-ensemble de variables à bissecter de manière adaptative. L'enveloppe des boîtes feuilles du sous-arbre TEC étant contrainte à être parallèle aux axes, la boîte résultante surestime l'espace de recherche. Pour éviter cet inconvénient, nous proposons Graham-TEC, qui permet d'inférer des contraintes implicites linéaires. Graham-TEC génère une enveloppe convexe avec des demi-plans portant sur des paires de variables. Des expériences préliminaires portant sur un banc d'essais de l'état de l'art valident l'intérêt de notre approche, en montrant des gains significatifs sur plusieurs instances difficiles.

## 1 Introduction

Les solveurs par intervalles permettent de résoudre des systèmes de contraintes numériques (c'est-à-dire des équations ou inégalités à valeurs dans $\mathbb{R}$) en prenant en compte des incertitudes (bornées) dans les paramètres (dues à des erreurs de mesure en physique par exemple) et les erreurs d'arrondis des calculs sur les nombres flottants. Des calculs rigoureux sont rendus possibles par la manipulation de domaines des variables bornés par des nombres flottants et l'utilisation de l'arithmétique d'intervalles.

Le processus de résolution est basé sur une stratégie de branchement et de contraction (*Branch & Contract*). Le branchement consiste à sélectionner une variable du système et à couper son domaine en (généralement) deux sous-intervalles. Ce processus génère un arbre d'exploration. La contraction (ou filtrage) consiste à réduire les domaines des variables du système sans perte de solution afin de limiter l'explosion combinatoire. Les opérateurs de contraction existants proviennent de trois domaines de recherche : la programmation par contraintes, l'analyse par intervalles et la programmation mathématique. Le processus de branchement et contraction enchaîne couramment un contracteur de programmation par contraintes et un opérateur de convexification polyhédrale.

En programmation par contraintes sur domaines finis, un principe de rognage (*shaving*) peut être utilisé pour filtrer les domaines. L'algorithme établissant la *singleton arc-consistency* (SAC [8]) affecte temporairement une valeur à une variable (les autres étant temporairement éliminées) et calcule une cohérence locale sur le sous-problème correspondant. Si une incohérence est obtenue, la valeur est supprimée du domaine dans tout le sous-arbre, sinon elle est maintenue. Dans le dernier cas, la réduction effectuée sur les domaines des autres variables est perdue. La disjonction constructive permet de mieux exploiter cette réduction en effectuant l'union des sous-problèmes générés par l'affectation de chaque valeur à la variable. Ce principe, mis en œuvre dans l'algorithme Partition-1-AC [6, 4], produit une cohérence strictement plus forte que la SAC. Elle donne un espoir de réduction sur potentiellement toutes les variables du système quand l'algorithme travaille sur (rogne) une seule variable.

Sur les domaines continus qui contiennent une infinité de nombres réels, les intervalles sont divisés en un nombre spécifié de sous-intervalles qui jouent le rôle de valeurs dans le CSP sur domaines finis. L'opérateur de contraction CID [19] applique ainsi le principe

de la disjonction constructive sur chaque variable du système prise une à une. Il déclenche un algorithme de propagation de contraintes, comme 2B [11, 12, 5] ou Mohc [2], sur chaque sous-problème correspondant à un sous-intervalle de la variable traitée. L'union des différents domaines contractés est alors retournée. Plus précisément, la boîte (produit cartésien des intervalles) *enveloppe* de cette union est rendue car les intervalles sont réduits seulement aux bornes (pas de *trous* au milieu des intervalles).

Dans cet article, nous proposons un nouvel opérateur de contraction, appelé TEC (*Tree for Enforcing Consistency*), qui généralise l'opérateur CID en appliquant le rognage sur plusieurs variables simultanément. TEC effectue un parcours arborescent de type *Branch & Contract* dont le nombre de nœuds est borné par un paramètre donné. La contraction est obtenue en retournant la boîte enveloppe des feuilles de cet arbre, c'est-à-dire la plus petite boîte (parallèle aux axes) qui englobe toutes les boîtes feuilles.

La boîte enveloppe, qui peut se voir comme un ensemble de contraintes (linéaires) unaires sur les variables du système, peut grandement surestimer l'union des boîtes feuilles. C'est pourquoi nous proposons une variante de TEC, appelée Graham-TEC, qui infère des contraintes linéaires implicites pour chaque paire de variables. Une enveloppe convexe de la projection des boîtes feuilles en deux dimensions permet de mieux approximer l'union des feuilles. Ces contraintes binaires linéaires sont ajoutées au polytope généré par l'opérateur X-Newton [3] de convexification polyhédrale utilisé pendant la résolution pour mieux contracter les domaines.

Après les définitions et les notations présentées à la section 2, la section 3 décrit l'opérateur TEC et sa mise au point expérimentale. La section 4 présente Graham-TEC et la section 5 détaille des expériences préliminaires encourageantes obtenues par une stratégie enchaînant Graham-TEC et l'opérateur X-Newton enrichi par les contraintes linéaires fournies par Graham-TEC.

## 2 Définitions et notations

Les algorithmes présentés dans cet article permettent de résoudre des systèmes ou réseaux de contraintes numériques. Un **réseau de contraintes numériques** est défini par un triplet $P = (X, [X], C)$, où $X$ représente un ensemble de $n$ variables réelles à valeurs dans $[X]$. Nous notons $[x_i] = [\underline{x_i}, \overline{x_i}]$, l'intervalle/domaine de la variable $x_i \in X$, où $\underline{x_i}$ et $\overline{x_i}$ sont des nombres flottants. Une contrainte de $C$ est une équation ou une inégalité sur les nombres réels définie par une expression mathématique classique pouvant contenir des opérateurs arithmétiques, trigonométriques, d'exponentiation, etc. Une solution de $P$ est un vecteur $n$-dimensionnel de $[X]$ satisfaisant l'ensemble des contraintes de $C$.

Le domaine $[X] = [x_1] \times ... \times [x_n]$ est un produit cartésien d'intervalles communément nommé boîte (parallèle aux axes). La **largeur** ou **taille** d'un intervalle $[x_i]$ est définie par $\omega(x_i) = \overline{x_i} - \underline{x_i}$. La largeur d'une boîte est donnée par la plus grande largeur sur l'ensemble des intervalles de la boîte, c'est-à-dire : $\max_{i=1..n}\ \omega(x_i)$.

L'union de plusieurs boîtes n'étant généralement pas une boîte, un opérateur $Hull$ d'enveloppe a été défini.

