HABILITATION
À DIRIGER DES RECHERCHES

présentée à

L'UNIVERSITÉ DE NICE – SOPHIA
UFR SCIENCES

ECOLE DOCTORALE STIC

Spécialité : INFORMATIQUE

par

Gilles TROMBETTONI

Résolution de systèmes d'équations : l'essor de la programmation par contraintes sur intervalles

Soutenue publiquement le mardi 8 décembre 2009

après avis des rapporteurs :

M. **Frédéric Benhamou**	Professeur à l'Université de Nantes
M. **Luc Jaulin**	Professeur à l'ENSIETA, Brest
M. **Pascal Van Hentenryck**	Professeur à l'Université Brown (USA)

devant la commission d'examen composée de :

M. **Michel Rueher**	Professeur à l'Université de Nice–Sophia	Président
M. **Frédéric Benhamou**	Professeur à l'Université de Nantes	Examinateur
M. **François Fages**	Directeur de recherche à l'INRIA	Examinateur
M. **Boi Faltings**	Professeur à l'EPFL (Suisse)	Examinateur
M. **Luc Jaulin**	Professeur à l'ENSIETA, Brest	Examinateur
M. **Jean-Pierre Merlet**	Directeur de recherche à l'INRIA	Examinateur
M. **Dominique Michelucci**	Professeur à l'Université de Bourgogne	Examinateur
M. **François Laburthe**	Directeur R&D à Amadeus	Invité

Résumé

Ce mémoire traite essentiellement de la *résolution de systèmes de contraintes* (équations et/ou inégalités) non linéaires (plus précisément, non convexes) sur les réels par des méthodes à *intervalles*. Ces méthodes font principalement appel à des algorithmes de réduction de l'espace de recherche issus de trois communautés aux cultures relativement éloignées : l'analyse numérique (mathématiques), la programmation mathématique (recherche opérationnelle) et la programmation par contraintes (informatique).

Je détaille différentes contributions en programmation par contraintes sur intervalles (PPCI). Parmi elles, trois algorithmes nouveaux forment une stratégie de réduction de l'espace de recherche très compétitive. Un prétraitement est d'abord effectué par un algorithme qui exploite les sous-expressions communes dans les contraintes et permet d'obtenir du filtrage additionnel par la suite. Pendant la recherche arborescente qui suit, un deuxième algorithme de *rognage* (*shaving*) effectue des "points de choix en temps polynomial" pour éliminer des sous-intervalles sans solution aux bornes des intervalles. Il réhabilite l'algorithme existant de 3B-consistance en l'étendant avec, entre autres, de la *disjonction constructive* issue de la programmation par contraintes sur domaines discrets. Le troisième algorithme est la procédure utilisée par l'algorithme de rognage pour éliminer un sous-intervalle donné. Il s'agit d'un algorithme de propagation de contraintes qui exploite la monotonie des fonctions et qui se pose comme alternative aux algorithmes de référence `Box` et `HC4`.

Je souligne deux raisons expliquant l'essor des méthodes à intervalles dans des domaines applicatifs variés comme la robotique, l'automatique ou la chimie : d'abord, le fait que les méthodes à intervalles sont parfois seules capables de traiter certains systèmes de contraintes (par exemple ceux dont les coefficients des contraintes sont connus avec une erreur bornée) ; ensuite, le fait que les algorithmes de PPCI entraînent déjà et vont entraîner encore prochainement des gains de performance importants. Ces progrès offrent l'espoir de compléter ou d'être compétitif avec les méthodes de relaxation linéaire ou convexe.

J'introduis également mes contributions dans trois autres domaines de recherche abordés depuis la thèse : les contraintes fonctionnelles, la décomposition et résolution de systèmes de contraintes (géométriques), la recherche locale pour l'optimisation combinatoire.

Remerciements

Je remercie tout d'abord les rapporteurs et les membres du jury d'avoir accepté cette charge avec autant d'enthousiasme.

Considérant la recherche comme une œuvre collective, je tiens à partager les contributions présentées dans ce mémoire avec les membres de l'équipe COPRIN et avec ceux qui ont étroitement collaboré avec moi sur ces travaux : mon incontournable co-auteur Bertrand Neveu, mais aussi les docteurs que j'ai encadrés ou co-encadrés : Christophe Jermann, Gilles Chabert et Ignacio Araya, ainsi que Marta Wilczkowiak.

Je tiens aussi à remercier pour leur confiance et leurs conseils mes responsables d'équipe depuis la thèse, Michel Rueher et Jean-Pierre Merlet. Merci enfin à mes collègues du département Réseaux et Télécoms de l'IUT pour leur professionnalisme et leur amitié.

Table des matières

Chapitre 1

Introduction et principales contributions

Après la thèse soutenue en 1997 et encadrée par Bertrand Neveu, un post-doctorat de 13 mois à l'EPFL dans le laboratoire de Boi Faltings en 1997, et un poste de maître assistant invité à l'Ecole des Mines de Nantes (avec Patrice Boizumault et Olivier Lhomme) pendant 5 mois, j'ai obtenu en 1998 un poste de maître de conférences en informatique à l'Université de Nice-Sophia. J'effectue depuis mon service statutaire à l'IUT, département Réseaux-télécoms (R&T), situé à Sophia-Antipolis.

J'ai intégré en 1998 l'équipe Contraintes de l'I3S, dirigée par Michel Rueher, et spécialisée en programmation par contraintes et en preuve automatique. En 2002, Michel Rueher, Claude Michel et moi (de l'I3S), Bertrand Neveu (laboratoire des Ponts et Chaussées), Yves Papegay et Jean-Pierre Merlet (INRIA) avons formé l'équipe COPRIN commune aux trois laboratoires/instituts. Le projet de recherche portait essentiellement sur les méthodes à intervalles et la robotique, les méthodes à intervalles étant une approche très pertinente dans la conception de robots parallèles, comme nous le mentionnons au chapitre 3. Le départ de Michel Rueher et Claude Michel de l'équipe pour créer l'équipe CeP à l'I3S ont transformé COPRIN en une équipe INRIA classique, maintenant tournée prioritairement vers la robotique, dans laquelle Bertrand Neveu et moi constituons le noyau des informaticiens.

Contributions

Ce mémoire synthétise mes travaux de recherche depuis la thèse en 1997. Les douze chapitres en annexe correspondent à des articles de *workshop*, de congrès ou de revues publiés de 2004 à 2009.

Méthodes à intervalles

Les chapitres 2 et 3 mettent l'accent sur mes activités plus récentes concernant les méthodes à intervalles pour la résolution de systèmes de contraintes sur les nombres réels. Parmi les différentes contributions, trois algorithmes forment une stratégie de réduction de l'espace de recherche très compétitive :

– Le chapitre A en annexe détaille l'algorithme de pré-traitement `I-CSE` qui réécrit les contraintes en éliminant les sous-expressions communes, ce qui permet d'obtenir du filtrage additionnel par la suite.
– Le chapitre B détaille les algorithmes `CID` et `3BCID` qui sont utilisés pendant la recherche de solutions et étendent l'algorithme de contraction existant `3B`.
– Le chapitre D détaille la contribution la plus importante de ce mémoire (à mon avis). L'algorithme de propagation de contraintes `Mohc` exploite la monotonie des fonctions pour mieux contracter les contraintes individuellement.
  Le chapitre E décrit un algorithme utilisable par `Mohc` pour faire apparaître plus de monotonie dans le système.

Le chapitre 4 retrace l'histoire de mes premières amours (scientifiques) et de mes autres activités.

Décomposition de systèmes de contraintes (géométriques)

Ma thèse portait sur le maintien de solution de problèmes de contraintes *fonctionnelles* par propagation de valeurs (cf. section 4.1). Un algorithme `GPDOF` a été proposé pour traiter un modèle plus général permettant de prendre en compte des sous-systèmes de contraintes pour lesquels on connaît une méthode de résolution. Ce modèle m'a orienté vers la décomposition et la résolution de systèmes de contraintes géométriques :

– Le chapitre G décrit une application de `GPDOF` à la reconstruction de modèles 3D sous contraintes (photogrammétrie, vision par ordinateur).
– Le chapitre H reproduit un *survey* très détaillé sur les différentes méthodes de décomposition de systèmes de contraintes géométriques. Ce travail est le fruit du travail de collaboration avec notre ancien doctorant, Christophe Jermann (avec qui nous avons développé un algorithme exploitant la rigidité des systèmes – cf. section 4.2.2), et avec la communauté française des contraintes géométriques pilotée pendant quelques années par D. Michelucci et P. Schreck.
– Le chapitre I décrit la version la plus aboutie de notre algorithme `IBB` qui permet de résoudre un système décomposé sous-système par sous-système. Chaque sous-système est résolu par des méthodes à intervalles.
  Le chapitre J présente un algorithme entièrement basé sur les intervalles qui généralise `IBB`. Il peut en effet résoudre les systèmes structurés (creux), que ceux-ci soient décomposables ou irréductibles.

Recherche locale pour l'optimisation

Nos travaux sur les algorithmes incomplets d'optimisation ont produit deux principales contributions :

– Le chapitre K donne une description de l'algorithme de recherche locale `ID Walk`. Cette métaheuristique très efficace comprend un ou deux paramètres utilisateur, dont le plus important peut se régler facilement ou automatiquement.
– Les chapitres L et M proposent une approche incomplète pour un problème de placement de rectangles en 2D (ou découpe). L'approche est basée sur `ID Walk`, sur un mouvement rendu très efficace en exploitant une propriété des placements de rectangles et sur une heuristique gloutonne originale.

Chapitre 2

Introduction aux méthodes à intervalles

Dans ce chapitre, nous donnons une introduction aux méthodes à intervalles utilisées pour la résolution de systèmes de contraintes (équations, inégalités). Nous décrivons notamment les algorithmes de contraction/filtrage existants qui sont à l'origine de leur efficacité croissante. Nous détaillerons au chapitre 3 nos contributions en programmation par contraintes sur intervalles (PPCI ; en anglais *ICP*), domaine auquel je m'intéresse de plus en plus depuis 2004. La section 3.6 conclut ce chapitre en soulignant l'essor des méthodes à intervalles en général, et de la PPCI en particulier.

2.1 Notations

Commençons par donner quelques notions de base.

Definition 1 *Pour tout couple de réels a et b vérifiant $a \leq b$, on appelle* **intervalle** *et on note $[a, b]$ l'ensemble : $\{x \in \mathbb{R}$ t.q. $a \leq x \leq b\}$.*

$\mathbb{IR}$ désigne l'ensemble de tous les intervalles.

On désigne par **boîte** *le produit cartésien d'intervalles.*

En pratique, une boîte est un vecteur n-dimensionnel d'intervalles qui définit l'espace de recherche dans lequel se trouvent les valeurs des inconnues. Les bibliothèques manipulant des intervalles utilisent des bornes représentables sur des ordinateurs, notamment des rationnels ou des nombres flottants. Nous supposons par la suite que a et b sont des nombres flottants.

Les symboles désignant un intervalle de $\mathbb{IR}$ ou une boîte de $\mathbb{IR}^n$ seront mis entre crochets[1]. Notamment, $[x] \in \mathbb{IR}$ désigne l'intervalle associé à la variable x. La borne inférieure (resp.

[1] D'autres notations courantes utilisent des caractères majuscules ou en gras [166].

supérieure) de l'intervalle $[x]$ est notée $\underline{x}$ (resp. $\overline{x}$). On définit `Diam`($[B]$), le *diamètre* de la boîte $[B]$, par la taille ($\overline{x}-\underline{x}$) de la plus grande dimension $[x]$ de $[B]$. On définit `Perim`($[B]$), le *périmètre* de la boîte $[B]$, par la somme des diamètres de ses composantes. L'opérateur `Mid`($[x]$) retourne le point milieu de l'intervalle $[x]$ (en pratique, un flottant proche du réel au milieu des bornes). La plupart des opérations ensemblistes peuvent s'effectuer sur des boîtes, comme l'inclusion et l'intersection. L'opérateur `Hull` d'*enveloppe* (en anglais : *hull*) permet d'approximer un ensemble S de boîtes par la plus petite boîte (dite *extérieure*) contenant S. Comme nous utilisons peu de matrices dans nos travaux, nous désignons les entités zéro-dimensionnelles par des caractères minuscules mais, de manière moins usuelle, nous notons les vecteurs, comme les matrices, en caractères majuscules.

2.2 Extension d'une fonction réelle aux intervalles

L'extension d'une fonction réelle f aux intervalles permet de raisonner sur des ensembles de valeurs (contiguës) plutôt que sur des points. Une fonction $[f]$ étendue aux intervalles, que l'on appelle *extension*, effectue des calculs *conservatifs* sur des opérandes qui sont des intervalles, c'est-à-dire que l'image par $[f]$ est un intervalle qui contient tous les résultats possibles de f quand on sélectionne un réel dans chaque opérande. Plus formellement, en considérant une fonction $f(x_1, ..., x_n)$ de $\mathbb{R}^n$ vers $\mathbb{R}$, et une boîte $[B] = \{[x_1], [x_2], ..., [x_n]\}$, on appelle *extension* aux intervalles (conservative) de f, une fonction intervalle $[f]$ de $\mathbb{IR}^n$ vers $\mathbb{IR}$ qui vérifie :

$$[f]([B]) \supseteq \{f(B) \ t.q. \ B \in [B]\}$$

On peut généraliser à des fonctions de $\mathbb{R}^n$ vers $\mathbb{R}^m$ qui retournent des vecteurs de dimension m.

On note $[f]_O$ l'extension optimale, et $[f]_O([B])$ (image obtenue par *évaluation* optimale) est l'intervalle conservatif le plus petit, c'est-à-dire l'enveloppe de l'image de $[B]$ par f. Il existe plusieurs extensions célèbres, la difficulté pour une fonction donnée étant de trouver l'extension qui produit, à bas coût, l'image optimale ou une approximation fine de l'image optimale.

2.2.1 Arithmétique des intervalles, extension naturelle

La bibliothèque `Ibex` [52, 50] utilisée pour développer nos algorithmes, comme la plupart des outils implémentant les techniques d'intervalles, comprend les opérateurs arithmétiques $+$, $-$, $\times$, $/$, et d'autres fonctions élémentaires (univariées) : *exp*, *log*, x^a (a entier), *sqrt*, *sin*, *cos*, *tan*, *arcsin*, *arccos*, *sinh*, *cosh*, *tanh*, *arctan*, *arcsh*, *arccosh* et *arctanh*. Pour l'ensemble de ces **fonctions élémentaires**, **"l'arithmétique" des intervalles** calcule rapidement l'image optimale (aux arrondis près). Par exemple, on définit la somme de deux intervalles de la manière suivante : $[x] + [y] = [a, b] + [c, d] = [a + c, b + d]$. De manière générale, les bornes obtenues pour les fonctions élémentaires s'obtiennent par analyse des ensembles de définition et des parties monotones de la fonction. L'image d'un intervalle par une fonction élémentaire f s'exprime en fonction des bornes des opérandes et des valeurs correspondant aux changements de sens de

variation de f sur $[x]$ (cf. [219, 49]). Par exemple, $sqr([-4,2]) = [0,16]$. Autre exemple : la division $[x]/[y]$ donne l'intervalle $]-\infty,+\infty[$ quand $[y]$ contient 0 (mais n'est pas réduit à 0).

Calculer l'image d'une fonction intervalle quelconque peut se faire par composition des opérateurs élémentaires. On utilise l'arithmétique des intervalles en chaque opérateur, c'est-à-dire en remontant l'arbre syntaxique de l'expression f et en appliquant récursivement les définitions élémentaires. C'est ce qu'on appelle l'extension *naturelle* $[f]_N$ d'une fonction. Par exemple, en considérant $f(x) = x^3 - 3x^2 + x$, l'image de $[3,4]$ par la fonction $[y] = [f]_N(x)$ donne : $[y] = [3,4]^3 - 3 \times [3,4]^2 + [3,4] = [27,64] - 3 \times [9,16] + [3,4] = [-18,41]$.

2.2.2 Causes d'évaluation non optimale

Pour n'importe quelle fonction élémentaire univariée f mentionnée ci-dessus, l'arithmétique des intervalles produit une image $[f]_N([B])$ optimale (aux arrondis près). Notons aussi que l'extension naturelle, comme toute autre extension par définition, trouve l'image optimale d'une boîte *dégénérée* dont tous les intervalles sont réduits à un point, cas pour lequel une fonction $[f]$ coïncide évidemment avec f. Malheureusement, trois grandes causes empêchent d'atteindre facilement l'optimalité dans le cas général. Les problèmes d'arrondis liés à l'implantation des intervalles à bornes flottantes, mais aussi deux autres causes intrinsèques au calcul sur intervalles : le manque de continuité de la fonction dans la boîte et le problème de la dépendance (occurrences multiples des variables dans l'expression).

Problèmes d'arrondis pour les intervalles à bornes flottantes

Le fait que les intervalles gérés sur un ordinateur ont comme bornes des nombres flottants entraîne que tout calcul sur intervalles doit effectuer un arrondi extérieur de son intervalle résultat afin qu'il soit conservatif. Un arrondi vers $-\infty$ est effectué pour la borne de gauche et un arrondi vers $+\infty$ est effectué pour la borne de droite. La communauté qui se préoccupe de ces calculs avec des bornes flottantes [246, 19] a développé des algorithmes pour chaque fonction élémentaire afin de minimiser le nombre de flottants "perdus". La plupart des bibliothèques d'arithmétique par intervalles existantes proposent des opérateurs arithmétiques ($+$, $-$, $\times$, $/$) optimaux (qui perdent moins que la largeur entre deux flottants) et des algorithmes sophistiqués qui limitent la surestimation pour les opérateurs plus complexes, comme les fonctions trigonométriques.

Continuité

Dans sa première définition [211], l'arithmétique des intervalles suppose que la fonction f est continue dans la boîte $[B]$. L'arithmétique des intervalles dite *étendue*, en pratique utilisée dans la plupart des outils actuels, et en passe de devenir la norme IEEE qui sera bientôt adoptée, lève cette restriction afin de pouvoir jouer son rôle de calcul "ensembliste". Par exemple, si $\sqrt{[-7,4]}$ provoque une erreur dans l'arithmétique classique, l'arithmétique étendue permet au contraire de renvoyer l'image $[0,2]$. Ce raisonnement ensembliste fait malheureusement perdre l'espoir d'un calcul optimal en temps polynomial. Prenons pour exemple $f(x) = (\frac{1}{x})^2$, avec

$[x] = [-1, 1]$. L'évaluation de $\frac{1}{x}$ donne $[-\infty, -1] \cup [1, +\infty]$, dont l'enveloppe est $[-\infty, +\infty]$, et $[-\infty, +\infty]^2 = [0, +\infty]$. Or, l'image optimale est $[1, +\infty]$ et peut être obtenue en faisant un point de choix sur les deux sous-parties continues ($[-\infty, -1]$ et $[1, +\infty]$) obtenus par $\frac{1}{x}$, au lieu d'en faire l'enveloppe.

Problème de dépendance (occurrences multiples de variables)

La cause la plus documentée de surestimation de l'image (ou *pessimisme*) est le fait qu'une même variable apparaisse plusieurs fois dans l'expression f. Ce phénomène est souvent désigné par problème de *dépendance* (entre les différentes occurrences). L'exemple trivial est que la fonction $f(x) = x - x$, avec $[x] = [-1, 1]$, renvoie la même image $[-2, 2]$ que $x - y$, au lieu de $[0, 0]$. Cela explique aussi dans l'exemple $f(x) = x^3 - 3x^2 + x$ que l'intervalle $[-18, 41]$ obtenu par évaluation naturelle dans $[x] = [3, 4]$ est très pessimiste par rapport à l'intervalle image $[3, 20]$ optimal (que nous obtiendrons grâce à une autre extension basée sur la monotonie de la fonction). Le problème de dépendance s'observe dans le cas général ou pour les opérateurs arithmétiques binaires, mais évidemment pas pour les fonctions élémentaires qui sont unaires.

Une des conséquences théoriques fâcheuses du problème de dépendance est que $\mathbb{IR}$ n'est pas un groupe pour l'addition ($[a] - [a] \neq [0, 0]$) et $\mathbb{IR} \setminus \{0\}$ n'est pas un groupe pour la multiplication ($\frac{[a]}{[a]} \neq [1, 1]$). On observe en fait généralement une inclusion stricte. De même, $[a]([b] + [c]) \subset [a] \times [b] + [a] \times [c]$.

Notons que la surestimation liée aux occurrences multiples diminue avec la taille des boîtes, ce qui justifie (encore plus) le principe de *brancher et contracter* utilisé pour résoudre les systèmes de contraintes (cf. section 2.3) : le branchement réduit la taille des intervalles, ce qui limite la surestimation liée aux occurrences multiples.

Quelques extensions de fonctions aux intervalles

On trouve dans [178] un résultat théorique négatif établissant que trouver l'image optimale d'un polynôme est un problème NP-difficile. Sauf si $P = NP$, aucune extension de fonction sur intervalles "peu coûteuse" permet de calculer optimalement l'image d'une boîte. Une des préoccupations de l'analyse par intervalles est donc de trouver des extensions de fonctions qui permettent de bien approximer l'image d'une fonction.

En plus de l'extension naturelle vue plus haut, deux autres extensions sur intervalles sont relativement utilisées en pratique dans les outils de résolution. L'*extension de Taylor* (à l'ordre 1 ou 2) utilise les dérivées des fonctions [219]. Sans entrer dans les détails, l'extension naturelle et l'extension de Taylor sont incomparables. Cette dernière donne généralement une approximation bien plus mauvaise que l'extension naturelle quand la boîte $[X]$ est grande (soit au début de la recherche de solutions, quand on résout les systèmes de contraintes par des méthodes à intervalles) ; elle donne au contraire une bonne approximation quand la boîte est petite (soit en bas de l'arbre de recherche, quand la boîte courante converge vers une solution).

L'*extension basée sur la monotonie* $[f]_M$ de $\mathbb{IR}^n \to \mathbb{IR}$ de f est plus coûteuse que l'extension

naturelle $[f]_N$. Comme l'extension de Taylor, $[f]_M$ utilise également les dérivées partielles de la fonction, non pas pour approximer linéairement la fonction, mais pour détecter si elle est monotone. Soit $[y] = [f]_M([X])$. $[f]_M$ effectue un calcul séparé de $\underline{y}$ et $\overline{y}$ de la manière suivante :

1. Soit x_i une des variables du vecteur X. On calcule $[y_i] = [\frac{\partial f}{\partial x_i}]([X])$.
2. Si $[y_i]$ ne contient pas 0 (par exemple $[y_i]$ est positif ou nul), cela signifie que f est monotone (croissante) par rapport à x_i dans la boîte $[X]$.
3. On sait alors que l'on peut remplacer $[x_i]$ par $\overline{x_i}$ (resp. $\underline{x_i}$) dans l'expression f pour le calcul de $\overline{y}$ (resp. $\underline{y}$) :
   - $\overline{y} = \overline{[f](..., \overline{x_i}, ...)}$
   - $\underline{y} = \underline{[f](..., \underline{x_i}, ...)}$

Reprenons l'exemple de la fonction (univariée) $[y] = [f](x) = x^3 - 3x^2 + x$ dont l'image de $[3, 4]$ obtenue par l'extension naturelle était $[-18, 41]$. La dérivée est $f'(x) = 3x^2 - 6x + 1$ et $f'([3, 4]) = [3, 30]$. Comme $[3, 30] > 0$, $[f]$ est monotone croissante sur $[x] = [3, 4]$. On obtient alors $[y] = [f]_M([3, 4]) = [f(3), f(4)] = [3, 20]$.

De manière (plus) générale, si $f(x_1, x_2, x_3)$ est croissante par rapport x_1, décroissante par rapport à x_2 et non détectée monotone par rapport x_3 dans $[X] = \{[x_1], [x_2], [x_3]\}$, alors :

$$[f]_M([X]) = [\,\underline{[f]_N(\underline{x_1}, \overline{x_2}, [x_3])}, \overline{[f]_N(\overline{x_1}, \underline{x_2}, [x_3])}\,] \subseteq [f]_N([X])$$

L'intérêt principal de l'extension basée sur les monotonies est de faire disparaître le problème de dépendance lié à la variable x_i si f est monotone par rapport à x_i, puisque $[x_i]$ est remplacé par l'une de ses bornes dans l'évaluation. Autrement dit, nous avons la propriété : $[f]_M([X]) \subseteq [f]_N([X])$.

Calcul matriciel

On peut étendre aussi l'arithmétique des intervalles au calcul matriciel. Une matrice intervalle contient des coefficients intervalles. Notamment, différents algorithmes de traitement d'un système de n équations $f = f_1, ..., f_n$ à m inconnues $X = \{x_1, ..., x_n\}$ utilisent la matrice jacobienne intervalle $[J]$ de f. $[J]$ est une matrice $m \times n$ dont chaque élément est $[\frac{\partial f_j}{\partial x_i}]([X])$, la dérivée partielle de la fonction f_j par rapport à la variable x_i évaluée (de manière naturelle ou autre) sur la boîte $[X]$. Le calcul matriciel sur intervalles est particulièrement utile pour la résolution de systèmes d'équations linéaires sur intervalles, brique essentielle des algorithmes d'analyse par intervalles (cf. section 2.4.1).

La surestimation liée au phénomène de dépendance apparaît aussi dans le calcul matriciel à cause des occurrences multiples de variables.

2.3 Schéma de résolution des systèmes d'équations

La première motivation des méthodes à intervalles est la *fiabilité* des calculs sur les réels à l'aide d'un ordinateur, c'est-à-dire le contrôle des erreurs d'arrondis dues aux calculs sur les flottants. Par exemple, pour évaluer la circonférence d'un cercle, on calcule l'expression $2 \times [3.14, 3.15] \times [1.9, 2.1] = [11.932, 13.230]$. Cet exemple simple illustre les deux principales origines des approximations par des intervalles : un nombre Π qui n'est pas représentable sur un ordinateur et un rayon qui est connu approximativement à cause d'une erreur de mesure ou de fabrication.

La deuxième motivation des méthodes à intervalles, un peu moins connue, est le traitement *complet* des systèmes de contraintes (équations, inégalités) non-linéaires, où chaque inconnue a un intervalle initial de valeurs possibles. Par complétude, nous entendons la capacité de trouver toutes les solutions (sans perte) ou de prouver qu'il n'y en a pas. C'est bien le traitement des systèmes de contraintes qui nous intéresse dans ce mémoire.

Le processus de résolution naïf suivant permet de trouver l'ensemble des solutions réelles à un système d'équations $F(X) = 0$ (F est un vecteur de fonctions) dans un espace de recherche initial qui est une boîte $[X]$ de taille quelconque. Il est basé sur un principe de *branch and prune* (brancher et élaguer) qui enchaîne les deux opérations suivantes à partir de $[X]$:

- Elaguer : On calcule l'image de $[X]$ par F. S'il existe une fonction f de F telle que $0 \notin [f]([X])$, cela signifie bien que l'équation correspondante, et donc le système entier, n'a pas de solution dans $[X]$. On désigne souvent cette procédure par *test d'existence*.
- Brancher : Dans le cas contraire, si $\forall f \in F, 0 \in f([X])$, l'intervalle de l'une des variables de X est coupé en deux parties, typiquement en son milieu, pour lancer récursivement la recherche de solution sur la boîte de gauche et sur la boîte de droite. Cette opération de **bissection** rend bien le processus de résolution combinatoire.

La résolution se termine quand la boîte courante devient **atomique**, c'est-à-dire quand le diamètre de la boîte est inférieur à une précision ω donnée.

Ce principe de résolution est malheureusement inefficace en pratique, même quand le nombre d'inconnues est très petit. Les outils de résolution actuels remplacent ainsi le schéma "brancher et élaguer" par un schéma "brancher et contracter" (*branch and contract*) où des algorithmes de contraction, appelés aussi **contracteurs**, au lieu de simplement déduire qu'une boîte ne contient pas de solution, sont en plus capables de réduire aux bornes les intervalles de la boîte courante sans perte de solution. Nous décrivons ci-dessous à des fins didactiques un schéma assez fréquemment utilisé par les outils existants pour trouver l'ensemble des solutions du système.

L'algorithme `RechercheSolutions` (cf. Algorithm 1) développe un arbre de recherche en profondeur d'abord (en utilisant une file dans cette version itérative). Il effectue une exploration combinatoire exhaustive de l'espace de recherche afin d'encadrer au plus près l'ensemble des solutions. L'algorithme débute avec une boîte $[X]$ initiale, c'est-à-dire avec un intervalle de recherche pour chaque variable du système. L'algorithme termine avec un ensemble de boîtes atomiques stockées dans deux ensembles : `solutionsNonCertifiees` et `solutionsCertifiees`. L'algorithme de résolution renonce à couper une boîte atomique qui est susceptible de contenir une solution.

Algorithm 1 `RechercheSolutions` (**in** F, X, ϵ; **out** `solutionsNonCertifiees`, `solutionsCertifiees`)

```
L ← {[X]}   /* On part d'une boîte initiale [X] */
while L ≠ ∅ do
  [B] ← L.front() ; L.remove([B]) /* Sélection d'une boîte */
  /* Contraction de [B] : */
  [B] ← ContractICP([B],...)
  if [B] ≠ ∅ then [B] ← ContractRelax([B],...) end if
  if [B] ≠ ∅ then ([B], certified?) ← ContractNewton([B],...) end if
  if [B] ≠ ∅ then
    if Diam([B]) < ε then
      if certified? then
        solutionsCertifiees ← solutionsCertifiees ∪ {[B]}
      else
        solutionsNonCertifiees ← solutionsNonCertifiees ∪ {[B]}
      end if
    else
      /* Branchement : */
      ([B_g], [B_d]) ←Bisect([B], X, F)
      L ← L ∪ {[B_g]} ∪ {[B_d]}
    end if
  end if
end while
```

Les procédures principales de cet algorithme sont :

- `ContractICP` : les contracteurs de programmation par contraintes sur intervalles ont un rôle de filtrage ou contraction qui permet de réduire la boîte $[B]$ sur les bornes sans perte de solution. Dans le cas idéal, une contraction peut mener à un intervalle vide (cas où $[B]$ devient vide), démontrant l'absence de solution dans la boîte.
- `ContractRelax` : les contracteurs de relaxation linéaire approximent le système par un ensemble de fonctions linéaires (ou convexes) et contractent la boîte $[B]$ à l'aide d'un algorithme du simplexe ou autre.
- `ContractNewton` : les opérateurs d'analyse numérique, comme les opérateurs de programmation par contraintes, ont un rôle de filtrage.
  Si certaines conditions sont respectées, ces opérateurs sont en plus capables de certifier que la boîte $[B]$ contient une solution unique (`certified?` devient *vrai*). Dans ce cas, quelques itérations de l'algorithme de point-fixe (Newton intervalles) permettent de converger très rapidement vers une boîte atomique contenant la solution certifiée.
- `Bisect` : quand les opérateurs ci-dessus n'ont plus d'effet de contraction sur $[B]$, l'intervalle de l'une des variables de X est coupé en deux parties, typiquement en son milieu, pour lancer récursivement la recherche de solution sur la boîte de gauche $[B_g]$ et sur la boîte de droite $[B_d]$.

Ce dernier opérateur de bissection rend bien le processus combinatoire, mais l'explosion combinatoire se produit seulement si les autres opérateurs algorithmiques sont inefficaces.

Remarque 1 : différents degrés de fiabilité

On peut classer les outils de résolution qui parcourent l'espace de recherche de manière exhaustive en trois catégories selon le choix des opérateurs de contraction et selon leur fiabilité :

1. *Outils exhaustifs, non complets* :
   Si l'outil n'utilise pas une implantation fiable de ses contracteurs, alors il peut perdre des solutions, tout en pouvant retourner des solutions parasites (boîtes atomiques sans solution). C'est le cas de `Baron` [250] qui n'utilise pas d'opérateur `ContractNewton` et qui utilise des opérateurs de type `ContractRelax` non fiables.
2. *Outils complets* :
   Si l'outil utilise des implantations fiables de `ContractRelax` et de `ContractICP`, alors aucune solution n'est perdue. En revanche, s'il n'utilise pas d'opérateur `ContractNewton` (c'est-à-dire si aucune distinction n'est faite entre les ensembles `solutionsNonCertifiees` et `solutionCertifiees`), alors il peut retourner des solutions parasites.
3. *Outils complets et fiables* :
   Si, en plus, l'outil utilise un opérateur `ContractNewton`, alors toutes les solutions sont trouvées et certaines solutions parasites sont éliminées.

Malheureusement, `ContractNewton` ne peut en théorie pas toujours éliminer toutes les solutions parasites, c'est-à-dire ne peut pas certifier toutes les solutions. Ainsi, il n'y a pas de différence théorique entre les deux derniers cas. En pratique néanmoins, on peut dire que l'utilisation de `ContractNewton` permet de calculer un sur-ensemble plus fin de l'ensemble des solutions.

On peut bien-sûr ranger dans la catégorie des *outils non exhaustifs/complets* tous les autres outils qui ne parcourent pas l'espace de recherche de manière exhaustive : méthodes numériques classiques (locales), recherche locale, algorithmes génétiques, à particules, etc.

Remarque 2 : optimisation globale

Le schéma ci-dessus est sensiblement complexifié pour minimiser une fonction objectif sous contraintes. Sans détailler, en plus des contracteurs, l'algorithme de *branch and bound* met à jour au cours de la recherche un minorant l et un majorant u de l'objectif jusqu'à ce que $u - l < \epsilon$:

– Un majorant est souvent obtenu par recherche locale. Sa certification par une variante de `ContractNewton` est alors une opération coûteuse.
– Un minorant est obtenu par une approximation linéaire (ou convexe) conservative de la fonction objectif (cf. section 2.4.2). La meilleure solution de l'approximation fournit en effet une borne inférieure de l'objectif.

2.4 Algorithmes de contraction/filtrage

Cette section donne des précisions sur les algorithmes de contraction issus de l'analyse par intervalles (`ContractNewton`), de la programmation mathématique (`ContractRelax`) et de la programmation par contraintes (`ContractICP`).

2.4.1 Opérateurs d'analyse par intervalles

Nous désignons par *Newton intervalles* multivarié une catégorie d'algorithmes qui sont une adaptation aux intervalles des algorithmes d'analyse numérique. Le principe du Newton intervalles est le suivant :

- On part d'une boîte initiale $[X]$.
- On calcule jusqu'à point-fixe : $[X] \leftarrow [X] \cap N([X])$.
- Si $N([X]) \subset [X]$, alors garantie de solution unique dans $[X]$.

Soit $\dot{X}$ un point de $[X]$, par exemple $\texttt{Mid}([X])$. On définit $N([X]) := \dot{X} + [Y_s]$, où $[Y_s]$ est la boîte obtenue par résolution du système linéarisé $[A][Y] + [B] = 0$.

Par exemple, $[A]$ est la matrice jacobienne du système F (chaque élément est l'intervalle $[a_{ji}] = [\frac{\partial f_j}{\partial x_i}]_N([X])$) ; $[B]$ est $F(\dot{X})$; $[Y] = [X] - \dot{X}$.

Newton intervalles comprend en fait un grand ensemble de variantes qui diffèrent par :

- l'utilisation de dérivées ou de pentes ;
- la matrice utilisée : jacobienne, de Hansen [118], etc ;
- l'utilisation de calcul formel sur la matrice. Par exemple, Jean-Pierre Merlet utilise Maple pour manipuler formellement les coefficients de la matrice afin de réduire les occurrences multiples de variables ;
- la linéarisation : la manière de linéariser le système (à chaque itération de Newton) influe sur sa résolution et donne le nom à différentes variantes de Newton intervalles. Citons Krawczyck, Borsuc, Kantorovitch (utilisant les dérivées secondes) ;
- la résolution du système linéaire : inversion de matrice, Gauss-Seidel, Hansen-Bliek [118], LU (variante d'une méthode de pivot de Gauss), etc ;
- un préconditionnement pour rendre la contraction effective.

Le préconditionnement le plus employé est un préconditionnement à gauche qui multiple $[A]$ à gauche par une matrice ponctuelle $P = (\texttt{Mid}[A])^{-1}$. Ce préconditionnement vise à mettre sur la diagonale de $P.[A]$ un intervalle $[a'_{ii}]$ plus large et des intervalles $[a'_{ji}](j \neq i)$ plus petits, ce qui va permettre une contraction plus forte de l'intervalle $[x_i]$. Par exemple, Gauss-Seidel effectue, jusqu'à obtention d'un point fixe, une opération de contraction sur chacune des lignes (équation i) de $[A][X] + [B] = 0$:

$$[a'_{1i}]x_1 + [a'_{2i}]x_2 + ... + [a'_{ii}]x_i + ... + b_i = 0$$

L'opération suit l'isolement de la variable x_i, comme suit :

$$[x_i] \leftarrow [x_i] \cap -\frac{[a'_{1i}]x_1 + ... + [a'_{(i-1)i}]x_{i-1} + [a'_{(i+1)i}]x_{i+1} + ... + b_i}{[a'_{ii}]}$$

Ainsi, le préconditionnement effectue en quelque sorte un changement de repère de tout le système linéaire qui améliore cette opération atomique. Le préconditionnement rend global le contracteur Newton intervalles, c'est-à-dire qu'il raisonne sur le système dans son ensemble pour apporter une contraction sur $[X]$.

L'opérateur de Newton intervalles implanté dans `Ibex` [50, 52] utilise la matrice de Hansen préconditionnée à gauche et résout le système linéaire avec un algorithme de Gauss-Seidel.

2.4.2 Relaxation linéaire ou convexe

La relaxation convexe est une approche de base utilisée en optimisation globale pour minimiser un objectif sous contraintes. Elle est utilisée pour approximer les contraintes non-linéaires et/ou la fonction objectif. L'approche est utilisée par exemple dans les outils `GlobSol` de Kearfott [164], `QuadSolver` de Lebbah, Rueher, Michel [183] ou bien le célèbre `Baron` de Sahinidis et Twarmalani [250]. Le principe consiste simplement à approximer le système non-linéaire par un ensemble d'inégalités linéaires englobantes traitées généralement par un algorithme du simplexe. Trois remarques importantes :

- Le système linéaire englobant explique le terme relaxation. En effet, le système linéarisé contient plus de solutions que le système non-linéaire. Relaxer linéairement l'objectif à minimiser permet de trouver un minorant (un point plus bas) à la valeur de l'optimum global.
- Si la linéarisation se fait sans soin particulier concernant l'implantation sur les nombres flottants, celle-ci peut parfois, dans des cas pathologiques/singuliers, s'avérer être non conservative, d'où la perte possible de solution ou de l'optimum global. `QuadSolver` et `GloptSol` utilisent des procédures rigoureuses pour calculer les coefficients flottants des relaxations linéaires, contrairement à `Baron` [250].
- De même, la plupart des algorithmes du simplexe disponibles manipulent des nombres flottants et sont donc non rigoureux à cause des problèmes d'arrondis. Pour obtenir une borne supérieure de l'optimum qui soit fiable, `QuadSolver` implémente par exemple une procédure peu coûteuse de post-traitement introduite par Neumaier et Shcherbina [220], basée à la fois sur des arrondis *dirigés* et de l'arithmétique d'intervalles.

Pour réduire l'espace de recherche en utilisant le système de contraintes (avec ou sans optimisation), l'opérateur `ContractRelax` réduit les bornes de chaque variable de la manière suivante : pour chaque variable, chacune des deux bornes est modifiée par un appel à un simplexe sur le système linéaire : la borne de gauche est maximisée et la borne de droite est minimisée pour obtenir un intervalle réduit[2].

Nous donnons à titre illustratif quelques détails sur un des algorithmes implémentant `ContractRelax`. Cet algorithme de contraction, dénommé `Quad` et en partie sophipolitain [183], mélange

[2]En pratique, on peut ne pas traiter toutes les variables en utilisant le dual...

de la relaxation linéaire avec le célèbre algorithme `HC4` de PPCI décrit à la section 2.4.3. Le principe de `Quad` est le suivant :

– Linéarisation des termes quadratiques des contraintes[3].
– Utilisation du sous-système linéaire résultant pour réduire les bornes des n variables en faisant au plus $2n$ appels au simplexe, comme décrit ci-dessus.
– Propagation des réductions obtenus par `HC4` (cf. section 2.4.3) sur l'ensemble du système, incluant notamment les contraintes non polynomiales.

Outre les différences de fiabilité (perte de solution en cas de relaxation non conservative ou d'utilisation non rigoureuse de l'algorithme du simplexe), les différents algorithmes de cette communauté se distinguent par leur manière d'obtenir une relaxation convexe du système ou de la fonction objectif. Le lecteur intéressé peut consulter le livre de Hanif Sherali et Warren Adams pour avoir une description exhaustive de différentes Techniques [267] de Reformulation et de Linéarisation (RLT). Le solveur Baron utilise aussi une méthode polyhédrale très performante [280]. Plus proche de l'analyse par intervalles, mentionnons enfin que l'on peut approximer linéairement le système en implémentant les fonctions élémentaires à l'aide de l'*arithmétique affine* [173]. Cette approche semble prometteuse comme le montrent les résultats obtenus par l'algorithme de contraction `CIRD` de Vu et Sam-Haroud [298].

Compte-tenu du succès de ces approches en optimisation globale combinatoire ou continue, il semble important de plus les confronter encore aux méthodes de l'analyse par intervalles ou de la programmation par contraintes sur intervalles. Arnold Neumaier et le projet européen COCONUT ont d'ores et déjà contribué à transférer l'algorithme `2B` (décrit ci-dessous) dans `Baron` [249]. L'algorithme `CIRD` [298], l'outil d'optimisation de Lebbah, Rueher, Michel, Goldsztejn [183, 245] et l'outil de Baharev [17] constituent des premiers exemples d'intégration réussie d'algorithmes de relaxation linéaire et de programmation par contraintes.

2.4.3 Algorithmes de propagation de contraintes

Ces algorithmes de contraction sont issus de la programmation par contraintes sur intervalles (`ContractICP`). Ils raisonnent sur une contrainte individuellement, la "relaxation" correspondante éliminant (provisoirement) toutes les autres contraintes, et réduisent les intervalles des variables correspondantes sans perte de solution pour cette contrainte. Un *algorithme de propagation*, proche du célèbre `AC3` utilisé dans les CSP sur domaines finis, assure jusqu'à l'obtention d'un point quasi-fixe l'intersection incrémentale de la boîte courante avec la boîte calculée pour chaque contrainte. Le principe de l'algorithme de propagation est le suivant. Au départ, on empile toutes les contraintes dans une *queue de propagation*. On effectue ensuite les opérations suivantes itérativement jusqu'à ce que la queue devienne vide :

1. On sélectionne et on élimine de la queue la première contrainte c.
2. On applique sur c une procédure de contraction/révision (appelée *Revise*) qui réduit les intervalles des variables V de c.

[3]Les termes de degré supérieur à 2 peuvent être transformés en termes quadratiques ou bilinéaires. Les termes non polynomiaux sont laissés intacts, remplacés seulement par une variable auxiliaire...

3. Pour toute variable v de V, si $[v]$ est (suffisamment) réduit, on ajoute en fin de queue chaque contrainte c' ($c' \neq c$) impliquant v, si c' ne se trouve pas déjà dans la queue.

Les algorithmes de propagation de contraintes existants diffèrent généralement entre eux par leur procédure de révision. Par exemple, l'algorithme `HC4` utilise une procédure `HC4-Revise` ; l'algorithme `Box` utilise la procédure `BoxNarrow` ; les algorithmes nouveaux `Mohc` et `PolyBox` présentés au chapitre suivant font appel aux procédures `Mohc-Revise` et `PolyBoxRevise` respectivement.

Les algorithmes existants gèrent un paramètre de précision pour obtenir plus ou moins rapidement un point-fixe de la propagation[4]. Ce paramètre utilisateur est une taille d'intervalle fixe ou relative à l'intervalle considéré. Par exemple, `Ibex` gère un paramètre τ_{propag} fixé par défaut à 10%. Si l'intervalle d'une variable x est réduit de moins de τ_{propag} par rapport à sa taille initiale (avant la procédure de révision), alors les contraintes c' attenantes à x ne sont pas empilées dans la queue de propagation.

Algorithmes `2B` et `HC4`

L'algorithme `2B` est l'algorithme de propagation de contraintes (sur intervalles) le plus reconnu par toutes les sous-communautés. Il est implanté dans la plupart des outils de résolution de systèmes basés sur les intervalles, y compris la dernière stratégie d'optimisation globale robuste/fiable de Rueher et al. [245] et Baron [250].

Bien qu'existant sous des formes diverses depuis longtemps, l'algorithme `2B` a été décrit formellement pour la première fois par Olivier Lhomme (qui travaille aujourd'hui dans la société IBM/Ilog à Sophia-Antipolis) dans un article de congrès très cité [186]. La propagation de contraintes suit le schéma décrit ci-dessus. La procédure de révision applique ce que l'on appelle des *fonctions de projection* associées à chaque occurrence d'une variable dans la contrainte. Prenons par exemple la contrainte $c : f(X) = (x + y + z)^2 + 3(x + z) = 30$. La fonction de projection $f_c^{z_2}$ isole la deuxième occurrence de z dans c et permet de contracter $[z]$ en effectuant : $[z] \leftarrow [z] \cap f_c^{z_2}([x], [y], [z])$, avec :

$$f_c^{z_2}([X]) = [f_c^{z_2}]_N([X]) = \frac{1}{3}\left(30 - ([x] + [y] + [z])^2\right) - [x]$$

Les contractions obtenues par des fonctions de projection portant sur deux occurrences d'une même variable se recoupent par intersection des intervalles obtenus pour chaque occurrence (ex : $[z_1]$ et $[z_2]$).

L'outil ALIAS, développé par Jean-Pierre Merlet [201], optimise chaque fonction de projection avec Maple en transformant formellement son expression (par exemple une forme factorisée faisant disparaître des occurrences multiples produit souvent une meilleure évaluation). Si une option dans ALIAS est choisie pour utiliser les dérivées, les fonctions de projection sont alors

[4] Dans le cas général, ce point-fixe n'est pas unique et dépend de la précision demandée, comme le montrent des exemples du cours de Michel Rueher sur les intervalles. Seule une précision au flottant près garantit l'obtention d'un point-fixe unique.

en plus évaluées non pas naturellement, mais par monotonie (ex : remplacement de $[f_c^{z_2}]_N$ par $[f_c^{z_2}]_M$).

Si l'on n'effectue pas de manipulation formelle des fonctions de projection, on peut utiliser une structure d'arbre représentant l'expression de f pour appliquer "simultanément" toutes les fonctions de projection associées à f à moindre coût. Cette version de l'algorithme `2B-Revise` est l'algorithme `HC4-Revise`. L'algorithme de propagation correspondant `HC4` a été proposé en 1999 par l'équipe nantaise de PPCI et Jean-François Puget de Ilog [28]. La figure 2.1 donne une trace de `HC4-Revise` sur la même contrainte $c : f(X) = (x + y + z)^2 + 3(x + z) = 30$. La phase montante est tout simplement l'évaluation naturelle de f et produit à la racine de f l'intervalle $[-6, 121]$. Le calcul en chaque nœud utilise l'arithmétique d'intervalles pour la fonction élémentaire correspondante. Par exemple, au nœud + (n_1), on attache le résultat du calcul $[0, 1] + [-2, 6] = [-2, 7]$. Au nœud =, on intersecte l'évaluation $[f]_N([X])$ avec le membre droit $[30, 30]$ (si l'intersection est vide, on a montré l'absence de solution pour cette contrainte et donc pour le système, comme le fait le test d'existence décrit plus haut), avant d'entamer la phase descendante.

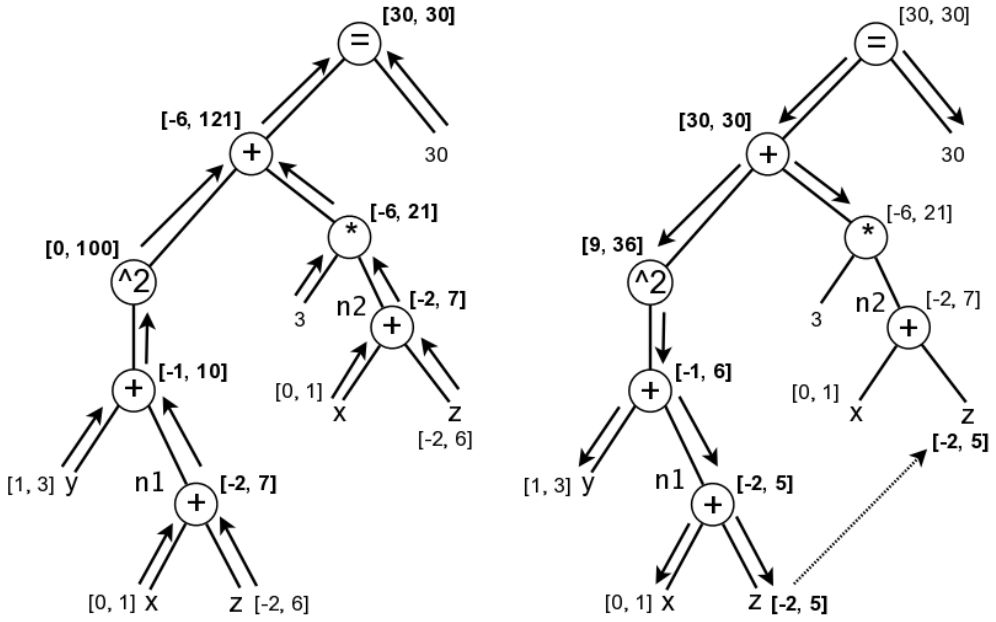


FIG. 2.1 – Phases montantes et descendantes de la procédure `HC4-Revise` appliquées à la contrainte $(x + y + z)^2 + 3(x + z) = 30$.

Cette deuxième phase applique les fonctions élémentaires "inverses" en chaque nœud. Considérons par exemple la somme la plus haute : $n_3 + n_4 = 30$. Le traitement du nœud n_3 donne : $[n_3] = [n_3] \cap [30, 30] - [n_4] = [0, 100] \cap [30, 30] - [-6, 21] = [9, 36]$; la différence est l'opération inverse de la somme et permet d'isoler n_3 dans l'expression. Le traitement du nœud $n_4 = 30 - n_3$ ne produit aucune réduction de $[n_4]$ si bien que le parcours sur le sous-arbre de n_4 est interrompu. Le processus sur le sous-arbre n_3 produit par propagation la réduction de $[z]$ à la valeur $[-2, 5]$.

Cette phase de projection implique de disposer d'un catalogue proposant une fonction inverse étendue aux intervalles pour chaque fonction élémentaire. Plusieurs chercheurs militent pour que ces opérateurs inverses entrent dans la première norme IEEE sur l'arithmétique d'intervalles à bornes flottantes en cours d'élaboration.

Soit e le nombre de nœuds dans cette arbre, c'est-à-dire le nombre d'opérateurs élémentaires binaires ou unaires dans la fonction. La procédure `HC4-Revise` s'exécute en temps $O(e)$.

Les solveurs nantais (dont `RealPaver` [112]), la bibliothèque `Ilog Solver`, `Icos` [183]) et `Ibex` [52, 50] utilisent une telle structure d'arbre, voire même un graphe orienté acyclique (en anglais : DAG) permettant de synthétiser des sous-expressions communes (cf. section 3.1). Notons que la structure d'arbre permet également de calculer en deux parcours (en $O(e)$) le gradient de f sur un boîte donnée, c'est-à-dire les dérivées partielles de f en chaque variable [28].

Consistance d'enveloppe (*hull-consistency*)

Considérons l'équation $c : f(X) = 0$. De même que l'on cherche à obtenir une évaluation optimale $[f]_O([X])$ dans la boîte $[X]$, le but d'une procédure de révision est de contracter $[X]$ optimalement en prenant en compte seulement (le sous-problème réduit à) c. Le problème de trouver l'enveloppe la plus petite contenant toutes les solutions de c dans $[X]$ est désigné par *consistance d'enveloppe* (en anglais : *hull-consistency*). (La consistance d'enveloppe se rapproche de la consistance de bornes (*bound-consistency*) utilisée pour les CSP sur domaines finis [294].)

Nous ne connaissons pas d'algorithme polynomial permettant d'obtenir la consistance d'enveloppe d'une fonction quelconque composée des fonctions élémentaires citées plus haut. Les mêmes causes produisant les mêmes effets, outre les problèmes d'arrondis, le manque de continuité de la fonction (ou de ses fonctions de projection) et les occurrences multiples de variables rendent le problème difficile. Si une fonction f est continue dans $[X]$, alors la phase ascendante produit une évaluation optimale (cf. section 2.2.2), mais pas nécessairement la phase descendante car les fonctions de projection ne sont, elles, pas nécessairement continues. L'incapacité de `HC4-Revise` à calculer la consistance d'enveloppe en présence d'occurrences muliples apparaît sur l'exemple $c : f(X) = (x + y + z)^2 + 3(x + z) = 30$, où l'on voit que la fonction de projection $f_c^{z_2}$ censée isoler z utilise aussi $[z]$ en argument (dans le cas général, en plusieurs occurrences décorrélées).

Un exemple minimal, $f(x, y) = (x\,y)^2 - 1 = 0$ dans la boîte $\{[x], [y]\} = \{[0, 1], [-1, 1]\}$, montre que l'absence d'occurrences multiples ne suffit pas à `HC4-Revise` pour calculer la consistance d'enveloppe. Le phase ascendante attache $[-1, 1]$ au nœud '$\times$' et $[0, 1]$ au nœud *sqr* (carré). Dans la phase descendante, l'égalité réduit l'intervalle du nœud *sqr* à $[1, 1]$. La perte d'optimalité se produit au traitement de la fonction inverse de *sqr* ($\pm\sqrt{}$) qui contracte l'intervalle associé au nœud $\times$. Cette fonction retourne $[-1, 1] = [-1, 1] \cap Hull([-1, -1] \cup [1, 1])$. Du coup, la projection sur x laisse x inchangé, au lieu de le réduire à $[1, 1]$ qui est l'intervalle optimal.

En fait, en l'absence d'occurrences multiples, mais sans hypothèse quelconque de continuité, les deux parcours d'arbre devraient *en théorie* effectuer des points de choix en chaque nœud pour obtenir la consistance d'enveloppe. L'analyse par cas menée par chaque fonction élémentaire (directe et inverse) fournit des sous-parties continues (des unions de plusieurs intervalles pour

une même variable) qui ne devraient pas être fusionnées par l'opérateur $Hull$ (sauf pour la boîte $[X]$ finale), mais au contraire ajouter une combinatoire dans les calculs. Sur l'exemple, le choix de $[-1,-1]$ pour le nœud $\times$ produit bien la réduction $B_1 = \{[x],[y]\} = \{[1,1],[-1,-1]\}$ aux feuilles de l'arbre ; le choix de $[1,1]$ produit $B_2 = \{[x],[y]\} = \{[1,1],[1,1]\}$, d'où finalement la boîte optimale $Hull(B_1,B_2) = \{[x],[y]\} = \{[1,1],[-1,1]\}$, toutes les bornes apparaissant dans l'une des deux solutions de la contrainte.[5]

Nous appelons `TAC-Revise` [53] cette variante combinatoire de `HC4-Revise`. Heureusement *en pratique*, ces points de choix sont souvent inutiles, et `HC4-Revise` calcule très tôt dans l'arbre de recherche la même boîte que `TAC-Revise`. Après quelques bissections seulement, et donc assez haut dans l'arbre de recherche, on observe souvent que les fonctions sont continues dans l'espace de recherche (boîte) courant [53].

En pratique ainsi, on considère que `HC4-Revise` approxime finement la consistance d'enveloppe en l'absence d'occurrences multiples des variables. En cas d'occurrences multiples au contraire, il faut souvent faire appel à d'autres procédures de révision pour améliorer la contraction obtenue.

Algorithme `Box`

L'algorithme `Box` [293, 28] est un algorithme de propagation dont la procédure de révision `BoxNarrow` est plus puissante [59] que `HC4-Revise` en présence d'occurrences multiples de variables. Si *une seule* variable x apparaît plusieurs fois dans la contrainte (et que f est continue dans la boîte $[X]$), alors `BoxNarrow` calcule pour $[x]$ l'intervalle le plus petit sans perte de solution (à une précision ϵ donnée).

`BoxNarrow` effectue un travail spécifique pour chaque variable $x \in X$ qui apparaît plusieurs fois dans f. Notons $c : f(x, y_1, ..., y_k) = 0$ la contrainte d'arité $|X| = k+1$. `BoxNarrow` transforme d'abord $f(X) = 0$ en $c' : f_{[Y]}(x) = 0$ en remplaçant dans f chaque occurrence des autres variables Y par leur intervalle courant dans $[X]$: $f_{[Y]}(x) = f(x,[y_1],...,[y_a])$. Un exemple de contrainte "épaisse" ainsi obtenue est représenté à la figure 2.2. `BoxNarrow` cherche à calculer l'enveloppe optimale des zéros de cette fonction épaisse en approximant finement le zéro le plus à gauche (l) et le zéro le plus à droite (r). Sur la figure, les segments en gras représentent ces zéros et `BoxNarrow` doit calculer deux intervalles de diamètre ϵ qui approximent finement l et r pour réduire $[x]$ à $[l,r]$ (à ϵ près). On dit alors que `BoxNarrow` calcule la Box-consistance.

Décrivons comment `BoxNarrow` approxime finement la borne gauche l de $[x]$, avec une précision donnée ϵ.

1. De gauche à droite, on découpe l'intervalle $[x]$ en bandes/tranches $[s_x]$ de diamètre ϵ.

[5] La preuve de ce résultat est une application directe d'un résultat connu sur les CSP en domaines finis [86], mais aussi dans d'autres domaines de l'informatique comme les bases de données. Sans détailler, la contrainte composée peut se voir comme une conjonction de contraintes élémentaires (les nœuds de l'arbre) formant un graphe de contraintes (élémentaires) sans cycle. En chaque nœud, si on se restreint à une partie continue pour chaque variable, la fonction élémentaire calcule l'arc-cohérence de la contrainte élémentaire, c'est-à-dire que chaque réel dans une partie continue a au moins un *support* dans chaque intervalle en argument. Un double-parcours d'arc-consistance dirigée (en anglais : DAC) calcule alors la consistance globale du problème sans cycle, soit l'arc-cohérence de la contrainte (composée) c dont l'enveloppe donne justement la consistance d'enveloppe...

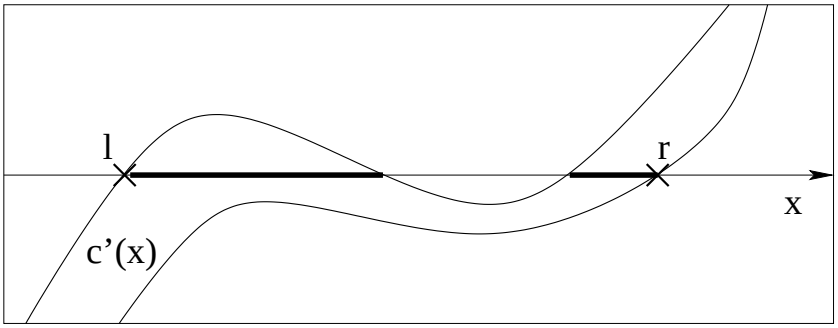


FIG. 2.2 – Contraction de $[x]$ à $[l, r]$ (à ϵ près) par `BoxNarrow`

2. Si $0 \notin [f]_N([s_x], [y_1], ..., [y_a])$, on a la garantie que la bande $[s_x]$ ne contient pas de zéro. On élimine alors $[s_x]$ du domaine de x et on passe à la bande suivante.

3. Sinon, si $0 \in [f]_N([s_x], [y_1], ..., [y_a])$, un zéro se trouve *peut-être* dans $[s_x]$ et on interrompt la boucle.

   On essaie de certifier la présence d'un zéro dans $[s_x]$ en appelant un Newton intervalles (univarié) sur $f(x, [y_1], ..., [y_a]) = 0$. Si Newton garantit une solution (épaisse) unique, il converge rapidement (de manière quadratique) vers un intervalle $[s_x]' \subset [s_x]$. On retourne $\underline{s'_x}$ qui est très proche de l. Sinon, on retourne $\underline{s_x}$, sans garantie d'être à moins de ϵ de l.

Si le test est positif (item 3 ci-dessus), on n'a pas la garantie d'obtenir un zéro dans la bande traitée à cause de la surestimation calculée par $[f]_N$ et liée aux occurrences multiples des y_i, mais aussi aux occurrences multiples de x si ϵ est "grand".

La description ci-dessus est naïve et `BoxNarrow`, pour la borne gauche (resp. droite), découpe en fait l'intervalle initial $[x]$ de manière dichotomique, en recherchant d'abord le zéro dans le sous-intervalle de gauche (resp. droite) [293, 28]. Le Newton intervalles devient alors très utile s'il parvient à certifier un zéro dans une bande $[s_x]$ assez large, car il permet alors d'économiser de nombreux appels sur des tranches plus petites.

L'algorithme `BoxNarrow` obtient quelques bons résultats en pratique quand la contrainte ne contient qu'une variable "critique" apparaissant plusieurs fois. Dans le cas général où *plusieurs* variables ont des occurrences multiples, le coût supplémentaire de `BoxNarrow` par rapport à `HC4-Revise` ne rapporte généralement pas un gain en contraction suffisant, à cause des occurrences multiples des y_i, et s'avère être moins performant en temps de calcul.

Une des principales contributions décrites dans ce mémoire est de justement proposer une nouvelle procédure `Mohc-Revise` qui s'intéresse au cas général. Celle-ci ne fait pas appel systématiquement à une procédure `BoxNarrow`. Elle fait appel à une variante de `BoxNarrow` plus efficace, et ne le fait que quand on détecte que la fonction est monotone par rapport à x dans la boîte courante (cf. section 3.3).

Autres approches

Un algorithme de propagation dédié aux contraintes quadratiques a été proposé par Domes et Neumaier [70]. Chaque équation est écrite sous la forme quadratique développée suivante :

$$\sum_k (a_k\, x_k^2 + b_k\, x_k) + \sum_{j,k,j>k} (b_{jk}\, x_j\, x_k) = c$$

où a_k, b_k et b_{jk} sont des coefficients réels.

La procédure de révision sépare la contrainte en deux parties et commence par borner ou approximer les termes bilinéaires. La phase ascendante d'évaluation calcule une enveloppe de chaque terme univarié $p_k(x_k) := a_k\, x_k^2 + b_k\, x_k$. Les intervalles $[p_k]$ obtenus sont utilisés pour vérifier que la contrainte est faisable. Si c'est le cas, la phase descendante (projection) réduit les $[x_k]$. L'intérêt de cette propagation est qu'elle obtient la consistance d'enveloppe (optimale) quand il n'y a pas de termes bilinéaires. Autrement dit, les termes $a_k\, x_k^2 + b_k\, x_k$ sont exploités optimalement. Pour autant, cette approche demande validation, à cause de l'approximation des termes bilinéaires (non triviale) et de la mise sous forme quadratique développée qui peut introduire de nombreuses variables auxiliaires.

2.4.4 Algorithmes de consistance forte, `3B`

L'algorithme `3B` (cf. algorithme 2) prend en paramètre un système de contraintes P formé d'un vecteur X de variables, d'un vecteur C de contraintes, et d'une boîte $[X]$ qui est contractée par l'algorithme. Il prend aussi 3 paramètres utilisateur qui influent sur son comportement : `SubContractor` qui est une procédure de contraction (par exemple : `HC4`, `Box`) et deux paramètres de précision ϵ_{outer}, ϵ_{inner}. On notera par la suite `3B(HC4)` l'algorithme `3B` qui utilise `HC4` comme sous-contracteur. Nous désignons par *sous-contracteur* une procédure de contraction utilisée par un autre contracteur.

Algorithm 2 `3B` (**in-out** $P(X, C, [X])$; **in** `SubContractor`, ϵ_{outer}, ϵ_{inner})

```
P_old ← P
repeat
  for all variable x ∈ X do
    VarShaving (x, P, SubContractor, ε_inner)
  end for
until StopCriterion(ε_outer, P, P_old)
```

La boucle `for all` effectue un travail sur toutes les variables itérativement. A l'issue de ce travail, `StopCriterion` teste si l'un des intervalles de $[X]$ est réduit de plus de ϵ_{outer}. Dans l'affirmative, la boucle `repeat` rappelle le travail sur toutes les variables et ainsi de suite jusqu'à atteinte d'un point quasi-fixe (à ϵ_{outer} près).

Le cœur de l'algorithme est la procédure `VarShaving` qui se charge de la contraction (rognage ou *shaving*) de l'intervalle d'une variable x à gauche et à droite, en faisant appel au sous-contracteur.

Pour améliorer la borne gauche de $[x]$, `VarShaving` découpe $[x]$ en sous-intervalles $[s_x]$ de taille ϵ_{inner}. De gauche à droite, pour chaque tranche $[s_x]$:

1. On crée $P_{[x]=[s_x]}$ un sous-système de P où l'intervalle $[x]$ est réduit à $[s_x]$.
2. On déclenche alors la procédure de contraction `SubContractor` sur le sous-problème $P_{[x]=[s_x]}$.
3. Si on obtient un échec (`SubContractor`$(P_{[x]=[s_x]}) = \emptyset$), alors cela signifie que l'on peut éliminer $[s_x]$ de $[x]$. Sinon, on interrompt la boucle.

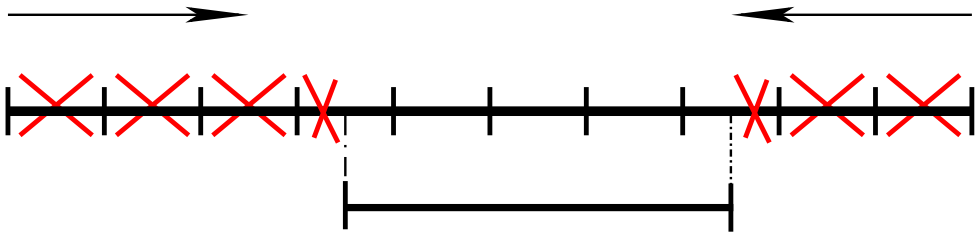


FIG. 2.3 – Illustration de `VarShaving` sur un intervalle $[x]$.

Nous illustrons `VarShaving` sur la figure 2.3. Les 3 premières tranches conduisent à un échec de `SubContractor` et donc à leur élimination de $[x]$. La quatrième réduit l'intervalle $[s_x]$ sans échec. $[x]$ est réduit d'autant et on stoppe l'opération de rognage. Notons que rien ne garantit que cette quatrième tranche contienne une solution. En revanche, le sous-contracteur, quoiqu'imparfait, a permis de détecter de manière certaine l'absence de solution sur les trois premières tranches. Le même processus est lancé de manière symétrique pour la borne droite.

Remarques

L'algorithme `3B` n'est pas un algorithme de propagation incrémental. Dès qu'un intervalle est réduit de plus de ϵ_{outer}, la boucle `repeat` doit rappeler le travail sur *toutes* les variables (boucle `for`) car les causes de cette réduction sont globales à tout le système (à cause du sous-contracteur). C'est pourquoi on l'appelle algorithme de consistance *forte*. `3B(propag)` obtient une boîte plus petite qu'un algorithme de propagation `propag`, même optimal, ne le ferait seul en raisonnant sur les contraintes individuelles.

L'algorithme kB se définit récursivement par un appel au sous-contracteur $(k-1)B$, la $3B$ étant définie comme `3B(2B)`. Par exemple, la $4B$ utilise la $3B$ comme sous-contracteur.

A condition de maîtriser le nombre de tranches traitées dans `VarShaving` et le paramètre ϵ_{outer} (cf. section 3.2), l'algorithme `3B` est polynomial en temps. Pourtant, de la même manière que l'opérateur de bissection (qui peut provoquer l'explosion combinatoire dans l'arbre de recherche), il effectue des points de choix en "essayant" des tranches. Ces points de choix sont cependant polynomiaux car un seul intervalle est le résultat du travail de `VarShaving`.

Le principe de rognage de `VarShaving` est similaire à celui de la procédure `BoxNarrow`. Dans les solveurs, `VarShaving` découpe d'ailleurs généralement les tranches de manière dichotomique, comme le fait `BoxNarrow`. La différence majeure est que, dans `BoxNarrow`, une tranche ne peut être réfutée qu'en raisonnant sur la contrainte (unique) considérée. Une tranche dans `3B` est éliminée par `VarShaving` en considérant le système entier. C'est ce point essentiel qui explique que des versions avancées de l'algorithme `3B` semble offrir des performances généralement meilleures que `Box`. Une des contributions présentées à la section 3.2 est d'offrir des versions améliorées de l'algorithme `3B` qui justifient de réhabiliter cet algorithme aussi simple que puissant et bien trop sous-estimé actuellement.

L'algorithme `3B` a été proposé pour la première fois par Olivier Lhomme en 1993 [186]. L'idée de rognage a été proposée aussi pour les problèmes discrets. Elle est à la base de l'algorithme de propagation SAC [64] pour les CSP. Elle est un des ingrédients essentiels de `SATZ` [209] qui reste un algorithme de référence pour résoudre les instances aléatoires (non structurées) du problème SAT (satisfiabilité de formules logiques booléennes). On la met en œuvre aussi depuis des années pour résoudre des problèmes d'ordonnancement. A titre récréatif, on peut résoudre pratiquement toutes les instances de Sudoku (en tout cas de taille 3, c'est-à-dire les grilles 9×9) sans développer le moindre arbre de recherche avec un algorithme `SAC(GAC)`[6]. Cela signifie que cet algorithme simple subsume la plupart des "règles" proposées pour résoudre une grille de Sudoku.

Autres représentations des contraintes

Il existe aussi des algorithmes qui mettent les contraintes sous une forme différente, quand celles-ci sont polynomiales. Mentionnons notamment les travaux réécrivant les polynômes dans la base de Berntein pour produire de nouveaux contracteurs puissants [90, 217, 215].

2.5 Problèmes plus difficiles

Le schéma de résolution ci-dessus se place dans l'hypothèse favorable où le système de contraintes possède un ensemble fini de solutions ponctuelles pouvant être approximées par un ensemble de boîtes atomiques. Cela se produit quand le système bien contraint est carré, c'est-à-dire possède n équations indépendantes portant sur n variables.

Ce problème doit être traité par des techniques combinatoires (cf. le paragraphe sur la complexité théorique à la section 3.5), c'est-à-dire exponentielles en temps dans le pire des cas. C'est le cas des techniques algébriques utilisant les bases de Gröbner [45], des méthodes de continuation [5] ou bien des méthodes à intervalles. Il existe néanmoins des problèmes encore plus difficiles :

– L'optimisation d'une fonction F multivariée non-linéaire (sans contrainte) peut s'aborder avec un schéma proche de l'algorithme `trouverSolutions` (cf. section 2.3) en générant un système

[6] Il s'agit d'un algorithme proche de `3B(HC4)`... Des contraintes de différence (`AllDiff` de Régin [241]) sont définies pour chaque ligne, chaque colonne et chaque bloc carré 3×3 ; elles sont *révisées* par un algorithme de flot et propagées par un algorithme d'arc-cohérence généralisé (`GAC`).

$n \times n$ où chaque équation est de la forme $\frac{\partial f_j}{\partial x_i}(x) = 0$ [293] (les dérivées partielles de f_j s'annulent toutes en chaque minimum et chaque maximum local).
– L'optimisation d'une fonction sous contraintes, dont le schéma de résolution a été esquissé à la section 2.3.
– Le traitement de systèmes d'équations différentielles (ordinaires) [218, 143] ou d'intégrales.
– Le traitement de systèmes où les variables sont quantifiées universellement ou existentiellement [51, 100, 102, 103].
– Le traitement de systèmes *hybrides* contenant à la fois des contraintes non-linéaires sur des variables à valeurs réelles et des contraintes sur des variables discrètes (entières par exemple).

Nous introduisons ci-dessous comment aborder les systèmes d'inégalités et les systèmes quantifiés.

2.5.1 Traitement des inégalités et boîtes intérieures

Des approches basées sur les intervalles ont été proposées pour traiter les systèmes qui comportent un ensemble infini de solutions, notamment un système d'*inégalités* avec un *continuum* de solutions. L'approche standard consiste à extraire de ce continuum une des solutions qui minimise un critère donné. Selon les applications, l'optimisation sous contraintes ne satisfait pas toujours l'utilisateur, notamment quand plusieurs critères doivent être optimisés en même temps (optimisation multi-critères).

Pour représenter un continuum de solutions de manière relativement compacte, une alternative est de le paver avec des boîtes extérieures et des *boîtes intérieures*. Les boîtes extérieures sont celles que nous connaissons et qui sont susceptibles de contenir des solutions : les contracteurs garantissent qu'il n'y a pas de solution en dehors de ces boîtes. Une boîte intérieure est une boîte dont *tous* les points sont solutions : $\forall f \in F, \forall X \in [X],\ f(X) = 0$. Autrement dit, une boîte intérieure est contenue dans un continuum de solutions. L'intérêt principal est de ne pas devoir découper ces boîtes intérieures jusqu'à la précision ϵ, ce qui améliore sensiblement le nombre total de boîtes dans le pavage et donc les performances.

Seules les méthodes à intervalles, dans les petites dimensions, peuvent aujourd'hui approximer finement un continuum de solutions. Plusieurs techniques ont vu le jour.

Un test intérieur simple raisonne sur chaque contrainte individuellement et peut se voir comme le dual d'un algorithme de propagation de contraintes pour les boîtes extérieures. Le raisonnement logique est un peu tordu (quelques lignes d'équivalence avec des double-négations), mais conduit au principe suivant. Soit un système comportant les contraintes (inégalités) $c_1 \wedge ... \wedge c_n$. Si une boîte $[B]$ est éliminée à la fois par $\neg c_1$, $\neg c_2$,... et $\neg c_n$ (en générant la négation de chaque inégalité), cela signifie que $[B]$ est une boîte intérieure. Cette idée a été utilisée dans diverses applications par des chercheurs comme Luc Jaulin [147] et Jean-Pierre Merlet [203]. Elle a été formalisée notamment par Rueher et al. [59] et Benhamou, Goualard [27]. Elle est intégrée dans certains outils de résolution, dont `Ibex` [52, 50].

2.5.2 AE-systèmes

Il existe aussi une généralisation des systèmes classiques qui comportent des continuums de solutions. Il s'agit des AE-systèmes (*All-Exist systems*) qui contiennent des fonctions paramétrées par des paramètres U quantifiés universellement et des paramètres V quantifiés existentiellement. Une fonction paramétrée est de la forme suivante : $\forall U \in [U],\ \exists V \in [V], \quad f(X,U,V) = 0$.

Sans détailler, deux ingrédients essentiels permettent d'aborder ces problèmes. Premièrement, en plus des variables (X), il faut aussi bissecter les paramètres U et V en déployant un *arbre de recherche et/ou*. Deuxièmement, il existe des tests intérieurs applicables aux AE-systèmes qui sont plus puissants que celui décrit ci-dessus, mais utilisables seulement quand le système respecte certaines propriétés, parfois vérifiables en pratique :

- Quand un système d'inégalités est seulement quantifié universellement ($\forall U \in [U], f(X,U) \leq 0$), Goldsztejn, Michel et Rueher proposent d'ajouter, au test intérieur simple décrit ci-dessus, une heuristique de bissection spécifique des paramètres U [106].
- Dans le cas (rare en pratique) où chaque paramètre existentiel $v \in V$ n'apparaît que dans une seule équation et peut être isolé ($f(X,v) \equiv g(X) = v$), alors la condition $G([X]) \subseteq [V]$ est suffisante pour montrer que $[X]$ est une boîte intérieure [110].
- Quand le nombre de paramètres existentiels est égal au nombre de variables, une permutation variable-paramètre permet à un algorithme de Newton intervalles multivarié (épais) d'identifier une boîte intérieure, dans le cas où il garantit une solution "unique" dans le système ayant subi la permutation [101].
- Un test simple utilise les *intervalles généralisés* [163, 100] dans le cas (rare) où chaque composante de V n'apparaît au plus qu'une fois dans l'ensemble F des fonctions[7].
- Quand le système est linéaire en les paramètres existentiels, un test intérieur résout (en V) ce système linéaire.

Un problème de positionnement de la plateforme mobile d'un robot parallèle étudié dans mon équipe (cf. [109], [205] et la future thèse de Julien Hubert) tombe précisément dans ce dernier cas. Le but est de déterminer l'espace des positions (6 dimensions/variables) de la plateforme défini par $\{X \in \mathbb{R}^n \ t.q.\ \forall F \in [F], \exists \tau \in [\tau] \quad A(X)\,\tau = F\}$. Physiquement, cela signifie que *quelles que soient* les forces dans $[F]$ que peut subir la plateforme, il *existe* une force dans $[\tau]$ que les jambes peuvent supporter sans casser. $A(X)$ est une matrice qui dépend des positions X seulement.

Un test de boîte intérieure spécifique utilise le fait que le problème est linéaire en τ. Pour une boîte $[X]$ donnée, on calcule une extension aux intervalles $[A]$ de la matrice $A(X)$ sur $[X]$. On résout ensuite le système linéaire $[A].Y = [F]$ et on obtient $[Y]$. Si $[Y] \subseteq [\tau]$, alors $[X]$ est une boîte intérieure.

[7] Pour les initiés : Si $0 \subseteq F(dual[X], dual[U], V)$, alors $[X]$ est une boîte intérieure [100].

2.5.3 Problème traité dans ce mémoire

Nous nous limitons dans ce mémoire au problème de résolution de systèmes de contraintes non-linéaires (non convexes) le moins difficile, à savoir celui consistant à trouver (ou approximer) l'ensemble des solutions réelles.

Si des stratégies de résolution ont été conçues pour les problèmes plus difficiles mentionnés ci-dessus, les performances actuelles ne permettent pas de traiter de très gros systèmes. Mon intérêt pour ce domaine est relativement récent, et j'ai préféré concentrer mes efforts sur le problème de satisfaction (trouver les solutions du système) pour trois raisons principales. D'abord, le chapitre 3 soulignera que les briques algorithmiques traitant ce problème sont déjà en soi très perfectibles. De plus, attaquer les problèmes plus difficiles peut sembler prématuré si l'on ne dispose pas de briques algorithmiques plus efficaces. A l'appui de cette remarque, on note pendant trente ans des progrès très lents, voire la stagnation de l'analyse par intervalles d'un point de vue applicatif, alors que les problèmes d'optimisation globale ou d'équations différentielles étaient abordés depuis le début. Un peu comme ce qui s'est produit pour les domaines de programmation par contraintes sur domaines finis (CSP) ou la satisfiabilité de formules logiques (SAT) : le problème d'optimisation correspondant, NP-difficile (Max-SAT ou Max-CSP), a connu un intérêt décalé de 10 à 20 ans par rapport au problème de satisfaction moins difficile (NP-complet).

Enfin, nos contributions proposées au chapitre 3 n'ont pas été expérimentées sur les problèmes ci-dessus, mais peuvent en théorie s'y appliquer. En effet, les algorithmes de contraction issues de la programmation par contraintes sont très généraux et ne connaissent pas de restriction particulière si l'on se limite aux fonctions élémentaires énoncées à la section 2.2.1.

2.6 Origines des méthodes à intervalles

Historiquement, la première motivation du calcul par intervalles est la fiabilité des calculs. Pour la petite histoire, un théorème d'Archimède [14] montre que le nombre Π appartient[8] à l'intervalle $[3+\frac{10}{71}, 3+\frac{1}{7}]$. Pour la démonstration de ce théorème, il utilise une approximation de $\sqrt{3}$ compris dans l'intervalle $[\frac{265}{153}, \frac{1351}{780}]$. Ce souci d'approximation conservative des calculs est ainsi présent chez de nombreux mathématiciens depuis longtemps, et plusieurs formalisations sont proposées au 20^e siècle. Citons la mathématicienne anglaise Rosalind Cecily Young pour ses travaux sur *l'algèbre des nombres multi-valués* dans les années 1930 [309]. Dans le cadre du calcul numérique au cours des années 1950, le chercheur polonais Mieczyslaw Warmus propose son *calcul des approximations* [301] et le chercheur japonais Teruo Sunaga propose son *algèbre d'intervalles* [276].

Ramon E. Moore écrit en 1966 le livre fondateur de *l'analyse par intervalles* [211] qui marque le début de l'histoire de cette communauté en unifiant les connaissances connues à cette période, incluant entre autres les travaux précités, ainsi que ses propres recherches menées depuis les années 1950. L'arithmétique par intervalles y est définie afin de rendre conservatifs les calculs

[8] Archimède ne parle d'ailleurs pas d'intervalle, mais de bornes pour la valeur de Π, défini comme la fraction de la circonférence d'un cercle par son diamètre : $3+\frac{10}{71} < \Pi$ et $\Pi < 3+\frac{1}{7}$.

sur les nombres réels pour les opérateurs mathématiques de base (somme, produit, division, racine carrée, puissances, etc) comme pour les fonctions quelconques combinant ces opérateurs (extension aux intervalles d'une fonction). Moore discute déjà de l'implantation pratique sur un ordinateur de l'arithmétique par intervalles, où les intervalles ont des bornes flottantes.

Moore pose également un schéma algorithmique pour la résolution d'un système d'équations non-linéaires. Ce schéma comprend l'opérateur de Newton-Raphson étendu aux intervalles (cf. `ContractNewton` à la section 2.3), dont les multiples variantes seront la principale préoccupation de cette sous-communauté de l'analyse numérique pendant des décennies. Ce schéma comprend aussi un opérateur algorithmique de branchement combinatoire : la bissection qui permet une exploration complète de l'espace de recherche et de limiter le problème de la surestimation lié aux occurrences multiples de variables.

Attirés par l'aspect combinatoire de cette méthode de résolution et pour satisfaire leur besoin de traiter des systèmes non-linéaires en programmation logique avec contraintes, une poignée de chercheurs informaticiens spécialistes en programmation (logique) par contraintes sont venus, au début des années 1990, ajouter à ce schéma des opérateurs de *cohérence partielle* qui permettent de réduire l'espace de recherche en temps polynomial. La communauté française était alors réduite à Alain Colmeraeur, Frédéric Benhamou, Michel Rueher et Olivier Lhomme.

Plus récemment, grâce aux gains de performance croissants de ces méthodes, des chercheurs de divers domaines de mathématiques appliquées ou de physique, notamment en traitement du signal, robotique et automatique robuste (c'est-à-dire, en France, des chercheurs de la 61^e section du CNU) commencent à utiliser ces techniques pour prendre en compte les incertitudes dans leurs problèmes.

2.7 Communautés de recherche travaillant sur les intervalles

Au total, plus de 300 chercheurs s'intéressent aujourd'hui aux intervalles, mais ils sont répartis en quatre ou cinq communautés scientifiques.

- **Arithmétique d'intervalles** : Chercheurs implantant des opérateurs de bas niveau sur les architectures d'ordinateurs, et des bibliothèques logicielles de bas niveau. Ils cherchent entre autres à limiter la surestimation entraînée par les arrondis sur les bornes flottantes des opérateurs élémentaires.
- **Analyse par intervalles** : Chercheurs en analyse numérique intéressés par le calcul sur intervalles (complet et fiable). Depuis les années 1960, ils produisent notamment des versions intervalles des algorithmes numériques classiques (Newton multi-dimensionnel).
- **Optimisation globale / programmation mathématique** : Chercheurs en recherche opérationnelle intéressés par l'optimisation globale robuste sous contraintes non-linéaires.
- **Programmation par contraintes sur intervalles (PPCI)** : Chercheurs informaticiens issus de la programmation logique avec contraintes et appartenant aujourd'hui à la communauté de programmation par contraintes. Depuis les années 1990, ils proposent des algorithmes de contraction/filtrage et développent des outils de résolution basés sur des algorithmes d'analyse par intervalles et de PPCI.

– **"Applicatifs"** : Chercheurs (en France, notamment de la 61^e section du CNU) appliquant, et parfois développant, des méthodes à intervalles dans divers domaines. Il s'agit d'une communauté émergente.

Je rebondis sur cette classification en soulignant que l'équipe COPRIN dans laquelle je travaille comporte des chercheurs des différentes sous-communautés. Les chercheurs de l'INRIA appartiennent à cette dernière communauté (notamment en robotique) avec une forte compétence en analyse par intervalles (analyse numérique). Bertrand Neveu, Michel Rueher et moi-même sommes proches de la communauté informatique de programmation par contraintes.

2.8 Points d'entrée

Le lecteur intéressé par une description plus complète et formelle des méthodes à intervalles pourra consulter les ouvrages suivants :

– [212] : Ramon E. Moore, R. Baker Kearfott, Michael J. Cloud, *"Introduction to Interval Analysis"*, SIAM, 2009
– [118] : Eldon Hansen, *"Global Optimization using Interval Analysis"*, 1992
– [219] : Arnold Neumaier, *"Interval Methods for Systems of Equations"*, Cambridge University Press, 1990
– [293] : Pascal Van Hentenryck, Laurent Michel, Yves Deville, *"Numerica : A Modeling Language for Global Optimization"*, MIT Press, 1997.
– [29] : Frédéric Benhamou, Laurent Granvilliers, *"Continuous and Interval Constraints"*, chapitre 16 du livre *"Handbook of Constraint Programming"*, Elsevier, 2006.
– [147] : Luc Jaulin, Michel Kieffer, Olivier Didrit, Eric Walter, *"Applied Interval Analysis"*, Springer, 2001.

2.9 Outils de résolution

Le paysage des outils de résolution exhaustifs basés sur les intervalles est aujourd'hui relativement riche. Les outils suivants sont issus de chercheurs francophones, pour la plupart spécialistes en PPCI.

– Numerica [293] est l'un des premiers outils autonomes proposé en 1997 par P. Van Hentenryck, L. Michel et Y. Deville. Il ne possédait pas d'opérateur `ContractRelax`, mais permettait d'optimiser une fonction objectif (avec ou sans contraintes). Il n'est plus maintenu et est resté disponible pendant plusieurs années sous la forme d'une bibliothèque en `C++` de Ilog Solver nommée `IlcNum`.
  Les mêmes auteurs envisagent à moyen terme d'intégrer une nouvelle bibliothèque intervalles dans leur ambitieux outil/langage `Comet` [292] développé maintenant par la société Dynadec (cf. `www.dynadec.com`) et qui offre pour l'instant des opérateurs de recherche locale et de programmation par contraintes, sur domaines finis et non sur les réels.
– `RealPaver` [112] est né à la fin des années 1990 dans l'équipe nantaise de F. Benhamou,

son auteur principal étant L. Granvilliers. Spécialisé dans la résolution de systèmes (sans optimisation), il est disponible depuis 2008 sous forme d'une bibliothèque en `C++`.
- ALIAS [201] est l'outil interne utilisé par l'équipe COPRIN pour les applications en robotique. Spécialisé dans la résolution de systèmes, il est doté d'opérateurs de PPCI et d'analyse par intervalles (pouvant utiliser la matrice hessienne). Il possède également un algorithme du simplexe (non fiable à ma connaissance). La version ALIAS-Maple est la première intégration aboutie d'un outil basé sur les intervalles et d'un outil de calcul formel.
- `Icos` [183] est piloté par Yahia Lebbah en collaboration notamment avec M. Rueher et C. Michel et plus récemment A. Goldsztejn. `Icos` est spécialisé en optimisation globale sous contraintes et son dernier schéma pour l'optimisation [245] est très prometteur.
- `Ibex` [50] est développé par G. Chabert depuis sa thèse dans notre équipe. C'est aujourd'hui une bibliothèque `C++` libre et ouverte. Cantonnée pour l'instant à la résolution de systèmes (sans optimisation), `Ibex` permet néanmoins de prendre en compte des paramètres quantifiés et de paver l'espace des solutions avec des boîtes intérieures et extérieures. Depuis 2008, une couche nommée `Quimper` [52] a été conçue en collaboration avec L. Jaulin. `Quimper` offre un langage de plus haut niveau aux ingénieurs non spécialistes.