**Définition 1.** *Soit $box_1, ..., box_m$ un ensemble de boîtes $n$-dimensionnelles. L'opération* ***Hull*** *de $box_1, ..., box_m$, notée $Hull(box_1, ..., box_m)$, est la boîte minimale incluant l'ensemble de ces $m$ boîtes.*

Un réseau de contraintes numériques peut être résolu par une stratégie de branchement et de contraction (*Branch & Contract*) mentionnée en introduction. Ce processus, détaillé à l'algorithme 1, débute avec le domaine initial $box = [X]$ et se termine lorsque la taille des boîtes feuilles de l'arbre d'exploration est inférieure à une précision $\epsilon$ donnée. Le parcours est effectué en profondeur d'abord. Il est garanti que les boîtes éliminées par contraction ne contiennent pas de solution réelle. Les boîtes feuilles rangées dans *AtomicBoxes* puis renvoyées par l'algorithme peuvent contenir ou non des solutions. Des algorithmes comme celui de *Newton sur intervalles* [13] peuvent être utilisées pour essayer de garantir l'existence d'une solution réelle dans les boîtes atomiques.

**Algorithm 1:** `Branch&Contract`

```
   Input: N = (X, box, C) : an NCN, ε : precision parameter
   Output: A set of atomic boxes including all the solutions
1  Boxes ← {box}                          /* stack (LIFO) of boxes */
2  AtomicBoxes ← ∅                        /* set of atomic boxes */
3  while Boxes ≠ ∅ do
4      box' ← Boxes.pop()
5      box' ← CPContract(box')
6      box' ← ConvexifyContract(box')
7      if box' ≠ ∅ then
8          if ω(box') < ε then
9              AtomicBoxes.push(box')
10         else
11             (box1, box2) ← Bisect(box')
12             Boxes.push(box1)
13             Boxes.push(box2)
14 return AtomicBoxes
```

La **contraction** (ou filtrage) se définit informellement par la réduction des domaines aux bornes sans perte de solution. L'algorithme 1 souligne que la plupart des solveurs à intervalles font appel à deux opérateurs de filtrage en séquence, l'un issu de la programmation par contraintes ($CPcontract$) et l'autre de l'analyse par intervalles ou de la programmation mathématique. Ce dernier ($ConvexifyContract$) se

base sur une convexification polyhédrale de l'espace-solution pour contracter les domaines.

Plusieurs opérateurs de contraction ont été proposés par la communauté de programmation par contraintes. L'opérateur HC4 [5, 12] permet de calculer la 2B-cohérence [11]. Cette cohérence locale s'apparente à la cohérence de bornes (*Bound-consistency*), une forme d'arc-cohérence restreinte aux bornes des intervalles.

Comme expliqué en introduction et de manière similaire à SAC pour le CSP sur domaines finis [8], l'algorithme établissant la 3B-cohérence [11] se base sur un principe de rognage qui élimine un sous-intervalle aux bornes des domaines quand l'appel à un algorithme comme la 2B-cohérence sur le sous-problème correspondant produit un domaine vide (prouvant l'absence de solution dans le sous-problème). L'opérateur CID (*Constructive Interval Disjunction*) se base sur le principe de la disjonction constructive appliqué aux domaines des variables. La procédure principale de CID, VarCID, sélectionne une variable $x_i$. L'intervalle $[x_i]$ est découpé en $s_{cid}$ sous-intervalles ; chaque sous-problème correspondant est contracté par un opérateur de contraction tel que HC4, puis le $Hull$ de l'ensemble des boîtes résultantes est retourné. Ce processus est itéré sur chaque variable tant qu'une contraction est obtenue.

# 3 L'opérateur TEC

La disjonction constructive de l'opérateur CID peut produire une contraction sur l'ensemble du réseau lorsqu'un rognage est effectué sur une seule variable. L'opérateur TEC généralise ce principe en ne rognant qu'un sous-ensemble des variables mais de façon simultanée. TEC construit en fait un sous-arbre d'exploration de taille bornée avant d'effectuer le *Hull* des boîtes feuilles générées. En chaque nœud du *sous-arbre TEC*, est appliquée une cohérence locale telle que la 2B-cohérence. L'algorithme 2 détaille le fonctionnement de TEC.

## 3.1 Algorithme

TEC met en œuvre un processus de bissection et de contraction en construisant en largeur d'abord un sous-arbre de taille bornée. Ce processus combinatoire (recherche arborescente) est réalisé par la boucle `while` (ligne 4). La boîte initiale est tout d'abord contractée par le sous-contracteur $\Phi$ avant d'initialiser la file $Boxes$. A chaque itération, la première boîte de $Boxes$ est extraite selon un principe *First In First Out* mettant en œuvre le parcours en largeur d'abord. Si cette boîte n'est pas vide (non détectée incohérente par la cohérence locale) et si sa taille est supérieure à un paramètre de precision $\epsilon$, elle est bissectée sur une dimension et chacune des sous-boîtes générées est alors contractée par $\Phi$ puis placée dans $Boxes$. Si la boîte est trop petite (de taille inférieure à $\epsilon$), elle est ajoutée dans $AtomicBoxes$. Ce processus s'arrête lorsque le parcours arborescent a contracté un nombre de noeuds maximum (paramètre $TECnodes$ [1]) ou lorsque la file $Boxes$ est vide. L'enveloppe ($Hull$) de l'ensemble des boîtes feuilles ($Boxes \cup AtomicBoxes$) est retournée par l'opérateur.

**Algorithm 2:** `TEC`

```
   Input: N = (X, box, C) : an NCN, Φ : subcontractor,
          TECnodes : max calls, ε : precision parameter
   Output: box : the contracted box
1  Boxes ← {Φ(X, box, C, ε)}        /* queue (FIFO) of boxes */
2  AtomicBoxes ← ∅                  /* set of atomic boxes */
3  #nodes ← 1
4  while Boxes ≠ ∅ and #nodes ≤ TECnodes do
5      box' ← Boxes.pop()
6      if box' ≠ ∅ then
7          if ω(box') > ε then
8              (box1, box2) ← Bisect(box')
9              Boxes.push(Φ(X, box1, C, ε))
10             Boxes.push(Φ(X, box2, C, ε))
11             #nodes ← #nodes + 2
12         else
13             AtomicBoxes.push(box')
14 box ← Hull(Boxes ∪ AtomicBoxes)
```

## 3.2 Le choix de paramètres pour TEC

Nous détaillons dans cette section les paramètres choisis pour l'opérateur TEC. Ces choix ont été validés par des expérimentations préliminaires non reportées dans cet article.

### Parcours en largeur d'abord

Le paramètre $TECnodes$ permet de borner le nombre de noeuds du sous-arbre TEC. L'utilisation d'un parcours en largeur d'abord découle naturellement de ce choix de paramètre. Soulignons toutefois qu'une première version de l'opérateur TEC bornait l'arbre TEC par sa hauteur, ce qui rendait indifférent l'ordre du parcours (en largeur ou en profondeur d'abord), l'arbre étant complet. En revanche, cette version avait un défaut majeur. Le doublement potentiel du nombre de noeuds à chaque niveau du sous-arbre rendait le contrôle de l'opérateur délicat. C'est pourquoi cette version de TEC bornée par la hauteur a été abandonnée au profit de la version présentée.

### Stratégie de choix de variables

TEC ne rogne/bissecte pas l'ensemble des variables du réseau. Par conséquent, il est nécéssaire d'introduire une stratégie de choix de variables. Le choix

1. Notons que ce nombre est nécessairement impair dans l'implantation actuelle.

d'un sous-ensemble de variables prédéfinies a rapidement été écarté. En effet, un tel choix ne prend pas en compte l'évolution du réseau et peut manquer cruellement de pertinence dans certains cas. Notre choix s'est donc orienté vers une étude expérimentale des stratégies dynamiques de choix de variables.

#### Stratégie de branchement

Le branchement effectué en chaque nœud a aussi fait l'objet d'une étude expérimentale permettant d'observer le comportement de l'opérateur en fonction du nombre de sous-boîtes générées (bissection, trissection, ...). La bissection est la plus avantageuse au vu des résultats. L'utilisation de stratégies de choix de variables dynamiques semble expliquer ces résultats. En effet, on peut remarquer qu'une variable peut être traitée plusieurs fois dans une même branche de l'arbre. La stratégie peut alors sélectionner consécutivement deux fois la même variable, et ainsi appliquer un principe de trissection. L'avantage est que le choix du nombre de sections s'adapte ainsi dynamiquement en fonction de la contraction du domaine.

### 3.3 Valeurs par défaut des paramètres de TEC

Une étude expérimentale préliminaire a permis de calculer les valeurs par défaut de l'opérateur TEC. Ces valeurs permettent de définir une version de TEC stable et comparable à l'algorithme 3BCID [19] de l'état de l'art.

Le sous-contracteur $\Phi$ de TEC est un contracteur de programmation par contraintes (cf. la procédure CPcontract à l'algorithme 1). Des études non reportées dans cet article montrent que l'utilisation de HC4 est préférable à celle de 3BCID au sein de l'arbre TEC [16].

Selon l'instance considérée (cf. section 5), la meilleure valeur du paramètre $TECnodes$ oscille entre 10 et 100. Nos études ne nous ont pas permis de trouver un critère intelligent permettant de sélectionner $TECnodes$ selon l'instance traitée. Toutefois, la valeur 25 donne un compromis toujours acceptable et définit ainsi la valeur par défaut de $TECnodes$ lors de la comparaison à l'existant.

Enfin, nos comparaisons expérimentales des différentes stratégies de choix de variables existantes nous ont conduits à choisir la stratégie $SmearSumRel$ [2].

2. *SmearSumRel* est une stratégie de choix de variables connue comme l'une des plus performantes [18]. Elle se base sur la fonction *smear* [10]. $smear(x_i, c_j)$ mesure l'impact de la variable $x_i$ sur la fonction de $c_j$. Son calcul implique la largeur de $[x_i]$ et la dérivée partielle de $c_j$ par rapport à $x_i$ évaluée sur la boîte $[X]$.

## 4 L'opérateur Graham-TEC

Nous présentons dans cette section une variante de TEC, nommée Graham-TEC, qui infère des contraintes binaires implicites. Ces contraintes linéaires sont ensuite ajoutées au polytope généré par l'opérateur de convexification polyhédrale utilisé pendant la résolution. Plus précisément, si l'on reprend l'algorithme 1, nous proposons de mettre en œuvre $CPContract$ par Graham-TEC, qui transmet des contraintes linéaires à la procédure suivante ($ConvexifyContract$) pour enrichir son polytope. Cet ajout compense la relaxation de l'espace solution par la convexification et permet ainsi de mieux contracter les domaines.

Comme le fait TEC, l'opérateur Graham-TEC construit un arbre en largeur d'abord. Il calcule ensuite, pour chaque paire de variables, l'enveloppe convexe des boîtes feuilles de l'arbre TEC. Finalement, l'algorithme retourne, comme TEC, une boîte contractée issue de l'enveloppe des boîtes feuilles, mais retourne en plus un ensemble de contraintes binaires linéaires formant un polyèdre convexe.

### 4.1 Construction du polytope

L'algorithme 3 est appelé à la fin de l'algorithme 2. Cet algorithme prend en entrée l'ensemble des boîtes feuilles de l'arbre TEC, la boîte $Hull$ de celles-ci ainsi qu'un paramètre $\gamma$ permettant de décider si un demi-plan construit est ajouté ou non au polytope. Il retourne au final un ensemble de contraintes binaires linéaires.

Pour chaque paire $(x_i, x_j)$ de variables, et pour chaque boîte feuille $l$ de l'arbre TEC, les quatre sommets du rectangle correspondant à la projection de $l$ sur les deux dimensions $(x_i, x_j)$ sont ajoutés à une liste $Points$. Si un sommet est trop proche d'un sommet du rectangle $[h_i] \times [h_j]$ (correspondant à la projection de la boîte enveloppe $h$), le quadrant est considéré comme "couvert" et aucun demi-plan ne sera ajouté dans cette zone. Dans le cas extrême d'une seule boîte feuille confondue avec son enveloppe, ces conditions calculent bien une distance nulle pour chacun des 4 quadrants, si bien que `PolytopeGenerator` termine immédiatement sans construire de contraintes linéaires implicites. Les conditions précises décrites aux lignes 10 à 17 mesurent la distance à un sommet du rectangle $[h_i] \times [h_j]$ en pourcentage $\gamma$ de la largeur $[h_i]$ ou $[h_j]$. La figure 1 montre un exemple.

Un algorithme de génération d'enveloppe convexe [9] est ensuite appelé sur chaque quadrant sélectionné. L'algorithme produit une liste ordonnée des points composant l'enveloppe convexe de tous les points de $Points$. Cet algorithme classique de géométrie algorithmique commence par trier les points candidats

**Algorithm 3:** `PolytopeGenerator`

```
   Input: h : the box representing the hull
   TECLeaves : set of TEC leaves, γ : the min gain
   Output: S : a set of linear constraints
1  S ← ∅
2  [h_i] ← h[x_i] ; [h_j] ← h[x_j]
3  foreach (x_i, x_j) ∈ X^2 with i < j do
4      Points ← ∅
5      foreach sector ∈ {NE, NW, SW, SE} do
6          cover(sector) ← false
7      foreach l ∈ TECLeaves do
8          [l_i] ← l[x_i] ; [l_j] ← l[x_j]
9          Points ← Points ∪ {(l̄_i, l̄_j), (l̄_i, l_j), (l_i, l̄_j), (l_i, l_j)}
10         if |h̄_i − l̄_i| < γ × ω(h_i) and |h_j − l_j| < γ × ω(h_j) then
11             cover(SE) ← true
12         if |h̄_i − l̄_i| < γ × ω(h_i) and |h̄_j − l̄_j| < γ × ω(h_j) then
13             cover(NE) ← true
14         if |h_i − l_i| < γ × ω(h_i) and |h̄_j − l̄_j| < γ × ω(h_j) then
15             cover(NW) ← true
16         if |h_i − l_i| < γ × ω(h_i) and |h_j − l_j| < γ × ω(h_j) then
17             cover(SW) ← true
18     foreach sector ∈ {NE, NW, SW, SE} do
19         if ¬cover(sector) then
20             //Graham outputs the pareto front on a sector
21             paretoFront ← Graham(Points, sector)
22             S ← S ∪ getRigorousConstraints(h, x_i,
23                                 x_j, paretoFront, sector)
24 return S
```