L'équipe de Neumaier à Vienne est en train de concevoir l'outil `GloptLab` [69], développé en Matlab par Ferenc Domes. `GloptLab` est spécialisé dans la résolution de systèmes quadratiques.

Quoique non fiable, l'outil d'optimisation globale `Baron` [250, 280] de Sahinidis et Tawarmalani constitue un étalon de performances à atteindre pour les méthodes à intervalles. `Baron` peut utiliser ses approximations convexes et sa méthode polyhédrale pour traiter également les problèmes combinatoires (sur les entiers). Soulignons aussi la simplicité de l'opérateur OBR (*Optimality-Based Reduction*) spécifique à l'optimisation globale qui permet de réduire les intervalles de variables en utilisant les bornes inférieure et supérieure de l'objectif [248].

Chapitre 3

Contributions en programmation par contraintes sur intervalles

Ce chapitre introduit nos six premières contributions en PPCI donnant lieu concrètement à une dizaine d'algorithmes nouveaux :

- La section 3.1 présente l'algorithme de prétraitement `I-CSE` qui permet de détecter et d'exploiter les sous-expressions communes qui apparaissent dans les contraintes. Le nouveau système de contraintes généré permet aux algorithmes de propagation de contraintes comme `HC4` de mieux filtrer les domaines, avec un surcoût pratiquement toujours négligeable.
- La section 3.2 traite de l'amélioration et de la réhabilitation de l'algorithme `3B` (introduit à la section 2.4.4).
- La section 3.3 présente l'algorithme de propagation de contraintes `Mohc` qui exploite tout simplement la monotonie des fonctions. `Mohc` a l'ambition de devenir une alternative aux algorithmes de référence `HC4` et `Box` (cf. section 2.4.3).
  La section 3.3.6 introduit une nouvelle extension d'une fonction aux intervalles qui est basée sur un algorithme de *groupement d'occurrences*. Cet algorithme réécrit très efficacement (à la volée au cours de la propagation de contraintes) une fonction pour qu'elle devienne en quelque sorte plus monotone par rapport à chaque variable dans la boîte courante. Cet algorithme constitue aussi une procédure avancée du contracteur `Mohc`.
- La section 3.4.1 propose l'algorithme `PolyBox` qui est une amélioration de l'algorithme `Box` (cf. section 2.4.3) utilisant les *fonctions extrêmes* de la fonction épaisse manipulée par `BoxNarrow`.
  La section 3.4.2 résume des recherches permettant de mieux comprendre comment exploiter les "trous" dans les intervalles, c'est-à-dire comment gérer des unions d'intervalles pour mieux filtrer les domaines.
  La section 3.4.3 aborde la question des outils/solveurs et mentionne la bibliothèque libre en `C++ Ibex`, développée essentiellement par mon ancien doctorant Gilles Chabert, et qui nous a permis d'implanter la plupart des algorithmes ci-dessus.

Cette description des contributions sera suivie des perspectives de recherche dans ce domaine (cf. section 3.5) et de notre thèse prédisant un essor important des méthodes à intervalles et un essor des briques de PPCI dans ces méthodes (cf. section 3.6).

3.1 Exploitation des sous-expressions communes

(Travail en collaboration avec Ignacio Araya et Bertrand Neveu ; article repris au chapitre A.)

On appelle sous-expression commune (CS) une expression numérique qui apparaît plusieurs fois dans une ou plusieurs contraintes. Si par exemple une somme $x + z$ apparaît deux fois dans un système, l'idée consiste à remplacer les deux CS par une variable auxiliaire v et à ajouter au système une nouvelle contrainte $v = x + z$. Si les expressions sont représentées sous forme de nœuds et d'arcs, cela revient, de manière alternative, à fusionner les deux nœuds '+' en un unique nœud avec deux parents. Si chaque contrainte est représentée par un arbre, après le remplacement de certaines CS, le système global devient alors un graphe orienté sans circuit (DAG).

3.1.1 Filtrage additionnel

De nombreux spécialistes des méthodes à intervalles ont observé l'intérêt de remplacer les sous-expressions communes (CS) dans un système d'équations. Van Hentenryck, Michel et Deville l'ont fait manuellement dans leur expérimentations avec Numerica [293]. Merlet le fait avec ALIAS, soit manuellement, soit en laissant Maple réécrire le système sous forme de DAG. Kearfott le fait également dans son outil `GlobSol` [165] en stockant les contraintes dans un DAG comme le font les outils de calcul formel. Schichl et Neumaier [254], puis Vu, Schichl, Sam-Haroud [300] ont développé leur propre structure de DAG et ont adapté leurs algorithmes de propagation de contraintes (de type `HC4`) pour fonctionner dans cette structure. La communauté pensait que les gains obtenus provenaient d'une évaluation plus rapide des sous-expressions communes (évaluées une seule fois après détection), comme en optimisation de code où l'élimination de CS (CSE) est employée précisément dans ce but.

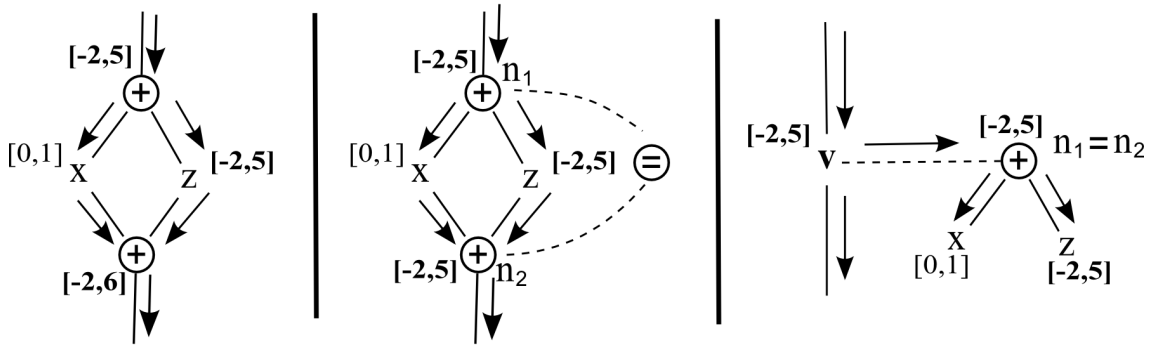


FIG. 3.1 – Phases ascendantes et descendantes de `HC4-Revise` avec et sans remplacement de CS.

Notre première contribution a été de montrer qu'il n'en est rien et que le remplacement de CS (par fusion de nœuds comme par introduction de variables auxiliaires) apporte en fait de la *contraction supplémentaire*, ce qui explique les gains de performance possibles de plusieurs ordres de grandeur. La figure 3.1 reprend notre micro-exemple pour l'expliquer. Soit n_1 et n_2

les deux sous-expressions égales à $x + z$. Si la phase descendante de `HC4-Revise` propage via n_1 un intervalle $[-2, 5]$ (provenant du reste du système), cette contraction obtenue se "perd" dans la phase ascendante de `HC4-Revise` pour n_2, et l'on obtient $[-2, 6]$. Faire le remplacement des CS avec une variable auxiliaire v et une contrainte $v = x + z$ (figure de droite) revient en fait à ajouter une contrainte redondante $n_1 = n_2$ (figure du milieu) qui permet de conserver l'intervalle $[-2, 5]$. Ainsi, de manière générale, le remplacement de CS revient à ajouter des contraintes redondantes au système si bien qu'un contracteur comme `HC4` produit un point fixe (une boîte) plus petit.

Cet exemple souligne la cause de la perte d'information. La phase descendante de `HC4-Revise` via n_1 utilise l'opération '$-$', censée être l'opposé du '$+$', pour obtenir de nouveaux intervalles $[x]$ et $[z]$. Or, ce calcul est imparfait du fait que $\mathbb{IR}$ muni de l'opération '$+$' n'est pas un groupe (cf. section 2.2.2). On observe évidemment le même phénomène pour la multiplication. On a également montré que le remplacement de CS peut apporter des gains de contraction pour les fonctions élémentaires unaires, sauf pour celles qui sont continues et monotones, comme x^a (a impair) ou log.

Notre deuxième contribution a donc été d'identifier les CS qu'il est *utile* de remplacer pour espérer une contraction supplémentaire : les opérateurs arithmétiques et les fonctions élémentaires qui sont non continues ou non monotones.

3.1.2 Algorithme `I-CSE`

La troisième contribution a été de proposer un nouvel algorithme nommé `I-CSE` (pour *interval common sub-expression elimination*). `I-CSE` a été implanté en Mathematica (par Ignacio Araya). Il génère un nouveau système de contraintes qui est traité ensuite par `Ibex`. `I-CSE` est capable de détecter plus de CS que les outils existants. Il peut trouver des CS correspondant à des sommes (n-aires) ou à des produits (n-aires), en prenant en compte la commutativité et l'associativité de ces opérateurs. Il permet enfin de détecter des CS en *conflit*, c'est-à-dire les sommes ou les produits qui se recouvrent partiellement. Le système d'équations suivant illustre l'ensemble de ces points :

$$\begin{aligned} x^2 + y + (y + x^2 + y^3 - 1)^3 + x^3 &= 2 \\ \frac{(x^2 + y^3)(x^2 + cos(y)) + 14}{x^2 + cos(y)} &= 8 \end{aligned}$$

On trouve dans ce système plusieurs CS, comme $x^2+cos(y)$, x^2+y^3 et $y+x^2$. Notons que les deux CS $x^2 + y^3$ et $y + x^2$ sont en conflit dans le terme au cube de la première équation. Ils partagent x^2, mais ont chacun un terme propre : y^3 d'une part et y de l'autre. Il semble donc impossible de remplacer à la fois ces deux CS par une même variable auxiliaire, le premier remplacement empêchant le deuxième. Ce sont précisément ces points de choix qui rendent le problème CSE, c'est-à-dire le remplacement de CS, NP-difficile en optimisation de code. C'est aussi pour cette raison que les outils de calcul formel appliquent des heuristiques d'une complexité linéaire en le nombre de nœuds et choisissent de manière heuristique l'une des deux CS quand ils stockent les expressions sous forme de DAG. La figure 3.2 montre le DAG construit par `I-CSE`.

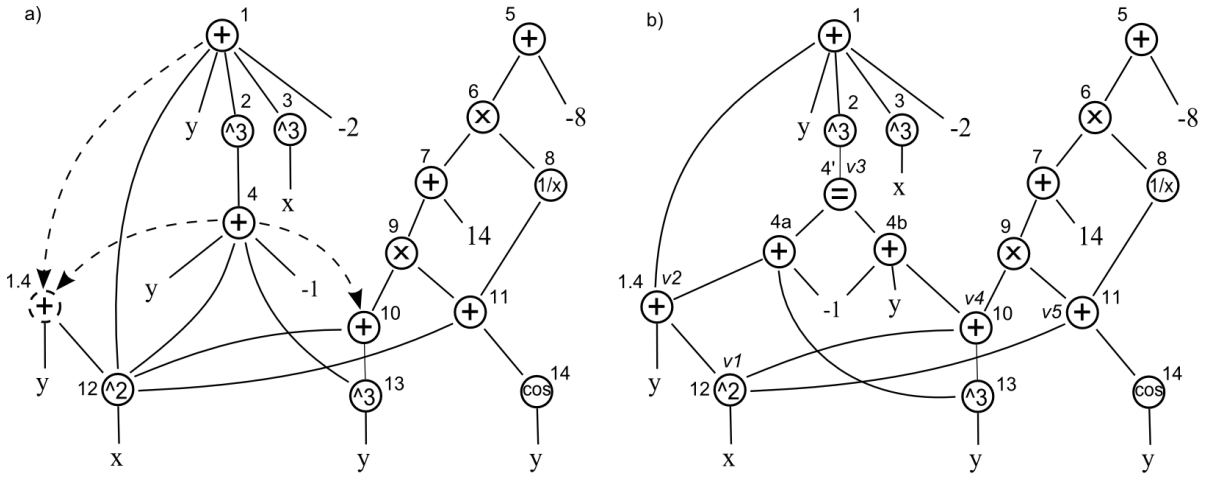


FIG. 3.2 – DAG calculé par `I-CSE`

La première étape de `I-CSE` génère le DAG formé de nœuds et d'arcs pleins à la figure 3.2-(a). Il s'agit de l'algorithme en temps linéaire classique utilisé en optimisation de code ou dans les outils de calcul formel. (Par souci de clarté, les variables à occurrences multiples ne sont pas fusionnées sur la figure.)

La deuxième étape de `I-CSE` est plus originale et ajoute les nœuds et arcs en pointillés à la figure 3.2-(a). Cette étape effectue une *intersection* deux à deux de tous les nœuds '+' d'une part, et l'intersection deux à deux de tous les nœuds '×' d'autre part. Sur l'exemple, cette étape d'intersection crée le nœud 1.4, intersection des nœuds 1 et 4, ainsi que les arcs vers 1.4. L'intersection des nœuds 4 et 9 existe déjà. Il s'agit du nœud 10, si bien que seul un arc vers 10 est ajouté. Le DAG final apparaît à la figure 3.2-(b). Pour les sous-expressions en conflit (nœuds 1.4 and 10), sont créés un nœud redondant $4b$ et un nœud '=' étiqueté $4'$.

3.1.3 Propriétés, complexité en temps, résultats expérimentaux

Si l'on considère toutes les paires de sous-expressions existant dans le système initial, `I-CSE` est capable d'en extraire *toutes* les CS maximales (au sens ensembliste).

La complexité en temps de `I-CSE` est en $O(n + a\,log(a)\,k^2 + k^3)$, où n est le nombre de variables, k est le nombre d'opérateurs a-aires (sommes ou produits) et a est l'arité maximale d'une somme ou d'un produit dans le système. En pratique, le temps de prétraitement par `I-CSE` est négligeable devant le temps de résolution total.

Les résultats expérimentaux de `I-CSE` sont très encourageants. Le surcoût lié à la gestion des contraintes et variables ajoutées est pratiquement toujours compensé par les gains en contraction. On enregistre des gains de performance de un ou plusieurs ordres de grandeur sur 10 des 40 instances testées [204], par rapport à une stratégie sans prétraitement basée sur `HC4` ou

3BCID (+ Newton). Soulignons quelques résultats inégalés de I-CSE suivi de 3BCID(HC4) sur les instances Trigexp2 [190] et Discrete Integral [213], ce dernier problème correspondant au calcul (discrétisé) d'une intégrale.

3.1.4 Génération d'un nouveau système

Les algorithmes de propagation de contraintes pourraient travailler sur le DAG généré par I-CSE, comme le font les outils à intervalles suisse [300], autrichien [69], hongrois [17] (sur un DAG contenant moins de CS). Cependant, il faut comprendre que les algorithmes de propagation de contraintes doivent être entièrement réécrits et adaptés pour fonctionner dans un DAG unique plutôt que contrainte par contrainte. En particulier, les détails de l'algorithme décrit dans [300] soulignent la difficulté à préserver un algorithme de propagation incrémental dans un DAG unique.

De manière alternative, I-CSE préfère ajouter une dernière étape qui parcourt ce DAG et génère un nouveau système avec de nouvelles contraintes et des variables auxiliaires correspondant aux CS utiles (ex : v_1, v_2, v_4, v_5) et aux nœuds '=' gérant les conflits (v_3) :

$$\begin{array}{rcl} v_2 + (v_3)^3 + x^3 - 2 & = & 0 \\ \dfrac{v_4 \times v_5 + 14}{v_5} - 8 & = & 0 \end{array} \qquad \begin{array}{rcl} v_1 & = & x^2 \\ v_2 & = & y + v_1 \\ v_3 & = & v_2 + y^3 - 1 \end{array} \qquad \begin{array}{rcl} v_3 & = & -1 + y + v_4 \\ v_4 & = & v_1 + y^3 \\ v_5 & = & v_1 + cos(y) \end{array}$$

Le système final montre bien comment sont gérés les conflits. Deux contraintes $v_3 = y+x^2+y^3-1$ sont ajoutées dans le nouveau système. L'une contient la CS $v_2 = y + x^2$, l'autre contient la CS en conflit $v_4 = y^3 + x^2$.

Dans la version actuelle, le système généré par I-CSE est géré de manière spécifique par Ibex, les variables auxiliaires n'étant ni bissectées, ni traitées par Var3BCID (cf. figure 3.3). En travaillant sur ACID (cf. section 3.2.2), nous visons idéalement à ne faire aucune distinction entre variables originales et variables auxiliaires, sous réserve d'utiliser la fonction *smear* [167] pour le choix des variables bissectées ou rognées. I-CSE constituerait alors un algorithme de prétraitement relativement indépendant des méthodes de résolution.

De plus, les algorithmes de propagation de type 2B travaillant sur un DAG unique et l'algorithme HC4 travaillant sur le système généré par I-CSE atteignent un même point fixe en termes de contraction (en considérant une précision de un u.l.p.). Ainsi, même si les temps requis pour atteindre leur point fixe différaient quelque peu, les deux approches obtiendraient sensiblement les mêmes performances quand elles sont intégrées à un arbre de recherche. Ainsi, dernière contribution, I-CSE permet d'intégrer facilement dans un outil à intervalles toutes les méthodes de contraction existantes, avec un surcoût ou un gain en temps de résolution négligeable par rapport à un algorithme de propagation de contraintes travaillant sur un DAG.

3.2 Réhabilitation de l'algorithme 3B

(Travail en collaboration avec Gilles Chabert, et plus récemment aussi avec Ignacio Araya et Bertrand Neveu ; article repris au chapitre B.)

Nous avons apporté plusieurs améliorations à l'algorithme 3B décrit à la section 2.4.4 (cf. algorithme 2 page 33). Celles décrites à la section 3.2.4 relèvent de détails d'implémentation qui ont une incidence sur les performances ; les autres sont plus importantes. Ces améliorations seront à la base des trois algorithmes CID, 3BCID et ACID dont les deux derniers (voire le dernier) remplaceront l'actuel 3B disponible dans la bibliothèque Ibex. Les principales améliorations sont décrites dans l'article [283] quoique plusieurs expérimentations n'y figurent pas. Le travail de contraction permet aussi à 3BCID, par effet de bord, de sélectionner une prochaine variable pertinente à bissecter dans l'arbre de recherche (cf. section 3.2.3), et le critère retenu dans l'implantation courante a légèrement évolué depuis sa publication. L'amélioration décrite à la section 3.2.2 relève en partie d'une perspective à moyen terme qui produira l'algorithme ACID (*Adaptive* CID).

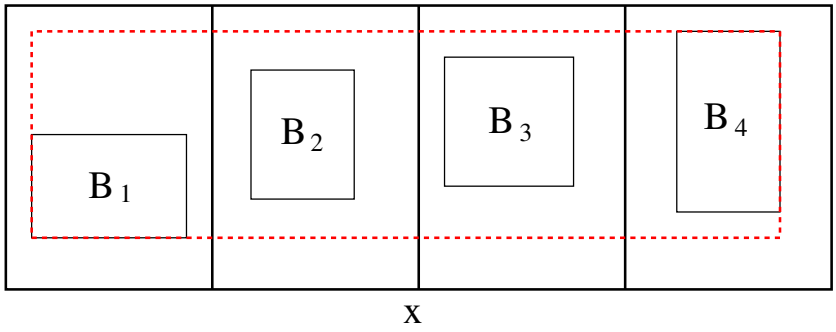
3.2.1 Disjonction constructive sur intervalles (CID)

La première idée qui nous a motivés à nous intéresser à l'algorithme 3B est basée sur la *disjonction constructive*, initialement introduite par Pascal Van Hentenryck et al. dans les années 1990 [294] pour traiter l'extension du modèle des CSP aux disjonctions de contraintes. Cette idée s'applique également à un domaine fini dans un CSP classique, où chaque domaine peut se voir comme une disjonction de contraintes unaires imposant une valeur parmi les différentes valeurs possibles : $x = v_1 \vee ... \vee x = v_i \vee ... \vee x = v_s$, où x désigne une variable discrète et $v_1, ..., v_s$ sont les différentes valeurs du domaine. Le principe de cette *disjonction constructive de domaine* a des similarités avec celui du rognage utilisé dans la 3B :

1. Chaque valeur v_i est assignée à x à tour de rôle, les autres étant temporairement supprimées. On applique un algorithme de filtrage au sous-problème $P_{x=v_i}$ correspondant.
2. Une fois cette boucle terminée, pour ne pas provoquer d'explosion combinatoire, l'espace de recherche est remplacé par l'*union* des différents espaces de recherche obtenus pour chaque sous-problème. Autrement dit, une valeur v_i est définitivement supprimée du domaine de x si elle est enlevée par sous-filtrage dans chacun des sous-problèmes.

Cette idée peut s'appliquer presque directement à un domaine continu en divisant un intervalle en s tranches, comme l'illustre la figure 3.2.1. L'exemple en deux dimensions montre un découpage en $s = 4$ tranches sur $[x]$. A chaque tranche i, une contraction sur le sous-problème, par exemple avec HC4, conduit à la boîte contractée $[B_i]$. La contraction finale obtenue par *construction disjonctive sur intervalles* (en anglais : *CID*) est *l'enveloppe* de $\{[B_1], ..., [B_4]\}$ (le rectangle en pointillés).

Cette procédure appliquée à un intervalle est appelée VarCID et peut remplacer VarShaving dans l'algorithme 2, page 33. VarCID est paramétré par un nombre de tranches s_{cid} traitées pour chaque variable. Elle a le même coût dans le pire cas que VarShaving (le coût de l'enveloppe



étant négligeable devant les appels au sous-contracteur), mais peut apporter une contraction potentiellement dans *toutes les dimensions*. Notons que le nombre d'itérations dans `VarCID` peut être strictement inférieur à s_{cid} si l'enveloppe courante des boîtes $[B_i]$ (calculée donc au fur et à mesure) est égale à la boîte initiale sur toutes les autres variables que x.

Cette dernière remarque m'a conduit à la variante `3BCID` qui est le contracteur généraliste utilisé par défaut par Ignacio Araya, Bertrand Neveu et moi depuis plusieurs années pour son efficacité. L'algorithme `3BCID` utilise une procédure `Var3BCID` qui est une hybridation de `VarShaving` et de `VarCID`. Elle est paramétrée par une précision relative ϵ_{inner} (pour la partie rognage) et par un nombre de tranches s_{cid} (pour la partie CID). Le principe est décrit ci-dessous et illustré par la figure 3.3 :

1. L'intervalle $[x]$ est d'abord rogné à gauche et à droite par un processus de rognage paramétré par ϵ_{inner}. La seule différence est que nous conservons les deux premières boîtes $[B_g]$ (à gauche) et $[B_d]$ (à droite) qui ne sont pas exclues par le sous-contracteur.
2. L'intervalle restant $[x]'$ est alors découpé en au plus s_{cid} tranches de diamètre égal, qui sont traitées par le sous-contracteur (voir plus haut). La procédure `VarCID` renvoie l'enveloppe $[B_{hull}]$ des s_{cid} boîtes contractées.
3. On retourne finalement l'enveloppe de $[B_g]$, $[B_{hull}]$ et $[B_d]$.

`3BCID` est le fruit de l'expérimentation de quelques dizaines de variantes de `CID`, dont deux seulement s'avèrent être aussi efficaces que `3BCID`. `3BCID` a été retenu pour sa simplicité.

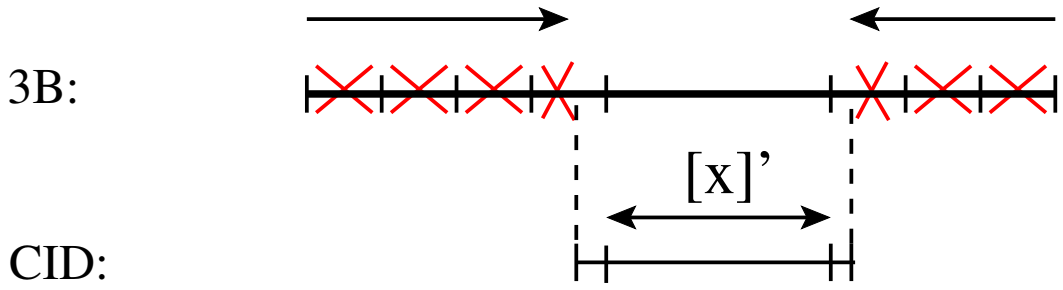


FIG. 3.3 – Travail de `3BCID` sur une variable (`Var3BCID`). ϵ_{inner} vaut 10% et s_{cid} vaut 1.

Le paramétrage par défaut de `3BCID` souligne une petite déception : ϵ_{inner} est fixé à 10%, mais s_{cid} est fixé à 1, ce qui conduit à un seul appel supplémentaire lié à la partie CID et à l'enveloppe de 3 boîtes seulement. Choisir $s_{cid} = 1$ a certes un impact significatif sur les performances par rapport à l'algorithme `3B` ($s_{cid} = 0$), mais augmenter encore s_{cid} est très souvent contre-productif, sauf pour quelques instances avec un grand nombre de variables comme `Mechanism` [222]. Cette observation empirique souligne que `CID` ne contracte pas beaucoup plus que `3B`, mais atteint un point quasi-fixe plus vite, à cause de la contraction en plusieurs dimensions de `VarCID`. Autrement dit, s'il est certain que `CID` contracte mieux que `3B` en théorie, le gain en pratique semble être souvent marginal.

3.2.2 Quelles variables rogner

L'obtention de la propriété théorique (3B-consistance) justifie l'utilisation de la boucle extérieure `repeat` (cf. algorithme 2, page 33) et du paramètre ϵ_{outer} contrôlant l'arrêt de cette boucle pour atteindre un point quasi-fixe. Après maintes expérimentations cherchant à trouver un critère adaptatif de relance de l'itération suivante, je me suis rendu compte que les performances optimales étaient souvent atteintes en une seule itération, voire moins ! C'est-à-dire qu'il ne faut pas raisonner en nombre d'itérations sur toutes les variables, mais en nombre n' de variables à rogner au total, ce nombre étant souvent inférieur ou égal au nombre n de variables dans le système.

Une première amélioration revient donc à supprimer cette boucle extérieure, ainsi que le paramètre correspondant ϵ_{outer}, et à lancer la boucle intérieure sur n' variables (à tour de rôle si $n' > n$), n' étant un paramètre utilisateur fixé par défaut à n.

Une amélioration plus ambitieuse consiste à régler de manière adaptative le paramètre n' c'est-à-dire automatiquement et dynamiquement pendant la recherche combinatoire. Si les critères présentés à la section suivante ne sont pas pertinents pour déterminer n', les heuristiques de choix de bissection "intelligentes" semblent prometteuses, en particulier la très reconnue heuristique utilisant la fonction *smear* [167] basée sur le diamètre des intervalles courants et la valeur des dérivées partielles (reflétant l'impact d'une variable dans une fonction). Cela s'explique intuitivement par le fait que le rognage peut se voir comme une bissection polynomiale et repose aussi sur la diminution artificielle de la taille des intervalles pour limiter le problème des occurrences multiples et pour accroître le nombre de fonctions continues dans la boîte courante.

Pour résumer l'algorithme `ACID` (*Adaptive* `CID`), il s'agit de l'algorithme `3BCID` où le nombre n' d'appels à `Var3BCID` n'est pas connu à l'avance : après chaque appel à `Var3BCID`, on estime si on a atteint un point quasi-fixe en termes de gain en contraction (périmètre des boîtes) sur les derniers appels. Dans le cas contraire, la fonction *smear* permet de sélectionner la prochaine variable à rogner. Ainsi, en plus des paramètres de son sous-contracteur (notamment `HC4` ou `Mohc`), `ACID` ne gère que deux paramètres utilisateur : le ratio ϵ_{inner} et le nombre de tranches s_{cid}. De plus, la valeur par défaut $\epsilon_{inner} = 10\%$ est robuste (les espoirs de gain d'un réglage fin sont faibles) et la valeur $s_{cid} = 1$ est très robuste (espoirs très faibles).

3.2.3 Heuristique de choix de variable à bissecter basée sur CID

En plus de sa vitesse élevée de convergence vers une boîte quasi-fixe, la disjonction constructive sur intervalles permet de fournir une heuristique efficace de choix de prochaine variable à bissecter.

Le principe consiste à calculer un ratio `ratioCID[i]` pour chaque variable rognée et à sélectionner la variable x_i qui possède le ratio le plus petit. Le critère simple retenu actuellement pour `3BCID` est quelque peu différent de celui décrit dans [283]. Il semble pertinent même quand s_{cid} vaut 1. Pour la variable x_i, soit $[B_g^i]$, $[B_d^i]$ les boîtes renvoyées par le processus de rognage et $[B_1^i], ..., [B_{s_{cid}}^i]$ les boîtes contractées par la construction disjonctive sur intervalles. On calcule :

$$\texttt{ratioBis}[i] = \frac{\texttt{Perim}([B_g^i]) + \texttt{Perim}([B_d^i]) + \sum_{j=1}^{s_{cid}} \texttt{Perim}([B_j^i])}{(s_{cid}+2) \times \texttt{Perim}(\texttt{Hull}([B_g^i], [B_d^i], [B_1^i], ..., [B_{s_{cid}}^i]))}$$

Ce ratio quantifie en quelque sorte la perte en "taille" de boîte causée par l'enveloppe effectuée par `Var3BCID` pour éviter l'explosion combinatoire. Si le ratio est très petit, cela signifie qu'une grande partie du travail de contraction obtenu dans chaque boîte $[B_g^i]$, $[B_d^i]$, $[B_1^i], ..., [B_{s_{cid}}^i]$ est perdue par l'opérateur d'enveloppe et qu'il vaudrait mieux à la place effectuer un "vrai" point de choix sur ces boîtes. Le fait qu'une bissection s'effectue en coupant l'intervalle en deux sous-intervalles, et non pas $s_{cid}+2$, ne semble rien enlever en pratique à la pertinence du critère donné ci-dessus. Les expérimentations montrent que cette heuristique de choix de variable (à bissecter) est un peu meilleure que les heuristiques à *tour de rôle* et *plus large domaine d'abord*. L'heuristique semble de plus être relativement robuste, c'est-à-dire qu'elle provoque moins souvent d'explosion combinatoire que ses concurrentes. Elle est en revanche probablement un peu moins efficace que la fonction *smear* quoique nous n'ayons pas effectué de comparaisons systématiques entre les deux heuristiques.

3.2.4 Gestion des tranches dans la procédure VarShaving

L'algorithme `3B` décrit dans la littérature [186] utilise un paramètre ϵ_{inner} qui est un nombre fixe de flottants. Le paramètre implanté dans les outils existants est un diamètre d'intervalle absolu (taille fixe). Nous proposons un paramètre ϵ_{inner} qui est un pourcentage de la taille des intervalles, autrement dit, dont l'inverse est un nombre fixé de tranches pour chaque intervalle traité par `VarShaving`. Ce détail d'implémentation permet de bien mieux contrôler l'effort de rognage indépendamment du diamètre de l'intervalle traité, et on a observé que le rognage d'un petit intervalle peut parfois entraîner la contraction de domaines plus larges. Si l'on veut continuer à spécifier une référence absolue, mieux vaut ajouter un autre paramètre permettant de ne pas déclencher le processus de rognage quand l'intervalle excède le diamètre spécifié (par défaut sous la précision des solutions).

Un autre aspect est la manière de découper les tranches dans la procédure de rognage. La littérature utilise un processus dichotomique, généralement par tranches de taille décroissante, et parfois par tranches de taille croissante, c'est-à-dire en partant de la borne avec une petite tranche de diamètre $\frac{\texttt{Diam}([x])}{\epsilon_{inner}}$, comme dans l'outil ALIAS [201]. Nous proposons en fait une technique de

découpage hybride dépendant de la valeur du paramètre ϵ_{inner}. Un simple découpage linéaire en tranches de taille $\frac{\texttt{Diam}([x])}{\epsilon_{inner}}$ est effectué quand la précision demandée (ϵ_{inner}) n'est pas trop grande, le processus dichotomique étant réservé aux précisions fines demandant la création de nombreuses tranches.

3.2.5 `3B` : l'algorithme sous-estimé

Autant `HC4` (ou l'algorithme `2B`) a réussi à faire consensus dans les communautés d'optimisation globale et d'analyse par intervalles, autant l'algorithme `3B` reste méconnu. Pire encore, `3B` ne fait même pas consensus au sein de la mini-communauté de programmation par contraintes sur intervalles. En fait, seuls deux sous-groupes de l'équipe COPRIN de manière relativement indépendante rapportent des résultats expérimentaux montrant la supériorité de l'algorithme `3B` : les roboticiens (Jean-Pierre Merlet, David Daney) avec ALIAS et les informaticiens (Ignacio Araya, Gilles Chabert, Bertrand Neveu et moi) avec `Ibex`. Le sentiment des informaticiens de COPRIN repose sur de nombreuses expérimentations sur la plupart des instances zéro-dimensionnelles de la page Web de COPRIN [204]. Pour résumer, `Box`, en tout cas l'algorithme `BC4` implanté dans `Ibex` ou l'algorithme `Polybox` présenté plus loin, ne semble pas concurrentiel par rapport à `3BCID(HC4)`. Pour confirmer nos premières expérimentations, il faudrait comparer `3BCID(HC4)` à des versions avancées de `Box`, développées notamment à Nantes [104]. Nous sommes aussi relativement confiants pour affirmer qu'un schéma de résolution basé sur `HC4` (+ Newton) n'est généralement pas concurrentiel par rapport au même schéma utilisant `3BCID(HC4)`, comme le suggère l'annexe C.

Même si nos résultats étaient trop optimistes ou biaisés par un jeu de tests insuffisant, le minimum serait de doter de `3BCID` tous les outils de résolution disposant de `HC4` (ou d'un autre algorithme de `2B`) ou de `Mohc` décrit ci-après.

3.3 `Mohc` : un algorithme de propagation de contraintes exploitant la monotonie

(Travail en collaboration avec Ignacio Araya et Bertrand Neveu ; article repris au chapitre D.)

Quoique récente, il s'agit probablement de la méthode de contraction la plus intéressante de ce chapitre. Elle utilise une version monotone des procédures existantes `HC4-revise` et `BoxNarrow` pour mieux contracter une boîte dès que la monotonie d'une fonction par rapport à l'une de ses variables est détectée (ce qui se produit de plus en plus au fur et à mesure que l'on descend dans l'arbre de recherche, traitant des boîtes plus petites).

`Mohc` a l'ambition de remplacer les algorithmes de propagation de contraintes `HC4` et `Box`.

3.3.1 Description de l'algorithme Mohc

L'algorithme Mohc (de l'anglais : *MOnotonic Hull-Consistency*) suit une boucle de propagation classique de type AC3. Son originalité tient à la procédure Mohc-Revise, décrite à l'algorithme 3, qui traite une contrainte individuellement.

Algorithm 3 Mohc-Revise (**in-out** $[B]$; **in** f, Y, W, ρ_{mohc}, τ_{mohc}, ϵ)

```
HC4-Revise(f(Y,W) = 0, Y, W, [B])
if W ≠ ∅ and ρ_mohc[f] < τ_mohc then
  [G] ← GradientCalculation(f, W, [B])
  (f^og, W) ← OccurrenceGrouping(f, W, [B], [G])
  (f_max, f_min, X, W) ← ExtractMonotonicVars(f^og, W, [B], [G])
  MinMaxRevise([B], f_max, f_min, Y, W)
  MonotonicBoxNarrow([B], f_max, f_min, X, [G], ε)
end if
```

La procédure cherche à contracter la boîte courante $[B]$ en travaillant sur une équation[1] $f(Y,W) = 0$, dans laquelle les variables de Y apparaissent *une seule* fois dans l'expression f et les variables de W apparaissent *plusieurs* fois dans f.

Mohc-Revise commence par appeler HC4-Revise. Une exception terminant la procédure est levée si une boîte vide est obtenue, prouvant l'absence de solution. Si f contient des occurrences multiples ($W \neq \emptyset$) et si une autre condition liée à un paramètre utilisateur est satisfaite (cf. section 3.3.4), alors plusieurs procédures sont appelées en séquence pour exploiter la monotonie de f par rapport à ses variables dans la boîte courante.

La fonction GradientCalculation calcule le gradient de f et le stocke dans un tableau $[G]$ indicé sur chaque variable : $[g_i] = [\frac{\partial f}{\partial x_i}]([B])$ est la i^e composante du tableau $[G]$. La fonction OccurrenceGrouping n'est pas nécessaire à l'algorithme Mohc mais en constitue une amélioration. Elle réécrit l'expression f sous une nouvelle forme f^{og} qui calcule, grâce à l'extension de monotonie, une image $[f^{og}]_M([B])$ plus petite ou égale à $[f]_M([B]) \subset [f]_N([B])$. Cette procédure est introduite à la section 3.3.6.

En utilisant le vecteur $[G]$, pour chaque variable $w \in W$ (avec plusieurs occurrences), si $0 \notin [\frac{\partial f}{\partial w}]([B])$, alors la fonction ExtractMonotonicVars bascule w de l'ensemble W vers un nouvel ensemble X de variables monotones. A l'issue de cet appel, W contient seulement les variables qui ne sont pas détectées comme étant monotones. (Par la suite, $[g_i] = [\frac{\partial f^{og}}{\partial x_i}]([B])$ est la i^e composante de $[G]$. $[g_i]$ dénote la dérivée partielle de f^{og} par rapport à la i^e variable x_i de X.) ExtractMonotonicVars renvoie les deux fonctions f_{min} and f_{max} qui vont être utilisées par la suite par les deux procédures principales exploitant, pour chaque variable $x_i \in X$, la monotonie de f^{og} par rapport à x_i. f_{max} est l'expression f où chaque variable x_i est remplacée par $\overline{x_i}$ (resp. $\underline{x_i}$) si f est croissante (resp. décroissante) par rapport à x_i dans la boîte. Pour f_{min}, chaque intervalle $[x_i]$ est remplacé par $\underline{x_i}$ (resp. $\overline{x_i}$) si f est croissante (resp. décroissante).

Les deux routines suivantes sont au cœur de Mohc-Revise. Elles travaillent principalement avec

[1] La procédure peut s'étendre facilement pour traiter une inégalité.

les deux fonctions f_{min} and f_{max}. La procédure `MinMaxRevise` contracte les intervalles des variables dans Y (à occurrence simple) et dans W (à occurrences multiples). La procédure `MonotonicBoxNarrow` contracte les intervalles des variables monotones qui sont dans X. Les sections suivantes illustrent les trois principales procédures de `Mohc` : `OccurrenceGrouping`, `MinMaxRevise`, `MonotonicBoxNarrow`, et soulignent quelques propriétés de l'algorithme.

3.3.2 La procédure `MinMaxRevise`

L'idée clef repose sur le fait que vérifier $0 \in [f]_M([B])$ revient à vérifier :

$$[f_{min}]([B]) \leq [0,0] \leq [f_{max}]([B])$$

Chacune des deux inégalités est alors utilisée par `HC4-Revise` pour contracter les intervalles des variables de Y et W. La figure 3.4 illustre comment la première partie de `Mohc-Revise` contracte la boîte $\{[x] = [4,10],\ [y] = [-80,30]\}$ pour la contrainte : $x^2 - 3\,x + y = 0$.

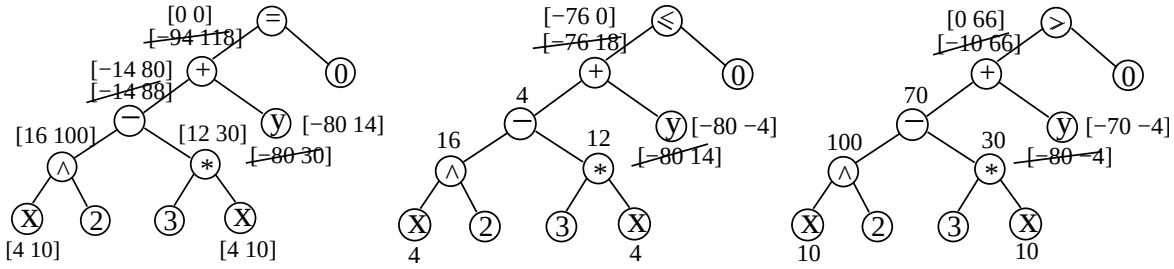


FIG. 3.4 – `HC4-Revise` (**gauche**), `MinRevise` (**centre**) et `MaxRevise` (**droite**) appliqués à $x^2 - 3\,x + y = 0$.

La figure 3.4-gauche illustre le travail de `HC4-Revise`. La deuxième phase, à cause du nœud d'égalité, intersecte l'intervalle $[-94, 118]$ avec 0 avant d'appliquer les fonctions "inverses" de projection de haut en bas. Par exemple, puisque $nplus = nminus + y$, la fonction inverse de cette somme produit la différence $[y] \leftarrow [y] \cap [nplus] - [nminus] = [0,0] - [-14,80] = [-80,14]$. L'opération symétrique sur $nminus$ produit $[-14,80]$, mais cette contraction ne se propage pas plus bas dans l'arbre et laisse $[x]$ inchangé. Après l'appel à `HC4-Revise`, `OccurrenceGrouping` détecte que la fonction est (entièrement) monotone par rapport à x (cas classique où la dérivée est positive), si bien que `ExtractMonotonicVars` bascule x dans l'ensemble X des variables monotones.

La figure 3.4-centre illustre la première étape de `MinMaxRevise`. L'arbre représente l'inégalité $f_{min}(4,y) \leq 0$. Appeler `HC4-Revise` sur cette expression produit une nouvelle contraction de $[y]$ car x est remplacée par $\underline{x} = 4$. La phase descendante intersecte $[-76,18]$ avec $[-\infty,0]$ (inégalité), et la première fonction de projection donne $[y] \leftarrow [y] \cap [nplus] - [nminus] = [-76,0] - [4,4] = [-80,-4]$. En suivant le même principe, `MaxRevise` applique `HC4-Revise` à $f_{max}(10,y) \geq 0$ et contracte $[y]$ à $[-70,-4]$ (cf. figure 3.4-droite).

3.3.3 La procédure MonotonicBoxNarrow

Cette procédure cherche à contracter l'intervalle de chaque variable monotone x de X. Par souci de concision, supposons par la suite que x est croissante. Une sous-procédure `LeftNarrowFmax` contracte la borne gauche de $[x]$ avec la fonction f^x_{max}. Une sous-procédure `RightNarrowFmin` contracte la borne droite de $[x]$ avec la fonction f^x_{min}.[2] Les fonctions intervalles univariées f^x_{max} et f^x_{min} dépendent de x. Elles s'apparentent à la fonction utilisée par la procédure `BoxNarrow` de l'algorithme `Box`. f^x_{max} et f^x_{min} sont issues du remplacement dans f de toutes les variables sauf x par des valeurs intervalles ou ponctuelles : les variables de X, sauf x, sont remplacées par une borne ; les variables de Y et W sont remplacées par leur intervalle courant dans $[B]$.

Comme la procédure `BoxNarrow` classique, la procédure `LeftNarrowFmax` découpe l'intervalle $[x]$ en tranches de taille décroissante jusqu'à une taille minimale de $\epsilon \times \texttt{Diam}([x])$, où ϵ est une précision relative. La grosse différence avec son pendant classique est que la propriété de monotonie permet à cette procédure de suivre un processus vraiment dichotomique, produisant au maximum un nombre logarithmique de tranches. Illustrons le fonctionnement de `LeftNarrowFmax` appliquée à la fonction f^x_{max} de la figure 3.5.

Le but consiste à contracter les bornes de $[x]$ pour produire une approximation fine du point L, correspondant à la nouvelle borne gauche de $[x]$. Un premier test d'existence vérifie que $\overline{f^x_{max}(\underline{x})} < 0$, c'est-à-dire que le point A sur la figure 3.5-gauche est au dessous de zéro. (Sinon, cela signifie qu'un zéro apparaît dans $\underline{x}$ si bien que $[x]$ ne peut pas être contracté à gauche, entraînant une terminaison rapide de la procédure.)

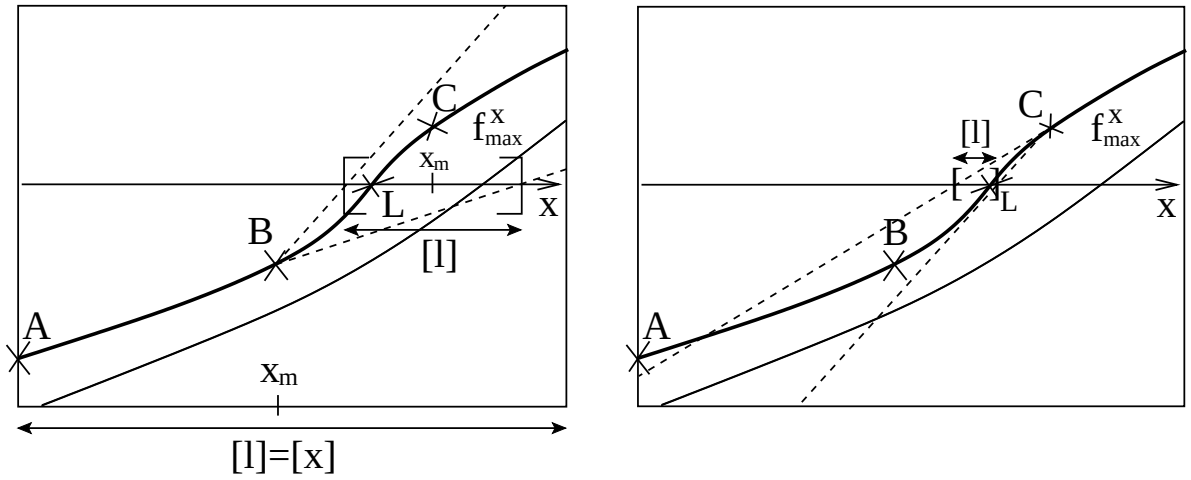


FIG. 3.5 – Deux premières itérations de Newton pour contracter la borne gauche de $[x]$.

Un processus dichotomique est alors déclenché. En initialisant l'intervalle $[l]$ avec $[x]$, des itérations de Newton intervalles univarié sont lancées itérativement à partir du point x_m milieu de $[l]$, c'est-à-dire du point B sur la figure 3.5-gauche, puis du point C sur la figure 3.5-droite.

[2] Si x est décroissante, la fonction f^x_{max} est utilisée pour réduire la borne droite et la fonction f^x_{min} pour la borne gauche.

Graphiquement, une itération de Newton intersecte $[l]$ avec la projection sur l'axe x du cône (en lignes pointillées) qui contient toutes les dérivées partielles. Comme la fonction est monotone, le cône forme un angle d'au plus 90 degrés et le diamètre de $[l]$ est divisé au moins par deux à chaque itération. Le processus dichotomique s'arrête quand $Diam([l])$ devient inférieur à $\epsilon \times Diam([x])$, le paramètre de précision (relative) ϵ étant spécifié par l'utilisateur.

Ce résumé cache plusieurs subtilités, en particulier plusieurs conditions permettant de ne pas déclencher systématiquement `LeftNarrowFmax` ou les procédures symétriques. Sans détailler (cf. chapitre D) :

- Si `MaxRevise` parvient à contracter la boîte (avec f_{max}), cela implique que `MonotonicBox-Narrow` ne peut contracter aucun intervalle $[x_i]$ plus finement avec $f_{min}^{x_i}$.
- De manière duale, si la procédure `MonotonicBoxNarrow` contracte $[x_i]$, alors il est inutile de rappeler `MinMaxRevise` qui ne peut espérer obtenir aucune contraction supplémentaire. C'est pourquoi `Mohc-Revise` ne contient pas de boucle !
- Si une procédure de type `[Left|Right]Narrow[Fmin|Fmax]` parvient à améliorer une borne de $[x_i]$ ($x_i \in X$), alors les autres appels de `[Left|Right]Narrow[Fmin|Fmax]` ne peuvent améliorer qu'*une seule* borne de *chaque* variable $x_j \neq x_i$.

La dernière propriété est une généralisation triviale de la propriété trouvée par Jaulin et Chabert et utilisée dans leur algorithme `Octum` (cf. ci-dessous). Ces propriétés expliquent le surcoût relativement faible en pratique de `MonotonicBoxNarrow` par rapport à `MinMaxRevise`.

3.3.4 Paramètres utilisateurs

L'algorithme `Mohc-Revise` possède deux paramètres ϵ et τ_{mohc}. Le paramètre de précision ϵ règle en quelque sorte un équilibre entre la finesse de contraction sur les variables monotones et le temps de calcul de la procédure `MonotonicBoxNarrow` pour y parvenir. Nos premières expérimentations montrent que ϵ peut être fixé à 10% sans grande incidence quand `Mohc` est utilisé comme sous-contracteur de `3BCID`[3]. Sans détailler, cette bonne nouvelle repose sur les propriétés mentionnées ci-dessus et sur les itérations de Newton qui permettent d'atteindre rapidement la précision requise, voire une précision plus grande.

Le deuxième paramètre τ_{mohc} est bien plus important en pratique et régule le recours à toute la machinerie liée aux monotonies (cf. algorithme 3). Pour chaque contrainte, le ratio $\rho_{mohc}[f]$ essaie de prédire si le traitement basé sur les monotonies qui suit est prometteur : les procédures correspondantes sont appelées seulement si $\rho_{mohc}[f] < \tau_{mohc}$. Ces ratios sont calculés dans une procédure de prétraitement (qui fait également office de test d'existence) appelée après chaque bissection, c'est-à-dire après chaque branchement dans l'arbre de recherche, comme suit :

$$\rho_{mohc}[f] = \frac{Diam([f]_M([B]))}{Diam([f]([B]))} = \frac{Diam([\underline{[f_{min}]}([B]), \overline{[f_{max}]([B])}])}{Diam([f]([B]))}$$

Ce ratio indique si l'image de f calculée par l'extension de monotonie est suffisamment plus

[3] D'autres expérimentations permettent de fixer aussi ϵ quand `Mohc` est utilisé seul...

petite (en pourcentage) que l'intervalle calculé par évaluation naturelle. Ce ratio est pertinent pour les phases ascendantes de `Minrevise` et de `Maxrevise`, ainsi que pour la procédure `MonotonicBoxNarrow` qui déclenche de nombreuses évaluations de f. Nos premières expérimentations suggèrent un réglage grossier de τ_{mohc} entre les valeurs 50% et 99% selon l'instance traitée.

3.3.5 Complexité, propriétés, travaux connexes

Complexité en temps

La complexité en temps de `Mohc-Revise` est en $O(n\,(e\,log_2(\frac{1}{\epsilon}) + k\,log_2(k)))$, où n est le nombre de variables dans la contrainte traitée c ; k est le nombre maximal des occurrences d'une même variable ; e est le nombre maximal d'opérateurs unaires et binaires dans c (c'est-à-dire le nombre de nœuds dans l'arbre représentant l'expression numérique de c) ; ϵ est la précision gérée par `MonotonicBoxNarrow` exprimée comme un pourcentage du diamètre des intervalles.

Propriétés

La procédure `Mohc-Revise` vérifie une propriété intéressante quand l'ensemble W est vide. Si les variables dans Y sont monotones ou bien si l'on utilise une procédure `TAC-Revise` (cf. section 2.4.3) à la place de `HC4-Revise`, `Mohc-Revise` est capable de calculer la consistance d'enveloppe (d'où le nom de l'algorithme), c'est-à-dire une boîte optimale. L'identification d'une sous-classe polynomiale au problème de l'enveloppe optimale a des implications pratiques, même dans le cas général. Nos recherches sur les unions d'intervalles (cf. section 3.4.2) suggèrent en effet que `HC4-Revise` produit souvent la même contraction que `TAC-Revise` assez haut dans l'arbre de recherche, si bien que la propriété ne devrait pas nécessiter l'utilisation de la coûteuse procédure `TAC-Revise` en pratique.

Travaux connexes

A notre connaissance, il existe deux autres algorithmes de propagation de contraintes sur intervalles cherchant à utiliser la monotonie des fonctions. Quand l'option *GradientSolve* est sélectionnée dans le solveur ALIAS [201], l'algorithme de `2B` conçu par J.-P. Merlet génère d'abord toutes les fonctions de projection de manière explicite (pas d'utilisation de `HC4-Revise`) et les évalue ensuite par monotonie dans la boucle de propagation.

La monotonie des fonctions a aussi été utilisée dans les systèmes quantifiés pour contracter rapidement une variable quantifiée universellement qui est monotone [106].

Jaulin et Chabert proposent dans [146] un contracteur appelé `Octum`, issu de travaux entamés au premier semestre de 2009 de manière indépendante de `Mohc`. Les deux algorithmes partagent une procédure `MonotonicBoxNarrow` très similaire. `Octum` se place néanmoins dans l'hypothèse plus restrictive où une fonction doit être monotone par rapport à chacune de ses variables, alors que `Mohc` commence à travailler avec les fonctions épaisses f_{min} et f_{max} dès qu'*une* variable devient monotone. Autrement dit, `Mohc` traite le cas général où la fonction possède aussi

des variables non détectées monotones à occurrence simple (Y) ou multiples (W). En particulier, `Octum` n'utilise pas la procédure `MinMaxRevise` pour contracter ces variables. De plus, la procédure `OccurrenceGrouping` de `Mohc` permet de réécrire rapidement les expressions pour détecter plus de cas de monotonies, comme nous l'introduisons à la section suivante.

3.3.6 Groupement d'occurrences pour augmenter les cas de monotonies

(Travail en collaboration avec Ignacio Araya, très moteur dans ces travaux, et Bertrand Neveu ; article repris au chapitre E.)

La procédure `OccurrenceGrouping` fonctionne sur chaque paire (f, x) de manière indépendante, où x est une variable à occurrences multiples. Pour bien comprendre l'idée derrière cette procédure, il faut d'abord rappeler comment le gradient de f est calculé par un procédé simple de différentiation automatique [28] qui s'applique sur l'arbre représentant l'expression de f. Une phase ascendante est la même que celle de `HC4-Revise` et attache à chaque nœud l'évaluation du sous-arbre correspondant. La phase descendante calcule en chaque nœud n la dérivée partielle $[\frac{\partial f}{\partial n}]([B])$ (en utilisant les dérivées de chaque fonction élémentaire et la composition des fonctions qui entraîne des produits d'intervalles). Quand cette phase atteint chaque occurrence x_i dans une feuille de l'arbre, on obtient donc bien (en $O(e)$ si e est le nombre de nœuds de l'arbre) chaque dérivée partielle $[\frac{\partial f}{\partial x_i}]([B])$. Le point important est que l'on déduit $[g_x] = [\frac{\partial f}{\partial x}]([B])$ en *sommant* les dérivées $[\frac{\partial f}{\partial x_i}]([B])$ obtenues pour chacune de ses occurrences x_i.

La première idée de Ignacio Araya a donc été de faire la remarque suivante. Si $[g_x]$ contient 0, on ne peut pas déduire que f est monotone par rapport x. Mais il est possible qu'il existe un *sous-groupe de ces occurrences*, que l'on remplace par une nouvelle variable auxiliaire x_a, tel que $[\frac{\partial f}{\partial x_a}]([x]) \geq 0$. Cette idée se formalise parfaitement en un programme linéaire en 0,1 sous contraintes. Le programme permet de caser chaque occurrence de x dans exactement l'un des trois groupes x_a (croissant), x_b (décroissant) ou x_c (non monotones), les contraintes garantissant le signe des dérivées par rapport à x_a et x_b et que les trois sous-groupes forment une partition. La fonction objectif est plus complexe, mais revient à minimiser le diamètre d'une surestimation de l'image de f utilisant une forme de Taylor, le but étant d'obtenir un intervalle image le plus étroit possible avec l'évaluation de monotonie.

Une troisième idée pousse l'idée encore plus loin. Plutôt que d'utiliser un programme linéaire en 0,1 (NP-difficile dans le cas général), on peut formaliser l'idée par un programme linéaire fractionnaire, sur les réels, qui peut donc se résoudre en temps polynomial avec une méthode du point intérieur et facilement avec une méthode du simplexe. Pour notre problème de groupement d'occurrences, cela signifie que chaque occurrence peut être remplacée par une combinaison linéaire convexe $r_a x_a + r_b x_b + r_c x_c$, avec $r_a + r_b + r_c = 1$. Le programme linéaire fractionnaire calcule les coefficients r_a, r_b, r_c pour chaque occurrence et produit ainsi une nouvelle formulation f^{og} de f où les occurrences sont en quelque sorte éclatées en trois parties.

Prenons par exemple la fonction $f_1(x) = -x^3 + 2x^2 + 6x$, pour laquelle nous voulons obtenir une bonne évaluation quand $[x] = [-1.2, 1]$. L'image de $[-1.2, 1]$ par la dérivée $f_1'(x) = -3x^2 + 4x + 6$

contient 0. L'image optimale[4] $[f_1]_O([x])$ est $[-3.05786, 7]$, mais l'évaluation naturelle donne $[-8.2, 10.608]$ et l'évaluation de Horner [136] (après factorisation) donne $[-11.04, 9.2]$. Le groupement fractionnaire d'occurrences donne $f_1(x) = f_1^{og}(x_a, x_b, x_c) = -x_a^3 + 2(0.35x_a + 0.65x_c)^2 + 6x_a$. L'extension aux intervalles $[f_1]_{og}$ de f_1 par groupement d'occurrences produit alors : $[f_1]_{og}([x]) = [f_1^{og}]_M([x]) = [-5.472, 7]$. On vérifie bien : $[f_1]_O([x]) \subseteq [f_1]_{og}([x]) \subseteq [f_1]_N([x])$.

Une dernière contribution est de proposer un algorithme ad-hoc pour résoudre le programme linéaire fractionnaire. Cet algorithme comporte deux procédures principales selon si $0 \in [G_m]$ ou $0 \notin [G_m]$, où $[G_m]$ désigne la somme des dérivées par rapport à chaque occurrence $[\frac{\partial f}{\partial o}]([x])$ qui ne comprennent pas 0. Cet algorithme est très rapide, dominé par un simple tri en $O(k\, log_2(k))$ sur les intervalles représentant les dérivées en chaque occurrence, où k est le nombre d'occurrences de x dans f. Cette complexité est à multiplier par le nombre de variables à occurrences multiples quand il est utilisé dans la procédure `Mohc-Revise`. De plus, l'algorithme prend en compte les problèmes d'arrondis sur les nombres flottants qui compliquent toujours l'utilisation d'un algorithme du simplexe dans les méthodes à intervalles. En pratique, les premières expérimentations montrent que le surcoût lié à l'appel à `OccurrenceGrouping` dans `3BCID(Mohc)` est faible alors qu'il permet de gagner parfois plus d'un ordre de grandeur grâce aux meilleures évaluations de monotonie offertes.

3.3.7 Conclusion

Nos premières expérimentations suggèrent que l'algorithme `Mohc` a le potentiel pour remplacer les contracteurs `HC4` et `Box` partout où ces derniers sont utilisés. Les premières expérimentations comparent `Mohc` à `HC4` et `Box` quand ces 3 algorithmes sont des sous-contracteurs de `3BCID`. Pour résumer, sur les instances traitées, avec une valeur de $\tau_{mohc} > 60\%$, `3BCID(Mohc)` est toujours meilleur en temps CPU ou équivalent à ses compétiteurs. `3BCID(Box)` n'est jamais compétitif et `3BCID(HC4)` enregistre une perte de performance comprise entre un facteur 2.7 et deux ordres de grandeur sur 9 des 17 instances.

Le succès par rapport à `Box` s'explique facilement. En plus de la procédure `MinMaxrevise` qui est peu coûteuse, `Mohc` déclenche la procédure `BoxNarrow` beaucoup moins souvent (seulement quand la fonction est monotone) que ne le fait `Box`. Il le fait en outre de manière beaucoup moins coûteuse à cause du processus dichotomique et des conditions de lancement de `LeftNarrowFMax` (et comparses). Des tests de *profiling*, qui apparaîtront dans la thèse de I. Araya, valident empiriquement ces remarques.

D'un point de vue académique, `Mohc` fournit une bonne illustration du succès des techniques de PPC : attendre que les points de choix nous amènent à vérifier des propriétés simples (ici, la monotonie), où des mathématiques de lycée résolvent facilement le problème !

[4] obtenue en résolvant préalablement $f_1'(x) = 0$, ce qui est un problème en soi dans le cas général...

3.4 Autres contributions

3.4.1 `PolyBox` : un algorithme de `Box` utilisant les fonctions extrêmes

(Travail en collaboration avec Yves Papegay, Gilles Chabert et Odile Pourtallier ; résumé accepté et présenté à SCAN 2008 [286] ; article court soumis et accepté à CP 2010 [287].)

Motivation

Rappelons d'abord que la procédure `BoxNarrow` du contracteur `Box` travaille sur une paire (f, x), c'est-à-dire avec une contrainte univariée $c_{[Y]} : f_{[Y]}(x) = f(x, [y_1], ..., [y_a]) = 0$, où chaque occurrence des variables Y de f autre que x est remplacée par son intervalle courant (cf. section 2.4.3). `BoxNarrow` contracte $[x]$ en un nouvel intervalle $[l, r]$ où l est le zéro le plus à gauche de $f_{[Y]}(x) = 0$ et r est le zéro le plus à droite. Un défaut de cette procédure est de parfois converger lentement pour trouver l et r du fait que $f_{[Y]}$ est une fonction intervalle, "épaisse".

L'idée derrière `PolyBox` consiste à transformer $f_{[Y]}$ en une nouvelle forme analytique $g_{[Y]}$ telle que les *fonctions extrêmes* $\underline{g_{[Y]}}$ et $\overline{g_{[Y]}}$ de $g_{[Y]}$ peuvent être extraites rapidement. On définit les fonctions extrêmes comme suit :

- Fonction *minimale* : $\underline{g_{[Y]}} = \min_{Y_s \in [Y]} f(Y_s, x)$
- Fonction *maximale* : $\overline{g_{[Y]}} = \max_{Y_s \in [Y]} f(Y_s, x)$

Les fonctions extrêmes sont tout simplement les courbes qui encadrent la fonction intervalle de la figure 2.2, page 32. Les travaux de Odile Pourtallier sur les polynômes dans l'équipe COPRIN ont eu comme effets de bord positifs d'identifier une forme "développée" favorable à l'extraction des fonctions extrêmes :

$$g_{[Y]}(x) = \sum_{i=0}^{i=k} f_i([Y]) \times h_i(x)$$

où :

- $h_i(x)$ dépend seulement de x, et non pas de Y.
- $h_i(x)$ a un signe qui ne change pas "trop souvent" comme : x^i (i entier), $log(x)$, e^x.

Dans ce cas en effet, $\underline{g_{[Y]}}(x)$ and $\overline{g_{[Y]}}(x)$ sont des fonctions par morceaux avec des coefficients ponctuels pris aux bornes des $f_i([Y])$, c'est-à-dire dans $\{\overline{f_i([Y])}, \underline{f_i([Y])}\}$.

Notre première implémentation dans `Ibex` se limite aux polynômes. $f_{[Y]}(x)$ est un polynôme et $g_{[Y]}(x) = \sum_{i=0}^{i=d} f_i([Y]) \times x^i$, d étant le degré du polynôme. Notons que si i est pair alors x^i est positif. Si i est impair alors le signe de x^i peut changer une seule fois en $x = 0$.

Considérons par exemple $f(x, \{y, z\}) = (y + z) \times x^2 + (2yz) \times x + sin(z)$ et $g_{\{[y],[z]\}}(x) = [-2, 3]x^2 + [-4, -2]x + [-1, 1]$. Les fonctions extrêmes sont alors :

- $x \geq 0 : \overline{g_{\{[y],[z]\}}}(x) = 3x^2 - 2x + 1$
- $x \leq 0 : \overline{g_{\{[y],[z]\}}}(x) = 3x^2 - 4x + 1$
- $x \geq 0 : \underline{g_{\{[y],[z]\}}}(x) = -2x^2 - 4x - 1$

– $x \leq 0 : \underline{g_{\{[y],[z]\}}}(x) = -2x^2 - 2x - 1$

L'idée principale est de produire une variante de `BoxNarrow`, appelée `polyBoxRevise` qui travaille avec ces fonctions ponctuelles.

L'algorithme `PolyBox`

Le cœur de la procédure `PolyBoxRevise` détermine un intervalle $[l, r]$ obtenu par contraction de $[x]$. Sans perte de généralité, détaillons le principe de détermination de la borne gauche l.

Une première étape triviale détermine avec quelle fonction extrême travailler. Trois cas peuvent se présenter :

1. Si $\underline{g_{[Y]}}(\underline{x}) \leq 0$ et $0 \leq \overline{g_{[Y]}}(\underline{x})$: $l = \underline{x}$ (pas de contraction).
2. Si $\underline{g_{[Y]}}(\underline{x}) > 0$: on travaille avec $\underline{g_{[Y]}}$.
3. Si $\overline{g_{[Y]}}(\underline{x}) < 0$: on travaille avec $\overline{g_{[Y]}}$.

Le cas 2 (ou 3) est intéressant et requiert un dernier test du degré du polynôme.

Si le degré d de $g_{[Y]}$ est supérieur à 4 (dans notre implémentation : $d \geq 4$), alors la plus petite racine l de $\overline{g_{[Y]}}(x) = 0$ dans $[x]$ est calculée avec la procédure `BoxNarrow` (ou `LeftNarrow`) classique, la différence étant que la convergence est plus rapide puisque la fonction est ponctuelle.

Si au contraire $d \leq 4$ (dans notre implémentation : $d < 4$), alors on calcule toutes les racines *réelles* de manière analytique/formelle et on retourne la plus petite solution dans $[x]$. Le cas $d = 2$ est trivial et fait appel aux souvenirs de collège ou lycée. Il faut simplement veiller à rendre les évaluations successives conservatives, à cause des problèmes d'arrondis sur les flottants, en prenant initialement un intervalle dégénéré autour de chaque coefficient.[5] Le cas $d = 3$ est plus compliqué et il fallu quelques heures et la polyvalence de mon collègue Yves Papegay pour choisir la célèbre méthode de Cardano et la transformer en une procédure informatique.

Au final, nous proposons une procédure `PolyBoxRevise` bâtie sur le modèle de l'algorithme `Box` nantais, appelé `BC4` [28]. Si $f_{[Y]}(x)$ ne contient pas d'occurrence multiple de x, la procédure fait un simple appel à `HC4-Revise` (comme `BC4`). Dans le cas contraire, `Mathematica` essaie de transformer $f_{[Y]}(x)$ en $g_{[Y]}(x) = \sum_{i=0}^{i=d} f_i([Y]) \times x^i$. Quatre cas peuvent se présenter :

1. `Mathematica` échoue, ce qui signifie que $f_{[Y]}$ n'est pas polynomiale :
   la procédure fait un simple appel à `BoxNarrow` (ou `HC4-Revise` dans une version hybride).
2. $g_{[Y]}(x)$ contient une seule occurrence de x :
   `HC4-Revise` est appliquée à $g_{[Y]}(x)$.
3. $g_{[Y]}(x)$ a des occurrences multiples et $d < 4$:
   détermination analytique de la plus petite racine de $g_{[a]}(x) = 0$.
4. $g_{[Y]}(x)$ a des occurrences multiples et $d \geq 4$:
   détermination numérique (avec `BoxNarrow`) de la plus petite racine de $g_{[Y]}(x) = 0$.

[5] La version intervalles de la détermination des solutions réelles souligne le raisonnement ensembliste permis par les intervalles. Il suffit en effet de tester que la racine carrée du discriminant est "vide" (au sens intervalle/ensembliste) pour détecter l'absence de solutions réelles.

Remarques

Le deuxième cas ci-dessus s'illustre par exemple sur l'une des équations de l'instance `Caprasse` : $-2x+2txy-z+y^2z=0$ que `Mathematica` réécrit sous la forme : $x(-2+2ty)+(-1+y^2)z=0$ (pour la contraction de $[x]$ ou $[z]$). On voit bien que la forme réécrite a fait disparaître les occurrences multiples de x et z.

L'utilisation d'un outil de calcul formel a plusieurs vertus. D'abord, la nouvelle forme produite n'est pas à proprement parlée une forme entièrement développée. Il s'agit bien d'une forme développée par rapport à x, mais la procédure de calcul formel cherche au contraire une forme des coefficients $f_i([Y])$ qui limite les occurrences multiples des variables de Y et donc généralement la surestimation de $[f_i]_N([Y])$. L'exemple de l'équation dans l'instance `Caprasse` montre une transformation réussie qui se traduit au final par un gain en temps. Une équation de l'instance `6body` montre au contraire une transformation contreproductive de $5(B-D)+3(b-d)(B+D-2F)=0$ en $B(5+3(b-d))+(-5+3b-3d)D+6(-b+d)F=0$ (pour la contraction de $[B]$ ou $[D]$) qui augmente la surestimation à cause des occurrences multiples additionnelles des variables a, b et d. Cet exemple souligne que la contraction obtenue au final dépend de la forme utilisée. Si `PolyBoxRevise` obtient cette contraction rapidement grâce aux fonctions extrêmes (ponctuelles), la contraction obtenue est en revanche incomparable à celle obtenue par `BoxNarrow` sur la forme initiale. (C'est pourquoi notre implantation ajoute systématiquement au début de `PolyBoxRevise` un appel à `HC4-Revise` sur la forme initiale.)

Notons aussi qu'un outil de calcul formel permet de produire une forme spécifique pour chaque paire (f,x) considérée dans `PolyBoxRevise`.

Comparaison avec la `Box` de Numerica

Van Hentenryck, Michel et Deville ont également utilisé l'idée des polynômes extrêmes (sans utiliser ce vocabulaire) dans leur outil de résolution à intervalles `Numerica`. L'idée n'est pas décrite dans leur célèbre livre [293] mais introduite en deux pages dans un article technique [291]. L'intégration de cette idée dans leur outil est sensiblement différente de la nôtre. `Numerica` utilise différentes formes des équations : une forme "naturelle", une forme de Taylor (pour y appliquer un Newton intervalles sans préconditionnement) et une forme développée qui permet d'utiliser les fonctions extrêmes. Une propagation séparée est lancée sur le système sous forme développée.

`PolyBox` suit au contraire un schéma proche de `BC4` en gérant un unique système avec des procédures de révision adaptées à chaque paire (f,x). Notons que `BC4` a en quelque sorte oublié les polynômes extrêmes de `Numerica` et que `PolyBox` peut donc se voir comme un moyen de les réhabiliter dans un schéma moderne. `PolyBox` apporte quelques améliorations par rapport à son prédécesseur dans `Numerica` :

- Il peut utiliser une forme adaptée à la contraction de chaque variable, alors qu'une forme entièrement développée, par rapport à toutes les variables, entraîne souvent une forte surestimation, ce qui souligne l'intérêt d'utiliser un outil de calcul formel (voir les remarques plus haut).
- Comme `BC4`, `PolyBox` utilise aussi `HC4-Revise` quand x apparaît une seule fois dans f.

– La résolution analytique des polynômes de bas degré est ajoutée.

Résultats obtenus

On trouve à l'annexe F des résultats expérimentaux qui montrent des gains intéressants d'une stratégie de recherche basée sur `PolyBox`+Newton par rapport aux stratégies `Box` + Newton et `HC4`+Newton. Malheureusement, à quelques exceptions près, `PolyBox` n'est pas concurrentiel avec `3BCID(HC4)` et surtout avec `3BCID(Mohc)` (voir les commentaires à la section 3.3.7 sur la comparaison entre `BoxNarrow` et `MohcRevise`). A cause du coût très peu élevé de `PolyBoxRevise` quand la contraction est obtenue de manière analytique (polynômes de degrés 2 et 3 dans notre implantation), une idée à retenir peut-être serait de garder uniquement ces procédures comme des contracteurs spécifiques, similaires à des *contraintes globales* [241, 1, 183], et de les ajouter automatiquement dans une stratégie de recherche.

3.4.2 Domaines représentés par des unions d'intervalles

(Travail en collaboration avec Gilles Chabert et Bertrand Neveu ; Cf. la thèse de Gilles Chabert, ainsi que les articles à SAC 2005 [53] et JFPC 2005 [54])

La motivation de ce travail est d'essayer de réduire les domaines des variables non pas seulement sur les bornes, mais aussi à l'intérieur, dans les "trous", en gérant non pas de simples intervalles mais des domaines représentés par des unions d'intervalles. Un algorithme de propagation de contraintes qui travaillerait avec des unions d'intervalles et essaierait de calculer l'arc-cohérence du système pourrait malheureusement parfois générer une infinité d'intervalles dans les domaines dans un modèle théorique sur les réels. En utilisant les nombres flottants, cela signifie qu'un algorithme de propagation pourrait générer très rapidement un nombre d'intervalles dans les domaines égal au nombre de flottants. Un exemple dû à Chabert et soulignant cette explosion est illustré à la figure 3.6. Il est formé des deux contraintes $c_1 : x = y$ et $c_2 : (\frac{3}{4}(x-5)^2) = y$ dans la boîte $[1,9] \times [1,9]$.

La première projection sur x de la contrainte c_2 (avec `TAC-Revise`, la variante combinatoire de `HC4-Revise` qui ne calcule pas d'enveloppe) produit un premier "trou" dans $[x]$. La projection sur y de c_1 produit un trou dans $[y]$, etc. On obtient ainsi, au bout de k itérations, des unions d'intervalles contenant 2^k intervalles. C'est précisément ce que calcule l'algorithme de Hyvönen [140] en effectuant un point de choix à chaque découpage *naturel*, c'est-à-dire au niveau d'un trou découvert dans une projection. Nous avons proposé deux algorithmes de propagation de contraintes permettant d'éviter cette explosion combinatoire en pratique et permettant de calculer une consistance partielle plus forte que l'arc-cohérence.

Algorithme *I-cohérence globale* (IGC)

L'idée derrière IGC consiste à enrichir la structure d'union d'intervalles en étiquetant chaque intervalle avec une information indiquant sur quelles boîtes arc-cohérentes il prend support. Cette vue macroscopique est mise en œuvre par un ensemble d'*étiquettes* attachées à chaque intervalle

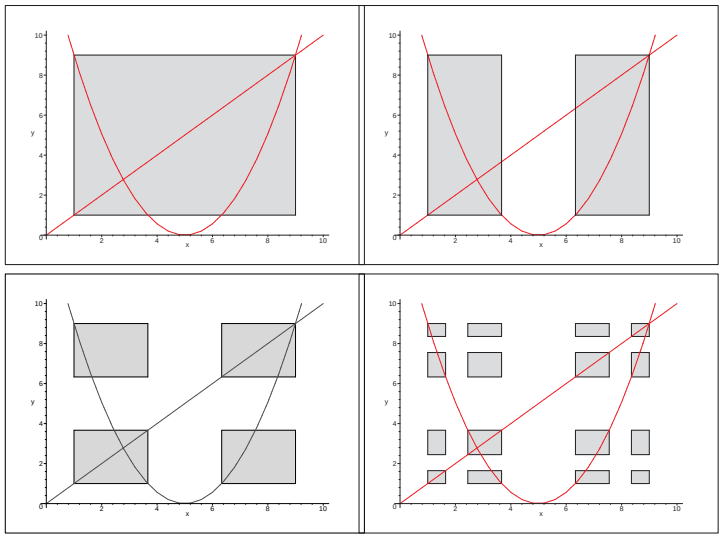


FIG. 3.6 – Filtrage sur des unions d'intervalles

et propagées au fil des projections. Illustrons ce principe sur notre exemple. Lors de la première projection sur x (quadrant en haut à droite), IGC attache l'étiquette x^- (correspondant à la projection $-\sqrt{}$) à l'intervalle le plus petit et l'étiquette x^+ (correspondant à la projection $+\sqrt{}$) au plus grand. Lors de la première projection sur y (quadrant en bas à gauche), l'intervalle le plus petit de y "hérite" de l'étiquette x^-, alors que l'intervalle le plus grand hérite de x^+. Ainsi, la situation décrite dans le quadrant en bas à droite ne peut pas se produire. L'intervalle de x le plus petit est éliminé car il est étiqueté x^- mais provient d'une projection d'un intervalle de y étiqueté x^+, ce qui est contradictoire. Le troisième intervalle de x est également éliminé car il est étiqueté x^+ alors qu'il provient d'une projection de y étiquetée x^-.

Les étiquettes permettent ainsi de maintenir au fil des projections l'ensemble des intervalles qui sont compatibles entre eux. Cette vue macroscopique permet d'éliminer simplement les faux supports, c'est-à-dire les intervalles qui ne forment pas une clique dans cette structure supplémentaire (d'où le terme de cohérence globale - au niveau macroscopique - dans le nom de l'algorithme). L'implantation de l'algorithme correspondant fait appel à une structure sophistiquée, apparentée au BDD de Bryant [44], qui permet de stocker de manière concise pour chaque variable un ensemble d'étiquettes compatibles entre elles. Chaque projection consulte alors cette structure pour éliminer parfois un intervalle donné du domaine de la variable considerée. Cet algorithme est limité aux systèmes polynomiaux (pour lesquels il suffit de gérer deux parties monotones), mais est théoriquement extensible aux systèmes quelconques en gérant un nombre arbitraire de parties monotones, lors d'une projection d'une fonction *sinus* par exemple. IGC calcule une consistance partielle plus forte que l'arc-cohérence du système ternarisé, ce qui est une contribution académique appréciable. L'explosion combinatoire liée à ces faux supports

est effectivement maîtrisée en pratique. En théorie, en utilisant les slogans à la mode, on peut considérer cet algorithme comme un algorithme à paramètres fixes (*fixed-parameter algorithm*), c'est-à-dire qu'il troque un facteur exponentiel en le nombre de flottants (qui produit parfois une explosion en pratique) contre un facteur exponentiel en le nombre de fonctions élémentaires pouvant créer un trou. Même si nous n'avons pas observé d'explosion liée à ce dernier facteur, le surcoût du maintien de la structure de données sophistiquée ne rend pas l'approche concurrentielle avec l'algorithme suivant.

Algorithme *Lazy BoxSet*

Pour le décrire simplement, Lazy BoxSet peut se voir comme l'assemblage d'une variante de l'algorithme de 2B-cohérence (comme `HC4`) et d'une stratégie de découpage *naturel* qui s'applique sur un domaine formé d'une union d'intervalles dans un trou entre deux intervalles consécutifs. Trois étapes principales constituent cet algorithme :

1. L'algorithme `HC4` est d'abord appelé. L'algorithme `HC4` est en fait modifié, sans pratiquement aucun surcoût, de manière à mémoriser une contrainte susceptible de produire un trou.
2. Une fois le point-fixe de `HC4` atteint, cette contrainte est alors traitée par l'algorithme `TAC-Revise`, la variante combinatoire de `HC4-Revise` (cf. section 2.4.3).
3. Le branchement suivant dans l'arbre de recherche effectue alors un découpage naturel sur l'une de ces variables.

Ce processus est itéré jusqu'à l'obtention en chaque feuille de cet arbre d'une boîte arc-cohérente. Vue comme une consistance partielle, la propriété obtenue est plus forte que l'IGC-cohérence. Autrement dit, l'ensemble des feuilles arc-cohérentes (obtenu par Lazy BoxSet) est plus restreint que l'ensemble des boîtes obtenues par produit cartésien des domaines (union d'intervalles) IGC-cohérents.

Un algorithme similaire à *Lazy Box-set* a été obtenu (par des voies différentes) par Heikel Batnini dans sa thèse [22, 23].

Les premiers résultats expérimentaux de Chabert comme de Batnini sont décevants. Si Lazy Box-set ne produit pratiquement aucun surcoût, les exemples de gain produits par le découpage naturel restent à identifier. Les gains sont généralement marginaux voire négatifs puisque le découpage naturel empêche en quelque sorte l'utilisation d'une autre heuristique de choix de variable. Dans les rares cas d'amélioration sensible, il semble que cela soit dû à un choix précoce d'une seule variable qui permette de résoudre rapidement le problème indépendamment des trous dans son domaine, bref à un heureux effet de bord. Pourtant une étude plus exhaustive mériterait d'être menée, simplement à cause d'une perspective possible liée à l'algorithme `Mohc` (introduit à la section 3.3).

Perspective : une variante de Lazy BoxSet en amont de l'algorithme `Mohc` ?

Une variante de Lazy BoxSet permettrait d'augmenter les cas de monotonie que `Mohc` pourrait exploiter. Lors de la projection d'une fonction élémentaire, il suffirait de mettre dans le domaine du nœud correspondant l'union de *chaque* intervalle correspondant à une partie monotone de la fonction. (La version actuelle peut rassembler en un seul intervalle non monotone deux parties monotones qui sont simplement continues sur l'intervalle englobant.) Un découpage naturel augmenterait alors les cas de monotonie que l'algorithme de contraction `Mohc` pourrait exploiter.

Cette idée séduisante mérite approfondissement mais ne me paraît pas prometteuse à première vue. Un découpage naturel sur les variables comme sur les nœuds intermédiaires est en effet coûteux et calcule une propriété bien trop forte. Pour obtenir une fonction monotone, il n'est en effet pas nécessaire d'obtenir la monotonie sur tous ses opérateurs élémentaires (les nœuds intermédiaires)...

3.4.3 `Ibex`

`Interval-Based EXplorer` est un outil de résolution de systèmes de contraintes utilisant des méthodes à intervalles [52, 50]. `Ibex` est essentiellement l'œuvre de Gilles Chabert qu'il a commencée pendant sa thèse (appelée alors `IcosAlias`). Après sa thèse, Chabert a fait de `Ibex` une bibliothèque libre et invite depuis les contributeurs à se joindre à cette micro-communauté. Lors de son post-doctorat avec Luc Jaulin à Brest, il a également produit avec celui-ci une couche au dessus de `Ibex`, nommée `Quimper`, qui masque la hiérarchie de classes et offre un petit interprète et une interface graphique facilitant l'accès de l'outil aux utilisateurs non informaticiens issus de l'ingéniérie (physique, automatique, robotique, traitement du signal, etc). Bertrand Neveu et moi-même sommes responsables des algorithmes de programmation par contraintes inclus dans `Ibex` et, avec Ignacio Araya, utilisons `Ibex` pour développer l'ensemble de nos méthodes à intervalles. Nous participons également aux discussions sur l'architecture générale et la hiérarchie de classes.

Cette bibliothèque en `C++` permet de trouver l'ensemble des solutions d'un système mais ne permet pas (encore) d'optimiser une fonction objectif. En plus d'un système classique, `Ibex` permet de définir facilement un système quantifié contenant des paramètres existentiels ou universels ou de générer des boîtes intérieures (ne contenant que des solutions). L'arithmétique des intervalles est mise en œuvre par la bibliothèque BIAS/PROFIL (corrigée). `Ibex` a comme priorités la flexibilité, afin de facilement piloter les différentes briques algorithmiques incluses, et l'extensibilité. En particulier, une liste de *contracteurs* (de boîtes intérieures ou extérieures) est appliquée entre chaque branchement. Cela permet, entre autres, de mettre en œuvre des *sous-systèmes filtrants* qui me semblent importants en pratique. Similaires à des contraintes globales [241, 1, 183], ces contracteurs filtrent un sous-système (déterminé manuellement ou automatiquement) à l'aide d'un algorithme dédié et propagent les réductions obtenues dans le reste du système. Un exemple d'application est l'algorithme `Box-k` décrit au chapitre J.

Un autre atout de `Ibex` est sa robustesse, c'est-à-dire une absence de perte de solution en utilisant les contracteurs présents dans la distribution actuelle (à ma connaissance). `Ibex` possède

également des limites, notamment la non-utilisation de briques de programmation linéaire fiable, comme dans l'algorithme `Quad` de `Icos` [183]. Soulignons aussi, contrairement à ALIAS [201], la non-utilisation des dérivées secondes des fonctions qui permettraient une évaluation plus fine des dérivées premières et l'appel à d'autres algorithmes d'analyse numérique, comme Kantorovitch.

Pour conclure, `Ibex` a été et demeure toujours un outil indispensable pour développer la plupart des méthodes à intervalles décrites dans ce mémoire. Certains choix d'architecture et la hiérarchie de classes maintenant plus stable méritent dorénavant d'être présentés à la communauté. L'architecture d'`Ibex` est aujourd'hui suffisamment mûre pour que des spécialistes enrichissent cette bibliothèque. `Ibex` a le potentiel pour fédérer une partie des contributions sur les méthodes à intervalles, notamment celles issues de PPCI et des applicatifs (cf. sections 2.7 et 2.9). La section suivante place `Ibex` dans un cadre plus large et souhaitable d'évolution des logiciels de résolution de systèmes de contraintes.

3.5 Perspectives

En plus des perspectives à court terme mentionnées dans les sections correspondantes, les paragraphes suivants donnent quelques perspectives de recherche à plus long terme.

Apports de la PPCI

Dans les années 1960 et 1970, la communauté d'analyse par intervalles proposait schématiquement un modèle de résolution comprenant entre chaque point de choix un appel à un test d'existence basé sur la monotonie et un appel à un Newton intervalles. La communauté de programmation par contraintes a proposé notamment d'ajouter à ce schéma le contracteur `HC4` ou `Box` (parfois `3B`).

Notre première contribution en PPCI est de proposer un schéma sensiblement différent, à savoir :

1. un prétraitement par `I-CSE` pour extraire les sous-expressions communes ;
2. pendant la recherche arborescente, entre deux bissections, un appel à `3BCID(Mohc)` – ou `ACID(Mohc)` (suivi d'un appel à Newton intervalles).

Nous pensons que ce schéma est robuste dans le sens que la perte de performance est peu importante par rapport au schéma précédent de PPCI sur les instances de la page de COPRIN [204] les moins favorables (quelques pourcentages, rarement un facteur 2 ou 3), alors que des ordres de grandeur sont gagnés sur les instances les plus favorables. Par rapport au schéma historique de l'analyse par intervalles (sans appel à des opérateurs de PPC), les gains sont d'un à plusieurs ordres de grandeur sur la plupart des instances.

Ces comparaisons reposent sur des expérimentations sur un nombre donné d'instances, certes variées, mais ne préjugent en rien de l'intérêt de telle ou telle méthode pour une classe de problèmes particulière. Par exemple, les problèmes de détermination de valeurs propres avec incertitudes constituent des systèmes quasi-linéaires sur intervalles qui sont généralement mieux résolus par des techniques adaptées aux systèmes linéaires [126].

Le schéma proposé n'est évidemment pas figé et n'échappera notamment pas à la confrontation (et/ou union) avec l'opérateur `ContractRelax` (programmation linéaire) que nous avons peu étudié jusqu'à présent.

Linéarisation versus PPC

Notons que `ContractNewton` comme `ContractRelax` (programmation linéaire) dans le schéma présenté à la section 2.3 reposent sur une approximation linéaire du système. L'opérateur générique `ContractPPC` ne linéarise pas le système. Il "attend" au contraire qu'après quelques branchements dans l'arbre de recherche, une contrainte soit suffisamment simple pour en tirer le maximum d'informations et contracter la boîte courante. Autrement dit, il travaille avec les parties continues (`HC4-Revise`) ou monotones (`Mohc-Revise`), pour autant non linéaires, des contraintes. `3B` et `3BCID` ont une portée plus globale et peuvent être considérés comme effectuant des points de choix polynomiaux, qui n'ont toujours aucun rapport avec une relaxation linéaire (ou convexe). Il est important de souligner cette différence.

Quel "modèle" de résolution ?

En adoptant une vision purement combinatoire, on peut faire une comparaison rapide entre le modèle des CSP en domaines finis, le problème SAT (satisfiabilité d'une formule booléenne) et la résolution des systèmes de contraintes abordée dans ce mémoire (problème appelé parfois CSP numérique/continu ou NCSP). Le modèle SAT est celui qui s'exprime le plus simplement, ce qui explique les progrès impressionnants de ce domaine depuis 20 ans. Le modèle des NCSP qui nous intéresse s'exprime à mon avis plus simplement que le modèle des CSP sur domaines finis. Les contraintes sont définies en effet avec un jeu d'opérateurs donné et relativement restreint (opérateurs arithmétiques, puissances entières, fonctions trigonométriques, exponentiel, logarithme), et les domaines s'expriment de manière homogène par deux bornes flottantes. Cela pose néanmoins une question importante :

– Sous quelle forme réécrire le système de contraintes posé par l'utilisateur ?

Cette question dépend évidemment de l'algorithme utilisé, d'où l'intérêt de proposer un solveur pouvant utiliser plusieurs contracteurs, avec éventuellement une forme adaptée à chacun des contracteurs. Mais il serait plus enthousiasmant d'un point de vue académique de trouver un petit jeu de formes de systèmes (idéalement une seule) traitées par un petit jeu d'algorithmes. Cette question a déjà été abordée sous différents angles.

Pour les sous-systèmes polynomiaux, l'algorithme `Quad` [183] et les méthodes utilisées dans `GloptOpt` [69] (propagation de contraintes, coniques) suggèrent de réécrire les termes de fort degré en termes quadratiques.

`I-CSE` souligne la grande importance d'extraire les sous-expressions communes dans un système. On peut néanmoins aller beaucoup plus loin en jouant sur la forme des contraintes et du système. En jouant sur les factorisations, les développements ou les combinaisons linéaires entre les contraintes, on peut diminuer les occurrences multiples de variables (feuilles des arbres

représentant les expressions) ou de sous-expressions (variables auxiliaires, nœuds intermédiaires de ces arbres). J.-P. Merlet effectue souvent ce genre de manipulations avec Maple avant de traiter le système résultant avec ALIAS.
L'automatisation et la formalisation de ce problème est sans nul doute un défi académique et pratique majeur.

Un troisième angle de vue est lié aux occurrences multiples des variables. Si on met chaque contrainte sous forme *ternarisée*, c'est-à-dire si l'on remplace chaque opérateur par une variable intermédiaire, les contraintes appartiennent toutes à un catalogue restreint de contraintes *élémentaires* sans occurrences multiples et on obtient ainsi un système sous une forme "canonique". Quoique le nombre de variables ainsi créées puisse être très important, on peut se demander lesquelles bissecter, lesquelles rogner (avec `3BCID` ou `ACID`), lesquelles traiter par propagation de contraintes. Ali Baharev semble avoir obtenu quelques résultats en bissectant prioritairement certaines variables auxiliaires dans les (gros) systèmes de contraintes qu'il traite. L'automatisation de ces intuitions semble très intéressante malgré les résultats plutôt décevants de nos premières études sur ce sujet.

Ces résultats mitigés suggèrent pour le moment de garder, en l'améliorant, notre schéma actuel. `I-CSE` peut être amélioré en jouant sur les formes des équations (factorisées, développées) et en faisant apparaître sous forme de variables auxiliaires à occurrences multiples les sous-expressions communes. `ACID(Mohc)` s'applique ensuite sur des contraintes contenant des occurrences multiples de variables (initiales ou auxiliaires), que `Mohc-Revise` se charge de traiter efficacement dès que des monotonies apparaissent au cours de la recherche.

Finalement, la capacité de `Mohc-Revise` à surmonter le problème des occurrences multiples en cas de monotonie suggère, non pas de décomposer les contraintes initiales en contraintes élémentaires, mais au contraire de combiner les contraintes initiales en des contraintes équivalentes ou redondantes que `Mohc-Revise` traiterait comme des sous-systèmes (contraintes globales).

Newton intervalles

Rappelons que seuls les algorithmes de type Newton intervalles multivarié sont capables de garantir qu'une solution à un système *d'équations* existe dans une boîte donnée, ou qu'un optimum global courant (trouvé par exemple par une recherche locale) corresponde vraiment à un point faisable. Pourtant, on sait aussi que Newton intervalles est intrinsèquement incapable de garantir l'existence d'une solution dans certains cas, par exemple quand une solution double apparaît (ex : cas $x^2 = 0$). D'autre part, la complexité cubique de cet algorithme rend son utilisation rhédibitoire pour garantir un point faisable en optimisation globale où sont traitées parfois des centaines de contraintes. C'est une conclusion de Yahia Lebbah (communication personnelle) concernant la comparaison des performances entre l'outil d'optimisation globale dont il est le principal auteur [183] et le célèbre `Baron` [250].

D'un point de vue contraction, Newton intervalles est de plus en plus remis en question, mis en concurrence avec les opérateurs de PPCI et de programmation linéaire. Dans nos expérimentations, Newton intervalles semble très utile pour éliminer des boîtes atomiques proches d'une

solution, mais surtout quand on utilise `HC4` comme autre algorithme de contraction. L'intérêt de Newton intervalles diminue sensiblement quand on utilise un contracteur plus fort comme `3BCID`. Kolev fait également un constat négatif d'un point de vue théorique (et pratique ?) quand il compare Newton à sa technique de linéarisation [173]. Nos expérimentations sur ses exemples jouets ne semblent pas confirmer ses constatations. Il est en tout cas curieux de constater que la première version du solveur `GloptLab` [69] développé par Ferenc Domes, doctorant de Arnold Neumaier à Vienne, ne comprend pour le moment pas de Newton intervalles !

Des recherches (vaines) pour améliorer Newton intervalles nous ont au moins permis de comprendre l'intérêt du préconditionnement dans cet algorithme. Pour les performances, le temps requis pour atteindre un point-fixe est bien moins important que la qualité de la contraction obtenue grâce au préconditionnement. L'étape de prétraitement reste peut-être une (dernière ?) source d'amélioration des algorithmes de type `ContractNewton`, et le préconditionnement à droite proposé par Goldsztejn et Granvilliers [105] offre une alternative qui mérite d'être approfondie.

Complexité théorique

Si nous sommes quelques uns à considérer la résolution de systèmes de contraintes sur les réels comme un problème combinatoire, au même titre que les modèles SAT, CSP ou PLNE (MIP), on peut constater le manque de résultats de complexité théorique concernant les problèmes sur les réels [171]. La nature non discrète de ces problèmes rend en fait inopérants l'utilisation des outils classiques de mathématiques discrètes issus de la logique.

Les premiers résultats de complexité théorique connus sur le "continu" concernent la programmation quadratique dans les années 1970. Sur les *nombres rationnels* (codés sur des bits), optimiser une fonction objectif quadratique sous contraintes d'inégalités linéaires est NP-difficile [251, 96]. Le même article montre que le problème est également NP-difficile quand la fonction objectif est linéaire et les contraintes d'inégalités quadratiques. Ces problèmes tombent dans P quand les contraintes et la fonction objectif deviennent convexes. En revanche, sur une machine de Turing classique, on ne sait pas si le problème $\mathtt{HN}_{\mathbb{R}}$ est décidable ou non. Le fameux problème $\mathtt{HN}_R$ (*Hilbert's Nullstellensatz*) consiste à décider si un système de n équations polynomiales a une solution sur R^n, où R est un corps quelconque (contrairement à la programmation quadratique, il n'y a pas d'objectif à optimiser).

A la fin des années 1980, des chercheurs ont travaillé sur des extensions de la machine de Turing aux nombres réels [36, 35]. Leurs résultats sont en fait applicables à n'importe quel corps R. Comme le problème SAT dans la théorie "classique", Blum et al. ont montré que le problème $\mathtt{HN}_{\mathbb{R}}$ est $\mathrm{NP}_{\mathbb{R}}$-complet sur leur machine de Turing étendue, dite universelle (appelée aussi BSS - Blum, Schub, Smale). D'autre part, on a trouvé des algorithmes en temps exponentiel pour décider de la satisfaction des systèmes polynomiaux [242, 243], on sait donc que $\mathtt{HN}_{\mathbb{R}} \in EXP_{\mathbb{R}}$.

Plusieurs articles sur le sujet montrent que le problème $\mathtt{HN}_{\mathbb{R}}$ joue un rôle central en théorie de la complexité sur les réels. Concernant $\mathbb{R}$, une série de cinq articles de Mike Shub et Steve Smale sur le théorème de Bezout, qui est lié à $\mathtt{HN}_{\mathbb{R}}$, donne une analyse détaillée de la complexité de ce problème y compris quand il est résolu approximativement ou de manière probabiliste [259,

260, 261, 262, 263].

Revenons à nos méthodes à intervalles en remarquant qu'elles cherchent également à résoudre $\mathtt{HN}_{\mathbb{R}}$ de manière approximative, tout en continuant à travailler avec nos machines de Turing classiques (à 500 euros) ! Une autre voie plus modeste que le modèle BSS, mais aussi plus simple, et permettant de justifier directement les explorations arborescentes de nos solveurs, consisterait ainsi peut-être à déterminer la complexité théorique du problème $\mathtt{HN}_{\mathbb{R}}$ traduit dans notre "langage intervalles", en approximant une solution par une boîte atomique (de largeur un u.l.p. sur chaque dimension) non rejetable par évaluation des fonctions. La solution est donc non garantie (le problème correspondant étant probablement indécidable), mais approximée comme on le fait dans les solveurs à intervalles utilisant des contracteurs de PPCI ou de relaxation linéaire "fiables". Un peu plus formellement :

- Une instance $P = (X, C, [X])$ de NCSP est définie par un ensemble X de variables dans $\mathbb{IR}$, un ensemble C d'équations polynomiales (avec des coefficients flottants, non intervalles) et une boîte $[X]$ avec des bornes flottantes (ou rationnelles, ou appartenant à tout autre ensemble discret représentable sur un ordinateur...). ($[X]$ contient donc un ensemble fini de points sur $\mathbb{F}^n$. On peut se restreindre aux problèmes carrés de taille $n = |X| = |C|$. La taille de l'entrée est polynomiale en n et en le nombre e d'opérations binaires ou unaires dans chaque expression.)
- Une solution $[B']$, si elle existe, est une boîte atomique incluse dans $[X]$ telle que pour toute fonction f ($f(X) = 0 \in C$), on a : $0 \in [f]_N([B'])$.

Le test d'existence de "solution" (non garantie) s'effectue par évaluation naturelle ($[f]_N([B'])$) ou toute autre évaluation se calculant en temps polynomial.

Le problème de l'existence d'une solution d'un NCSP, défini comme ci-dessus, est évidemment dans la classe NP, la vérification d'une solution par évaluation naturelle étant polynomiale en temps. Cependant, NCSP n'a pas été montré NP-complet à ma connaissance. Il faudrait trouver une transformation polynomiale à partir d'un problème comme SAT (3SAT) ou à partir de problèmes "proches" sur les intervalles prouvés NP-difficiles par Kreinovich et al. (mais dont la transformation polynomiale vers NCSP ne me paraît pas si triviale) :

- l'évaluation optimale de l'image (intervalle) d'un polynôme sur intervalles [178] ;
- le calcul de l'enveloppe optimale d'un système linéaire sur intervalles [177, 244, 123] : les équations sont de la forme $\sum[a_{ij}]x_j = [b_i]$ (les $[a_{ij}]$ et les $[b_i]$ ont des bornes rationnelles), l'ensemble solution est l'ensemble des x_j tel qu'il existe des $a_{ij} \in [a_{ij}]$ et des $b_i \in [b_i]$ vérifiant les équations $\sum a_{ij}x_j = b_i$.[6]

Conception de nouveaux algorithmes

Une voie de recherche est l'amélioration de nos algorithmes de contraction.

Le travail sur `ACID` pour rendre l'opérateur `3B` plus adaptatif devrait finaliser les recherches en cours sur cet opérateur, du moins en ce qui concerne la contraction de boîtes extérieures. De nouvelles avancées pourraient provenir de :

[6] Ce problème reste fortement NP-difficile quand la matrice correspondante est régulière, ou quand les coefficients $[a_{ij}]$ et $[b_i]$ valent $[0, 0]$, $[1, 1]$ ou $[0, 1]$.

– nouvelles stratégies de bissection sensiblement différentes au sein d'un même processus de contraction "`3B`" ;
– une comparaison avec les algorithmes de type `Quad` [183] qui rognent les bornes des variables en optimisant une approximation convexe de l'ensemble du système ;

L'algorithme `Mohc` est plus récent et laisse à mon avis encore plus de marge d'amélioration. A court terme, il faut fournir une validation expérimentale poussée pour asseoir `Mohc` comme alternative à `HC4` et `Box`. Les tests de *profiling* qui apparaîtront dans la thèse de Ignacio Araya confirmeront la maturité de l'implantation actuelle, mais des améliorations plus ambitieuses sont envisageables.

Régler le paramètre τ_{mohc} de manière adaptative et/ou trouver automatiquement quand il est prometteur de déployer l'arsenal exploitant la monotonie d'une paire (contrainte, variable) semblent être des voies de recherche intéressantes. Comme mentionnée plus haut, une autre voie est la conception d'heuristiques de branchement favorisant les monotonies des fonctions. Enfin, certains résultats obtenus par Merlet avec ALIAS en utilisant les dérivées secondes des fonctions laissent également penser que trouver une meilleure évaluation des dérivées premières des fonctions permettrait de détecter plus de cas de monotonies dans `Mohc-Revise`.

La plupart des recherches concernant les méthodes à intervalles portent sur les algorithmes de contraction et les preuves d'existence d'une solution dans une boîte. En revanche, très peu de travaux portent sur la recherche arborescente. C'est très surprenant de la part de la communauté de PPC qui connaît les progrès spectaculaires obtenus dans la dernière décennie par les techniques de retours en arrière intelligents, d'apprentissage (des variables de branchement) et les heuristiques de choix de variables dans les domaines SAT notamment, mais aussi PLNE (programmation linéaire en nombres entiers), CSP, ou l'ordonnancement. A court terme, nous étudierons une variante de la fonction *smear* [167] reconnue comme étant la meilleure heuristique de branchement actuelle.

Finalement, les très bons résultats obtenus par `I-CSE` et différents résultats obtenus par Merlet avec ALIAS suggèrent un fort potentiel de la manipulation symbolique des équations (cf. la discussion sur le modèle de résolution plus haut) ou des coefficients de la matrice jacobienne.

Outils de résolution

Comme pour tous les problèmes combinatoires difficiles, il semble nécessaire de disposer d'un outil de résolution capable d'offrir plusieurs briques algorithmiques qu'il faut combiner de manière pertinente dans une stratégie de résolution adaptée à une instance donnée. La plupart des outils existants aujourd'hui sont des outils autonomes, disposant d'un langage propre (ex : Quimper, RealPaver) ou des bibliothèques d'un langage généraliste (ex : `Ibex` en `C++`). Deux autres types d'outils répondent à un besoin particulier.

Pour faciliter la prise en main des méthodes à intervalles par des ingénieurs, le choix d'un langage dédié au calcul scientifique comme Matlab ou Scilab fait du sens. L'outil GloptLab de Domes [69], IntLab de Rump [247] ou les premiers travaux de notre équipe sur l'outil `Int4Sci` (Intervals for Scilab) vont dans ce sens.

L'importance de la manipulation symbolique des contraintes pour les performances et les communautés d'utilisateurs existantes plaident également pour l'intégration des méthodes à intervalles dans un outil de calcul formel comme Mathematica ou Maple. Ces outils intègrent d'ailleurs déjà des couches basses d'arithmétique d'intervalles. Des premiers travaux de Granvilliers, Monfroy, Benhamou [113] ou sur `I-CSE` ont montré l'intérêt de la combinaison des deux approches. La version ALIAS-Maple de ALIAS [201] constitue probablement l'exemple le plus abouti d'intégration des deux types d'outils.

Le paysage des outils de résolution est aujourd'hui florissant (cf. section 2.9) et une fusion de certains de ces outils, les francophones notamment, ou au moins des échanges de bonnes procédures, seraient profitables au domaine.

Pour convaincre les industriels et motiver ou promouvoir des recherches académiques, il semble enfin pertinent de produire des solveurs dédiés à un champ "d'application" donné, comme l'optimisation globale, ou plus spécifiquement des domaines comme la localisation en robotique ou la chimie, comme mentionnés plus loin. En suivant l'idée de produits "métier" dans l'industrie, les équipes de recherche travailleraient à instancier leur outil générique (peu accessible à l'utilisateur final) pour produire des outils clefs en main, avec peu de poignées d'entrée pour l'utilisateur final.

Je voudrais conclure cette section par un souhait, qui peut aussi s'interpréter négativement comme une crainte. Il est difficile aujourd'hui d'avoir une comparaison entre les opérateurs de PPCI, entre ceux de programmation mathématique ou entre ceux d'analyse par intervalles. De même, aucun travail d'expérimentation poussé ne permet de comparer les contracteurs des différentes sous-communautés. Ne parlons pas de confrontations avec les outils de résolution de systèmes algébriques ou les méthodes de continuation. Les méthodes à intervalles ont tardé à montrer des performances intéressantes en pratique et il n'y a plus de temps à perdre pour promouvoir ce domaine de recherche. Certaines dérives de la publication scientifique et de la recherche en général fournissent une partie de l'explication, mais des domaines comme SAT, mais aussi l'optimisation globale (non fiable), avec l'organisation annuelle de compétitions de solveurs, montrent la voie à suivre. Sans aller jusqu'à l'organisation de telles compétitions sur un serveur dédié, le modèle des NCSP est suffisamment simple pour espérer connaître les performances des différents outils (avec un paramétrage réglé automatiquement), même si l'évaluation de la qualité des solutions et l'influence de la forme des équations sur les temps de résolution compliquent sensiblement la tâche. Des benchmarks existent déjà (page Web de COPRIN, COCONUT), mais il faut aller plus loin. Une confrontation scientifique saine doit avoir lieu pour passer à une autre échelle dans l'intérêt de l'industrie pour ce domaine.

3.6 Conclusion : l'essor de la programmation par contraintes sur intervalles

Nous concluons par différents arguments expliquant l'essor naissant des méthodes à intervalles dans plusieurs champs d'application, ainsi que l'essor des techniques de PPCI dans ces méthodes de résolution :

– De nouveaux *outils de résolution généralistes* basés sur les intervalles voient et verront le jour, contribuant ainsi à convaincre les ingénieurs de l'efficacité de l'approche. Certains outils sont autonomes, d'autres constituent des bibliothèques `C++`, d'autres sont intégrés à des outils d'analyse numérique (Matlab/Scilab), ou seront bientôt intégrés à des outils de calcul formel (Mathematica/Maple).
  Les *outils dédiés* à des classes de problèmes spécifiques devraient impacter plus vite encore le monde industriel.
– De plus, les *gains en performance déjà obtenus ou à venir* devraient permettre de traiter des problèmes plus importants ou plus difficiles (optimisation, avec quantificateurs, équations différentielles, etc).
– Enfin, les méthodes à intervalles sont *irremplaçables* pour modéliser et résoudre certains problèmes.

Les paragraphes suivants approfondissent les deux derniers arguments. Ajoutons comme argument peu technique la venue d'une nouvelle génération de jeunes chercheurs à forte et double-compétence en PPCI (informatique) et en analyse par intervalles (mathématiques), comme en France Alexandre Goldsztejn (CNRS, LINA) et Gilles Chabert (Ecole des Mines de Nantes).

Gains en performance déjà obtenus ou à venir

Nous avons souligné plus haut les gains déjà obtenus par les schémas de résolution basés sur les contracteurs de PPCI et Newton intervalles par rapport aux schémas précédents (tests d'existence + Newton). Notons que finalement peu d'outils exploitent à la fois des opérateurs de PPCI, des algorithmes de type Newton intervalles et des techniques de relaxation linéaire.

Le récent schéma d'optimisation globale robuste sous contraintes proposé par Rueher et al. [245] et intégré dans `Icos` montre des performances encourageantes, qui sont intermédiaires entre `GlobSol` de Kearfott [165] et `Baron` [250]. Il s'agit d'un point de départ incontournable pour tout travail de recherche en optimisation globale robuste. Les derniers travaux pour améliorer la borne supérieure menés notamment par A. Goldzstejn et L. Granvilliers (non encore publiés) et qui seront intégrés dans `RealPaver` [112] devraient pratiquement combler le manque de performance en optimisation globale des outils à intervalles par rapport à `Baron`.

Concernant les contributions en PPCI, rappelons l'intégration dans `Baron` de l'algorithme `2B`. Rappelons que certains algorithmes existants de PPCI ont été sous-estimés et sous-utilisés jusqu'à présent, notamment l'algorithme `3B`. De plus, des gains en performance sont à venir à court ou moyen terme grâce à de nouveaux contracteurs comme `MohC` (exploitant la monotonie des fonctions) et à la manipulation symbolique des contraintes ou de la matrice jacobienne. Ces arguments suggèrent que notre schéma de contraction (`I-CSE` suivi de `3BCID(Mohc)`) pourrait apporter des gains conséquents dans d'autres domaines connexes comme l'optimisation globale.

Enfin, des gains en performance sont à espérer grâce à des nouvelles heuristiques de choix de variables à bissecter et des parcours intelligents de l'arbre de recherche issus de la PPC.

D'un point de vue expérimental, la confrontation croissante (ou à venir) entre les différentes sous-communautés intervalles, et avec les approches algébriques, les méthodes de continuation et l'optimisation globale devraient faire progresser les méthodes à intervalles.

Champs d'application incontournables

Les méthodes à intervalles sont utilisées dans un nombre croissant d'applications dans des domaines variés comme la robotique, l'automatique, la chimie, l'imagerie. Comme je ne suis pas un spécialiste de ces applications, le lecteur pourra obtenir de multiples autres exemples que ceux décrits ci-après en consultant (les travaux de) Luc Jaulin et Jean-Pierre Merlet.

Les stratégies de recherche de *l'ensemble des solutions* sont d'abord cruciales dans certaines applications. Nous donnons trois exemples. Une application en chimie est proposée par Baharev et Rev concernant la recherche des états stables (et instables) des processus de distillation [16, 18, 17]. Calculer ces états stables de manière fiable est crucial dans la construction, la simulation et le contrôle des colonnes de distillation. Les systèmes correspondants comprennent des dizaines voire des centaines de contraintes non-linéaires. Si les méthodes algébriques semblent écartées à cause de la taille des systèmes traités, les méthodes à intervalles entrent en concurrence avec les méthodes de continuation.

Un deuxième exemple est le lancer de rayon (*ray tracing*) pour tracer sur un écran des surfaces implicites définies par les solutions d'une équation du type $f(x, y, z) = 0$. Sans détailler, l'approche revient à intersecter, pour chaque pixel dessiné, le rayon (allant de l'oeil au pixel dessiné) et la surface. En plaçant un repère sur le rayon, cela revient à trouver la plus petite racine positive (la plus proche de l'oeil) d'une fonction univariée. Les méthodes à intervalles commencent à être très compétitives pour cette recherche de racine, au point de parvenir à du temps réel en utilisant un processeur de type GPU [170].

Le troisième exemple peut se voir comme une version très simplifiée du problème traité dans mon équipe et mentionné à la section 2.5.1. Il s'agit de la détection de singularités pour un robot parallèle, où il faut garantir que le robot ne casse pas quand l'organe terminal atteint n'importe quel point dans une boîte donnée (espace de travail). Un moyen de résoudre ce problème consiste à s'assurer que le déterminant d'une certaine matrice J ne s'annule pas. A cette fin, on vérifie que le système $det(J(X)) = 0$ n'a pas de solution dans la boîte, ce qui demande une exploration exhaustive de l'espace de recherche.

Les méthodes à intervalles semblent être, avec les méthodes d'optimisation globale sous contraintes non-linéaires qui définissent l'espace de recherche par des polytopes, la *seule approche possible* pour prendre en compte des *erreurs de fabrication ou de mesure bornées*, ou pour ajouter des *quantificateurs* sur ces coefficients. On peut reprendre l'exemple précédent que l'on rend plus réaliste en considérant les erreurs sur les points de pose des jambes sur les plateformes (ou sur les longueurs de ces jambes). Un autre exemple est celui de la conception d'un robot qui doit être capable d'atteindre un espace de travail : pour toute perturbation sur les points d'attache et sur les mesures des jambes, existe-t-il un point atteignable (par l'organe terminal) dans un espace de travail donné ? Les travaux de Luc Jaulin et Eric Walter sur l'estimation robuste de paramètres en présence de mesures aberrantes (en nombre borné) constituent un autre exemple de problème difficile à aborder autrement que par les méthodes à intervalles [148].

De l'autre côté du spectre, les méthodes à intervalles, en l'occurrence la propagation de contraintes, sont parfois capables de résoudre des problèmes comprenant des milliers de contraintes. Un exemple est la détection de mines par un robot sous-marin et son auto-localisation (SLAM :

simultaneaous localization and map building) qui a été abordé par Luc Jaulin [145]. Le robot sous-marin est muni de plusieurs appareils de mesure sophistiqués et le problème est modélisé essentiellement par une équation d'état (différentielle ordinaire) dont la discrétisation produit des centaines de milliers de contraintes. Comme les appareils du sous-marin sont de grande qualité et produisent des erreurs de mesure faibles et garanties par le fabriquant, la résolution de ce problème est essentiellement assurée par l'algorithme `2B` (aucun point de choix n'est nécessaire). La propagation de contraintes intersecte en quelque sorte les différentes boîtes obtenues par ces multiples mesures au cours du temps.

Chapitre 4

Autres contributions

Ce chapitre résume mes contributions dans les autres domaines de recherche auxquels je me suis intéressé depuis la thèse : le modèle des contraintes fonctionnelles, la décomposition et la résolution des systèmes de contraintes (géométriques) et la recherche locale pour l'optimisation combinatoire.

4.1 Modèle des contraintes fonctionnelles

Ma thèse portait sur le maintien de solution de problèmes de contraintes appliqué à des systèmes interactifs sous contraintes, comme les interfaces graphiques ou les logiciels de dessin. L'utilisateur définit et place ses objets graphiques, mais aussi des contraintes que ces objets doivent respecter dynamiquement, en réponse aux interactions de l'utilisateur (déplacement d'objets ou ajout de contraintes). Ce problème pose deux difficultés majeures. D'abord, une difficulté technique liée à la *résolution*. Selon l'application, les contraintes peuvent être de natures diverses : des contraintes spatiales linéaires (ex : alignements, centrages), des contraintes non-linéaires (ex : distance entre objets), voire des contraintes non continues. La deuxième difficulté est sémantique, liée à la réponse attendue par l'utilisateur. En particulier, dans la plupart des logiciels interactifs, l'utilisateur attend un résultat unique à sa perturbation. La *prédictibilité de la prochaine solution* est une tâche ardue pour les modèles à base de contraintes car les systèmes de contraintes ont souvent plusieurs solutions possibles. Prenons l'exemple de deux fenêtres graphiques F_1 et F_2 qui doivent demeurer centrées verticalement en respectant une contrainte du type $x_1 + \frac{1}{2}l_1 = x_2 + \frac{1}{2}l_2$. En cas d'agrandissement de la fenêtre F_1, la deuxième pourrait s'agrandir à son tour pour rétablir la contrainte (modification de l_2) ou se déplacer (modification de x_2)...

Le modèle des contraintes dites *fonctionnelles* (en anglais : *dataflow constraints*) a été proposé pour traiter ce problème de maintien de solution. Ce modèle simple a été proposé par Ivan Sutherland en 1963, et intégré dans son célèbre outil Sketchpad [278], puis défendu notamment par Alan Borning et son équipe dans les années 1980 et 1990, en adoptant un nouveau type d'algorithme de résolution et en autorisant des contraintes flexibles (pondérées). Le modèle de base comprend un ensemble de variables (de domaines quelconques), un ensemble de contraintes

entre ces variables et un ensemble de *r-méthodes* (méthodes de résolution) qui sont des fonctions permettant de rétablir la cohérence d'une contrainte individuelle. Sur l'exemple de nos fenêtres centrées verticalement, la contrainte $x_1 + \frac{1}{2}l_1 = x_2 + \frac{1}{2}l_2$ est associée à quatre r-méthodes : m_1 : $x_1 \leftarrow x_2 + \frac{1}{2}l_2 - \frac{1}{2}l_1$; $m_2 : l_1 \leftarrow 2x_2 - 2x_1 + l_2$; $m_3 : x_2 \leftarrow x_1 + \frac{1}{2}l_1 - \frac{1}{2}l_2$; $m_4 : l_2 \leftarrow 2x_1 - 2x_2 + l_1$. Appliquer la r-méthode m_1 par exemple calcule une nouvelle valeur pour la *variable de sortie* x_1 après remplacement des *variables d'entrée* x_2, l_2, l_1 par leur valeur courante (à l'écran).

Ce modèle se nomme plus précisément modèle des contraintes fonctionnelles *multi-directionnelles*, car une contrainte donnée peut être associée à plusieurs r-méthodes possibles la résolvant dans "plusieurs directions". Si chaque contrainte possède une seule r-méthode possible (avec qui elle est confondue), on tombe alors dans le modèle plus simple des contraintes uni-directionnelles (*one-way*) qui est tout simplement le *modèle des tableurs*, une r-méthode étant une "formule Excel". Autre variante possible : autoriser des r-méthodes à plusieurs sorties, ce qui permet d'agréger plusieurs contraintes élémentaires en une contrainte équivalente et de lui associer des r-méthodes plus élaborées [253].

En cas de perturbation dans un système de contraintes uni-directionnelles, on retrouve la cohérence du système en effectuant un tri topologique des formules avant de les appliquer dans un des ordres totaux associés. En cas de perturbation dans un système de contraintes multi-directionnelles, une *phase de planification* doit d'abord sélectionner quelle r-méthode appliquer pour chaque contrainte (problème combinatoire), avant de procéder au tri topologique et à l'application des r-méthodes dans l'ordre obtenu.

Soulignons que les r-méthodes sélectionnées forment une structure de graphe orienté dans un graphe biparti (appelé *graphe de r-méthodes*) dont les nœuds sont les variables et les contraintes. Pour une contrainte c donnée, un arc relie c à sa variable de sortie, et des arcs relient les variables d'entrée à c. Trois types d'algorithmes existent pour mettre en œuvre la phase de planification :

- Un *couplage maximum* (entre les variables de sortie et les contraintes) permet de calculer un graphe de r-méthodes pouvant contenir des circuits [92]. Pendant la phase d'exécution, les contraintes contenues dans une composante fortement connexe (circuit) doivent alors être résolues simultanément par un résolveur annexe, une méthode numérique par exemple.
- La *propagation des degrés de liberté* (`PDOF`) est l'approche proposée par Sutherland [278] qui calcule un graphe de r-méthodes sans circuit (un `DAG`) des feuilles vers les racines (les perturbations).
- La *propagation des conflits* proposée par Borning [39, 85] calcule un graphe de r-méthodes à partir des perturbations.

Au début des années 1990, de nombreux logiciels interactifs non industriels (!) implantaient différents modèles de contraintes multi-directionnelles [281]. Certains acceptaient des r-méthodes à sortie simple, d'autres des r-méthodes à sorties multiples. Certains autorisaient seulement des contraintes requises, d'autres acceptaient aussi des contraintes flexibles. Certains construisaient des graphes de r-méthodes pouvant contenir des circuits, d'autres des graphes sans circuit. Enfin, certains outils étaient polynomiaux en temps, alors que d'autres étaient exponentiels.

A la fin des années 1990, le modèle des contraintes multi-directionnelles était pratiquement mort et enterré ! Que s'est-il passé ?

Trois raisons principales expliquent à mon avis la disparition de ce modèle : les mauvais choix algorithmiques faits par la communauté (à l'exception de Vander Zanden), l'incapacité du modèle à résoudre plusieurs contraintes simultanément et son incapacité à prédire la prochaine solution. Ma thèse n'a finalement pas eu d'autre intérêt que de fournir une autopsie de cette disparition, ce qui est une maigre consolation ! D'autres modèles ont survécu : des modèles plus simples limités aux contraintes linéaires, le modèle des tableurs, la programmation événementielle. La section suivante détaille ces points.

4.1.1 Principales contributions sur les contraintes multi-directionnelles

Une des principales contributions de la thèse [281] est de fournir une classification et une comparaison des différents algorithmes et problèmes traités. Une analyse fine des algorithmes existants, des résultats de complexité théorique des différents problèmes de planification de r-méthodes [284], et la conception de quelques algorithmes nouveaux ont notamment permis de montrer la supériorité du schéma `PDOF` sur les deux autres. Le résultat allait à l'encontre de la communauté installée qui utilisait essentiellement le schéma par propagation des conflits. Malgré son manque d'incrémentalité, `PDOF` peut être étendu tout en restant polynomial quand il traite des r-méthodes plus complexes : les r-méthodes à sorties multiples [295] ou les r-méthodes dites *générales*, introduites dans la thèse, qui permettent de résoudre un sous-système quelconque formé de plusieurs contraintes.

Deux extensions du schéma `PDOF` ont notamment été proposées pour prendre en compte les r-méthodes générales : `OpenPlan` (cf. [281], la dernière partie de [34], [156]) qui n'a jamais vraiment été appliqué, et surtout `GPDOF` [282] qui a été appliqué quelques années plus tard pour de la reconstruction de modèles 3D sous contraintes (cf. section 4.2.1 et chapitre G). `PDOF` calcule une séquence de r-méthodes de la dernière vers la première (en considérant l'ordre d'exécution). Sans détailler, `GPDOF` (*General Propagation of Degrees of Freedom*) suit le même schéma, mais peut sélectionner plusieurs fois une même r-méthode pour rester complet, c'est-à-dire pour garantir de trouver une séquence de r-méthodes, s'il en existe une. La terminaison est assurée par le fait qu'au moins une contrainte supplémentaire est sélectionnée à chaque itération. `GPDOF` reste un algorithme structurel qui joue avec le graphe de contraintes, les r-méthodes générales étant vues comme des hyper-arêtes qui se recouvrent partiellement entre elles...

La thèse a proposé également d'étendre le modèle des contraintes multi-directionnelles pour le rendre plus prédictif. Un algorithme exponentiel appelé `SemPlan` traite ce modèle. En 2001, avec Bertrand Neveu, nous avons identifié une sous-classe polynomiale de ce problème, soluble en temps linéaire par un algorithme appelé `AzureLink`. Le modèle propose d'ajouter aux variables, aux contraintes et aux r-méthodes un quatrième type d'entités appelées *règles de déclenchement de r-méthodes* qui introduisent le paradigme des événements dans le modèle des contraintes fonctionnelles. Il est évidemment hors de question de demander à l'utilisateur final de définir ces 4 types d'entités, mais plutôt de laisser le concepteur de l'application (un générateur d'interfaces graphiques, un logiciel de dessin, un traitement de texte du futur...) le soin de définir une bibliothèque contenant de telles contraintes "fonctionnelles avec règles de déclenchement". Une telle règle est un triplet. Sur notre mini-exemple, la règle (c, x_1, m_3) signifie qu'en cas de violation de la contrainte c de centrage des deux fenêtres graphiques par modification de x_1 (déplacement

de F_1), alors c est maintenue consistante par application de la r-méthode m_3 (déplacement de F_2), et non pas par retaillage de F_2 (trois autres règles complètent la description de cette contrainte) ! Comme le modèle des tableurs, toute perturbation du système entraîne le calcul d'au plus un seul plan (graphe de r-méthodes), tout en gardant l'aspect multi-directionnel (que le modèle des tableurs ne possède pas). De plus, ce modèle ne connaît pas les problèmes de terminaison ou de complexité exponentielle de la programmation événementielle.

Quoique disposant d'une description très détaillée de cet algorithme et des preuves de NP-complétude du problème plus général[1], ces travaux sont restés inappliqués. Pour convaincre de la pertinence de ce nouveau modèle, il faudrait développer une application graphique complète. Une tentative a eu lieu avec une jeune pousse américano-française N-generate qui cherchait à concevoir un logiciel permettant de vérifier la cohérence des chartes graphiques. Des échanges ont eu lieu avec cette société à qui j'ai exposé cette technologie et fourni un code Java implantant `AzureLink`, code déposé parallèlement à l'APP. A ma connaissance, cette société n'en n'a pas fait usage car absorbée par d'autres tâches plus prioritaires et semble avoir fermé ses portes en fin 2006 ! Je n'ai pas fait d'autres tentatives car absorbé par d'autres thématiques, notamment les contraintes géométriques rencontrées en CAO ou en reconstruction de scènes en 3D.

4.2 Décomposition et résolution de systèmes de contraintes (géométriques)

De nombreux problèmes peuvent être modélisés par de gros systèmes de d'équations non linéaires, voire non polynomiales, mais qui sont relativement peu denses (creux). Ces systèmes se prêtent à une décomposition du système global en sous-systèmes plus petits qui sont solubles plus facilement. Les solutions (partielles) des sous-systèmes sont alors combinées pour fournir les solutions globales. Un champ d'application qui semble particulièrement prometteur est celui des contraintes géométriques et de la CAO.

Les algorithmes `OpenPlan` et `GPDOF` proposés dans la thèse me semblaient prometteurs pour traiter ce type de problèmes. Une collaboration avec Marta Wilczkowiak a notamment permis d'appliquer `GPDOF` à un problème de reconstruction de scènes en 3D (cf. section 4.2.1 et chapitre G). Une autre contribution plus théorique s'est faite dans le cadre de la thèse de Christophe Jermann que j'ai co-encadrée (cf. section 4.2.2). Ces travaux m'ont rapproché pendant plusieurs années de la communauté travaillant sur les contraintes géométriques et des groupes de travail français "AS contraintes géométriques" et "Math-STIC solveurs géométriques" (CNRS) pilotés par les professeurs Dominique Michelucci (Dijon) et Pascal Schreck (Strasbourg). Ces travaux ont débouché aussi sur un *survey* très détaillé sur les différentes méthodes de décomposition de systèmes de contraintes géométriques, article publié dans une revue [156] et donné en annexe (cf. chapitre H). Résumer l'ensemble des travaux dans ce domaine doublerait la taille de ce mémoire (hors annexes), si bien que je me contente de souligner quelques contributions (plus) marquantes dans les sections qui suivent.

[1] Une première description a été présentée dans un *workshop* [285]. Une version plus claire, mais non publiée, est recevable sur demande.

4.2.1 Application de l'algorithme GPDOF à de la reconstruction de scène en 3D

Article dans la revue IJCGA [288], reproduit au chapitre G.
Auteurs : Gilles Trombettoni, Marta Wilczkowiak.

Pour avoir croisé Pierre Macé et sa doctorante Phil Massan-Kuzo [195] en 1998 à l'Ecole des Mines de Nantes, je pensais que l'algorithme GPDOF pouvait s'appliquer à un problème de reconstruction de scène en 3D à partir de photographies et de contraintes posées par l'utilisateur sur le modèle 3D. En 2002, Marta Wilczkowiak, en thèse à l'INRIA Rhône-Alpes sous la direction d'Edmond Boyer et de Peter Sturm, a contacté Christophe Jermann et moi-même pour ajouter des contraintes dans son application en vision par ordinateur. Après analyse, nous avons conclu que GPDOF semblait plus adapté au problème de Marta, qui était sous-contraint, que les algorithmes par rigidification récursive sur lesquels travaillait Christophe à cette époque (cf. section suivante). J'ai donc travaillé avec Marta pendant plusieurs mois, le plus souvent à distance, ce qui a produit un outil assez complet de reconstruction qui a obtenu de très bons résultats sur deux scènes représentant une place de Grenoble. Marta s'est occupée de toutes les briques de l'outil de reconstruction de modèle, y compris l'outil d'optimisation classique et une méthode d'étalonnage des caméras exploitant le sous-ensemble des contraintes linéaires. J'ai implanté GPDOF, l'algorithme ajoutant les r-méthodes dans la scène à partir d'un dictionnaire de r-méthodes génériques, et l'algorithme d'exécution du plan calculé par GPDOF basé sur un algorithme de backtracking.

Je ne détaille pas l'application en vision (cf. chapitre G), mais souligne simplement une caractéristique de notre outil : contrairement aux outils classiques, qui incorporent la violation des contraintes sous forme de pénalités dans la fonction objectif du processus d'optimisation numérique (Levenberg-Marquadt), notre approche permet de satisfaire les contraintes *exactement* (aux arrondis sur les flottants près), ce qui est important pour certaines applications architecturales où une erreur de parallélisme, même minime, se voit à l'œil nu.

Le système de contraintes du modèle est généralement sous-contraint car l'ensemble des contraintes fournies par l'utilisateur ne suffit pas à placer tous les objets de la scène architecturale construite. Une première partie des variables (que nous appelons paramètres) sont donc déterminées par l'optimisation classique et les autres variables sont déterminées par le plan de reconstruction. GPDOF s'est avéré utile à la fois pour calculer un plan de reconstruction (c'est-à-dire une séquence de r-méthodes à appliquer) et identifier un ensemble de paramètres. Après un appel à GPDOF, le processus d'optimisation entrelace une optimisation classique sur les paramètres et le recalcul des autres variables par exécution des r-méthodes. Plus précisément, chaque itération de l'optimisation (Levenberg-Marquadt) porte seulement sur les paramètres du modèle et est suivie par l'exécution de la séquence de r-méthodes qui recalcule une nouvelle valeur des variables satisfaisant l'ensemble des contraintes. Comme les r-méthodes satisfaisant des contraintes non-linéaires ont plusieurs solutions, une seule et "même" solution est choisie à chaque exécution.

Un prétraitement est effectué avant l'optimisation afin de déterminer quelle solution sélectionner systématiquement dans chaque sous-système/r-méthode de contraintes non-linéaires. Un algorithme de *branch and bound* fait un point de choix entre les différentes solutions de chaque

r-méthode non-linéaire (cf. section 4.2.3), *en commençant par la solution qui minimise les erreurs de reprojection.*

Les r-méthodes dans cette application sont des procédures de résolution qui implantent des méthodes à la "règle et au compas" ou d'autres théorèmes de géométrie. Par exemple, une des r-méthodes permet de calculer les 3 coordonnées d'un point A placé à distances données de 3 autres points B, C, D déjà placés. La r-méthode calcule les (au plus) 2 positions possibles du point A en intersectant les 3 sphères centrées resp. en B, C, D. Dans l'application, compte-tenu du jeu de contraintes défini (contraintes de distance, incidence, parallélisme, orthogonalité entre points, droites, plans), Marta a créé un dictionnaire contenant 60 r-méthodes génériques de ce type permettant de calculer (au plus) un objet en sortie en résolvant un sous-système d'au plus 3 équations. J'ai conçu un algorithme simple d'isomorphisme de sous-graphe pour reconnaître automatiquement toutes les occurrences des r-méthodes génériques du dictionnaire dans le système, et ajouter ainsi l'ensemble des r-méthodes réelles dont `GPDOF` a besoin pour calculer son plan. (Par exemple, le motif de r-méthode "intersection de 3 sphères" peut apparaître plusieurs fois dans une scène donnée.) En simplifiant l'algorithme de Avis [15], cet algorithme parcourt tous les sous-graphes connexes de taille au plus k (ici $k = 3$ contraintes) qui sont comparés aux entrées du dictionnaire avec une fonction de hachage spécifique.

Bilan

`GPDOF` s'est avéré très utile pour calculer une paramétrisation du modèle ainsi qu'un plan de recalcul. Les performances sont très bonnes grâce aux procédures câblées implantant les r-méthodes. Pour fixer les idées, le modèle le plus important contenait 452 équations et 819 variables. L'ensemble des étapes précédant l'optimisation (addition des r-méthodes par isomorphisme de sous-graphe, `GPDOF`, *branch and bound*) prend moins de 5 secondes alors que l'optimisation dure 467 secondes. L'optimisation exécute la séquence des r-méthodes, c'est-à-dire résout l'ensemble des contraintes, 7866 fois !

La limite la plus importante est que le système ne doit pas contenir d'équations redondantes car `GPDOF` est un algorithme structurel. Comme l'hypothèse d'un utilisateur qui ne se tromperait pas est irréaliste dans ce type d'application, nous avons introduit des règles simples d'ingéniérie permettant de les détecter au fur et à mesure de la définition des contraintes. Cette difficulté est un point sensible majeur pour un outil industriel qui se baserait sur ces travaux.

Soulignons que `GPDOF` est un algorithme structurel, mais que le dictionnaire contient des r-méthodes qui prennent en compte les paramètres et coefficients des équations et la géométrie des objets (c'est-à-dire ne se contentent pas d'un simple compte des degrés de liberté).

Enfin, l'outil proposé répond à un type de reconstruction bien spécifique pour lequel la qualité du modèle obtenu est grande, mais le temps de calcul également. La tendance de la communauté de vision par ordinateurs s'oriente plutôt vers des modèles produits en temps quasi-réel, mais de moindre qualité...

4.2.2 Degré de rigidité

Ces travaux sont issus de la thèse de Christophe Jermann que j'ai co-encadrée avec B. Neveu et M. Rueher de 1999 à 2002. Le contexte est la décomposition et la résolution de CSP géométriques, c'est-à-dire des systèmes de points, droites, plans soumis à des contraintes d'angle, distance, incidence, etc.

Il existe globalement quatre schémas algorithmiques pour décomposer un système de contraintes géométriques en sous-systèmes (cf. chapitre H). Les plus employés d'entre eux fonctionnent par *rigidification récursive*. L'étape clef de cet algorithme est l'identification d'un sous-système rigide. La caractérisation de la rigidité est un sujet très étudié dans des communautés aussi différentes que les graphes (rigidité combinatoire) [266, 302] et la théorie des mécanismes [117]. L'algorithme le plus connu [130, 269] utilise un algorithme de flot pour identifier un sous-système *structurellement rigide*. La rigidité structurelle est basée sur un compte des degrés de liberté du sous-système (et de toutes ses sous-parties...). Elle n'implique malheureusement pas la rigidité géométrique (dans le cas générique), sauf dans le cas de points en 2D soumis à des contraintes de distance, comme l'a montré Laman en 1970. C'est pourquoi l'outil utilise un ensemble d'exceptions pour traiter les cas où la rigidité structurelle se trompe.

Les travaux de Christophe Jermann ont permis d'identifier la cause du problème et de la corriger de manière générale et élégante, mais aussi coûteuse. La caractérisation structurelle de la rigidité ne traite en fait pas correctement les contraintes *booléennes/singulières*, c'est-à-dire les contraintes non euclidiennes comme les incidences et les parallélismes. Pour simplifier, un ensemble d'objets géométriques rigide possède 3 degrés de liberté en 2D et 6 degrés de liberté en 3D, mais peut en posséder moins s'il doit satisfaire des contraintes booléennes, qui introduisent en quelque sorte une singularité dans le système. Par exemple, un système en 2D composé de deux droites aura 2 degrés de liberté si celles-ci sont liées par une contrainte de parallélisme, au lieu de 3. Nous avons appelé ce nombre *degré de rigidité* et l'avons placé au cœur de l'algorithme de flot afin de le corriger et le simplifier.

Cela signifie malheureusement qu'il faudrait en quelque sorte résoudre le sous-ensemble de toutes les contraintes booléennes pour connaître le degré de rigidité à chaque étape de la rigidification récursive... qui sert elle-même à résoudre le système global ! Quoique `GPDOF` serait une piste intéressante pour l'identification du degré de rigidité (le sous-ensemble des contraintes projectives étant sous-contraint), cette voie n'a jamais été explorée de manière approfondie.

Ces travaux sur le degré de rigidité et la correction de l'algorithme de rigidification récursive ont notamment donné lieu à plusieurs publications à CP [157], à ADG [159, 150], à IJCAI [151] et dans la revue francophone JEDAI [154].

4.2.3 L'algorithme `Inter-Block Backtracking`

Travaux ayant donné lieu à plusieurs articles dans des congrès [34, 222, 221]. L'article le plus récent apparaît dans la revue Constraints [230]. Il est reproduit au chapitre I.

Cette section traite de la résolution d'un système préalablement décomposé en sous-systèmes par une technique de décomposition décrite ci-dessus, comme `GPDOF` ou la rigidication récursive.

Les techniques de décomposition produisent un ensemble de sous-systèmes bien-contraints (ou blocs). Un sous-système bien-contraint contient k équations indépendantes et k variables. Il possède un ensemble fini de solutions. Ces blocs forment en fait un graphe acyclique orienté (DAG). En effet, si un bloc i comprend une variable impliquée, en tant que paramètre, dans une équation d'un bloc j, il faut d'abord résoudre le bloc i, puis ensuite le bloc j. On choisit une des solutions du bloc i, dont les valeurs correspondantes constituent les paramètres des équations du bloc j. La phase de résolution traite donc les sous-systèmes en respectant l'ordre partiel imposé par le DAG. Pour garantir de trouver une solution globale si elle existe, elle effectue un point de choix sur les solutions (partielles) de chaque bloc et les combinent entre elles. Ce principe très général suppose l'existence d'une méthode capable de trouver l'ensemble des solutions d'un sous-système. L'application en reconstruction de modèles 3D utilisait des r-méthodes résolvant un sous-système avec une procédure câblée (du type "règle et compas") obtenue par manipulation symbolique préalable (manuelle) du système.

L'algorithme *Inter-Block Backtracking* (`IBB`) résout les blocs en utilisant une méthode à intervalles (cf. chapitres 2 et 3). Cet algorithme a notamment été étudié au cours des ans par Bertrand Neveu, qui en fait évoluer le code, et moi-même. Il a donné lieu à différentes évolutions coïncidant avec la collaboration initiale avec Christian Bliek en 1998, puis avec Christophe Jermann en 2003, et enfin avec Gilles Chabert en 2006 pour l'intégration avec `Ibex` [52, 50]. Les premiers travaux sur `IBB` en 1998 utilisaient, un peu comme une boîte noire (grise disons), la bibliothèque `ilcNum`, intégration de l'outil Numerica [293] dans Ilog Solver. Les derniers développements utilisent `Ibex` et notre expérience dans le domaine des intervalles. Je ne détaille pas les nombreuses versions de cet algorithme, mais souligne quelques tendances importantes.

Les premières versions adaptaient des algorithmes de retours en arrière entre blocs très sophistiqués pour exploiter la structure de DAG, comme l'algorithme *partial order backtracking* [198, 33] en 1998, et *Graph-based BackJumping* (GBJ) [65] en 2003. Les dernières versions se contentent d'un retour en arrière chronologique ! La seule exploitation du DAG qui demeure est une simple condition de non recalcul du bloc courant en cas de non changement des solutions des blocs parents.

Pour résoudre un bloc donné, les premières versions de `IBB` faisaient appel à un algorithme de `2B`. Les dernières versions utilisent en plus un Newton intervalles, qui a des conséquences importantes sur les autres choix algorithmiques, et parfois même un algorithme de `3B` pour les blocs importants.

Les premières expérimentations concluaient à l'inefficacité de propager les réductions de domaine d'un bloc à un autre (*Interblock filtering*), ce qui est moins clair dans les dernières versions.

(Ajoutons le fait que les dernières versions sont fiables, grâce à l'abandon d'une heuristique qui rendait le processus global incomplet en théorie.)

Pour résumer, plus nous avons augmenté le travail de filtrage/contraction des méthodes à intervalles, moins la sophistication de `IBB` s'est avérée importante. Cette tendance s'est confirmée dans l'algorithme `Box-k`, présenté à la section suivante, qui peut se voir comme une quatrième phase dans le développement de `IBB`.

`IBB` a été expérimenté sur une dizaine de systèmes décomposés par des techniques de couplage

maximum ou par rigidification récursive. Le plus gros d'entre eux, `Chair`, représente une chaise modélisée par 178 équations de distance, de parallélisme, etc. Il a été construit par Christophe Jermann et modifié par Bertrand (pour faire disparaître des singularités). Les performances obtenues par `IBB` sont très bonnes, avec des gains de un à plusieurs ordres de grandeur par rapport à une méthode à intervalles classique. Les dernières avancées sur les méthodes à intervalles (cf. sections 3.2 et 3.3) ont néanmoins réduit l'écart avec `IBB` de un ordre de grandeur environ.

Autre point positif : la réimplantation de `IBB` dans l'équipe de Laurent Zimmer à Dassault Aviation (ce qui fut une bonne surprise quand L. Zimmer nous l'a annoncé plusieurs années après la publication de 1998). Malheureusement, j'ai bien peur que `IBB` n'en soit resté au stade du prototype à Dassault Aviation.

`IBB` possède des défauts intrinsèques (expliquant son non passage en phase de production ?). Il est limité aux systèmes décomposés et ne présente aucun intérêt si le système est peu dense mais non entièrement décomposable. Il est ensuite dépendant d'algorithmes de décomposition dont nous avons mentionné le manque de robustesse dans les sections précédentes. L'algorithme suivant permet de dépasser ces limites.

L'algorithme `Box-k`

Travaux décrits dans un article du congrès CP [12]. L'article est reproduit au chapitre J.
Auteurs : Ignacio Araya, Gilles Trombettoni, Bertrand Neveu.

L'algorithme `Box-k` a été imaginé au départ comme un algorithme de contraction (intervalles) général pouvant traiter n'importe quel type de systèmes de contraintes. Il a montré pourtant des performances décevantes dans le cas général, mais au contraire très encourageantes quand il est utilisé pour des systèmes peu denses, que ceux-ci soient décomposables ou bien seulement structurés quoique irréductibles. La contribution semble donc être intéressante dans ces dernières applications, d'où son rattachement à cette partie du mémoire.

L'algorithme `Box-k` est un algorithme de propagation de contraintes sur intervalles assez classique dont l'originalité réside dans la procédure `Box-k-Revise`. Contrairement à la procédure `BoxNarrow` qu'elle généralise, à `HC4-Revise` ou à tout autre procédure de *révision* contractant la boîte vis à vis d'*une seule* contrainte (cf. section 2.4.3), la procédure `Box-k-Revise` travaille sur un ensemble de k contraintes. On peut ainsi voir le sous-système traité par cette procédure comme une *contrainte globale*. Un sous-système $S = (E, V)$ traité par `Box-k-Revise` satisfait les critères suivants :

- S forme un sous-graphe connexe, sans quoi on pourrait l'éclater en autant de sous-composantes connexes plus faciles à traiter.
- S contient un ensemble E de k équations et un ensemble V de k variables. Il s'agit donc d'un sous-système bien-contraint qui va apporter une contraction importante et bénéficier de l'algorithme de Newton intervalles (cf. section 2.4.1).

Notons que les autres variables impliquées dans une équation de E mais qui ne sont pas dans V sont considérées comme des paramètres, des coefficients dans les équations. Elles sont remplacées

par leur intervalle courant lors du traitement de S. La principale différence entre un bloc traité par `IBB` et un bloc traité par `Box-k-Revise` tient justement à ces paramètres. Dans `IBB`, ils sont quasiment réduits à des points (solutions des blocs parents), alors qu'ils peuvent être de taille arbitraire dans `Box-k-Revise` (en théorie).

Sans détailler, `Box-k-Revise` déroule un arbre de recherche (local) portant seulement sur les variables V et retourne l'enveloppe des feuilles obtenues, apportant ainsi une réduction des bornes des variables de V. Pour des raisons d'efficacité, l'arbre est développé en largeur d'abord, est de faible profondeur, et les feuilles obtenues sont préservées d'un appel de `Box-k-Revise` sur S au suivant. De plus, un paramètre utilisateur ρ_{io} permet de déclencher toute cette machinerie seulement si le volume des paramètres (en entrée) est suffisamment petit par rapport au volume des k variables (en sortie) du bloc traité.

Le processus global est le suivant. Au départ, des blocs de différentes tailles sont définis dans le système, soit automatiquement, soit manuellement, comme des contraintes globales. Ensuite, la recherche de solutions alterne classiquement des phases de propagation `Box-k` et des points de choix. `Box-k` utilise une propagation de type GAC pour propager les réductions obtenues dans les différents blocs. Une fois un point quasi-fixe atteint en termes de contraction, un point de choix est effectué. Deux types de points de choix sont envisagés, selon des mesures effectuées pendant la phase de propagation. S'il existe un sous-système dont le ratio de la somme du volume des feuilles de l'arbre local sur le volume de l'enveloppe retournée est inférieure à un seuil donné (par exemple 1%), alors un *point de choix multi-dimensionnel* (*multisplit*) est pratiqué : la boîte courante est remplacée par les feuilles (de l'arbre local) du bloc considéré. L'intuition est la suivante. L'arbre local a obtenu une forte contraction des différentes feuilles, mais l'enveloppe perd une grande partie de ce travail. Mieux vaut alors effectuer un vrai point de choix dans ce cas. Le *multisplit* est aussi une généralisation d'une étape de `IBB`, celle où une solution est trouvée dans un bloc et injectée dans les blocs en aval. Contrairement à `IBB`, un multisplit peut rendre un ensemble de boîtes non atomiques (non réduites à la précision des solutions).

L'algorithme `Box-k` a été implanté avec la bibliothèque en `C++ Ibex` (cf. section 3.4.3). (Soulignons la capacité d'`Ibex` - et de Ignacio Araya - à permettre de développer des algorithmes sophistiqués.) Les résultats obtenus par la stratégie `Box-k`+multisplit sont bons. Sur les problèmes décomposés traités par `IBB`, les blocs de la décomposition fournissent aussi les blocs traités par `Box-k-Revise`. La stratégie `Box-k`+multisplit obtient des résultats similaires à `IBB`, en perdant seulement un facteur 2 en moyenne par rapport à ce dernier, mais en gagnant toujours des ordres de grandeur par rapport à des stratégies à intervalles classiques. Sur 8 problèmes irréductibles trouvés sur la page de COPRIN [204] pour lesquels `IBB` n'apporte rien (le système étant décomposé en un seul bloc), `Box-k` parvient parfois à surpasser une stratégie basée sur `3B` et Newton intervalles.

La stratégie `Box-k`+multisplit est une version plus modulaire de `IBB` et est entièrement intervalles. Plus aucun ordre partiel entre les blocs n'est requis. Autoriser les circuits de blocs permet justement de faire apparaître des blocs bien-contraints dans un système irréductible. L'algorithme `IBB` est en fait remplacé par deux modules dans une recherche arborescente classique : une queue de propagation qui gère ses blocs et un algorithme de choix de branchement qui écoute si un bloc se porte volontaire pour un multisplit (en forçant le trait). Soulignons enfin (l'article

ne le mentionne pas) que `Box-k`+multisplit est bien plus robuste que `IBB`. Si un bloc est mal construit et contient une infinité de solutions, `Box-k-Revise` effectuera un travail inutile sur ce bloc mais limité en effort. Rien n'empêchera d'autres blocs d'apporter leur contraction.

La principale perspective de ce travail me semble être la sélection *automatique* de sous-systèmes "prometteurs" dans le système global. Nous pensons qu'une adaptation d'algorithmes de flot ou de graphes prenant en compte les critères de taille de blocs et ρ_{io} pourrait mener à des heuristiques efficaces.

4.3 Recherche locale

J'ai également fait quelques détours vers un domaine assez différent, celui de l'optimisation combinatoire, en travaillant sur des heuristiques, des méta-heuristiques et un problème de placement de rectangles en 2D (*strip packing*). Les motivations derrière ces travaux sont multiples, certaines sont avouables (initialement, le plaisir de relire un article [66] dans ce domaine bien formalisé sur l'algorithme "Go with the Winners" ; le plaisir enfantin de constater que "ça marche" ; le jeu d'intuitions et d'expérimentations permettant de faire les bons choix algorithmiques) et d'autres moins (la joie de voir Bertrand basculer des algorithmes génétiques vers la recherche locale ; la fierté d'échanger avec l'inventeur de la méthode taboue ; de beaux séjours au Chili). Tous les travaux décrits ci-dessous se sont faits en étroite collaboration avec Bertrand Neveu. Nous nous plaçons, sans perte de généralité, dans l'hypothèse de problèmes de *minimisation*.

La première contribution a été d'améliorer l'algorithme incomplet *Go With the Winners* (`GWW`). `GWW` est un algorithme à population qui baisse un seuil progressivement jusqu'à ce que toutes les *particules* (configurations, individus) ne parviennent plus à descendre vers de meilleures solutions. Entre chaque baisse du seuil, les différentes particules effectuent d'abord une marche aléatoire de longueur spécifiée. Les particules rattrapées par le seuil sont alors redistribuées sur les autres (les *winners*) [67, 68].

L'usage intensif de l'aléatoire a facilité les calculs théoriques effectués par les auteurs concernant la probabilité de trouver une solution ou la répartition des particules dans l'espace de recherche. Il a en revanche des effets négatifs sur les performances car la méthode manque d'intensification (capacité à descendre vers les solutions de meilleurs coûts). Nous avons proposé une amélioration de `GWW` pour le rendre plus compétitif. D'abord, le seuil est baissé de manière plus sophistiquée, de manière à pouvoir s'adapter à des paysages très chahutés. Ensuite, la marche aléatoire est remplacée par une technique de recherche locale, produisant le schéma `GWW-LS` (`GWW` *with Local Search*). De bons résultats ont été obtenus sur des instances difficiles de coloriage de graphes et d'affectation de fréquences (CELAR). Deux articles décrivent `GWW-LS`, l'instance `GWW-idw` retenue pour nos expérimentations et les expérimentations elles-mêmes [224, 225].

`GWW-idw` possède néanmoins 4 paramètres : le nombre de particules, un paramètre utilisé pour la gestion du seuil, la longueur des marches, un nombre de voisins utilisé par la recherche locale. Comme cela faisait trop pour notre compréhension, nous avons sorti la méta-heuristique `ID walk` utilisée par `GWW-idw` pour nous consacrer par la suite à son étude de manière indépendante. `ID Walk` s'avère être une métaheuristique aussi simple qu'efficace et constitue probablement la

contribution la plus importante de ces travaux (cf. section suivante).

Pour tester les différents algorithmes, Bertrand Neveu a développé une bibliothèque en `C++`, appelée INCOP, incluant les principales métaheuristiques connues (méthode taboue, recuit simulé, etc), `GWW-LS` et `ID walk`. J'ai simplement participé au choix d'architecture de la bibliothèque [231]. Cette bibliothèque sert notamment à nos développements internes en optimisation combinatoire. Elle a été néanmoins utilisée par Christine Solnon pour comparer les métaheuristiques classiques à ses algorithmes de fourmis [271]. Elle est également intégrée dans les outils `ToolBar` et `ToulBar2` maintenus par Simon de Givry. `ToolBar` et `ToulBar2` sont des outils d'optimisation complète spécialisés dans les CSP pondérés (WCSP) [252]. INCOP a été par exemple utilisé pour trouver un majorant initial lors de la compétition de Max-CSP en 2007 [42].

4.3.1 ID Walk

Article publié dans les actes du congrès CP [231] et reproduit au chapitre K.
Auteurs : Bertrand Neveu, Gilles Trombettoni, Fred Glover.

Le principe de la *recherche locale* consiste à chercher la meilleure solution d'un problème d'optimisation combinatoire en effectuant une *marche*, c'est-à-dire une suite de *mouvements* élémentaires qui modifient la *configuration* courante. L'ensemble des mouvements possibles à partir de la configuration courante est appelé *voisinage* [99]. A la fin de la marche, on retourne la configuration de coût le plus bas trouvée en chemin. La recherche locale est une approche incomplète en ce qu'elle est incapable de garantir dans le cas général d'avoir trouvé une des configurations de meilleur coût. La définition du voisinage dépend de la modélisation du problème. Par exemple, pour une formule booléenne dont il faut trouver un modèle (solution), une configuration c correspond à une affectation de toutes les variables booléennes à la valeur 0 ou 1 et possède un coût égal au nombre de clauses violées. Un voisin c' de c est la configuration c dans laquelle on prend la négation d'*une seule* variable booléenne. Dans une autre définition de voisinage plus *intensifiant*, on peut choisir un voisin c' où l'on modifie une seule variable, mais telle que le nombre de clauses violées diminue ou reste égal à celui de c. Si au cours de la marche, on passe par une configuration de coût nul, alors on a trouvé une solution ; dans le cas contraire, on ne peut rien conclure. Les *métaheuristiques* sont des algorithmes applicables à tout problème combinatoire et qui recherchent toutes à satisfaire deux critères importants : l'*intensification* pour descendre prioritairement vers les meilleures configurations et la *diversification* pour ne pas rester confiné dans une seule région de l'espace de recherche.

`ID Walk` est une métaheuristique basée sur la recherche locale. Elle est une alternative aux métaheuristiques classiques que sont la méthode taboue [99] et le recuit simulé [169]. Tout comme ses célèbres compétititrices, `ID Walk` possède peu de paramètres. En plus de la longueur de la marche qui est un paramètre déterminant plutôt le temps de calcul (sauf en cas de

redémarrage/*restart*), `ID Walk` possède un ou deux paramètres. Dans sa version la plus générale, disponible dans INCOP et décrite dans un résumé à ROADEF [232], les deux paramètres `MaxNeighbors` et `SpareNeighbors` (`SpareNeighbors` $\leq$ `MaxNeighbors`) sont des nombres de voisins candidats, d'où son appartenance à la catégorie dite des stratégies basées sur la liste des candidats (*candidate list strategies*). En utilisant ses paramètres, `ID Walk` effectue un mouvement d'une configuration x vers une configuration x' de la manière suivante :

1. `ID Walk` pioche aléatoirement les voisins candidats un par un et évalue leur coût. Le *premier* voisin x' avec un coût meilleur ou égal à celui de x est accepté (c'est-à-dire qu'un mouvemement sera effectué de x vers x').
2. Si `MaxNeighbors` candidats ont été rejetés lors de cette première boucle, le moins mauvais des `SpareNeighbors` premiers voisins examinés est néanmoins sélectionné (`SpareNeighbors` $\leq$ `MaxNeighbors`).

Deux "instances" immédiates de `ID Walk` sont décrites dans nos articles. Si `SpareNeighbors` vaut `MaxNeighbors`, la variante appelée `ID(best)`, en cas d'épuisement des `MaxNeighbors` voisins piochés, retourne le *meilleur* d'entre eux (ou plutôt le moins mauvais). Si au contraire `SpareNeighbors` vaut 1, la variante appelée `ID(any)` retourne *n'importe lequel* d'entre eux.

`ID(best)` a un point commun avec une des (multiples) variantes de la méthode taboue. Il s'agit de l'étape 2 ci-dessus[2]. Cependant, les différences sont l'absence de liste taboue et l'étape 1 ci-dessus.

Sans détailler, `MaxNeighbors` offre un compromis entre effort d'intensification et diversification. `SpareNeighbors` règle un degré de diversification. Nous avons médité à bien d'autres variantes, comprenant des paramètres spécialisés chacun dans la tâche d'intensification ou de diversification, sans grand succès en pratique. Nous avons également cherché à rendre `ID Walk` adaptatif, en vain !

Faute de grives, la procédure de *réglage automatique de paramètres* disponible dans INCOP se comporte très bien sur `ID Walk`. Cette procédure est utilisée dans une stratégie (ou méta-métaheuristique) de recherche locale qui prend en paramètre une métaheuristique et rien d'autre. Cette recherche locale enchaîne des sous-marches de taille croissante effectuées par la métaheuristique avec des valeurs de paramètres fixés. Chaque sous-marche est précédée d'une phase de réglage automatique qui balaie l'espace des valeurs des paramètres sur un nombre réduit de mouvements.

La procédure de réglage des paramètres est très robuste sur `ID(best)` et `ID(any)` car un seul paramètre doit être réglé et car le balayage dichotomique des valeurs des paramètres ne peut généralement pas diverger. En effet, une valeur trop faible de `MaxNeighbors` empêche `ID Walk` d'intensifier sa recherche et produit une mauvaise valeur de l'objectif. De même pour une valeur trop forte de `MaxNeighbors` qui ne diversifie pas assez sa recherche. La procédure de réglage appliquée à `ID Walk` (à deux paramètres) est relativement robuste. La part du réglage dans le temps total de la méta-métaheuristique est de l'ordre de 20% à 60% du temps selon l'instance traitée, la métaheuristique utilisée, le nombre de paramètres à régler (1 ou 2).

[2]Des tests sur plusieurs instances suggèrent d'ailleurs que ce paramètre a plus d'impact sur les performances que la liste taboue !

Indépendamment de la procédure de réglage automatique, retenons un avantage très important de `ID Walk` (par rapport à une méthode utilisant une liste taboue par exemple) : son paramètre `MaxNeighbors` a un impact très important sur les performances et se règle très bien !

`ID Walk` et ses deux instances ont été appliqués avec succès à de nombreux problèmes comme le CSP (instances générées aléatoirement), le coloriage de graphe, l'allocation de fréquences (CELAR), la variante du carré latin spatialement équilibré ou l'ordonnancement de voitures sur une chaîne de montage (*car sequencing*). C'est également `ID(any)` (désigné par `IDWa` dans l'outil INCOP) qui est souvent utilisé dans ToolBar pour calculer un premier majorant [42]. `ID(best)` a enfin été utilisé plus récemment pour traiter une variante d'un problème de placement de rectangles.

4.3.2 Problème du *Strip Packing*

Ces travaux ont été décrits dans différents articles [227, 229, 226]. L'article paru dans la revue IJAIT [229] est reproduit au chapitre L. L'article [226], reproduit au chapitre M, résume nos dernières avancées sur le sujet.
Auteurs : Bertrand Neveu, Gilles Trombettoni, Ignacio Araya, Maria-Cristina Riff.

Ces recherches ont été menées dans le cadre d'une coopération avec le Chili (accords entre CONICYT et l'INRIA), en collaboration avec Maria-Cristina Riff. Tout a commencé par le choix d'un problème de placement de rectangles, le *Strip Packing* qui est une variante du célèbre *Bin Packing* en 2D, variante un peu moins étudiée que ce dernier.

Le problème consiste à placer, sans recouvrement, des rectangles de hauteur et largeur données sur une bande rectangulaire (*strip*) de largeur donnée et de hauteur infinie. Le but consiste à minimiser la hauteur de bande utilisée. Ce problème est parfois posé comme un problème de découpe. Il en existe plusieurs variantes, notamment selon si les opérations de découpe doivent ou non se faire sur toute la largeur de la bande (coupe guillotine) ou si l'on peut tourner ou non les rectangles d'un angle de 90 degrés. Nous avons étudié le problème de base et sa variante autorisant les rotations de rectangles.

Pour résoudre ce problème, nous avons proposé un algorithme incomplet *sans aucun paramètre utilisateur*, divisé en deux phases :

1. une heuristique gloutonne produit un premier placement des rectangles sur la bande ;
2. la variante `ID(best)` de `ID Walk`, dont le paramètre est réglé automatiquement (cf. section 4.3.1), répare cette première solution.

Sur ce problème, `ID(best)`, `ID Walk` (à deux paramètres) et la méthode taboue implantées dans INCOP produisaient à peu près les mêmes résultats, sensiblement devant le recuit simulé. `ID(best)` a été choisie pour sa simplicité et la facilité à régler automatiquement son unique paramètre.

Nous avions conscience que, de manière générale, la mise en œuvre d'un mouvement était une étape atomique clef dans les algorithmes de recherche locale. Or, l'efficacité d'un mouvement repose directement sur l'incrémentalité de la mise à jour des structures de données utilisées pour

stocker la configuration courante. C'est pourquoi un important effort a été fourni pour trouver un mouvement incrémental. Notre première idée a été de chercher un mouvement prenant en compte la géométrie 2D du problème (contrairement à d'autres approches plus naïves). Cela a produit le mouvement illustré à la figure 4.1.

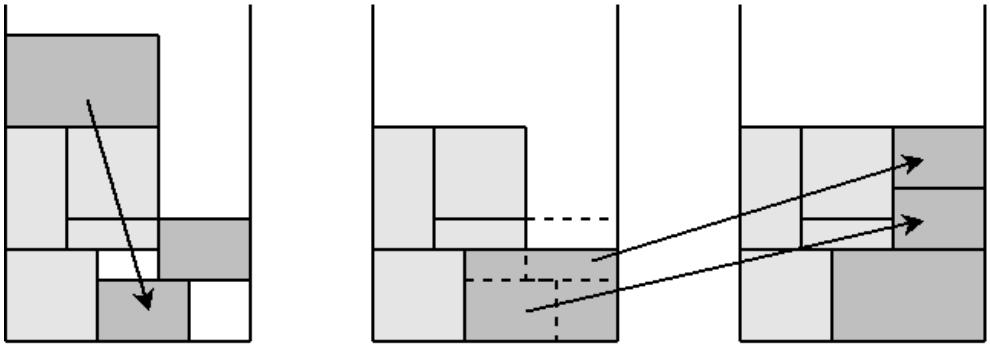


FIG. 4.1 – Un mouvement de notre algorithme incomplet.

Un rectangle R en haut du placement est enlevé et replacé au milieu du placement (figure de gauche). Les rectangles qui intersectent alors R à sa nouvelle position sont enlevés (figure du milieu) et replacés sur la bande avec une heuristique gloutonne (figure de droite).

Il faut comprendre que la plupart des algorithmes de placement de rectangles, complets comme incomplets d'ailleurs, maintiennent une propriété dite *Bas-Gauche* (BG ; en anglais : *Bottom-left*) au cours de la recherche, c'est-à-dire un placement où les segments bas et gauche de chaque rectangle touchent le conteneur ou un autre rectangle. La propriété BG limite l'explosion combinatoire en limitant le nombre de positions possibles pour les rectangles à placer. On peut en fait montrer que toute solution peut se transformer en une solution BG (de coût meilleur ou équivalent) en appliquant une simple procédure de compactage. Maintenir la propriété BG est simple lors d'un ajout de rectangle, mais s'avère plus compliqué lors d'un retrait qui peut impacter toute la solution courante.

La principale contribution de notre travail a été de proposer une approche alternative qui *ne respecte pas* la propriété BG au cours des mouvements, mais qui maintient plutôt l'ensemble des *trous maximaux*, c'est-à-dire l'ensemble des zones rectangulaires maximales (au sens ensembliste) où l'on va pouvoir placer un rectangle. Le nombre total de trous maximaux est en $O(n^2)$, où n est le nombre de rectangles placés dans le conteneur. Ce nombre est linéaire en pratique comme l'ont montré nos expérimentations. Les opérations d'ajout et de retrait de rectangles deviennent alors incrémentales, ce qui nous a permis de rendre efficace la mise en œuvre du mouvement illustré à la figure 4.1. Notons que cette structure astucieuse est utilisable par tout algorithme de placement de rectangles.

Plus récemment, nous avons amélioré notre algorithme en appelant occasionnellement une procédure de compactage (d'intérêt sensible mais non crucial) et surtout en employant un nouvel algorithme glouton pour le calcul de la solution initiale comme pour le replacement des rectangles au cours d'un mouvement.

Cet algorithme, appelé *Randomized Best Fit*, (`RBF`) est une variante aussi triviale qu'efficace de la très reconnue heuristique gloutonne *Best Fit decreasing*. Plutôt que de décrire formellement ces deux heuristiques, donnons-en une idée (un peu simplificatrice) en 3D sur le problème plus connu du *rangement des valises dans un coffre d'une voiture avant un départ en vacances* (problème très reconnu par la communauté des hommes martyrisés par leur femme) ! L'heuristique classique *Best Fit* est paramétrée par un critère de "taille" pour classer l'ensemble des valises, par exemple le *volume* ou la *hauteur*. En partant d'un coin prédéfini du coffre, elle préconise de remplir itérativement l'emplacement "le plus près" du coin avec la première valise qui peut y rentrer, par ordre de taille décroissante. Quand plusieurs critères de taille semblent donner de bons résultats pour une instance donnée, une approche courante est de lancer *Best Fit* sur des essais avec les différents critères, ce qui offre une plus grande diversité des solutions. Néanmoins, pour un essai donné, tous les objets sont placés avec *le même* critère.

Contrairement à *Best Fit*, un essai de l'heuristique `RBF` utilise *plusieurs* critères de taille. Le critère est simplement pioché aléatoirement à chaque nouvelle valise placée : le mari demande à sa femme de tirer à pile ou face entre *plus haute* et *plus volumineuse* pour savoir sur quel critère il choisira la prochaine valise à placer.

Au final, notre algorithme de placement sans paramètre utilisateur est basé sur `ID Walk`, un mouvement très étudié et l'heuristique gloutonne `RBF`. Sur de nombreuses instances testées, il est très compétitif avec les meilleures approches incomplètes connues. Notre approche est très générale et pourrait être encore améliorée en trouvant une heuristique gloutonne encore plus efficace (quitte à investir plus d'effort). Une piste est évoquée dans un des articles en annexe.

4.3.3 Bilan sur les méthodes incomplètes d'optimisation combinatoire

Nos différents travaux de recherche sur les méthodes incomplètes d'optimisation combinatoire nous ont donné un sentiment sur quelques types d'approches assez différentes : la recherche locale, GRASP, les algorithmes génétiques/évolutionnaires, GWW. Nous avons obtenu de nombreux résultats positifs avec la recherche locale et ses mécanismes d'intensification et de diversification (parfois avec redémarrage). Quand des heuristiques gloutonnes efficaces sont connues pour un problème donné (comme les problèmes de placement de rectangles ou l'ordonnancement de voitures), les méthodes de type GRASP (*greedy randomized adaptive search procedure*) me semblent aussi très pertinentes [82].

4.4 Conclusion : le désir d'intervalles

Hormis les travaux récréatifs sur les méthodes incomplètes d'optimisation combinatoire, un fil conducteur existe entre mes trois autres domaines de recherche : le modèle des contraintes fonctionnelles, les techniques de décomposition et de résolution de systèmes de contraintes géométriques, les méthodes à intervalles.

Les limites du modèle des contraintes fonctionnelles établies pendant la thèse m'ont fait naturellement chercher un domaine plus prometteur. Le concept des r-méthodes générales ca-

pables de résoudre un sous-système de contraintes et applicables à un domaine déterminé comme les contraintes géométriques m'ont orienté vers la CAO, à condition d'envisager deux généralisations. Les algorithmes de planification pour les contraintes fonctionnelles deviennent des algorithmes de décomposition d'un système de contraintes en sous-systèmes. L'exécution d'une séquence de r-méthodes qui calculent une solution partielle unique devient un algorithme de résolution combinant les solutions partielles des différents sous-systèmes, comme `IBB` ou l'algorithme de *branch and bound* utilisé dans l'application de `GPDOF` en reconstruction de scène.

Pourtant, demeurent des écueils majeurs. Si l'algorithme de décomposition identifie un sous-sytème de manière géométrique par une règle ou un motif, l'approche devient coûteuse ou incomplète (`GPDOF` est efficace mais son dictionnaire de motifs de r-méthodes générales est nécessairement incomplet). Si l'identification d'un sous-système se fait de manière structurelle par un décompte des degrés de liberté (par un algorithme de flots ou de graphe) ignorant la nature des objets et les paramètres, de nombreux problèmes liés à des singularités ou des contraintes redondantes peuvent entraîner l'échec de la phase de résolution. Or, les travaux sur le degré de rigidité suggèrent que combler l'écart entre rigidité structurelle et rigidité géométrique est un problème difficile. Les travaux de Dominique Michelucci sur la méthode de la configuration témoin (cf. section H.9) buttent sur le même écueil. La méthode probabiliste pour tester la rigidité demande comme prérequis de connaître les contraintes booléennes/singulières, directes ou induites, d'incidence, de parallélisme, etc, qui relient les objets du système. L'algorithme `GPDOF` muni d'un dictionnaire contenant des théorèmes de géométrie et lancé sur le sous-ensemble des contraintes booléennes me semble être une approche intéressante (au moins pour les théorèmes de taille finie).

Les grands risques d'erreur de toutes ces méthodes de décomposition et l'écart entre rigidité structurelle et géométrique font pourtant douter de la pertinence de l'approche. Les méthodes à intervalles abordent un problème exprimé dans un langage beaucoup plus simple (des variables réelles dont le domaine est un intervalle borné par deux nombres flottants, et des équations ou des inégalités non linéaires) mais ignorent toute propriété par exemple géométrique qui permettrait de mieux résoudre le système. Cependant, les progrès accomplis sur d'autres modèles simples d'expression (PLNE, SAT et à moindre titre CSP) laissent présager que des algorithmes polynomiaux de réduction de l'espace de recherche et des stratégies de recherche pourraient capter les structures cachées d'une instance donnée et faire gagner des ordres de grandeur par rapport aux méthodes à intervalles actuelles.

Annexe A

Exploiting Common Subexpressions in Numerical CSPs

Article [9] : paru au congrès CP, Constraint Programming, en 2008

Auteurs : Ignacio Araya, Bertrand Neveu, Gilles Trombettoni

Abstract

It is acknowledged that the symbolic form of the equations is crucial for interval-based solving techniques to efficiently handle systems of equations over the reals. However, only a few automatic transformations of the system have been proposed so far. Vu, Schichl, Sam-Haroud, Neumaier have exploited common subexpressions by transforming the equation system into a unique directed acyclic graph. They claim that the impact of common subexpressions elimination on the gain in CPU time would be only due to a reduction in the number of operations.

This paper brings two main contributions. First, we prove theoretically and experimentally that, due to interval arithmetics, exploiting certain common subexpressions might also bring additional filtering/contraction during propagation. Second, based on a better exploitation of n-ary plus and times operators, we propose a new algorithm I-CSE that identifies and exploits *all* the "useful" common subexpressions. We show on a sample of benchmarks that I-CSE detects more useful common subexpressions than traditional approaches and leads generally to significant gains in performance, of sometimes several orders of magnitude.

A.1 Introduction

Granvilliers et al. [113] show in a survey several ways to combine symbolic and interval methods to improve performance of solvers. They noticed that Gröbner basis computation [45] introduces redundancies that often improve the pruning effect of interval techniques. The use of several forms of the equations together in the same system (e.g., the natural and centered forms) has the same effect.

The presence of multiple occurrences of the same variable in a given equation is well-known to lower the power of interval arithmetics [219]. Thus, several practitioners apply by hand symbolic transformations of their systems, such as factorizations, to limit the number of occurrences of variables [203, 113].

Common subexpression elimination (CSE) is an important feature of compiler optimization [216]. CSE searches in the code for common subexpressions with identical evaluation and replaces them by auxiliary variables. It generally fasten the program by decreasing the number of instructions. Symbolic tools like Mathematica [43] or Maple [122] represent equations by directed acyclic graphs (DAGs), where nodes with several parents correspond to *common subexpressions* (CSs). This decreases the number of evaluations and also stores all the expressions with less memory. Ceberio and Granvilliers in [48] use Gaussian elimination to reduce the number of non-linear terms in equations. As a side effect, their algorithm identifies some CSs.

Following the representation used by symbolic tools, Schichl and Neumaier have proposed a unique DAG to represent a system of equations handled by interval analysis techniques [254]. CSs are the nodes of the DAG with several parents and the main interval analysis operators are redefined on this data structure : evaluation of functions, computation of derivatives, etc. Vu, Schichl and Sam-Haroud have described in [299] how to carry out propagation in the DAG. In particular, an interval is attached to internal nodes and the propagation is performed in a sophisticated way : two queues are managed, one for the evaluation, the other for the narrowing/propagation (see below), and the top-down narrowing operations have priority over the bottom-up evaluation. All the researchers who have exploited CSs manually or automatically [113, 299] think that the gain in performance due to common subexpressions would be only implied by a reduction of the number of operations.

The first good news is that CSE in interval analysis might bring *a stronger contraction/filtering power*. Section A.3 clearly states which types of CSs are useful for bringing additional filtering. Section A.4 presents a new algorithm `I-CSE` (Interval CSE) to detect CSs and generate a new system of equations. For a given form of the equations, `I-CSE` is able to find *all* the "useful" CSs, because it finds all the n-ary maximal CSs corresponding to sums and products, modulo the commutativity and the associativity of these operators, including overlapping CSs. In addition, `I-CSE` is not intrusive in that it produces a new system that can be handled by any interval solver using a classical propagation scheme. Finally, experiments shown in Section A.6 highlight that the CSs are extracted very quickly. The new system of equations then leads solving algorithms using `HC4` to significant gains in performance (of sometimes several orders of magnitude).

A.2 Background

The algorithms presented in this paper aim at solving systems of equations or, more generally, numerical CSPs.

Definition 2 *A* **numerical CSP** *(NCSP)* $P = (V, C, B)$ *contains a set of constraints* C *and a set* V *of* n *variables. Every variable* $x_i \in V$ *can take a real value in the interval* X_i *and* $\mathbf{B}$ *is the cartesian product (called a* **box***)* $X_1 \times ... \times X_n$*. A solution to* P *is an assignment of the variables in* V *satisfying all constraints in* C*.*

Finding all the solutions to an NCSP follows a scheme analogous to branch and prune for CSPs. *Branch :* Bisections divide the domain of one variable into two sub-domains in a combinatorial way. *Prune :* Two types of algorithms are used. Algorithms from interval analysis, like interval Newton [219], contract/filter the current box in all the dimensions simultaneously and can often guarantee that a box contains a unique solution. Algorithms from constraint programming are also useful. `HC4` follows a propagation loop like that of AC3 and handles the constraints individually with a procedure `HC4-revise` that removes inconsistent values on the bounds of intervals [28, 182]. Stronger consistencies like 3B [186], similar to SAC [64] for finite domain CSPs, often obtain a better performance. At the end, the solving algorithm finds an approximation of all the solutions of the NCSP. The algorithm `HC4-revise` uses a tree representation of one constraint, where leaves are constants or variables, and internal nodes correspond to *primitive operators* like `+`, $\times$, `sinus`. An interval is associated with every node. `HC4-revise` works in two phases. The *evaluation* phase is performed bottom-up from the leaves (variables and constants) to the root. Using the *natural extension* of primitive functions, this phase evaluates the intervals of the sub-expressions represented by the tree nodes (see Fig. A.1-left). The *narrowing* phase traverses the tree top-down from root to leaves and applies in every node a narrowing operator (also called projection ; see Fig. A.1-right). The narrowing operator contracts the intervals of the nodes eliminating inconsistent values w.r.t. the corresponding unary or binary primitive operator. In Fig. A.1, the intervals in bold have been narrowed. If an empty interval is obtained during the narrowing phase, this means that the constraint is inconsistent w.r.t. the initial domains. The intervals computed in the internal nodes are not stored from one call to `HC4-revise` to another, as opposed to the intervals of the leaves (i.e., the variables).

A.3 Properties of HC4 and CSE

We call *Common Subexpression* (in short CS) a numerical expression that occurs several times in one or several constraints.