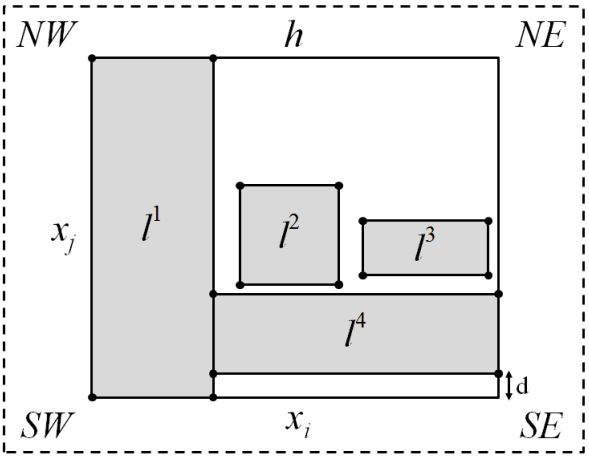


FIGURE 1 – Illustration de la procédure `PolytopeGenerator`. Le rectangle en pointillés représente la boîte initiale (avant contraction de l'opérateur) projetée sur $x_i$ et $x_j$. $h$ représente la boîte enveloppe retournée par l'opérateur. Les quadrants $NW$, $SW$ et $SE$ ne pouvant pas apporter de gain suffisant, ils ne sont pas traités ($d$ étant inférieure à $\gamma \times \omega(h_j)$).

par angle polaire en fonction d'un point « pivot ». Il parcourt ensuite les points dans ce nouvel ordre en construisant l'enveloppe convexe dans le sens trigonométrique direct. L'algorithme décide de mettre le point courant dans l'enveloppe, plus précisément dans une liste *paretoFront*, si l'angle entre les deux derniers segments de l'enveloppe en construction est positif. La complexité en temps de cet algorithme est dominée par le tri en $O(p.log(p))$ où $p$ est le nombre de points traités.

A partir de cette liste de points, la procédure `getRigorousConstraints` génère un ensemble de demi-plans (contraintes binaires linéaires) qui seront ajoutés au polytope créé par $ConvexifyContract$.

### 4.2 Génération rigoureuse des contraintes

La procédure `getRigorousConstraints` (cf. algorithme 4) permet de générer de manière rigoureuse des contraintes linéaires à partir des points de l'enveloppe convexe. Chaque paire de points consécutifs $((x_i, y_i), (x_{i+1}, y_{i+1}))$ prise dans la liste *paretoFront* est traitée itérativement. Le but est de générer une droite quasi-confondue avec le segment entre ces deux points. La difficulté réside dans le fait que la droite réelle $y = ax + b$ qui passe par ces deux points a généralement des coefficients $a$ et $b$ qui ne sont pas des nombres flottants (cf. figure 2). Or, les problèmes d'arrondis des calculs sur les flottants pourraient entraîner la génération d'un demi-plan à coefficients flottants qui n'englobe pas ces deux points. Pour éviter ce problème, les demi-plans sont calculés de manière rigoureuse grâce à l'arithmétique d'intervalles. A partir des points $(x_i, y_i)$ et $(x_{i+1}, y_{i+1})$, l'arithmétique d'intervalles calcule des (tout petits) intervalles $[a]$ et $[b]$ contenant nécessairement les coefficients réels. Le lecteur pourra facilement vérifier le choix des bornes permettant d'approximer les deux coefficients de manière rigoureuse :

- pour les 4 quadrants, le coefficient directeur est donné par $\underline{a}$ ;
- pour les deux quadrants nord, la droite calculée est définie avec le coefficient $\overline{b}$ ;
- pour les deux quadrants sud, la droite calculée est définie avec le coefficient $\underline{b}$ ;

La figure 2 illustre ce calcul rigoureux sur le quadrant SE.

La ligne 19 ajoute une condition pour garantir que le point précédent satisfasse l'inégalité, c'est-à-dire que le demi-plan contienne aussi le point précédent $(x_{i-1}, y_{i-1})$. En effet, l'arrondi $\underline{a}$ peut entraîner la violation de cette contrainte en théorie (dans un cas pathologique où les 3 points seraient quasi-alignés).

Pour un quadrant donné, l'ensemble *paretoFront* peut contenir au maximum $r$ points, avec $r$ le nombre de boîtes feuilles du sous-arbre TEC. La boucle principale de l'algorithme 4 peut donc produire au plus $r-1$ contraintes linéaires. Certaines de ces contraintes linéaires peuvent être presque parallèles entre elles (ou parallèles aux bords de la boîte). Ces contraintes ne couperaient alors presque pas de volume de la boîte et ralentiraient l'opérateur $ConvexifyContract$ sans en

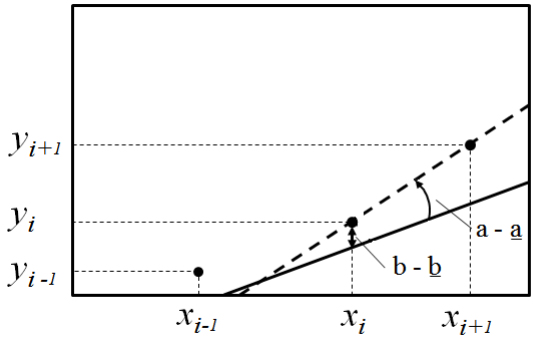


FIGURE 2 – Schéma représentant le calcul rigoureux sur le quadrant SE. La droite en pointillés correspond à la droite de coefficients $a$ et $b$ à valeur dans $\mathbb{R}$. La deuxième correspond à la droite de coefficients $a$ et $b$ arrondis sur les nombres flottants.