If we observe carefully the `HC4-revise` algorithm, we can note that the contraction obtained by a narrowing operator on a given expression f is in general partially lost in the next evaluation of f. Consider for instance a sum $x + z$ that is shared by two expressions n_1 and n_2. Following Fig. A.1, the narrowing phase of `HC4-revise` applied to n_1 contracts its interval to $[-2, 5]$. Then, when the evaluation phase of `HC4-revise` applies to n_2, its interval is set to $[-2, 6]$ (Fig. A.2–left). Clearly, the interval of n_2 is larger than that of n_1. To avoid this loss of information, the

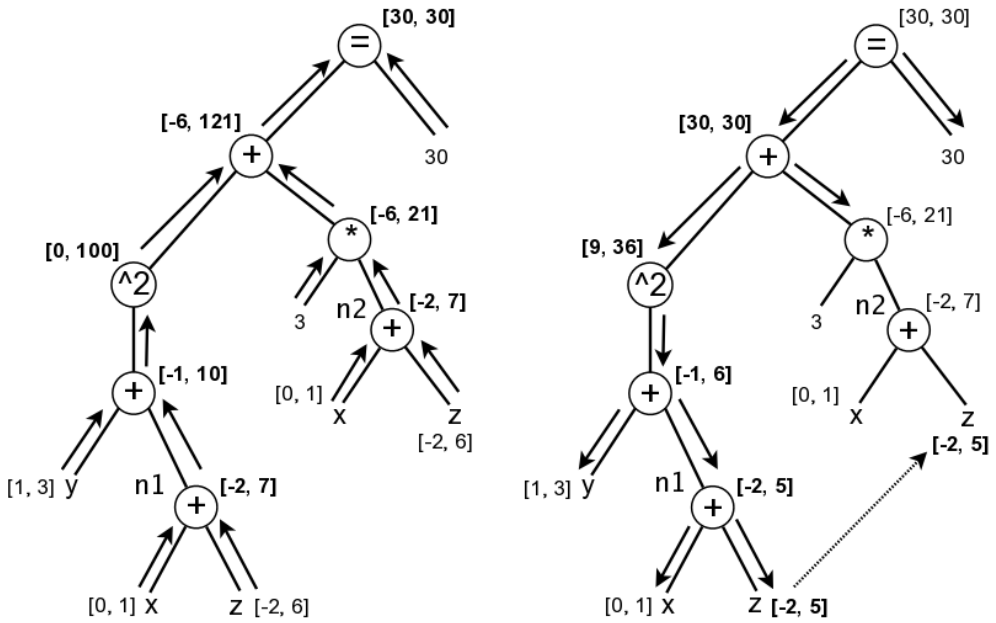


FIG. A.1 – Evaluation and narrowing in the `HC4-revise` algorithm. The tree represents the constraint : $(x + y + z)^2 + 3(x + z) = 30$.

idea is to replace n_1 and n_2 by a common variable v, and to add a new constraint $v = x + z$. The new system is equivalent to the original one (both have the same solutions) while it improves the contraction power of HC4. The introduction of v (Fig. A.2–right) amounts to adding a redundant equation $n_1 = n_2$ (Fig. A.2–center). If one applies evaluation and narrowing phases of `HC4-revise` until the fixed-point on the new system, one will obtain the interval $[0.19, 4.14]$ for z, instead of $[-0.36, 4.58]$.

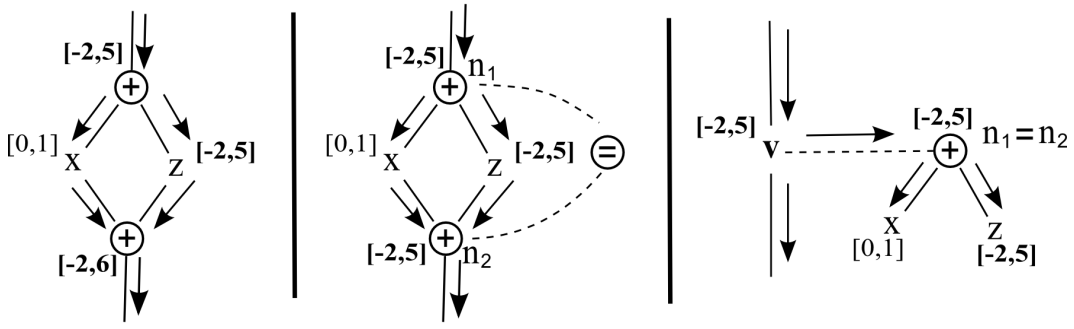


FIG. A.2 – Narrowing/Evaluation without and with CSE.

A.3.1 Additional propagation

Proposition 1 underlines that HC4 might obtain a better filtering when new auxiliary variables and equations corresponding to CSs are added in the system.

Proposition 1 *Let S be a NCSP and S' be the NCSP obtained by replacing in S one CS f in common between two expressions (belonging to constraints in S) by an auxiliary variable v, and by adding the new equation $v = f$. Then,* HC4 *(with a floating-point precision) applied to S' produces a contracted box B' that is smaller than or equal to the box B produced by* HC4 *applied to S.*

Proof. One first produces a system S_1 by replacing in S the first occurrence of f by an auxiliary variable v_1 and the second one by v_2. We add the equations $v_1 = f$ and $v_2 = f$. Because HC4-revise works on acyclic graphs, HC4 computes the 2B-consistency of the decomposed system (i.e., ternary system equivalent to S where all the operators are replaced by auxiliary variables). It is thus well-known that S and S_1 are equivalent : HC4 applied to S_1 and HC4 applied to S produce the same contracted box B [59]. Finally, creating S' amounts to adding the constraint $v_1 = v_2$ to S_1. Thus, the box B' is smaller than or equal to B. □

Of course, this result is useless if the box B' is equal to B, and we want to determine conditions for obtaining a box B' that might be *strictly* smaller than B. Among the set of *primitive operators* that are defined in a standard implementation of HC4, the analysis presented below highlights that the following subset of non-monotonic or non-continuous operators might bring additional contraction when they occur several times (as CS) in the same system : $sin(x)$, $cos(x)$, $tan(x)$ with non-monotonic domains, x^{2c} (c positive integer and $0 \in X$), $cosh(x)$ with $0 \in X$, $1/x$ with $0 \in X$ and binary operators $(+,-,\times,/)$.

A.3.2 Unary operators

Let us first introduce some definitions. An *evaluation* function associated with a function f computes a *conservative* interval, i.e., the application of f on any tuple of values picked inside the input intervals falls inside the computed interval.

Definition 3 *Let $\mathbb{IR}$ be the set of all the intervals over the reals. $F : \mathbb{IR} \to \mathbb{IR}$, $Y = [\underline{y}, \overline{y}] = F(X)$ is an* **evaluation operator** *associated with a unary primitive operator f if :*

$$\forall x \in X, \exists y \in Y \, such \; that \; f(x) = y$$

A *narrowing* operator N_F^x associated with a function f allows us to filter/contract the domain of a variable x.

Definition 4 *Let X be the domain of a variable x, let F be an evaluation operator associated with f, and let Y be an interval. N_F^x is a* **narrowing operator** *of F on x, if $X' = [\underline{x}, \overline{x}] = N_F^x(Y)$*

verifies :

$$f(\underline{x}) \in Y \wedge f(\overline{x}) \in Y \wedge \forall y \in Y, \forall x \in (X - X') : f(x) \neq y$$

Definition 5 *Let f be a function defined on $I(f)$. f is a* **monotonic function** *on an interval X if :*

$$\forall x_1, x_2 \in (X \cap I(f))^2, x_1 \leq x_2 : f(x_1) \leq f(x_2) \text{ **or** } \forall x_1, x_2 \in (X \cap I(f))^2, x_1 \leq x_2 : f(x_1) \geq f(x_2)$$

As said above, a necessary condition to replace a CS is when the contraction obtained by a narrowing operator on a given expression f is partially lost in the next evaluation of f. More formally :

Condition 1 $\exists Y \subseteq F(X), X' = N_F^x(Y) : F(X') \not\subseteq Y$
where X is the domain of variable x, F is the evaluation operator associated with f, and N_F^x is the projection narrowing operator of F on x.

The following proposition indicates a simple condition to identify a *useless CS* for which no filtering is expected.

Proposition 2 *Let F be the evaluation operator associated with a unary operator f. Let N_F^x be the narrowing operator of F on a variable x of domain X.*
If f is a monotonic and continuous function, then : $\forall Y \subseteq F(X), X' = N_F^x(Y) : F(X') \subseteq Y$

Proof. WLOG we suppose that f is monotonically increasing. $X' = N_F(Y)$, then using Def. 4 : $f(\underline{x}), f(\overline{x}) \in Y$, where $X' = [\underline{x}, \overline{x}]$. Finally, with Defs. 3 and 5, $F(X') = [f(\underline{x}), f(\overline{x})] \subseteq Y$. □

Proposition 3 *With the same notations as above, if f is a non-monotonic function, then :*

$$\exists Y \subseteq F(X), X' = N_F^x(Y) : F(X') \not\subseteq Y$$

Proof. The non monotonicity of f means :

$$\exists x_1, x_2, x_3 \subseteq X^3,\ x_1 \leq x_2 \leq x_3 \ s.t.\ f(x_2) > f(x_1)\ \wedge\ f(x_2) > f(x_3)$$

Using values x_1, x_2 and x_3 that satisfy the existency condition, we can suppose that $Y = [f(x_1), f(x_3)]$. As $(f(x_2) > f(x_1)) \wedge (f(x_2) > f(x_3))$, $f(x_2) \notin Y$. $X' = N_F(Y)$, then, with Def. 1, $[x_1, x_3] \subseteq X'$. Since $x_1 \leq x_2 \leq x_3$, $x_2 \in X'$, with Def. 2, $f(x_2) \in F(X')$. Finally, $F(X') \not\subseteq Y$. □

Example. Let $X = [-1, 3]$ be the domain of a variable x, and x^2 be an expression shared by two or more constraints. Suppose that in the narrowing phase of `HC4-revise`, the node corresponding to one of the expressions x^2 is contracted to : $Y = [3, 4]$. Applying the narrowing operator on x produces $X' = [-1, 2]$. In the next evaluation of the expression, $F(X') = [0, 4] \not\subseteq Y$.

Considering the standard operators managed in `HC4` (except operators like `floor`), the *useful CSs* do not satisfy Proposition 2 and satisfy Proposition 3.

A.3.3 N-ary operators (sums, products)

For binary (n-ary) primitive functions, Condition 1 above can be extended to the following **Condition 2** :

$$\exists Z \subseteq F(X,Y), X' = N_F^x(Z,Y), Y' = N_F^y(Z,X) : F(X',Y') \not\subseteq Z$$

where X, Y are the domains of variables x and y respectively, F is the evaluation operator associated with f, N_F^x and N_F^y are the narrowing/projections operators on x and y resp. This condition 2 is generally satisfied by the n-ary operators + and × (resp. − and /). Many examples prove this result (see below). The result is due to intrinsic "bad" properties of interval arithmetics. First, the set of intervals $\mathbb{IR}$ is not a group for addition. That is, let I be an interval : $I - I \neq [0,0]$ (in fact, $[0,0] \subset I - I$). Second, $\mathbb{IR} \setminus \{0\}$ is not a group for multiplication, i.e., $I/I \neq [1,1]$.

The proposition 4 provides a *quantitative idea* of *how much* we can win when replacing additive CSs. It estimates the width Δ that is lost in binary sums (when an additive CS is not replaced by an auxiliary variable). Note that an upper bound of Δ is $2 \times \min(Diam(X), Diam(Y))$ and depends only on the initial domains of the variables.

Proposition 4 *Let $x + y$ be a sum related to a node n inside the tree representation of a constraint. The domains of x and y are the intervals X and Y resp. Suppose that* `HC4-revise` *is carried out on the constraint : in the evaluation phase, the interval of n is set to $V = X + Y$; in the narrowing phase, the interval V is contracted to $V_c = [\underline{V} + \alpha, \overline{V} - \beta]$ (with $\alpha, \beta \geq 0$ being the decrease in left and right bounds of V); X and Y are contracted to X_c and Y_c resp. The difference Δ between the diameter of V_c (current projection) and the diameter of the sum $X_c + Y_c$ (computed in the next evaluation) is :*

$$\Delta = \min(\alpha, Diam(X), Diam(Y), Diam(V) - \alpha) \; + \; \min(\beta, Diam(X), Diam(Y), Diam(V) - \beta)$$

Example. Consider $X = [0,1]$ and $Y = [2,4]$. Thus, $V = X + Y = [2,5]$. Suppose that after applying HC4-revise we obtain $V_c = [2 + \alpha, 5 - \beta] = [4,4]$ ($\alpha = 2$, $\beta = 1$). With Proposition 4, the narrowing operator yields $X_c = [0,1]$ and $Y_c = [3,4]$. Finally, $X_c + Y_c = [3,5]$ is $\Delta = 2$ units larger than $V_c = [4,4]$.

The properties related to multiplication are more difficult to establish. Concise results (not reported here) have been obtained only in the cases when 0 does not belong to the domains or when 0 is a bound of the domains.

A.4 The I-CSE algorithm

The novelty of our algorithm `I-CSE` lies in the way additive and multiplicative CSs are taken into account.

First, I-CSE manages the commutativity and associativity of + and × in a simple way thanks to *intersections* between expressions. An **intersection** between two sums (resp. multiplications) f_1 and f_2 produces the sum (resp. multiplication) of their common terms. For example : $+(x, \times(y, +(z, x^2)), \times(5, z)) \cap +(x^2, x, \times(5, z)) = +(x, \times(5, z))$. Consider two expressions $w_1 \times x \times y \times z_1$ and $w_2 \times y \times x \times z_2$ that share the CS $x \times y$. We are able to view these two expressions as $w_1 \times (x \times y) \times z_1$ and $w_2 \times (x \times y) \times z_2$ since $\times(w_1, x, y, z_1) \cap \times(w_2, y, x, z_2) = \times(x, y)$.

Second, contrarily to existing CSE algorithms, I-CSE handles *conflictive* subexpressions. Two CSs f_a and f_b included in f are **in conflict** (or **conflictive**) if $f_a \cap f_b \neq \emptyset$, $f_a \not\subseteq f_b$ and $f_b \not\subseteq f_a$. An example of conflictive CSs occurs in the expression $f : x \times y \times z$ that contains the conflictive CSs $f_a : x \times y$ and $f_b : y \times z$. Since $x \times y$ and $y \times z$ have a non empty intersection y, it is not possible to directly replace both f_a and f_b in f.

I-CSE works with the n-ary trees encoding the original equations[1] and produces a DAG. The roots of this DAG correspond to the initial equations ; the leaves correspond to the variables and constants ; every internal node f corresponds to an *operator* ($+, \times, sin, exp$, etc) applied to its *children* $t_1, t_2..., t_n$. f represents the expression $f(t_1, t_2, ..., t_n)$ and $t_1, ..., t_n$ are the *terms* of the expression. *The CSs extracted by* I-CSE *are the nodes with several parents.*

We illustrate I-CSE with the following system made of two equations.

$$\begin{aligned} x^2 + y + (y + x^2 + y^3 - 1)^3 + x^3 &= 2 \\ \frac{(x^2 + y^3)(x^2 + cos(y)) + 14}{x^2 + cos(y)} &= 8 \end{aligned}$$

A.4.1 Step 1 : DAG Generation

This step follows a standard algorithm that traverses simultaneously the n-ary trees corresponding to the equations in a bottom-up way (see e.g. [83]). By labelling nodes with identifiers, two nodes with common children and with the same operator are identified equivalent, i.e., they are CSs.

A.4.2 Step 2 : Pairwise intersection between sums and products

Step 2 pairwise intersects, in any order, the nodes corresponding to n-ary sums on one hand, and to n-ary products on the other hand. This step creates *intersection nodes* corresponding to CSs. *Inclusion arcs* link their parents to intersection nodes. If the intersection expression is already present in the DAG, an inclusion arc is just added from each of the two intersected parents to this node.

For instance, on Fig. A.3-a, the node 1.4 is obtained by intersecting the nodes 1 and 4, and we create inclusion arcs from the nodes 1 and 4 to the node 1.4. This means that 2 (i.e., y and x^2) among the 3 terms of the sum/node 4 are in common with 2 among the 4 terms of the sum/node 1. (Note that the two terms are in different orders in the intersected nodes.) The

[1]The + and × operators are viewed as **n-ary** operators. They include − and /. For example, the 3-ary expression $x^2y/(2-x)$ is viewed as $*(x^2, y, 1/(2-x))$.

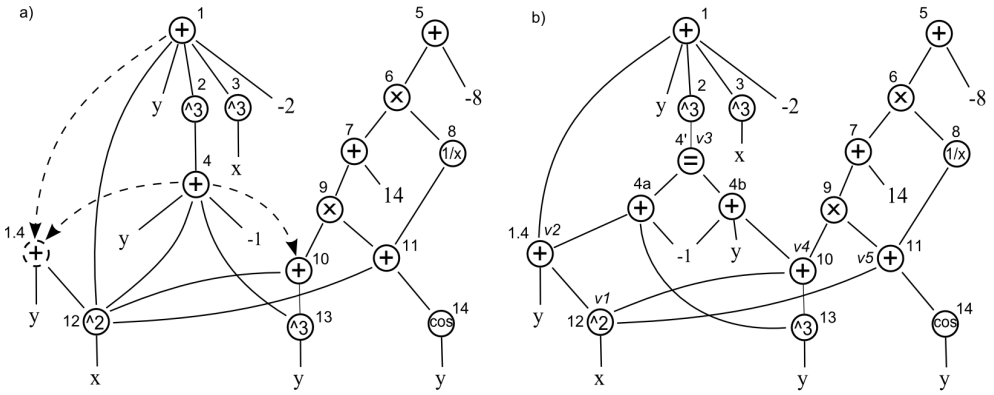


FIG. A.3 – **(a)** DAG obtained after the first two steps of I-CSE. **Step 1** : the system is transformed into a DAG including all the nodes, excepting 1.4, together with the arcs in plain lines. (For the sake of clarity, we have not merged the variables with multiple occurrences.) **Step 2** : the $\times$ and $+$ nodes are pairwise intersected, resulting in the creation of the node 1.4 and the three *inclusion arcs* in dotted lines. **(b)** DAG obtained after **Step 3** : all the inclusion arcs have been integrated into the DAG. For the conflictive subexpressions (nodes 1.4 and 10), a redundant node $4b$ and an equality node $4'$ have been created. The node 1.4 is attached to 4 whereas the node 10 is attached to $4b$. **Step 4** generates the auxiliary variables corresponding to the useful CSs ($v1$, $v2$, $v4$ and $v5$) and to the equality nodes ($v3$).

node 10 corresponds to the intersection between nodes 4 and 10 (in fact the node 10 is included in the node 4), but it has already been created at the first step.

Step 2 is a key step because it makes appear CSs modulo the commutativity and associativity of $+$ and $\times$ operators, and creates at most a quadratic number of CSs. By storing the maximal expressions obtained by intersection, intersection nodes and inclusion arcs enable I-CSE to compute all the CSs before adding them in the DAG in the next step[2].

A.4.3 Step 3 : Integrating intersection nodes into the DAG

In this step, all the intersection nodes are integrated into the DAG, creating the definitive DAG. The routine is top-down and follows the inclusion arcs. Every node f is processed to incorporate into the DAG its "children" reached by an inclusion arc.

If f has no conflictive child, the inclusion arcs outgoing from f are transformed into plain arcs. Also, to preserve the equivalence between the DAG and the system of equations, one removes arcs from f to the children of its intersection terms/children. For instance, on Fig. A.3-b, Step 3 modifies the inclusion arc $1 \rightarrow 1.4$ and removes the arcs $1 \rightarrow 12$ and $1 \rightarrow y$.

Otherwise, f has children/CSs in conflict, i.e., there exist at least two CSs f_a and f_b, included in f, such that $f_a \cap f_b \neq \emptyset$, $f_a \not\subseteq f_b$ and $f_b \not\subseteq f_a$. In this case, one or several nodes $(f_1, f_2, ..., f_r)$

[2]If one did not want to manage conflictive expressions, one would incorporate directly, in Step 2, the new CSs into the DAG.

equivalent to f are added such that : the set $f, f_1, f_2, ..., f_r$ cover all the CSs included in f. To maintain the DAG equivalent to the original system, a node *equal* is created with the children : $f, f_1, f_2, ..., f_r$. For example on Fig. A.3, the node 4 associated with the expression $y+x^2+y^3-1$ has two CSs in conflict. Step 3 creates a node $4b$ (attaching the conflictive CS x^2+y^3) redundant to the node 4 (attaching the other conflictive CS $y+x^2$).

A greedy algorithm has been designed to generate a small number r of redundant nodes, with a small number of children. r is necessarily smaller than the number of CSs included in f. We illustrate our greedy algorithm handling conflictive CSs on a more complicated example, in which an expression (node) $u = s+t+x+y+z$ contains 3 CSs in conflict : $v_1 = s+t$, $v_2 = t+x$, $v_3 = y+z$ (Fig. A.4-a). The greedy algorithm works in two phases. In the first phase, several occurrences of u are generated until all the CSs are replaced. On the example, $u = v_1 + x + v_3$ and $u_1 = s + v_2 + y + z$ are created. The second phase handles all the redundant equations that have been created in the first phase. In a greedy way, it tries to introduce CSs into every equation to obtain a shorter equation that improves filtering. On the example, it transforms $u_1 = s + v_2 + y + z$ into $u_1 = s + v_2 + v_3$. Finally, an equality node (=) is associated with the node u and the redundant node u_1 (Fig. A.4-b).

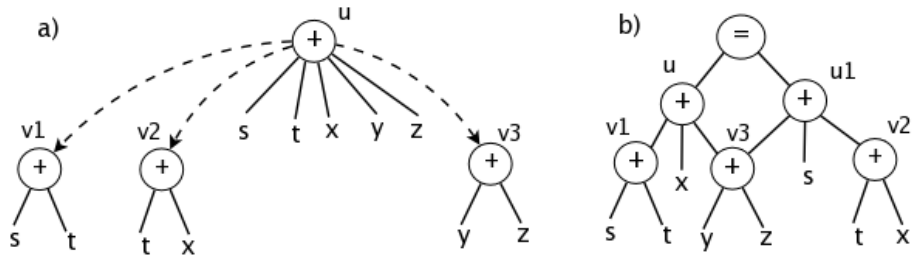


FIG. A.4 – Integrating intersection nodes into the DAG. a) The node u has three CSs in conflict. b) The DAG, with an equality node, obtained by the greedy algorithm.

A.4.4 Step 4 : Generation of the new system

A first way to exploit CSs for solving an NCSP is to use the DAG obtained after Step 3. As shown by Vu et al. in [299], the propagation phase cannot still be carried out by a pure `HC4`, and a more sophisticated propagation algorithm must consider the unique DAG corresponding to the whole system.

Alternatively, in order to still be able to use `HC4` for propagation, and thus to be compatible with existing interval-based solvers, Step 4 generates a new system of equations in which an auxiliary variable v and an equation $v = f$ are added for every *useful* CS. Avoiding the creation of new equations for useless CSs, which cannot provide additional contraction, decreases the size of the new system. In addition, redundant expressions $(f, f_1, f_2, ..., f_r)$ linked by an equality node, add a new auxiliary variable v' and the equations $v' = f$, $v' = f_1$,...,$v' = f_r$. To achieve these tasks, Step 4 traverses the DAG bottom-up and generates variables and equations in every node.

Finally, the new system will be composed by the modified equations (in which the CSs are replaced by their corresponding auxiliary variable), by the auxiliary variables and by the new

constraints $v = f$ corresponding to CSs. The new system corresponding to the example in Fig. A.3 is the following :

$$\begin{array}{rcl} v_2 + (v_3)^3 + x^3 - 2 & = & 0 \\ \dfrac{v_4 \times v_5 + 14}{v_5} - 8 & = & 0 \end{array} \qquad \begin{array}{rcl} v_1 & = & x^2 \\ v_2 & = & y + v_1 \\ v_3 & = & v_2 + y^3 - 1 \end{array} \qquad \begin{array}{rcl} v_3 & = & -1 + y + v_4 \\ v_4 & = & v_1 + y^3 \\ v_5 & = & v_1 + cos(y) \end{array}$$

For a given system of equations, our interval-based solver manages two systems : the new system generated by I-CSE is used only for HC4 and the original system is used for the other operations (bisections, interval Newton). The intervals in both systems must be synchronized during the search of solutions. First, this allows us to clearly validate the interest of I-CSE for HC4. Second, carrying out Newton or bisection steps on auxiliary variables would need to be validated both in theory and in practice. Finally, this implementation is similar to the DAG-based solving algorithm proposed by Vu et al. which also considers only the initial variables for bisections and interval Newton computations, the internal nodes corresponding to CSs being only used for propagation [299].

A.4.5 Time complexity

The time complexity of I-CSE mainly depends on the number n of variables, on the number k of a-ary operators and on the maximum arity a of an a-ary sum or multiplication expression in the system. $k + n$ is the size of the DAG created in Step 1, so that the time complexity of Step 1 is $O(k + n)$ on average if the identifiers are maintained using hashing. In Step 2, the number i of intersections performed is quadratic in the number of sums (or products) in the DAG, i.e., $i = O(k^2)$. Every intersection requires $O(a)$ on average using hashing (a worst-case complexity $O(a\,log(a))$ can be reached with sets encoded by trees/heaps). The worst-case for Step 3 depends on the maximum number of inclusion arcs which is $O(k^2)$. Step 4 is linear in the size of the final DAG and is $O(k + n + i)$. Overall, I-CSE is thus $O(n + a\,log(a)\,k^2)$.

Table A.1 illustrates how the time complexity evolves in practice with the size of the system.

Tab. A.1 – Time complexity of I-CSE on three representative scalable systems of equations (see Section A.6). The CPU times have been obtained with a processor Intel 2.40 GHz. The CPU time increases linearly in the size $k + n$ of the DAG for Trigexp1 and Katsura while the time complexity for Brown reaches the worst-case one.

Benchmark	Trigexp1			Katsura			Brown		
Number n of variables	10	20	40	5	10	20	10	20	40
Number k of operators	46	96	196	15	55	208	10	20	40
I-CSE time in second	0.19	0.28	0.63	0.08	0.19	0.91	0.05	0.20	1.26

A.5 Implementation of I-CSE

I-CSE has been implemented using Mathematica version 6. Mathematica first automatically transforms the equations into a canonical form, where additions and multiplications are n-ary and where are performed reductions, i.e., factorizations by a constant. For instance, the expression $2x - y + x + z$ is transformed into $+(\times(3, x), -y, z)$. The n-ary representation of equations is useful for the pairwise intersections of I-CSE (Step 2).

The solving algorithms are developed in the open source interval-based library in C++ called Ibex [50]. A given benchmark is solved by a branch and prune process : the variables are bisected in a round-robin manner and contracted by constraint propagation (HC4 only, or 3BCID using HC4 – 3BCID is a variant of 3B [283]) and interval Newton. As mentioned above, Ibex offers facilities to create two systems of equations in memory for which domains of variables are synchronized during the search of solutions.

I-CSE-B and I-CSE-NC

We have proven theoretically that the interest of I-CSE resides in the additional pruning it permits and not only in a decrease of the number of operations. To confirm in practice this significant result, we have designed two variants of I-CSE that compute fewer CSs. I-CSE-B (Basic I-CSE) simply ignores the step 2 of I-CSE. The commutativity and associativity of $+$ and $\times$ are not taken into account. Additive and multiplicative n-ary expressions are considered in a fixed binary form in which only a few subexpressions can be detected. For instance, the CS $x + y$ is detected in two expressions $x + y + z_1$ and $x + y + z_2$, but not in expressions $x + z_1 + y$ and $x + z_2 + y$.

I-CSE-NC (I-CSE with No Conflicts) completely exploits the commutativity and associativity of $+$ and $\times$, but does not take into account conflictive CSs. I-CSE-NC lowers the worst case time complexity of I-CSE, but does not replace all the CSs. If a given system does not contain CSs in conflict, I-CSE and I-CSE-NC return the same new system (with no redundant equations). In the example, I-CSE-NC does not create the redundant equation $v_3 = y + v_7$, so that the equation $v_7 = v_9 + y^3$ is not created either. The second (initial) equation finally becomes : $\frac{(x^2+y^3)\times v_8+14}{v_8} = 8$.

Existing CSE algorithms take place between I-CSE-B and I-CSE-NC in terms of number of detected useful CSs. We assume here that the algorithm by Vu et al. [299] is similar to I-CSE-NC.

A.6 Experiments

Benchmarks have been taken in the first two sections (polynomial and non-polynomial systems) of the COPRIN page[3]. The selected sample fulfills systematic criteria : every tested benchmark is an NCSP with a finite number of isolated solutions (no optimization) ; all the solutions can

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

TAB. A.2 – Selected benchmarks. The columns yield the name of the benchmark, the number of solutions (#s), the number of variables (n), the number of useful CSs (#cs) found by `I-CSE-B`, `I-CSE-NC`, `I-CSE`, the number of redundant constraints created by `I-CSE` due to conflictive CSs (#rc).

Benchmark			I-CSE-B	ICSE-NC	I-CSE		Benchmark			I-CSE-B	ICSE-NC	I-CSE	
	#s	n	#cs	#cs	#cs	#rc		#s	n	#cs	#cs	#cs	#rc
6body	5	6	2	3	3	0	Katsura-20	7	21	90	90	90	0
Bellido	8	9	0	1	1	0	Kin1	16	6	13	13	19	3
Brown-7	3	7	3	7	21	24	Pramanik	2	8	0	15	15	0
Brown-7*	3	7	3	1	1	0	Prolog	0	21	0	7	7	0
Brown-30	2	30	26	53	435	783	Rose	16	3	5	5	5	0
BroyBand-20	1	20	22	37	97	73	Trigexp1-30	1	30	29	29	29	0
BroyBand-100	1	100	102	119	479	473	Trigexp1-50	1	50	49	49	49	0
Caprasse	18	4	6	7	11	2	Trigexp2-11	0	11	15	15	15	0
Design	1	9	3	3	3	0	Trigexp2-19	0	19	27	27	27	0
Dis-Integral-6	1	6	4	6	18	9	Trigonom-5	2	5	7	9	20	14
Dis-Integral-20	3	20	18	34	207	171	Trigonom-5*	2	5	7	6	6	0
Eco9	16	8	0	3	7	1	Trigonom-10	24	10	15	15	26	15
EqCombustion	4	5	7	8	11	1	Trigonom-10*	24	10	15	12	12	0
ExtendWood-4	3	4	2	2	2	0	Yamamura-8	7	8	5	10	36	48
Geneig	10	6	11	14	14	0	Yamamura-8*	7	8	5	1	1	0
Hayes	1	8	9	8	8	0	Yamamura-12	9	12	9	18	78	119
I5	30	10	3	4	10	5	Yamamura-12*	9	12	9	1	1	0
Katsura-19	5	20	81	81	81	0	Yamamura-16	9	16	13	26	136	224

be found by the ALIAS system [201] in a time comprised between one second and one hour ; selected systems are written with the following primitive operators : `+`, `-`, $\times$, `/`, `sin`, `cos`, `tan`, `exp`, `log`, `power`. With these criteria, we have selected 40 benchmarks. The `I-CSE` algorithm detects no CS in 16 of them. There are also two more benchmarks (`Fourbar` and `Dipole2`) for which no test has finished before the timeout (one hour), providing no indication. 9 of the remaining 22 benchmarks are scalable, that is, can be defined with any number of variables. Table A.2 provides information about the selected benchmarks. When there is no conflictive CS, `I-CSE` and `I-CSE-NC` return the same new system and there is no redundant constraints (#rc=0). The interval-based solver results will be the same.

For all the benchmarks, the CPU time required by `I-CSE` *(and variants) is often negligible and always less than 1 second.*

Remark. In the benchmarks marked with a star (*), the equations have not been initially rewritten into the canonical form by `Mathematica` (see Section A.5). This leads to fewer CSs, but these CSs correspond to larger subexpressions shared by more expressions, providing generally better results.

Tables A.3 and A.4 compare the CPU times required by `Ibex` to solve the initial system (Init) and the systems generated by `I-CSE-B`, `I-CSE-NC` and `I-CSE`. Table A.3 reports results obtained by a standard branch and prune approach with bisection, Newton and **HC4**. Table A.4 reports

TAB. A.3 – Results obtained with HC4 and interval Newton

Benchmark	Time in second				Time(Init) / Time			#Boxes		
	Init	ICSE-B	ICSE-NC	I-CSE	ICSE-B	ICSE-NC	I-CSE	Init	ICSE-NC	I-CSE
EqCombustion	>3600	26.1	0.35	**0.14**	>137	>10000	**>25000**	>1e+08	3967	**1095**
Rose	>3600	500	**101**	**101**	>7.2	**>35**	**>35**	>3e+07	**865099**	**865099**
Hayes	141	51.9	**15.7**	**15.7**	2.7	**9**	**9**	550489	**44563**	**44563**
6-body	0.22	**0.07**	**0.07**	**0.07**	**3.1**	**3.1**	**3.1**	4985	**495**	**495**
Design	176	65.2	**63.2**	**63.2**	2.7	**2.8**	**2.8**	425153	**122851**	**122851**
I5	>3600	>3600	**1534**	1565	?	**>2.3**	>2.3	>3e+07	**7e+06**	7e+06
Geneig	3323	2910	2722	2722	1.14	**1.22**	**1.22**	7e+08	4e+08	4e+08
Kin1	8.52	8.32	8.32	**8.01**	1.02	1.02	**1.06**	**905**	909	**905**
Pramanik	89.3	92.1	**84.9**	**84.9**	0.97	**1.05**	**1.05**	487255	**378879**	**378879**
Bellido	15.7	15.9	**15.6**	**15.6**	0.99	**1.01**	**1.01**	29759	**29319**	**29319**
Eco9	**23.9**	**23.9**	24	24.1	**1.00**	**1.00**	0.99	126047	117075	**110885**
Caprasse	**1.56**	1.81	1.68	2.16	0.86	0.93	0.72	8521	7793	**7491**
Brown-7*	500	350	**0.01**	**0.01**	1.42	**49500**	**49500**	6e+06	**95**	**95**
Dis-Integral-6	201	0.46	1.3	**0.03**	437	155	**6700**	653035	4157	**47**
ExtendWood-4	29.9	**0.03**	**0.03**	**0.03**	**997**	**997**	**997**	422705	**353**	**353**
Brown-7	500	350	30.7	**1.49**	1.42	16.1	**332**	6e+06	258601	**3681**
Trigexp2-11	1118	208	**56.2**	**56.2**	5.38	**19.9**	**19.9**	1e+06	**316049**	**316049**
Yamamura-8*	13	13.3	**0.75**	**0.75**	0.98	**17.3**	**17.3**	29615	**2161**	**2161**
Broy-Banded-20	778	759	261	**58.1**	1.03	2.98	**13.4**	172959	46761	**12623**
Trigonometric-5*	15.8	12.3	**1.49**	**1.49**	1.28	**10.6**	**10.6**	10531	**1503**	**1503**
Trigonometric-5	15.8	12.3	8.94	**6.97**	1.28	1.77	**2.27**	10531	7369	**5307**
Yamamura-8	13	13.3	44.6	**10.8**	0.98	0.3	**1.20**	29615	115211	**13211**
Katsura-19	**1430**	1583	1583	1583	0.90	0.90	0.90	**145839**	153193	153193
Trigexp1-30	**2465**	3244	3244	3244	0.76	0.76	0.76	1e+07	1e+07	1e+07

results obtained by a branch and prune approach with bisection, Newton and **3BCID** (using HC4 as a refutation algorithm). Both tables report CPU times in seconds obtained on a 2.40 GHz Intel Core 2 processor with 1 Gb of RAM, and the corresponding gain w.r.t. the solving of the original system. The time limit has been set to 3600 seconds. The tables also report the number of generated boxes (#Boxes) during the search. This corresponds to the number of nodes in the tree search and highlights the additional pruning due to I-CSE. The precision of solutions has been set to 10^{-8} for all the benchmarks. The parameter used by HC4 has been set to 1% in Table A.3. The parameters used by HC4 and 3BCID have been set to 10% in Table A.4. We have put at the end of both tables the results corresponding to scalable benchmarks. To return a fair comparison between algorithms, we have selected for the scalable systems the instance with the largest number of variables n such that the solver on the original system finds the solutions in less than one hour. This number n is greater with 3BCID (Table A.4) than with only HC4 (Table A.3) because 3BCID is generally more efficient than HC4.

Tables A.3 and A.4 clearly highlight that I-CSE is very interesting in practice. We observe a gain in performance greater than a factor 2 on 15 among the 24 lines (on both tables). The gain is of two orders of magnitude (or more) for 5 benchmarks with HC4 (corresponding to 4 different systems) and for 10 benchmarks with 3BCID (corresponding to 8 different systems).

TAB. A.4 – Results obtained with `3BCID` using `HC4` and interval Newton

Benchmark	Time in second				Time(Init) / Time			#Boxes		
	Init	ICSE-B	ICSE-NC	I-CSE	ICSE-B	ICSE-NC	I-CSE	**Init**	ICSE-NC	I-CSE
Rose	2882	5.17	**4.04**	**4.04**	557	**713**	**713**	4e+06	**5711**	**5711**
Prolog	38.5	60	**0.14**	**0.14**	0.64	**275**	**275**	4647	**11**	**11**
EqCombustion	0.42	0.37	**0.06**	**0.06**	1.35	7	7	427	**23**	**23**
Hayes	32.6	27.2	**5.67**	**5.67**	1.13	**5.7**	**5.7**	17455	**1675**	**1675**
Design	52	17.9	**13.3**	**13.3**	2.9	**3.9**	**3.9**	16359	**4401**	**4401**
I5	33.5	41.1	**17.9**	**17.8**	0.81	1.9	**1.9**	10619	4387	**4281**
6-body	0.14	**0.08**	0.1	0.1	**1.75**	1.4	1.4	173	**51**	**51**
Kin1	1.66	2.66	1.76	**1.23**	0.62	0.94	**1.35**	**85**	161	197
Bellido	10.3	10.4	**9.98**	**9.98**	1	**1.03**	**1.03**	4487	**4341**	**4341**
Eco9	**11.6**	**11.6**	12.4	13.2	**1**	0.94	0.88	6205	6045	**5749**
Pramanik	**73.8**	114	96.8	96.8	0.65	0.76	0.76	124663	**95305**	**95305**
Caprasse	1.96	2.51	2.5	2.92	0.74	0.78	0.67	1285	1311	**1219**
Geneig	**696**	1050	1050	1050	0.66	0.66	0.66	362225	**362045**	**362045**
Trigexp2-19	2308	2.23	**0.03**	**0.03**	1035	**77000**	**77000**	250178	**7**	**7**
Brown-7*	600	318	**0.01**	**0.01**	1.88	**60000**	**60000**	662415	**9**	**9**
ExtendWood-4	185	**0.03**	**0.03**	**0.03**	6167	6167	6167	669485	**35**	**35**
Dis-Integral-6	135	0.18	0.51	**0.03**	750	264	**4500**	86487	185	**7**
Brown-7	600	318	4.75	**0.22**	1.88	126	**2700**	662415	2035	**23**
Yamamura-12*	1751	1842	**1.01**	**1.01**	0.95	**1700**	**1700**	364105	**307**	**307**
Yamamura-12	1751	1842	31.1	**8.72**	0.95	56.3	**200**	364105	5647	**445**
Trigonometric-10*	1344	506	**19.4**	**19.4**	2.67	**69**	**69**	140512	**2033**	**2033**
Trigonometric-10	1344	506	156	**49.6**	2.67	8.62	**27**	140512	19883	**3339**
Broy-Banded-100	9.96	20.3	14.8	**8.21**	0.49	0.67	**1.21**	13	23	**11**
Trigexp1-50	**0.15**	0.19	0.17	0.17	0.79	0.88	0.88	**1**	**1**	**1**
Katsura20	**3457**	5919	5919	5919	0.58	0.58	0.58	**62451**	120929	120929
Brown-30	>3600	>3600	>3600	**22.9**	?	?	>150	>210021	>151527	**31**
Dis-Integral-20	>3600	>3600	>3600	**1.12**	?	?	**>3200**	>111512	>75640	**39**
Yamamura-16	>3600	>3600	681	**35.6**	?	>5	**>100**	>522300	96341	**919**

`I-CSE` clearly outperforms the variants extracting fewer useful CSs, as shown on Table A.3 (see `Brown-7`, `Dis-Integral-6`, `Broyden-Banded-20`) and Table A.4 (see `Brown-7`, `Dis-Integral-6`, `Yamamura-12`, `Trigonometric-10`). In these cases, the gains in CPU time are significant. They are sometimes of several orders of magnitude. The few exceptions for which `I-CSE` is worse than its simpler variants give only a slight advantage to `I-CSE-NC` or `I-CSE-B`.

The number of boxes is generally decreasing from the left to the right of tables. This confirms our theoretical analysis that expects gains in filtering when a system has additional equations due to CSs. This experimentally proves that exploiting conflictive CSs is useful. This confirms an intuition shared by a lot of practitioners of partial consistency algorithms that redundant constraints are often useful because they allow a better pruning effect [121]. Benchmarks like `Brown-30`, `Dis-Integral-20` and `Yamamura-16`, have been added at the end of Table A.4 to highlight this trend : `I-CSE` produces a gain in performance of 3 orders of magnitude while it adds hundreds of redundant equations.

Most of the obtained results are good or very good, but four benchmarks observe a loss of per-

formance lying between 20% and 42% : `Caprasse` with both strategies, and `Pramanik`, `Geneig`, `Katsura` with `3BCID`. The loss in performance observed for `Katsura-20` (10% or 42% according to the strategy) is due to the domains of the variables that are initialized to [0,1]. Without detailing, such domains imply that the pruning in the search tree is due to the evaluation (bottom-up) phase and not to the (top-down) narrowing phase of `HC4-revise`.

A.7 Conclusion

This paper has presented the algorithm `I-CSE` for exploiting common subexpressions in numerical CSPs. A theoretical analysis has shown that gains in filtering can only be expected when CSs do not correspond to monotonic and continuous operators like x^3 or log. Contrarily to a belief in the community, this means that CSs can bring significant gains in filtering/contraction, and not only a decrease in the number of operations. These are good news for the significance of this line of research.

Experiments have been performed on 40 benchmarks among which 24 contain CSs. Significant gains of one or several orders of magnitude have been observed on 10 of them. `I-CSE` differs from existing CSEs in that it also detects conflictive CSs. As compared to `I-CSE-NC` (similar to existing CSEs), the additional contraction involved by the corresponding redundant equations leads to improvements of one or several orders of magnitude on 4 benchmarks (`Brown`, `Dis-Integral`, `Yamamura` and, only for `HC4`, `BroyBanded`).

A future work is to compare our implementation based on the standard `HC4` algorithm (and the management of two systems), with the sophisticated propagation algorithm carried out on the elegant DAG-based structure proposed by Vu, Schichl and Sam-Haroud. However, our experimental results have underlined that the gain in contraction has a greater impact on efficiency than the time required to reach the fixed-point of propagation. Thus, we suspect that both implementations will show similar performances.

Annexe B

Constructive Interval Disjunction

Article [283] : paru au congrès CP, Constraint Programming, en 2007

Auteurs : Gilles Trombettoni, Gilles Chabert

Abstract

This paper presents two new filtering operators for numerical CSPs (systems with constraints over the reals) based on *constructive disjunction*, as well as a new splitting heuristic. The first operator (`CID`) is a generic algorithm enforcing constructive disjunction with intervals. The second one (`3BCID`) is a hybrid algorithm mixing constructive disjunction and *shaving*, another technique already used with numerical CSPs through the algorithm `3B`. Finally, the splitting strategy learns from the CID filtering step the next variable to be split, with no overhead.

Experiments have been conducted with 20 benchmarks. On several benchmarks, `CID` and `3BCID` produce a gain in performance of orders of magnitude over a standard strategy. `CID` compares advantageously to the `3B` operator while being simpler to implement. Experiments suggest to fix the CID-related parameter in `3BCID`, offering thus to the user a promising variant of `3B`.

B.1 Introduction

We propose in this paper new advances in the use of two refutation principles of constraint programming : shaving and constructive disjunction. We first introduce shaving and then proceed to constructive disjunction that will be considered an improvement of the former.

The shaving principle is used to compute the singleton arc-consistency (SAC) of finite-domain CSPs [64] and the 3B-consistency of numerical CSPs [186]. It is also in the core of the SATZ algorithm [209] proving the satisfiability of boolean formula. Shaving works as follows. A value is temporarily assigned to a variable (the other values are temporarily discarded) and a partial consistency is computed on the remaining subproblem. If an inconsistency is obtained then the value can be safely removed from the domain of the variable. Otherwise, the value is kept in the domain. This principle of refutation has two drawbacks. Contrarily to arc consistency, this consistency is not incremental [31]. Indeed, the work of the underlying refutation algorithm on the *whole* subproblem is the reason why a *single value* can be removed. Thus, obtaining the *singleton arc consistency* on finite-domain CSPs requires an expensive fixed-point propagation algorithm where all the variables must be handled again every time a single value is removed [64]. SAC2 [21] and SAC-optim [31] and other SAC variants obtain better average or worst time complexity by managing heavy data structures for the supports of values (like with AC4) or by duplicating the CSP for every value. However, using these filtering operators inside a backtracking scheme is far from being competitive with the standard MAC algorithm in the current state of research. In its QuickShaving [187], Lhomme uses this shaving principle in a pragmatic way, i.e., with no overhead, by learning the promising variables (i.e., those that can possibly produce gains with shaving in the future) during the search. Researchers and practitioners have also used for a long time the shaving principle in scheduling problems. On numerical CSPs, the 2B-consistency is the refutation algorithm used by 3B-consistency [186]. This property limited to the bounds of intervals explains that 3B-consistency filtering often produces gains in performance.

Example

Figure B.1 (left) shows the first two steps of the 3B-consistency algorithm. Since domains are continuous, shaving does not instantiate a variable to a value but restricts its domain to a subinterval of fixed size located at one of the endpoints. The subproblems are represented with slices in light gray. The 2B-consistency projects every constraint onto a variable and intersects the result of all projections. In the leftmost slice, 2B-consistency leads to an empty box since the projections of the first and the second constraint onto x_2 are two intervals I_1 and I_2 with empty intersection. On the contrary, the fixed-point of projections in the rightmost slice is a nonempty box (with thick border).

The second drawback of shaving is that the pruning effort performed by the partial consistency operator to refute a given value is lost, which is not the case with constructive disjunction[1].

[1]Note that optimized implementations of SAC reuse the domains obtained by subfiltering in subsequent calls to the shaving of a same variable.

Constructive disjunction was proposed by Van Hentenryck et al. in the nineties to handle efficiently disjunctions of constraints, thus tackling a more general model than the standard CSP model [294]. The idea is to propagate independently every term of the disjunction and to perform the union of the different pruned search spaces. In other terms, a value removed by every propagation process (run with one term/constraint of the disjunct) can be safely removed from the ground CSP. This idea is fruitful in several fields such as scheduling, where a common constraint is that two given tasks cannot overlap, or 2D bin packing problems where two rectangles must not overlap.

Constructive disjunction can also be used to handle the classical CSP model (where the problem is viewed as a conjunction of constraints). Indeed, every variable domain can be viewed as a unary disjunctive constraint that imposes one value among the different possible ones ($x = v_1 \vee ... \vee x = v_n$, where x is a variable and $v_1, ..., v_n$ are the different values). In this specific case, similarly to shaving, the constructive disjunction principle can be applied as follows. Every value in the domain of a variable is assigned in turn to this variable (the other values are temporarily discarded), and a partial consistency on the corresponding subproblems is computed. The search space is then replaced by the union of the resulting search spaces. One advantage over shaving is that the (sub)filtering steps performed during constructive disjunction are better reused. This constructive "domain" disjunction is not very much exploited right now while it can sometimes produce impressive gains in performance. In particular, in addition to *all-diff* constraints [241], incorporating constructive domain disjunctions into the famous Sudoku problem (launched, for instance, when the variables/cases have only two remaining possible values/digits) often leads to a backtrack-free solving. The same phenomenon is observed with shaving but at a higher cost [268].

This observation has precisely motivated the research described in this paper devoted to the application of constructive domain disjunction to numerical CSPs. The continuous nature of interval domains is particularly well-suited for constructive domain disjunction. By splitting an interval into several smaller intervals (called *slices*), constructive domain disjunction leads in a straightforward way to the *constructive interval disjunction* (`CID`) filtering operator introduced in this paper.

After useful notations and definitions introduced in Section B.2, Sections B.3 and B.4 describe the *CID* partial consistency and the corresponding filtering operator. A hybrid algorithm mixing shaving and CID is described in Section B.5. Section B.6 presents a new CID-based splitting strategy. Finally, experiments are presented in Section B.7.

B.2 Definitions

The algorithms presented in this paper aim at solving systems of equations or, more generally, numerical CSPs.

Definition 6 *A* **numerical CSP** *(NCSP)* $P = (X, C, B)$ *contains a set of constraints* C *and a set* X *of* n *variables. Every variable* $x_i \in X$ *can take a real value in the interval* $\mathbf{x_i}$ *and* $\mathbf{B}$ *is the cartesian product (called a* **box***)* $\mathbf{x_1} \times ... \times \mathbf{x_n}$*. A solution of* P *is an assignment of the variables in* X *satisfying all the constraints in* C*.*

Remark 1 *Since real numbers cannot be represented in computer architectures, the bounds of an interval* $\mathbf{x_i}$ *should actually be defined as floating-point numbers.*

CID filtering performs a union operation between two boxes.

Definition 7 *Let* B_l *and* B_r *be two boxes corresponding to a same set* V *of variables.*
We call **hull** *of* B_l *and* B_r*, denoted by* `Hull`(B_l, B_r)*, the minimal box including* B_l *and* B_r*.*

To compute a bisection point based on a new (splitting) strategy, we need to calculate the *size* of a box. In this paper, the size of a box is given by its perimeter.

Definition 8 *Let* $B = \mathbf{x_1} \times ... \times \mathbf{x_n}$ *be a box. The* **size** *of* B *is* $\sum_{i=1}^{n}(\overline{\mathbf{x_i}} - \underline{\mathbf{x_i}})$*, where* $\overline{\mathbf{x_i}}$ *and* $\underline{\mathbf{x_i}}$ *are respectively the upper and lower bounds of the interval* $\mathbf{x_i}$*.*

B.3 CID-consistency

The CID-consistency is a new partial consistency that can be obtained on numerical CSPs. Following the principle given in introduction (i.e., combining interval splitting and constructive disjunction), the CID(2)-consistency can be formally defined as follows (see Figure B.1).

Definition 9 (CID(2)-consistency)
Let $P = (X, C, B)$ *be an NCSP. Let* F *be a partial consistency.*

Let B_i^l *be the sub-box of* B *in which* $\mathbf{x_i}$ *is replaced by* $[\underline{\mathbf{x_i}}, \check{\mathbf{x}}_\mathbf{i}]$ *(where* $\check{\mathbf{x}}_\mathbf{i}$ *is the midpoint of* $\mathbf{x_i}$*).*
Let B_i^r *be the sub-box of* B *in which* $\mathbf{x_i}$ *is replaced by* $[\check{\mathbf{x}}_\mathbf{i}, \overline{\mathbf{x_i}}]$*.*

A variable x_i *in* X *is CID(2)-consistent w.r.t.* P *and* F *if*
$B = Hull(F(X, C, B_i^l), F(X, C, B_i^r))$*. The NCSP* P *is CID(2)-consistent if all the variables in* X *are CID(2)-consistent.*

For every dimension, the number of slices considered in the CID(2)-consistency is equal to 2. The definition can be generalized to the CID(s)-consistency in which every variable is split into s slices.

In practice, like 3B-w-consistency, the CID-consistency is obtained with a precision that avoids a slow convergence onto the fixed-point. We will consider that a variable is CID(2,w)-consistent if the hull of the corresponding left and right boxes resulting from subfiltering reduces no variable more than w.

Definition 10 (CID(2,w)-consistency)
With the notations of Definition 9, put $B' = Hull(F(X, C, B_i^l), F(X, C, B_i^r))$*.*

A variable x_i *in* X *is CID(2,w)-consistent if* $|\mathbf{x_i}| - |\mathbf{x_i'}| \leq w$*, where* $\mathbf{x_i}$ *and* $\mathbf{x_i'}$ *are the domain of* x_i *in resp.* B *and* B'*.*
The NCSP is CID(2,w)-consistent if all the variables in X *are CID(2,w)-consistent.*

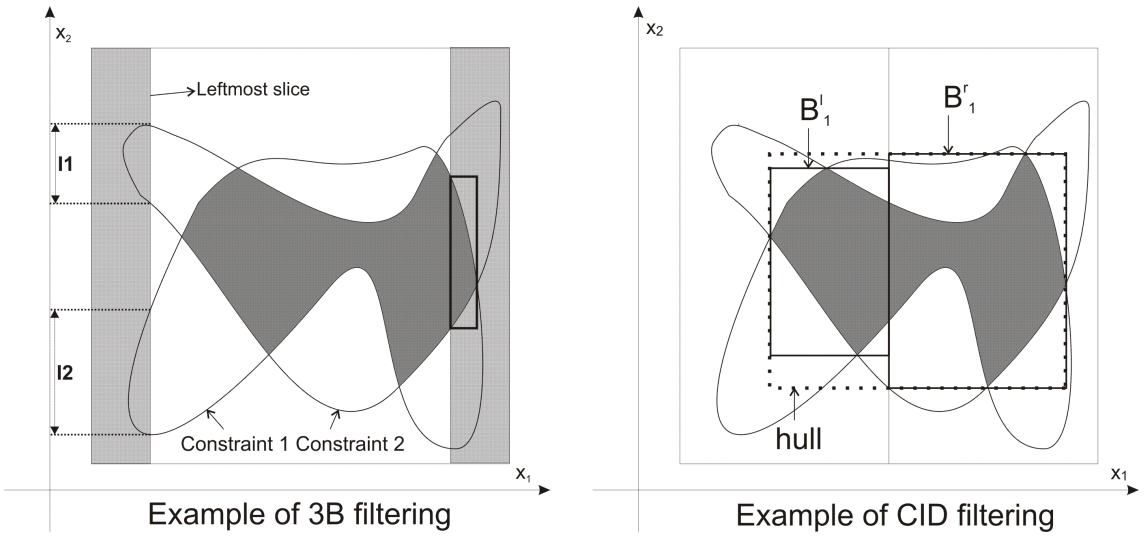


FIG. B.1 – Shaving and CID-consistency on a simple example with two constraints. The constraints are represented with their intersection in dark gray. **Left :** The first two steps of 3B. **Right :** the result of `VarCID` on variable x_1 (with 2 slices). The subboxes B_1^l and B_1^r are represented with thick borders ; the resulting box appears in dotted lines.

Algorithm `CID` details the CID(s,w)-consistency filtering algorithm. Like the 3B-consistency algorithm, `CID` iterates on all the variables until a stop criterion, depending on w, is reached (see Definition 10) : the `Repeat` loop is interrupted when no variable interval has been reduced more than w [2].

Every variable x_i is "*varcided*", i.e., handled by the procedure `VarCID`. The domain of x_i is split into s slices of size $\frac{|\mathbf{x_i}|}{s}$ each by the procedure `SubBox`. The partial consistency operator F (e.g., 2B or a variant of the latter called Box-consistency [293]) filters the corresponding sub-boxes, and the union of the resulting boxes is computed by the `Hull` operator. Note that if the subfiltering operator F applied to a given sub-box `sliceBox` detects an inconsistency, then the result `sliceBox'` is empty, so that there is no use of performing the union of `sliceBox'` with the current box in construction.

B.4 A CID-based solving strategy

To find all the solutions of a numerical CSP, we propose a strategy including a bisection operation, CID filtering and an interval Newton [219]. Between two bisections, two operations are run in sequence :

1. a call to `CID` (how to fix the parameters is discussed just below),
2. a call to an interval Newton operator.

[2] From a theoretical point of view, in order that the stop criterion leads to a (non unique) fixed-point, it is necessary to return the box obtained just before the last filtering, i.e., the box in P_{old} in Algorithm `CID`.

Algorithm `CID` *(s : number of slices, w : precision, in-out $P = (X, C, B)$: an NCSP, F : subfiltering operator and its parameters)*

```
repeat
    P_old ← P
    LoopCID (X, s, P, F)
until StopCriterion(w, P, P_old)
end.
Procedure LoopCID (X, s, in-out P, F)
for every variable xi ∈ X do
    VarCID (xi, s, P, F)
end
end.
Procedure VarCID (xi, s, (X, C, in-out B), F)
B' ← empty box
for j ← 1 to s do
    sliceBox ← SubBox (j, s, xi, B)   /* the j^th sub-box of B on xi */
    sliceBox' ← F(X, C, sliceBox) /* perform a partial consistency */
    B' ← Hull(B', sliceBox')          /* Union with previous sub-boxes */
end
B ← B'
end.
```

The right set of parameters for CID filtering

First of all, the subfiltering operators we chose for `CID` are `2B` and `Box`. The classical user-defined parameter `w-hc4` (percentage of the interval width) is used by the subfiltering operator (`Box` or `2B`) inside CID to control the propagation loop. The parameter s is the number of slices used by the `VarCID` procedure.

Although useful to compute the CID-consistency property, the fixed-point repeat loop used by Algorithm `CID` does not pay off in practice (see below). In other words, running more than once `LoopCID` on all the variables is always counterproductive, even if the value of w is finely tuned. Thus, we have set the parameter w to ∞, that is, we have discarded w from the user-defined parameters.

Endowed with the parameters s and `w-hc4`, CID filtering appears to be an efficient general-purpose operator that has the potential to replace 2B/Box (alone) or 3B. For some (rare) benchmarks however, the use of only 2B/Box appears to be more efficient.

To offer a compromise between pure 2B/Box and `CID`, we introduce a third parameter n' defined as the number of variables that are varcided between two bisections in a round-robin strategy : given a predefined order between variables, a `VarCID` operation is called on n' variables (modulo n), starting at the lattest varcided variable plus one in the order. (This information is transmitted through the different nodes of the tree search.) Thus, if n' is set to 0, then only 2B/Box filtering is called between two bisections ; if $n' = n$ (n is the number of variables in the system), then `CID` is called between two bisections.

We shall henceforth consider that CID has three parameters : the number of slices s, the propagation criterion w-hc4 of 2B/Box used for subfiltering and the number n' of varcided variables.

Default values for CID parameters

So far, we have proposed an efficient general-purpose combination of CID, interval Newton and bisection. However, such a strategy is meaningful only if default values for CID are available. In our solver, the default values provide the following CID operator : CID(s=4, w-hc4=10%, n'=n).

Experiments have led us to select $s = 4$ (see Section B.7.2) :

– The optimal number of slices lies between 2 and 8.
– Selecting $s = 4$ generally leads to the best performance. In the other cases, the performance is not so far from the one obtained with a tuned number of slices.

Discarded variants

Several variants of CID or combinations of operators have been discarded by our experiments. Mentioning these discarded algorithms may be useful.

First, different combinations of bisection, CID and interval Newton have been tried. The proposed combination is the best one, but adding an interval Newton to the 2B/Box subfiltering (i.e., inside CID filtering) produces interesting results as well.

Second, we recommend to forget the fixed-point parameter w in CID. Indeed, a lot of experiments confirmed that relaunching LoopCID several times in a row between two bisections is nearly *never* useful. The same phenomenon has been observed if the relaunch criterion concerns several dimensions, i.e., if the reduction of box size (perimeter or volume) is sufficiently large. A third experiment running LoopCID twice between two splits leads to the same conclusion. Finally, the same kind of experiments have been conducted with no more success to determine which specific variables should be varcided again in an adaptive way. All these experiments give us the strong belief that one LoopCID, i.e., varciding all the variables once, is rather a *maximal* power of filtering.

Third, in a previous workshop version of this paper, we had proposed a CID246 variant of CID, in which the number s of slices was modified between two splits : it was alternatively 2, 4 or 6 : $s = ((i \, modulo \, 3) + 1) \times 2)$, where i indicates the i^{th} call to LoopCID. The comparison with the standard CID was not fair because the parameter s in CID was set to 2. It turns out that CID with $s = 4$ yields nearly the same results as CID246 while being simpler.

Finally, we also investigated the option of a reentrant algorithm, under the acronym k-CID. As k-B-consistency [186] generalizes the 3-B-consistency, it is possible to use $(k-1)$-CID as subfiltering operator inside a k-CID algorithm. Like 4-B-consistency, 2-CID-consistency remains a theoretical partial consistency that is not really useful in practice. Indeed, the pruning power is significant (hence a small number of required splits), but the computational time required to obtain the solutions with 2-CID plus bisection is often not competitive with the time required by 1-CID plus bisection.

B.5 3B, CID and a 3BCID hybrid version

As mentioned in the introduction, the CID partial consistency has several points in common with the well-known 3B-consistency partial consistency [186].

Definition 11 (3B(s)-consistency)
Let $P = (X, C, B)$ be an NCSP. Let B_i^l be the sub-box of B in which $\mathbf{x_i}$ is replaced by $[\underline{\mathbf{x_i}}, \underline{\mathbf{x_i}} + \frac{|\mathbf{x_i}|}{s}]$. Let B_i^r be the sub-box of B in which $\mathbf{x_i}$ is replaced by $[\overline{\mathbf{x_i}} - \frac{|\mathbf{x_i}|}{s}, \overline{\mathbf{x_i}}]$.

Variable x_i is 3B(s)-consistent w.r.t. P if $2B(X, C, B_i^l) \neq \emptyset$ and $2B(X, C, B_i^r) \neq \emptyset$. The NCSP P is 3B(s)-consistent if all the variables in X are 3B(s)-consistent.

For practical considerations, and contrarily to finite-domain CSPs, a partial consistency of an NCSP is generally obtained with a precision w [186]. This precision avoids a slow convergence to obtain the property. Hence, as in `CID`, a parameter w is also required in the outer loop of `3B`.

When the subfiltering operator is performed by *Box consistency*, instead of 2B-consistency, we obtain the so-called *Bound consistency* property [293].

The `3B` algorithm follows a principle similar to `CID`, in which `VarCID` is replaced by a shaving process, called `VarShaving` in this paper. In particular, both algorithms are not incremental, hence the outside repeat loop possibly reruns the treatment of all the variables, as shown in Algorithm `3B`.

Algorithm `3B` *(w : stop criterion precision,* `s` *: shaving precision, in-out $P = (X, C, B)$: an NCSP, F : subfiltering operator and its parameters)*

```
repeat
    for every variable xi ∈ X do
        VarShaving (xi, s, P, F)
    end
until StopCriterion(w, P)
end.
```

The procedure `VarShaving` reduces the left and right bounds of variable x_i by trying to refute intervals with a width at least equal to $\frac{|\mathbf{x_i}|}{s}$. The following proposition highlights the difference between 3B filtering and CID filtering.

Proposition 5 *Let $P = (X, C, B)$ be an NCSP. Consider the box B' obtained by* `CID(s,w)` *w.r.t.* `2B` *and the box B'' obtained by* `3B(s,w)` [3]*. Then,* `CID(s,w)` *filtering is stronger than* `3B(s,w)` *filtering, i.e., B' is included in or equal to B''.*

[3] An additional assumption related to floating-point numbers and to the fixed-point criterion is required in theory to allow a fair comparison between algorithms : the 2B/Box subfiltering operator must work with a subdivision of the slices managed by 3B and CID.

This theoretical property is based on the fact that, due to the hull operation in `VarCID`, the whole box B can be reduced on several, possibly all, dimensions. With `VarShaving`, the pruning effort can impact only x_i, losing all the temporary reductions obtained on the other variables by the different calls to F.

Proposition 1 states that the pruning capacity of `CID` is greater than the one of `3B`. In the general case however, 3B-consistency and CID-consistency are not comparable because s is the exact number of calls to subfiltering F inside `VarCID` (i.e., the upper bound is reached), while s is an upper bound of the number of calls to `2B` by `VarShaving`. Experiments will confirm that a rough work on a given variable with `CID` (e.g., setting $s = 4$) yields better results than a more costly work with `3B` (e.g., setting $s = 10$).

The `3BCID` filtering algorithm

As mentioned above, the `3B` and `CID` filtering operators follow the same scheme, so that several hybrid algorithms have been imagined. The most promising version, called `3BCID`, is presented in this paper.

`3BCID` manages two parameters : the numbers $s_{\texttt{CID}}$ and $s_{\texttt{3B}}$ of slices for the CID part and the shaving part. Every variable x_i is handled by a shaving and a `VarCID` process as follows.

The interval of x_i is first split into $s_{\texttt{3B}}$ slices handled by shaving. Using a subfiltering operator F, a simple shaving procedure tries to refute these slices to the left and to the right (no dichotomic process is performed). Let s_{left} (resp. s_{right}) be the leftmost (resp. rightmost) slice of x_i that has not been refuted by subfiltering, if any. Let $\mathbf{x_i'}$ be the remaining interval of x_i, i.e. :
$\overline{s_{left}} \leq \underline{\mathbf{x_i'}} \leq \overline{\mathbf{x_i'}} \leq \underline{s_{right}}$.

Then, if $\mathbf{x_i'}$ is not empty, it is split into $s_{\texttt{CID}}$ slices and handled by `CID`. One performs the hull of the (at most) $s_{\texttt{CID}} + 2$ boxes handled by the subfiltering operator F : s_{left}, s_{right} and the $s_{\texttt{CID}}$ slices between s_{left} and s_{right}.

It is straightforward to prove that the obtained partial consistency is stronger than 3B($s_{\texttt{3B}}$)-consistency.

The experiments will show that `3BCID` with $s_{\texttt{CID}} = 1$ can be viewed as an improved version of `3B` where constructive disjunction produces an additional pruning effect with a low overhead.

B.6 A new CID-based splitting strategy

There are three main splitting strategies (i.e., variable choice heuristics) used for solving numerical CSPs. The simplest one follows a *round-robin* strategy and loops on all the variables. Another heuristic selects the variable with the largest interval. A third one, based on the *smear function* [219], selects a variable x_i implied in equations whose derivative w.r.t. x_i is large.

The round-robin strategy ensures that all the variables are split in a branch of the search tree. Indeed, as opposed to finite-domain CSPs, note that a variable interval is generally split (i.e.,

instantiated) several times before finding a solution (i.e., obtaining a small interval of width less than the precision). The largest interval strategy also leads the solving process to not always select a same variable as long as its domain size decreases. The strategy based on the smear function sometimes splits always the same variables so that an interleaved schema with round-robin, or a preconditionning phase, is sometimes necessary to make it effective in practice.

We introduce in this section a new CID-based splitting strategy. Let us first consider different box sizes related to (and learnt during) the `VarCID` procedure applied to a given variable x_i :

- Let $\texttt{OldBox}_i$ be the box B just before the call to `VarCID` on x_i. Let $\texttt{NewBox}_i$ be the box obtained after the call to `VarCID` on x_i.
- Let $B_i^{l'}$ and $B_i^{r'}$ be the left and right boxes computed in `VarCID`, after a reduction by the F filtering operator, and before the `Hull` operation.

The ratio `ratioBis` leads to an "intelligent" splitting strategy. The ratio `ratioBis` is equal to $\frac{f(Size(B_i^{l'}),Size(B_i^{r'}))}{Size(NewBox)}$, where f is any function that aggregates the size of two boxes (e.g., sum), in a sense computes the size lost by the `Hull` operation of `VarCID`. In other words, $B_i^{l'}$ and $B_i^{r'}$ represent precisely the boxes one would obtain if one splits the variable x_i (instead of performing the hull operation) immediately after the call to `VarCID` ; `NewBox` is the box obtained by the `Hull` operation used by `CID` to avoid a combinatorial explosion due to a choice point.

Thus, after a call to `LoopCID`, the CID principle allows us to learn about a good variable interval to be split : *one selects the variable having led to the lowest* `ratioBis`. Although not related to constructive disjunction, similar strategies have been applied to finite-domain CSPs [97, 239].

Experiments, not reported here, have compared a large number of variants of `ratioBis` with different functions f. The best variant is `ratioBis`$= \frac{Size(B_i^{l'})+Size(B_i^{r'})}{Size(NewBox)}$.

B.7 Experiments

We have performed a lot of comparisons and tests on a sample of 20 instances. These tests have helped us to design efficient variants of CID filtering.

B.7.1 Benchmarks and interval-based solver

Twenty benchmarks are briefly presented in this section. Five of them are sparse systems found in [221] : `Hourglass`, `Tetra`, `Tangent`, `Ponts`, `Mechanism`. They are challenging for general-purpose interval-based techniques, but the algorithm `IBB` can efficiently exploit a preliminary decomposition of the systems into small subsystems [221]. The other benchmarks have been found in the Web page of the COPRIN research team or in the COCONUT Web page where the reader can find more details about them [204]. The precision of the solutions. i.e., the size of interval under which a variable interval is not split, is $1e-08$ for all the benchmarks, and $5e-06$ for `Mechanism`. `2B` is used for all the benchmarks but one because it is the most efficient local consistency filtering when used alone inside `3B` or `CID`. `Box+2B` is more adequate for `Yamamura8`.

All the selected instances can be solved in an acceptable amount of time by a standard algorithm in order to make possible comparisons between numerous variants. No selected benchmark has been discarded for any other reason!

All the tests have been performed on a `Pentium IV 2.66 Ghz` using the interval-based library in `C++` developed by the second author. This new solver provides the main standard interval operators such as `Box`, `2B`, interval Newton [219]. The solver provides round-robin, largest-interval and CID-based splitting strategies. Although recent and under developement, the library seems competitive with up-to-date solvers like `RealPaver` [112]. For all the presented solving techniques, including `3B` and `3BCID`, an interval Newton is called just before a splitting operation iff the width of the largest variable interval is less than $1e-2$.

B.7.2 Results obtained by `CID`

Table B.1 reports the results obtained by `CID`(s, `w-hc4`, n'), as defined in Section B.4.

The drastic reduction in the number of required bisections (often several orders of magnitude) clearly underlines the filtering power of `CID`. In addition, impressive gains in running time are obtained by `CID` for the benchmarks on the top of the table, as compared to standard strategy using `2B` or `2B`+`Box`, an interval Newton and a round-robin splitting policy[4].

`CID` often obtains better running times than the standard strategy, except on `Bellido`, `Trigexp2-5` and `Caprasse`, for which the loss in performance is small. However, it is not reasonable to propose to the user an operator for which three parameters must be tuned by hand. That is why we have also reported the last three `CID` columns where only 0 or 1 parameter has been tuned. The default values (fourth `CID` column with $s = 4$, `w-hc4`$= 10\%$, $n' = n$) yield very good results : it outperforms the standard strategy in 15 of the 21 instances. In particular, setting $s = 4$ provides the best results in 12 of the 21 instances.

The second and third `CID` columns report very good results obtained by a filtering algorithm with only one parameter to be tuned (like with `2B` or `Box`). Hence, since `CID` with only parameter n' to be tuned (second `CID` column) allows a continuum between pure `2B` and `CID` ($n' = 0$ amounts to a call to `2B`), a first recommendation is to propose this `CID` variant in interval-based solvers.

The second recommendation comes from a combinatorial consideration. In a sense, constructive interval disjunction can be viewed as a "polynomial-time splitting" since `VarCID` performs a polynomial-time hull after having handled the different slices. However, the exponential aspect of (classical) bisection becomes time-consuming only when the number of variables becomes high. This would explain why `CID` cannot pay off on `Trigexp2-5` and `Caprasse` which have a very small number of variables and lead to no combinatorial explosion due to bisection. (The intuition is confirmed by the better behavior of `CID` on the (scalable) variant of `Trigexp2` with 9 variables.) Thus, the second recommendation would be to use `CID` on systems having a minimal number of variables, e.g., 5 or 8.

[4]Note that this strategy is inefficient for `Trigexp1` (solved in 371 seconds) and for `Mechanism` (that is not solved after a timeout (TO) of several hours). However, a reasonable running time can be obtained with a variant of `2B` that push *all* the constraints of the NCSP in the propagation queue after every bisection.

Name	n	#s	w-hc4	2B/Box + Newton	CID $(s, \texttt{whc4}, n')$	CID $(4, 10\%, n')$	CID $(4, \texttt{whc4}, n)$	CID $(4, 10\%, n)$	s	w-hc4	n'
BroydenTri	32	2	15%	758 2e+07	**0.12** 46	0.28 44	0.19 65	0.45 50	4	80%	40
Hourglass	29	8	5%	24 1e+05	**0.44** 109	0.44 109	0.52 80	0.52 80	4	10%	17
Tetra	30	256	0.02%	401 1e+06	**10.1** 2116	11.6 1558	11.7 1690	14.5 1320	4	30%	20
Tangent	28	128	15%	32 1e+05	**3.7** 692	3.7 692	4.9 447	5.1 450	4	50%	28
Reactors	20	38	5%	156 1e+06	**15.6** 2588	16.4 2803	16.7 2381	17.7 2156	4	15%	18
Trigexp1	30	1	20%	371(3.4) 5025	**0.12** 1	0.15 3	0.14 2	0.14 3	8	2%	30
Discrete25	27	1	0.01%	5.2 1741	**0.62** 2	1.08 3	0.84 12	2.13 99	8	0.5%	35
I5	10	30	2%	692 3e+06	**126** 23105	147 60800	150 20874	157 32309	6	2%	5
Transistor	12	1	10%	179 1e+06	**66** 11008	79.4 31426	91.4 16333	91.4 16333	8	10%	6
Ponts	30	128	5%	10.8 34994	**2.7** 388	2.9 338	2.9 380	3.1 304	4	30%	25
Yamamura8	8	7	1%	13 1032	**7.5** 104	9.5 60	9.5 60	9.5 60	4	10%	4
Design	9	1	10%	395 3e+06	**275** 200272	278 256000	313 76633	313 76633	5	10%	2
D1	12	16	5%	4.1 35670	**1.7** 464	1.7 464	1.7 464	1.7 464	4	10%	12
Mechanism	98	448	0.5%	TO(111) 24538	**43.1** 3419	45.2 3300	46.6 2100	47.8 2420	4	2%	50
Hayes	8	1	0.01%	155 3e+05	**75.8** 1e+05	77 1e+05	111 81750	147 58234	4	80%	2
Kin1	6	16	10%	84 70368	**76.8** 6892	76.8 6892	83.5 4837	87.4 4100	4	10%	3
Eco9	8	16	10%	26 2e+05	**18** 55657	19.4 46902	26.6 10064	26.6 10064	3	10%	1
Bellido	9	8	10%	**80** 7e+05	94.4 1e+05	94.4 1e+05	106 45377	106 45377	4	10%	3
Trigexp2-9	9	1	20%	61.8 3e+05	**50.4** 4887	65.14 14528	62.4 11574	68.4 9541	6	10%	9
Trigexp2-5	5	1	20%	**3.0** 13614	3.8 10221	4.6 4631	6.2 2293	6.6 1887	2	20%	1
Caprasse	4	18	30%	**2.6** 37788	2.73 18176	3.0 12052	4.7 5308	5.1 5624	2	5%	1

TAB. B.1 – Comparison between [CID + interval Newton + round-robin bisection strategy] and [a standard strategy] : 2B/Box + interval Newton + round-robin splitting. n is the number of variables. The column #s yields the number of solutions. The first column w-hc4 is the user-defined parameter w-hc4 used by 2B or Box. The last 3 columns s, w-hc4 and n' indicate the values of parameters that have been tuned for CID (first CID column). The second CID column reports the results of CID when $s = 4$ and w-hc4 = 10%, i.e., when only n' is tuned. The third CID column reports the results of CID when $s = 4$ and $n' = n$, i.e., when only w-hc4 is tuned. The fourth CID column reports the results of CID with the default values for parameters, i.e., $s = 4$, w-hc4= 10%, $n' = n$. Every cell contains two values : the CPU time in seconds to compute all the solutions (top), and the number of required bisections (bottom). For every benchmark, the best CPU time is bold-faced.

B.7.3 Comparing CID, 3B and 3BCID

Table B.2 reports the results obtained by CID, 3B and 3BCID (see Section B.5). All the results have been obtained with a parameter w-hc4 set to 5%.

Several trials have been performed for every algorithm, and the best result is reported in the table. For 3B, seven values have been tried for the parameter $s_{\texttt{3B}}$: 4, 5, 7, 10, 20, 50, 100.

For CID, nine values of the parameters have been tried : five values for the number of slices $s_{\texttt{CID}}$ (2, 3, 4, 6, 8 ; fixing $n' = n$), and four values for the parameter n' (1, $0.5n$, $0.75n$, $1.2n$; fixing $s_{\texttt{CID}} = 4$). For 3BCID, eight combinations of parameters have been tried : four values for $s_{\texttt{CID}}$ (1,2,3,4) combined with two values for $s_{\texttt{3B}}$ (10, 20).

Name	n	2B/Box	CID	3B	3BCID($s_{\texttt{CID}} = 1$)	3BCID($s_{\texttt{CID}} = 2$)
BroydenTri	32	758	0.23	0.22	**0.18**	0.19
Hourglass	29	24	0.45	0.73	**0.43**	0.50
Tetra	30	401	**13.6**	20.7	17.1	18.8
Tangent	28	32	4.13	8.67	**3.18**	4.13
Reactors	20	156	18.2	24.2	**15.5**	16.9
Trigexp1	30	3.4	**0.10**	0.26	0.12	0.11
Discrete25	27	5.2	1.37	2.19	1.26	**1.13**
I5	10	692	139	144	**115**	123
Transistor	12	179	71.5	77.9	49.3	**46.9**
Ponts	30	10.8	**3.07**	5.75	4.19	4.43
Yamamura8	8	13	**9.0**	9.1	10.3	10.7
Design	9	395	300	403	**228**	256
D1	12	4.1	1.78	2.99	**1.64**	1.76
Mechanism	98	111	**79**	185	176	173
Hayes	8	155	**99**	188	102	110
Kinematics1	6	84	**76.1**	136	76.6	81.4
Eco9	8	26	**19.3**	40.1	27.0	30.3
Bellido	9	**80**	95	143	**93**	102
Trigexp2-9	9	61.8	52.2	74.5	**39.9**	45.1
Caprasse	4	**2.6**	**3.1**	9.38	4.84	5.35

TAB. B.2 – Comparison between CID, 3B and 3BCID. The first three columns recall resp. the name of the benchmark, its number of variables and the CPU time in seconds required by the strategy 2B/Box + Newton + bisection with round-robin. The other columns report the CPU time required to solve the benchmarks with CID, 3B and 3BCID. The best CPU time result is bold-faced.

The main conclusions drawn from Table B.2 are the following :

– 3BCID and CID always outperform 3B. Even the standard strategy outperforms 3B for 9 benchmarks in the bottom of the table.

– 3BCID is competitive with CID. 3BCID is better than CID concerning 11 benchmarks. CID remains even better for Mechanism. We wonder if it is related to its large number of variables.

The good news is that the best value for $s_{\texttt{CID}}$ in 3BCID is often $s_{\texttt{CID}} = 1$. In only four cases, the best value is greater, but the value $s_{\texttt{CID}} = 1$ also provides very good results. This suggests to propose 3BCID with $s_{\texttt{CID}} = 1$ as an alternative of 3B. In other words, 3BCID with $s_{\texttt{CID}} = 1$ can be viewed as a promising implementation of 3B. This is transparent for a user who has to specify the same parameter $s_{\texttt{3B}}$, the management of constructive disjunction being hidden inside 3BCID (that performs a Hull operation of at most 3 boxes since $s_{\texttt{CID}} = 1$).

B.7.4 Comparing splitting strategies

Table B.3 applies the three available splitting strategies to CID with $n' = n$ and $s = 4$. We underline some observations.

Filtering	CID	CID	CID
Splitting	Round-robin	Largest Int.	CID-based
BroydenTri	0.21	0.18	**0.17**
Hourglass	0.52	0.51	**0.37**
Tetra	**12.1**	28.2	16.4
Tangent	**3.7**	21.7	5.2
Reactors	17.0	13.2	**12.7**
Trigexp1	0.15	0.19	**0.14**
Discrete25	**0.84**	1.49	1.06
I5	**151**	421	179
Transistor	93	**36**	41
Ponts	2.92	5.51	**2.31**
Yamamura8	9.5	6.9	**5.1**
Design	318	334	**178**
D1	**1.72**	2.96	2.50
Mechanism	47	49	**46**
Hayes	**115**	564	318
Kinematics1	83	70	**63**
Eco9	26.7	31.4	**26.1**
Bellido	107	102	**99**
Trigexp2-9	62	55	**53**
Caprasse	5.16	5.43	**5.04**

TAB. B.3 – Comparison on CID with three splitting strategies : Round-robin, Largest interval and the new CID-based strategy.

The new CID-based splitting strategy is better than the other strategies on 13 of the 20 instances, especially on Design. The largest interval strategy is the best on only one instance. The round-

robin strategy is the best on 6 instances. On the 7 instances for which the CID-based strategy is not the best, the loss in performance is significant on `Hayes`.

The behavior of the CID-based strategy with $s = 6$ (not reported here) is even better, the round-robin strategy being the best on only 3 instances. This would suggest that the `ratioCID` learned during a `VarCID` operation is more accurate with a higher number of slices.

B.8 Conclusion

This paper has introduced two new filtering operators based on the constructive disjunction principle exploited in combinatorial problems. The first experimental results are very promising and we believe that `CID` and `3BCID` have the potential to become standard operators in interval constraint solvers. The `CID` operator also opens the door to a new splitting strategy learning from the work of CID filtering.

The experiments lead to clear recommendations concerning the use of these new filtering operators. First, `CID` can be used with fixed values of parameters s and `w-hc4`, letting the user only tune the third parameter n' (i.e., the number of variables that are varcided between 2 bisections). This allows the user to select in a sense a rate of CID filtering, $n' = 0$ producing the pure `2B/Box`. Used this way, `CID` could maybe subsume existing filtering operators. Second, `3BCID` with $s = 1$ can be provided as a promising alternative of a `3B` operator.

Several questions remain open. It seems that, due to combinatorial considerations, `CID` is not convenient for small problems while it seems more interesting for large-scale systems. A more complete experimental study should confirm or contradict this claim. Also, `3BCID` should be compared to the weak-3B operator implemented in RealPaver [112][5]. A comparison with filtering algorithms based on linearization, like `Quad` [183], will be performed as well.

Moreover, the CID-based splitting strategy merits a deeper experimental study. In particular, a comparison with the *smear* function will be performed once the latter is implemented in our solver.

An interesting future work is to propose *adaptive* variants of `CID` that can choose which specific variable should be varcided or bisected next.

Acknowledgements

Special thanks to Olivier Lhomme for useful discussions about this research. Also thanks to Bertrand Neveu, Arnold Neumaier and the anonymous reviewers.

[5] An implementation of both operators in a same solver would lead to a fair comparison.

Annexe C

First comparison between 3B and 2B

Travaux préliminaires

Auteurs : Gilles Trombettoni, Gilles Chabert, Bertrand Neveu, Ignacio Araya

The reader can compare both contractors by analyzing the experimental results obtained in our latest articles that intensively use `3BCID(HC4)` [9, 12, 10, 11]. These experiments have been performed in a fair way with a small number of parameter values (sometimes only once, i.e., with default parameter values). The same protocol was followed by all the competitors.

Table C.1 yields results obtained on roughly the same instances, but with the *best* possible parameter values. Bertrand Neveu has manually (in the aim of being as fair as possible) tuned the parameters of `HC4`, `3BCID`, interval multivariate Newton, and the variable selection heuristics (among largest-first, round-robin, *smear* function [167]).

These results confirm those obtained with default parameter values. The strategy using `3BCID(HC4)` sometimes shows speedups of orders of magnitude compared to that using `HC4`. In a minority of cases, `3BCID(HC4)` shows a small loss of performance (except for `Bratu`).

Gain=$\frac{Time(HC4)}{Time(3BCID)}$	**Systems**
HC4 timeout $>> 1\,h$	Tangent(geo), Sierpinski3(geo), Sierpinski3(eq), Chair, Latham, Virasoro, Yamamura-12, Yamamura-modif-14, Prolog, I5, BroydenBanded-20, BroydenTri-30, ExtendedFreudenstein-40,
`3BCID` $< 1\,h$	Brown-7, Reactors-30, Trigexp1-100
Gain > 10	BroydenTri-20, ExtendedFreudenstein-20, Reactors-20, Tetra
$10 >$ Gain > 2	Design, Hayes, Star, Brent-10, Trigexp2-9, Brent-8, Brown-6, Ponts, Redeco, 6body, Kin1
$2 >$ Gain > 1.2	Bellido, Trigexp2-7, Rose
$1.2 >$ Gain > 0.8	Brown-5, Caprasse, DiscreteBoundary-1000, Katsura-20, EqCombustion, Eco9, DisIntegral-6, Trigo1-10
$0.8 >$ Gain > 0.5	Pramanik, Geneig, Sjirkboon
$0.5 >$ Gain	Bratu-60 (0.3), Bratu-100 (0.2)

Tab. C.1 – Comparaison between solving strategies based on `3BCID(HC4)` and on `HC4` only. Most of the tested nonlinear square systems of equations appear in the Web page of the team COPRIN [204]. Some of them are defined with an arbitrary number of equations (this number is specified after the system name) and appear sometimes several times in the table with different sizes. Some others are sparse and are described in our works concerning the `IBB` algorithm (see Section 4.2.3 and Chapter I).

Annexe D

An Interval Constraint Propagation Algorithm Exploiting Monotonicity

Article [11] : paru au workshop IntCP du congrès CP, Constraint Programming, en septembre 2009

(Le point d'entrée est maintenant la référence [13] correspondant à la publication de `Mohc` au congrès AAAI en juillet 2010. Les détails de l'algorithme se trouvent dans la thèse de Ignacio Araya [7].)

Auteurs : Ignacio Araya, Bertrand Neveu, Gilles Trombettoni

Abstract

When a function f is monotonic w.r.t. a variable x in a given box, it is well-known that the monotonicity-based interval extension of f computes a sharper image than the natural interval extension does. Indeed, the overestimation due to the variable x with multiple occurrences in f disappears. However, monotonicity has not been exploited in interval filtering/contraction algorithms for solving systems of nonlinear constraints over the reals.

We propose in this paper a new interval constraint propagation algorithm, called MOnotonic Hull Consistency (`Mohc`), that exploits monotonicity of functions. The propagation is standard, but the `Mohc-Revise` procedure, used to filter/contract the variable domains involved in an individual constraint, is novel. This revise procedure uses two main bricks for narrowing intervals of the variables involved in f. One procedure is a monotonic version of the well-known `HC4-Revise`. A second procedure performs a dichotomic process calling interval Newton iterations, close to (while less costly than) the procedure `BoxNarrow` used in the `Box` contraction algorithm.

When f is monotonic w.r.t. every variable with multiple occurrences, `Mohc` is proven to compute the optimal/sharpest box enclosing all the solutions of the constraint (hull consistency). Experiments show that `Mohc` is a relevant approach to handle constraints having *several* variables with multiple occurrences, contrarily to `HC4` and `Box`.

D.1 Introduction

Interval-based solvers can solve systems of numerical constraints (i.e., equations or inequalities over the reals). They are becoming useful for handling numerical CSPs encountered in dynamic systems defined in robust control or autonomous robot localization [168]. Also, novel applications emerge from various domains such as robotics design and kinematics [203], or proof of conjectures (e.g., the proof of Lorentz's strange attractors detailed in [289]).

Two main types of algorithms allow solvers to remove inconsistent values from the domains of variables. Interval Newton and related algorithms generalize to intervals standard numerical analysis methods [147, 212]. Contraction algorithms issued from constraint programming are also in the heart of interval-based solvers. The constraint propagation algorithms `HC4` [28] and `Box` [293, 28] are very often used by solvers. They perform a propagation loop and filter the variable domains (i.e., improve their bounds) with a specific *Revise* procedure (called `HC4-Revise` and `BoxNarrow`) handling the constraints individually.

In practice, `HC4-Revise` often computes an optimal box enclosing all the solutions to one constraint c when no variable appears twice in c. When one critical variable appears several times in c, `HC4-Revise` is generally *not* optimal. In this case, `BoxNarrow` is proven to compute a sharper box. The new revise algorithm presented in this paper, called `Mohc-Revise`, tries to handle the general case when *several* variables have multiple occurrences in c.

When a function f is monotonic w.r.t. to a variable x in a given box, it is well-known that the monotonicity-based interval extension of f produces no overestimation related to the multiple occurrences of x. `Mohc-Revise` exploits this property to improve contraction/filtering. Monotonicity is generally verified for a few pairs (f, x) at the beginning of the search, but can be detected for more pairs as long as one goes to the bottom of the search tree, handling smaller boxes.

After introducing notations and background in Section D.2, Sections D.3 and D.4 describe the `Mohc-Revise` algorithm, leading to two constraint propagation variants called `LazyMohc` and `Mohc`. Section D.5 details related properties. In particular, when f is monotonic w.r.t. every variable with multiple occurrences, `Mohc` (a variant in fact) is proven to compute the optimal/sharpest box enclosing all the solutions of the constraint (hull consistency property). Experiments shown in Section D.6 highlight that `Mohc` is a relevant approach to handle constraints having several variables with multiple occurrences.

D.2 Background

The algorithms presented in this paper aim at solving systems of equations or, more generally, numerical CSPs.

Definition 12 *A* **numerical CSP** *(NCSP) $P = (X, C, B)$ contains a set of constraints C and a set X of n variables. Every variable $x_i \in X$ can take a real value in the interval $[x_i]$ and B is the Cartesian product (called a* **box***) $[x_1] \times ... \times [x_n]$. A solution of P is an assignment of the variables in X satisfying all the constraints in C.*

Since real numbers cannot be represented in computer architectures, note that the bounds of an interval $[x_i]$ should actually be defined as floating-point numbers. Most of the set operations can be achieved on boxes, such as inclusion and intersection. An operator `Hull` is often used to compute an outer approximation of the union of several boxes. It returns the minimal box including the input boxes.

$\underline{x}$, resp. $\overline{x}$, denotes the lower bound, resp. the upper bound, of $[x]$. `Mid`$([x])$ denotes the midpoint of $[x]$ while `Diam`$([x]) \equiv \overline{x} - \underline{x}$ denotes the diameter, or size, of the interval $[x]$.

To find all the solutions of an NCSP with interval-based techniques, the solving process starts from an initial box representing the search space and builds a search tree. The tree search **bisects** the current box, that is, **splits** on one dimension (variable) the box into two sub-boxes, thus generating one choice point. At every node of the search tree, filtering (also called **contraction**) algorithms reduce the bounds of the current box. These algorithms comprise **interval Newton** algorithms issued from the numerical analysis community [147, 219] along with contraction algorithms issued from the constraint programming community. The process terminates with **atomic boxes** of size at most ω on every dimension.

The contraction algorithm presented in this paper proposes new procedures. They are adaptations of the classical `HC4-Revise` [28] and `BoxNarrow` [293, 28] procedures that take advantage of the monotonicity of functions.

The `HC4` algorithm performs an AC3-like propagation loop. Its revise procedure, called `HC4-Revise`, traverses twice the tree representing the mathematical expression of the constraint for narrowing all the involved variable intervals. An example is shown in Fig. D.1.

`Box` is also a propagation algorithm. For every pair (f, x), where f is a function of the considered NCSP and x is a variable involved in f, the a other variables in f are replaced with their interval $[y_1], ..., [y_a]$, and the `BoxNarrow` procedure reduces the bounds of $[x]$ such that the new left (resp. right) bound is the leftmost (resp. rightmost) solution of the univariate equation $f(x, [y_1], ..., [y_a]) = 0$. Existing *revise* procedures use a *shaving* principle to narrow $[x]$: Slices $[s_i]$ inside $[x]$ with no solution are discarded by checking whether $f([s_i], [y_1], ..., [y_a])$ does not contain 0 and by using a univariate interval Newton. `Box` is stronger than `HC4` in that the narrowing effort performed by `Box` on a variable x with multiple occurrences removes the overestimation effect on it. However, it is *not optimal in case the other variables y_i also have multiple occurrences.*

These algorithms are used in our experiments as a sub-contractor embedded in a `3B` algorithm [186] (or a variant `3BCID` [283]). `3B` uses a shaving refutation principle similar to SAC [64]. The main procedure splits an interval into slices. A slice at the bounds is discarded if calling a sub-contractor (e.g. `HC4`) on the resulting subproblem leads to no solution.

The `Mohc` algorithm exploits the monotonicity of functions to better contract intervals. Let us first introduce the well-known monotonicity-based interval extension of f, denoted $[f]_M$ in this article. It appears that the overestimation due to a variable x occurring several times in f disappears when f is monotonic w.r.t. x in a given box. (For the sake of conciseness, we sometimes write that "x is monotonic".) Let us take $f(x) = x^3 - 3x^2 + x$ as an example. The image of $[3, 4]$ calculated by the **natural** interval extension $[f]$ of f (i.e., using interval arithmetic for each primitive operator) yields $[-18, 41]$, which is an overestimation of the optimal image.

But the derivative $f'(x) = 3x^2 - 6x + 1$, and $f'([3,4]) = [3,30] > 0$. We deduce that f is monotonic (increasing) in $[x] = [3,4]$. In this case, the image can be optimally obtained at both bounds of $[x]$: $[f]_M([3,4]) = [f(3), f(4)] = [3,20]$.

D.3 The Monotonic Hull-Consistency Algorithm

The MOnotonic Hull-Consistency algorithm (in short `Mohc`) is a new constraint propagation algorithm that exploits monotonicity of functions to better contract a box. The propagation loop is exactly the same AC3-like algorithm performed by the famous `HC4` and `Box`. Its novelty lies in the `Mohc-Revise` procedure handling one contraint individually and described in Algorithm 4.

Algorithm 4 `Mohc-Revise` (**in-out** $[B]$; **in** f, Y, W, ρ_{mohc}, τ_{mohc}, ϵ)

```
HC4-Revise(f(Y,W) = 0, Y, W, [B])
if W ≠ ∅ and ρ_mohc[f] < τ_mohc then
  [G] ← GradientCalculation(f, W, [B])
  (f^og, W) ← OccurrenceGrouping(f, W, [B], [G])
  (f_max, f_min, X, W) ← ExtractMonotonicVars(f^og, W, [B], [G])
  MinMaxRevise([B], f_max, f_min, Y, W)
  MonotonicBoxNarrow([B], f_max, f_min, X, [G], ε)
end if
```

This procedure aims at narrowing the current box $[B]$. It works on a unique equation[1] $f(Y,W) = 0$, in which the variables in Y occur once in the expression f whereas the variables in W occur several times in f.

`Mohc-Revise` starts by a call to `HC4-Revise` (an exception terminating the procedure is raised if an empty box is obtained, proving the absence of solution). If f contains variables with multiple occurrences ($W \neq \emptyset$) and if another condition is fulfilled (see Section D.4.1), then five procedures are called to detect and exploit the monotonicity of f.

The `GradientCalculation` function simply computes the gradient of f. More precisely, it computes the partial derivatives w.r.t. every variable w in W having multiple occurrences.

The `OccurrenceGrouping` function is not required in `Mohc-Revise` and can be viewed as an improvement of it. It rewrites the expression f into a new form f^{og} such that the image $[f^{og}]_M([B])$ computed by the monotonicity-based interval extension is sharper than, or at worst equal to, the image $[f]_M([B]) \subset [f]([B])$. This sophisticated function is briefly introduced in Section D.4.2.

Using the vector $[G]$, for every variable $w \in W$ (with multiple occurrences), the function `ExtractMonotonicVars` checks whether 0 belongs to the partial derivative $\frac{\partial f^{og}}{\partial w}([B])$. If it does not, it means that f^{og} is monotonic w.r.t. w, so that w is removed from W to be added in X. At the end, X contains variables with multiple occurrences that are monotonic ; W contains those that are not detected to be monotonic. (In the following, $[g_i] = \frac{\partial f^{og}}{\partial x_i}([B])$ is the i^{th} component of $[G]$. It denotes the partial derivative on the i^{th} variable x_i in X.) Finally, the function returns

[1] The procedure can be straightforwardly extended to handle an inequality.

the two expressions exploiting, for every variable $x_i \in X$, the monotonicity of f^{og} w.r.t. x_i. f_{min} is the expression f^{og} in which every $[x_i]$ is replaced by $\underline{x_i}$ (resp. $\overline{x_i}$) if f^{og} is increasing (resp. decreasing) w.r.t. x_i. For f_{max}, every $[x_i]$ is replaced by $\overline{x_i}$ (resp. $\underline{x_i}$) if f^{og} is increasing (resp. decreasing) w.r.t. x_i.

The next two routines are in the heart of `Mohc-Revise` and are detailed below. They mainly work with the two functions f_{min} and f_{max}. The procedure `MinMaxRevise` narrows the variables in Y (appearing once in f and thus in f^{og}) and those in W. The procedure `MonotonicBoxNarrow` narrows the monotonic variables in X.

At the end, if `Mohc-Revise` has contracted the interval of a variable in W (more than a user-defined ratio τ_{propag}), then the constraint is pushed into the propagation queue in order to be handled again in a subsequent call to `Mohc-Revise`. Otherwise, we know that a fixpoint in terms of filtering has been reached (under some assumptions). Indeed, nice properties presented in Section D.5 explain why `MinMaxRevise` contracts $[Y]$ optimally while `MonotonicBoxNarrow` contracts $[X]$ optimally. The constraint is thus not pushed into the propagation queue.

Note that a variable y in Y, appearing once in f, is handled by `MinMaxRevise`, and not by `MonotonicBoxNarrow`, *even if it is monotonic*. We have made this choice because `MinMaxRevise` is less costly than `MonotonicBoxNarrow` (see Proposition 8) and has *the same power of filtering on* $[y]$ (see Lemma 2).

D.3.1 The `MinMaxRevise` procedure

Algorithm 5 `MinMaxRevise` (**in-out** $[B]$; **in** f_{max}, f_{min}, Y, W)

`HC4-Revise`$(f_{min}(Y,W) \leq 0, Y, W, [B])$ /* also called `MinRevise` */
`HC4-Revise`$(f_{max}(Y,W) \geq 0, Y, W, [B])$ /* also called `MaxRevise` */

`MinMaxRevise` brings a contraction on variables in Y and W. The monotonicity-based interval evaluation yields $[f]_M([B]) = [\underline{[f_{min}]([B])}, \overline{[f_{max}]([B])}]$, where $[f_{min}]$, resp. $[f_{max}]$, denotes the natural extension of f_{min}, resp. f_{max}.

Checking that $0 \in [f]_M([B])$ ($[f]_M([B]) \subset [f]([B])$) amounts in checking that :

$$[f_{min}]([B]) \leq [0,0] \leq [f_{max}]([B])$$

This inequality is thus split into two parts and each of both is used by `HC4-Revise` to narrow intervals of variables in Y and W. Figure D.1 illustrates how the first part of `Mohc-Revise` narrows the box of the constraint : $x^2 - 3\,x + y = 0$, with $[x] = [4, 10]$ and $[y] = [-80, 30]$.

`HC4-Revise` works in two phases (Fig. D.1-left). The evaluation phase evaluates every node bottom-up and attachs the result to the node. The second phase, due to the equality node, starts by intersecting the top interval $[-94, 118]$ with 0, and proceeds top-down by applying "inverse" (projection) functions. For instance, since $nplus = nminus + y$, the inverse function of this sum yields the difference $[y] \leftarrow [y] \cap [nplus] - [nminus] = [0,0] - [-14, 80] = [-80, 14]$. The symmetric operation on $nminus$ produces $[-14, 80]$, but this narrowing does not allow a

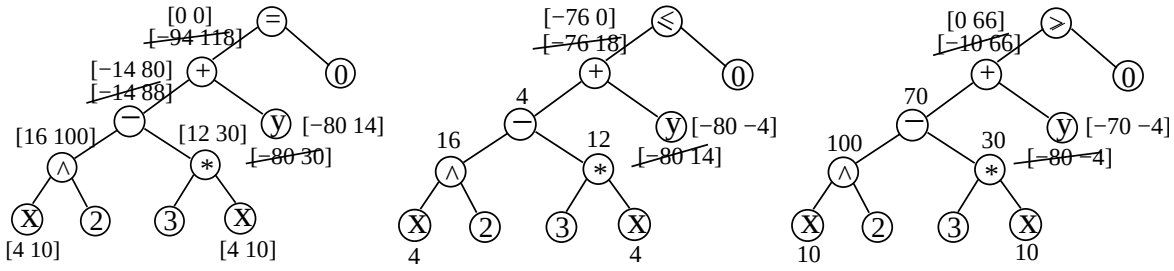


FIG. D.1 – `HC4-Revise` (**Left**), `MinRevise` (**Center**) and `MaxRevise` (**Right**) applied to the equation $x^2 - 3\,x + y = 0$.

contraction lower in the tree, so that $[x]$ is left unchanged. After this step, `OccurrenceGrouping` detects that the function is monotonic w.r.t. x (trivial case when the derivative is positive so that $f = f^{og}$), hence `ExtractMonotonicVars` puts x in the set X of monotonic variables.

Fig. D.1-center shows the first step of `MinMaxRevise`. The inequality $f_{min}(4, y) \leq 0$ is represented by the tree. Calling `HC4-Revise` on this expression produces a new contraction on y because $[x]$ is replaced by $\underline{x} = 4$. On the top of the tree, the narrowing phase intersects $[-76, 18]$ with $[-\infty, 0]$ (inequality), and the first inverse projection function yields $[y] \leftarrow [y] \cap [nplus] - [nminus] = [-76, 0] - [4, 4] = [-80, -4]$. Following the same principle, `MaxRevise` applies `HC4-Revise` to $f_{max}(10, y) \geq 0$ and narrows $[y]$ to $[-70, -4]$ (see Fig. D.1-right).

D.3.2 The `MonotonicBoxNarrow` procedure

Algorithm 6 `MonotonicBoxNarrow` (**in-out** $[B]$; **in** f_{max}, f_{min}, X, $[G]$, ϵ)

for all variable $x_i \in X$ /* x_i contains multiple occurrences and is monotonic */ **do**
  if $\overline{[f_{min}]([B])} < 0$ **and** `applyFmax`$[i]$ **then**
    if $[g_i] > 0$ **then**
      `LeftNarrowFmax`$([x_i], f_{max}^{x_i}, [g_i], \epsilon)$
    else if $[g_i] < 0$ **then**
      `RightNarrowFmax`$([x_i], f_{max}^{x_i}, [g_i], \epsilon)$
    end if
  end if
  if $\underline{[f_{max}]([B])} > 0$ **and** `applyFmin`$[i]$ **then**
    if $\overline{[g_i]} > 0$ **then**
      `RightNarrowFmin`$([x_i], f_{min}^{x_i}, [g_i], \epsilon)$
    else if $[g_i] < 0$ **then**
      `LeftNarrowFmin`$([x_i], f_{min}^{x_i}, [g_i], \epsilon)$
    end if
  end if
end for

The procedure `MonotonicBoxNarrow` (Algorithm 6) aims at narrowing the interval of every monotonic variable x_i in X. Without loss of generality, assume in the following that x_i is increasing. If the condition $\overline{[f_{min}]([B])} < 0$ *and* `applyFmax`$[i]$, detailed in Section D.4.3, is not fulfilled, then the left bound of $[x_i]$ cannot be improved. Otherwise, the `LeftNarrowFmax` procedure is used to improve the left bound of $[x_i]$ with the function $f_{max}^{x_i}$. Also, `RightNarrowFmin` is used to narrow the right bound of $[x_i]$ with the function $f_{min}^{x_i}$ (if $\underline{[f_{max}]([B])} > 0$ *and* `applyFmin`$[i]$). $f_{max}^{x_i}$ and $f_{min}^{x_i}$ denote univariate thick/interval functions depending on x_i. They have similarities with the interval function used in the classical `Box` algorithm. $f_{max}^{x_i}$ and $f_{min}^{x_i}$ are produced by replacing in the f^{og} expression all the variables except x_i with a punctual or an interval value. All the monotonic variables in X, *except* x_i, are replaced with the right bound and all the variables in Y and W are replaced with their current interval in $[B]$.

We detail below how the left bound of $[x_i]$ is improved by the `LeftNarrowFmax` procedure. Note that if `MonotonicBoxNarrow` is called, this requires `MinMaxRevise` be terminated with no failure. This means that `LeftNarrowFmax` will never return an empty interval for $[x_i]$: either $[x_i]$ is left unchanged, or $[x_i]$ is narrowed to a non-empty interval (see cases 1 and 2 in Fig. D.4).

D.3.3 The `LeftNarrowFmax` procedure

This procedure has a close connection with the `LeftNarrow` procedure used by the well-known `Box` algorithm [293, 28]. However, because f is monotonic w.r.t. x_i, the contraction process is faster, that is, it is a true dichotomic, and thus log-time, process.

Algorithm 7 `LeftNarrowFmax` (**in-out** $[x]$; **in** f_{max}^{x}, $[g]$, ϵ)

if $\overline{[f_{max}^{x}](\underline{x})} < 0$ /* test of existence */ **then**
  $[l] \leftarrow [x]$
  $size \leftarrow \epsilon \times$ `Diam`$([l])$
  while `Diam`$([l]) > size$ **do**
    $x_m \leftarrow$ `Mid`$([l])$; $z_m \leftarrow \overline{f_{max}^{x}(x_m)}$ /* $z_m \leftarrow \underline{f_{min}^{x}(x_m)}$ in {`Left|Right`}`NarrowFmin` */
    $[l] \leftarrow [l] \cap x_m - \frac{z_m}{[g]}$ /* Newton iteration */
  end while
  $[x] \leftarrow \left[\underline{l}, \overline{x}\right]$
end if

Let us illustrate `LeftNarrowFmax` (Algorithm 7) applied to the f_{max}^{x} function depicted in Fig. D.2. Starting with $[l] = [x]$, the goal is to narrow the bounds of $[l]$ for providing a tight approximation of the point L, i.e., the new left bound of $[x]$. `LeftNarrowFmax` provides a sharp enclosure of L and keeps only its left bound at the end (last line of Algorithm 7).

A first existence test checks that $\overline{f_{max}^{x}(\underline{x})} < 0$, i.e., the point A in Fig. D.2-left is below zero. Otherwise, a zero occurs in $\underline{x}$ so that $[x]$ cannot be narrowed, leading to an early termination of the procedure.

A dichotomic process is then run until $Diam([l]) < size$. A classical Newton iteration is iteratively launched from the midpoint x_m of $[l]$, e.g., from the point B in Fig. D.2-left, and from the point C in Fig. D.2-right.

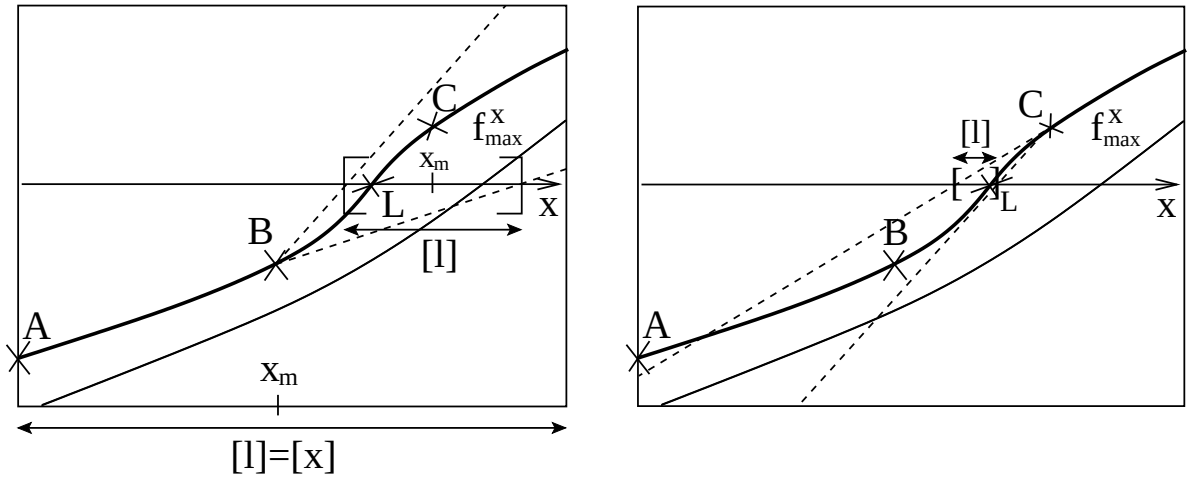


FIG. D.2 – Deux premières itérations de Newton pour contracter la borne gauche de $[x]$.

Graphically, an iteration of a univariate interval Newton [219] intersects $[l]$ with the projection on the x axis of the cone (dotted lines) enclosing all the partial derivatives. Note that the cone forms an angle of at most 90 degrees because the function is monotonic and the derivative $[g]$ is positive[2]. This explains why the diameter of $[l]$ is divided by at least 2 at each iteration. Indeed, if $z_m < 0$ (Fig. D.2-left), then the term $-\frac{z_m}{[g]}$ is positive and the dichotomic process will continue in the right side of x_m. If $z_m > 0$ (Fig. D.2-right), then $-\frac{z_m}{[g]} < 0$ and one will proceed with the left side of x_m.

Lemma 1 *The procedures* `LeftNarrowFmax`*,* `RightNarrowFmin`*,* `LeftNarrowFmin`*,* `RightNarrowFmax` *terminate and run in time* $O(log_2(\frac{1}{\epsilon}))$*, where* ϵ *is a precision expressed as a ratio of interval diameter.*

Finally note that Newton iterations called inside `LeftNarrowFmax` and `RightNarrowFmax` work with $z_m = \overline{f^x_{max}(x_m)}$, that is, a *punctual* curve (in bold in the figure), and not with a thick function. In the same way, `LeftNarrowFmin` and `RightNarrowFmin` work with $z_m = \underline{f^x_{min}(x_m)}$.

D.4 Advanced features of `Mohc-Revise`

D.4.1 The user-defined parameter τ_{mohc} and the array ρ_{mohc}

The user-defined parameter $\tau_{mohc} \in [0, 1]$ allows the monotonicity-based procedures to be called more or less often (see Algorithm 4). For every constraint, $\rho_{mohc}[f]$ tries to predict whether the monotonicity-based treatment that follows is promising : the procedures exploiting monotonicity

[2]Every Newton iteration could recompute a tighter partial derivative $[g]$ although we do not perform it in our implementation.

are called only if $\rho_{mohc}[f] < \tau_{mohc}$. These ratios are computed in a preprocessing procedure called after every bisection (i.e., choice point) on the current box $[B]$, as follows :

$$\rho_{mohc}[f] = \frac{Diam([f]_M([B]))}{Diam([f]([B]))} = \frac{Diam([\underline{[f_{min}]([B])}, \overline{[f_{max}]([B])}])}{Diam([f]([B]))}$$

This ratio thus indicates whether the monotonicity-based image of a function is sufficiently sharper than the natural one. As confirmed by the experiments detailed in Section D.6, this ratio is relevant for the bottom-up evaluation phases of `Minrevise` and `Maxrevise`, and also for `MonotonicBoxNarrow` in which a lot of evaluations are performed.

The experiments below show that the parameter τ_{mohc} is useful when `Mohc` is a sub-contractor of `3BCID` [283]. In this case, `Mohc` is called many times between two branching points, so that the CPU time required by the preprocessing procedure filling the array ρ_{mohc} is negligible. This trend would also be true for any other sophisticated contractor calling a constraint propagation sub-contractor, such as `3B` [186], `Quad` [183] or `Box-k` [12]. Future experiments will confirm if tuning a parameter τ_{mohc} is still useful when `Mohc` is called only once between two branching points.

D.4.2 The `OccurrenceGrouping` function

The motivation is the following. If $0 \in \frac{\partial f}{\partial x}([B])$, then one cannot deduce that x is monotonic. However, it is possible that there exists a subgroup of the occurrences of x, which are replaced by a new auxiliary variable x_a, such that $\frac{\partial f}{\partial x_a}([B]) \geq 0$. Also, another subgroup of occurrences, which are replaced by a new auxiliary variable x_b, may verify $\frac{\partial f}{\partial x_b}([B]) \leq 0$. In this case, such an occurrence grouping rewrites the expression f into a new form f^{og} such that the image $[f^{og}]_M([B])$ computed by the monotonicity-based interval extension is sharper than, or at worst equal to, the image $[f]_M([B]) = [f]([B])$.

This problem can be formalized by a linear program and solved by a specific algorithm `OccurrenceGrouping` in time $O(k\, log_2(k))$ for each variable x occurring k times in a function f. Details can be found in the companion paper [10].

Consider the function $f_1(x) = -x^3 + 2x^2 + 6x$, with $[x] = [-1.2, 1]$. f_1 is not detected monotonic since the image of $[-1.2, 1]$ by natural evaluation of the derivative $f_1'(x) = -3x^2 + 4x + 6$ contains 0. `OccurrenceGrouping` produces :
$f_1(x) = f_1^{og}(x_a, x) = -x_a^3 + 2(0.35x_a + 0.65x)^2 + 6\, x_a$. We can observe that :
$[f_1^{og}]_M([x]) = [-5.472, 7] \subseteq [f_1]_M([x]) = [-8.2, 10.608]$.

At the end, `OccurrenceGrouping` returns the new expression f^{og} and a new set W that includes the new variables x_a and x_b, if any. (`ExtractMonotonicVars` tranfers x_a and x_b into the set X of monotonic variables.)

D.4.3 Avoiding calls to the dichotomic process

This section gathers several features that have been introduced in `MonotonicBoxNarrow` to avoid calls to `LeftNarrowFmax` (and symmetric procedures).

First condition

We first depict in Figure D.3 the increasing functions $f_{max}^{x_i}$ and $f_{min}^{x_i}$ enclosing f^{og}. The function between the two curves in bold is sharper than the natural enclosure used in the standard `BoxNarrow` because the monotonic variables in $X \setminus \{x_i\}$ are replaced by points. However, it is still an overestimation of the optimal function because of the multiple occurrences of variables in W.

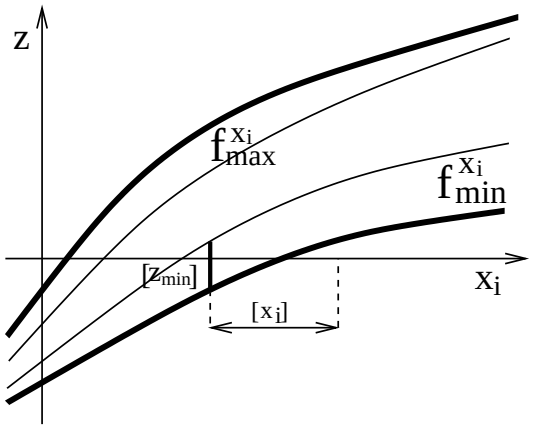


FIG. D.3 – Monotonic functions $f_{max}^{x_i}$ and $f_{min}^{x_i}$

$f_{max}^{x_i}$ is above $f_{min}^{x_i}$. Note that both functions can overlap in the general case although the upper function $\overline{f_{max}^{x_i}}$ (in bold) is always above $f_{min}^{x_i}$.

We have represented on the figure an initial value for $[x_i]$, before a call to `LeftNarrowFmax`. The figure justifies the condition $\overline{[f_{min}]([B])} \geq 0$ in Algorithm 6 that avoids the call to `LeftNarrowFmax`. Let $[z_{min}]$ be $[f_{min}]([B])$ (vertical segment in bold on the figure). It immediately shows that since $\overline{z_{min}} \geq 0$, $\underline{x_i}$ is solution, so that the lower bound of $[x_i]$ cannot be improved.

Arrays `applyFmin` and `applyFmax`

For every monotonic variable $x_i \in X$, the booleans `applyFmin[i]` and `applyFmax[i]` are initialized to *true*. They are updated in the four procedures `LeftNarrowFmax`, `RightNarrowFmax`, `LeftNarrowFmin` and `RightNarrowFmin`. With no loss of generality, let us explain the principle assuming that f is increasing w.r.t. x_1 ($x_1 \in X^+$; X^+ denotes the set of increasing variables ; X^- denotes the set of decreasing variables ; $\overline{X^+}$ denotes the vector in which every $x_i \in X^+$ is replaced with the upper bound $\overline{x_i}$.) Since x_1 is increasing, `LeftNarrowFmax` is called to narrow

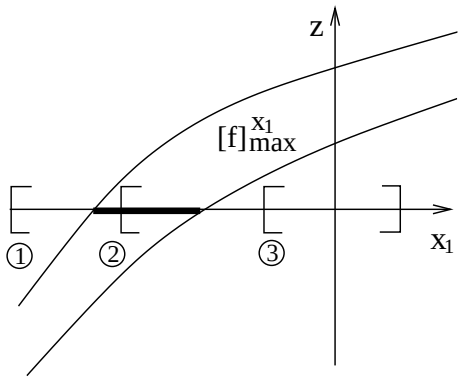


FIG. D.4 – Three possible cases for the left bound of $[x_1]$.

the left bound of $[x_1]$. Three cases may occur according to $\underline{x_1}$, as illustrated in Fig. D.4.

In the cases 1 and 2, and *not* in the case 3, there is a point $v_1 \in [x_1]$ which is a zero (i.e., which belongs to the segment in bold). That is : $\exists v_1 \in [x_1]$ *s.t.* $0 \in [z_1]$, with : $[z_1] = f^{x_1}_{max}(v_1) = [f](v_1, \overline{X^+ \setminus \{x_1\}}, \underline{X^-}, [Y], [W])$.

Cases 1 and 2 are easily detected at the first line of `LeftNarrowFmax`, by slightly complexifying the existence test. The key point is that the relation $0 \in [z_1]$ implies :

$$\forall i \neq 1, \forall v_i \in [x_i] : \underline{z_i} \leq \underline{z_1} \leq 0$$

With[3] :

$$[z_i] = [f](v_i, \underline{X^+ \setminus \{x_i\}}, \overline{X^-}, [Y], [W]) = f^{x_i}_{min}(v_i)$$

The rightmost inequality comes directly from $0 \in [z_1]$. The leftmost inequality comes from the study of monotonicity of f : $v_i \leq \overline{x_i}$ and the bounds of X selected in $[z_1]$ maximize f while the bounds of X selected in $[z_i]$ minimize f.

Thus, in cases 1 and 2, we know that $\underline{f^{x_i}_{min}(v_i)} \leq 0$. This means that for every $i \neq 1$, $f^{x_i}_{min}$ cannot bring any additional narrowing to $[x_i]$, the relation used to shave the interval being always true, even for $v_i = \overline{x_i}$ (if x_i is increasing). In other terms, if x_i is increasing (resp. decreasing), then it is useless to call `RightNarrowFmin` (resp. `LeftNarrowFmin`) to contract $[x_i]$. That is why, in the cases 1 or 2, for all $i \neq 1$, `applyFmin[i]` is set to *false*.

This advanced feature is a slight generalization of the feature proposed by Jaulin and Chabert in [146]. In their paper, $f^{x_i}_{min}$ and $f^{x_i}_{max}$ are punctual functions because there are no sets Y and W.

[3] If x_i is decreasing, then $[z_i] = [f](v_i, \underline{X^+}, \overline{X^- \setminus \{x_i\}}, [Y], [W])$.

D.4.4 Lazy evaluations of f_{min} and f_{max}

The implemented `MohcRevise` procedure is in fact optimized in the aim of reusing as much as possible the different evaluations of f_{min} and f_{max}. In Algorithm 6, intervals $[z_{min}] = [f_{min}]([B])$ and $[z_{max}] = [f_{max}]([B])$ do not need be recomputed at each iteration. Furthermore, these intervals have been previously computed in the bottom-up evaluation phases of `MinMaxRevise` and can be transmitted from a procedure to the other.

The value $\overline{z_{max}}$ can also be used to add a first and cheap call to a Newton iteration at the very beginning of `LeftNarrowFmax` : $[x] \leftarrow [x] \cap \overline{x} - \frac{\overline{z_{max}}}{[g]}$.

Finally, the existence test of `LeftNarrowFmax` makes it possible to reuse the value $[z_l] = [f^x_{max}](\underline{x})$. This allows us to add, just after the existence test, a second cheap Newton iteration : $[x] \leftarrow [x] \cap \underline{x} - \frac{\overline{z_l}}{[g]}$.

D.4.5 The `LazyMohc` variant

`LazyMohc` is a simplified version of `Mohc` in which the `MonotonicBoxNarrow` procedure only calls the first and cheap Newton iteration described above for every bound of x_i in X. In other words, the `LazyMohc` variant runs `MinMaxRevise`, two cheap Newton iterations per monotonic variable, but no dichotomic process.

D.5 Properties

A very interesting property concerning `Mohc` is that the `Mohc-Revise` procedure can compute an optimal box w.r.t. an individual constraint if certain conditions are fulfilled. These conditions thus identify a polynomial subclass (Propositions 6 and 7) of the hull-consistency problem (i.e., searching for an optimal box) which is difficult when there exist multiple occurrences of variables. The corresponding propositions appear below and the proofs can be found in appendix.

Proposition 6 *Let $c : f(X, Y, W) = 0$ be a constraint such that f is continuous and differentiable w.r.t. every variable in the box $[B]$. Variables in X and W appear several times in f while variables $y_i \in Y$ appear once. For every $x_i \in X$, f is monotonic w.r.t. x_i.*

If W is empty and if for all $y_i \in Y$, $0 \notin \frac{\partial f}{\partial y_i}([B])$ (implying that y_i is monotonic), then :
One call to `Mohc-Revise` *computes the hull-consistency of the constraint c in $[B]$ (with a precision ϵ).*

The proof is based on the lemma 2 related to the intervals $[Y]$ contracted by `MinMaxRevise`, and on the lemma 3 related to the intervals $[X]$ contracted by `MonotonicBoxNarrow`.

Lemma 2 *With the same hypotheses as in Proposition 6, if W is empty and if for all $y_i \in Y$, $0 \notin \frac{\partial f}{\partial y_i}([B])$ (implying that y_i is monotonic), then :*

One call to `MinMaxRevise` *contracts optimally every $[y_i] \in [Y]$.*

Lemma 3 *With the same hypotheses as in Proposition 6 :*

One call to `MonotonicBoxNarrow` *(following a call to* `MinMaxRevise`*) contracts optimally every* $[x_i] \in [X]$.

Proposition 7 is in a sense stronger than Proposition 6 because *no monotonicity hypothesis is required for the variables* y_i *occurring once* in the expression. However, a stronger and more expensive procedure is used instead of `HC4-Revise`. Replacing `HC4-Revise` with a so-called `TAC-revise` is a way to make a system hull-consistent when all the constraints contain only variables with single occurrence. The fact that a given function f, having only variables with single occurrence, is continuous ensures that the image produced by the natural extension $[f]$ is optimal. But the top-down narrowing phase manages in a sense inverse functions of f that are not necessarily continuous. `HC4-Revise` returns the hull of the different continuous subparts provided by the piecewise analysis performed at each node for the inverse functions. Instead, `TAC-revise` combinatorially combines the continuous subparts of different nodes for narrowing optimally the variables, thus achieving hull-consistency. Details about `TAC-revise` can be found in [53].

Proposition 7 *Let us call* `Mohc-Revise'` *a variant of* `Mohc-Revise` *in which* `HC4-Revise` *is replaced by* `TAC-revise`*. Let us consider the same hypotheses as in Proposition 6. Then :*

If W *is empty, one call to* `Mohc-Revise'` *computes the hull-consistency of the constraint* c *in* $[B]$ *(with a precision* ϵ*).*

This proposition is significant because in practice, after only a few bisections in the search tree, `HC4-Revise` generally computes a box as sharp as `TAC-revise` does, except in pathological cases. Thus, `Mohc-Revise` (calling the standard `HC4-Revise` procedure) *often* computes hull-consistency when all the variables appear once or are monotonic.

Finally, Proposition 8 details the time complexity of `Mohc-Revise`.

Proposition 8 *Let* c *be a constraint. Let* n *be its number of variables,* e *be the number of nodes in its expression tree (i.e., twice its number of unary and binary operators), and* k *be the maximum number of occurrences of a variable. Let* ϵ *be the precision expressed as a ratio of interval diameter.*

`Mohc-Revise` *is time* $O(n\,(e\,log_2(\frac{1}{\epsilon})+k\,log_2(k)))$. `LazyMohc-Revise` *is time* $O(e+n+n\,k\,log_2(k))$.

D.6 Experiments

We have implemented `Mohc` and `LazyMohc` with the interval-based `C++` library `Ibex` [52, 50]. `Mohc` has been tested on 17 benchmarks with a finite number of zero-dimensional solutions issued from COPRIN's web page [204]. In the presented experiments, all the `Mohc`-based solving strategies embed `Mohc` as a sub-contractor of `3BCID` [283].

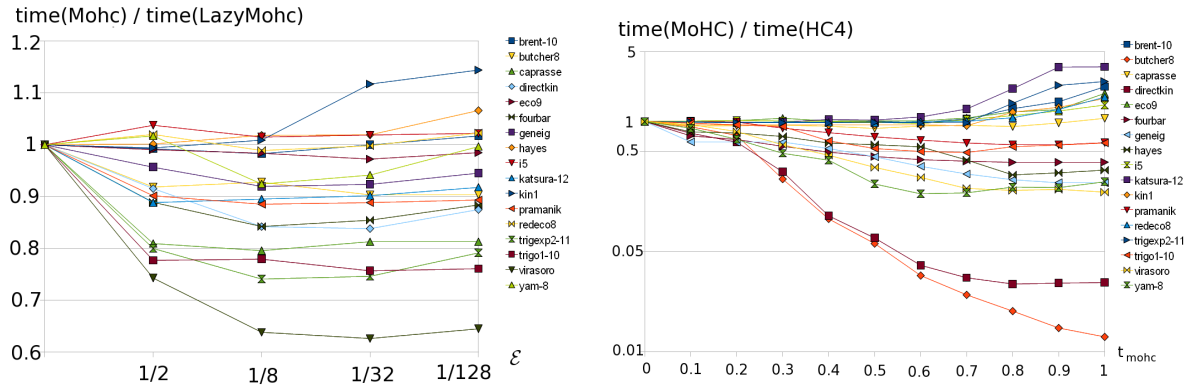


FIG. D.5 – Tuning the user-defined parameters. **Left :** Tuning ϵ. For every benchmark, the curves shows how a ratio $\frac{Time(Mohc)}{Time(LazyMohc)}$ evolves when ϵ decreases (i.e., the reached precision in `MonotonicBoxNarrow` increases). **Right :** Tuning τ_{mohc}. For every benchmark, the curve shows how a ratio $\frac{Time(Mohc)}{Time(HC4)}$ evolves when τ_{mohc} increases.

D.6.1 Tuning the user-defined parameters

The first experiments allow us to get an idea of relevant values for τ_{mohc} and ϵ. Fig. D.5-left is interesting because it shows that finely tuning ϵ has no significant impact on performance. For most of the NCSPs, the best value falls between $\frac{1}{32}$ and $\frac{1}{8}$, and the curves are rather flat. We have thus fixed ϵ to 10% (at least when `Mohc` is a sub-contractor of `3BCID`).

Fig. D.5-right shows that, for all the NCSPs, the best value falls between 0.6 and 0.99. [4] More precisely, even on NCSPs that do not benefit from our monotonicity-based procedures, setting τ_{mohc} to 0.6 only slightly decreases the performance (only 11% in the worst case, i.e., `Katsura`). That is why the experiments that follow perform two trials with $\tau_{mohc} = 0.7$ and $\tau_{mohc} = 0.99$ (for the most favorable NCSPs).

D.6.2 Experimental protocol

We have selected all the NCSPs with multiple occurrences of variables found in the first two sections (polynomial and non polynomial systems) of the Web page. We have added `Brent`, `Butcher`, `Direct Kinematics` and `Virasaro` from the section describing the *difficult problems*. All the competitors are also available in `Ibex`, thus making the comparison fair.

Our `Mohc`-based solving strategy uses a round-robin variable selection. Between two branching points, three procedures are called in sequence. First, a monotonicity test just checks whether every monotonicity-based function evaluation contains zero. Its specificity is that the test does not apply to the initial functions f in the NCSP, but to the functions f^{og} produced by `OccurrenceGrouping`. Second, the contractor `3BCID(Mohc)` is called. Third, an interval Newton is run if the current box has a diameter 10 or less. All the parameters in `3BCID`, `HC4`, `Box` and

[4] Although not shown on curves, the value 0.99 seems always better than the value 1.0...

Mohc (except τ_{mohc}) have been fixed to default values. The precision ratio in 3B and Box is 10% ; the number of additional slices handled by the CID part is 1 ; a constraint is pushed into the propagation queue if the interval of one of its variables is reduced more than $\tau_{propag} = 10\%$.

All the experiments have been performed on an Intel 6600 2.4 GHz, and a timeout of at least one hour has been chosen for each benchmark.

D.6.3 Results

Table D.1 compares the CPU time and number of choice points obtained by Mohc with those obtained by competitors : HC4 and Box.

The table reports very good results obtained by Mohc, both in terms of filtering power (low number of choice points) and CPU time. As expected, the bad results obtained by Box highlight that Box is not relevant when several variables in a same contraint have multiple occurrences. For the NCSPs Yamamura1, Brent and Kin1, Box shows a slightly better contraction power than Mohc, but it does not pay off in terms of performance. This underlines that it is better to perform a box narrowing effort less often, when monotonicity has been detected for a given variable.

The comparison with HC4 is more interesting. Mohc and HC4 obtain similar results on 8 of the 17 benchmarks. With $\tau_{mohc} = 0.7$, note that the loss in performance w.r.t. HC4 is negligible. It is inferior to 5%, except for Redeco8 (9%) and Katsura (25%). On 6 NCSPs, Mohc shows a gain comprised between 2.7 and 7.9. On Butcher, Direct Kinematics and Fourbar, a very good gain in CPU time of resp. 153, 48 and 37 is observed.

Although the difference is not so significant, Mohc generally provides better results than LazyMohc, in particular when $\tau_{mohc} = 0.99$.

D.7 Related Work

The first constraint propagation algorithm exploiting monotonicity appears in the interval-based solver ALIAS [201] developed by Merlet. It is a 2B-consistency algorihm that is improved when the "*GradientSolve*" option is selected to compute and exploit the gradient of functions appearing in constraints. Every projection function f^o_{proj} used to narrow a given occurrence o is first explicitly and symbolically generated (and simplified) before calling on it a procedure close to ExtractMonotonicVars (contrarily to HC4-Revise, no expression tree is used). The monotonicity-based evaluation $[f^o_{proj}]_M$ then brings a better contraction on $[o]$ than $[f^o_{proj}]$ would do.

Monotonicity of functions has also be used in quantified NCSPs to easily contract a universally quantified variable that is monotonic [106].

Jaulin and Chabert propose in [146] a constraint propagation algorithm called Octum. Note that Octum and Mohc have been initiated independently in the first semester of 2009. The most common part is MonotonicBoxNarrow that is nearly the same in both algorithms. We have already mentioned in Section D.4.3 that our arrays applyFmin and applyFmax are inspired by Octum

TAB. D.1 – Results obtained by `Mohc`. The first column includes the name of the benchmark; the bottom of the cell contains the corresponding number of equations and the number of solutions. The other columns report the results obtained by different algorithms. Every cell shows the CPU time in second (above) and the number of choice points (below). The contraction algorithms are `3BCID(HC4)` (column `HC4`), `3BCID(Box)` (column `Box`), `MonoTest` followed by `3BCID(HC4)` (column `MonoTest`). `MonoTest` is also used before the contractor shown in the four last columns : `3BCID(LazyMohc)` with $\rho_{mohc} = 0.7$ (column Lazy(0.7)), the same with $\rho_{mohc} = 0.99$ (column Lazy(0.99)), `3BCID(Mohc)` with $\rho_{mohc} = 0.7$ and $\epsilon = 10\%$ (column Mohc(0.7)), the same with $\rho_{mohc} = 0.99$ (column Mohc(0.99)). All the algorithms are followed by a call to interval Newton before the next bisection. The last column yields the gain obtained by `Mohc`, i.e., $\frac{Time(3BCID(HC4)\ based\ strategy)}{Time(3BCID(Mohc)\ based\ strategy)}$ (the denominator is given by the best of the previous two columns).

NCSP	HC4	Box	MonoTest	Lazy(0.7)	Lazy(0.99)	Mohc(0.7)	Mohc(0.99)	Gain
Butcher	282528	25867	281664	5221	1986	5026	**1842**	153
8 3	1.8e+8	1.7e+6	1.8e+8	2.2e+6	346063	2.2e+6	**324669**	554
Direct kin.	17515	>28800	17507	481	431	458	**363**	48
11 2	1.4e+6		1.4e+6	11931	8811	9541	**5609**	250
Fourbar	13121	11011	1069	429	420	366	**353**	37
4 3	8.5e+6	732429	965343	79697	67397	58571	**45695**	186
Virasoro	7158	>28800	7173	1538	1413	1241	**902**	7.9
8 224	2.6e+6		2.6e+6	135833	102997	79211	**38739**	67
Geneig	598	>7200	390	117	95.2	116	**87.6**	6.8
6 10	205859		161211	17059	9083	15341	**6975**	30
Yamam.1	11.8	15.3	11.7	2.26	2.91	**2.02**	2.69	5.8
8 7	3017	**183**	3017	357	345	303	297	10
Pramanik	95.9	278	35.9	21.9	22.1	19.6	**19.6**	4.9
3 2	124661	23017	69259	13817	9649	12691	**8435**	15
Hayes	41.7	282	41.6	17.2	**13.6**	17.3	13.9	3.0
8 1	17763	7247	17763	4447	**1707**	4437	1717	10
Trigo1	150.7	773	151	74.0	92.3	**55.8**	71.9	2.7
10 9	2565	1005	2565	641	603	461	455	5.6
Caprasse	2.77	32.2	2.73	2.51	2.94	2.74	**2.34**	1.18
4 18	1309	719	1309	951	499	903	**391**	3.3
Kin1	1.95	68.7	1.96	**1.78**	3.33	1.97	3.36	0.99
6 16	87	**65**	87	83	83	87	81	1.1
Trigexp2	90.1	>3600	90.9	**88.2**	228	91.4	169	0.99
11 0	15187		15187	14303	12567	15099	7717	2.0
I5	**55.7**	>3600	55.9	58.4	81.7	58.5	82.9	0.95
10 30	10621		10621	9809	8849	9811	**8715**	1.2
Eco9	**13.9**	102.0	**13.9**	15.1	26.5	14.6	26.0	0.95
9 16	6193	4991	6193	6047	4707	6037	**4343**	1.4
Brent	19.0	311.0	**18.9**	20.0	42.1	20.2	41.4	0.94
10 1008	3923	**2137**	3923	3807	3341	3815	3189	1.2
Redeco8	**6.23**	69.8	6.28	6.32	11	6.82	10.88	0.91
8 8	2441	1913	2441	2231	1741	2347	**1537**	1.6
Katsura	**77.4**	2265	77.8	104	274	103	245	0.75
12 7	4251	3557	4251	3671	3373	3573	**3151**	1.3

and have been adapted to the general case (functions having as well non monotonic variables appearing once (Y) or several times (W)). Compared to Jaulin and Chabert's algorithm, `Mohc` presents additional features :

- `Mohc` exploits monotonicity of one function in one variable interval once the box becomes sufficiently small to make appear this property. `Octum` requires a function be monotonic in *all* its variable intervals simultaneously.
- Occurrence grouping quickly rewrites the constraint expressions in order to detect more cases of monotonicity.
- Contrarily to `Octum`, `Mohc` uses the `MinMaxRevise` function to quickly contract the intervals of variables occurring once (Y) and of those which are not monotonic (W). Due to Proposition 7, `Mohc` does not need to call the more costly `MonotonicBoxNarrow` procedure to handle *monotonic variables that appear once* in the expression.

D.8 Conclusion

This paper has presented a new interval constraint propagation algorithm exploiting the monotonicity of functions. Using ingredients present in the existing `HC4-Revise` and `BoxNarrow`, `Mohc` has the potential to replace advantageously `HC4` and `Box`, as shown by our first experiments. To confirm this claim, we need to also conduct our experiments with strategies using only `Box`, `HC4` or `Mohc`, i.e., without `3B` or `3BCID`.

D.9 Proofs

D.9.1 Proof of Lemma 2

First recall that in every node of the expression tree T_{fmax} (resp. T_{fmin}) representing f_{max} (resp. f_{min}), there is no overestimation due to the variables in X because they are replaced by points. The proof of Lemma 2 requires us prove that two traversals of T_{fmax} (with `HC4-Revise`) is sufficient to reach a fixpoint in contraction on variables $y_j \in Y$ occurring once in T_{fmax}. A second proof must ensure that a second call to `MinMaxRevise` following the call to `MonotonicBoxNarrow` would not bring additional contraction to $[y_j]$.

Only two traversals of T_{fmax}

Since f is a continuous function and, for every $y_j \in Y$, $0 \notin \frac{\partial f}{\partial y_j}([B])$, then the (bottom-up) evaluation or (top-down) narrowing functions g called in *every* node during `HC4-Revise` of T_{fmin} (or T_{fmax}) are continuous and monotonic in every of their arguments intervals. This comes from the rule of composition of functions stating $\frac{\partial f}{\partial y_j}([B]) = \frac{\partial f}{\partial g}(g([B])) \times \frac{\partial g}{\partial y_j}([B])$. Since the

multiplication preserves the 0 in the resulting interval, $0 \notin \frac{\partial f}{\partial y_j}([B])$ implies that $0 \notin \frac{\partial f}{\partial g}(g([B]))$ and $0 \notin \frac{\partial g}{\partial y_j}([B])$. The same rule applies to two nodes g_1 and g_2 for which g_2 is an argument of g_1 : $\frac{\partial f}{\partial g_2}([B]) = \frac{\partial f}{\partial g_1}(g_1([B])) \times \frac{\partial g_1}{\partial g_2}([B])$. The same reasoning yields that $0 \notin \frac{\partial g_1}{\partial g_2}([B])$. Thus, every bottom-up evaluation function g is monotonic in every of their arguments intervals. The same reasoning applies to the top-down narrowing functions g since an "inverse" function of a monotonic (continuous) function is also monotonic.

Because every node function g in T_{fmin} (or T_{fmax}) is monotonic, g computes an arc-consistent output interval $[z_g]$, i.e., every real number $z \in [z_g]$ has a support in the input intervals. Since T_{fmin} (or T_{fmax}) is a tree, the two traversals of T_{fmin} performed by `HC4-Revise` then optimally narrow every $y_j \in Y$. This is an application [81] of a result by Freuder [86] concerning the finite-domain CSPs stating that two (directed) arc-consistent traversals of an acyclic CSP makes it globally-consistent. That is, no propagation loop is necessary and the two traversals are sufficient to reach the fixpoint in filtering.

No impact of `MonotonicBoxNarrow`

During `MinMaxRevise`, every $x_i \in X$ is replaced by one of its bound $\overline{x_i}$ or $\underline{x_i}$, although these bounds are not optimal before the call to `MonotonicBoxNarrow`. Fortunately, it turns out that the points picked during `MinMaxRevise` inside every $[x_i]$ have no impact on the contraction brought to any $[y_j]$. Let us detail this point on T_{fmax} (the same reasoning holds for T_{fmin}) and with a variable x_i being increasing. Before the call to `MinMaxRevise`, since x_i is increasing, it is replaced by $\overline{x_i}$ in T_{fmax}. The question becomes : if `MonotonicBoxNarrow` reduces $\overline{x_i}$ to a better value $\overline{x_i'}$, cannot a second call to `MinMaxRevise` with $\overline{x_i'}$ bring a new contraction on $[y_j]$?

During `MaxRevise`, at the end of the bottom-up evaluation phase of T_{fmax}, the root interval is $[z] = [z_l, z_u]$. Three cases may occur according to the signs of z_l and z_u. Fig. D.6 illustrates the two interesting cases.

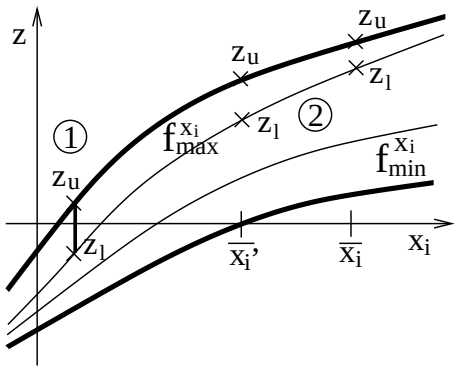


FIG. D.6 – Two main cases in `Mohc-Revise` when `MaxRevise` is applied to f_{max}.

The case 1 (left side of Fig. D.6) occurs when $z_l < 0 < z_u$, resulting in $[z'] = [0, z_u]$ after intersection of $[z]$ with $[0, +\infty]$ in the top of T_{fmax} (because $[f_{max}]([B]) \geq 0$), and thus in

a potential contraction of $[y_j]$. However, `MonotonicBoxNarrow` cannot contract $[x_i]$ with $f_{min}^{x_i}$ (i.e., $\overline{x_i'} = \overline{x_i}$) since $\overline{x_i}$ is already a zero. This explains the condition $(\underline{z_l} =)\underline{[f_{max}]([B])} > 0$ in Algorithm 6.

The case 2 (right side of Fig. D.6) occurs when $0 < z_l < z_u$, resulting in $[z'] = [z_l, z_u] = [z]$ after intersection of $[z]$ with $[0, +\infty]$. Since $[z]$ is not narrowed, no $[y_j]$ is contracted in the top-down narrowing phase of $T_{f_{max}}$. Next, `MonotonicBoxNarrow` reduces $\overline{x_i}$ to $\overline{x_i'}$, but $0 < z_l' < z_u'$ remains true (see figure). Thus, a second call to `MinMaxRevise` could not contract further $[y_j]$.

These cases highlight the duality of the contraction process.

A last trivial case occurs when $z_l < z_u < 0$. `MinMaxRevise` detects that there is no solution due to an empty intersection with $[0, +\infty]$. (Any other smaller values for z_l and z_u, due to $\overline{x_i} \to \overline{x_i'}$, would imply the same failure.) □

D.9.2 Proof of Lemma 3

First recall that `MonotonicBoxNarrow` cannot result in an empty box (no solution) because `MinmaxRevise` has been called before with success. This precondition implies that for any bound of each $x_i \in X$, only the three cases depicted in Fig. D.4 may occur. When the case 1 occurs, the dichotomic narrowing process performed by `LeftNarrowFmax` (or symmetric procedures) converges onto a final interval $[l]$ including surely a zero of $f_{max}^{x_i}$. Contrarily to the classical `LeftNarrow` procedure used by `Box` [293, 28], $[l]$ surely contains the zero because the $f_{max}^{x_i}$ evaluations lead to no overestimation (see proof of Lemma 2). When the cases 2 or 3 occur (see Fig. D.4), $[x_i]$ cannot be (left) narrowed because the bound is a zero of the function.

After only one loop on each variable $x_i \in X$ $(i \in 1, ..., n)$, each of the $2 \times n$ bounds of $[x_i]$ are optimal either because computed by the dichotomic process (case 1 explained above), or because `applyFmin([i])=true` (or `applyFmax([i])=true`), which ensures that no further reduction is expected in these cases 2 or 3 (see Section D.4.3). □

D.9.3 Proof of Proposition 6

Proposition 6 follows Lemmas 2 and 3. One call to `MinMaxRevise` and one loop on the monotonic variables in X ensure that hull-consistency is achieved. □

D.9.4 Proof of Proposition 7

The lemma 3 also holds for Proposition 7, but the lemma 4 must replace the lemma 2.

Lemma 4 *With the same hypotheses as in Proposition 7, one call to* `MinMaxRevise'` *(calling* `TAC-revise`*) contracts optimally every* $[y_j] \in [Y]$.

The second part of the proof of the lemma 4 follows that of the lemma 2. The first part of the proof is based on the correctness of `TAC-revise`.

Remark

We assume in Proposition 7 that f is continuous in the current box. This hypothesis can be relaxed provided that the bottom-up expression evaluations are performed by a "`TAC-eval`" procedure that would combinatorially combine the continuous parts extracted in the different nodes of the expression tree.

D.9.5 Proof of Proposition 8 (time complexity)

Calls to `HC4-Revise` (two traversals of an expression tree), `GradientCalculation` (computing all the partial derivatives also amounts in two tree traversals) and `MinMaxRevise` are $O(e)$. A call to `OccurrenceGrouping` [12] is $O(n\,k\,log_2(k))$. The complexity of `MonotonicBoxNarrow` is time $O(n\,e\,log_2(\frac{1}{\epsilon}))$. One Newton iteration takes $O(e)$ (one function evaluation, plus one gradient calculation). The maximum number of possible slices in one interval $[x_i]$ is $\frac{1}{\epsilon}$. The number of Newton iterations is $O(log_2(\frac{1}{\epsilon}))$ (see Lemma 1). In `LazyMohc-Revise`, `MonotonicBoxNarrow` is time $O(n)$ because are called only two constant-time Newton iterations per interval.

Overall, the time complexity of `Mohc-Revise` is $O(e)$ plus $O(n\,k\,log_2(k))$, plus $O(n\,e\log(\frac{1}{\epsilon}))$, or $O(n)$. □

Annexe E

A New Monotonicity-Based Interval Extension Using Occurrence Grouping

Article [10] : paru au workshop IntCP du congrès CP, Constraint Programming, en septembre 2009

Auteurs : Ignacio Araya, Bertrand Neveu, Gilles Trombettoni

Abstract

When a function f is monotonic w.r.t. a variable in a given domain, it is well-known that the monotonicity-based interval extension of f computes a sharper image than the natural interval extension does.

This paper presents a so-called "occurrence grouping" interval extension $[f]_{og}$ of a function f. When f is *not* monotonic w.r.t. a variable x in the given domain $[B]$, we try to transform f into a new function f^{og} that is monotonic in two subsets x_a and x_b of the occurrences of x. f^{og} is increasing w.r.t. x_a and decreasing w.r.t. x_b. $[f]_{og}$ is the interval extension by monotonicity of f^{og} and produces a sharper interval image than the natural extension does.

For finding a good occurrence grouping, we propose an algorithm that minimizes a Taylor-based overestimation of the image diameter of $[f]_{og}$. Finally, experiments show the benefits of this new interval extension for solving systems of equations.

E.1 Introduction

The computation of sharp interval image enclosures is in the heart of interval arithmetics. It allows a computer to evaluate a mathematical formula while taking into account in a reliable way round-off errors due to floating point arithmetics. Sharp enclosures also allow interval methods to quickly converge towards the solutions of a system of constraints over the reals. At every node of the search tree, a *test of existence* checks that, for every equation $f(X) = 0$, the interval extension of f returns an interval including 0 (otherwise the branch is cut). Also, constraint propagation algorithms can be improved when they use better interval extensions. For instance, the Box algorithm uses a test of existence inside its iterative splitting process [28].

This paper proposes a new interval extension and we first recall basic material about interval arithmetics [211, 219, 147] to introduce the interval extensions useful in our work.

An interval $[x] = [a, b]$ is the set of real numbers between a and b. $\underline{[x]}$ denotes the minimum of $[x]$ and $\overline{[x]}$ denotes the maximum of $[x]$. The diameter of an interval is : $diam([x]) = \overline{[x]} - \underline{[x]}$, and the absolute value of an interval is : $|[x]| = max(|\overline{[x]}|, |\underline{[x]}|)$. A Cartesian product of intervals is named a *box*, and is denoted by $[B]$ or by a vector $\{[x_1], [x_2], ..., [x_n]\}$.

An interval function $[f]$ is a function from $\mathbb{IR}$ to $\mathbb{IR}$, $\mathbb{IR}$ being the set of all the intervals over $\mathbb{R}$. $[f]$ is an *interval extension* of a function f if the following condition is verified :

– The image $[f]([x])$ must be a *conservative* interval containing the set $\mathcal{I}f([x]) = \{y \in \mathbb{R}, \exists x \in [x], y = f(x)\}$. The computation of the image is called *evaluation* of f in this article.

We can extend this definition to functions with several variables, as follows :

– Let $f(x_1, ..., x_n)$ be a function from $\mathbb{R}^n$ to $\mathbb{R}$ and let box $[B]$ be the vector of intervals $\{[x_1], [x_2], ..., [x_n]\}$. The image of $[B]$ by $[f]$ must be an interval containing the set $\mathcal{I}f([B]) = \{y \in \mathbb{R}, \exists \{x_1, x_2, ..., x_n\} \in [B], y = f(x_1, x_2, ..., x_n)\}$.

The *optimal* image $[f]_{opt}([B])$ is the sharpest interval containing $\mathcal{I}f([B])$. There exist many possible interval extensions for a function, the difficulty being to define an extension that computes the optimal image, or a sharp approximation of it.

The first idea is to use interval arithmetics. Interval arithmetics extends to intervals arithmetic operators $+$, $-$, $\times$, $/$ and elementary functions (*power*, *exp*, *log*, *sin*, *cos*, ...). For instance, $[a, b] + [c, d] = [a + c, b + d]$. The *natural interval extension* $[f]_n$ of a function f evaluates with interval arithmetics all the arithmetic operators and elementary functions in f.

When f is continuous inside a box $[B]$, the *natural evaluation* of f (i.e., the computation of $[f]_n([B])$) yields the optimal image when each variable occurs only once in f. When a variable appears several times, the evaluation by interval arithmetics generally produces an overestimation of $[f]_{opt}([B])$, because the correlation between the occurrences of a same variable is lost. Two occurrences of a variable are handled as independent variables. For example $[x] - [x]$, with $[x] \in [0, 1]$ gives the result $[-1, 1]$, instead of $[0, 0]$, as does $[x] - [y]$, with $[x] \in [0, 1]$ and [y] $\in [0, 1]$.

This main drawback of interval arithmetics causes a real difficulty for implementing efficient

interval-based solvers, since the natural evaluation is a basic tool for these solvers.

One way to overcome this difficulty is to use monotonicity [119]. In fact, when a function is monotonic w.r.t. each of its variables, this problem disappears and the evaluation (using a monotonicity extension) becomes optimal. For example, if $f(x_1, x_2)$ is increasing w.r.t. x_1, and decreasing w.r.t. x_2, then the *extension by monotonicity* $[f]_m$ of f is defined by :

$$[f]_m([B]) = [f(\underline{[x_1]}, \overline{[x_2]}), f(\overline{[x_1]}, \underline{[x_2]})] = [\underline{[f]_n(\underline{[x_1]}, \overline{[x_2]})}, \overline{[f]_n(\overline{[x_1]}, \underline{[x_2]})}]$$

It appears that $[f]_m([B]) = [f]_{opt}([B])$. This property can also be used when f is monotonic w.r.t. a subset of variables, replacing in the natural evaluations the intervals of monotonic variables by intervals reduced to their maximal or minimal values [119]. The obtained image is not optimal, but is sharper than, or equal to, the image obtained by natural evaluation. For example, if f is increasing w.r.t. x_1, decreasing w.r.t. x_2, and not monotonic w.r.t. x_3 :

$$[f]_{opt}([B]) \subseteq [f]_m([B]) = [\underline{[f]_n(\underline{[x_1]}, \overline{[x_2]}, [x_3])}, \overline{[f]_n(\overline{[x_1]}, \underline{[x_2]}, [x_3])}] \subseteq [f]_n([B])$$

This paper explains how to use monotonicity when a function is not monotonic w.r.t. a variable x, but is monotonic w.r.t. subgroups of occurrences of x. We present the idea of grouping the occurrences into 3 sets (increasing, decreasing and non monotonic auxiliary variables) in the next section. Linear programs for obtaining "interesting" occurrence groupings are described in Sections 3 and 4. In Section 5 we propose an algorithm to solve the linear programming problem presented in Section 4. Finally, in Section 6, some experiments show the benefits of this occurrence grouping for solving systems of equations, in particular when we use a filtering algorithm like `Mohc` [11] exploiting monotonicity.

E.2 Evaluation by monotonicity with occurrence grouping

In this section, we study the case of a function which is not monotonic w.r.t. a variable with multiple occurrences. We can, without loss of generality, limit the study to a function of one variable : the generalization to a function of several variables is straightforward, the evaluations by monotonicity being independent.

Example 1 *Consider $f_1(x) = -x^3 + 2x^2 + 6x$. We want to calculate a sharp evaluation of this function when x falls in $[-1.2, 1]$. The derivative of f_1 is $f_1'(x) = -3x^2 + 4x + 6$ and contains a positive term (6), a negative term $(-3x^2)$ and a term containing zero $(4x)$.*

$[f_1]_{opt}([B])$ is $[-3.05786, 7]$, but we cannot obtain it directly by a simple interval function evaluation (one needs to solve $f_1'(x) = 0$, which is in the general case a problem in itself).

In the interval $[-1.2, 1]$, the function f_1 is not monotonic. The natural interval evaluation yields $[-8.2, 10.608]$, the Horner evaluation $[-11.04, 9.2]$ (see [136]).

When a function is not monotonic w.r.t. a variable x, it sometimes appears that it is monotonic w.r.t. some occurrences. A first naive idea for using the monotonicity of these occurrences is the

following. We replace the function f by a function f^{nog}, regrouping all increasing occurrences into one variable x_a, all decreasing occurrences into one variable x_b, and the non monotonic occurrences into x_c. The domain of the new auxiliary variables is the same : $[x_a] = [x_b] = [x_c] = [x]$.

For f_1, this grouping results in $f_1^{nog}(x_a, x_b, x_c) = -x_b^3 + 2x_c^2 + 6x_a$. The evaluation by monotonicity of f_1^{nog} computes the lower (resp. upper) bound replacing the increasing (resp. decreasing) instances by the minimum (resp. maximum) and the decreasing (resp. increasing) instances by the maximum (resp. minimum), i.e.,

$$\underline{[f_1^{nog}]_m([-1.2, 1])} = \underline{[f_1^{nog}]_n(-1.2, 1, [-1.2, 1])} = \underline{-1^3 + 2[-1.2, 1]^2 - 7.2} = -8.2$$

$$(resp. \overline{[f_1^{nog}]_m([-1.2, 1])} = 10.608).$$

Finally, the evaluation by monotonicity is $[f_1^{nog}]_m([-1.2, 1]) = [-8.2, 10.608]$.

It appears that the evaluation by monotonicity of the new function f^{nog} always provides the same result as the natural evaluation. Indeed, when a node in the evaluation tree corresponds to an increasing function w.r.t. a variable occurrence, the natural evaluation automatically selects the right bound (among both) of the occurrence domain during the evaluation process.

The main idea is then to change this grouping in order to reduce the dependency problem and obtain sharper evaluations. We can in fact group some occurrences (increasing, decreasing, or non monotonic) into an increasing variable x_a as long as the function remains increasing w.r.t. this variable x_a.

For example, if one can move a non monotonic occurrence into a monotonic group, the evaluation will be the same or sharper. Also, if it is possible to transfer all decreasing occurrences into the increasing part, the dependency problem will now occur only on the occurrences in the increasing and non monotonic parts.

For f_1, if we group together the positive derivative term with the derivative term containing zero we obtain the new function : $f_1^{og}(x_a, x_b) = -x_b^3 + 2x_a^2 + 6x_a$, where f_1^{og} is increasing w.r.t. x_a and decreasing w.r.t. x_b. We can then use the evaluation by monotonicity obtaining the interval $[-5.32, 9.728]$. We can in the same manner obtain $f_1^{og}(x_a, x_c) = -x_a^3 + 2x_c^2 + 6x_a$, the evaluation by monotonicity yields then $[-5.472, 7.88]$. We remark that we find sharper images than the natural evaluation of f_1 does.

In Section E.3, we present a linear program to perform *occurrence grouping* automatically.

Interval extension by occurrence grouping

Consider the function $f(x)$ with multiple occurrences of x. We obtain a new function $f^{og}(x_a, x_b, x_c)$ by replacing in f every occurrence of x by one of the three variables x_a, x_b, x_c, such that f^{og} is increasing w.r.t. x_a in $[x]$, and f^{og} is decreasing w.r.t. x_b in $[x]$.

Then, we define the *interval extension by occurrence grouping* of f by :

$$[f]_{og}([B]) := [f^{og}]_m([B])$$

Unlike the natural interval extension and the interval extension by monotonicity, the interval extension by occurrence grouping is not unique for a function f since it depends on the occurrence grouping (og) that transforms f into f^{og}.

E.3 A 0,1 linear program to perform occurrence grouping

In this section, we propose a method for automatizing occurrence grouping. First, we calculate a Taylor-based overestimation of the *diameter* of the image computed by $[f]_{og}$. Then, we propose a linear program performing a grouping that minimizes this overestimation.

E.3.1 Taylor-Based overestimation

On one hand, as f^{og} could be not monotonic w.r.t. x_c, the evaluation by monotonicity considers the occurrences of x_c as different variables such as the natural evaluation would. On the other hand, as f^{og} is monotonic w.r.t. x_a and x_b, the evaluation by monotonicity of these variables is optimal. The following two propositions are well-known.

Proposition 9 *Let $f(x)$ be a continuous function in a box $[B]$ with a set of occurrences of x : $\{x_1, x_2, ..., x_k\}$. $f^{\circ}(x_1, .., x_k)$ is a function obtained from f considering all the occurrences of x as different variables. $[f]_n([B])$ computes $[f^{\circ}]_{opt}([B])$.*

Proposition 10 *Let $f(x_1, x_2, ..., x_n)$ be a monotonic function w.r.t. each of its variables in a box $[B] = \{[x_1], [x_2], ..., [x_n]\}$. Then, the evaluation by monotonicity is optimal in $[B]$, i.e., it computes $[f]_{opt}([B])$.*

Using these propositions, we observe that $[f^{og}]_m([x_a], [x_b], [x_c])$ is equivalent to $[f^{\circ}]_{opt}([x_a], [x_b], [x_{c_1}], ..., [x_{c_{ck}}])$, considering each occurrence of x_c in f^{og} as an independent variable x_{c_j} in f°. Using Taylor evaluation, an upper bound of $diam([f]_{opt}([B]))$ is given by the right side of (E.1) in Proposition 11.

Proposition 11 *Let $f(x_1, ..., x_n)$ be a function with domains $[B] = \{[x_1], ..., [x_n]\}$. Then,*

$$diam([f]_{opt}([B])) \leq \sum_{i=1}^{n} \big(diam([x_i]) \times |[g_i]([B])|\big) \tag{E.1}$$

where $[g_i]$ is an interval extension of $g_i = \frac{\partial f}{\partial x_i}$.

Using Proposition 11, we can calculate an upper bound of the **diameter** of $[f]_{og}([B]) = [f^{og}]_m([B]) = [f^\circ]_{opt}([B])$:

$$diam([f]_{og}([B])) \leq diam([x])\Big(|[g_a]([B])| + |[g_b]([B])| + \sum_{i=1}^{ck} |[g_{c_i}]([B])|\Big)$$

Where $[g_a]$, $[g_b]$ and $[g_{c_i}]$ are the interval extensions of $g_a = \frac{\partial f^{og}}{\partial x_a}$, $g_b = \frac{\partial f^{og}}{\partial x_b}$ and $g_{c_i} = \frac{\partial f^{og}}{\partial x_{c_i}}$ respectively. $diam([x])$ is factorized because $[x] = [x_a] = [x_b] = [x_{c_1}] = ... = [x_{c_{ck}}]$.

In order to respect the monotonicity conditions required by f^{og} : $\frac{\partial f^{og}}{\partial x_a} \geq 0$, $\frac{\partial f^{og}}{\partial x_b} \leq 0$, we have the sufficient conditions $\underline{[g_a]([B])} \geq 0$ and $\overline{[g_b]([B])} \leq 0$, implying $|[g_a]([B])| = \overline{[g_a]([B])}$ and $|[g_b]([B])| = -\underline{[g_b]([B])}$. Finally :

$$diam([f]_{og}([B])) \leq diam([x])\Big(\overline{[g_a]([B])} - \underline{[g_b]([B])} + \sum_{i=1}^{ck} |[g_{c_i}]([B])|\Big) \tag{E.2}$$

E.3.2 A linear program

We want to transform f into a new function f^{og} that minimizes the right side of the relation (E.2). The problem can be easily transformed into the following integer linear program :

Find the values r_{a_i}, r_{b_i} and r_{c_i} for each occurrence x_i that minimize

$$G = \overline{[g_a]([B])} - \underline{[g_b]([B])} + \sum_{i=1}^{k} \big(|[g_i]([B])| r_{c_i}\big) \tag{E.3}$$

subject to :

$$\underline{[g_a]([B])} \geq 0 \tag{E.4}$$

$$\overline{[g_b]([B])} \leq 0 \tag{E.5}$$

$$r_{a_i} + r_{b_i} + r_{c_i} = 1 \quad \text{for } i = 1, ..., k \tag{E.6}$$

$$r_{a_i}, r_{b_i}, r_{c_i} \in \{0, 1\} \quad \text{for } i = 1, ..., k,$$

where a value r_{a_i}, r_{b_i} or r_{c_i} equal to 1 indicates that the occurrence x_i in f will be replaced, respectively, by x_a, x_b or x_c in f^{og}. k is the number of occurrences of x, $[g_a]([B]) = \sum_{i=1}^{k} [g_i]([B]) r_{a_i}$, $[g_b]([B]) = \sum_{i=1}^{k} [g_i]([B]) r_{b_i}$, and $[g_i]([B]), ..., [g_k]([B])$ are the derivatives w.r.t. each occurrence.

We can remark that all the gradients (e.g., $[g_a]([B])$, $[g_b]([B])$) are calculated using only the derivatives of f w.r.t. each occurrence of x (i.e., $[g_i]([B])$).

Linear program corresponding to Example 1

We have $f_1(x) = -x^3+2x^2+6x$, $f_1'(x) = -3x^2+4x+6$ for $x \in [-1.2, 1]$. The gradient values for each occurrence are : $[g_1]([-1.2,1]) = [-4.32, 0]$, $[g_2]([-1.2,1]) = [-4.8, 4]$ and $[g_3]([-1.2,1]) = [6,6]$. Then, the linear program is :

Find the values r_{a_i}, r_{b_i} and r_{c_i} that minimize
$G = \sum_{i=1}^{3} \overline{[g_i]([B])} r_{a_i} - \sum_{i=1}^{3} \underline{[g_i]([B])} r_{b_i} + \sum_{i=1}^{3} \big(|[g_i]([B])| r_{c_i}\big)$
$= (4r_{a_2} + 6r_{a_3}) + (4.32r_{b_1} + 4.8r_{b_2} - 6r_{b_3}) + (4.32r_{c_1} + 4.8r_{c_2} + 6r_{c_3})$

subject to :

$$\sum_{i=1}^{3} \underline{[g_i]([B])} r_{a_i} = -4.32r_{a_1} - 4.8r_{a_2} + 6r_{a_3} \geq 0$$

$$\sum_{i=1}^{3} \overline{[g_i]([B])} r_{b_i} = 4r_{b_2} + 6r_{b_3} \leq 0$$

$$r_{a_i} + r_{b_i} + r_{c_i} = 1 \quad \text{for } i = 1, ..., 3$$

$$r_{a_i}, r_{b_i}, r_{c_i} \in \{0, 1\} \quad \text{for } i = 1, ..., 3$$

We obtain the minimum 10.8, and the solution $r_{a_1} = 1, r_{b_1} = 0, r_{c_1} = 0, r_{a_2} = 0, r_{b_2} = 0, r_{c_2} = 1, r_{a_3} = 1, r_{b_3} = 0, r_{c_3} = 0$, which is the last solution presented in Section 2. We can remark that the value of the overestimation of $diam([f]_{og}([B]))$ is equal to 23.76 ($10.8 \times diam[-1.2, 1]$)) whereas $diam([f]_{og}([B])) = 13.352$. Although the overestimation is quite rough, the heuristic works well on this example. Indeed, $diam([f]_n([B])) = 18.808$, and $diam([f]_{opt}([B])) = 10.06$.

E.4 A tractable linear programming problem

The linear program above is a 0,1 linear program and is known to be NP-hard in general. We can render it continuous and tractable by allowing r_{a_i}, r_{b_i} and r_{c_i} to get real values. In other words, we allow each occurrence of x in f to be replaced by a convex linear combination of auxiliary variables, x_a, x_b and x_c, f^{og} being increasing w.r.t. x_a, and decreasing w.r.t. x_b. Each occurrence x_i is replaced in f^{og} by $r_{a_i}x_a + r_{b_i}x_b + r_{c_i}x_c$, with $r_{a_i} + r_{b_i} + r_{c_i} = 1$, $\frac{\partial f^{og}}{\partial x_a} \geq 0$ and $\frac{\partial f^{og}}{\partial x_b} \leq 0$. We can then remark that f and f^{og} have the same natural evaluation.
In Example 1, we can replace f_1 by f^{og_1} or f^{og_2} in a way respecting the monotonicity constraints of x_a and x_b. Considering the interval $[x] = [-1.2, 1]$:

1. $f_1^{og_1}(x_a, x_b) = -(\frac{5}{18}x_a + \frac{13}{18}x_b)^3 + 2x_a^2 + 6x_a$: $[f_1^{og_1}]_m([x]) = [-4.38, 8.205]$
2. $f_1^{og_2}(x_a, x_b, x_c) = -x_a^3 + 2(0.35x_a + 0.65x_c)^2 + 6x_a$: $[f_1^{og_2}]_m([x]) = [-5.472, 7]$

Example 2 *Consider the function $f_2(x) = x^3 - x$ and the interval $[x] = [0.5, 2]$. f_2 is not monotonic and the optimal image $[f_2]_{opt}([x])$ is $[-0.385, 6]$.*

The natural evaluation yields $[-1.975, 7.5]$*, the Horner evaluation* $[-1.5, 6]$*. We can replace* f_2 *by one of the following functions.*

1. $f_2^{og_1}(x_a, x_b) = x_a^3 - (\frac{1}{4}x_a + \frac{3}{4}x_b)$: $[f_2^{og_1}]_m([x]) = [-0.75, 6.375]$
2. $f_2^{og_2}(x_a, x_b) = (\frac{11}{12}x_a + \frac{1}{12}x_b)^3 - x_b$: $[f_2^{og_2}]_m([x]) = [-1.756, 6.09]$

Taking into account the convex linear combination for realizing the occurrence grouping, the new linear program is :

Find the values r_{a_i}, r_{b_i} and r_{c_i} for each occurrence x_i that minimize (E.3) subject to (E.4), (E.5), (E.6) and

$$r_{a_i}, r_{b_i}, r_{c_i} \in [0, 1] \quad \text{for } i = 1, ..., k. \tag{E.7}$$

Linear program corresponding to Example 1

In this example we obtain the minimum 10.58 and the new function $f_1^{og}(x_a, x_b, x_c) = -x_a^3 + 2(0.35x_a + 0.65x_c)^2 + 6x_a$: $[f_1^{og}]_m([x]) = [-5.472, 7]$. The minimum 10.58 is less than 10.8 (obtained by the 0,1 linear program). The evaluation by occurrence grouping of f_1 yields $[-5.472, 7]$, which is sharper than the image $[-5.472, 7.88]$ obtained by the 0.1 linear program presented in Section E.3.

Linear program corresponding to Example 2

In this example we obtain the minimum 11.25 and the new function $f_2^{og}(x_a, x_b) = (\frac{44}{45}x_a + \frac{1}{45}x_b)^3 - (\frac{11}{15}x_a + \frac{4}{15}x_b)$. The image $[-0.75, 6.01]$ obtained by occurrence grouping is sharper than the interval computed by natural and Horner evaluations. Note that in this case the 0,1 linear program of Section E.3 yields the naive grouping due to the constraints.

Note that the continuous linear program not only makes the problem tractable but also improves the minimum of the objective function.

E.5 An efficient Occurrence Grouping algorithm

Algorithm 8 finds r_{a_i}, r_{b_i}, r_{c_i} (r-values) that minimize G subject to the constraints. The algorithm also generates the new function f^{og} that replaces each occurrence x_i in f by $[r_{a_i}]x_a + [r_{b_i}]x_b + [r_{c_i}]x_c$. Note that the r-values are represented by thin intervals, of a few u.l.p. large, for taking into account the floating point rounding errors appearing in the computations.

Algorithm 8 uses a vector $[g_*]$ of size k containing interval derivatives of f w.r.t. each *occurrence* x_i of x. For the sake of conciseness, each component of $[g_*]$ is denoted by $[g_i]$ hereafter, instead of $[g_i]([B])$, i.e., $[g_i]$ is the interval $\frac{\partial f}{\partial x_i}([B])$.

An asterisk (*) in the index of a symbol represents a vector (e.g., $[g_*]$, $[r_{a_*}]$).

Algorithm 8 `Occurrence_Grouping`(**in :** f, $[g_*]$ **out :** f^{og})

1: $[G_0] \leftarrow \sum_{i=1}^{k} [g_i]$
2: $[G_m] \leftarrow \sum_{0 \notin [g_i]} [g_i]$
3: **if** $0 \notin [G_0]$ **then**
4: `OG_case1`$([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
5: **else if** $0 \in [G_m]$ **then**
6: `OG_case2`$([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
7: **else**
8: /* $0 \notin [G_m]$ and $0 \in [G_0]$ */
9: **if** $[G_m] \geq 0$ **then**
10: `OG_case3`$^+([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
11: **else**
12: `OG_case3`$^-([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
13: **end if**
14: **end if**
15: $f^{og} \leftarrow$ `Generate_New_Function`$(f, [r_{a_*}], [r_{b_*}], [r_{c_*}])$

We illustrate the algorithm using the two univariate functions of our examples : $f_1(x) = -x^3 + 2x^2 + 6x$ and $f_2(x) = x^3 - x$ for domains of x : $[-1.2, 1]$ and $[0.5, 2]$ respectively.

The interval derivatives of f w.r.t. each occurrence of x have been previously calculated. For the examples, the interval derivatives of f_2 w.r.t. x occurrences are $[g_1] = [0.75, 12]$ and $[g_2] = [-1, -1]$; the interval derivatives of f_1 w.r.t. x occurrences are $[g_1] = [-4.32, 0]$, $[g_2] = [-4.8, 4]$ and $[g_3] = [6, 6]$.

In line 1, the partial derivative $[G_0]$ of f w.r.t. x is calculated using the sum of the partial derivatives of f w.r.t. each occurrence of x. In line 2, $[G_m]$ gets the value of the partial derivative of f w.r.t. the monotonic occurrences of x. In the examples, for f_1 : $[G_0] = [g_1] + [g_2] + [g_3] = [-3.12, 10]$ and $[G_m] = [g_1] + [g_3] = [1.68, 6]$, and for f_2 : $[G_0] = [G_m] = [g_1] + [g_2] = [-0.25, 11]$.

According to the values of $[G_0]$ and $[G_m]$, we can distinguish 3 cases. The first case is well-known ($0 \notin [G_0]$ in line 3) and occurs when x is a monotonic variable. The procedure `OG_case1` does not achieve any occurrence grouping : *all the occurrences* of x are replaced by x_a (if $[G_0] \geq 0$) or by x_b (if $[G_0] \leq 0$). The evaluation by monotonicity of f^{og} is equivalent to the evaluation by monotonicity of f.

In the second case, when $0 \in [G_m]$ (line 5), the procedure `OG_case2` (Algorithm 9) performs a grouping of the occurrences of x. Increasing occurrences are replaced by $(1 - \alpha_1)x_a + \alpha_1 x_b$, decreasing occurrences by $\alpha_2 x_a + (1 - \alpha_2)x_b$ and non monotonic occurrences by x_c (lines 7 to 13 of Algorithm 9). f_2 falls in this case : $\alpha_1 = \frac{1}{45}$ and $\alpha_2 = \frac{11}{15}$ are calculated in lines 3 and 4 of Algorithm 9 using $[G^+] = [g_1] = [0.75, 12]$ and $[G^-] = [g_2] = [-1, -1]$. The new function is : $f_2^{og}(x_a, x_b) = (\frac{44}{45}x_a + \frac{1}{45}x_b)^3 - (\frac{11}{15}x_a + \frac{4}{15}x_b)$.

The third case occurs when $0 \notin [G_m]$ and $0 \in [G_0]$. W.l.o.g., if $[G_m] \geq 0$, the procedure `OG_case3`$^+$ (Algorithm 10) first groups all the decreasing occurrences with the increasing group, i.e., it replaces every monotonic occurrence x_i by x_a (lines 2–5). The non monotonic occurrences

Algorithm 9 `OG_case2`(**in :** $[g_*]$ **out :** $[r_{a_*}], [r_{b_*}], [r_{c_*}]$)

1: $[G^+] \leftarrow \sum_{[g_i] \geq 0} [g_i]$
2: $[G^-] \leftarrow \sum_{[g_i] \leq 0} [g_i]$
3: $[\alpha_1] \leftarrow \dfrac{\underline{[G^+]}\overline{[G^-]} + \overline{[G^-]}\underline{[G^-]}}{\underline{[G^+]}\overline{[G^-]} - \underline{[G^-]}\overline{[G^+]}}$
4: $[\alpha_2] \leftarrow \dfrac{\underline{[G^+]}\overline{[G^+]} + \overline{[G^-]}\underline{[G^+]}}{\underline{[G^+]}\overline{[G^-]} - \underline{[G^-]}\overline{[G^+]}}$
5:
6: **for all** $[g_i] \in [g_*]$ **do**
7: **if** $\underline{[g_i]} \geq 0$ **then**
8: $(\overline{[r_{a_i}]}, \underline{[r_{b_i}]}, [r_{c_i}]) \leftarrow (1 - [\alpha_1], [\alpha_1], 0)$
9: **else if** $\overline{[g_i]} \leq 0$ **then**
10: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow ([\alpha_2], 1 - [\alpha_2], 0)$
11: **else**
12: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (0, 0, 1)$
13: **end if**
14: **end for**

are then replaced by x_a in a determined order stored by an array $index$[1] (line 7) as long as the constraint $\sum_{i=1}^{k} r_{a_i}\underline{[g_i]} \geq 0$ is satisfied (lines 9-13). The first non monotonic occurrence $x_{i'}$ that cannot be replaced because it would make the constraint unsatisfiable is replaced by :
$\alpha x_a + (1-\alpha)x_c$, with α such that the constraint is satisfied and equal to 0, i.e.,
$(\sum_{i=1, i\neq i'}^{k} r_{a_i}\underline{[g_i]}) + \alpha\underline{[g_{i'}]} = 0$ (lines 15–17). The rest of the non monotonic occurrences are replaced by x_c (lines 20–22). f_1 falls in this case. The first and third occurrences of x are monotonic and are then replaced by x_a. Only the second occurrence of x is not monotonic, and it cannot be replaced by x_a because it would make the constraint unsatisfiable. It is then replaced by $\alpha x_a + (1-\alpha)x_c$, where $\alpha = 0.35$ is obtained forcing the constraint (E.4) to be $0 : \underline{[g_1]} + \underline{[g_3]} + \alpha\underline{[g_2]} = 0$. The new function is : $f_1^{og} = -x_a^3 + 2(0.35x_a + 0.65x_c)^2 + 6x_a$.