**Algorithm 4:** `getRigorousConstraints`

**Input**: $paretoFront = ((x_1, y_1), \ldots, (x_m, y_m))$, $h$ the box representing the hull, $x$ and $y$ two dimensions, $sector$
**Output**: $P$ a set of rigorous half-planes

$P \leftarrow \emptyset$
**foreach** $(x_i, y_i) \in paretoFront$ **do**
  //Build the degenerated intervals :
  $[x_i] \leftarrow [x_i, x_i]$ ; $[y_i] \leftarrow [y_i, y_i]$
  $[x_{i+1}] \leftarrow [x_{i+1}, x_{i+1}]$ ; $[y_{i+1}] \leftarrow [y_{i+1}, y_{i+1}]$
  //Interval of slope value of the linear function
  $[a] \leftarrow ([y_{i+1}] - [y_1])/([x_{i+1}] - [x_i])$
  //Interval of y-intercept value of the linear function
  $[b] \leftarrow [y_i] - ([x_i] \times [a])$
  **switch** $sector$ **do**
    **case** $SE$
      $p \leftarrow \underline{a} \times x - y \leq -\underline{b}$
    **case** $NE$
      $p \leftarrow \underline{a} \times x - y \geq -\overline{b}$
    **case** $NW$
      $p \leftarrow \underline{a} \times x - y \geq -\overline{b}$
    **case** $SW$
      $p \leftarrow \underline{a} \times x - y \leq -\underline{b}$
  **if** $(x_{i-1}, y_{i-1})$ satisfies $p$ **then** $P \leftarrow P \cup \{p\}$
return $P' \subseteq P$

améliorer la contraction. Aussi, à la dernière ligne de *getRigorousConstraints*, nous prévoyons de ne renvoyer qu'un sous-ensemble des contraintes générées. On pourrait imaginer de ne retourner qu'un top-$k$ des contraintes rognant le plus d'espace, ce qui reviendrait à explorer $\binom{k}{r-1}$ combinaisons. Le choix des contraintes à conserver implanté dans la version courante est assez simple. Nous utilisons un algorithme glouton qui traite chaque contrainte linéaire individuellement et vérifie qu'elle rogne au moins une proportion $\gamma$ sur l'une des dimensions. Ce test est illustré sur la figure 3. La contrainte en pointillés rouges ne permet de rogner qu'une portion $d$ sur la dimension $x_i$, inférieure à $\gamma \times h[x_i]$, et elle n'est donc pas posée. En revanche, la contrainte linéaire en noir permet de couper significativement $[h_i]$ et $[h_j]$ et sera donc transmise à *ConvexifyContract*. Cette procédure glouton ne filtre en quelque sorte que les contraintes linéaires presque parallèles aux axes, modulo une normalisation des deux largeurs $\omega([h_i])$ et $\omega([h_i])$. Nous l'étendrons à court terme pour prendre en compte les contraintes parallèles entre elles. Elle permet de réduire en pratique le nombre de contraintes transmises à *ConvexifyContract*, réduction que nous quantifierons expérimentalement.

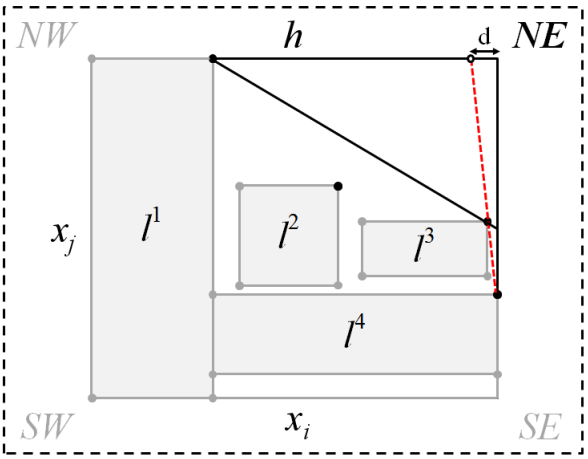


FIGURE 3 – Illustration de la procédure `getRigorousConstraints`. La droite en pointillés n'apportant pas un gain suffisant par rapport au bord de la boîte, elle n'est pas ajoutée au polytope généré.

## 5 Validation expérimentale

Dans cette section, nous présentons les résultats de l'opérateur TEC et de son extension Graham-TEC, et nous les comparons aux deux opérateurs de contraction HC4 et 3BCID définissant l'état de l'art.

### 5.1 Bancs d'essais et outil de résolution

Nous avons implanté TEC et Graham-TEC dans la bibliothèque à intervalles libre `Ibex` (Interval-Based EXplorer) [7]. Cette bibliothèque a facilité l'implantation de nos opérateurs en fournissant un certain nombre de briques telles que la stratégie de résolution *Branch & Contract*décrite par l'algorithme 1, HC4 et 3BCID comme contracteurs issus de la PPC (*CPContract*) et X-Newton comme opérateur de convexification polyhédrale (*ConvexifyContract*).

Les comparaisons sont faites sur des problèmes de résolution traités par une stratégie proche de celle décrite par l'algorithme 1, mais aussi sur des problèmes d'optimisation globale sous contraintes traités par l'optimiseur IbexOpt [18]. En plus du branchement et de la contraction (qui enchaîne *CPContract* et *ConvexifyContract*), IbexOpt cherche à calculer en chaque nœud de l'arbre un minorant et un majorant de la fonction objectif dont la différence décide de

la terminaison. *CPContract* (cf. algorithme 1, ligne 5) représente les opérateurs que nous comparons (HC4, 3BCID, TEC, Graham-TEC) tandis que l'opérateur X-Newton (*ConvexifyContract*) est utilisé dans chacune des stratégies. L'heuristique de choix de variables utilisée est l'heuristique *SmearSumRel* présentée à la section 2.

Tous les paramètres ont été fixés à des valeurs communes à toutes les instances testées. La précision d'une boîte solution (cf. algorithme 1, ligne 8) a été fixée à $\epsilon$ = `1.e-10` et le temps-limite de chaque instance a été fixé à 1800 secondes. Pour les instances d'optimisation, la valeur du paramètre de précision sur la fonction objectif a été fixée à $\epsilon_{obj}$ = `1.e-08`. Les paramètres de TEC ainsi que ceux de Graham-TEC utilisés lors de nos expériences sont ceux présentés à la section 3.3, à savoir : $TECnodes$=25, bissection=SmearSumRel. L'ensemble de nos expériences ont été réalisées sur un *Intel* $64\,bits$ $2.6\,GHz$, doté de $32\,Go$ de RAM sous *Linux*.

Nous avons effectué nos différentes expériences sur deux bancs d'essais. Un échantillon de 23 instances provenant du banc d'essais COPRIN [3], qui porte sur des problèmes de résolution et un échantillon de 82 instances provenant du banc d'essais COCONUT [4] qui porte sur des problèmes d'optimisation globale sous contraintes numériques. Ces instances ont fait l'objet d'une sélection lors de travaux de membres de l'équipe Ibex portant sur un autre sujet [14]. Elles correspondent à toutes les instances difficiles des séries 1 et 2, mais ne dépassant pas 50 variables [5].

## 5.2 Résultats obtenus par TEC

Sur les 23 instances en résolution, TEC est légèrement meilleur que HC4 en temps de résolution ($-7\%$ en moyenne) mais a une capacité de contraction plus importante ($-53\%$ du nombre de nœuds explorés en moyenne). Notons que l'opérateur X-Newton utilisé après l'opérateur HC4 permet le plus gros de la contraction. En effet, en laissant HC4 comme seul contracteur de la stratégie, le temps de résolution explose et dépasse souvent le temps-limite. En revanche, l'opérateur 3BCID permet une meilleure contraction que TEC sur certaines instances, avec une perte d'un facteur 2 en temps de résolution sur 3 instances (`butcher8-a`, `butcher8-b` et `Synthesis`).

Nous présentons plus en détail les résultats obtenus sur les 82 instances d'optimisation globale sous contraintes. La figure 4 présente le comparatif des opérateurs HC4 et TEC en fonction de leurs temps d'exécution.

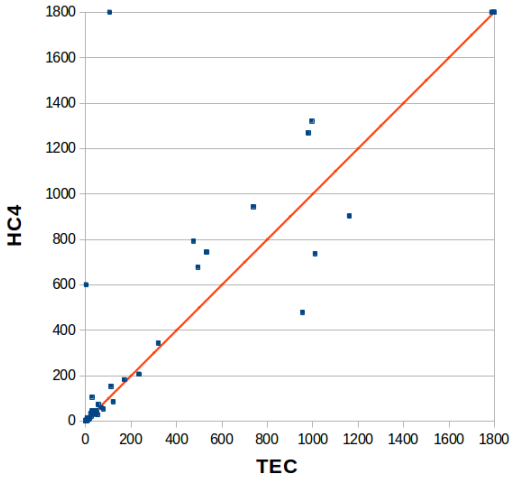


FIGURE 4 – TEC versus HC4 : Les coordonnées de chaque point représente le temps d'exécution (en secondes) requis par les deux stratégies.

Sur 2 instances, TEC obtient un gain en temps supérieur à un facteur 2 par rapport à HC4. En particulier, l'instance `haldmads` est résolue en moins de 120 secondes par TEC alors qu'elle n'est pas résolue par HC4 dans le temps-limite de 30 mn. Le tableau 1 présente les gains détaillés en temps d'exécution de TEC par rapport à HC4.

| | loss $< .2$ | loss [.2,.5] | loss [.5,.7] | loss [.7,.9] | equiv [.9,1.1] | gain [1.1,1.4] | gain [1.4,2] | gain [2,5] | gain $>5$ |
|---|---|---|---|---|---|---|---|---|---|
| $\leq$ 60s | 0 | 2 | 6 | 12 | 13 | 10 | 4 | 2 | 1 |
| $>$ 60s | 0 | 0 | 1 | 4 | 17 | 6 | 2 | 0 | 2 |

TABLE 1 – Gains détaillés de TEC par rapport à HC4. Le gain est exprimé par $\frac{temps(HC4)}{temps(TEC)}$. Les valeurs représentent le nombre d'instances (résolues en moins de 60s et en plus de 60s) correspondant à chaque plage de gains.

Bien que TEC permette d'améliorer significativement le temps d'exécution sur certaines instances difficiles, il est souvent non compétitif avec HC4 sur des instances faciles (rapidement résolues). Une explication à ce phénomène tient au surcoût de la contraction supplémentaire. Le tableau 2 permet de visualiser les gains détaillés en nombre de nœuds explorés par la stratégie utilisant TEC par rapport à celle utilisant HC4. Ces résultats confirment le gain en contraction.

| | loss $< .2$ | loss [.2,.5] | loss [.5,.7] | loss [.7,.9] | equiv [.9,1.1] | gain [1.1,1.4] | gain [1.4,2] | gain [2,5] | gain $>5$ |
|---|---|---|---|---|---|---|---|---|---|
| $\leq$ 60s | 0 | 0 | 0 | 1 | 27 | 13 | 4 | 3 | 2 |
| $>$ 60s | 0 | 0 | 0 | 0 | 11 | 12 | 2 | 4 | 3 |

TABLE 2 – TEC vs. HC4 : Résumé des gains en *#noeuds*

La figure 5 présente le comparatif des opérateurs 3BCID et TEC en fonction de leurs temps d'exécution.

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

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

5. Peu d'instances dépassant cette taille peuvent être traitées aujourd'hui par les optimiseurs déterministes (à intervalles ou non rigoureux d'ailleurs) sans paramétrage ad-hoc

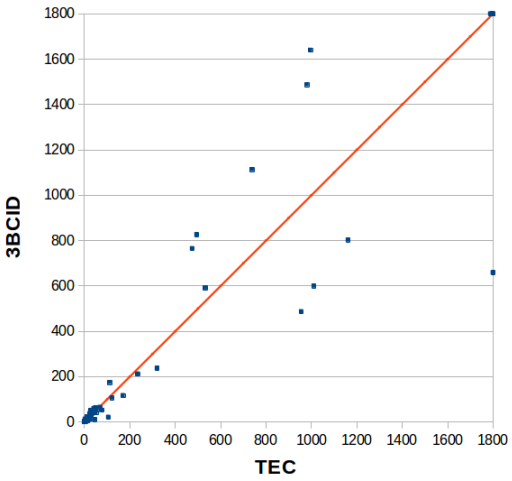


FIGURE 5 – TEC vs. 3BCID (temps CPU)

TEC et 3BCID sont deux opérateurs de contraction forte. Par conséquent, ces deux opérateurs sont coûteux en chaque nœud. Sur la figure 5 on constate que sur certaines instances difficiles, TEC semble plus approprié que 3BCID et sur certaines autres 3BCID obtient de meilleurs résultats. Toutefois, 3BCID permet de résoudre l'instance `ex8_4_5` tandis que TEC ne le permet pas, même en étendant le temps-limite à une heure. Le tableau 3, présente les gains détaillés en temps d'exécution.

| | loss < .2 | loss [.2,.5] | loss [.5,.7] | loss [.7,.9] | equiv [.9,1.1] | gain [1.1,1.4] | gain [1.4,2] | gain [2,5] | gain >5 |
|---|---|---|---|---|---|---|---|---|---|
| ≤ 60s | 1 | 1 | 6 | 12 | 13 | 10 | 5 | 2 | 0 |
| > 60s | 1 | 0 | 4 | 2 | 17 | 2 | 6 | 0 | 0 |

TABLE 3 – TEC vs. 3BCID : Résumé des gains en temps

Ce tableau montre une répartition quasiment uniforme des instances résolues en plus de 60 s. En effet, 3BCID est plus performant sur 7 instances et TEC sur 8 instances pour les instances résolues en plus de 60 s. Pour celles résolues en moins d'une minute, 3BCID est légèrement meilleur. Cette observation nous a poussés à explorer des versions hybrides de ces deux opérateurs, toutefois, sans résultats.