Finally, the procedure `Generate_New_Function` (line 15 of Algorithm 8) creates the new function f^{og} symbolically.

Observations

Algorithm 8 respects the four constraints (E.4)–(E.7). We are currently proving that the minimum of the objective function in (E.3) is also reached.

Instead of Algorithm 8, we may use a standard Simplex algorithm, providing that the used

[1] An occurrence x_{i_1} is handled before x_{i_2} if $|\overline{[g_{i_1}]}/\underline{[g_{i_1}]}| \leq |\overline{[g_{i_2}]}/\underline{[g_{i_2}]}|$. $index[j]$ yields the occurrence index i such that $[g_i]$ is the j^{th} interval in the sorting order.

Algorithm 10 `OG_case3`$^+$**(in :** $[g_*]$ **out :** $[r_{a_*}], [r_{b_*}], [r_{c_*}]$**)**

$[g_a] \leftarrow [0,0]$
for all $[g_i] \in [g_*]$, $0 \notin [g_i]$ **do**
  $[g_a] \leftarrow [g_a] + [g_i]$ /*All positive and negative derivatives are absorbed by $[g_a]$ */
  $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (1, 0, 0)$
end for

$index \leftarrow$ `ascending_sort`$(\{[g_i] \in [g_*], 0 \in [g_i]\},$ `criterion` $\rightarrow |\overline{[g_i]}/\underline{[g_i]}|)$
$j \leftarrow 1$; $i \leftarrow index[1]$
while $\underline{[g_a]} + \underline{[g_i]} \geq 0$ **do**
  $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (1, 0, 0)$
  $[g_a] \leftarrow [g_a] + [g_i]$
  $j \leftarrow j + 1$; $i \leftarrow index[j]$
end while

$[\alpha] \leftarrow -\frac{\underline{[g_a]}}{\overline{[g_i]}}$
$([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow ([\alpha], 0, 1 - [\alpha])$
/* $[g_a] \leftarrow [g_a] + [\alpha][g_i]$ */
$j \leftarrow j + 1$; $i \leftarrow index[j]$

while $j \leq$ `length`$(index)$ **do**
  $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (0, 0, 1)$
  $j \leftarrow j + 1$; $i \leftarrow index[j]$
end while

Simplex implementation is adapted to take into account rounding errors due to floating point arithmetics. In a future work, we will compare the performances of Algorithm 8 and Simplex.

Time complexity

The time complexity of `Occurrence_Grouping` for a variable with k occurrences is $O(k\,log_2(k))$. It is dominated by the complexity of `ascending_sort` in the `OG_case3` procedure. As shown in the experiments of the next section, the time required in practice by `Occurrence_Grouping` is negligible when it is used for solving systems of equations.

E.6 Experiments

`Occurrence_Grouping` has been implemented in the `Ibex` [52, 50] open source interval-based solver in `C++`. The goal of these experiments is to show the improvements in CPU time brought by `Occurrence_Grouping` when solving systems of equations. Sixteen benchmarks are issued from the COPRIN website [204]. They correspond to square systems with a finite number of zero-dimensional solutions of at least two constraints involving multiple occurrences of variables and requiring more than 1 second to be solved (considering the times appearing in the website). Two instances (`<name>-bis`) have been simplified due to the long time required for their resolution : the input domains of variables have been arbitrarily reduced.

E.6.1 Occurrence grouping for improving a monotonicity-based existence test

First, `Occurrence_Grouping` has been implemented to be used in a monotonicity-based existence test (`OG` in Table E.1), i.e., an occurrence grouping transforming f into f^{og} is applied after a bisection and before a contraction. Then, the monotonicity-based existence test is applied to f^{og} : if the evaluation by monotonicity of f^{og} does not contain 0, the current box is eliminated.

The competitor (`¬OG`) applies directly the monotonicity-based existence test to f without occurrence grouping.

The contractors used in both cases are the same : `3BCID` [283] and Interval Newton.

Problem	`3BCID`	`¬OG`	`OG`	**Problem**	`3BCID`	`¬OG`	`OG`
brent-10	18.9 3941	19.5 3941	19.1 3941	butcher-bis	351 228305	360 228303	340 228245
caprasse	2.51 1305	2.56 1301	2.56 1301	fourbar	13576 8685907	6742 4278767	**1091** 963113
hayes	39.5 17701	41.1 17701	40.7 17701	geneig	593 205087	511 191715	**374** 158927
i5	55.0 10645	56.3 10645	56.7 10645	pramanik	100 124661	66.6 98971	**37.2** 69271
katsura-12	74.1 4317	74.5 4317	75.0 4317	trigexp2-11	82.5 14287	87.0 14287	86.7 14287
kin1	1.72 85	1.77 85	1.77 85	trigo1-10	152 2691	155 2691	156 2691
eco9	12.7 6203	13.5 6203	13.2 6203	virasoro-bis	21.1 2781	21.5 2781	19.8 2623
redeco8	5.61 2295	5.71 2295	5.66 2295	yamamura1-8	9.67 2883	10.04 2883	9.86 2883

TAB. E.1 – Experimental results using the monotonicity-based existence test. The first and fifth columns indicate the name of each instance, the second and sixth columns yield the CPU time (above) and the number of nodes (below) obtained on an `Intel 6600 2.4 GHz` by a strategy based on `3BCID`. The third and seventh columns report the results obtained by the strategy using a (standard) monotonicity-based existence test and `3BCID`. Finally, the fourth and eighth columns report the results of our strategy using an existence test based on occurrence grouping and `3BCID`.

From these first results we can observe that only in three benchmarks `OG` is clearly better than `¬OG` (`fourbar`, `geneig` and `pramanik`). In the other ones, the evaluation by occurrence grouping seems to be useless. Indeed, in most of the benchmarks, the existence test based on occurrence grouping does not cut branches in the search tree. However, note that it does not require additional time w.r.t. `¬OG`. This clearly shows that the time required by `Occurrence_Grouping` is negligible.

E.6.2 Occurrence Grouping inside a monotonicity-based contractor

`Mohc` [11] is a new constraint propagation contractor (like `HC4` or `Box`) that uses the monotonicity of a function to improve the contraction/filtering of the related variables. Called inside a propagation algorithm, the `Mohc-revise(`f`)` procedure improves the filtering obtained by `HC4-revise(`f`)` by mainly achieving two additional calls to `HC4-revise(`$f_{min} \leq 0$`)` and `HC4-revise(`$f_{max} \geq 0$`)`, where f_{min} and f_{max} correspond to the functions used when the evaluation by monotonicity calculates the lower and upper bounds of f. It also performs a monotonic version of the `BoxNarrow` procedure used by `Box` [28].

Table E.2 shows the results of `Mohc` without the `OG` algorithm (¬`OG`), and with `Occurrence-Grouping` (`OG`), i.e., when the function f is transformed into f^{og} before applying `MohcRevise(`f^{og}`)`.

Problem	`Mohc`			**Problem**	`Mohc`		
	¬`OG`	`OG`	#`OG` calls		¬`OG`	`OG`	#`OG` calls
brent-10	20	20.3		butcher-bis	220.64	**7.33**	
	3811	3805	30867		99033	2667	111045
caprasse	2.57	2.71		fourbar	4277.95	**385.62**	
	1251	867	60073		1069963	57377	8265730
hayes	17.62	17.45		geneig	328.34	**111.43**	
	4599	4415	5316		76465	13705	2982275
i5	57.25	58.12		pramanik	67.98	**21.23**	
	10399	9757	835130		51877	12651	395083
katsura-12	100	103		trigexp2-11	90.57	88.24	
	3711	3625	39659		14299	14301	338489
kin1	1.82	1.79		trigo1-10	137.27	**57.09**	
	85	83	316		1513	443	75237
eco9	13.31	13.96		virasoro-bis	18.95	**3.34**	
	6161	6025	70499		2029	187	241656
redeco8	5.98	6.12		yamamura1-8	11.59	**2.15**	
	2285	2209	56312		2663	343	43589

TAB. E.2 – Experimental results using `Mohc`. The first and fifth columns indicate the name of each instance, the second and sixth columns report the results obtained by the strategy using `3BCID(Mohc)` without `OG`. The third and seventh columns report the results of our strategy using `3BCID(OG+Mohc)`. The fourth and eighth columns indicate the number of calls to `Occurrence_Grouping`.

We observe that, for 7 of the 16 benchmarks, `Occurrence_Grouping` is able to improve the results of `Mohc`; in `butcher8-bis`, `fourbar`, `virasoro-bis` and `yamamura-8` the gains in CPU time ($\frac{\neg OG}{OG}$) obtained are 30, 11, 5.6 and 5.4 respectively.

E.7 Conclusion

We have proposed a new method to improve the monotonicity-based evaluation of a function f. This *Occurrence Grouping* method creates for each variable three auxiliary, respectively increasing, decreasing and non monotonic variables in f. It then transforms f into a function f^{og} that groups the occurrences of a variable into these auxiliary variables. As a result, the *evaluation by occurrence grouping* of f, i.e., the evaluation by monotonicity of f^{og}, is better than the evaluation by monotonicity of f.

Occurrence grouping shows good performances when it is used to improve the monotonicity-based existence test, and when it is embedded in a contractor algorithm, called `Mohc`, that exploits monotonicity of functions.

Annexe F

First results obtained by PolyBox

Résultats présentés à SCAN (GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations) en octobre 2008.

Auteurs : Gilles Trombettoni, Yves Papegay, Gilles Chabert, Odile Pourtallier

We have compared PolyBox to HC4 and Box. All these contractors have been implemented in the free Ibex interval-based C++ library (by Gilles Chabert) [52, 50]. Before using PolyBox, for every pair (f, x), Mathematica is used to transform f into a form which is expanded w.r.t. x.

To find all the solutions to the tested systems of equations, the solving strategy bisects the variables in a round-robin way. Between two bisections :

1. constraint propagation (PolyBox, HC4 or Box, i.e., BC4 implemented in Ibex) is performed before
2. an interval Newton using a Hansen-Sengupta matrix, a preconditioning of the matrix, and a Gauss-Seidel method to solve the interval linear system.

Among the 44 polynomial systems with isolated solutions found in the Web page of the COPRIN team [204], we have selected the 12 instances that :

– are solved by at least one of the 3 strategies in a time comprised between 1 second and 1 hour (on a Pentium 3 GHz),
– have equations with multiple occurrences of the variables.

Table F.1 shows the results obtained on these instances.

TAB. F.1 – Experimental results. A cell in the last three columns contains the CPU time in second (above) and the number of nodes in the search tree (below).

Name	**#variables**	**#solutions**	HC4	BC4	PolyBox
Caprasse	4	18	5.53 9539	37.2 6509	**2.16** 2939
Yamamura1	8	7	34.3 42383	13.4 4041	**2.72** 2231
Extended Wood	4	3	**0.76** 4555	1.94 1947	1.12 3479
Broyden Banded	20	1	> 3600 ?	0.62 1	**0.09** 1
Extended Freudenstein	20	1	> 3600 ?	0.19 121	**0.11** 121
6body	6	5	**0.58** 4899	2.93 4797	0.73 4887
Rose	3	18	> 3600 ?	> 3600 ?	**4.11** 12521
Discrete Boundary	39	1	179 185617	29.5 3279	**16.1** 3281
Katsura	12	7	**102** 14007	404 11371	104 13719
Eco9	8	16	**66.4** 132873	191 125675	71 131911
Broyden Tridiagonal	20	2	470 269773	495 163787	**349.6** 164445
Geneig	6	10	3657 79472328	> 7200 ?	**3363** 4907705

Annexe G

GPDOF : A Fast Algorithm to Decompose Under-constrained Geom. Systems

Full title : "GPDOF : A Fast Algorithm to Decompose Under-constrained Geometric Constraint Systems : Application to 3D Modeling"

Article [288] : publié dans la revue IJCGA (Int. Journal of Computational Geometry and Applications) en 2006

Auteurs : Gilles Trombettoni, Marta Wilczkowiak

Abstract

Our approach exploits a general-purpose decomposition algorithm, called GPDOF, and a dictionary of very efficient solving procedures, called r-methods, based on theorems of geometry. GPDOF decomposes an equation system into a sequence of small subsystems solved by r-methods, and produces a set of input parameters.[282]

Recursive assembly methods (decomposition-recombination), maximum matching based algorithms, and other famous propagation schema are not well-suited or cannot be easily extended to tackle geometric constraint systems that are under-constrained. In this paper, we show experimentally that, provided that redundant constraints have been removed from the system, GPDOF can quickly decompose large under-constrained systems of geometrical constraints.

We have validated our approach by reconstructing, from images, 3D models of buildings using interactively introduced geometrical constraints. Models satisfying the set of linear, bilinear and quadratic geometric constraints are optimized to fit the image information. Our models contain several hundreds of equations. The constraint system is decomposed in a few seconds, and can then be solved in hundredths of second.

Keywords : geometric constraints ; decomposition ; computer vision

G.1 Introduction

Solving a set of geometric constraints is a challenging problem in parametric Computer Aided Design. Because of the intrinsic complexity of the corresponding equation systems, several researchers have followed the "divide and conquer" paradigm. Several rule-based or graph-based decomposition algorithms have thus been designed to split the constraint system into several subsystems that can be solved more efficiently.[277, 234, 176, 3, 89, 129, 95, 151]

Although some difficulties still need to be overcome, these methods behave rather well on well-constrained or rigid problems[1]. It turns out that only little has been done to treat under-constrained problems that have an infinite set of solutions (even modulo the group of direct isometries). However, it would allow a designer to build its mechanism or to draw its figure in an incremental way by adding geometric objects and constraints one by one.

In this paper, we are interested in specific geometric constraint systems for which an initial value is known for all (or most of) the variables involved in the objects. The values are often known approximately and may of course not satisfy "exactly" the constraints. Three main types of problems fall in this category :

- *Solution maintenance :* a figure is corrected and completed incrementally on the screen, but all the already drawn objects have a current, and not definite, position and orientation.
- *Free-hand drawing using a sketch :* the values given by the sketch are used by the constraint solver to draw the final figure.
- *3D model reconstruction from images using geometric constraints* (2D images along with additional constraints on the 3D scene are given to the software) : before the constraints are solved, initial point values are computed by a standard optimization algorithm based on images.

When an initial value is available for the variables, the decomposition problem can be tackled by :

- Selecting a set of variables and transforming them into *input parameters* (i.e., constants) : selecting them makes the remaining system well-constrained. *The values given to the input parameters, combined with the geometric constraints, fully fix all the degrees of freedom and completely describe the figure.*
- Decomposing the constraint system into subsystems and solving the decomposed system in a given order. In the end, new values are computed for all the variables (except the input parameters) so that all the constraints are satisfied.

Note that this approach can be viewed as a specific way to add non-user defined constraints in order to make the system well-constrained. Indeed, it amounts to adding a unary constraint on every input parameter in order to fix its value (e.g., using the sketch).

It seems that no existing decomposition method is really well-suited to handle under-constrained systems.

Existing propagation mechanisms cannot guarantee finding a correct order between subsystems without a backtracking step, i.e., in a polynomial time. The propagation of the *known states*

[1] A rigid system is well-constrained modulo the group of direct isometries.

(see Ref. [39]) and the *reactive* propagation algorithm (see Refs. [38, 72]) become exponential when the system is under-constrained (see Ref. [281], pages 64 and 70, Ref. [303], page 172, and Ref. [139]).

Geometric decomposition methods

Bottom-up decomposition methods, also known as recursive assembly methods, reduction analysis or decomposition-recombination methods, use recursively a procedure to identify a rigid subsystem in the whole system.[89, 129, 95, 151] The different corresponding rule-based or graph-based identification procedures have a common point : they are designed for identifying a well-constrained subsystem (modulo the group of direct isometries). As pointed in Ref. [89], when the system is globally under-rigid, they are intrinsically unable to assemble the different rigid subparts together.

Top-down methods, also known as decomposition analysis or recursive division methods, exhibit recursively certain vertices in the constraint graph for splitting a given system in (potentially) rigid subparts.[89, 161] These methods can tackle under-rigid systems and "complete" them to make them well-constrained, by adding automatically non user-defined constraints. However, the vertex selection, as well as the constraint completion, are limited to geometric constraint systems in 2D made of specific sets of objects and constraints.

Equational decomposition methods

Other approaches apply the famous Maximum-matching graph algorithm on a dependency graph between equations and variables.[265, 92, 180] When the system is well-constrained (i.e, no degree of freedom must be fixed), there is a unique decomposition into subsystems, i.e., into strongly connected components.[174] When the system is rigid in 2D, it is yet tractable to fix the 3 remaining degrees of freedom in a coordinate system (all the possible coordinate systems must be tried in a combinatorial way). In this case, the approach is similar to certain graph-based recursive rigidification methods.[129]

When the system is under-rigid, the combinatorial process above is not tractable. Instead, following any maximum matching of the system, Dulmage and Mendelsohn's decomposition allows us to determine a structurally under-constrained part in which input parameters can be extracted.[74, 180, 34] Unfortunately, this structural analysis has serious drawbacks :

– Different decompositions can be obtained according to the computed matching. The decompositions differ in the set of input parameters (and thus in the actual equation system that is handled), and in the computed subsystems.
– Certain decompositions may contain large subsystems (in terms of number of equations), while others contain smaller subsystems[2]. It appears that finding the decomposition of a structurally under-constrained system minimizing the size of the largest subsystem is the dual of the *minimum dense* problem that has been proven to be NP-hard.[188]

[2]For instance, all the equations of the didactic example below could be put into a unique subsystem if the variables c_{la}, c_{la} and y_{pb} are chosen as input parameters.

– Certain decompositions are geometrically correct while others are incorrect, that is, contain subsystems with redundant or contradictory equations.[282, 303]
– No specific solving procedure exists for solving the subsystems, so that a generic solver must be called to solve the involved equations.

To sum up, faced to an under-constrained system, Maximum-matching may select subsystems that are very large or geometrically incorrect, and no fast procedure is known for solving the subsystems.

Using GPDOF

On the contrary, our GPDOF algorithm is particularly well-suited to select input parameters and to compute a decomposition of an under-constrained system. Indeed, the identified subsystems belong to a set of predefined patterns for which a fast solving procedure is known. Also, the identified subsystems are geometrically correct : they contain neither redundant nor contradictory equations in the generic case.

The procedures used to solve the subsystems are called *r-methods* (resolution methods). They are based on theorems of geometry and incorporated into a *dictionary*. The dictionary allows the general-purpose equational algorithm GPDOF to make a bridge with the geometric level. Provided that the system contains no redundant equation, GPDOF produces a set of input parameters and a sequence of r-methods present in the dictionary in polynomial time (if any).

GPDOF (see Ref. [282]) is a generalization of the PDOF local propagation algorithm.[278, 39, 295] In those approaches, constraints are selected and solved one by one (i.e., the subsystems contain only one equation). Extensions to geometric constraints select subsystems that include several equations but place exactly one object.[2, 195, 78] General-PDOF (GPDOF) subsumes all these approaches by selecting r-methods of any type : r-methods solving one equation (like in the standard PDOF), r-methods solving several equations used to compute one object and, potentially, r-methods computing several objects simultaneously.

We have validated GPDOF in 3D model reconstruction, an important field in computer vision. Constraints are incorporated into the 3D model acquisition system. The user can select in images objects such as points, lines or planes and can define geometrical dependencies between them, such as parallelism, orthogonality, and distance constraints. Then a model satisfying those constraints, conforming to the image information (by minimization of the reprojection error) is computed. Our system has been applied to two architectural models including several hundreds of constraints. Most of them are bilinear, some of them are linear or quadratic (distance constraints). In both models, the preprocessing step is performed by GPDOF in a few seconds, and then all the constraints are solved in hundredths of second.

Contribution and limits

As explained above, GPDOF is of no particular interest for tackling well-constrained or well-rigid geometric constraint systems because Maximum-matching can be used instead. On the contrary,

GPDOF is well-suited for under-rigid constraint systems because it selects input parameters and quickly solves a sequence of geometrically correct subsystems.

GPDOF overcomes the main known drawback of local propagation algorithms : the presence of loops in the constraint graph. Indeed, GPDOF selects "general" r-methods that solve any type of equation subsystem that may include cycles.

Also, GPDOF works at two different levels and takes the best of both. Indeed, some r-methods may correspond to theorems of geometry while others may be defined at the variable/equation level. However, in the end, all the r-methods are translated into hyper-edges in the equation graph, that is, at the variable/equation level. Thus, constraint and equation graphs are both taken into account while losing no semantic (i.e., geometric) information. This allows GPDOF to tackle systems made of geometric and non geometric constraints.

Finally, the work presented in this article is the first realistic application of GPDOF. In particular, it describes for the first time a pre-processing phase (called automatic r-method addition phase in the article) which is needed by GPDOF in practice. The implementation experimentally shows that GPDOF and the pre-processing phase constitute a rule-based decomposition method with impressive performance on large under-constrained systems.

Section G.6.5 underlines the main limit of GPDOF : the redundant constraints and over-constrained subparts must be removed before running the algorithm.

Also, it is important to understand that geometric (in particular, top-down) decomposition methods and equational decomposition approaches behave differently on under-constrained geometric systems, and in a sense do not tackle the same types of applications. Indeed :

– Top-down approaches must "complete" the system with additional constraints such that the whole construction is rigid (i.e., with 6 degrees of freedom in 3D).
– Equational approaches, such as Maximum-matching and GPDOF, ignore this rigidity property and extract a set of input parameters. That is why they are well-suited to tackle the applications mentioned above : solution maintenance, free-hand drawing using a sketch, and the 3D model reconstruction described in this article.

Section G.6.6 illustrates this difference and highlights why geometric decomposition methods can generally better decompose a geometric system while GPDOF is a general-purpose algorithm able to handle geometric and non geometric equations.

Outline

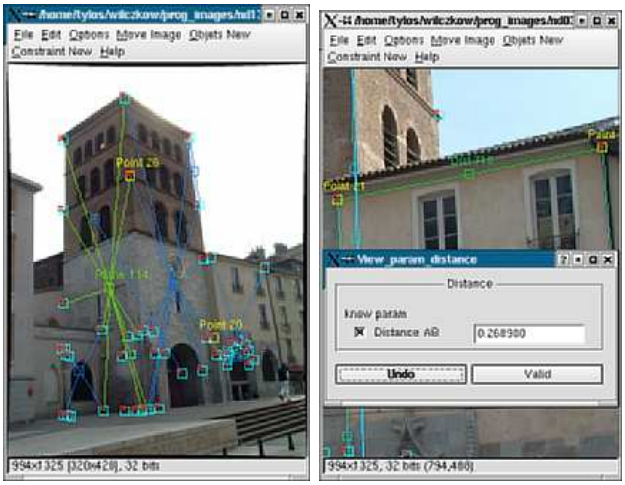
Section G.2 presents an overview of our method. Section G.3 introduces a scene in 2D illustrating the main algorithms used by our system. Section G.4 details how objects and constraints are modeled, and includes the background necessary to understand the constraint satisfaction part. Section G.5 details the automatic r-method addition phase which is a preliminary step for the use of GPDOF described in Section G.6. The optimization phase is described in Section G.7. Section G.8 shows the experiments we have performed on two models of a church. Section G.9 describes the related work in computer vision.

G.2 Overview of the approach used in 3D model reconstruction

Our 3D model acquisition makes use of geometric constraints. It is divided into three main phases : initialization, constraint planning and optimization.

Initialization

In addition to 2D images and feature projections matched between images, geometric objects and constraints between them must be defined as input to create the 3D model. The model is represented here by *points*, *lines* and *planes*. They are subject to linear, bilinear and quadratic constraints such as *distance*, *incidence*, *parallelism* and *orthogonality*. All the objects and constraints are introduced using a graphical user interface (see Figure G.1). The cameras are



(a) (b)

FIG. G.1 – Interface used for the scene definition. (a) Definition of points and planes in one of the images belonging to the Notre Dame sequence. (b) Definition of a distance constraint between two points.

then calibrated using the linear method described in Ref. [304]. An initial reconstruction is provided by a multi-linear approach exploiting projections and geometric constraints combined with an unconstrained bundle adjustment.[305] However, any other approach for calibration and reconstruction could be used here. *Note that after this phase all the variables (camera and model parameters) have an initial value.*

Constraint planning

The goal of the constraint planning phase is to transform a model defined by constraints among objects into a parametric model. Thus, the output of this step is composed of :

– *Input parameters* : ideally, the number of input parameters is equal to the number of degrees of freedom of the model $n_m = n_o - n_c$, where n_o is the sum of degrees of freedom of the model objects (i.e., the number of involved variables) and n_c is the sum of degrees of freedom of the model constraints (i.e., the number of independent equations).
– *A plan* : a sequence of routines (called *r-methods*) which, given values assigned to the input parameters, computes the coordinates of all the scene objects in a way that all the inter-object relations are satisfied.

The model reconstruction system we propose requires an input set of *r-methods* which allow us to decompose the whole equation system into small subsystems. An r-method is a hard-coded procedure used to solve here a subset of geometric constraints.[282, 281] An r-method computes the coordinates of *output objects* based on the current value of *input object* coordinates with respect to the underlying constraints between input and output objects.

An example is an r-method that computes the parameters of a line based on the current position of two points incident to this line. Another example is an r-method that computes the position of a 3D point A located at known distances from three other points B, C, D. This r-method computes the (at most) two possible positions for A by intersecting the three spheres centered respectively in B, C, and D. Sixty r-method patterns have been incorporated in a dictionary used by our system. They correspond to ruler-and-compass routines used in geometry or, more generally, to standard theorems of geometry.

The geometric constraint system and the corresponding algebraic equations are represented by graphs. The *constraint graph* yields the dependencies between constraints and objects. The *equation graph* yields the dependencies between equations and variables. Based on these graphs, the constraint planning uses two algorithms :

1. *R-method addition phase :* Add *automatically* in the equation graph all the r-methods corresponding to r-method patterns present in the dictionary. For instance, the r-method pattern "line incident to two known points" may occur a lot of times in the system. Every time two points incident to a line are recognized in the constraint graph, a corresponding r-method is added to the equation graph (see example below).
   This phase thus produces an equation graph "enriched" with r-methods.
2. *Planning phase :* Perform `GPDOF` on the enriched equation graph. `GPDOF` produces :
   – a set of input parameters, that is, a subset of the variables describing the scene such that, when a value is given to them, there exists a finite set of solutions for the rest of the system satisfying the constraints ;
   – a sequence of r-methods (called *plan*) to be executed one by one.

Model optimization

The model is refined by an unconstrained optimization process run over the input parameters only. The optimization produces values for all the model variables so that all the constraints are satisfied (exactly) and the sum of reprojection errors is minimal. At each iteration of the optimization algorithm, the input parameter values are modified and the plan is executed, resulting

in new coordinate values for all the other scene objects. The cost function is then computed as the reprojection error of all the points (and possibly lines).

G.3 Example of constraint planning

To illustrate the algorithms presented in this article, we will take a small example describing a parallelogram in 2D in terms of lines, points, incidence constraints and parallelism constraints (see Figure G.2). Of course, the scenes we handle with our tool are in 3D, and this example is just presented for didactic reasons. Figure G.2 also shows the bipartite constraint graph containing four points P_a,...,P_d with coordinates (x_{pa}, y_{pa}), ..., (x_{pd}, y_{pd}), four lines L_a,...,L_d with coordinates (a_{la}, b_{la}, c_{la}), ..., (a_{ld}, b_{ld}, c_{ld}), eight incidence constraints C_1,...,C_8 and two parallelism constraints C_9, C_{10}.

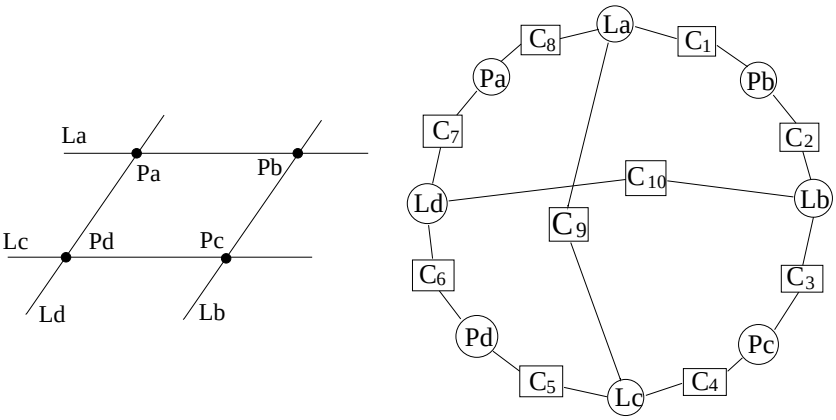


FIG. G.2 – **Left :** A didactic example of a 2D scene. **Right :** The corresponding constraint graph.

The equation graph corresponding to the 2D scene is shown in Figure G.3-left. The right side of Figure G.3 shows the equation graph enriched with a set of r-methods that can be used to solve subparts of the system. An r-method is represented by a hyper-arc including its equations and its output variables. These r-methods belong to one of the three following categories (these patterns could appear in a dictionary of 2D r-methods) :

– line incident to two points (e.g., r-methods m_1 and m_7) ;
– point at the intersection of two known lines (e.g., m_2, m_4, m_6, m_8) ;
– line passing through a known point and parallel to another line (e.g., m_3, m_5).

The GPDOF algorithm, presented in Section G.6, works on the enriched equation graph. It is able to select for example the coordinates of P_a, P_b and P_d as a set of input parameters. It also produces a plan, e.g., the sequence of r-methods (m_1, m_7, m_3, m_5, m_4), whose execution results in new coordinate values for objects L_a, L_d, L_b, L_c, P_c respectively. Figure G.4 illustrates an execution of this plan.

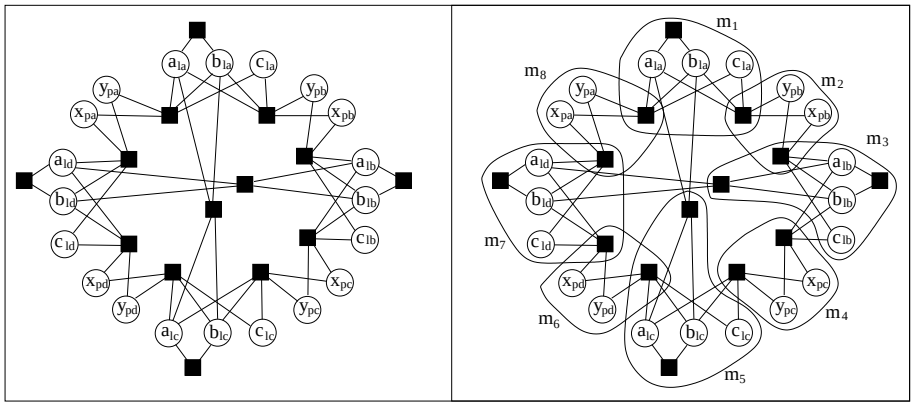


FIG. G.3 – **Left :** The equation graph of the 2D scene. Variables are represented by circles and equations are represented by black rectangles. An equation involving only the a and b coordinates of a line has the form $a^2+b^2 = 1$ to allow a unique representation of a line. **Right :** The enriched equation graph. Only 8 of the 16 possible r-methods are depicted for the sake of clarity.

G.4 Scene Modeling and Background

G.4.1 Geometric objects

In the current implementation of the system, available objects are points, lines and planes. Their representation has a significant impact on the performance of the system. It is necessary to use a representation that corresponds to the number of degrees of freedom characterizing the objects and allows an efficient computation. We briefly detail how the objects are represented in the system.

Points : the 3 degrees of freedom (dof) of a point are represented by 3 variables x, y, z.

Lines : the 4 degrees of freedom of a line are represented by 6 variables (called Plücker coordinates) and 2 internal relations. The coordinates correspond to the line direction vector **d** and the vector **n** normal to the plane passing through the line and the coordinate system origin. The line coordinates are related by two conditions : the directional vector is constrained to be of unit length and vectors **n** and **d** are constrained to be perpendicular. See Ref. [120] for details about the Plücker line representation.

Planes : the 3 degrees of freedom of a plane are represented by 4 variables and 1 internal relation. The coordinates correspond to the plane normal **n** and its distance d from the origin. The normal vector **n** is constrained to be of unit length.

Note that the solving process described in this article can also support other parameterizations and other types of primitives. Also note that, in the computer vision application, the final model

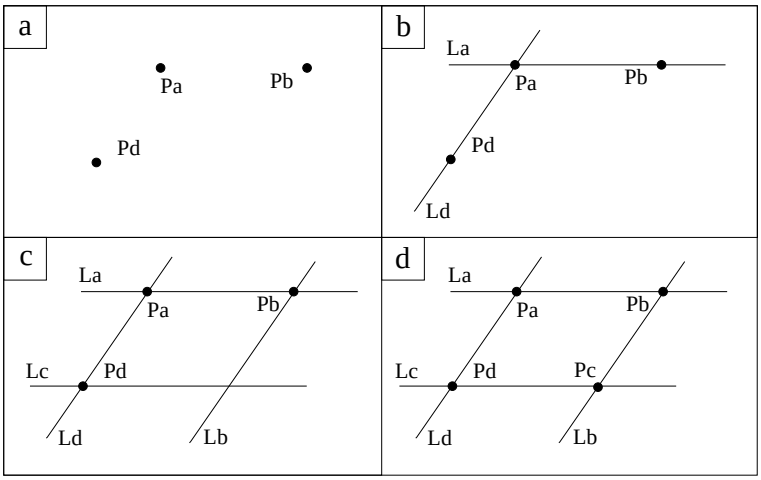


FIG. G.4 – Execution of r-methods in the plan (m_1, m_7, m_3, m_5, m_4). (a) The input parameters (i.e., coordinates of P_a, P_b and P_d) are replaced by their current value. (b) m_1 and m_7 place lines L_a and L_d resp. (c) m_3 and m_5 place L_b and L_c resp. (d) Finally, m_4 places point P_c.

is represented only by a set of points, that is, lines and planes are viewed as collinearity and coplanarity constraints between points.

G.4.2 Geometric constraints

Constraints are also characterized by a number of degrees of freedom they fix on the involved objects. In the current implementation, we consider the following constraints :

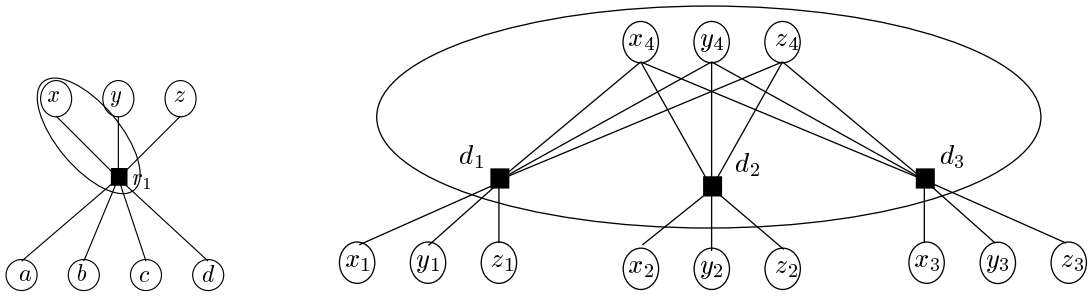
– *distance :* point-point (1 dof), point-line (1 dof), point-plane (1 dof),
– *incidence :* point-line (2 dofs), point-plane (1 dof), line-plane (2 dofs),
– *parallelism :* line-line (2 dofs), line-plane (1 dof), plane-plane (2 dofs),
– *orthogonality :* line-line (1 dof), line-plane (2 dofs), plane-plane (1 dof).

Any other constraint which can be expressed as equations in terms of objects coordinates, like angles, distance and angle ratios can be incorporated.

G.4.3 R-methods

Our model reconstruction system is based on a dictionary containing an input set M of *r-methods*.

An r-method is a routine executed to satisfy a subset E_m of equations in E by calculating values for its *output variables* as a function of the other variables implied in the equations. Two examples of r-methods are shown in Figure G.5.



(a) (b)

FIG. G.5 – Representation of r-methods by hyper-arcs that include the satisfied equations and the output variables. (a) An r-method fixing the remaining degrees of freedom of a point incident to a plane. (b) An r-method computing the position of a point using 3 point-point distance constraints.

Definition 13 *An* **r-method** m *in* M *is a function over a set of* **input variables** I*. The variables involved in the equations* $E_m \subset E$ *are divided into a non-empty set of* **output variables** $O \subset V$*, and a set of input variables* $I \subset V$*.*

Given a set of values $\overline{I}$*, the r-method* m *yields the set of solutions for* O *satisfying* E_m*. This operation is called* **execution** *of the r-method* m*.*

The r-method m is **free** if no variable v in O is involved in a constraint in $E \setminus E_m$. Thus, executing a free method cannot violate other equations in $E \setminus E_m$.

Note that a given equation can generally be solved by several r-methods. For instance, 3 r-methods (computing a value for x, y, *or* z) could be used to solve the point-plan incidence shown in Fig. G.5(a)). This makes combinatorial the problem of computing a sequence of r-methods to solve all the equations.

The current dictionary of our system contains 60 r-methods. These r-methods use only 45 different execution procedures. For instance, plane-plane parallelisms and line-line parallelisms are solved by the same procedure.

The dictionary includes all the r-methods that solve constraints by computing (output) parameters of *one* object. More complicated r-methods computing more than one object at a time can be envisaged. The algorithms used in our system can deal with any type of r-method, although the time complexity will grow with the number of constraints involved in r-methods (see Section G.8.2). Details on the design and implementation of r-methods are given in Ref. [303].

An **r-method pattern** present in the dictionary is a generic constraint graph corresponding to the equations solved by the r-method. We design this constraint graph by a pattern because a similar constraint graph (pattern) may occur several times in the actual constraint graph corresponding to the model, thus leading to the creation of several similar r-methods. For instance, the r-method pattern "line incident to two known points" is a graph made of 4 vertices (3 objects and 1 constraint). Every time two points incident to a line are recognized in the constraint graph (as a subgraph), a corresponding r-method is added to the equation graph. Thus, an r-method

in the dictionary is defined at both levels : geometric one (i.e., a pattern made of objects and constraints) and equational one (i.e., a hard-coded procedure solving the corresponding equations). Finally, it is important for our application to distinguish so-called *linear r-methods* giving one solution and *non-linear r-methods* giving generally several solutions.

Definition 14 *Let m be an r-method, E_m be the set of equations solved by m, and O (resp. I) be the set of output (resp. input) variables of m.*

The r-method m is a **linear r-method** *iff all the equations in E_m are linear in terms of variables in O (i.e., E_m in which the variables in I are replaced by a constant become linear). m is a* **non-linear r-method** *iff at least one equation in E_m (in terms of O) is non-linear : m may produce several solutions.*

For instance, an r-method that computes the position of some 3D point A located at known distances from three other points B, C, D is a non-linear r-method. This r-method generally produces two possible positions for A.

An r-method that computes the parameters of a line based on the current position of two points incident to this line is a linear r-method. Even though an incidence constraint is bilinear, when the coordinates of the involved points are known, this gives linear equations relating line coordinates.

Hypotheses on r-methods

R-methods must compute a finite set of solutions for the output variables. In other words, the dimension of the variety of the solutions is 0. Therefore r-methods have generally as many equations as output variables. This is the case for the r-methods in our dictionary.

In addition, an r-method, especially a non-linear r-method, must be able to compute *all* the solutions satisfying the involved equations. Indeed, this allows the backtracking phase described in Section G.7.1 to combine the solutions computed by the different r-methods in the plan, without losing any solution.

The remark above highlights that numerical local minimization methods cannot be used for executing an r-method because they cannot obtain *all* the solutions for the output variables.

Interests of r-methods

Using r-methods for decomposing the constraint system of the model has a lot of advantages :

- The code of an r-method allows a very good performance. Executions of r-methods in our dictionary run in several microseconds.
- A lot of r-methods in our dictionary are linear while the involved constraints are bilinear (e.g., incidence, parallelism). This highlights that using r-methods to decompose a system of equations is an interesting way to lower the complexity of the equations and thus to improve the performance.

– The semantics behind a given r-method (i.e., the fact that it is a theorem of geometry) ensures that the implied geometric constraints can be solved and helps to detect singular configurations. On the contrary, the subsystems of equations created by pure graph-based decomposition methods, such as the Maximum-matching, are arbitrary (see Refs. [282, 303]).
– As said above, r-methods yield all the solutions to the implied equations.

G.4.4 Graph representation of the model and data structures

The algorithms used by our system require a structural view of the entities in the scene. The geometric constraint system and the equation system are respectively represented by a *constraint graph* and an *equation graph* (see Figures G.2 and G.3). An equation graph indicates the dependencies between equations and variables in the scene.

Definition 15 *A* **constraint graph** *is a bipartite graph where vertices are constraints and objects, represented by rectangles and circles respectively. Each constraint is connected to its objects.*
An **equation graph** *is a bipartite graph* (V, E, A) *where vertices are equations in* E *and variables in* V*, represented by rectangles and circles respectively. Each equation is connected to its variables by an edge in* A*.*
An **enriched equation graph** (V, E, A, M) *is an equation graph* (V, E, A) *enriched with a set of hyper-edges corresponding to r-methods in* M*. A hyper-edge of a given r-method* $m \in M$ *is a subgraph induced by the variables and equations of* m*.*

Our system is implemented in `C++`. The different entities (constraints, geometric objects, equations, variables and r-methods) are represented by structured objects. Several fields have been added to allow a direct access to the entities. For example, for a given variable v, we can know in constant time the set of equations involving v, of which r-method v is an output variable, to which geometric object v belongs, and so on. In this implementation, the constraint graph, the equation graph and the enriched equation graph share the same data structures.

The dictionary of r-methods is implemented as a hash table in order to make the automatic r-method addition phase quicker. Details about this hash table are given in Section G.5.

The next two sections detail the algorithms used by the constraint planning : the automatic addition of r-methods to the equation graph (based on the dictionary), and the computation of a set of input parameters and a sequence of r-methods (based on the enriched equation graph).

G.5 Automatic R-method Addition Phase

This phase enriches the equation graph with r-methods found in the dictionary. It considers as input :

– the constraint graph corresponding to the scene ;
– the dictionary of r-method patterns.

This phase works on the constraint graph. It performs a matching between subgraphs (made of constraints and objects) in the constraint graph and r-method patterns present in the dictionary. More precisely, we handle a *subgraph isomorphism* problem. All the connected subgraphs of the constraint graph with a "small" size are explored. When a subgraph corresponds to an entry in the dictionary, the corresponding r-methods are added to the equation graph.

For instance, on the 2D scene, a certain iteration of the r-method addition phase considers the subgraph made of nodes P_a, C_8, L_a, C_1, P_b (see Figure G.2). A corresponding subgraph pattern (i.e, 2 points incident to a line) is found in the dictionary, so that the r-method m_1 (i.e., line L_a passing through two known points P_a and P_b) is created and added to the equation graph.

The algorithm explores all the connected subgraphs of size less than or equal to a small value k (see next paragraph). For every found subgraph, the procedure `Subgraph_recognition` compares it with the subgraph patterns in our dictionary. If the subgraph matches, the corresponding r-methods are added to the equation graph[3].

G.5.1 Exploring all connected subgraphs of size at most k

The value k is the maximum number of nodes (objects+constraints, or constraints only in the current implementation) implied in any r-method of the dictionary (e.g., 7 in our system ; 4 in the last version - see Section G.8). Starting from a single node (S is a singleton), the subgraphs are built by incrementally adding a neighbor node to the current connected subgraph S until the size k is reached. This depth-first search algorithm detailed in Algorithm `All_connected` is a simplification of the algorithmic scheme presented in Ref. [15].

```
Algorithm All_connected (S : set of nodes ; d : current depth ; k : max size ; G : constraint graph) :
    Subgraph_recognition (S)
    if d < k then
        N' ← Selected_neighbors (S, d, k, G)
        for every neighbor n in N' do
            All_connected (S ∪ {n}, d + 1, k, G)
        end
    end
end.
```

The time complexity of this algorithm[4] is $O(N \times a \times k^4)$, where N is the actual number of connected subgraphs of size k or less and a is the maximum degree of nodes in the graph. N is $O(n^k)$, where n is the number of vertices in the constraint graph.

[3]Remember that several r-methods may exist for the same set of constraints.

[4]The call to `Subgraph_recognition` is not taken into account.

G.5.2 Subgraph recognition

The function `Subgraph_recognition` compares every subgraph S found in the constraint graph with the subgraph patterns in our dictionary. However, the problem of deciding whether two graphs are *isomorphic* is still an open problem for which no polynomial algorithm is known.[235] In order to quickly know whether a subgraph S is isomorphic with a subgraph S' in the dictionary, we proceed as follows :

1. We first compute a string e corresponding to the number of nodes in S in every category : the number of points in S, its number of lines, number of planes, number of point-line incidence constraints, and so on. The dictionary is implemented as a hash table, and such strings are the keys for indexing into the hash table (hash functions). If the key e corresponds to no entry in the dictionary, the second step below must not be performed and no r-method will be created based on the subgraph S.

   Otherwise, this means that an entry in the hash table contains one set ES' of subgraphs (having the same number of nodes in every category). The step 2 below checks whether one subgraph pattern $S' \in ES'$ is isomorphic with S.

2. To know whether two graphs S and S' (with the same number of nodes in every category) are isomorphic, we use a combinatorial process inspired by the solving process of Constraint Satisfaction Problems (chronological backtracking). In short, objects in the subgraph S are reordered to be matched with objects in the pattern S'. Two objects at the same rank in the order must have the same type and also the same types of constraints with objects placed before.

3. If S and S' match, then the r-methods associated to S' are added to the equation graph.

In our dictionary, the 60 r-method patterns are generated by 45 different subgraph patterns. The hash table contains 45 entries, which means that all the subgraph patterns are discriminated by their number of nodes in every category (i.e., the size of ES' is always 1 in our current version). The example in Figure G.6 highlights why the combinatorial process (step 2) remains necessary.

G.5.3 Practical time complexity

In practice, as detailed in Section G.8, the time complexity of the r-method addition phase is negligible (one or two seconds for our 3D models). Two reasons explain this good behavior. First, in our current dictionary, the subgraph patterns are small, so that the size k is small. Second, constraint graphs corresponding to scenes are rather sparse : the number of equations is equal to about half of the number of variables.

The situation would worsen if the designer wanted to add in the dictionary more complicated r-methods, especially r-methods with more than one object as output. In this case, we think that more sophisticated subgraph isomorphism algorithms should be envisaged.[290, 240, 272]

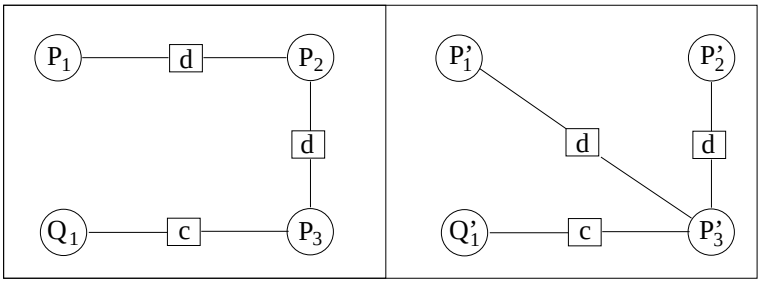


FIG. G.6 – Two subgraphs with the same entry in the hash table, i.e., characterized by the same number of points, planes, distance constraints and incidences. (a) a "bad" subgraph found in the constraint graph with no corresponding r-method ; (b) a subgraph pattern in our dictionary made of 3 points P'_1, P'_2, P'_3, a plane Q'_1, two distance constraints and one incidence.

G.6 Computing a Plan and a Set of Input Parameters

These computations are obtained by the GPDOF algorithm.[282] GPDOF[5] computes a sequence of r-methods to be executed for satisfying all the equations (the plan). GPDOF solves this combinatorial problem in polynomial time. The main advantages of GPDOF are the following :

– GPDOF is very fast (quasi-linear in practice).
– GPDOF can find a sequence of r-methods present in a dictionary if such a plan exists.
– A set of input parameters can be immediately deduced from the plan. Thus, GPDOF is also a procedure to determine a set of input parameters in polynomial time.

These attractive properties come under the assumption that the constraint system contains no redundant constraints, that is, the system must include only independent equations. Section G.6.5 details this point and explains the first procedures used by our tool for removing redundant constraints before the use of GPDOF.

G.6.1 Description of GPDOF

GPDOF works on an enriched equation graph (see Section G.5). GPDOF runs the three following steps until no more equation remains in the equation graph G (success) or no more free r-method is available (failure) :

1. select a **free** r-method [6] m,
2. remove from G the equations and the output variables of m,
3. call the Connect procedure : create all the *submethods* of every r-method m_i that share equations or output variables with m (see Section G.6.2).

[5] GPDOF stands for General Propagation of Degrees of Freedom.
[6] Recall that output variables of a *free r-method* appear in no "external" equations (see Definition 13).

A plan can be obtained by reversing the selection order : the first selected r-method will be executed last. The work of GPDOF is illustrated in Figure G.7.

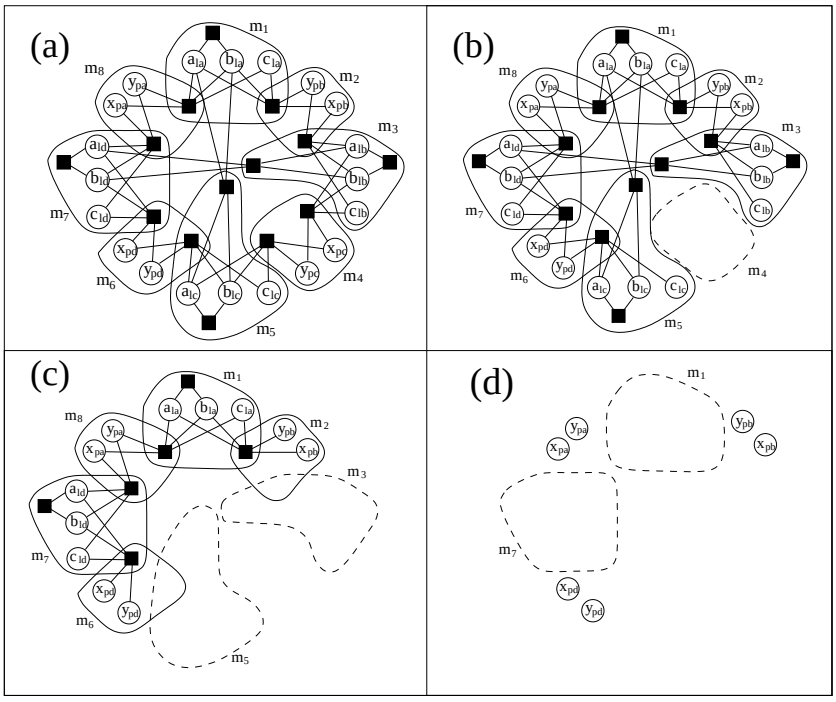


FIG. G.7 – A constraint planning phase performed by GPDOF on the 2D scene. (a) In the beginning, r-methods m_2, m_4, m_6, m_8 are free, so that one of them is selected, e.g., m_4. (b) This selection implies the removal of the equations and the output variables of m_4 from the equation graph. (c) This frees r-methods m_3 and m_5 which are selected and removed next in any order. (d) R-methods m_1 and m_7 become free and can be selected. The process ends since no more constraint remains in the equation graph. The obtained plan is the sequence (m_1, m_7, m_3, m_5, m_4) and the input parameters are the remaining variables.

The first two steps above define the standard PDOF algorithm on which GPDOF is based (PDOF accepts only r-methods solving one equation).[278] Iteratively selecting free r-methods ensures that no loop is created in the plan.

GPDOF may fail when no more free r-method is available. In this case, it is ensured that no complete plan can be computed. One obtains an incomplete plan which solves only a subset of the equations (i.e., the equations removed by step 2 of GPDOF) and contains thus more input parameters. This incomplete plan can nevertheless be executed, although the equations not removed in steps 2 will not be satisfied.

G.6.2 Overlap of r-methods and submethods

It appears that, when r-methods solve several equations, there is no guarantee that the standard PDOF finds a plan, even if one exists. This means that straightforward extensions of PDOF to geometry (see Refs. [195, 78, 2]) may be unable to compute a sequence of subsystems corresponding to patterns in the dictionary, even if one such sequence exists. A simple example is described in Figure 6 of Ref. [282], and the case does occur in real applications. To overcome this drawback, one needs to be able to select r-methods that "overlap", that is, sharing equations and variables. A plan that contains overlapping r-methods means that some constraints will be solved several times during one plan execution. This analysis has led to the design of GPDOF that introduces the notion of *submethod* and adds the third step above.

That step, called Connect in Ref. [282], maintains a connectivity property on the hyper-edges (r-methods). The procedure is called once a free r-method m has been selected in step 1 and removed in step 2, which removes some equations and/or output variables from m_i. The obtained subgraph of m_i is called *submethod* m'_i. The modified hyper-edge m'_i is maintained in the set of candidate r-methods. This means that a "partially" removed r-method m_i is still candidate for a future selection. If m_i becomes free during the process and is selected, it will overlap the r-method m. It appears that the subgraph corresponding to a submethod must be a connected graph, so that a given r-method m_i may be theoretically split in several connected submethods (m'_{i1}, m'_{i2}...) . Refs. [282, 281] detail how submethods are precisely constructed and why submethods are needed to ensure that a sequence of r-methods will be found (if any).

Examples of submethods are depicted in Figure G.8. R-methods m'_5, m'_7 are the submethods of resp. m_5, m_7 due to the selection of m_6 and the removal of equations solved by m_6 (black rectangles).

It is important to understand that the notion of submethod is only used during the work of GPDOF which is graph-based, but no submethod appears in the final plan : every time GPDOF selects a submethod, the corresponding r-method is added to the plan in order to be executed later.

The example also shows that, due to the selection of overlapping r-methods, some constraints are solved several times by different r-methods in the plan. For instance the two r-methods m_2 and m_3 belong to the same plan so that the incidence constraint between point P_b and line L_b will be solved twice.

Solving several times a same equation has no significant impact on the plan execution. In particular, all the constraints are solved in the end. The only drawback is that a plan with overlapping r-methods contains generally more r-methods (e.g., the plan shown in Fig. G.7 contains 5 r-methods, while the plan in Fig. G.8 contains 7 r-methods). As a result, we could expect a loss in performance.

However, r-methods are executed in several microseconds (see Section G.8). Moreover, the phenomenon of selecting overlapping r-methods can be easily limited by heuristics : when GPDOF has a choice between several free r-methods at a given iteration (step 1), GPDOF selects one r-method that is not a submethod (if any).

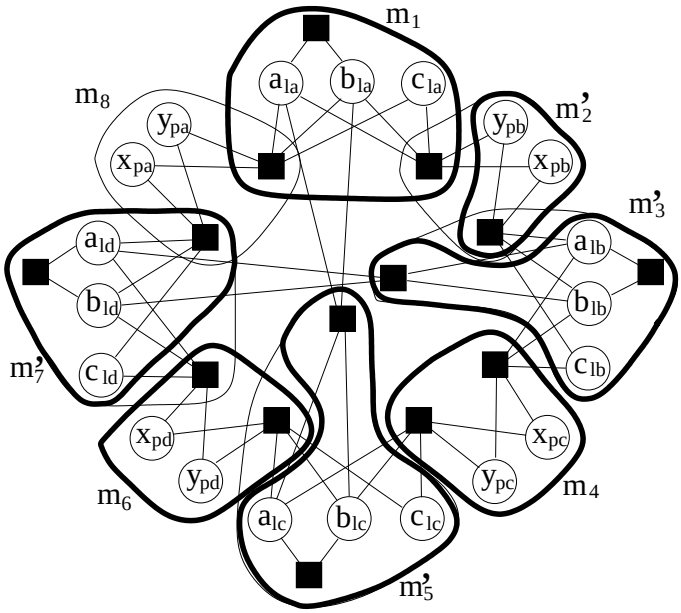


FIG. G.8 – GPDOF may first select m_6 that is free. Step 3 of GPDOF then creates the submethod m'_5 of m_5 and the submethod m'_7 of m_7. The process continues and selects m_4, m'_5, m_1 (creation of submethod m'_2), m'_2 (creation of submethod m'_3), m'_3, and finally m'_7. The obtained plan is the sequence (m_7, m_3, m_2, m_1, m_5, m_4, m_6). Selected r-methods (m_1, m_4, m_6) and submethods (m'_2, m'_3, m'_5, m'_7) are represented by thick hyper-arcs.

G.6.3 Properties of GPDOF

Due to the notion of submethod, it is proven that GPDOF guarantees to compute a sequence of r-methods present in the dictionary, if one such sequence exists.[282]

In addition, GPDOF solves this combinatorial problem in polynomial time. Its worst-case time complexity is $O(n \times dc \times dv \times m \times (g \times dc + g^2))$,[282] where n is the number of equations, m is the maximum number of r-methods per equation, dc and dv are the maximum degrees of respectively equations and variables in the equation graph, and g is the maximum number of equations and output variables involved in an r-method [7].

GPDOF (and the r-method addition phase) run in a few seconds on our two 3D models with several hundreds equations, showing that it should be acceptable for real applications.

[7] Theoretically, m is $O(n^g)$, rendering the approach practicable for small patterns only. A reviewer has built a scalable 2D example (made of points and distances between points) with a quadratic number of r-methods, due to a pair of points implied in all the constraints (i.e., with an unbounded value of dv). We know no pathological example when the number dv of constraints per variable is limited...

G.6.4 Computing the input parameters

Since GPDOF computes the plan in a reverse order, obtaining the input parameters is a side-effect of GPDOF. When no r-method in the plan corresponds to a submethod selection, the input parameters simply consist of the variables that are output of none of the r-methods in the plan. This yields the 6 coordinates of points P_a, P_b, P_d for the plan illustrated in Fig. G.7. The general case is a little bit more complicated and produces two disjoint subsets of input parameters :

- The set $\mathcal{P}_1$ contains the variables which are output by no r-method in the plan (as above).
- The set $\mathcal{P}_2$ comes from the overlap phenomenon. $\mathcal{P}_2$ contains variables v such that :
  - $v \notin \mathcal{P}_1$,
  - v is an input variable of an r-method m_j in the plan,
  - v is not an output variable of any r-method m_i placed before m_j in the sequence,
  - v is an output variable of an r-method m_k placed after m_j in the sequence.

The values of variables in $\mathcal{P}_1$ must be known before the plan is executed and will not be modified by this execution. On the opposite, the initial value of a variable v in $\mathcal{P}_2$ is used when an r-method m_j is executed, but v will be modified later by another r-method m_k in the plan.

In the plan illustrated in Fig. G.8, the subset $\mathcal{P}_1$ contains the coordinates of point P_a. The subset $\mathcal{P}_2$ contains the parameters of points P_b, P_c, P_d and lines L_a, L_b.

Number of input parameters in the system

The example has been chosen to illustrate the two subsets of input parameters and contains a large set $\mathcal{P}_2$. In practice however, due to the heuristics avoiding the selection of submethods by GPDOF, $\mathcal{P}_2$ is small. This will be underlined in the experiments made on the two realistic models presented below. Anyway, a natural question arises : what is the number of input parameters ?

Assume that all the r-methods are *square*, that is, they have as many output variables as equations[8]. If $\mathcal{P}_2$ is empty (because no submethod has been selected by GPDOF), it is straightforward to prove that $|\mathcal{P}_1| = n - e$ (n is the number of variables in the equation system, and e is the number of equations). In the general case however :

- $|\mathcal{P}_1| \leq n - e$ and
- $|\mathcal{P}_1| + |\mathcal{P}_2| \geq n - e$

Roughly, this means that the more GPDOF must select submethods to calculate a plan, the larger will be the set of input parameters.

It appears however that the plan produced by GPDOF can be used to yield a set of $n - e$ input parameters. Indeed, a set of parameters $\mathcal{P}'_2$ can be computed by a maximum matching such that $|\mathcal{P}_1| + |\mathcal{P}'_2| = n - e$. A maximum matching must be applied on every subgraph corresponding to the selected submethods. The variables that are not matched constitute the set of input parameters $\mathcal{P}'_2$ (see Ref. [303], page 151). Although interesting in theory, considerations about performance let us think that this variant is not promising in practice. Indeed, the transformed

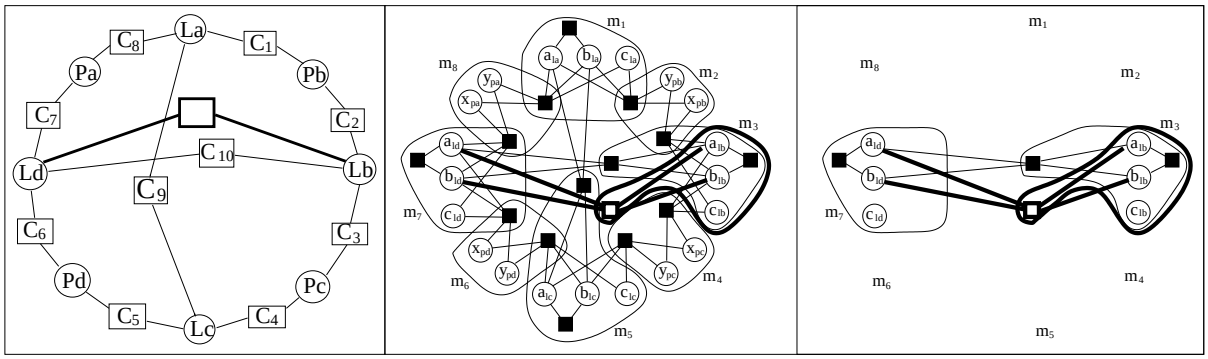
[8]This the case for the 60 r-methods in our dictionary.

submethods cannot be solved by a hard-coded procedure anymore, so that we need to resort to a generic solver.

G.6.5 Dealing with singularities and redundant constraints

Singularities have been the major cause of occasional divergence of the optimization process. For instance, a singularity may occur during an r-method execution that computes a plane based on 3 almost collinear points. Such singularities necessarily occur inside an r-method subsystem and can often be easily detected. In particular, before a linear r-method is added to the enriched graph, the corresponding subsystem of equations (in which the initial values of the input variables are used) is checked against a singularity using a Singular Value Decomposition (SVD).[238] Details are discussed in Ref. [303].

Another problem is that our graph-based algorithms may be misled by redundant constraints. However, this can often be fixed in practice by making use of geometric information.



(a) (b) (c)

FIG. G.9 – Failure of GPDOF in presence of a redundant parallelism constraint. (a) The parallelism constraint is redundant to the parallelism C_{10} (in the constraint graph). (b) The enriched equation graph with the corresponding additional equation (white rectangle) and two additional r-methods (only one is represented). (c) After having removed all possible free r-methods and submethods, GPDOF is stuck because the redundant equation prevents r-method m_3 from being free.

Redundant constraints involve non-independent equations. Because they correspond to theorems of geometry, the r-methods selected by GPDOF in the plan correspond necessarily to non-contradictory and independent systems of equations. However, GPDOF may fail in presence of redundant constraints because the selection of free r-methods is purely structural. As an example, consider Figure G.9 where an additional parallelism constraint has been added between lines L_b and L_d. This constraint is redundant with the existing parallelism constraint and prevents GPDOF from finding a plan. Since the selection step of GPDOF is structural, all occurs as if all equations were independent.

It is of course not acceptable to rely on the user to not introduce redundant constraints. Dealing with constraint redundancy has been a subject of research in the CAD community for a long

time and it is still an open problem in the general case. Straightforward procedures have been introduced in our tool to remove very common causes of redundancy. For example, one type of redundancy occurs if the user adds an incidence c_1 between a point P and a line L, an incidence c_2 between the line L and a plane A, and an incidence c_3 between the point P and the plane A. In this case, our procedure removes from the whole system all redundant constraints such as c_3. We found these procedures helpful in practice although we know that many occurring redundancies cannot be handled this way. More complicated procedures developed in the geometry/CAD community should be used to remove more redundancies. For example, a lot of redundant parallelisms and orthogonalities could be removed by using a straightforward routines closing the Desargues theorem.[274, 273] Also, a generalization of the numerical technique used to tackle singularities could be used to detect redundant constraints.

G.6.6 Comparison between geometric and equational decomposition techniques

In the introduction, we made a comparison between both main equational decomposition techniques. On one hand, as opposed to Maximum-matching, `GPDOF` can produce "small" and "correct" subsystems of equations that specific hard-coded procedures can solve quickly. On the other hand, Maximum-matching is not limited by predefined types of subsystems present in a dictionary. In fact, both algorithms can complete each other. The reader interested in this subject will find in Ref. [282] a description of a `MM-PDOF` hybrid algorithm, in which a maximum matching is incrementally updated and used to find a free subsystem when `PDOF` has failed to find one. This section compares equational (MM and PDOF based) and geometric (top-down and bottom-up) decomposition techniques. Four differences between both approaches are underlined below.

First, equational techniques are intrinsically not limited to pure geometric systems. They can also take into account constraints about thermodynamics, electricity, costs, and so on.

Second, equational techniques work at the variable/equation level and can then produce sometimes smaller subsystems than bottom-up or top-down techniques by distinguishing the different degrees of freedom of a same object. Figure G.10-left shows a simple example in 2D where the two variables of line D_1 (for instance, its angle $D_1.a$ and its distance to origin $D_1.x$) are not distinguished by a geometric solver. Without detailing, $D_1.x$ and $D_1.a$ are both put into a subsystem of size 2. An equational decomposition technique would produce a finer decomposition with two subsystems of size 1, computing first $D_1.a$ and then $D_1.x$ (see Figure G.10-right).

We do not believe that this observation gives a significant advantage to equational techniques. Indeed, to our knowledge, there exists no geometric rigid system that could be decomposed by an equational technique, but not by a geometric technique. If our intuition is true, for obtaining a finer decomposition and better handling systems like the one in Fig. G.10-left, one has to perform the following steps :

1. apply a geometric decomposition into clusters,
2. apply a final equational decomposition inside every cluster.

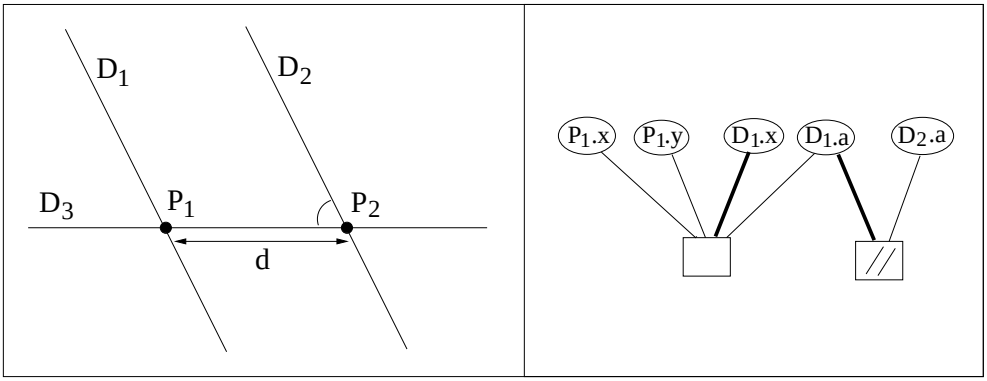


FIG. G.10 – **Left :** A rigid geometric system in 2D made of 2 points, 3 lines, 1 distance, 1 angle, 1 parallelism, and 4 point-line incidence constraints. **Right :** Equation subgraph including the incidence constraint between point P_1 and line D_1, and the parallelism between lines D_1 and D_2. An equational decomposition places line D_1 by solving the 2 equations in sequence : the angle of D_1 and then its distance to origin.

This principle has been followed for obtaining the decomposed systems studied in Ref. [155]. Note that rule-based geometric decomposition algorithms generally perform this type of fine decomposition implicitly.

Third, geometric techniques can generally better decompose a rigid system because their recursive use of the rigidity concept transforms (and, in a sense, simplifies) the system of equations. Figure G.11-left shows a small example in 2D made of points and distances between points.

This system can be decomposed by geometric techniques, but not by `GPDOF` (or Maximum-matching). For instance, several bottom-up solvers identify the two rigid "rhombuses", replace them by a representative (the distance constraints in dotted lines) and finally handle the last triangle. `GPDOF` identifies no free 2×2 r-method, and Maximum-matching (or `GPDOF` with a rich dictionary !) creates a large 10×10 subsystem including the point shared by the two rhombuses, the four neighbouring points and the 10 (bold-faced) distance constraints.

Fourth, as underlined at the end of the introduction, geometric and equational techniques do not tackle under-constrained (i.e., under-rigid) systems in the same way. Due to its inherent mechanism, a geometric decomposition method must complete an under-rigid figure so as to maintain the obtained figure invariant by displacements. `GPDOF` must not fulfill this property.

This point is illustrated in Figure G.11-right by a variant of the previous system from which a distance constraint (in dotted line) has been removed. A geometric top-down decomposition technique, such as the ones described in Refs. [89, 161], completes the figure by adding, among other possibilities, the dotted edge, while `GPDOF` works differently. It first selects and removes two subsystems in the "incomplete rhombus". Next, `GPDOF` selects and removes for example point P with the corresponding distance. This means that one coordinate of P (let us say $P.x$) is chosen as input parameter while the other ($P.y$) is computed by the selected 1×1 free r-method.

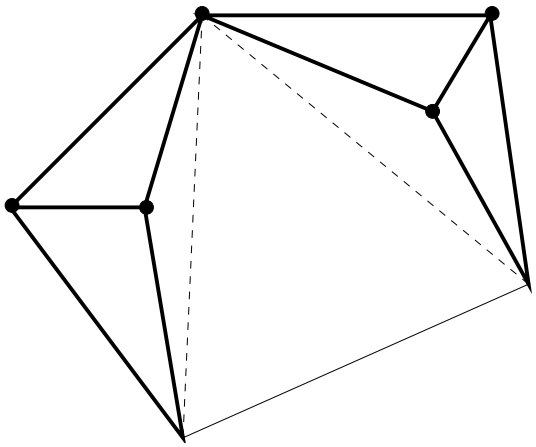


FIG. G.11 – **Left :** A rigid geometric system in 2D made of points and distance constraints. A geometric technique can decompose it into 2×2 subsystems while an equational approach computes one 10×10 subsystem (the 5 black points and the 10 bold-faced edges - distances). **Right :** An under-rigid geometric system tackled differently by equational and geometric decomposition techniques.

Finally, the last example also illustrates that the more under-constrained the system is, the better GPDOF works, because *it is easier to find a free r-method in the whole system*. Of course, an under-constrained system is not a sufficient condition to make it (well) decomposable by equational techniques[9].

G.7 Optimization Phase

As said in the overview, the optimization process produces values for the variables such that all the constraints are satisfied *exactly* and the reprojection error is minimized.

It is first important to understand that executing the plan computed by the constraint planning is very fast in our system because the r-methods are hard-coded. To give an idea, solving a plan made of 100 equations needs about 13 milliseconds. This explains why the plan execution step is called numerous times in our optimization phase, performed as follows :

1. Based on the plan computed by GPDOF and the variable values calculated by the initialization phase, a *backtracking phase* chooses which solution to select for every r-method.

   Indeed, if the execution of a non-linear r-method in the plan produces at most k different solutions, the number of total solutions given by the plan execution can potentially be multiplied by k. Thus, the total number of solutions is majored by k^r, where r is the

[9]For instance, the under-rigid graph of Figure 12 in Ref. [89] cannot be decomposed into 2×2 subsystems by GPDOF (or by Maximum-matching).

number of non-linear r-methods in the plan[10].

The backtracking phase is a preprocessing step called only once before the numerical optimization. Based on this selection, the following plan executions will always select the same solution to every non-linear r-method in the plan.

2. Then, a standard *numerical optimization algorithm* (a Levenberg-Marquardt algorithm in our system) interleaves input variable modifications and constraint satisfaction phases. The constraint satisfaction is obtained by executing the plan selected by the backtracking phase.

   The numerical algorithm only modifies the values of the input parameters ($\mathcal{P}_1$ and $\mathcal{P}_2$). Every time it does so, the plan is executed, resulting in new coordinate values for all the other scene objects. The cost function is then computed as the reprojection error of all the points.

G.7.1 Backtracking phase

This phase computes one solution with a lowest cost. This problem is called *root identification problem* in Ref. [89]. While some geometric constraint solvers use heuristics to select the desired solution, we resort here to a combinatorial process because the (non-linear) r-methods generally yield several (sub)solutions to the corresponding subsystems.[79, 155]

The precise cost to be minimized is $C = R + \alpha C_S$ (Experiments have led us to choose $\alpha = 1$.) The same cost function is taken into account in the backtracking phase and in the optimization. This multi-criteria cost function C has two components : the well-known reprojection error R, but also a constraint violation cost C_S. The latter is the sum of the constraint violation costs induced by all non-linear r-methods in the plan. The constraint violation cost associated to an r-method m_i is 0 if the r-method succeeds and yields one or more solutions. Otherwise, the constraint violation cost of m_i is a "smooth" measure of the error related to the semantics of constraints. For instance, the error of a point-point distance is the square difference between the distance value in the equation and the actual distance between the points. The error of an incidence constraint is the square of the actual distance between the two related objects.

The backtracking phase is a *branch and bound* algorithm executing all the r-methods in the plan in order. When a non-linear r-method m_i produces several solutions for its output variables, all the solutions are tried, which leads to a combinatorial process. However, this process is not so costly in practice because, among the k solutions given by an r-method, the algorithm tries first the solution that minimizes the reprojection error. In addition, branches of this tree search with a cost greater than the best current cost are cut. Ref. [303] explains how singularities are tackled during this phase.

The backtracking results depends on the initial values of the model variables. It may happen that the solution is not visually correct, especially when the initial reconstruction does not satisfy well the geometrical constraints. However, our experiments have shown that the reprojection error criterion is quite reliable.

[10] Our dictionary contains 7 non-linear r-methods (among 60), each yielding (at most) $k = 2$ solutions.

G.7.2 Exact constraint satisfaction

The optimization process described above satisfies exactly the equations involved in linear r-methods. This is also the case when the execution of non-linear r-methods succeeds. However, the execution of a non-linear r-method m_i may fail. In this case, the latest computed values for the corresponding variables are reused, and a measure of the constraint violations is added to the cost function.

However, after several optimization iterations, the model quality increases and the case occurs that a given non-linear r-method m_i succeeds for the first time. That is, m_i had always given 0 solution during the backtracking phase and in the previous plan execution steps as well. Consider for example an r-method computing the position of a point using distance constraints from 3 other points. It may happen that the initial positions of these points are inconsistent with the distance contraints, so that the r-method yields no solution. However, with the increasing quality of the reconstruction, the point positions can be moved to positions allowing the distance constraints to be satisfied, and the r-method to give the two possible positions for the output point. When m_i succeeds for the first time, among the k possible solutions yielded by m_i, our optimization process selects the one leading to the lowest reprojection error.

This means that the number of unsatisfied non-linear equations decreases as long as the model quality increases. This great behavior highlights the interest of our fast plan execution step included inside the numerical algorithm.

G.8 Experimental Results

We have used our approach to build a model of the *Place Notre-Dame* in Grenoble. A set of images have been used, together with architectural plans from which several distance measurements have been extracted. We have first built a medium-size model constructed from 5 images, called `ND1` hereafter. A larger model, called `ND2`, including peripheric walls and additional details, has then been built from 15 images. The characteristics of these models are reported in Table G.1. Three of the images used for reconstructing the models are shown on Fig. G.12.

	ND1	ND2		ND1	ND2
#images	5	15	#point projections	286	546
#variables	436	819	#equations	273	452
#objects	120	238	#constraints	151	279
#points	90	189	#incidences	124	234
#lines	23	28	#angles	17	34
#planes	7	21	#distances	10	11

Tab. G.1 – The two scenes : Notre Dame (`ND1`) and Extended Notre Dame (`ND2`).

G.8.1 Reconstruction results

The interest of our method is especially well illustrated in Figure G.13. The reconstruction results highlight that models are visually and geometrically correct. The first column contains the top and side views of the initial model, where constraints are respected approximately. The second column contains the top and side views of the model obtained using a standard unconstrained optimization method. One of the points is visible only in two images with a very small baseline and its position is false due to divergence of the optimization process. Other parts of the model also suffer from several artifacts such as an unsatisfied coplanarity. By imposing appropriate constraints, we have overcome these problems. The third column of Figure G.13 contains the top and side views of the model produced by our method. We show how the parts of the model mentioned above have been corrected, leading to a visually correct model.

(a) (b) (c)

FIG. G.12 – Three images from the sequence *Notre Dame*. Image (c) has been included only into the `ND2` sequence.

Several artifacts have been corrected after several optimization steps, which highlights the interest of our optimization phase and of our fast plan execution (due to r-methods). Also, when in the initial reconstruction the constraints are not sufficiently satisfied, it may happen that a plan execution causes artifacts in the scene, such as in the center of Figure G.14–(a). The optimization corrects the errors created this way (see Figure G.14–(b)).

Table G.2 (top) reports statistics about the number of r-methods added automatically to the equation graph (#r-methods), the number of r-methods selected in the plan (plan size), the number of bundle adjustment iterations (#iterations) and plan executions (#executions) performed by the optimization. Each optimization iteration calls several plan executions (on average, $185 = \frac{2040}{11}$ for `ND1` ; 462 for `ND2`), and chooses one of them according to the criterion. The reprojection error (reproj. error) and the number of violated constraints (#violated) is given before the optimization (i.e., after a single plan execution), and also after the optimization.

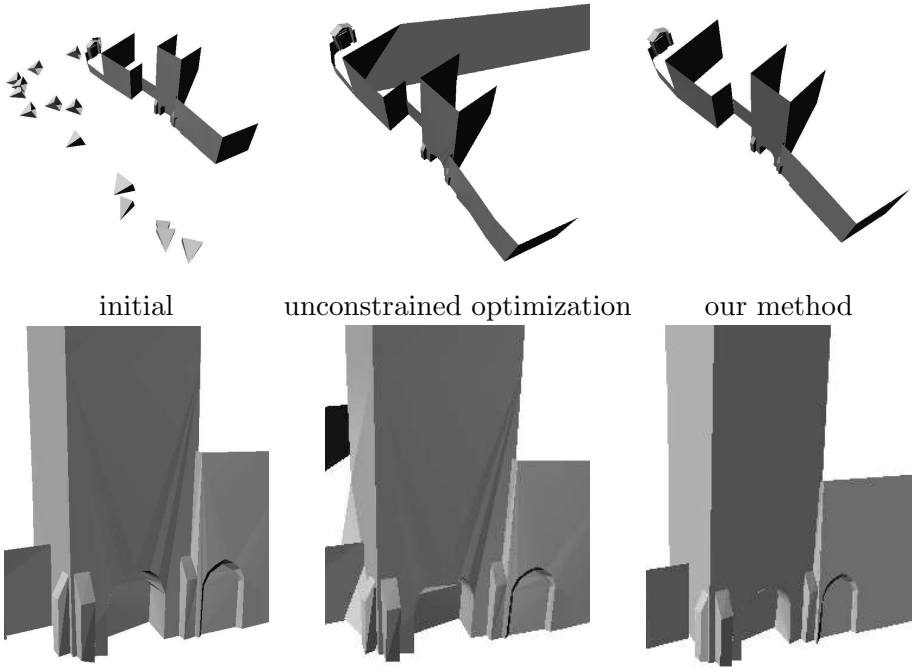


FIG. G.13 – Reconstruction of the ND2 scene. Top (1^{st} row) and side (2^{nd} row) views of the initial model (1^{st} column) ; the model obtained using unconstrained optimization (2^{nd} column) ; the model obtained using our method (3^{rd} column).

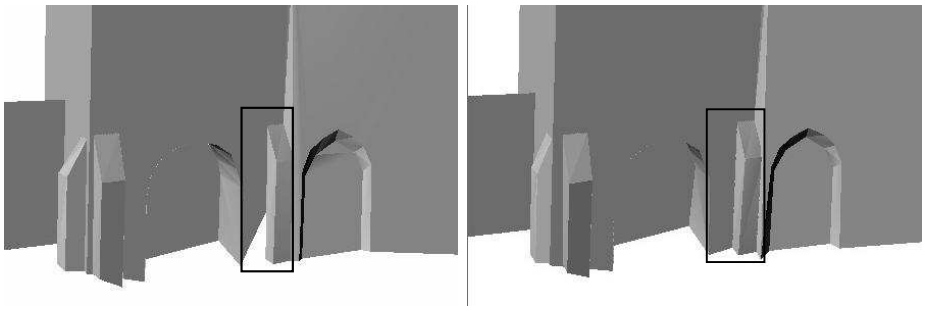
G.8.2 Performance tests

All the times reported below have been obtained with a Linux operating system on a Pentium IV 2 Ghz processor.

Recall that our r-method dictionary contains 60 r-methods. The most complex r-methods solve 3 geometric constraints (6 equations) and involve 4 geometric objects (1 as output and 3 as input).

We have first evaluated the time required to execute one r-method. This time varies from 20μs to 90μs. The execution time of non-linear r-methods is shorter ($\sim$ 28μs on average) because the corresponding procedures are hard-coded, while linear r-method routines use a generic SVD (Singular Value Decomposition). Table G.2 (bottom) details the times spent in the different phases of our model reconstruction system.

The time for the initialization phase is dominated by the non-linear unconstrained optimisation process ($\sim$ 80% of the total time) which is executed to refine the initial, parallelepiped-based calibration. A very interesting characteristic of our system is that the constraint planning phase (automatic r-method addition, GPDOF and backtracking) requires only a few seconds. Note also that the time required for executing a plan is really impressive. With ND2, the 451 equations can be solved 7866 times in 467 s (only one solution per subsystem is chosen). This means that the hard-coded r-methods allow us to solve 100 equations in 13 milliseconds on average !



(a) (b)

FIG. G.14 – Part of the ND2 scene after (a) one step of GPDOF ; (b) optimization : initial artifacts have been corrected.

	ND1	ND2		ND1	ND2
# r-methods	3017	6695	Plan size	118	228
# iterations	11	17	$\mathcal{P}_1$	158	352
# executions	2040	7866	$\mathcal{P}_2$	10	25
Reproj. error before	20.42	23.01	Reproj. error after	4.038	4.16
# violated before	2	2	# violated after	2	1

Phase	ND1	ND2
Initialization	21	331
R-method addition	0.7	1.7
GPDOF	1.4	3.9
Backtracking	0.2	0.2
Optimization	53	467

TAB. G.2 – Statistics on our GPDOF-based model reconstruction (top), and performance of the different phases in seconds (bottom).

Remark

Table G.2 (bottom) clearly shows that the exploration of all the connected subgraphs of size at most k is fast (0.72 s for ND1). As mentioned in Section G.5.1, the time complexity of this phase is highly dominated by the number of connected subgraphs. This number strongly depends on the size k of the largest subgraph. In the first version of our tool,[306] this phase was even more time-consuming (253 s on ND1). The exploration of the constraint graph considered connected subgraphs in terms of objects *and* constraints. The value of k was 7 because the largest r-methods in the dictionary include 3 geometric constraints and 4 objects. In the new version, $k = 3$ because the connected subgraphs are built in terms of constraints only. Two constraints are neighbors in the constraint graph iff they share an object.

This means that our simple automatic r-method addition algorithm can handle in practice

any type of r-method that outputs a single object (point, line or plan), even if other types of constraints are added, such as angles, distance ratios.[303]

G.8.3 Comparison with a penalty function method

We have compared our model reconstruction system with a classical constrained optimization method where the constraints introduce a penalty to the cost function.[236, 199] No free tool was rapidly available so that we have developed a simple version in our architecture.[303] Based on ND1 and ND2, we have created two models respecting all the geometrical and projection constraints perfectly. We then have introduced an increasing Gaussian noise on 2D coordinates on one hand and on 3D coordinates on the other. The comparisons between the two approaches are the following :

- The penalty-based approach is general, although it is difficult to compare penalties related to different types of constraints, and to tune the coefficient K increasing the constraint violation part in the cost function (as compared to the reprojection error) if one wants to avoid that the penalty-based method diverges.
- The visual aspect of the GPDOF-based method is very good ; the visual aspect of the penalty-based method is correct, although we can notice that the constraints are not exactly solved.
- The time required by the penalty-based method is about 2 or 3 times higher than the time spent by the GPDOF-based method.
- The constraint error with our tool is null, but is low with the penalty-based method, provided that a right value has been selected for the coefficient K.
- With very noisy initial data, the GPDOF-based method yields a significant reprojection error.

This study would suggest the interest of a hybrid optimization method where a call to the GPDOF-based method would follow a call to the penalty-based one to impose the constraints exactly. Also, another variant would use a final single execution of the plan produced by GPDOF.

G.9 Related Work in Computer Vision

(A more detailed version of this section can be found in Wilczkowiak's PhD thesis.[303])

In Computer Vision, geometrical constraints are traditionally expressed as a set of algebraic equations among real-valued variables and are solved by numerical methods. Depending on the complexity of the considered problem and the variety of considered objects and constraints, different approaches have been used.

Many systems limit the set of available constraints to collinearity, coplanarity and parallelism, so that linear approaches allow for fast scene reconstruction.[275, 116]

When dealing with uncalibrated cameras, and more complicated constraints, it is necessary to use non-linear methods. Constrained minimization techniques, such as Lagrange multipliers (see Ref. [200]) or the penalty-based method described above,[236, 199] do not guarantee however that the final model respects *exactly* the constraints, and lead to convergence problems because

it is difficult to choose the weights associated to different types of constraints.

Ref. [37] describes an elegant solution to compute a set of input parameters for systems of bilinear constraints. The approach is based on a symbolic method, but the computational cost turns out to increase very quickly with the number of variables in the system, making the use of this method difficult for large systems.

In Ref. [24], the scene is represented by points, segments, and parallelograms. The constraints between them are used to reduce the number of parameters representing the scene objects by applying rewrite rules. However, the local application of the rewrite rules does not guarantee that a solution satisfying all the constraints in the scene will be found.

G.10 Conclusion

We have presented a solution to the problem of decomposing and solving large under-constrained systems of geometric constraints in the 3D space. An application to 3D modeling from images has shown that plunging the dictionary of r-method patterns in the actual system and decomposing the enriched equation graph with `GPDOF` run in several seconds. The obtained plan can be executed in hundredths of second.

We should highlight that the approach is dedicated to, but not limited to, geometric constraints. The general-purpose `GPDOF` works at the equational level and can thus easily take into account systems including geometric and non geometric components.

Several simple numeric and symbolic solutions related to singularities and redundant constraints have been implemented. Further developments should be performed to detect other cases of redundant constraints.

In conclusion, we hope that the geometric constraint community will pay more attention to the `GPDOF` decomposition method which appears to be very useful for tackling under-constrained systems when approximate values are known for the variables.

Acknowledgments

Thanks to Edmond Boyer, Christophe Jermann and Peter Sturm for the collaboration on previous stages of this work. Thanks to David Daney, Bertrand Neveu and Théodore Papadopoulo for useful discussions. Thanks to Gilles Chabert for comments. Thanks to Mr. Dominique Chancel for the architectural drawings.