Le tableau 4 permet de visualiser les gains détaillés en nombre de nœuds de l'arbre d'exploration de TEC par rapport à 3BCID.

| | loss < .2 | loss [.2,.5] | loss [.5,.7] | loss [.7,.9] | equiv [.9,1.1] | gain [1.1,1.4] | gain [1.4,2] | gain [2,5] | gain >5 |
|---|---|---|---|---|---|---|---|---|---|
| ≤ 60s | 1 | 1 | 3 | 9 | 24 | 9 | 2 | 1 | 0 |
| > 60s | 1 | 0 | 2 | 7 | 15 | 6 | 0 | 1 | 0 |

TABLE 4 – TEC vs. 3BCID : Résumé des gains en *#noeuds*

La répartition des instances en fonction du gain en nombre de nœuds de l'arbre d'exploration montre que l'opérateur 3BCID semble apporter plus fréquemment une contraction supérieure. Toutefois, cette différence reste mineure.

### 5.3 Résultats obtenus par Graham-TEC

L'opérateur Graham-TEC obtient de meilleurs résultats que l'opérateur TEC pour les problèmes de résolution. Graham-TEC améliore les temps d'exécution sur les 23 instances par rapport à HC4 (−12% en moyenne) et produit une contraction supérieure, avec une réduction moyenne de 63% du nombre de nœuds explorés. Toutefois, Graham-TEC reste moins performant que 3BCID sur ce benchmark (+5% en moyenne en temps d'exécution).

Sur les instances d'optimisation, Graham-TEC améliore aussi l'opérateur TEC. La figure 6 présente le comparatif des opérateurs HC4 et Graham-TEC en fonction de leurs temps d'exécution.

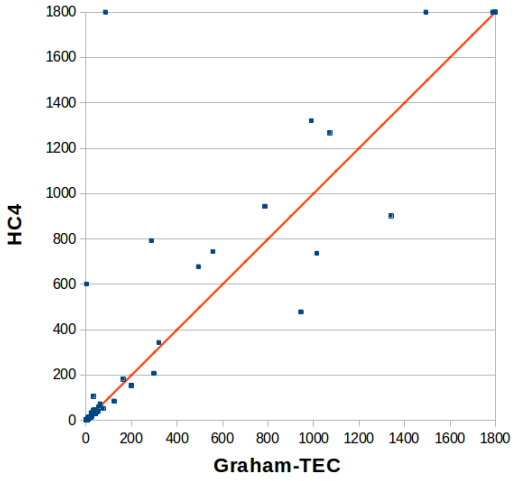


FIGURE 6 – Graham-TEC vs. HC4 : (temps CPU)

Pour les instances résolues en plus de 400 secondes, Graham-TEC est plus performant sur 9 instances alors que HC4 l'est sur 3 instances. Parmi les 9 instances où Graham-TEC est plus performant, 2 d'entres elles ne sont pas résolues par HC4 (`haldmads` et `disc2`). De plus, même en étendant la limite de temps à 3600 s, HC4 ne permet pas de résoudre ces deux instances.

Le tableau 5, présentant les gains détaillés en temps d'exécution montre que HC4 est majoritairement plus performant sur les instances résolues en moins de 60 s.

| | loss < .2 | loss [.2,.5] | loss [.5,.7] | loss [.7,.9] | equiv [.9,1.1] | gain [1.1,1.4] | gain [1.4,2] | gain [2,5] | gain >5 |
|---|---|---|---|---|---|---|---|---|---|
| ≤ 60s | 0 | 2 | 8 | 13 | 13 | 9 | 2 | 2 | 1 |
| > 60s | 0 | 0 | 3 | 3 | 16 | 7 | 0 | 1 | 3 |

TABLE 5 – Graham-TEC vs. HC4 : Résumé des gains en temps

De manière identique à TEC, Graham-TEC est un opérateur de contraction forte qui peut entraîner un surcoût sur des instances faciles.

La figure 7 présente le comparatif des opérateurs 3BCID et Graham-TEC en fonction de leurs temps d'exécution.

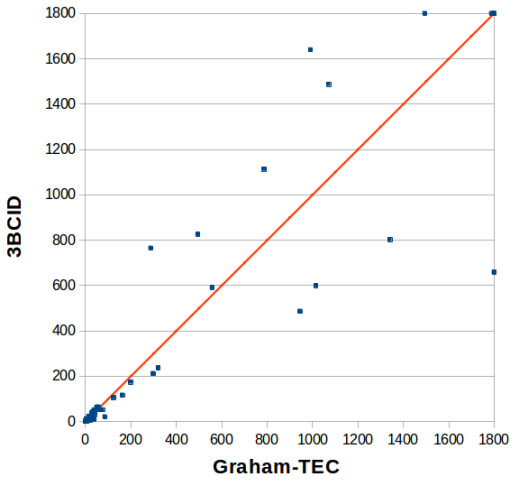


FIGURE 7 – Graham-TEC vs. 3BCID : (temps CPU)

Pour les instances résolues en plus de 400 secondes, Graham-TEC est plus performant que 3BCID sur 7 instances dont une instance (`disc2`) n'est pas résolue par 3BCID, même en étendant le temps-limite à une heure. A l'inverse, l'instance `ex8_4_5`, résolue uniquement par 3BCID, n'est pas résolue par Graham-TEC avec un temps-limite étendu à 3600 s. Le tableau 6 présente les gains détaillés en temps d'exécution.

| | loss < .2 | loss [.2,.5] | loss [.5,.7] | loss [.7,.9] | equiv [.9,1.1] | gain [1.1,1.4] | gain [1.4,2] | gain [2,5] | gain >5 |
|---|---|---|---|---|---|---|---|---|---|
| ≤ 60s | 1 | 1 | 5 | 10 | 19 | 8 | 4 | 2 | 0 |
| > 60s | 1 | 0 | 4 | 4 | 15 | 4 | 2 | 1 | 1 |

TABLE 6 – Graham-TEC vs. 3BCID : Résumé des gains en temps

Pour les instances résolues en plus de 60 s, Graham-TEC obtient un gain d'un facteur supérieur à 2 sur 2 instances tandis que 3BCID n'obtient un gain d'un facteur supérieur à 2 que sur une seule instance. Toutefois, la répartion des instances reste quasiment uniforme. Le tableau 7 présentant le gain en nombre de nœuds de l'arbre d'exploration montre que Graham-TEC permet une contraction plus importante que 3BCID.

| | loss < .2 | loss [.2,.5] | loss [.5,.7] | loss [.7,.9] | equiv [.9,1.1] | gain [1.1,1.4] | gain [1.4,2] | gain [2,5] | gain >5 |
|---|---|---|---|---|---|---|---|---|---|
| ≤ 60s | 1 | 1 | 1 | 6 | 23 | 12 | 4 | 1 | 1 |
| > 60s | 1 | 0 | 2 | 5 | 8 | 12 | 2 | 1 | 1 |

TABLE 7 – Graham-TEC vs. 3BCID : Résumé des gains en *#noeuds*

Ces résultats s'expliquent par le fait que Graham-TEC reste encore coûteux sur certaines instances à cause du travail supplémentaire effectué sur les contraintes inférées. Malgré nos améliorations, ce surcoût est parfois contreproductif, en particulier sur les instances contenant un grand nombre de variables. De ce fait, malgré une contraction potentiellement plus forte, le surcoût occasionné par les contraintes inférées peut, dans certains cas, engendrer une perte de performance à cause d'un faible gain en contraction.