Annexe H

Decomposition of Geometric Constraint Systems : A Survey

Article [156] : publié dans la revue IJCGA (Int. Journal of Computational Geometry and Applications) en 2006

Auteurs : Christophe Jermann, Gilles Trombettoni, Bertrand Neveu, Pascal Mathis

Abstract

Significant progress has been accomplished during the past decades about geometric constraint solving, in particular thanks to its applications in industrial fields like CAD and robotics. In order to tackle problems of industrial size, many solving methods use, as a preprocess, decomposition techniques that transform a large geometric constraint system into a set of smaller ones.

In this paper, we propose a survey of the decomposition techniques for geometric constraint problems[1]. We classify them into four categories according to their *modus operandi*, establishing some similarities between methods that are traditionally separated. We summarize the advantages and limitations of the different approaches, and point out key issues for meeting industrial requirements such as generality and reliability.

Keywords : geometric constraints ; decomposition techniques ; rigidity ; DOF analysis ; connectivity analysis ; DR-planner ; maximum matching ; PDOF ; WCM.

[1]The french CNRS has supported the working groups "AS contraintes géometriques" and "MathSTIC geometric solvers".

H.1 Introduction

The decomposition of constraint systems is an avatar of the well-known divide and conquer paradigm. Applied to geometric constraint systems, it gives birth to general algorithms that answer to the following questions :

– How to cut a system into several smaller subsystems easier to solve ?
– How to solve every subsystem separately ?
– How to merge the solutions of subsystems for producing the solutions to the whole system ?

Preliminary example

Let us consider the simple example shown in Fig. H.1 made of points A, B, C, D and lines $\delta_1, \delta_2, \delta_3$. This dimensioned sketch (left side) can be cut into two parts (right side) : the subsystem (1) and the subsystem (2).

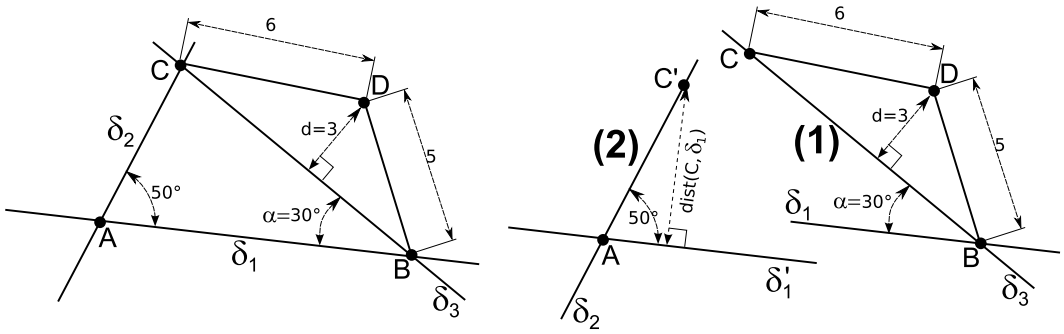


FIG. H.1 – **Left :** A simple dimensioned sketch. **Right :** A decomposition into two subsystems.

In the literature, several ways to produce such a decomposition have been described. Some algorithms identify subsystem (1) as an interesting (i.e., solvable) subsystem ; other methods find *points of weakness* in the constraint system like (C, δ_1) ; other techniques identify first the subsystem (2) as solvable provided that the complementary subsystem is solvable as well.

This example also shows an interesting phenomenon. The subsystem (2) is not rigid and a metric information coming from the subsystem (1) has to be added, namely the distance between C' and δ'_1 (dotted in Fig. H.1). This implies an order : the subsystem (1) has to be solved first to determine the value of this distance.

Usually, a geometric subsystem is translated into a system of algebraic equations that is handled by standard numerical or symbolic solving methods. Other solving methods work at the geometric level by applying well-known construction steps, like ruler&compass ones. When subsystems are well-constrained up to displacements, the algorithms need to *pin* some elements (e.g., the point D and the direction of the line δ_3 in the subsystem (1)) in order to make them solvable.

Once the subsystems have been solved, the *subsolutions* are assembled. In our example, this is

done using geometric transformations that make coincide point C' with point C, and line δ_1' with line δ_1.

Objectives and contribution

Other surveys provide different, and sometimes more detailed, views over the field. In Ref.[130], Hoffmann et al. propose *desirable properties* for decomposition methods and analyse in details different graph-based methods w.r.t. these properties. In Ref.[128], Hoffmann and Joan-Arinyo present a less detailed survey, however providing an important list of references to decomposition and solving algorithms and systems. In Ref.[270], Sitharam presents graph-based approaches, detailing how her system FRONTIER implements such a method, discussing the issues (constriction misclassification, solution browsing, dealing with over-constriction) of the approach and proposing new techniques to handle them.

However, solving and decomposition phases are often mingled in the description of geometric constraint solvers. The comparison between methods is also difficult because different terminologies are used. This paper aims at a didactic, synthetic and homogeneous presentation of the existing decomposition algorithms. General definitions introduce the required concepts in Section H.2. Based on only a few common data structures, in particular graphs, our presentation highlights the underlying operating principles, i.e., the ideas behind the techniques.

Thus, instead of a linear catalogue of methods, we propose a classification of the methods in four categories and illustrate them with examples (Sections H.3 to H.6). The homogeneous presentation highlights advantages and drawbacks of the methods, and allows us to compare them (Section H.7). Section H.8 provides guidelines to meet industrial requirements. Finally, Section H.9 presents a promising numerical method, called the witness configuration method, that handles several pathological cases.

H.2 Definitions

This section defines the basic notions and properties necessary to comprehend decomposition methods.

H.2.1 Constraint systems

A constraint system describes a set of properties (the constraints) that must be satified by a set of variables. The description respects syntactic rules and the variables have to take their values in specific sets. Constraint systems have thus two sides : syntactic and semantic.

Definition 1 *Given a language with a semantics, a* **constraint system** *is a triple $S = (C, X, A)$, where C is a set of* constraints *(terms of the language), X is a set of variables and A is a set of parameters.*

In *geometric* constraint systems, the variables are also called *geometric entities*. We distinguish the variables from the parameters. The variables are the unknowns to be determined while the parameters are provided, for instance, by a simulation or the user. Consider the system below :

$$\begin{cases} c_1 : dis(A, \delta_1) = 0 & c_7 : ang(\delta_1, \delta_2) = 50 \\ c_2 : dis(A, \delta_2) = 0 & c_8 : ang(\delta_1, \delta_3) = \alpha \\ c_3 : dis(B, \delta_1) = 0 & c_9 : dis(B, D) = 5 \\ c_4 : dis(B, \delta_3) = 0 & c_{10} : dis(C, D) = 6 \\ c_5 : dis(C, \delta_2) = 0 & c_{11} : dis(D, \delta_3) = d \\ c_6 : dis(C, \delta_3) = 0 \end{cases}$$

This constraint system is composed of 11 constraints (c_1 to c_{11}) and 7 variables (A to δ_3). If the system corresponds to the sketch in Fig. H.1, the 2 parameters α and d are respectively set to 30 and 3. Its description language implies several types (e.g., `Point`, `Angle`) and typed functional/predicative symbols (e.g., `dis(Point,Point) :Length`, `ang(Line,Line) :Angle`). This syntax has two different meanings, resulting in two different constraint systems : in 2D, a system called Geo_2 ; in 3D, a system called Geo_3.

The purpose of a constraint system is to encode in an elegant and concise manner a set of specific assignments for its variables, the so-called solutions.

Definition 2 *Let $S = (C, X, A)$ be a constraint system and let θ_A be a valuation of the parameters in A. A* **solution** *to S is a valuation θ_X of the variables in X such that every predicate in C is true. The set of solutions to S is denoted by $Sol(S)$.*

The values of the parameters are generally fixed before a system is solved. In our example, α and d could be fixed using the dimensioned sketch in Fig. H.1.

The notion of *subsystem* is paramount in decomposition methods since they are all based on the identification of solvable subparts.

Definition 3 *Let $S = (C, X, A)$ be a constraint system. A* **subsystem** *S' of S is a constraint system $S' = (C', X', A')$, where $C' \subseteq C$, and X' and A' are respectively the variables and the parameters related to C'.*

Being a subsystem is a syntactic property that does not depend on the interpretation of the constraint system. A straightforward property states that $Sol(S') \supseteq Sol(S)|_{X'}$.

H.2.2 Representation of geometric constraint systems

The presented methods operate with different representations of geometric entities and systems.

Representation of geometric constraints and entities

Several methods use a (full) geometric description of entities. They manage geometric entities, such as lines, points, planes and circles, that must satisfy geometric constraints such as distances, angles, etc.

Other algorithms work at the equational level. Geometric constraint systems are represented by systems of equations over the reals using a modeling of the geometric entities and constraints. For instance, a 2D point can be modeled by its Cartesian coordinates (x, y), and a 2D line by its polar coordinates (d, a). A point-point distance can be modeled by the classical equation $(x_1 - x_2)^2 + (y_1 - y_2)^2 = d^2$ and a line-line angle simply by $a_2 - a_1 = \alpha$.

Finally, some methods abstract the entities by their number of *degrees of freedom* (DOFs). For instance, these methods do not distinguish points and lines in 2D because both have 2 DOFs. They are often more general but less reliable.

The number of **DOFs** of a geometric entity is the number of independent coordinates used to represent it. It is equal to 2 for points and lines in 2D. The number of **DOFs** of a geometric constraint is the number of independent equations needed to represent it. For instance, angle or distance constraints have 1 DOF in 2D and in 3D. A parallelism between two planes has 2 DOFs in 3D. In the following, we denote by $dof(i)$ the number of DOFs of an entity or a constraint i.

Representation of systems of geometric constraints

Only a few decomposition methods manipulate directly the constraint system by using rewriting rules. Most of them use a graph-based abstraction of the system.

Definition 4 *Let $S = (C, X, A)$ be a constraint system. Its* **constraint graph**, *denoted by $G_S = (V, E)$, is an undirected graph where $V = X$ (every variable in S is a vertex in G_S) and $E = C$ (every constraint in S is an edge in G_S).*

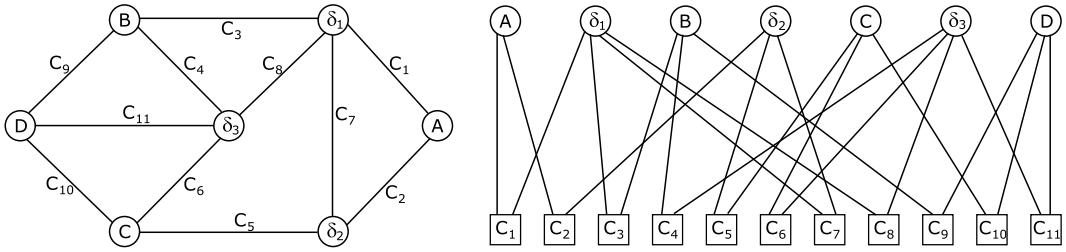


FIG. H.2 – Constraint graph representing Geo_2.

Figure H.2–left shows a constraint graph representing Geo_2. The entities (points and lines) are the vertices, the constraints (distances and angles) are the edges.

When the constraints are not binary (e.g., angles between triple of points), this graph includes hyper-edges. Instead of a hypergraph, one commonly uses a *bipartite constraint graph* where

both the constraints and the variables are vertices ; an edge connects each constraint to each entity it constrains (see Fig. H.2–right).

The methods working at the equational level operate with an **equation graph** the vertices of which are equations and entities coordinates (also called variables). The equation graph of Geo_2 is depicted in Fig. H.3. This graph can be seen as an expanded version of the bipartite constraint graph in Fig. H.2–right.

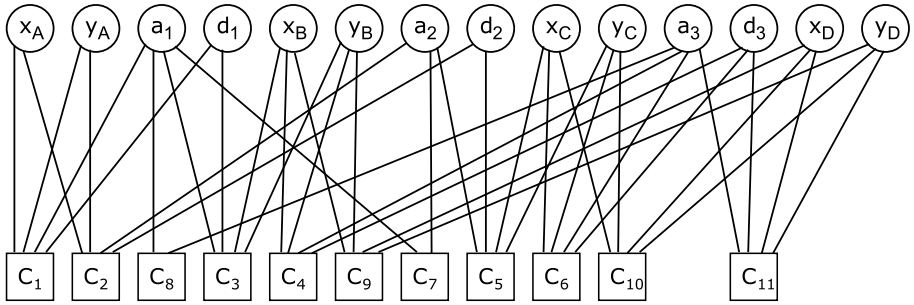


FIG. H.3 – Equation graph representing Geo_2.

H.2.3 Constriction

When studying a constraint system, the dimension of its solution space is important : whether the set of solutions is empty, finite or infinite changes the appropriate way to solve the problem. The notion of *constriction* captures this property.

Definition 5 ***(Constriction)*** *A constraint system S is* well-constrained *if $Sol(S)$ is finite,* over-constrained *if $Sol(S) = \emptyset$ and* under-constrained *if $Sol(S)$ is infinite.*

Geometric constraint systems are generally under-constrained, like Geo_2 and Geo_3, but their solutions are often identical modulo a geometric transformation (e.g., translation, rotation). Systems with such solutions are called *invariant*.

Definition 6 ***(Invariance)*** *Let $S = (C, X, A)$ be a constraint system, Θ_X be the set of all possible valuations of X. Let φ be a bijection from Θ_X onto itself. S is* **invariant** *by φ iff $s \in Sol(S) \Leftrightarrow \varphi(s) \in Sol(S)$. Given a group G of bijections from Θ_X onto itself, S is* **G-invariant** *iff it is invariant by every $\varphi \in G$.*

We denote $\mathbb{I} = \{id\}$ the trivial group reduced to the identity bijection. We denote $\mathbb{D}$ ($\mathbb{D}_2$ in 2D, $\mathbb{D}_3$ in 3D) the group of direct isometries. Geo_2 is $\mathbb{D}_2$-invariant and Geo_3 is $\mathbb{D}_3$-invariant. The interesting constraint systems considered in CAD are often $\mathbb{I}$-under-constrained while being $\mathbb{D}$-invariant. Hence, it is worth considering constriction modulo invariance groups.

Definition 7 ***(Constriction modulo a group or G-constriction)*** *Given a constraint system S and a group G of bijections from Θ_X onto itself, S is* well-constrained modulo G *(or* G-well-constrained*) iff the orbit set $Sol(S)/G$ is finite. It is G-over-constrained iff $Sol(S)/G$ is empty, and G-under-constrained iff $Sol(S)/G$ is infinite.*

For example, Geo_2 is $\mathbb{D}_2$-well-constrained while Geo_3 is $\mathbb{D}_3$-under-constrained since the quadrangle $ABCD$ can be folded continuously along line δ_3 in 3D. Systems of equations yielding a 0-dimensional solution space (i.e., having a finite number of solutions) are $\mathbb{I}$-well-constrained. $\mathbb{D}$-constriction is often referred to as *rigidity*.

The usual way to express (and to find) the solution set of an under-constrained system is to give a finite set of particular solutions $\{f_1, \ldots, f_k\}$ and an invariance group G. For instance, Geo_2 yields (generally) 2 particular solutions modulo $\mathbb{D}_2$, say f_1 and f_2, and $Sol(Geo_2) = \{\varphi(f_1)|\varphi \in \mathbb{D}_2\} \cup \{\varphi(f_2)|\varphi \in \mathbb{D}_2\}$.

H.2.4 Approximate characterizations of constriction

A crucial step in a decomposition method is to determine whether a candidate component is well-constrained (sometimes modulo a group G) or not. The constriction depends on the number of solutions. However, computing all the solutions is intractable, and would render the decomposition useless. Hence, approximate characterizations that can be checked in polynomial time are frequently used.

Pattern-based characterization

Several methods use a repertoire of known well-constrained systems. Then, checking whether a component is well-constrained amounts to matching it to a system in the repertoire. These systems are in the form of either predefined patterns or construction rules. $Pattern_1 = (C = \{dis(P, L) = k\},\ X = \{Line\ L,\ Point\ P\},\ A = \{Length\ k\})$ and $Pattern_2 = (C = \{dis(P_1, P_3) = l, dis(P_2, P_3) = k\},\ X = \{Point\ P_3\}, A = \{Point\ P_1,\ Point\ P_2,\ Length\ l,\ Length\ k\})$ are two such patterns. The first one matches $\mathbb{D}$-well-constrained components in 2D and 3D. The second one matches $\mathbb{I}$-well-constrained components in 2D only.

The patterns can also be implemented by geometric rules. For instance, $Pattern_2$ corresponds to two applications of the geometric rule *"if point P and length m are known, and $dis(P, Q) = m$, then point Q is on a determined circle C"*, and one application of the rule *"if point P is on two known independent loci $L1$ and $L2$, then P is determined"*. The rule version follows a logical approach and makes more explicit the construction associated to the corresponding pattern, yielding self-explainatory decompositions.

Pattern/rule-based characterizations use all the information available in a constraint system, i.e., its syntax and its semantics. That is why they often work directly on the system and not on an abstraction of it. This generally makes them correct, that is, a pattern-matched component is usually really well-constrained. Unfortunately, pattern-based characterizations are necessarily

incomplete because no finite repertoire of patterns can cover every well-constrained component in the general case, as stated by the following proposition.

Proposition 12 *In 2D and 3D, there exist irreducible $\mathbb{D}$-well-constrained systems of arbitrary size.*

In Ref.[149], page 2^7, such systems are composed of constraints with 1 DOF (e.g., distances) and entities with 3 DOFs (resp. 6 DOFs). They are generated by simple construction steps similar to Henneberg constructions.[125]

DOF-based approximations

A well-known characterization of constriction is based on the DOF abstraction of the geometric constraints and entities. It is often called *structural rigidity* because, for several decomposition methods using this characterization, the system is abstracted by a constraint graph in which vertices (i.e., entities) and edges (i.e., constraints) are labeled by their respective DOFs.[87]

Definition 8 ***(Structural G-constriction)*** *Let $S = (C, X, A)$ be a constraint system. Let G be an invariance group of dimension $\mathcal{D}$.*

The system S is structurally G-over-constrained *iff there exists a subsystem $S' = (C', X', A')$ of S such that $\sum_{x \in X'} dof(x) - \sum_{c \in C'} dof(c) < \mathcal{D}$.*

The system S is structurally G-well-constrained *iff it is not structurally G-over-constrained and $\sum_{x \in X} dof(x) - \sum_{c \in C} dof(c) = \mathcal{D}$.*

The system S is structurally G-under-constrained *iff it is not structurally G-over-constrained and $\sum_{x \in X} dof(x) - \sum_{c \in C} dof(c) > \mathcal{D}$.*

Hence, $\mathbb{I}$-constriction can be checked using $\mathcal{D} = 0$. $\mathbb{D}_2$-constriction (resp. $\mathbb{D}_3$-constriction) can be checked using $\mathcal{D} = 3$ (resp. $\mathcal{D} = 6$) which is the number of independent displacements in 2D (resp. 3D).

This characterization derives from Laman's theorem which characterizes the rigidity of *bar frameworks*, i.e., geometric constraint systems composed of points constrained by (generic) distances, systems much studied in the Rigidity Theory.[115]

Theorem 1 *(Laman, 1970) A constraint system in the 2D plane composed of n points linked by m generic distances is rigid if and only if $2 \times n - m = 3$ and for any subsystem composed of n' points and m' distances, $2 \times n' - m' \geq 3$.*

This theorem yielded several polynomial time algorithms.[189, 61, 124] More complicated combinatorial properties have been proposed to take into account constraints such as directions of pairs of points in 2D.[266] Unfortunately, except for 2D bar frameworks and similar systems,

the structural rigidity does not imply the (geometric) rigidity. It is only an approximate characterization of it. A famous counter-example is reported in Fig. H.4, where segments represent point-point distances in 3D. This system is structurally $\mathbb{D}_3$-well-constrained but actually $\mathbb{D}_3$-over-constrained[2].

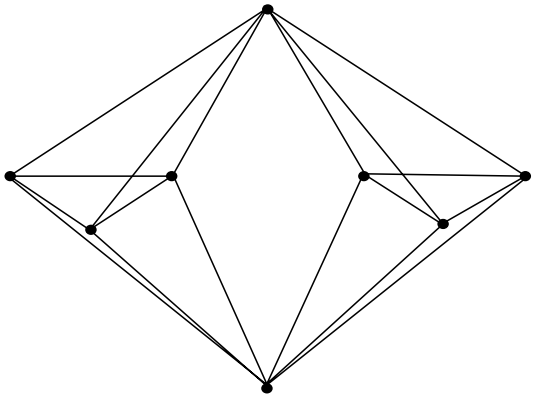


FIG. H.4 – The double-banana famous counter-example.

Another combinatorial characterization of rigidity for 3D bar frameworks, called module-rigidity, has been conjectured by Sitharam and Zhou.[191] It can deal with systems such as the double-banana. Module-rigidity is achievable by a polynomial algorithm and has been implemented in the FRONTIER solver.[269]

Another strong limitation of the structural characterization of rigidity ($\mathbb{D}$-constriction) is that it does not handle correctly *projective* and other *boolean* constraints such as incidences and parallelisms. However, these constraints are widely used in CAD systems. Jermann et al. introduced the notion of *degree of rigidity* which replaces, in the structural constriction, the dimension $\mathcal{D}$ of the invariance group G by a value depending on these constraints.[153]

Graph-based approximations

Another well-known approximate characterization of rigidity also derives from the Rigidity Theory community.[115] It relies on a *connectivity analysis* applied to the constraint graph of a system.

For general 2D geometric constraint systems, a necessary condition for rigidity derives from a corollary of Theorem 2.5 in Ref. [94] which states that the constraint graph must be "at most" triconnected, that is, it must contain no 4-connected subgraph. However, this condition is not sufficient. For instance, the complete graph composed of 4 points linked by 6 distances in 2D is triconnected but it is $\mathbb{D}_2$-over-constrained.

[2]This system is over-constrained in the generic case, if the two bananas have arbitrary heights. In the singular case in which the two bananas have the same height, the system is under-constrained since the two "bananas" can fold continuously along the line passing through their extremities...

Similar necessary conditions have been proposed for 3D systems with points, lines and planes.[94] $\mathbb{D}_3$-well-constrained geometric systems composed of points and planes must be "at most" 5-connected. $\mathbb{D}_3$-well-constrained geometric systems composed of points, lines and planes must be "at most" 7-connected.

H.2.5 Decomposition

Decomposition methods transform a constraint system into a set of small solvable subsystems. To obtain the solutions of the constraint system, the (sub)solutions of its subsystems are computed and then assembled.

Definition 9 *A* **decomposition** *of a constraint system S is a triple $(\{S_1, \ldots, S_n\}, \prec, \uplus)$ where :*

- *Each S_i is a G-well-constrained constraint system called a* component*.*
- *$\prec$ is a* partial order *for the solving of the S_is.*
- *$\uplus$ is a solution* assembling operator *such that if f_i is a solution of S_i, $\uplus(f_1, \ldots, f_n)$ is either an empty assignment (the f_is are incompatible) or a solution of S.*

The components of a decomposition are in general subsystems of the original system[3].

The partial order expresses dependences between the components : S_j depends on S_i (noted $S_i \prec S_j$) if S_j becomes G-well-constrained only once a solution of S_i is given. These dependences may derive from variables shared by components : among the components sharing a given variable, one will compute its value and the other ones will use this value as a parameter for their own solving (see example below).

The assembling operator depends on the group G under which the components are well-constrained. If $G = \mathbb{I}$ (i.e., no invariance), then the assembling amounts to simply concatenating the solutions of the components into a solution of the original problem. When $G = \mathbb{D}_2$ or $\mathbb{D}_3$, then it amounts to computing direct isometries which make coincide the variables shared by components. The assembling operator is applied in an order related to the partial order $\prec$.

For example, Geo_2 can be decomposed as follows :

- Components :
  - $S_1 = (\{dis(B, \delta_1) = 0, dis(B, \delta_3) = 0, dis(C, \delta_3) = 0, ang(\delta_1, \delta_3) = 60, dis(B, D) = 5, dis(C, D) = 5, dis(D, \delta_3) = 3\}, \{B, C, D, \delta_1, \delta_3\}, \emptyset)$
  - $S_2 = (\{dis(A, \delta_1') = 0, dis(A, \delta_2) = 0, ang(\delta_1', \delta_2) = 40, dis(C', \delta_2) = 0, dis(C', \delta_1') = dis(C, \delta_1)\}, \{A, C', \delta_1', \delta_2\}, \{C, \delta_1\})$
- Partial order : $S_1 \prec S_2$
- Assembling operator : displacement to make C coincide with C', and δ_1 with δ_1'

[3]However, some algorithms modify the original system by incorporating a metric information (extracted from some subsystems and reported in others). The corresponding components are thus not subsystems of the original problem.

This decomposition contains 2 components S_1 and S_2 corresponding to those depicted in Fig. H.1. S_1 is a $\mathbb{D}_2$-well-constrained component containing 7 constraints to determine 5 variables, the points B, C and D and the lines δ_1 and δ_3. S_2 solves 4 variables, the points A and C' and the lines δ_1' and δ_3, using 5 constraints. The constraint $dis(C', \delta_1') = dis(C, \delta_1)$ corresponds to the metric information transferred from S_1 to S_2. This constraint renders S_2 $\mathbb{D}_2$-well-constrained and ensures that assembling the solutions of S_1 and S_2 is always possible. Because of this constraint, C and δ_1 are parameters of S_2, implying the dependence $S_1 \prec S_2$.

H.2.6 Decomposition methods

Geometric constraint solvers often work in two phases : decomposition and solving. This paper only focusses on the decomposition phase. The reader will find in the literature details about the solving methods : local numerical methods,[137, 179, 208] continuation,[75] interval analysis,[211, 158, 155] symbolic computation,[307, 57] ruler and compass.[197, 306]

We propose to classify the existing decomposition methods into four categories with respect to the way they operate :

- the *recursive division* methods work iteratively. At each iteration, current components are split along assembly points, producing several subcomponents where additional constraints are introduced to maintain the consistency of the assembly points.
- the *recursive assembly* methods also work iteratively. At each iteration, they identify a component and condense it into a new set of variables and constraints in the constraint system.
- the *single-pass* methods produce all the components simultaneously. Most of these methods are based on the maximum matching theory.
- the *propagation of degrees of freedom (PDOF)* methods iteratively identify components in reverse order of their solving.

This classification is not traditional and allows us to establish methodological similarities between existing approaches.

In addition to an algorithmic description illustrated by examples, we provide the following information for each category of methods[4] :

- *Confluence* : a method is confluent if any choice during the running process leads to the same decomposition.
- *Completeness* : a method is complete if it always returns a decomposition when there exists one. In case a method is incomplete, we explicit restrictions under which it becomes complete.
- *Complexity* : the time complexity that generally depends on the size of the constraint system.
- *Correctness* : a method is correct if the obtained components are really G-well-constrained. In case a method is incorrect, we identify restrictions under which it is correct.

[4] A set of properties of decomposition methods has been proposed in Ref.[130], with an emphasize towards graph properties. In this paper, we prefer using more traditional algorithmic notions.

H.3 Recursive Division Approaches

Recursive division methods, also called division analysis, work by iteratively splitting the constraint system into components, themselves subject to further splitting. In 1991, Owen introduced the first approach of this kind.[234] This method handles 2D constraint systems with points and lines linked by distance and angle constraints. It uses the *constraint graph* presented in Sec. H.2.2 and the *biconnection* and *triconnection* conditions introduced in Sec. H.2.4.

The constraint graph G_S is decomposed at its *articulation pairs*, i.e., pairs of vertices whose removal splits the graph into disconnected components. Given an articulation pair (v_1, v_2), G_S is split into several subgraphs $(G_1, ..., G_n)$ by duplicating v_1 and v_2 in each subgraph. An edge, called a *virtual bond*, is introduced between v_1 and v_2 in each subgraph G_i except if there is already an edge between v_1 and v_2 in G_i or if G_i is biconnected, i.e., there exist 2 distinct paths between any pair of vertices in G_i. Virtual bonds represent constraints that fix the relative positions of the variables shared by subgraphs (see Fig. H.1). An interesting property states that if S is $\mathbb{D}_2$-well-constrained, then one of the subgraphs (say G_1) is $\mathbb{D}_2$-well-constrained and the others are $\mathbb{D}_2$-under-constrained (see Ref. [73] for a proof). Hence, a virtual bond is not needed in G_1. When constraints are already present between articulation pairs, they are simply duplicated in all the subgraphs, avoiding the use of virtual bonds.

This process is recursively repeated on each subgraph until no more splitting is possible. In the end, every remaining subgraph is either an edge, a triangle or a triconnected subgraph ; they form the set of components in the decomposition. The partial order between the components is induced by the virtual bonds : first components without virtual bonds, then the others. The assembly operator has to be applied everytime a virtual bond must be determined. It consists in computing a displacement to make coincide shared objects. Figure H.5 shows the application of Owen's method to Geo_2.

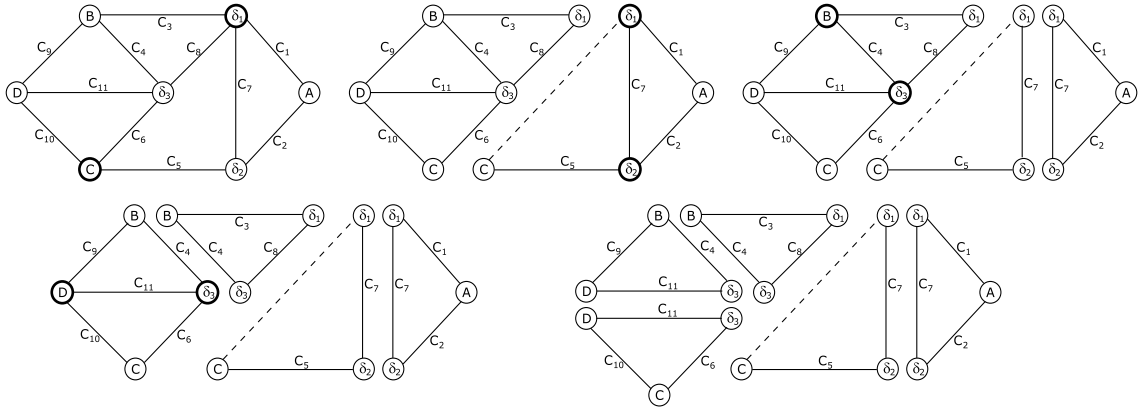


FIG. H.5 – Decomposition of Geo_2 by Owen's method.

This method has been extended to tackle $\mathbb{D}_2$-under-constrained systems using a completion mechanism which adds constraints in order to obtain $\mathbb{D}_2$-well-constriction.[89, 161] Fudos and

Hoffmann described a similar division method that is hybridized with a recursive assembly technique.[89] It is proved in Ref. [162] that their technique has the same power as Owen's method and obtains the same decomposition on $\mathbb{D}_2$-well-constrained systems. Gao and Zhang have extended the approach to 3D constraint systems made of points, planes and lines by ensuring 5-connection or 7-connection of the graph (see Sec. H.2.4).[94]

Properties

These methods suffer from strong limitations, mostly because they use graph-based approximations (connectivity) of $\mathbb{D}$-constriction, rendering them correct only when these characterizations match $\mathbb{D}$-constriction (see Sec. H.2.4). This also introduces another cause of incorrectness : the identified assembly pairs do not always constitute valid junctions. For instance, an articulation pair made of 2 parallel lines cannot be used to assemble components. Indeed, an infinite number of displacements can make coincide these lines, producing an infinite number of assembled solutions. Finally, this makes them incomplete in general. For instance, there exist triconnected constraint graphs that can be decomposed further. This means that the method cannot decompose all decomposable geometric constraint systems. Fig. H.13–left will show an example.

These methods run in polynomial time because they use polynomial time procedures to check k-connectivity.[133] For instance, Owen's method runs in $O(n^2)$ when the system has n constraints. They are also confluent since a choice of a given articulation pair does not eliminate another possible pair. Thus, in the end, the resulting decomposition is a unique set of components in the constraint graph.

H.4 Recursive Assembly Approaches

Unlike recursive division methods, recursive assembly methods adopt a bottom-up scheme. They iteratively aggregate $\mathbb{D}$-well-constrained (i.e., rigid) components into bigger ones. These methods are often referred to as *recursive rigidification*, *reduction analysis* or *decomposition-recombination* methods (DR-planners). If the constraint system contains both well-constrained and under-constrained subparts, recursive assembly methods return a maximal set of maximal well-constrained components.

H.4.1 Overview

Recursive assembly methods are all based on the same scheme :

1. **Identification phase :** Find a rigid subsystem, called *cluster*. This phase is itself divided into two steps :
   (a) **Merge step :** Merge several variables linked by constraints into a new rigid cluster K.

(b) **Extension step :** Perform as many *single extensions* as possible. A single extension adds a subset of variables and constraints to K such that K remains $\mathbb{D}$-well-constrained.

2. **Contraction phase :** This phase is itself divided into two steps :

   (a) **Registration step :** Place K in the current decomposition and update the partial order.

   (b) **Replacement step :** Replace K in the system by a *representative*, i.e., a set of new variables and constraints.

These two phases are repeated until no more cluster is found. The result of the method is a decomposition where each merge or single extension represents a component. Merge components are $\mathbb{D}$-well-constrained while components obtained by extension are $\mathbb{I}$-well-constrained. Indeed, merge steps identify a new subsystem to be solved in a new local reference system, while extensions position entities relatively to an existing local reference system. The partial order between components is induced by the aggregation of the cluster representatives (to be solved before) into a larger subsystem (to be solved later). The assembling operator amounts to concatenating all the components solved within the same local reference system (a merge step + all its extensions), and then assembling by displacements the clusters whose representatives are included in other clusters.

A significant advantage is that the solving phase can be interleaved with the identification phase : once a merge step or a single extension is performed, the corresponding component can be solved immediately.

Example

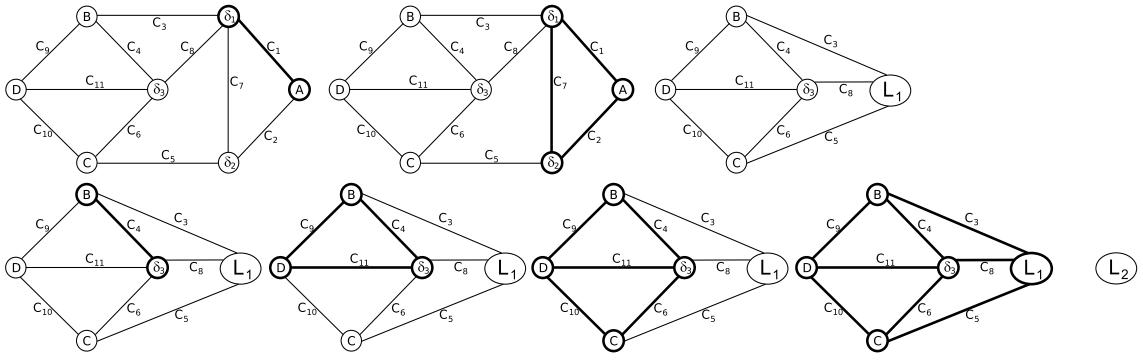


FIG. H.6 – Decomposition of Geo_2 by a recursive assembly method

Fig. H.6 shows how a recursive assembly can decompose Geo_2. A merge step finds $A\delta_1$ as a first rigid cluster. The cluster is extended to δ_2 with constraints C_2 and C_7. The cluster $A\delta_1\delta_2$ cannot be extended anymore (no neighbor entity can be added to the cluster keeping it rigid) so that

the identification phase ends. The contraction phase replaces this cluster by a single variable L_1 which represents the local reference system in which A, δ_1 and δ_2 will be determined. A second merge step identifies a rigid cluster $B\delta_3$. The cluster is extended to D, C, and finally to the cluster L_1, yielding the cluster L_2. This last extension amounts to positioning the local reference system of L_1 in the one of L_2.

Identification phase

Recursive assembly methods differ in the characterization of constriction (modulo $\mathbb{D}$ for merge step, modulo $\mathbb{I}$ for extensions steps) they employ (see Sec. H.2.4). Some use DOF-based characterizations and are called *structural recursive assembly methods*. They are described in Sec. H.4.2. Others use patterns or rules and are called *semantic recursive assembly methods*. They are described in Sec. H.4.3.

Contraction phase

The registration phase adds the components $K = \{S_1, ..., S_n\}$ returned by the identification phase (S_1 is the result of the merge step; S_i – $i \in \{2...n\}$ – corresponds to a single extension relative to S_{i-1}) into the decomposition and updates the partial order. If S_i includes the representative of a previously identified component S', then S_i depends on S', i.e., $S' \prec S_i$.

The replacement phase removes from S all the constraints and variables of K and replaces them by a *representative*. It is either a single variable related to the whole cluster K. Or it is a subset of the variables of $S_1...S_n$ shared by constraints in the rest of the system, like the virtual bonds in Owen's method.

H.4.2 Structural methods

Hoffmann et al. in Ref. [129] introduced a flow-based algorithm to check structural rigidity in polynomial time. The merge step identifies a minimal[5] *dense* (i.e., structurally well- or over-rigid) subsystem by computing a maximum flow in a network derived from the bipartite constraint graph. The source S is linked to each constraint, and each variable is linked to the sink T. The capacities correspond to the DOFs of the constraints (arcs from the source to constraints) and to the DOFs of the variables (arcs from variables to the sink).

A maximum flow in this network represents an optimal distribution of the constraints' DOFs onto the variables' DOFs. To identify rigid (or over-rigid) subsystems, the method adds an additional $\mathcal{D}$ capacity to one constraint at a time, thus fixing a local reference system onto the variables linked to the overloaded constraint.

Once a minimal cluster has been identified, the method performs an extension step which tries to include neighbor variables one by one.

[5]It has been proved in Ref. [188] that finding a minimum dense subsystem is an NP-hard problem. That is why Hoffmann et al. proposed to search minimal (with respect to set inclusion) dense subsystems only.

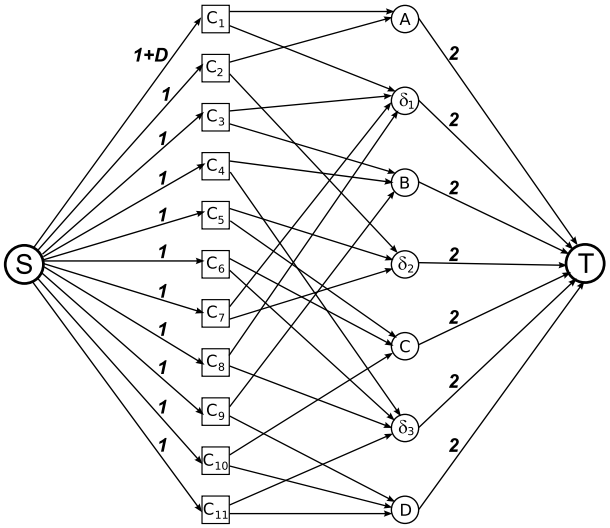


FIG. H.7 – Network representing Geo_2 ; the arc $S \rightarrow C_1$ is augmented by $\mathcal{D}$ to identify a rigid subsystem

This identification phase is used in two main methods by the same authors : the condensing algorithm (CA) and the frontier algorithm (FA).[130, 233] The two algorithms differ only in their contracting operator. CA contracts a cluster by removing all its variables and constraints from the constraint graph. Then it introduces a new vertex which represents the local reference system of the cluster. Every constraint which was incident to a variable inside the cluster is directly transferred to this new vertex. Fig. H.6 illustrates how CA decomposes Geo_2.

FA[6] contracts a cluster by removing all its *internal* variables, replacing them by a single vertex K. Variables are internal if they are not linked to any other variable outside the cluster. Otherwise, they are in the *frontier* of the cluster. The DOF of K is equal to the DOF of the subsystem it replaces. The constraints linking an internal variable i to a frontier variable f are transferred between K and f. The constraints linking frontier variables remain unchanged.

In these two variants, the partial order is induced by the integration of components' representatives into other components, yielding a *cluster tree*. When the representative of a cluster is solved in a subsequent component, a displacement is performed for assembling the components' solutions.

An extension of FA has been proposed for helping the user in case a system is over-constrained by only one equation. This algorithm provides the set of the constraints from which one can be removed to render the system well-constrained.[131] The method has been reproduced using a maximum matching algorithm, with a contraction step similar to CA.[308] Also, Gao *et al.* in Ref. [95] have obtained an algorithm similar to FA by implementing the merge step with a weighted maximum matching algorithm like the one designed by Latham *et al.* (see Sec. H.5).

[6]The FA algorithm was called MFA in the first descriptions.

Properties

Structural methods run in polynomial time, often quadratic since computing a maximum flow (or matching) can be done in $O(n^2)$. They are complete and correct with respect to the structural rigidity, but not with respect to the geometric rigidity (see Sec. H.2.4). They are not confluent since the decomposition depends on the identification order of the clusters.

H.4.3 Semantic methods

We distinguish semantic methods according to the way they handle the identification phase : either using patterns to be identified in the system, or using rules to be triggered on a base of facts.

Pattern-based methods

Sunde (Ref. [277]) introduced a method to deal with 2D problems composed of points linked by distances and angles. Initially, each segment (point-point distance constraint, CD in short) and each pair of segments constrained in angle (CA in short) is considered a rigid subsystem. The method uses a few patterns to aggregate subsystems into bigger ones. For instance, two CAs sharing a segment, or three CDs pairwise sharing a point, can be aggregated. The method succeeds when it returns a single CD. Each pattern use corresponds to a component of the decomposition. The partial order is derived from the application order of the rules. Verroust *et al.* in Ref. [296] introduced additional patterns and proved that there exist reducible constraint systems that cannot be decomposed with this method. Joan-Arinyo *et al.* in Ref. [160] have also extended the method by considering point-lines constraints (CHs).

Fudos *et al.* in Refs. [41, 88] generalized the approach by abstracting constraints and entities by their DOFs. In this method, each constraint is also initially considered a rigid subsystem, called a *cluster*. The main rule used by this method is *"3 clusters pairwise sharing a variable can be aggregated"*. This rule also appears in Sunde's method. The main difference is that CDs and CAs are not distinguished. We obtain a gain in generality but a loss in correctness. For instance, this method could assemble 3 lines linked by 3 angle constraints, which does not correspond to a $\mathbb{D}_2$-well-constrained subsystem. Applied to $\mathbb{D}_2$-well-constrained systems, this method was proved in Ref. [162] to have the same capabilities as Owen's method (see Sec. H.3). It was also combined with an equational decomposition method (see Sec. H.5) in order to handle mixed algebraic-geometric constraint systems.[127, 160]

Kramer in Ref. [176] proposed a similar approach to solve kinematics problems in 2D and 3D. It uses an extensive repertoire of construction steps that position one mechanical piece relatively to already known ones through both a DOF case-based reasoning (where translational DOFs from rotational ones are distinguished) and applications of the Grübler formula,[117] a property used in the theory of mechanisms and close to the structural rigidity. This method, like Fudos *et al.*'s one, mixes a structural approach and a semantic one.

Gao et al. have proposed a method using loci determination which is oriented towards 2D

problems, and presented as a generalization of Sunde's approach.[93]

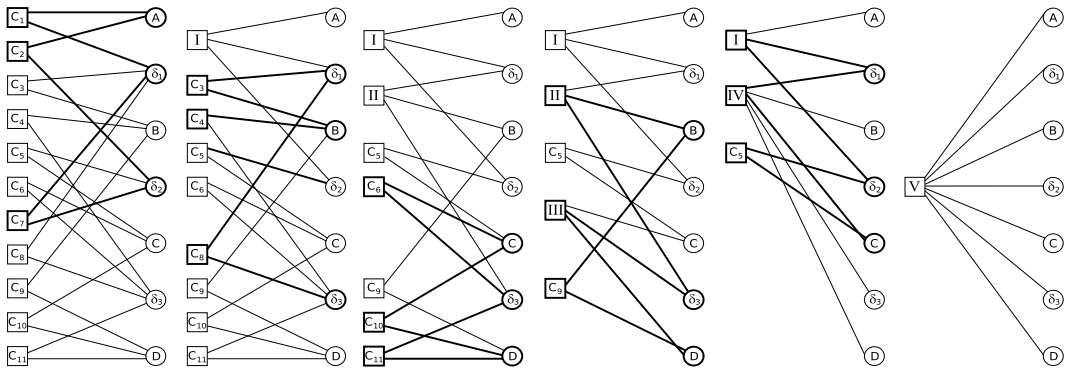


FIG. H.8 – Decomposition of Geo_2 computed by Sunde-like methods

Example : Figure H.8 illustrates the use of Sunde's method on Geo_2. The different graphs represent the iterations of the algorithm. Initially, each constraint is considered to induce a rigid subsystem. Then, a rule identifies 3 subsystems (1, 2 and 7, in bold in the first graph) pairwise sharing a variable (A, δ_1 and δ_2). The contraction results in the second step of the figure : the three subsystems are aggregated into a single one numbered I. The following iterations use the same pattern until all the system is aggregated into a single component. Note that once a subsystem has been identified, it becomes part of a subsequent aggregation. The partial order comes from the aggregation process : $1 \prec I$, $2 \prec I$, $7 \prec I$, $3 \prec II$, $4 \prec II$, ..., $I \prec V$, $IV \prec V$ and $5 \prec V$.

Properties : Pattern-based methods are polynomial time in the size s of the biggest pattern, since a pattern has to be compared at worst with all the subsystems of size s. They are complete with respect to their repertoire of patterns (see Sec. H.2.4). They are in general correct when the patterns are not DOF-based and can thus identify singular cases. Patterns can be applied in different orders, making these methods not confluent.

Knowledge-based methods

Knowledge-based methods derive from the artificial intelligence community. Seminal works were performed in the educational domain,[256, 58] and then extended to CAD.[4] The key idea consists in formalizing the geometric reasoning by a multi-typed language and a set of axioms.[257] This formalization is implemented by a rule-based solver with a forward-chaining inference system. These methods often lead to robust solvers able to give explanations in case of failure.

Mathis et al. in Ref. [73] introduced a knowledge-based method, taking $\mathbb{D}_2$-invariance into account, composed of :

– *Agents* which are mainly rule-based solvers devoted to subsystems discovery and solving. Each agent execution produces a component of the decomposition.
– A mechanism that injects a metric information (constraint) deduced from the solved subsystems into the remaining system (replacement step).

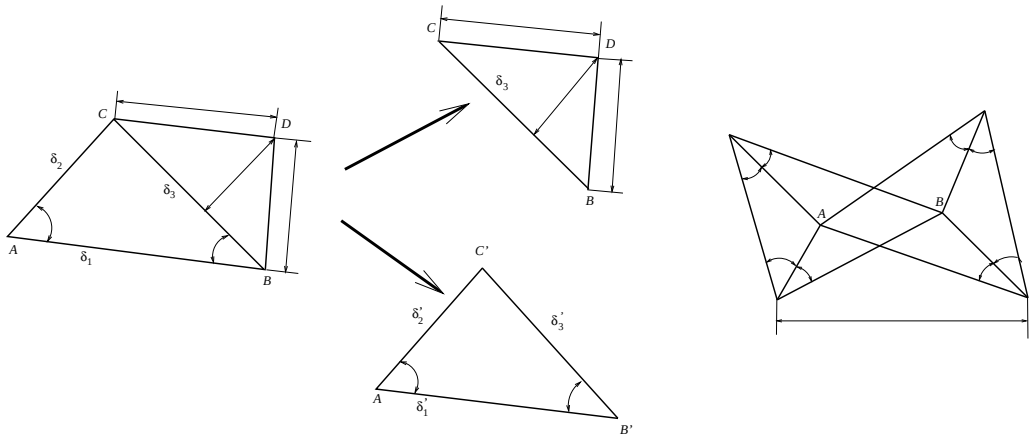


FIG. H.9 – **Left :** Decomposition of Geo_2 using multigroup invariance. **Right :** A constraint system decomposable by multigroup approaches only.

This framework has been extended to take other invariance groups into account, like the similarity group,[255] and even several invariances simultaneously.[258] For instance, Geo_2 could be decomposed into a similarity invariant subsystem S_1 and a rigid (displacement invariant) subsystem S_2 as illustrated in Fig. H.9–left. S_1 is solved modulo the similarities, i.e., it respects the angle constraints but can still be scaled and displaced. S_2 is solved modulo the displacements, i.e., in a local reference system. The assembling operator computes the similarity (displacement + scaling) to be applied to the solutions of S_1 such that points B and C coincide in S_1 and S_2. Using the multigroup invariance allows us to decompose systems that cannot be decomposed by traditional methods, as shown in Fig. H.9–right.

Properties : Knowledge-based methods are very similar to pattern-based ones, except they adopt an axiomatic approach for the decomposition phase. They run in polynomial time in general, with an exponent in the size of the biggest rule. However, some of these methods create intermediary variables, possibly leading to infinite inference chains. Knowledge-based methods are complete with respect to their repertoire of rules (see Sec. H.2.4). Like pattern-based methods, knowledge-based ones are generally correct but not confluent. Finally, existing knowledge-based methods are limited to 2D systems.

H.5 Single-pass Approaches

Single-pass methods compute a decomposition in a single iteration : all the components are produced simultaneously. They work at the equational level and are thus not restricted to geometric constraints. Applying a *maximum matching* to the equation graph, they identify structurally $\mathbb{I}$-well-constrained components (see Sec. H.2.4). In case the whole system is structurally $\mathbb{I}$-well-constrained, the decomposition into irreducible components is unique. Otherwise one obtains a coarse canonical decomposition into three subparts : an over-, a well- and an under-constrained subparts.

H.5.1 Principle

Serrano proposed the first matching-based solver for systems of nonlinear equations appearing in general design problems.[265] Since then, the method has been generalized and follows these steps :

1. Computation of a maximum matching of the equation graph.
2. Computation of a Dulmage & Mendelsohn (D&M) *coarse* decomposition which divides the equation graph into three canonical parts : $\mathbb{I}$-well-constrained, $\mathbb{I}$-over-constrained and $\mathbb{I}$-under-constrained.
3. The $\mathbb{I}$-over- and $\mathbb{I}$-under-constrained parts are either ignored or transferred into the $\mathbb{I}$-well-constrained one by a specific treatment.
4. Computation of a *fine* decomposition of the $\mathbb{I}$-well-constrained part. The components and the partial order between them are then determined.

Computation of a maximum matching

The decomposition is based on a maximum matching of the equation graph.

Definition 10 *A* **matching** *of a graph is a subset of its edges, such that no two edges in the matching share a vertex. A vertex is* saturated *if it belongs to an edge of the matching. A* perfect *matching saturates all the vertices. A matching is* maximum *iff no other matching of the same graph contains more edges.*

Intuitively, an edge (c, v) in the matching means that the equation c computes the variable v. Well-known polynomial algorithms compute a maximum matching in a bipartite graph.[132] The obtained matching yields an implicit orientation to the equation graph : each edge in the matching is oriented in both directions, while edges outside the matching are oriented from the variables to the equations. An example is shown on Fig. H.10[7].

[7]The point D and the direction a_3 of the line δ_3 are fixed by the unary constraints f_1, f_2, f_3 to render the whole system $\mathbb{I}$-well-constrained.

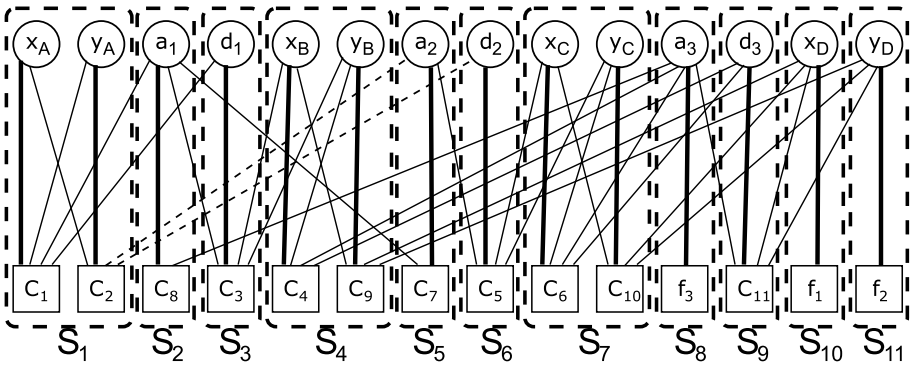


FIG. H.10 – D&M and scc decompositions of Geo_2. Edges in the matching are bold-faced. The components are represented by dashed hyper-edges.

A decomposition of a maximum matching into strongly connected components (i.e., cycles) is obtained by a linear depth first search algorithm by Tarjan.[279]

Definition 11 *A directed graph is* **strongly connected** *iff for any two vertices x and y in G, there exists a directed path from x to y and from y to x. The strongly connected components (scc) of G are its maximal strongly connected subgraphs.*

Arcs *between* the sccs yield the partial order of the decomposition. In our example, this yields : $S_{11} \prec S_9$, $S_{11} \prec S_7$, $S_{11} \prec S_4$, ..., $S_2 \prec S_5$, $S_2 \prec S_3$, $S_2 \prec S_1$.

Before a subsystem is solved, the variables outside the corresponding scc are replaced by the values that have been computed in previous subsystems. For example, for solving the last subsystem S_1, the variables a_1, d_1, a_2 and d_2 must be replaced by their values computed in subsystems S_2, S_3, S_5 and S_6. The geometric interpretation of this computation is that the point A is placed at the intersection of lines δ_1 and δ_2 which must be determined first.

When the whole system lies in the well-constrained part, i.e., it admits a perfect matching, it can be decomposed into sccs as described above. The following theorem ensures that this decomposition is independent from the matching used.[174]

Theorem 2 *(König, 1916) Every perfect matching of a bipartite graph leads to a unique decomposition into strongly connected components (i.e., irreducible $\mathbb{I}$-well-constrained subsystems).*

Dulmage and Mendelsohn (D&M) decomposition

If the obtained maximum matching is not perfect, some equations and/or some variables are not saturated. Unsaturated variables are not determined, i.e., they induce a structurally under-constrained subsystem. Unsaturated equations are not taken into account and induce a structurally over-constrained subsystem. Dulmage and Mendelsohn have proposed a decomposition

approach which applies to large sparse systems of linear or non linear equations.[237] Figure H.11 shows an example.

Theorem 3 *(Dulmage and Mendelsohn, 1958) Let G be an equation graph. Any maximum matching of G gives a canonic partition of the vertices in G into three disjoint subsets : the under-constrained part U_G, the over-constrained part O_G and the well-constrained part W_G.*

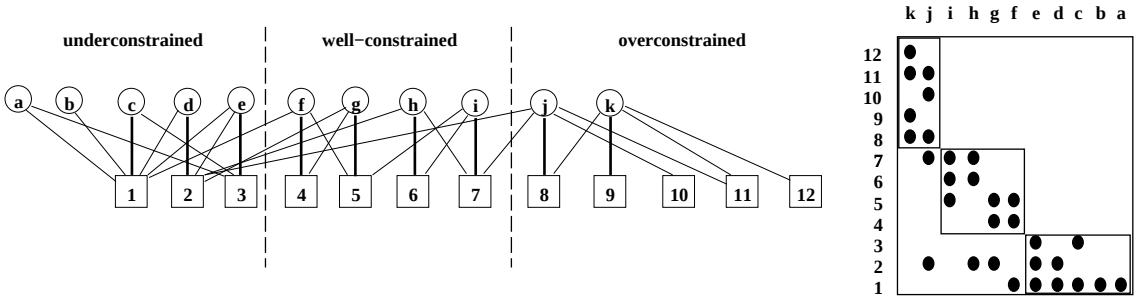


FIG. H.11 – **Left :** The three parts found by D&M's decomposition. The equation graph contains equations 1...12 and variables $a...k$. **Right :** Equivalent matrix representation (the square in the middle corresponds to the well-constrained part).

The existence of non-empty subparts O_G or U_G in a constraint system generally represents an error which can be returned to the user or automatically repaired.

H.5.2 Properties

The D&M and scc decompositions can be computed in polynomial time, basically quadratic : computing a maximum matching is done in quadratic time (assuming the number of equations is close to the number of variables, and the arity of equations is bounded by a constant). Identifying the three parts and the sccs requires only graph walks performed in linear time.

The method is complete and correct with respect to the structural $\mathbb{I}$-constriction. D&M's decomposition and the fine decomposition of the well-constrained part into sccs are confluent. However, the handling of the over- and under- constrained parts may induce some changes in the resulting decomposition.

H.5.3 Difficulties due to the application to geometry

Geometric constraint systems are often $\mathbb{D}$-well-constrained. Thus, corresponding equation systems are generally $\mathbb{I}$-under-constrained, and the whole equation graph falls into the under-constrained part. Because König's canonicity result does not hold anymore in this case, the scc decomposition then depends on the computed maximum matching, which leads to two major flaws :

– Some matchings lead to smaller subsystems than others. Fig. H.13–right will show a non optimal decomposition of Geo_2.
– Some matchings lead to geometrically incorrect decompositions. For instance, consider that Geo_2 is rendered well-constrained by fixing x_B, x_C and x_D. This results in a single scc (no decomposition) which is structurally $\mathbb{I}$-well-constrained while it is $\mathbb{I}$-over-constrained in reality : the prescribed distances BD and CD cannot be satisfied for every value of their abscissa.

To decompose rigid geometric constraint systems in a correct manner, Hendrickson proposed to fix the geometric system in a coordinate system, removing thus the 3 degrees in freedom in excess in 2D (6 in 3D).[124] When the geometric system is a 2D bar framework, every edge (representing a point-point distance in the constraint graph) is pinned in the plane by adding 3 similar edges, and a maximum matching is computed on the modified system. The system is rigid iff all the combinations yield a perfect matching. This method is performed incrementally, and only the first maximum matching has to be computed from scratch. The others are obtained in linear time by updating the previous one.

Latham and Middleditch proposed to replace the maximum matching computation by a weighted maximum matching one.[181] The constraint graph is weighted by the degrees of freedom of the variables and the constraints. The advantage is that it is performed directly on the constraint graph of the geometric system, avoiding the need to translate it into an equation system. This method was also able to add (resp. remove) constraints in an ad-hoc manner in case the system is under-rigid (resp. over-rigid). This algorithm has been used in several alternatives to the structural recursive assembly method due to Hoffmann *et al.* (see Sec. H.4.2).[308, 95]

H.6 Propagation of Degrees of Freedom Approaches (PDOF)

Initially implemented by Sutherland in his graphical tool Sketchpad,[278] PDOF[8] has been first used in local propagation solvers for handling interactive constraint systems, such as graphical layout systems or user interfaces.[39, 295] As we will see, pattern-based PDOF algorithms overcome the main drawbacks of the maximum matching approach when tackling structurally $\mathbb{I}$-under-constrained systems.

H.6.1 Description of a generic PDOF

The PDOF algorithm has first been applied at the equational level, that is, it works on an equation graph.[278, 39, 295] Several approaches in geometry directly work with the (geometric) constraint graph.[2, 138, 195, 78] In both cases, the principle of PDOF is to iterate the following steps :

1. select a **free** $\mathbb{I}$-well-constrained subsystem S' from the equation/constraint graph G,
2. remove from G vertices and edges corresponding to S'.

[8] PDOF stands for Propagation of Degrees of Freedom.

The algorithm stops when the graph is empty or when it contains no more free well-constrained subsystem. A subsystem is *free* if its variables appear only in the equations of the subsystem. Thus, solving a free subsystem cannot violate equations not in S', i.e., no future component will depend on S'. Each free subsystem identified by PDOF is a component of the decomposition. The partial order is obtained by considering variable dependences : a component S_j depends on a component S_i if S_i computes a variable which is involved in an equation of S_j.[9] The assembling operator amounts to concatenating the components solutions.

In case the constraint system is under-constrained, when all the equations have been removed from the equation graph, a set of variables (called *input parameters*) remains. Hence, PDOF is also a procedure that determines a set of input parameters to be fixed for rendering $\mathbb{I}$-well-constrained the system.

We illustrate how PDOF computes a decomposition and then we provide details about the existing algorithms that are based on the generic algorithm above.

H.6.2 Example

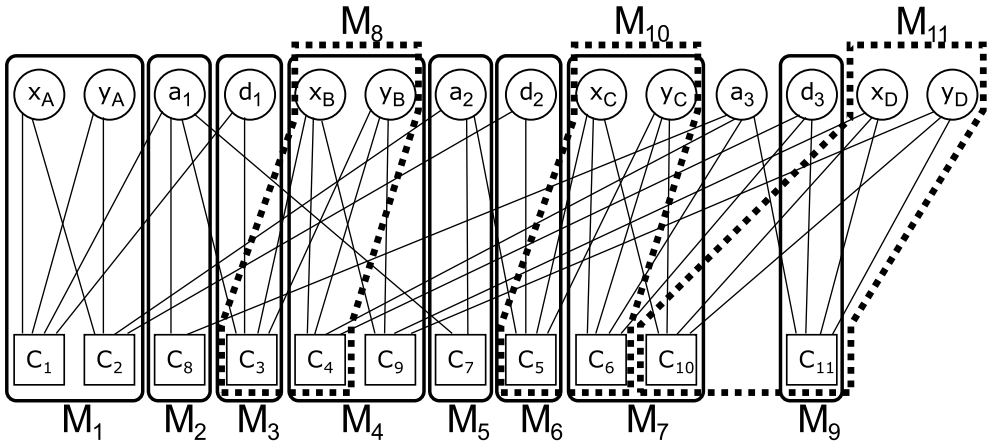


FIG. H.12 – Equation graph of Geo_2 with subsystems $M_1 ... M_{11}$ represented by hyper-edges corresponding to patterns in a dictionary (for the sake of clarity, all the subsystems have not been drawn). Several subsystems, namely the hyper-edges in plain lines, are selected by PDOF to decompose the system.

Fig. H.12 shows how PDOF decomposes Geo_2 (among other alternatives). It selects free subsystems in the order M_1 (computing point A at the intersection of δ_1 and δ_2), M_3, M_6, M_4, M_5, M_2, M_7, M_9. It selects first M_1 because M_1 is free, that is, variables x_A and y_A are connected only to C_1 and C_2 which belong to M_1. Once M_1 is selected and removed from the equation graph, M_3 and M_6 become free and can be selected next, and so on. At the end, the remaining

[9]Note that a total order can be trivially obtained by reversing the identification order : the first identified subsystem is solved last.

variables a_3, x_D and y_D constitute a set of input parameters. Selected subsystems are solved in reverse order (i.e., from M_9 to M_1).

H.6.3 Algorithms based on the generic PDOF

The basic `PDOF` algorithm, derived from local propagation, iteratively selects a free subsystem of size 1. This gives a triangular form to the equation-variable dependence matrix. However, it is often not possible to triangulate a geometric constraint system with only subsystems of size 1.

`OpenPlan` : a purely structural PDOF

Like Single-pass methods, `OpenPlan` works at the equational level with no geometric information.[34] It tries to select a free structurally $\mathbb{I}$-well-constrained subsystem *of minimum size* at each step. `OpenPlan` returns the best (in terms of size of the largest component) decomposition that can be obtained by a maximum matching applied to an under-constrained system. However, the problem of finding a free subsystem of smallest size in a structurally $\mathbb{I}$-under-constraint system is NP-hard. Indeed, it is the dual of the NP-hard *minimum dense* problem (searching for a structurally rigid subsystem of minimum size). [188] That is why a heuristic version of `OpenPlan` also uses a maximum matching of the equation graph to find a small, but not necessarily the smallest, free subsystem.[34]

Pattern-based PDOFs

As opposed to the standard PDOF, pattern-based PDOFs can identify free $\mathbb{I}$-well-constrained subsystems of bigger size. Pattern-based PDOFs make a bridge between the equational and the geometric levels by using a dictionary of subsystem patterns. These patterns correspond to geometric construction steps. In the example above, the pattern *"point to be placed at the intersection of two known lines"* in the dictionary corresponds to several subsystems in the actual system, such as M_1 (point A intersection of δ_1, δ_2), M_8 (point B) and M_{10} (point C).

Basic pattern-based PDOFs apply the generic scheme presented above.[195, 78, 2] They iteratively select free subsystems (of arbitrary size) present in the dictionary. However, it appears that these algorithms may be unable to compute a sequence of subsystems corresponding to patterns in the dictionary, even if one such sequence exists. A simple example can be found in Figure 6 of Ref. [282]. To overcome this drawback, one needs to be able to select subsystems that "overlap", that is, share equations and variables. This analysis has led to the design of *General PDOF* (`GPDOF`) that can be viewed as a robust implementation of a pattern-based PDOF.

`GPDOF` is able to compute a sequence of subsystems corresponding to patterns in the dictionary, if one such sequence exists. To be able to select subsystems that overlap, `GPDOF` requires that the equation graph be enriched in advance with hyper-edges corresponding to patterns. For instance, this preliminary enrichment phase produces the equation graph depicted in Figure H.12 enriched with hyper-edges $M_1 ... M_{11}$. See Refs.[306, 288] for more details on this algorithm.

H.6.4 Properties

Like Single-pass, `OpenPlan` is complete and correct, with the usual limitations of structural methods due to an incorrect constriction approximation related to redundancies or singularities. Provided that the system contains no redundant equations, `GPDOF` can always compute a sequence of subsystems present in the dictionary if one such sequence exists, but a strong limitation is of course the non exhausitivity of the dictionary.

Pattern-based PDOF methods run in polynomial time. The standard PDOF algorithm runs in $O(n \times dv \times dc^2)$ while `GPDOF` runs in $O(n \times dc \times dv \times r \times (g \times dc + g^2))$, where n is the number of equations, r is the maximum number of hyper-edges per equation, dc and dv are the maximum degrees of respectively an equation and a variable in the equation graph, and g is the maximum number of equations and variables involved in a hyper-edge. Note that r is $O(n^g)$, rendering the method practicable for small patterns only. In an application of `GPDOF` to constraint-based 3D scene reconstruction, large under-constrained geometric systems are decomposed in a few seconds and solved in hundredths of seconds.[306, 288]

H.7 Comparison Between the Four Decomposition Schemes

In this section, we compare the four decomposition schemes we have presented. These approaches differ not only in their algorithmic aspects, but also in the abstraction they use. Taking these differences into account allows us to better explain their relative strengths and weaknesses.

H.7.1 Recursive division versus recursive assembly

Recursive division methods use k-connectivity as an approximation of $\mathbb{D}_2$-constriction, which renders them incorrect and incomplete in general (see Sec. H.2.4). Consider the 2D constraint system composed of 12 points ($A...L$) linked by 21 distances whose constraint graph is depicted in Fig. H.13–left. This graph is triconnected so that Owen's method cannot decompose it. However, Hoffmann *et al.*'s recursive assembly method can decompose it further : all the small triangles are aggregated recursively and finally the big triangles ACF and GJL are aggregated using the three distances AJ, FG and CL.

This poor characterization of rigidity explains why recursive assembly is generally considered more powerful than recursive division. This difference is thus not really related to their respective schemes (bottom-up or top-down).

Concerning the schemes, recursive division methods can often decompose under-constrained systems because the more under-constrained the problem is, the more articulation pairs can be found. On the contrary, because merge operators intrinsically identify well-constrained or over-constrained components, recursive assembly methods are best suited for over-constrained systems.

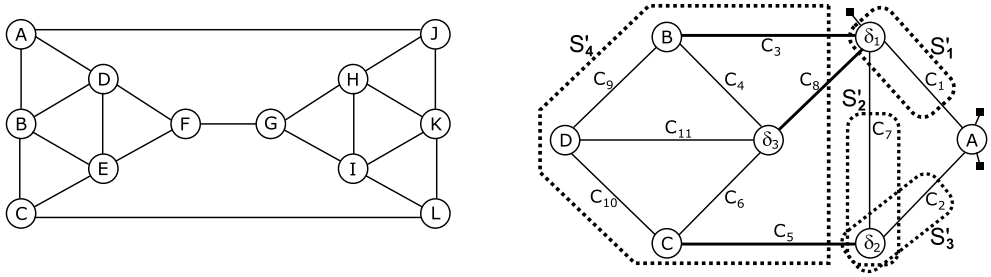


FIG. H.13 – **Left :** A decomposable problem that Owen's method cannot treat. **Right :** Coarse decomposition (in dashed lines) of Geo_2 by Maximum-matching.

H.7.2 Single-pass versus PDOF

Dulmage&Mendelsohn's decomposition of single-pass methods is always interesting as a preliminary step to decompose a geometric constraint system : it identifies a structurally $\mathbb{I}$-over-constrained part O, an $\mathbb{I}$-well-constrained part W and an $\mathbb{I}$-under-constrained part U of the system.

Part O cannot be solved : it requires some specific treatment (e.g., manual debugging or automatic relaxation of constraints). Part W admits a unique decomposition (see Sec. H.5.1). Part U requires a further study : if obtained by any maximum matching, the resulting decomposition often contains arbitrarily large subsystems (see Fig. H.13–right). On the opposite, a pattern-based PDOF provides only meaningful components of bounded size. In addition, the more under-constrained the system, the easier PDOF will find a free subsystem.

The weakness of a pattern-based PDOF is its incompleteness due to the limited number of patterns registered in its dictionary. Moreover, redundancies often block the decomposition process because redundant equations may prevent `PDOF` from finding a free subsystem (due to these constraints in excess). Redundant equations must therefore be detected and removed before PDOF is launched.[288]

H.7.3 Equational versus geometric approaches

On one hand, equational methods (single-pass and PDOF approaches) can deal with constraint systems combining geometrical and non geometrical entities, providing a greater generality. In addition, they sometimes better decompose a system because they work with a finer grain : a single geometric entity is represented by several variables that may be solved in different components of the decomposition. For instance, in Fig H.12, page 220, the variables a_1 and d_1 modeling line δ_1 appear in two components M_2 and M_3.

On the other hand, geometric decompositions (recursive division and recursive assembly approaches) generally use constriction modulo the displacements and exploit an assembling operator based on displacements. This allows them to use several times the 3 DOFs in 2D (resp. 6 in

3D) of a rigid geometric system, once for each identified rigid component. In comparison, equational methods can only aggregate $\mathbb{I}$-well-constrained components and hence can only use once these DOFs. That is why geometric decomposition methods often yield finer decompositions.

Combining the advantages of both methods is possible. Each component obtained by a geometric decomposition can sometimes be further decomposed by an equational technique that reduces the (geometric) components to several (equational) sub-components. This principle has been followed for obtaining the decomposed systems studied by Jermann *et al.*[155]

Finally, when facing $\mathbb{D}$-under-constrained systems, equational and geometric decomposition methods *repair* them differently. The former fix variables, while the latter add geometric constraints (e.g., distance constraints). The reason is that equational methods achieve $\mathbb{I}$-constriction while geometric ones achieve $\mathbb{D}$-constriction.

H.7.4 Semantic versus structural methods

Semantic methods (i.e., rule-based recursive assembly and pattern-based PDOF) cannot define all the patterns necessary to tackle every geometric constraint system (see Sec. H.2.4)[10]. On the other hand, the main advantage of rule-based methods lies in their proximity to geometry. They generally use geometrically sound operations that are protected against failure cases. For instance, a guard could detect that a point cannot be computed at the intersection of two lines because they are parallel. Also, a fast solving method is often known to solve the subsystems.

In comparison, structural methods (graph-based recursive division/assembly, and single-pass) are in general complete but ensure correctness only with respect to an approximation of constriction defined by a structural property (e.g., triconnection or a DOF count).

Structural and semantic methods could complement each other, ensuring the completeness of the approach while achieving a higher reliability. Two such hybrid algorithms are based on the same principle : a structural decomposition is updated while a pattern-based decomposition is performed. Every time the decomposition process is stuck because no existing pattern is available, a subsystem is picked in the structural decomposition. In a Maximum-Matching/PDOF hybrid algorithm introduced in Ref. [282], a *leaf* (in the partial order) of a maximum matching decomposition is free and is selected as the next solvable component. In a Maximum-Matching/bottom-up hybrid algorithm introduced in Ref. [196], a *root* (in the partial order) of a maximum matching decomposition is selected.

H.8 Towards Real-life Requirements

In this section, we establish a list of properties that are desirable to meet real-life applicative requirements : *generality* and *reliability*. Indeed, a decomposition method should be as *general* as possible in order to tackle the broadest class of systems. It should also be *reliable* in the sense

[10]However, for some particular classes of applications, for instance mechanical assemblies, such methods can be made almost complete.[176]

it should adapt to the expectations of the user and be able to overcome situations that often make the decomposition fail (e.g., singular or dependent constraint systems).

H.8.1 Generality

Methods able to handle 2D and 3D problems can address complex CAD applications where 2D constrained sketches are used in complement to 3D constrained models. However, it is certainly unnecessary to use a complex 3D method for bar-frameworks in 2D. Also, ruler & compass techniques suffice for some applications and offer a higher reliability than more general methods. Hence, determining the required level of generality for an application is important to select the appropriate method.

Geometric decomposition methods should be able to use several invariance groups to decompose systems as finely as possible. One of the strengths of recursive assembly method comes from its combination of $\mathbb{D}$-invariance (merge step) with $\mathbb{I}$-invariance (extension step). Extension is a cheap augmentation technique resulting in small $\mathbb{I}$-well-constrained components, while a more expensive merge operation ensures the completeness of the method.

A general decomposition method should be able to handle under-constrained and over-constrained problems, at least by identifying as precisely as possible the parts responsible for the non well-constriction, and ideally offering some automatic repairing tools. Note that certain decomposition schemes are more suited to handle these cases than others. Recursive division and PDOF are poor against over-constrained systems but stronger against under-rigid ones. Conversely, recursive assembly can handle over-constrained parts but requires some adaptations to deal with under-constrained systems. Finally, single-pass methods are able to characterize the well-, under- and over-constrained parts, but the non well-constrained ones require a specific treatment.

The decomposition returned by a method could be best employed by a user for debugging purpose (e.g., identification of over-/under-constrained components) if it corresponds to the view the user has of its system. For this reason, it is generally desirable to respect a coarse decomposition induced by a high-level user's manipulation of entities (e.g., mechanical pieces). Sitharam et al. proposed to adapt graph-based recursive assembly methods to this requirement.[269]

The decomposed system should return the solution expected by the user, e.g., the solution minimizing some criterion or closest to a given sketch. This problem is referred to as *root identification* by Fudos and Hoffmann.[89] Geometric constraint solvers use heuristics to select the desired solution, or use a combinatorial process where one branch in the search tree corresponds to one (sub)solution in a component.[79, 155]

Finally, it happens in several applications like robotics, CAD or structural biology that the constraint system contains not only geometric entities and constraints but also algebraic variables and equations, representing for instance physical laws, costs or energy. It is hence important to be able to handle these mixed algebraic-geometric systems.

Among the methods classified in the proposed four categories, none satisfies all these requirements simultaneously. We expect that hybrid variants between some of these algorithms should result in an increasing generality.

H.8.2 Reliability

Usual algorithmic properties (i.e., correctness, completeness, confluence and complexity) provide a first measure of the reliability of a decomposition method. The ability of a method to return the decomposition into the smallest components is also important. However, this requirement should be balanced with considerations about performance. Indeed, achieving the finest decomposition has been proved to be NP-Hard in several cases.[188]

Decomposition methods should be aware of dependences between constraints. A subset of constraints is dependent if it is either redundant or contradictory, i.e., if it contains the hypothesis and the conclusion, or the negation of the conclusion, of a geometric theorem. Some dependences can be detected structurally, e.g., 4 points linked by 6 distances in 2D. However, many dependences are not structural but are related to the geometric nature of the entities and constraints, e.g., the double-banana 3D system (see Fig. H.4). Handling dependences requires expensive automated theorem proving. This justifies the use of heuristics to handle the more common cases.

Decomposition methods should take into account the singularities. Indeed, many methods work under a genericity hypothesis and decompose systems into generically solvable components. However, it may be the case that a solution of a decomposed system lies into a singular variety, e.g., includes some unspecified collinearity or coplanarity. In this case, it happens that the generically solvable components are no more solvable. For instance, the double-banana system (see Fig. H.4) is generically over-constrained but becomes under-constrained if the height of both bananas is the same. Dealing with singularities requires time-consuming algebraic computations and is generally incomplete. Also, singularities sometimes introduce dependences or transform contradictory ones into redundant ones.

Among the decomposition schemes we have listed, none appears to perfectly deal with these reliability issues. They are generally incomparable with respect to the size of the decomposition they obtain. Structural methods can detect only structural dependences, cannot distinguish between redundant and contradictory cases and does not take any singularity into account. Semantic methods can include rules and guards that check certain non-structural dependences and singularities.

However, no *a priori* decomposition method can be made completely reliable since there exist systems that have both singular and non singular solutions (see Fig. H.14), so that certain error detections could only be performed *during* the solving phase. Such systems would be best handled by methods interleaving decomposition and solving phases (e.g., recursive assembly).

Although not able to cope with this last problem, the WCM presented in the next section makes a sensible step towards generality and reliability.

H.9 The Witness Configuration Method

We have seen that one cause of failure of decomposition methods is that they are unable to accurately characterize the rigidity property, i.e., they can identify components that are not solvable because they are either over-constrained (no solution) or under-constrained (infinitely many solutions).

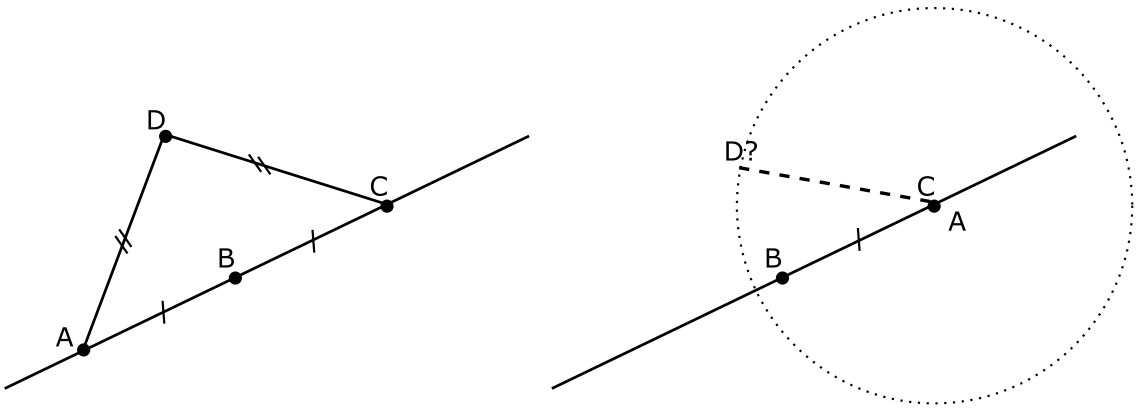


FIG. H.14 – A constraint system which is both well-constrained and under-constrained. An equality distance is defined between AB and BC, and also between AD and CD ; Distance constraints AB and AD are also defined ; A, B and C are incident to the line. One solution (left) for the position of A and C is generic while the other (right) is singular and yields an infinity of solutions for point D (it introduces a redundancy).

To conclude this survey, we present a recent development in geometric constraint processing that provides a more general and reliable characterization of solvability : the witness configuration method (WCM).[84, 207] We show that this principle can be integrated in existing decomposition methods or be the base of new ones.

H.9.1 Principle of the WCM

To determine whether a geometric constraint system $S = (C, X, A)$ is rigid, the WCM combines the following techniques :

1. a generalization of the algebraic rigidity check for bar frameworks by infinitesimal motions computation,[302, 115]
2. the Numerical Probabilistic Method (NPM) that numerically checks algebraic properties using a probabilistic argument and random configurations,[180]
3. a new method that generates *witness configurations* instead of random ones.[84, 207]

The probabilistic rigidity check

The principle of the algebraic infinitesimal rigidity[11] check is to compute the infinitesimal motions allowed by the constraints of a geometric system S. A *basis* of these infinitesimal motions is defined by the *kernel* K of the Jacobian matrix J of the equation system representing S. If

[11]Infinitesimal rigidity is a first-order version of rigidity. It is a stronger property than rigidity, i.e., every infinitesimally rigid system is also rigid ; the converse is not true.[114]

K reduces to a basis D of the infinitesimal displacements, then S is rigid ; otherwise it is not rigid : if D contains one vector independent from K, S is over-rigid ; else (i.e., K contains a vector independent from D) S is under-rigid.[302, 115] This principle was introduced for bar frameworks but can be generalized to any geometric entities in any dimension.

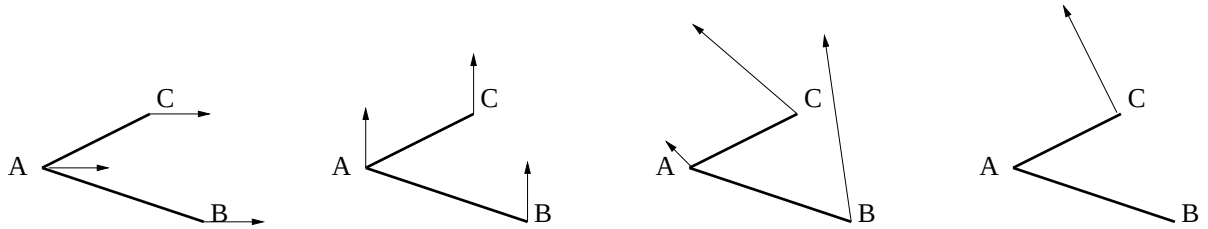


FIG. H.15 – An under-rigid 2D geometric constraint system

Let us illustrate the principle using the simple example S depicted in Fig. H.15, composed of 3 points linked by 2 distances. In this system, assuming the variable vector is $(x_A, y_A, x_B, y_B, x_C, y_C)$, a basis K of the infinitesimal motions of S is (illustrated from left to right in Fig. H.15) : $\dot{X}_1 = (1, 0, 1, 0, 1, 0)$ is the infinitesimal translation in x ; $\dot{X}_2 = (0, 1, 0, 1, 0, 1)$ is the infinitesimal translation in y ; $\dot{X}_3 = (-y_A, x_A, -y_B, x_B, -y_C, x_C)$ is the infinitesimal rotation around the origin ; $\dot{X}_4 = (0, 0, 0, 0, -y_C, x_C)$ is a last independent infinitesimal motion. $(\dot{X}_1, \dot{X}_2, \dot{X}_3)$ is also a basis D for the infinitesimal displacements allowed in 2D. Since K does not reduce to D, S is not rigid. However, K contains D and thus S is not over-rigid. Finally, since $\dot{X}_4$ is independent from D, it is a flexion and S is under-rigid.

To avoid resorting to exponential computer algebra methods for computing the infinitesimal motions, the NPM method can be used.[124, 180] In this case, the Jacobian matrix J of $S = (C, X, A)$ is evaluated at a randomly chosen configuration $(X, A) = (\theta_X, \theta_A)$. The kernel of $J(\theta_X, \theta_A)$ can then be computed using a simple Gaussian elimination in polynomial time. The result of this numerical computation on a sample configuration extends to the geometric constraint system using a probabilistic argument : if the point (θ_X, θ_A) is picked at random in a dense field, then the algebraic properties at this point are generically those of the algebraic system.[194]

However, this method (both algebraic and numerical flavor) applies only under a genericity assumption, e.g., null distances and null angles are not permitted. Indeed, it does not take into account the values of the parameters in A : computer algebra neglects the right part of the equation system, and the NPM uses a random (generic) assignment for the parameters. This strong assumption makes it difficult to use the method in practice since collinearities, coplanarities and other *singular* constraints are frequent in several applications (e.g., CAD).

The witness configuration

To overcome this limitation, Michelucci et al. proposed to use the NPM with a *witness configuration* instead of a random one.[207] A witness configuration of a system $S = (C, X, A)$

is a configuration $(X, A) = (\theta_X^w, \theta_A^w)$ that satisfies all the singular constraints in S, such as point/line/plane incidences. Indeed, these constraints imply that the solutions of S all lie in singular components of the configuration space, i.e., they may not have the generic properties of S. Hence, if the infinitesimal motions of S are computed using the NPM at a witness configuration instead of a random one, then the inferred constriction will hold for its solutions.

Assuming that all the singular constraints are explicitly stated by the user, a witness configuration can be computed as a solution of a reduced system S' which includes only the singular constraints of S. In theory, this problem is as complicated as solving the original system S. However, in practice, the reduced system S' is highly under-constrained and can generally be solved easily. For instance, it could be composed of only point-line incidences which can be satisfied by picking any points on lines themselves randomly positioned.

Let us illustrate the complete process on the simple system S introduced in Fig. H.15. Suppose that $dist(A, C) = 0$ (*singular* point-point distance) while $dist(A, B) = k$, $k > 0$ (generic point-point distance). To compute a witness configuration $(x_A^w, y_A^w, x_B^w, y_B^w, x_C^w, y_C^w, k^w)$, we can pick points A^w and B^w at random (or read them from a sketch if available) : $A^w = (0, 0)$, $B^w = (3, -1)$; this determines the value of $k^w = \sqrt{10}$. Then we set C^w to the same coordinates as A^w so that the coincidence constraint is satisfied : $C^w = (0, 0)$.

Now, the basis of the infinitesimal motions of S contains only 3 vectors : $\dot{X}_1(P) = (1, 0, 1, 0, 1, 0)$ (the translation in x), $\dot{X}_2(P) = (0, 1, 0, 1, 0, 1)$ (the translation in y) and $\dot{X}_3(P) = (0, 0, 1, 3, 0, 0)$ (the rotation around A). Indeed, $\dot{X}_4(P) = (0, 0, 0, 0, 0, 0)$ in this witness configuration P and is thus not independent from the 3 other motion vectors. Since this basis reduces to an infinitesimal displacement basis, we conclude that S is rigid at the witness configuration, and so must be all its solutions. Remark that S remains generically under-rigid.

Properties of the WCM

The WCM has some interesting properties, in particular in comparison to graph-based characterizations of rigidity :

- the WCM can be used for numerical geometric theorem proving. Indeed, if the witness configuration has some property (alignment of 3 points for instance), then this property is a consequence of the singular constraints of the system.
- the WCM can detect not only the structural dependences identified by structural methods, but also other subtle non structural ones, e.g., the double-banana configuration (see Fig. H.4) and other similar 3D configurations (see Fig. H.16). Michelucci et al. argue that the WCM detects all the dependences due to geometric theorems, which occurs each time the system contains the hypotheses and the conclusion (or its negation) of a geometric theorem.
- the WCM can detect bad values of the parameters in the constraints. Indeed, if a witness configuration exists and still the solver fails, i.e., the reduced system S' can be solved but not the complete system S, it is due to bad numerical values of the parameters of the constraints, e.g., 3 distance constraints that violate the triangular inequality.

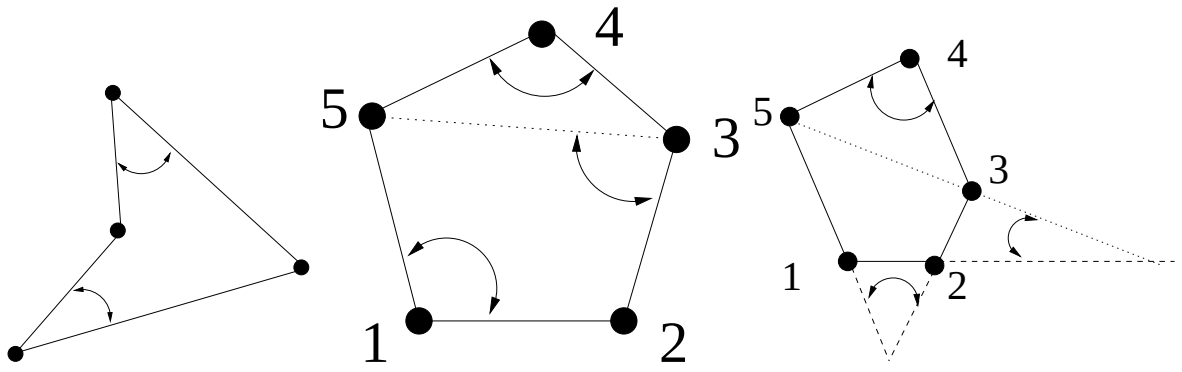


FIG. H.16 – Three 3D configurations (courtesy by A. Ortuzar, Dassault Systèmes).