Une étude qualitative, sur l'ensemble des 82 instances étudiées, indique qu'en moyenne Graham-TEC infère 12 contraintes par nœud. On peut constater une augmentation drastique de cette moyenne en fonction du nombre de variables. En effet, sur les instances de moins de 9 variables, la moyenne est de 6 contraintes, pour celles entre 9 et 14 variables, la moyenne est de 8 contraintes tandis que pour les instances de plus de 14 variables, la moyenne grimpe à 35 contraintes par nœud. Toutefois, il est important de remarquer que la perte de performance de Graham-TEC n'est pas directement liée à l'augmentation du nombre de contraintes inférées. Le gain de contraction peut compenser le surcoût engendré, apportant alors un gain de performance même sur des instances avec une moyenne élevée.

## 6 Travaux existants

Le principe de convexification polyédrale de l'ensemble-solution est souvent utilisé en optimisation et en programmation mathématique [17]. Il consiste à calculer une approximation extérieure de l'espace-solution en convexifiant chaque contrainte d'inégalité individuellement. Une approximation polyédrale de l'espace-solution permet de réduire l'espace de recherche et/ou d'améliorer la borne inférieure de la fonction objective à minimiser en faisant appel à un résolveur de programmation linéaire. L'idée de TEC est significativement différente puisque les contraintes linéaires apprises dépendent du nœud courant. Elles ne correspondent pas aux contraintes initiales mais sont déduites des réductions obtenues durant l'exploration de l'arbre TEC.

L'approche de Pelleau et al. [15] s'inspire de travaux effectués en interprétation abstraite pour étendre la notion de boîte dans le CSP numérique. Au lieu de représenter les domaines par des boîtes, chaque paire $(i, j)$ de variables du système est associée à un *octogone* qui peut se voir comme l'intersection de deux boîtes, la boîte « initiale » et une boîte ayant subie une rotation de 45 degrés. Pour chaque chaque paire $(i, j)$, quatre contraintes $\pm x_i \pm x_j \leq c_k$ sont ajoutées. Exprimer les contraintes dans la boîte pivotée revient à remplacer dans leur expression les variables $x_i$ et $x_j$ par leur représentation dans la nouvelle base. Gérer ces domaines étendus revient ainsi à contracter un système avec de nombreuses contraintes additionnelles.

Un contracteur arborescent proche de TEC a été introduit en 2009 [1]. Des sous-systèmes carrés de contraintes d'égalité (avec autant d'équations que d'inconnues et donc un ensemble fini de solution) doivent être préalablement sélectionnés dans le réseau. A chaque noeud de l'arbre de recherche général, une contraction basée sur une exploration arborescente est

lancée pour chaque sous-système. A la différence de l'arbre TEC, un appel au sous-contracteur (HC4) est suivi par un appel à Newton sur intervalles qui exploite le fait que le sous-système est carré, ce qui permet de converger parfois plus vite vers les solutions. Ce contracteur, appelé Box-k, obtient de bons résultats sur des systèmes de contraintes peu denses. Sa limite actuelle est de demander à l'utilisateur d'identifier un ensemble de sous-systèmes qui seront traités par Box-k. Au final, TEC est plus généraliste que Box-k et s'applique sur tout le système (quoique peu de variables soient en pratique bissectées à chaque appel compte-tenu de la faible valeur de TECnodes).

## 7 Conclusion

Cet article a introduit deux nouveaux opérateurs de contraction arborescents obtenant de bonnes performances par rapport à l'état de l'art. L'opérateur Graham-TEC permet de faire un lien entre les deux principaux types d'opérateurs de contraction utilisés dans la plupart des résolveurs à intervalles, à savoir les opérateurs de contraction issus de la programmation par contraintes et ceux reposant sur une convexification polyhédrale. Nos premiers résultats expérimentaux montrent l'intérêt de cette approche qui obtient des gains significatifs par rapport à l'état de l'art sur des instances difficiles d'optimisation globale.

L'opérateur Graham-TEC reste en revanche trop coûteux sur certains types d'instances, en particulier lorsque l'on monte en dimension. En effet, cet opérateur traite toutes les paires de variables et peut ainsi générer un nombre quadratique de contraintes linéaires. Bien que des heuristiques de sélection de contraintes aient été élaborées, il faut encore améliorer cet opérateur pour mieux prendre en compte cette croissance.

## Références


[1] I. Araya, B. Neveu, and G. Trombettoni. A New Monotonicity-Based Interval Extension Using Occurrence Grouping. In *Workshop IntCP*, 2009.

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

[3] I. Araya, G. Trombettoni, and B. Neveu. A Contractor Based on Convex Interval Taylor. In *CPAIOR 2012*, number 7298 in LNCS, pages 1–16, May 2012.

[4] A. Balafrej, C. Bessiere, E.H. Bouyakhf, and G. Trombettoni. Adaptive Singleton-based Consistencies. In *AAAI*, pages 2601–2607. AAAI Press, 2014.

[5] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proc. ICLP*, pages 230–244, 1999.

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

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

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

[9] R. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Inf. Process. Lett.*, 1(4) :132–133, 1972.

[10] R.B. Kearfott and M. Novoa III. INTBIS, a portable interval Newton/Bisection package. *ACM Trans. on Mathematical Software*, 16(2) :152–157, 1990.

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

[12] 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.

[13] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge Univ. Press, 1990.

[14] B. Neveu, G. Trombettoni, and I. Araya. Node Selection Strategies in Interval Branch and Bound Algorithms. *under submission*, page , 2015.

[15] M. Pelleau, C. Truchet, and F. Benhamou. Octagonal Domains for Continuous Constraints. In *Proc. CP, Constraint Programming, LNCS 6876*, pages 706–720. Springer, 2011.

[16] O. Sans. Opérateur arborescent de filtrage pour le CSP numérique. Stage de master, University of Montpellier, 2014.

[17] M. Tawarmalani and N. V. Sahinidis. A Polyhedral Branch-and-Cut Approach to Global Optimization. *Mathematical Programming*, 103(2) :225–249, 2005.

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

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