These properties make the WCM a very interesting and useful tool for modeling, solving and debugging geometric constraint systems.

H.9.2 Decomposing with the WCM

The basic step in many decomposition methods is to determine whether a part of the system is well-constrained or not, for the purpose of identifying solvable components for the decomposition. The WCM can achieve this task with a higher accuracy than typical structural approaches. Moreover, the WCM can handle more general systems than most of these approaches. Hence, the method should be able to extend the capabilities (generality and reliability) of structural methods in the four categories presented in this survey. At least, Michelucci et al. have proposed a recursive division method which uses solely the WCM.[207]

A WCM-based recursive division method

Let us call `WCM-RD` the recursive division method based on the WCM described in this section. This method operates as follows :

1. find a set of maximal rigid subsystems $\{S_1, ..., S_k\}$ using the WCM,
2. for each S_i, select a constraint e in S_i and apply `WCM-RD` to $S_i \setminus \{e\}$.

In the first step of this algorithm, the WCM is used to identify each maximal (w.r.t. set inclusion) rigid subsystem. For this purpose, an *anchor* A is selected at random in the system S. An anchor is a small subset of objects which fixes all the possible displacements when determined. Pairs of non coincident points in 2D, and triples of non aligned points in 3D are examples of such anchors.

To determine a maximal rigid subsystem with respect to an anchor A, the WCM computes the motions of each geometric entity o relatively to the anchor A. If the basis of these motions is empty, then o is fixed relatively to A. Once all objects fixed relatively to A have been identified, they form (with A) a maximal rigid subsystem.

To find a maximal set of maximal rigid subsystems in S, the process described above is repeated for all possible anchors in sequence.

In the second step, the `WCM-RD` method is recursively applied to every maximal rigid subsystem found at step (1) after having removed a constraint picked at random. The removed constraint is used for the assembly of the subsystems its removal generates. The method is illustrated in Fig. H.17.

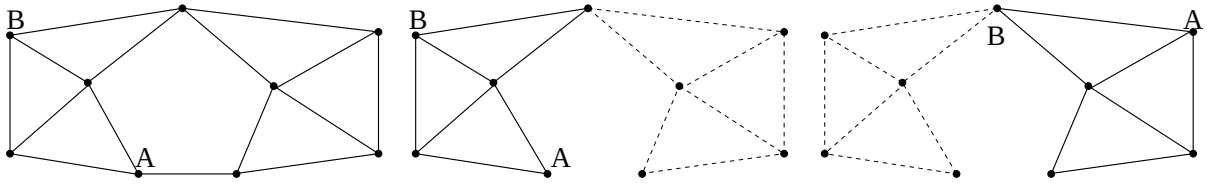


FIG. H.17 – First steps of the `WCM-RD` decomposition applied to a 2D bar framework. Maximal rigid parts with anchor AB are represented with solid lines. Middle and right : Removing one constraint creates two maximal rigid parts.

Properties

This algorithm is correct with respect to the WCM characterization of rigidity. It runs in polynomial time provided that the witness configuration is obtained in polynomial time : the number of anchors is polynomial, generally $O(n^2)$ in 2D and $O(n^3)$ in 3D, and each constraint is removed at most once. Note that a single witness configuration is used during the whole process.

H.10 Conclusion

Decomposition methods are appealing for the drastic gain in efficiency they offer (see Ref. [155] for a performance comparison). Moreover, decomposition methods help the user to debug its constraint systems by identifying under-/over-constrained components before they are solved and by localizing, inside small subsystems, the problems occurring at solving time.

Significant progress has been accomplished during the last decade. The rigidity theory has brought a solid base to the geometric methods. Although the four categories of approaches have gained in generality and reliability, a significant effort needs still to be done for meeting real-life requirements of the most challenging applications like CAD. We foresee that the design of novel hybrid approaches will allow significant advances towards practical requirements. To be specific, we think that a method that resorts as much as possible to predefined guarded patterns, and uses a general DOF-based constriction check enhanced by a WCM-based validation in a recursive assembly fashion (allowing to interleave decomposition and recombination phases), would be far better than any existing method w.r.t. generality and reliability.

Acknowledgments

The authors would like to heartily thank Dominique Michelucci and Pascal Schreck for their involvement and invaluable advices in preparing this survey. Also thanks to Marc Gouttefarde for useful comments.

Annexe I

Improving Inter-Block Backtracking with Interval Newton

Article [230] : publié dans la revue Constraints en 2009

Auteurs : Bertrand Neveu, Gilles Trombettoni, Gilles Chabert

Abstract

Inter-block backtracking (`IBB`) computes all the solutions of sparse systems of nonlinear equations over the reals. This algorithm, introduced in 1998 by Bliek et al., handles a system of equations previously decomposed into a set of (small) $k \times k$ sub-systems, called blocks. Partial solutions are computed in the different blocks in a certain order and combined together to obtain the set of global solutions. When solutions inside blocks are computed with interval-based techniques, `IBB` can be viewed as a new interval-based algorithm for solving decomposed systems of nonlinear equations.

Previous implementations used Ilog Solver and its `IlcInterval` library as a black box, which implied several strong limitations. New versions come from the integration of `IBB` with the interval-based library `Ibex`. `IBB` is now reliable (no solution is lost) while still gaining at least one order of magnitude w.r.t. solving the entire system. On a sample of benchmarks, we have compared several variants of `IBB` that differ in the way the contraction/filtering is performed *inside* blocks and is shared *between* blocks. We have observed that the use of interval Newton inside blocks has the most positive impact on the robustness and performance of `IBB`. This modifies the influence of other features, such as intelligent backtracking. Also, an incremental variant of inter-block filtering makes this feature more often fruitful.

Keywords : intervals, decomposition, solving sparse systems

I.1 Introduction

Interval techniques are promising methods for computing all the solutions of a system of nonlinear constraints over the reals. They are general-purpose and are becoming more and more efficient. They are having an increasing impact in several domains such as robotics [202] and robust control [147]. However, it is acknowledged that systems with hundreds (sometimes tens) nonlinear constraints cannot be tackled in practice.

In several applications made of nonlinear constraints, systems are sufficiently sparse to be decomposed by equational or geometric techniques. CAD, scene reconstruction with geometric constraints [306], molecular biology and robotics represent such promising application fields. Different techniques can be used to decompose such systems into $k \times k$ *blocks*. Equational decomposition techniques work on the *constraint graph* made of variables and equations [34, 156]. The simplest equational decomposition method computes a maximum matching of the constraint graph. The strongly connected components (i.e., the cycles) yield the different blocks, and a kind of triangular form is obtained for the system. When equations model geometric constraints, more sophisticated geometric decomposition techniques generally produce smaller blocks. They work directly on a geometric view of the entities and use a rigidity property [151, 129, 156].

Once the decomposition has been obtained, the different blocks must be solved in sequence. An original approach of this type has been introduced in 1998 [34] and improved in 2003 [155]. *Inter-Block Backtracking* (`IBB`) follows the partial order between blocks yielded by the decomposition, and calls a solver to compute the solutions in every block. `IBB` combines the obtained partial solutions to build the solutions of the problem.

Contributions

The new versions of `IBB` described in this paper make use of our new interval-based library called `Ibex` [52, 50].

- `Ibex` allows `IBB` to become reliable (no solution is lost) while still gaining one or several orders of magnitude w.r.t. solving the system as a whole.
- An extensive comparison on a sample of decomposed numerical CSPs allows us to better understand the behavior of `IBB` and its interaction with interval analysis.
- The use of an interval Newton operator inside blocks has the most positive impact on the robustness and performance of `IBB`. Interval Newton modifies the influence of other features, such as intelligent backtracking and filtering on the whole system (inter-block filtering – IBF).
- An incremental implementation of inter-block filtering leads to a better performance.
- It is counterproductive to filter inside blocks with `3B` [186] rather than with `2B`. However, first experiments show that using `3B` only inside large blocks might be fruitful.

I.2 Assumptions

We assume that the systems have a finite set of solutions. This condition also holds on every sub-system (block), which allows IBB to combine together a finite set of partial solutions. Usually, to produce a finite set of solutions, a system must contain as many equations as variables. In practice, the problems that can be decomposed are often under-constrained and have more variables than equations. However, in existing applications, the problem is made square by assigning an initial value to a subset of variables called *input parameters*. The values of input parameters may be given by the user (e.g., in robotics, the degrees of freedom, determined during the design of the robot, serve to pilot it), read on a sketch, or are given by a preliminary process (e.g., in scene reconstruction [306]).

I.3 Description of IBB

IBB works on a **Directed Acyclic Graph** of blocks (in short **DAG**) produced by any decomposition technique. A **block** i is a sub-system containing equations and variables. Some variables in i, called **input variables** (or parameters), are replaced by values when the block is solved. The other variables are called **(output) variables**. There exists an arc from a block i to a block j iff an equation in j involves at least one input variable assigned to a "value" in i. The block i is called a parent of j. The DAG implies a partial order in the solving process.

I.3.1 Example

To illustrate the principle of IBB, we take the 2D mechanical configuration example introduced in [34] (see Fig. I.1). Various points (white circles) are connected with rigid rods (lines). Rods impose a distance constraint between two points. Point h (black circle) is attached to the rod $\langle g, i\rangle$. The distance from h to i is one third of the distance from g to i. Finally, point d is constrained to slide on the specified line. The problem is to find a feasible configuration of the points so that all constraints are satisfied. An equational decomposition method produces the DAG shown in Fig. I.1-right. Points a, c and j constitute the input parameters (see Section I.2).

I.3.2 Description of IBB[BT]

The algorithm IBB[BT] is a simple version of IBB based on a chronological backtracking (BT). It uses several arrays :

- `solutions[i,j]` is the j^{th} solution of block i.
- `#sols[i]` is the number of solutions in block i.
- `solIndex[i]` is the index of the current solution in block i (between 0 and `#sols[i]` -1).
- `assignment[v]` is the current value assigned to variable v.

Respecting the order of the DAG, IBB[BT] follows one of the induced total orders, yielded by the list `blocks`. The blocks are solved one by one. The procedure `BlockSolve` computes the

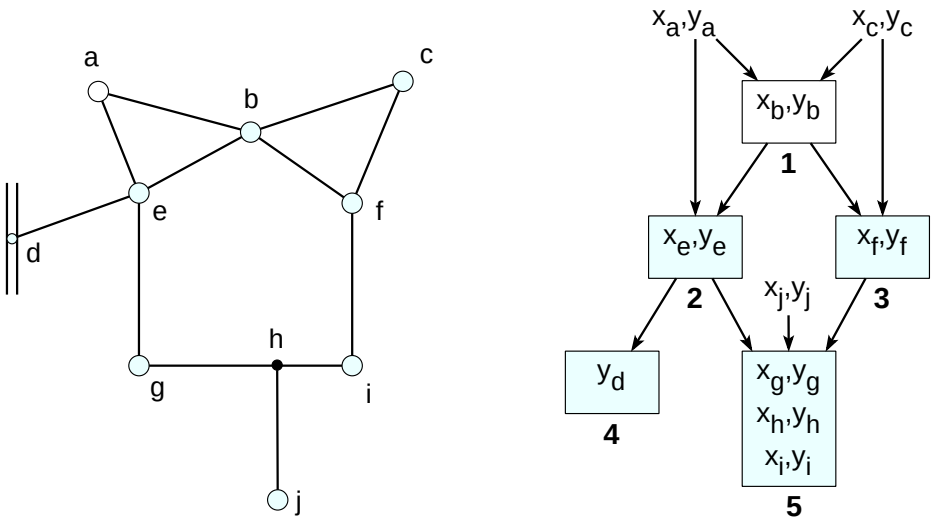


FIG. I.1 – Didactic problem and its DAG.

solutions of `blocks[i]`. It stores them in `solutions` and computes `#sols[i]`, the number of solutions in block i. The found solutions are assigned to block i in a combinatorial way. (The procedure `assignBlock` instantiates the variables in the block : it updates `assignment` with the values given by `solutions [i, solIndex[i] ]`.) The process proceeds recursively to the next block $i+1$ until a solution for the last block is found : the values in `assignment` are then stored by the procedure `storeTotalSolution`. Of course, when a block has no (more) solution, we have to backtrack, i.e., the next solution of block $i-1$ is chosen, if any.

The reader should notice a significant difference between `IBB[BT]` and the chronological backtracking schema used in finite-domain CSPs. The domains of variables in a CSP are static, whereas the set of solutions of a given block may change every time it is solved. Indeed, the system of equations itself may change from a call to another because the input variables, i.e., the parameters of the equation system, may change. This explains the use of the variable `recompute` set to *true* when the algorithm goes to a block downstream.

Let us emphasize this point on the didactic example. `IBB[BT]` follows one total order, e.g., block 1, then 2, 3, 4, and finally 5. Calling `BlockSolve` on block 1 yields two solutions for x_b. When one replaces x_b by one of its two values in the equations of subsequent blocks (2 and 3), these equations have a different coefficient x_b. Thus, in case of backtracking, block 2 must be solved twice, and with different equations, one for each value of x_b.

I.4 `IBB` with interval-based techniques

`IBB` can be used with any type of solver able to compute all the solutions of a system of equations (over the real numbers). In a long term, we intend to use `IBB` for solving systems of geometric

```
Algorithm IBB_c[BT] (blocks : a list of blocks, #blocks : the number of blocks)
    i ← 1
    recompute ← true
    while i ≥ 1 do
        if recompute then
            BlockSolve (blocks, i, solutions, #sols)
            solIndex[i] ← 0
        end
        if solIndex[i] ≥ #sols[i] /* all solutions of block i have been explored */ then
            i ← i − 1
            recompute ← false
        else
            /* solutions [i, solIndex[i] ] is assigned to block i */
            assignBlock (i, solIndex[i], solutions, assignment)
            solIndex[i] ← solIndex[i] +1
            if (i == #blocks) /* total solution found */ then
                storeTotalSolution (assignment)
            else
                i ← i + 1
                recompute ← true
            end
        end
    end
end.
```

constraints in CAD applications. In such applications, certain blocks will be solved by interval techniques while others, corresponding to theorems of geometry, will be solved by parametric hard-coded procedures obtained (off-line) by symbolic computation. In this paper, we consider only interval-based solving techniques, and thus view IBB as an interval-based algorithm for solving decomposed systems of equations.

I.4.1 Background in interval-based techniques

We present here a brief introduction of the most common interval-based operators used to solve a system of equations. The underlying principles have been developed in interval analysis and in constraint programming communities.

The whole system of equations, as well as the different blocks in the decomposition, are viewed as numerical CSPs.

Definition 16 *A numerical CSP $P = (X, C, B)$ contains a set of constraints C and a set X of n variables. Every variable $x_i \in X$ can take a real value in the interval $\mathbf{x_i}$ ($B = \mathbf{x_1} \times ... \times \mathbf{x_n}$). A solution of P is an assignment of the variables in V such that all the constraints in C are satisfied.*

The n-set of intervals B is represented by an n-dimensional parallelepiped called a **box**. Since real numbers cannot be represented in computer architectures, the bounds of an interval $\mathbf{x_i}$ should actually be defined as floating-point numbers. A solving process reduces the initial box until a very small box is obtained. Such a box is called an **atomic box** in this paper. In theory, an interval could be composed by two consecutive floats in the end. In practice, the process is interrupted when all the intervals have a width less than `w_biss`, where `w_biss` is a user-defined parameter. It is worthwhile noting that an atomic box does not necessarily contain a solution. Indeed, evaluating an equation with interval arithmetic may prove that the equation has no solution (when the image of the corresponding box does not contain 0), but cannot assert that there exists a solution in the box. However, several operators from interval analysis can often certify that there exists a solution inside an atomic box.

Our interval-based solver `Ibex` uses interval-based operators to handle the blocks (`BlockSolve`). In the most sophisticated variant of `IBB`, the following three steps are iteratively performed. The process stops when an atomic box of size less than `w_biss` is obtained.

1. *Bisection :* One variable is chosen and its domain is split into two intervals (the box is split along one of its dimensions). This yields two smaller sub-CSPs which are handled in sequence. This makes the solving process combinatorial.
2. *Filtering/propagation :* Local information is used on constraints handled individually to reduce the current box. If the current box becomes empty, the corresponding branch (with no solution) in the search tree is cut [186, 293, 182].
3. *Interval analysis/unicity test :* Such operators use the first and/or second derivatives of the functions. They produce a "global" filtering on the current box. If additional conditions are fulfilled, they may ensure that a unique solution exists inside the box, thus avoiding further bisection steps.

Filtering/propagation

Propagation is performed by an AC3-like fixed-point algorithm. Several types of filtering operators reduce the bounds of intervals (no gap is created in the current box). The `2B-consistency` (also known as Hull-consistency - HC) and the `Box-consistency` [293] algorithms both consider one constraint at a time (like AC3) and reduce the bounds of the involved variables. `Box-consistency` uses an iterative process to reduce the bounds while `2B-consistency` uses projection functions. The more expensive job performed by `Box-consistency` may pay when the equations contain several occurrences of a same variable. This is not the case with our benchmarks which are mostly made of equations modeling distances between 2D or 3D points, and of other geometric constraints. Hence, `Box-consistency` has been discarded. The algorithm `3B-consistency` [186] uses `2B-consistency` as a sub-routine and a refutation principle (shaving ; similar to the Singleton Arc Consistency [64] in finite domains CSPs) to reduce the bounds of every variable iteratively. On the tested benchmarks our experiments have led us to use the `2B-consistency` operator (and sometimes `3B-consistency`) combined with an interval Newton.

Interval analysis

We have implemented in our library the interval Newton (`I-Newton`) operator [219]. `I-Newton` is an iterative numerical process, based on the first derivatives of equations, and extended to intervals. Without detailing this algorithm, it is worth understanding the output of `I-Newton`. Applied to a box B_0, `I-Newton` provides three possible answers :

1. When the Jacobian matrix is not strongly regular, the process is immediatly interrupted and B_0 is not reduced [219]. This necessarily occurs when B_0 contains several solutions. Otherwise, different iterations modify the current box B_i to B_{i+1}.
2. When B_{i+1} exceeds B_i in at least one dimension, B_{i+1} is intersected with B_i before the next iteration. No existence or unicity property can be guaranteed.
3. When the box B_{i+1} is included in B_i, then B_{i+1} is guaranteed to contain a unique solution (existence and unicity test).

In the last case, when a unique solution has been detected, the convergence onto an atomic box of width `w_biss` in the subsequent iterations is very fast, i.e., quadratic. Moreover, the width of the obtained atomic box is often very small (even less than `w_biss`), which highlights the significant reduction obtained in the last iteration (see Table I.3).

I.4.2 Interval techniques and block solving

Let us stress a characteristic of the systems corresponding to blocks when they are solved by interval-based techniques : the equations contain coefficients that are not punctual but (small) intervals. Indeed, the solutions obtained in a given block are atomic boxes and become parameters of subsequent blocks. For example, the two possible values for x_b in block 1 are replaced by atomic boxes in block 2. This characterictic has several consequences.

The precision sometimes decreases as long as blocks are solved in sequence. A simple example is the a 1×1 block $x^2 = p$ where the parameter p is $[0, 10^{-10}]$. Due to interval arithmetics, solving the block yields a coarser interval $[-10^{-5}, 10^{-5}]$ for x. Of course, these pathological cases related to the proximity to 0, occur occasionnally and, as discussed above, interval analysis renders the problem more seldom by sometimes producing tiny atomic boxes.

The second consequence is that it has no sense to talk about a unique solution when the parameters are not punctual and can thus take an infinite set of possible real values. Fortunately, the unicity test of `I-Newton` still holds. Generalizing the unicity test to non punctual parameters has the following meaning : if one takes *any* punctual real value in the interval of every parameter, it is ensured that exactly one point inside the atomic box found is a solution. Of course, this point changes according to the chosen punctual values. Although this proposition has not been published (to our knowledge), this is straightforward to extend the "punctual" proof to systems in which the parameters are intervals.

Remark

In the old version using the `IlcInterval` library of Ilog Solver [155], the unicity test was closely associated to the `Box-consistency`. We now know that the benefit of mixing `Box` and `2B` was not due to the `Box-consistency` itself, but to the unicity test that avoided bisection steps in the bottom of the search tree. It was also due to the use of the *centered form* of the equations that produced additional pruning.

I.4.3 Inter-block filtering (`IBF`)

In all the variants of `IBB`, it is possible to add an *inter-block filtering* (*IBF*) process : instead of limiting the filtering process (e.g., `2B`) to the current block i, we apply the filtering process to the entire system.

In Section I.6.2, we will detail a more incremental version of *IBF*, called *IBF+*, in which the filtering process is applied to only certain blocks, called *friend blocks* (those that can be filtered).

I.5 Different versions of `IBB`

Since 1998, several variants of `IBB` have been implemented [34, 155]. We can classify them into three main categories from the simplest one to the most sophisticated one. They differ in the way they manage, during the search for solutions, the current block (with procedure `BlockSolve`) and the other blocks of the system.

The third version $\texttt{IBB}_c$ is the most sophisticated one. It corresponds to the pseudo-code described in Section I.3.2. $\texttt{IBB}_c$ defines one system per block in the decomposition. Data structures are used to manage the inter-block backtracking : for storing and restoring solutions of blocks, domains of variables and so on.

The first two versions are sufficiently simple to be directly integrated into `Ibex`.

$\texttt{IBB}_a$ can be viewed as a new splitting heuristic in a classical interval-based solving algorithm handling the whole system.

- $\texttt{IBB}_a$ handles the entire system of equations as a numerical CSP. Most of the operations, such as `2B` or `I-Newton`, are thus applied to the whole system so that $\texttt{IBB}_a$ necessarily calls inter-block filtering.
- The decomposition (i.e., the DAG of blocks) is just used to feed the new splitting heuristic. $\texttt{IBB}_a$ can choose the next variable to be split only inside the current block i. The specific variable inside the block i is chosen with a standard *round robin* strategy.

The second version $\texttt{IBB}_b$ is a bit more complicated and manages two systems at a time : the whole system, like for $\texttt{IBB}_a$, but also the current block which is managed as an actual system of equations by Ibex. Like for $\texttt{IBB}_a$, bisections are done in the whole system (selecting one variable in the current block). Also, it is possible to run a filtering process on the whole system, implementing a simple version of inter-block filtering.

Contrarily to IBB$_a$:

– It is possible to de-activate interblock-filtering (in the whole system).
– It is possible to run 2B and/or I-Newton in the current block only.

When 2B is run both in the current block and in the entire system (IBF), the two filtering processes are managed as follows. First, 2B is run on the current block until a fixed-point is reached. Second, 2B is run on the entire system.

The experiments will show that IBB$_a$ is not competitive with the other two versions because IBB$_a$ cannot run I-Newton in the current block. They also will show that IBB$_c$ is more robust than IBB$_b$ and can incorporate the sophisticated features described below.

I.6 Advanced features in IBB$_c$

The following sections mention or detail how are implemented advanced features in IBB$_c$ (called IBB for simplicity) : intelligent backtracking, the *recompute condition* which is a simple way to exploit the partial order between blocks provided by the decomposition, a sophisticated variant of inter-block filtering. We also detail the advantages of using interval-Newton inside blocks.

I.6.1 Exploiting the DAG of blocks

As shown in Section I.3.2, IBB[BT] uses only the total order between blocks and forgets the actual dependencies between them. However, IBB$_c$ is flexible enough to exploit the partial order between blocks. Figure I.1-right shows an example. Suppose block 5 had no solution. Chronological backtracking would go back to block 4, find a different solution for it, and solve block 5 again. Clearly, the same failure will be encountered again in block 5.

It is explained in [34] that the *Conflict-based Backjumping* and *Dynamic backtracking* schemes cannot be used to take into account the structure given by the DAG. Therefore, an intelligent backtracking, called IBB[GPB], was introduced, based on the *partial order backtracking* [198, 34]. In 2003, we have also proposed a simpler variant IBB[GBJ] [152] based on the *Graph-based BackJumping* (GBJ) proposed by Dechter [65].

However, there is an even simpler way to exploit the partial order yielded by the DAG of blocks : the **recompute condition**. This condition states that it is useless to recompute the solutions of a block with BlockSolve if the parent variables have not changed. In that case, IBB can reuse the solutions computed the last time the block has been handled. In other words, when handling the next block $i+1$, the variable recompute is not always set to *true* (see Section I.3.2). This condition has been implemented in IBB[GBJ] and in IBB[BT]. In the latter case, the variant is named IBB[BT+].

Let us illustrate how IBB[BT+] works on the didactic example. Suppose that the first solution of block 3 has been selected, and that the solving of block 4 has led to no solution. IBB[BT+] then backtracks on block 3 and the second position of point f is selected. When IBB[BT+] goes down again to block 4, that block should normally be recomputed from scratch due to the modification

of f. But x_f and y_f are not implied in equations of block 4, so that the two solutions of block 4, which had been previously computed, can be reused. It is easy to avoid this useless computation by using the `DAG` : when `IBB` goes down to block 4, it checks that the parent variables x_e and y_e have not changed.

Remark

Contrarily to $\texttt{IBB}_a$ and $\texttt{IBB}_b$, the recompute condition can be incorporated into $\texttt{IBB}_c$ thanks to the management of sophisticated data structures.

I.6.2 Sophisticated implementation of inter-block filtering (IBF+)

IBF is integrated into $\texttt{IBB}_b$ and $\texttt{IBB}_c$ in the following way. When a bisection is applied to a variable in a given block i, the filtering operators described above, i.e., `2B` and `I-Newton`, are first called inside the block. Second, *IBF* is launched on the entire system.

In the lattest versions of $\texttt{IBB}_b$ and $\texttt{IBB}_c$, *IBF* is launched in a more incremental way. The underlying local filtering (e.g., `2B`) is run with a propagation queue initially filled with only the variables inside the current block, which lowers the overhead related to *IBF* when it is not efficient.

To perform a more sophisticated implementation of *IBF*, called *IBF+*, before solving a block i, one forms a subsystem extracted from the *friend blocks* F_i' of block i. The filtering process will concern only the friend blocks, thus avoiding the management of the other ones. The friend blocks of i are extracted as follows :

1. take the set $F_i = \{i...\#blocks\}$ containing the blocks not yet "instantiated",
2. keep in F_i' only the blocks in F_i that are connected to i in the `DAG`[1].

To illustrate *IBF+*, let us consider the `DAG` of the didactic example. When block 1 is solved, all the other blocks are considered by *IBF+* since they are all connected to block 1. Any interval reduction in block 1 can thus possibly perform a reduction for any variable of the system. When block 2 is solved, a reduction has potentially an influence on blocks 3, 4, 5 for the same reasons. (Notice that block 3 is a friend block of block 2 that is not downstream to block 2 in the DAG.) When block 3 is solved, a reduction can have an influence only on block 5. Indeed, once blocks 1 and 2 have been removed (because they are "instantiated"), block 3 and 4 do not belong anymore to the same connected component. Hence, no propagation can reach block 4 since the parent variables of block 5, which belong to block 2, have an interval of width at most `w_biss` and thus cannot be reduced further.

IBF+ implements only a local filtering on the friend blocks, e.g., `2B-consistency` on the tested benchmarks. It turns out that `I-Newton` is counterproductive in *IBF*. First, it is expensive to compute the Jacobian matrix of the whole system. More significantly, it is likely that `I-Newton`

[1]The orientation of the `DAG` is forgotten at this step, that is, the arcs of the `DAG` are transformed into non-directed edges, so that the filtering can also be applied on friend blocks that are not directly linked to block i.

does not prune at all the search space (except when handling the last block) because it always falls in the singular case. As a rule of thumb, if the domain of one variable x in the last block contains two solutions, then the whole system will contain at least two solutions until x is bisected. This prevents `I-Newton` from pruning the search space. This explains why $\texttt{IBB}_a$ is not competitive with the two other versions of `IBB`.

The experiments confirm that it is always fruitful to perform a sophisticated filtering process inside blocks (i.e., `2B + I-Newton`), whereas *IBF* or *IBF+* (on the entire system) produces sometimes, but not always, additional gains in performance.

I.6.3 Mixing *IBF* and the recompute condition

Incorporating *IBF* or *IBF+* into $\texttt{IBB}_c\texttt{[BT]}$ is straightforward. This is not the case for the variants of `IBB` with more complicated backtracking schemes. Reference [152] gives guidelines for the integration of *IBF+* into `IBB[GBJ]`. More generally, *IBF+* adds in a sense some edges between blocks. It renders the system less sparse and complexifies the recomputation condition. Indeed, when *IBF+* is launched, the parent blocks of a given block i are not the only exterior causes of interval reductions inside i. The friend blocks of i have an influence as well and must be taken into account.

For this reason, when *IBF+* is performed, the recompute condition is more often true. Since the causes of interval reductions are more numerous, it is more seldom the case that all of them have not changed. This will explain for instance why the gain in performance of `IBB[BT+]` relatively to `IBB[BT]` is more significant than the gain of $\texttt{IBB}_c\texttt{[BT+,IBF+]}$ relatively to $\texttt{IBB}_c\texttt{[BT,IBF+]}$ (see experiments).

This remark holds even more for the simple *IBF* implementation where local filtering is run on the entire system (and not only on friend blocks). In this case, the recompute condition is simply always true, so that `BT+` becomes completely inefficient and avoids the recomputation of zero block. In other terms, $\texttt{IBB}_c\texttt{[BT,IBF]}$ and $\texttt{IBB}_c\texttt{[BT+,IBF]}$ are quasi-identical.

I.6.4 Discarding the non reliable midpoint heuristic

The integration of the Ibex solver underlies several improvements of `IBB`. As previously mentioned, using a white box allows us to better understand what happens. Also, $\texttt{IBB}_c$ is now reliable. The *parasitic solutions* problem has been safely handled (see Section I.6.5) and an ancient **midpoint heuristic** is now abandoned. This heuristic replaced every parameter, i.e., input variable, of a block by the midpoint of its interval. Such a heuristic was necessary because the `IlcInterval` library previously used did not allow the use of interval coefficients. Ibex accepts non punctual coefficients so that no solution is lost anymore, thus making `IBB` reliable. The midpoint heuristics would however allow the management of sharper boxes, but the gain in running time would be less than 5% in our benchmarks. The price of reliability is not so high!

I.6.5 Handling the problem of *parasitic solutions*

With interval solving, *parasitic solutions* are obtained when :

– several atomic boxes are returned by the solver as possible solutions ;
– these boxes are close one to each other ;
– only one of them contains an actual solution and the others are not discarded by filtering.

Even when the number of parasitic solutions is small, `IBB` explodes because of the multiplicative effect of the blocks, i.e., because the parasitic partial solutions are combined together. In order that this problem occurs more rarely, one can reduce the precision (i.e., enlarge `w_biss`) or mix several filtering techniques together. The use of interval analysis operators like `I-Newton` is also a right way to fix most of the pathological cases (see experiments).

It appears that $\texttt{IBB}_a$ and $\texttt{IBB}_b$ are not robust against the parasitic solutions problem that often occurs in practice. $\texttt{IBB}_c$ handles this problem by taking the union of the close boxes (i.e., the hull of the boxes). $\texttt{IBB}_c$ considers the different blocks separetely, so that all the solutions of a block can be computed before solving the next one. This allows $\texttt{IBB}_c$ to merge close atomic boxes together. Note that the previous implementations of $\texttt{IBB}_c$ might lose some solutions because no hull between close boxes was performed. Instead, only one of the close boxes was selected and might lead to a failure in the end when the selected atomic box did not contain a solution.

I.6.6 Certification of solutions

As mentioned in Section I.4.1, certifying the existence and the unicity of a solution inside an atomic box returned by the solver requires interval analysis techniques. With `IBB`, we use an interval Newton to contract every block and to guarantee the solutions that are inside. `IBB` can thus often certify solutions of a decomposed system. Indeed, a straightforward induction ensures that *a total solution is certified iff all the corresponding partial solutions are certified in every block.*

Among the ten benchmarks studied below, only solutions of `Mechanism` and `Chair` have not been certified.

I.6.7 Summary : benefit of running `I-Newton` inside blocks

Finally, as shown in the experiments reported below, the most significant impact on `IBB` is due to the integration of an interval Newton inside the blocks :

– `I-Newton` has a good power of filtering, thus reducing time complexity.
– Due to its quadratic convergence, `I-Newton` often allows us to reach the finest precision, i.e. an even better precision than `w_biss`. This is of great interest because the solutions of a given block become coefficients (input paramaters) in blocks that are downstream in the DAG. Thus, when no `I-Newton` is used, the precision may decrease during block solving, generally obtaining at the end a precision which is worse than `w_biss`.
  Second, with thiner input parameters, the loss in performance of `IBB`, as compared to the use

of the discarded (non reliable) midpoint heuristic, is negligible.
– `I-Newton` often allows us to certify the solutions.
– Hence, the combinatorial explosion due to parasitic solutions is drastically limited.

Moreover, the use of `I-Newton` alters the comparison between variants of `IBB`. In particular, in the previous versions, we concluded that *IBF* was counterproductive, whereas it is not always true today. Also, the interest of intelligent backtracking algorithms is clearly put into question, which confirms the intuition shared by the constraint programming community that a better filtering (due to `I-Newton`) removes backtracks (and backjumps). Moreover, since `I-Newton` has a good filtering power, obtaining an atomic box requires less bisections. Hence, the number of calls to *IBF* is reduced in the same proportion.

I.7 Experiments

We have applied several variants of `IBB` on the benchmarks described above.

I.7.1 Benchmarks

Exhaustive experiments have been performed on 10 benchmarks made of geometric constraints. They compare different variants of `IBB` and show a clear improvement w.r.t. solving the whole system.

Some benchmarks are artificial problems, mostly made of quadratic distance constraints. The problems `Mechanism` and `Tangent` have been found in [181] and [41]. `Chair` is a realistic assembly made of 178 equations induced by a large variety of geometric constraints : distances, angles, incidences, parallelisms, orthogonalities [152].

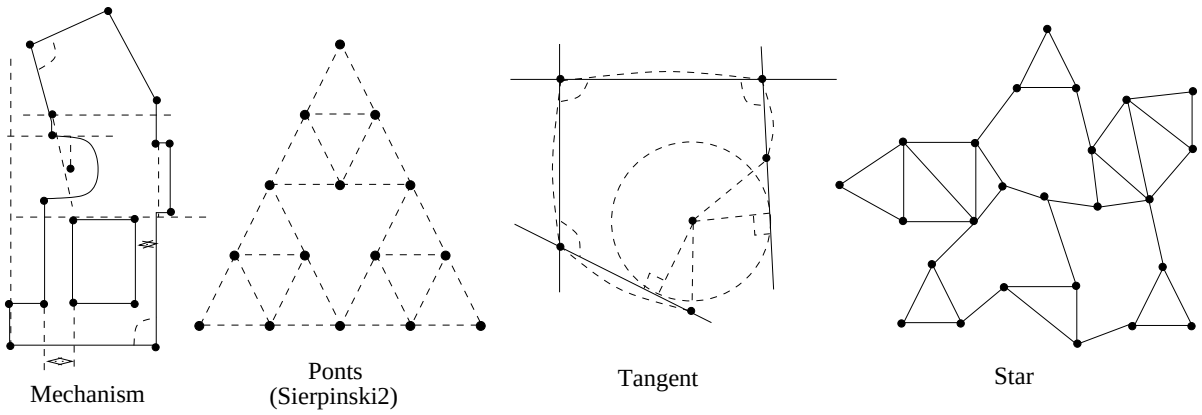


FIG. I.2 – 2D benchmarks : general view

The `DAGs` of blocks for the benchmarks have been obtained either with an equational method (abbrev. equ. in Table I.1) or with a geometric one (abbrev. geo.). `Ponts` and `Tangent` have been

decomposed by both techniques.

A problem defined with a domain of width 100 (see column 6 of Table I.1) is generally similar to assigning $(-\infty, +\infty)$ to every domain. The intervals in `Mechanism` and `Sierp3` have been selected around a given solution in order to limit the total number of solutions. In particular, the equation system corresponding to `Sierp3` would have about 2^{40} solutions, so that the initial domains are limited to a width 1. `Sierp3` is the *Sierpinski* fractal at level 3, that is, 3 Sierpinski at level 2 (i.e., `Ponts`) put together. The time spent for the equational and geometric decompositions is always negligible, i.e., a few milliseconds for all the benchmarks.

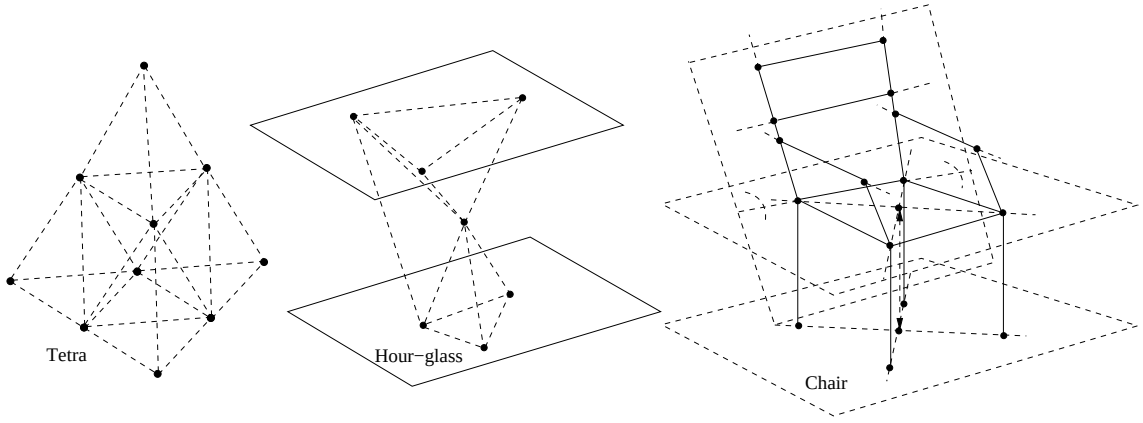


FIG. I.3 – 3D benchmarks : general view

GCSP	Dim.	Dec.	Size	Size of blocks	Dom.	#sols	`w_biss`
`Mechanism`	2D	equ.	98	98 = 1x10, 2x4, 27x2, 26x1	10	448	5.10^{-6}
`Sierp3`		geo.	124	124 = 44x2, 36x1	1	198	10^{-8}
`PontsE`		equ.	30	30 = 1x14, 6x2, 4x1	100	128	10^{-8}
`PontsG`		geo.	38	38 = 13x2, 12x1	100	128	10^{-8}
`TangentE`		equ.	28	28 = 1x4, 10x2, 4x1	100	128	10^{-8}
`TangentG`		geo.	42	42 = 2x4, 11x2, 12x1	100	128	10^{-8}
`Star`		equ.	46	46 = 3x6, 3x4, 8x2	100	128	10^{-8}
`Chair`	3D	equ.	178	178=1x15,1x13,1x9,5x8,3x6,2x4,14x3,1x2,31x1	100	8	5.10^{-7}
`Tetra`		equ.	30	30 = 1x9, 4x3, 1x2, 7x1	100	256	10^{-8}
`Hourglass`		equ.	29	29 = 1x10, 1x4, 1x3, 10x1	100	8	10^{-8}

TAB. I.1 – Details about the benchmarks. Type of decomposition method (Dec.) ; number of equations (Size) ; Size of blocks : NxK means N blocks of size K ; Interval widths of variables (Dom.) ; number of solutions (#sols) ; bisection precision, i.e., domain width under which bisection does not split intervals (`w_biss`).

I.7.2 Brief introduction to Ibex

All the tests with IBB have been conducted using the interval-based library, called Ibex, implemented in C++ by the third author [52, 50]. The hull consistency (i.e., 2B-consistency) is implemented with the famous HC4 that builds a syntactic tree for every constraint [28, 182]. A "width" parameter r_hc4 must be tuned : a constraint is pushed in the propagation queue if the projection on one of its variables has reduced the corresponding intervals more than r_hc4 (ratio of interval width). I-Newton is run when the largest interval in the current box has a width less than ceiling_newton. For using 3B-consistency [186], one must specify the width of the smallest interval that the algorithm tries to refute. This parameter ratio_var_shave (in short r_vs) is given as a ratio of interval width. Two atomic boxes are merged iff a unique solution has not been certified inside both and the boxes are sufficiently close to each other, that is, for every variable, there is a distance dist less than 10^{-2} between the two boxes (10^{-4} for the benchmark Star). Most of the reported CPU times have been obtained on a Pentium IV 3 Ghz.

I.7.3 Interest of system decomposition

The first results show the dramatic improvement due to IBB as compared to four interval-based solvers. All the solvers use a round-robin splitting strategy. Ilog Solver [141] uses a filtering process mixing 2B-consistency, Box-consistency and an interval analysis operator for certifying solutions. The relatively bad CPU times simply show that the IlcInterval library has not been improved for several years. They have been obtained on a Pentium IV 2.2 Ghz.

GCSP	Ibex	RealPaver	Ilog Solver	IBB$_c$[BT+]	IBB$_b$[BT,IBF]	$\frac{Ibex}{IBB}$	Precision
Mechanism	$>$ 4000 (117)	XXX	$>$ 4000	1.55	1.65	75	2.10^{-5}
Sierp3	36	$>$ 4000	$>$ 4000	4	1.08	33	4.10^{-11}
Chair	$>$ 4000	XXX	$>$ 128 eq.	0.36	1.36	$> 10^4$	10^{-7}
Tetra	18.2	42	$>$ 4000	0.96	1.42	19	2.10^{-14}
PontsE	7.2	6.9	103	0.97	1.21	7	7.10^{-14}
PontsG	3.2	1.9	294	1.52	0.43	8	10^{-13}
Hourglass	0.25	0.32	247	0.019	0.029	13	10^{-13}
TangentE	9.6	22*	191	0.15	0.15	64	5.10^{-14}
TangentG	14.6	XXX	XXX	0.10	0.16	146	4.10^{-14}
Star	18.8	12	1451	0.18	0.15	125	2.10^{-11}

TAB. I.2 – Interest of IBB. The columns in the left report the times (in seconds) spent by three interval-based solvers to handle the systems globally. The column IBB$_c$[BT+] and IBB$_b$[BT,IBF] report the CPU times obtained by two interesting variants of IBB. Gains of at least one order of magnitude are highlighted by the column *Ibex/*IBB. The last column reports the size of the obtained atomic boxes. The obtained precision is often better than the specified parameter w_biss (see Table I.1) thanks to the use of I-Newton. An entry XXX means that the solver is not able to isolate solutions (see Section I.6.5).

On Table I.2, Ibex uses a 3B+Newton filtering algorithm. RealPaver [111, 112] generally uses HC4+Newton filtering (better results are obtained by weak3B + Newton only for TangentE; the

weak 3B-consistency is described in [111]) The `3B` operator of `Ibex` behaves better than that of `RealPaver` on these benchmarks essentially because it manages a parameter `r_vs` which is a ratio of interval and not a fixed width. Note that `RealPaver` uses the default values of the different parameters. We have also applied the `Quad` operator [183] that is sometimes very efficient to solve polynomial equations. This operator appears to be very slow on the tested benchmarks.

	1	2	3	4	5	6	7
GCSP	IBB_b[BT]	IBB_c[BT]	IBB_a[BT,IBF]	IBB_b[BT,IBF]	IBB_c[BT,IBF+]	IBB_c[BT+]	IBB_c[BT+,IBF+]
	146	160	11000	165	196	155	195
Mechanism	22803	22803	35007	22111	22111	22680	22066
	1635	1635	–	1629	1629	1544	1156
	317	628	29660	**108**	307	402	258
Sierp3	35685	35685	2465	5489	5484	19454	4354
	21045	21045	–	4272	4272	12242	3455
	94	97	1080	136	163	**36**	127
Chair	7661	7661	6601	6825	6825	2368	4304
	344	344	–	344	344	97	148
	108	109	4370	142	142	**96**	127
Tetra	10521	10521	66933	9843	9843	9146	8568
	235	235	–	235	235	100	100
	99	100	706	121	123	**97**	118
PontsE	6103	6103	23389	5880	5880	5986	5776
	131	131	–	115	115	79	67
	147	233	224	**43**	71	152	71
PontsG	14253	14253	4505	2669	2669	7154	2669
	9283	9283	–	1155	1155	6585	1155
	2.1	2.7	18	2.9	4.2	**1.9**	4.2
Hourglass	59	59	341	59	59	48	59
	57	57	–	53	53	35	53
	11	15	5370	15	21	15	21
TangentE	313	313	40747	308	308	304	308
	427	427	–	427	427	423	427
	12	16	XXX	16	24	**10**	24
TangentG	817	817	–	817	816	102	817
	411	411	–	411	411	238	396
	31	35	2400	15	20	18	**6**
Star	2475	2475	5243	1347	1347	1534	192
	457	457	–	254	254	325	34

TAB. I.3 – Variants of `IBB` with `HC4+Newton` filtering inside blocks. Every entry contains three values : (top) the CPU time for obtaining all the solutions (in hundredths of second) ; (middle) the total number of bisections performed by the interval solver ; (bottom) the total number of times `BlockSolve` is called.

Tuning parameters

We would like to stress that it is even easier to tune the parameters used by `IBB` with `Ibex` (i.e., `r_hc4` and `ceiling_newton`) than the parameters used by Ibex applied to the entire system (i.e., `r_hc4`, `ceiling_newton` and `ratio_var_shave`). First, there is one less parameter to be tuned

with IBB. Indeed, as shown in Section I.7.7, this is counterproductive (w.r.t. CPU time) to use 3B-consistency with IBB. Second, the value of ceiling_newton can have a significant impact on CPU time with a global solving while it is not the case with IBB. Overall, a user of IBB must finely tune only the parameter r_hc4 used by HC4.

Table I.3 reports the main results we have obtained with several variants of IBB. Tables I.4, I.5 and I.6 highlight specific points.

I.7.4 Poor results obtained by $\texttt{IBB}_a$

Table I.3 clearly shows that $\texttt{IBB}_a$ is not competitive with $\texttt{IBB}_b$ and $\texttt{IBB}_c$. As shown in Table I.2, the results in CPU time obtained by $\texttt{IBB}_a$ are close to or better than those obtained by Ibex applied to the entire system (with 3B, I-Newton and a round-robin splitting strategy) : Tables I.2 and I.6 underline that $\texttt{IBB}_a$ is able to render Mechanism and Chair tractable.

Table I.6 also reports the results obtained by $\texttt{IBB}_a$ with 3B (i.e., *IBF* is performed by 3B). Recall that $\texttt{IBB}_a$ can be viewed as a splitting strategy driven by the decomposition into blocks (i.e., a total order between blocks). Thus, these results mainly underline that the $\texttt{IBB}_a$ splitting heuristic is better than round-robin. We see below that a relevant filtering *inside* blocks, performed in $\texttt{IBB}_b$ and $\texttt{IBB}_c$, bring an even better performance.

I.7.5 $\texttt{IBB}_b$ versus $\texttt{IBB}_c$

$\texttt{IBB}_a$ is significantly less efficient than $\texttt{IBB}_b$ and $\texttt{IBB}_c$ because it does not use I-Newton inside blocks.

The comparison between $\texttt{IBB}_b$ and $\texttt{IBB}_c$ can be summed up in several points mainly deduced from Tables I.3 and I.5 :

- All the variants of $\texttt{IBB}_b$ and $\texttt{IBB}_c$ obtain similar results on all the benchmarks (provided that HC4+Newton filtering is used inside blocks).
- $\texttt{IBB}_b$ is a simpler implementation of IBB than $\texttt{IBB}_c$. The main reason is that no sophisticated data structures are used by $\texttt{IBB}_b$. This explains that the CPU times obtained by $\texttt{IBB}_b$[BT] are better than those obtained by $\texttt{IBB}_c$[BT] (see columns 1 and 2 in Table I.3). Also, $\texttt{IBB}_b$[BT,IBF] is more efficient than $\texttt{IBB}_b$[BT,IBF] (compare column 4 of Table I.3 and column 2 of Table I.5).
  Thus, a same IBB algorithm is better implemented by the $\texttt{IBB}_b$ scheme.
- Using the BT+ backtracking scheme (related to the recompute solution) is always better than using the standard BT. The overhead is negligible and it avoids solving some blocks (compare for instance the number of solved blocks in columns 2 and 6 of Table I.3).
  This is a good argument in favor of the $\texttt{IBB}_c$ version.
- Tables I.3 and I.5 shows that the interest of *IBF* is not clear. However, it seems that *IBF* is useful for hard instances for which a lot of choice points lead to failure and backtracking. For instance, Chair has only 8 solutions and implies thrashing in the search tree.
  Table I.4 reports the only two benchmarks for which backjumps actually occur with an intelli-

gent backtracking. It shows that the gain obtained by an intelligent backtracking (`IBB[GBJ]`)) is compensated by a gain in filtering with `IBF`.
Note that *IBF* was clearly counterproductive in old versions of `IBB` that did not use `I-Newton` to filter inside blocks. Indeed, a smaller filtering power implied more bisections and thus a larger number of calls to *IBF*.

GCSP	IBB_c[BT]	IBB_c[BT+]	IBB_c[GBJ]	IBB_b[BT,IBF]	$IBBc$[BT,IBF+]	IBB_c[BT+,IBF+]	IBB_c[GBJ,IBF+]
	628	402	288	**108**	307	258	252
Sierp3	35684	19454	13062	5489	5484	4354	4260
	21045	12242	8103	4272	4272	3455	3175
			BJ=2974				BJ=135
	35	18	17	15	20	**6**	**6**
Star	2474	1534	1500	1347	1346	192	192
	457	325	277	254	254	34	34
			BJ=6				BJ=0

TAB. I.4 – No interest of intelligent backtracking. The number of backjumps is drastically reduced by the use of `IBF` ($6 \rightarrow 0$ on `Star`; $2974 \rightarrow 135$ on `Sierp3`). The times obtained with `IBF` are better than or equal to those obtained with intelligent backtracking schemes. Only a marginal gain is obtained by IBB_c[GBJ,IBF+] w.r.t. IBB_c[BT+,IBF+] for `Sierp3`.

Overall, the four versions of `IBB` that are the most efficient are IBB_b[BT], IBB_b[BT,IBF], IBB_c[BT+], IBB_c[BT+,IBF+] (see Tables I.2 and I.5).

Two other points must be considered for a fair comparison between these four versions of `IBB`.

Since IBB_b[BT,IBF] uses an incremental *IBF* (i.e., pushing initially in the propagation queue only the variables of the current block), the overhead w.r.t. IBB_b[BT] is small : it is less than 50% when *IBF* does not reduce anything. If you compare the numbers of bisections and solved blocks in columns 1 and 4 of Table I.3, these numbers are close in the two columns for `Mechanism`, `Chair`, `Tetra`, `PontsE`, `Hourglass`, `TangentE`, `TangentG`, which indicates that *IBF* prunes nothing or only a few. However, the loss in performance of IBB_b[BT, IBF] lies (only) between 15% and 50%. The gain for the three other instances is substantial.

This suggests than IBB_b[BT,IBF] is more robust (w.r.t. the CPU time complexity) than IBB_b[BT], making it more attractive.

The second point cannot be deduced from the tables because it is related to robustness.

Experiments with old versions of `IBB` without `I-Newton` inside blocks clearly showed the combinatorial explosion of IBB_b involved by the parasitic solution problem. The use of `I-Newton` limits this problem, except for `Mechanism`. Instead of computing the 448 solutions, IBB_b[BT] and IBB_b[BT,IBF] compute 680 solutions because it is not endowed with the parasitic solutions merging. However, it is important to explain that the problem needed also to be fixed by manually selecting an adequate value for the parameter `w_biss`. In particular, IBB_b undergoes a combinatorial explosion on `Chair` and `Mechanism` when the precision is higher (i.e., when `w_biss` is smaller). On the contrary, the IBB_c version automatically adjusts `w_biss` in every block according to the width of the largest input variable (parameter) interval. IBB_c is thus more robust

than $\texttt{IBB}_b$ and can add sophisticated features such as solution merging, the recompute condition and a finer implementation of *IBF* (with friend blocks).

These observations lead us to recommend 2 (or 3) versions of `IBB` :

– $\texttt{IBB}_b$`[BT,IBF]` which is simple to be implemented (available in `Ibex`) and is often efficient when `I-Newton` behaves well ;
– the more sophisticated version $\texttt{IBB}_c$`[BT+]` (generally without and sometimes with *IBF+*) that is very useful when the interval-based solver cannot isolate solutions.

The following sections detail some points leading to the above recommendation.

I.7.6 *IBF* versus *IBF+*

Table I.5 clearly shows that the *IBF+* implementation, that can only be used with $\texttt{IBB}_c$, leads to gains in performance w.r.t. the simple *IBF*.

One can also observe that *IBF+* lowers the interest of `BT+` w.r.t. to `BT`.

I.7.7 `IBB` and `3B`

Table I.6 yields some indications about the interest of `3B`. The columns 2 and 3 suggest that `3B` applied to the entire system seems fruitful on sparse systems. It is even generally better than $\texttt{IBB}_a$ (with `2B`). This suggests that a strong filtering process (i.e., `3B`) has about the same impact as a good splitting strategy (i.e., $\texttt{IBB}_a$).

The last three columns explain why we have chosen `2B` and not `3B` to filter inside blocks. The last column reports a new experiment in which `3B` is used only on the largest blocks. Indeed, due to combinatorial considerations, we believe that `3B` can seldom be efficient for handling small systems [283]. The first results are not very convincing, but experiments must be performed on more benchmarks.

Although not reported, we have also experimented $\texttt{IBB}_a$ and $\texttt{IBB}_b$ whose *IBF* is implemented with `3B`. This variant is counterproductive, but seems to be more robust, that is, `IBB` can more easily isolate solutions (when `I-Newton` is not effective). For instance, it does not require merging close atomic boxes of `Mechanism` to find exactly 448 solutions.

I.8 Conclusion

In this article, we have proposed new versions of `IBB` that use the new interval-based library `Ibex`. Discarding the old midpoint heuristic has rendered `IBB` reliable.

The main impact on robustness and performance is due to the combination of local filtering (e.g., `2B`) and interval analysis operators (e.g., interval Newton) inside blocks. Using `3B` instead of `2B` seems not promising except maybe for large blocks, as shown by first experiments.

Two other advanced features have shown their efficiency to limit choice points during search :

GCSP	IBB$_c$[BT+]	IBB$_c$[BT(+),IBF]	IBB$_c$[BT,IBF+]	IBB$_c$[BT+,IBF+]
	155	199	196	195
Mechanism	22680	22111	22111	22066
	1544	1629	1629	1156
	402	828	307	258
Sierp3	19454	5484	5484	4354
	12242	4272	4272	3455
	36	226	163	127
Chair	2368	6840	6825	4304
	97	344	344	148
	96	149	142	127
Tetra	9146	9842	9843	8568
	100	235	235	100
	97	126	123	118
PontsE	5986	5880	5880	5776
	79	115	115	67
	152	126	71	71
PontsG	7154	2669	2669	2669
	6585	1155	1155	1155
	1.9	5.5	4.2	4.2
Hourglass	48	59	59	59
	35	53	53	53
	15	32	21	21
TangentE	304	308	308	308
	423	427	427	427
	10	39	24	24
TangentG	102	816	816	816
	238	411	411	396
	18	32	20	6
Star	1534	1347	1347	192
	325	254	254	34

Tab. I.5 – Comparison between *IBF* and *IBF+* in IBB$_c$

inter-block filtering (*IBF*) and the recompute condition that avoids solving some blocks during search (BT+). We are now able to provide clear recommendations about these features.

– The best implementation of *IBF* (*IBF+*) is based on the computation of the subset of blocks that can actually be filtered when the current block is handled.
– It is not easy to know in advance which feature among *IBF+* and BT+ has the greatest impact on time complexity. In addition, using *IBF+* makes *BT+* less effective.
– Inter-block backtracking schemes that are more sophisticated than BT+, such as GBJ and GPB, have not proven their efficiency, especially thanks to the use of *IBF* that removes most of the potential backjumps.

Thus, we recommend two versions of IBB. First, IBB$_b$[BT,IBF] is a simple implementation directly available in Ibex [52, 50]. It is very simple, very fast (the overcost in CPU time related

GCSP	Ibex-2B	Ibex-3B	IBB$_a$[BT,IBF-2B]	IBB$_a$[BT,IBF-3B]	IBB$_c$[BT+]-2B	IBB$_c$[BT+]-3B	IBB$_c$[BT+]-3B>8
	>400000	>400000	11000	17300	155	389	156
Mechanism	–	–	22680	35007	2435	4236	23452
	–	–	–	–	1544	1544	1544
	>400000	3580	29660	3850	402	1028	–
Sierp3	–	453	2465	551	19454	12138	–
	–	–	–	–	12242	12242	–
	>400000	>400000	1080	4040	36	80	35
Chair	–	–	6601	329	2368	144	1458
	–	–	–	–	97	97	97
	75900	1820	4370	1310	96	123	80
Tetra	3409073	1337	66933	1999	9146	350	570
	–	–	–	–	100	100	100
	1070	720	706	354	97	117	117
PontsE	34457	843	23389	383	5986	320	334
	–	–	–	–	79	79	79
	623	317	224	306	152	400	–
PontsG	26099	407	4505	561	7154	5662	–
	–	–	–	–	6585	6585	–
	32	25	18	25	1.9	3.6	2.2
Hourglass	285	27	341	35	48	20	46
	–	–	–	–	35	35	35
	17800	960	5370	260	15	39	–
TangentE	568189	2457	40747	325	304	254	–
	–	–	–	–	423	423	–
	XXX	1460	XXX	380	10	18	–
TangentG	–	543	–	389	102	38	–
	–	–	–	–	238	238	–
	5240	1880	2400	1950	18	42	–
Star	28445	615	5243	691	1534	1126	–
	–	–	–	–	325	324	–

TAB. I.6 – Use of 3B. The columns 2 and 3 report the results obtained by Ibex on the entire system with resp. 2B filtering and 3B filtering. Columns 4 and 5 report the results obtained by IBB$_a$ with *IBF* performed resp. by 2B and 3B. For obtaining the last three columns, one investigated three different approaches for filtering inside blocks : with 2B, with 3B and with a mixed approach : 2B is used for blocks of size less than 9 and 3B is used for the largest blocks.

to the call to *IBF* is always less than 50% and sometimes pays off significantly). It works well when I-Newton inside blocks can isolate solutions. Second, IBB$_c$[BT+] (or IBB$_b$[BT,IBF+]) is a more sophisticated version that makes the approach more robust when I-Newton cannot certify or isolate solutions.

In addition to the dramatic gain in performance w.r.t. a global solving, IBB is simpler to be tuned. Indeed, only the parameter r_hc4 used by HC4 needs to be finely tuned.

Apart from minor improvements, IBB is now mature enough to be used in CAD applications. Promising research directions are the computation of sharper Jacobian matrices (because, in CAD, the constraints belong to a specific class) and the design of solving algorithms for equations with non punctual coefficients.

Annexe J

Filtering Numerical CSPs Using Well-Constrained Subsystems

Article [12] : paru dans les actes du congrès CP, Constraint Programming, en 2009

Auteurs : Ignacio Araya, Gilles Trombettoni, Bertrand Neveu

Abstract

When interval methods handle systems of equations over the reals, two main types of filtering/contraction algorithms are used to reduce the search space. When the system is well-constrained, interval Newton algorithms behave like a global constraint over the whole $n \times n$ system. Also, filtering algorithms issued from constraint programming perform an AC3-like propagation loop, where the constraints are iteratively handled one by one by a *revise* procedure. Applying a revise procedure amounts in contracting a 1×1 subsystem.

This paper investigates the possibility of defining contracting well-constrained subsystems of size k ($1 \leq k \leq n$). We theoretically define the *Box-k-consistency* as a generalization of the state-of-the-art Box-consistency. Well-constrained subsystems act as *global constraints* that can bring additional filtering w.r.t. interval Newton and 1×1 standard subsystems. Also, the filtering performed inside a subsystem allows the solving process to learn interesting multi-dimensional branching points, i.e., to bisect several variable domains simultaneously. Experiments highlight gains in CPU time w.r.t. state-of-the-art algorithms on decomposed and structured systems.

J.1 Introduction

When interval methods handle systems of equations over the reals, two main types of filtering/contraction algorithms are used to reduce the search space. When a system contains n unknowns/variables constrained by n equations, interval Newton algorithms behave like a global constraint over a linearization of the whole $n \times n$ system. Filtering algorithms issued from constraint programming handle 1×1 subsystems (one variable involved in one constraint) in an AC3-like propagation loop.

This paper investigates the possibility of filtering $k \times k$ subsystems, where the size $1 \leq k \leq n$. After introducing in Section J.2 the necessary background about intervals, we define in Section J.3 the Box-k-consistency achieved by our new algorithm. This partial consistency generalizes the well-known Box-consistency [28]. Due to the large amount of subsystems in a constraint system, we explain in Section 21 the criteria used to compute the Box-k-consistency in only certain subsystems that are made of equalities, connected and well-constrained. These subsystems are managed like *global constraints* [241, 183] for enhancing the filtering power. We detail in Section J.4 the filtering (revise) procedure that filters one subsystem and makes it box-k consistent. The procedure expands a local search tree whose choice points are limited inside the subsystem, and uses a local interval Newton. This revise procedure has common points with the algorithm proposed in [62]. Their algorithm also performs a tree search where every node is filtered, before returning an outer approximation of the obtained sub-boxes. But it is applied to the whole system of equations and not to subsystems.

Section J.5 details how the local search trees built inside subsystems allow a solving strategy to learn interesting multi-dimensional choice points in the global search tree, i.e., to bisect several variable domains simultaneously. These multi-dimensional branching points are called *multisplits*. Promising experiments highlight the benefits of our approach for *decomposed* and *structured* NCSPs.

J.2 Background

The algorithms presented in this paper aim at solving systems of equations or, more generally, numerical CSPs.

Definition 17 *A* **numerical CSP** *(NCSP) $P = (X, C, B)$ contains a set of constraints C and a set X of n variables. Every variable $x_i \in X$ can take a real value in the interval $[x_i]$ and B is the cartesian product (called a* **box***) $[x_1] \times ... \times [x_n]$. A solution of P is an assignment of the variables in X satisfying all the constraints in C.*

Since real numbers cannot be represented in computer architectures, note that the bounds of an interval $[x_i]$ should actually be defined as floating-point numbers. Most of the set operations can be achieved on boxes, such as inclusion and intersection. An operator `Hull` is often used to compute an outer approximation of the union of several boxes.

Definition 18 *Let $S = \{[b_1], ..., [b_n]\}$ be a set of boxes corresponding to a same n-set of variables. We call* **hull** *of S, denoted by* `Hull`*(S), the minimal box including $[b_1], [b_2], ..., [b_n]$.*

To find all the solutions of an NCSP with interval-based techniques, the solving process starts from an initial box representing the search space and builds a search tree. The tree search **bisects** the current box, that is, **splits** on one dimension (variable) the box into two sub-boxes, thus generating one choice point. At every node of the search tree, filtering (also called *contraction*) algorithms reduce the bounds of the current box. These algorithms comprise **interval Newton** algorithms issued from the numerical analysis community [147, 219] along with contraction algorithms issued from the constraint programming community. The process terminates with **atomic boxes** of size at most ϵ on every dimension.

The new contraction algorithm presented in this paper generalizes the famous `Box` algorithm that can enforce the *Box-consistency* property [28] defined as follows :

Definition 19 *An NCSP (X, C, B) is* **box-consistent** *if every pair (c, x) is box-consistent ($c \in C$, $x \in X$ and x is one of the variables involved in c).*

Consider a pair (c, x), where $c(x, y_1, ..., y_a) = 0$ is an equation of arity $a+1$.[1] *Let c' be the equation c where the variables y_i are replaced by the current interval in B : $c'(x) = c(x, [y_1], ..., [y_a]) = 0$. The pair (c, x) is box-consistent if :*

- $0 \in c'([\underline{[x]}, +]) = c([\underline{[x]}, +], [y_1], ..., [y_a])$;
- $0 \in c'([-, \overline{[x]}]) = c([-, \overline{[x]}], [y_1], ..., [y_a])$.

$\underline{[x]}$, resp. $\overline{[x]}$, denotes the lower bound, resp. the upper bound, of $[x]$. $[\underline{[x]}, +]$ denotes the tiny interval (of one u.l.p. large[2]*) bounded by $\underline{[x]}$ and the following float. $[-, \overline{[x]}]$ denotes the tiny interval bounded by $\overline{[x]}$ and the float preceding $\overline{[x]}$.*

In practice, the `Box` algorithm performs an AC3-like propagation loop. For every pair (c, x), it reduces the bounds of $[x]$ such that the new left (resp. right) bound is the leftmost (resp. rightmost) solution of the univariate equation $c'(x) = 0$. Existing *revise* procedures use a *shaving* principle to narrow $[x]$: Slices $[s_i]$ inside $[x]$ with no solution are discarded by checking whether $c([s_i], [y_1], ..., [y_a])$ does not contain 0 and by using a univariate interval Newton.

Two other contraction algorithms are often used in solvers. `HC4` [28] whose revise procedure traverses twice the tree representing the mathematical expression of the constraint for narrowing all the involved variable intervals. `3B` [186] or a variant `3BCID` [283] uses a shaving refutation principle similar to SAC [64].

[1] The definition of box-consistency can be straightforwardly extended to inequalities.

[2] One Unit in the Last Place is the gap between two very close floating-point numbers.

J.3 Box-k Partial Consistency

As explained above, the Box-consistency yields an outer approximation/box of 1×1 subsystems (c, x). The Box-k-consistency introduced in this paper generalizes Box-consistency by yielding an outer approximation of subsystems.

Definition 20 *Let $P' = (X', C', B')$ be a subsystem of a numerical CSP $P = (X, C, B)$ ($|X'| = k$), in which the (output) variables in X' are involved in at least one constraint in C' and the* **input variables** *(i.e., the variables involved in at least one constraint in C' which are not in X') are replaced by their current interval in B.*

The subsystem P' is **box-k-consistent** *if there exists a k-box of size 1 u.l.p. on every face of the k-box B' for which all the constraints c in C' are "satisfied", i.e., $0 \in c(X')$.*

If a box-k-consistent subsystem has an empty set of input variables, note that this subsystem is also global hull consistent [62]. Thus, like for the standard box-consistency, the presence of input variables makes the box-k-consistency weaker than global hull consistency.

Fig. J.1-left shows an example of a 2×2 subsystem. The outer box is box-consistent since it optimally approximates the solution set of constraints c_1 and c_2 individually. The inner box is box-2-consistent since it optimally approximates the set of six "thick" solutions to both constraints. Constraints are thick because the input variables (e.g., w_1, w_2, w_3) are replaced by intervals.

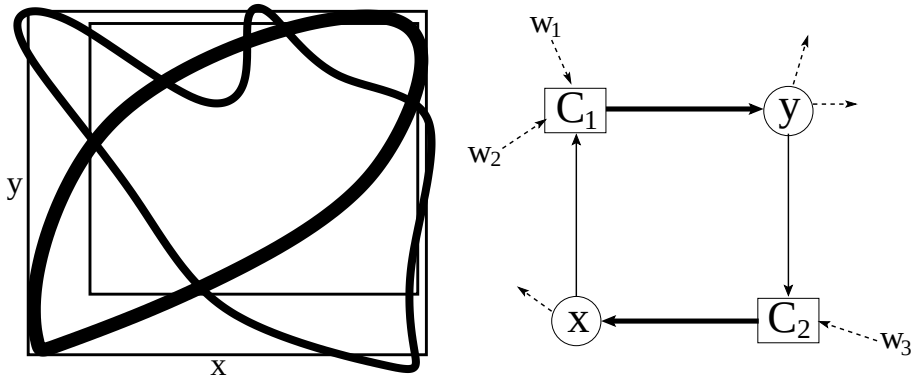


Fig. J.1 – Illustration of Box-2-consistency

Partial consistencies of NCSPs are generally defined modulo a precision ϵ that is used in practice by the corresponding algorithm to reach a fixpoint earlier. ϵ must then replace 1 u.l.p. in the previous definitions.

J.3.1 Benefits of Box-k-consistency

The following example theoretically shows that a contraction obtained by a $k \times k$ subsystem may be stronger than contraction on 1×1 subsystems and on the whole $n \times n$ system performed

by an interval Newton. Consider the NCSP $P = (\{x, y, z\},\ \{x - y = 0,\ x + y + z = 0,\ (z - 1)(z - 4)(2x + y + 2) = 0\},\ \{[-10^6, 10^6], [-10^6, 10^6][-10, 10]\})$.

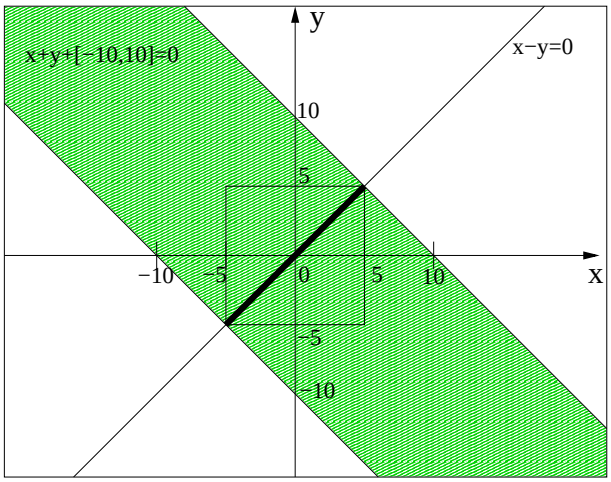


FIG. J.2 – Illustration of a subsystem of size 2, with $z = [-10, 10]$ as input variable. {[-5,5],[-5,5]} is box-2-consistent w.r.t. the 2 constraints $x - y = 0$ and $x + y + [z] = 0$.

Running `Box` and interval Newton on P does not filter the box. Achieving Box-2-consistency on the 2×2 subsystem ({x,y}, {x-y=0, x+y+z=0}) narrows the intervals of x and y to $[-5, 5]$ as shown in Fig. J.2. Also, if branching was used to find solutions, only two bisections (choice points) would be necessary to isolate the 3 solutions $\{(\frac{-2}{3}, \frac{-2}{3}, \frac{4}{3}), (-0.5, -0.5, 1), (-2, -2, 4)\}$. We should highlight that Newton on the whole system does not contract the box because it contains several solutions, whereas Newton on the 2×2 subsystem does because it contains only one (thick) solution (segment in bold). Of course, this small example is didactic. Experiments described in Section K.2.3 show larger and non-linear instances highlighting the benefits of structural partial consistencies over stronger partial consistencies like 3B-consistency [186].

J.3.2 Achieving Box-k-consistency in well-constrained subsystems of equations

Enforcing Box-k-consistency in every subsystem of given size k is too time-consuming and counter-productive in practice. The number of subsets of k variables in a NCSP with n variables is high and one needs to consider only promising subsystems.

We have thus used several criteria to reduce the number of subsystems that are candidate. We first select subsystems with only equations (no inequalities) because equations bring a great reduction of the search space and have nice properties. To understand these properties, we have to pay attention to NCSPs that admit a finite number of solutions. These NCSPs contains n variables but also the same number n of independent equations (additional inequalities can reduce the number of solutions). Also, the corresponding *bipartite constraint graph* verifies the following structural/graph property [3].

Definition 21 *Let P be a system of n independent equations constraining n variables. The vertices of the* **bipartite constraint graph** *G corresponding to P are the n variables and the n equations, and edges connect one equation to its involved variables.*

The system of equations P is **(structurally) well-constrained** *if its constraint graph G has a perfect matching [74].*

For instance, Fig. J.1-right shows the perfect matching (bold-faced edges) of the corresponding subgraph. This structural well-constriction can be viewed as a necessary condition to obtain a finite set of solutions. It appears that interval Newton also requires this condition (while it is of course not sufficient) for contracting a box. Indeed, if the system is not structurally well-constrained, the jacobian matrix will necessarily be singular [3]. Our subsystems fulfill this condition because Interval Newton is used by our new Box-k-Revise procedure (see Section J.4) to achieve faster a box-k-consistent subsystem. (Also, the time complexity of interval Newton is cubic in the number of variables, so that it is sometimes intractable to apply it to very large NCSPs. Instead, we could use interval Newton only inside subsystems.)

We finally require our subsystems be connected for performance considerations. Indeed, if a given subsystem of size k contained several disconnected components of size at most k' ($k' < k$), we could make it box-k-consistent by achieving box-k'-consistency in every component.

To sum up, restricting the subsystems to well-constrained and connected subgraphs of equations has two virtues. First, it allows a strong filtering in specific subparts of the system, which is useful for sparse NCSPs or for (globally) under-constrained ones, e.g., systems mixing equalities and inequalities. Second, it allows the use of an interval Newton to faster contract the subsystem.

J.4 Contraction Algorithm Using Well-constrained Subsystems as Global Constraints

Instead of contracting all the well-constrained subsystems of given size k, we have designed an AC3-like propagation that manages selected subsystems of different sizes : subsystems of size 1 but also well-constrained subsystems of larger size. Well-constrained subsystems are thus similar to *global constraints* [241, 183] that can be defined by the user or automatically (see Section K.2.3).

All the subsystems are first put into a propagation queue and revised in sequence. When a variable domain is reduced more than a ratio ρ_{propag}, all the subsystems involving this variable are pushed into the queue, if they are not already in it. This propagation process is just specialized by the revise procedure used for contracting the subsystems of size greater than 1 and detailed below.

J.4.1 The `Box-k` revise procedure

The revise procedure is based on a branch & prune method limiting the bisection to the k (output) variables X of the subsystem, and using a *breadth-first* search. At the end of this local

tree search, the current box is replaced by the hull of the leaves of the local tree. The algorithm `Box-k-Revise` is a generic procedure that achieves a box-k-consistent subsystem. The procedure manages a list L of nodes that are leaves of the local tree. A leaf l in L has three significant components : *l.box* designs the (n-dimensional) search space associated to the node ; *l.precise* is a boolean stating whether *l.box* has reached the precision ϵ in all the dimensions (ϵ also yields the precision of the global solution) ; *l.certified* is a boolean asserting whether *l.box* contains a unique solution. The `box` parameter is the current global box (search space) when the revise procedure is called.

Algorithm 11 `Boxk-Revise` (**in-out** L, `box` ; **in** X, C, ϵ, `subContractor`, τ_{leaves}, $\tau_{\rho_{io}}$)

```
UpdateLocalTree(L, box, X, C, ε, subContractor)
L' ← {l ∈ L s.t. ¬l.certified and ¬l.precise and ProcessLeaf ?(l,X,C,τ_ρio)}
while 0 < L'.size and L.size < τ_leaves do
  l ← L'.front() /* Select a leaf in breadth-first order */
  (l1, l2) ←bisect(l, X)
  contract(l1, subContractor, X, C, ε)
  contract(l2, subContractor, X, C, ε)
  if l1.box ≠ ∅ then L.pushBack(l1) end if
  if l2.box ≠ ∅ then L.pushBack(l2) end if
  L.remove(l)
  L' ← {l ∈ L s.t. ¬l.certified and ¬l.precise and ProcessLeaf ?(l,X,C,τ_ρio)}
end while
box ←hull(L) /* Outer approximation of the union of all the boxes l.box, l ∈ L */
```

A combinatorial process (tree search) is performed by the `while` loop. At every iteration, one leaf in L, which is not *precise* and not *certified*, is selected, bisected and the two new sub-boxes are contracted. The search ends if all the leaves are tagged as *certified* or *precise* or if a limit τ_{leaves} in the number of leaves is reached. τ_{leaves} limits the memory storage requirement (see Section J.4.5) and allows one to quickly propagate the obtained reductions to the other subsystems.

A leaf is simply selected in breadth-first order. We first tried a more sophisticated heuristic function for selecting a "large" box on the border of the hull of the different leaves. The idea was to maximize the gain in volume on the current global box in case the selected leaf would be eliminated by filtering. This multi-dimensional generalization of the `BoxNarrow` algorithm (that shaves the bounds of the handled interval in the `Box` algorithm) has been discarded because it did not bring a significant gain in performance.

Algorithm 12 `contract`(**in-out** l ; **in** `subContractor`, X, C, ϵ)

```
if ¬l.precise then
  if ¬l.certified then subContractor(l.box) end if
  if l.box ≠ ∅ and I-Newton(l.box,X) then l.certified ← true end if
  if maxDiameter(l.box) < ε then l.precise ← true end if
end if
```

The procedure `contract` is mainly parameterized by the contraction procedure `subContractor` (`HC4` [28] or `3BCID` [283] in our experiments). The scope C of `subContractor` is the considered k-set of equations. After a call to `subContractor`, an interval Newton limited to the $k \times k$ subsystem is launched. If Newton certifies a unique solution in a leaf, `I-Newton` contracts $l.box$ and returns *true* so that this leaf is tagged as *certified*.

J.4.2 The `S-kB-Revise` variant

`S-kB-Revise` is the name of a variant of `Box-k-Revise` for which the entire system is used in the `contract` procedure. That is, the scope C of `subContractor` includes the whole n-set of constraints, instead of the k-set of constraints attached to the subsystem. With `S-kB-Revise`, the k-set of constraints in the subsystem is just used by interval Newton. This variant brings additional filtering, but at a higher cost.

J.4.3 Reuse of the local tree (procedure `UpdateLocalTree`)

A simpler version of Algorithm 1 did not call the `UpdateLocalTree` procedure and simply initialized the list L with the current box. However, instead of performing an intensive search effort in only one subsystem, we preferred to quickly propagate the obtained reductions to the other subsystems. Therefore the `UpdateLocalTree` procedure reuses the local tree (i.e., its leaves) that has been saved in a previous call to Algorithm 1. Every leaf in the current list L is just updated by intersection with the current box and filtered with `subContractor`.

Algorithm 13 `UpdateLocalTree` (**in-out** L ; **in** `box`, X, C, ϵ, `subContractor`)

```
if L = ∅ then
  L ← {Leaf(box)} /* Initialize the root of the local tree with the current box */
else
  for all l ∈ L do
    /* Update and contract every leaf of the stored local tree */
    if l.box ≠ (l.box ∩ box) then
      l.box ← l.box ∩ box
      contract(l, subContractor, C, ϵ))
      if l.box = ∅ then L.remove(l) end if
    end if
  end for
end if
```

In fact, the leaves of the local trees are also maintained in the global search tree. To do so, the list L is implemented as a *backtrackable* data-structure updated in case of backtracking. It avoids redoing the same job in the subsystems several times, in particular when the *multisplit* splitting heuristic is chosen (see Section J.5).

J.4.4 Lazy handling of a leaf (procedure `ProcessLeaf ?`)

Our first experiments have shown us that handling a leaf in a local tree, i.e., bisecting it and contracting the two sub-boxes, was often counterproductive. We have then defined an input/output ratio ρ_{io} that decides whether a given leaf of box B must be handled in the local tree.

$$\rho_{io}(B, I, O, F) = \frac{Max_{x \in I}(smear(x))}{Max_{x \in O}(smear(x))}$$

The function `ProcessLeaf ?` calculates ρ_{io} in a leaf. If this ratio is larger than a threshold $\tau_{\rho_{io}}$, the leaf will not be handled in the current revise procedure.

ρ_{io} is based on the well-known *smear* function [167] defined by :
smear(x) $:= Max_{f \in F}(|\frac{\partial f}{\partial x}| \times Diam(x))$. This function is often used for selecting the next variable to be bisected in NCSPs (the variable with the largest smear evaluation).

The denominator of ρ_{io} can be directly explained by it : output variables (O) with a great smear evaluation (implying a small ratio ρ_{io}) often lead to a great contraction when they are bisected inside the local subsystem tree. Desiring a small impact of the input variables (I) is less intuitive. We understand that large input domains generally lead to large output domains (i.e., leaf boxes) in the subsystem and thus yields a poor reduction. The same argument holds in fact for the derivatives of functions. To illustrate this point, let us take a subsystem of size 1 like $0.001\,y + x^2 - 1 = 0$ (x is the output variable ; $[x] = [y] = [-1, 1]$) having $\rho_{io} = \frac{0.002}{4} = 0.0005$. After one bisection on x, the subsystem contraction leads to a very small interval for x. A large interval would be obtained for x if the considered subsystem was $y + x^2 - 1 = 0$ with $\rho_{io} = \frac{2}{4} = 0.5$.

J.4.5 Properties of the revise procedure

The following proposition formalizes the correctness, the memory and time complexities of the procedure `Box-k-Revise`.

Proposition 13 *Let $P' = (X', C', B)$ be a subsystem of a CSP $P = (X, C, B)$, with $|X| = n$, $|C| = m$, $|X'| = |C'| = k$.*

The procedure `Box-k-Revise`*, called with $\tau_{leaves} = +\infty$ and $\tau_{\rho_{io}} = +\infty$, makes P' box-k-consistent.*

Let Diam be the largest interval diameter in B. Let d be $log_2(\frac{Diam}{\epsilon})$, the maximum number of times a given interval must be bisected to reach the precision ϵ. [3]

The memory complexity of `Box-k-Revise` *is $O(k\,\tau_{leaves})$.*

The number of calls to `subContractor` *is $O(k\,d\,\tau_{leaves})$.*

[3] d generally falls between 20 and 60 in NCSPs occurring in practice.

Proof. The correction is based on the combinatorial process performed by the procedure `Box-k-Revise`. Called with $\tau_{leaves} = +\infty$ and with the subsystem made of C', the procedure computes all the atomic boxes of precision ϵ in the subsystem before returning the hull of them, thus achieving roughly (i.e., assuming that the actual values of input variables are unknown) the global consistency of P'.

The memory complexity comes from the breadth-first search that must store the $O(\tau_{leaves})$ leaves of the local tree. The revise procedure works with n-dimensional boxes but, in order to save memory, stores at the end only k intervals of a $k \times k$ subsystem.

The number of calls to `subContractor` is bounded by the number of nodes in the local search tree. The number of leaves of this tree is τ_{leaves} (corresponding to *living* boxes that can contain solutions) plus the number of *dead* leaves eliminated by filtering. For any living leaf l, the number of nodes created in the tree to reach l is at most $2 \times d \times k$ since the root must be at most bisected d times in all its k dimensions. Although numerous such internal nodes are "shared" by several living leaves, this bounds the number of calls to a sub-filtering operator with $O(k\, d\, \tau_{leaves})$. □

Another property allows us to better understand the gain in contraction obtained by the `S-kB-Revise` variant (see Section J.4.2).

Proposition 14 *Consider a propagation algorithm calling* `S-kB-Revise` *on all the subsystems of size k in a given NCSP P.*
This algorithm computes the $(k+2)B$-consistency of P.

The kB-consistency, introduced by Lhomme [186], is a strong partial consistency related to the k-consistency (in finite-domain CSPs) restricted to the bounds of intervals. $3B$-consistency is similar to SAC-consistency [64]. It is known to be stronger (i.e., to better contract) than box-consistency (i.e., box-1-consistency). It appears that this result can be generalized to any $k > 1$.

J.5 Multidimensional Splitting

It turns out that the `Box-k-Revise` procedure has not only a contraction effect, but also provides a new way to make choice points, that is, to build the (global) search tree. This new splitting strategy is called *multidimensional splitting* (in short *multisplit*).

Definition 22 *Consider a $k \times k$ subsystem P' defined inside an NCSP $P = (X, C, B)$. Consider a set S of m boxes associated to P' such that S contains all the solutions to P, and the m boxes obtained by projection on P' of the boxes in S are pairwise disjoint.*

A **multisplit** *of dimension k consists in splitting the search space B into the m boxes in S.*

In practice, the m boxes correspond to the leaves of a subsystem local tree. At the end of a `Box-k` propagation, our solving strategy makes a choice between a classical bisection and a multisplit. If all the subsystems have a ratio ρ_m larger than a user-defined threshold τ_m, then a standard

bisection is performed. Otherwise, we multisplit the subsystem with the smallest ratio ρ_m, i.e., we replace the current box by the set L of m leaves associated to the local tree.

$$\rho_m = \frac{\sum_{l \in L} Volume(l)}{Volume(Hull(L))}$$

Multisplit generalizes a procedure used by `IBB` (see Section J.6.1). `IBB` performs a multisplit once it finds the m solutions (i.e., atomic boxes) in a given block. The difference here is that a multisplit may occur with *non* atomic boxes whose size has not reached the required precision.

J.6 Experiments

The `Box-k` based propagation algorithm has been implemented in the `Ibex` open source interval-based solver in `C++` [52, 50]. The variant with multisplit (`msplit`) performs a multisplit of a subsystem with the minimum ratio ρ_m, provided that $\rho_m < \tau_m$=0.99. All the competitors are also available in the same library, making the comparison fair.

J.6.1 Experiments on decomposed benchmarks

Ten *decomposed* benchmarks, described in [222, 221], appear in Table J.1. They have been previously decomposed by equational algorithms (*eq*) like maximum-matching, or by more sophisticated geometrical algorithms (*geo*). They are challenging for general-purpose interval methods, but can efficiently be solved by `IBB` [222, 221].

Brief description of `IBB`

`IBB` is dedicated to decomposed systems, i.e., sparse systems of equations that have been first decomposed into a sequence of *irreducible* [3] well-constrained blocks/subsystems. Inter-Block Backtracking handles every block in the order provided by the sequence. It interleaves contraction steps (performed by `HC4` and interval Newton) and bisections inside the block until atomic boxes (solutions) are obtained. Choice points are then made : the variables of the block are replaced by one of the atomic boxes, i.e., they are considered constant in subsequent blocks.

We understand that the `Box-k-Revise` procedure plus multisplit represents a generalization of `IBB` in that the input variables domains of a subsystem are not necessarily atomic and that a multisplit is not necessarily performed after a subsystem handling. In other terms, the `IBB` block handling is not a revise procedure, it is just an ad-hoc procedure embedded in a dedicated algorithm. Applied to decomposed systems, the only information that our new approach does not exploit is the order between blocks which provides to `IBB` a useful splitting heuristic.

TAB. J.1 – Experimental results on IBB benchmarks. The first 3 columns include the name of the system, its number n of variables and its number of solutions. The next three columns yield the CPU time (above) and the number of boxes, i.e., choice points (below), obtained on an Intel 6600 2.4 GHz by existing strategies based on HC4, Box or 3BCID followed by interval Newton (between two bisections selected in a round-robin way for the variable selection). The last four columns report the results obtained by our algorithms on the same computer : Box-k-Revise parameterized by subContractor=HC4 or subContractor=3BCID, with multisplit (msplit) or without. To be the closest to IBB, Box-k-Revise, and not the S-kB-Revise variant, is used by our constraint propagation algorithm.

Benchmark	n	#sols	**HC4**	**Box**	**3BCID**	**IBB**	**Box-k(HC4)**		**Box-k(3BCID)**	
								msplit		**msplit**
Chair(eq)	178	8	>3600	>3600	>3600	**0.27**	>3600	16.5	>3600	**0.52**
1x15,1x13,1x9,5x8,3x6,...								575		15
Latham(eq)	102	96	>3600	>3600	39.9	**0.17**	**0.94**	1.35	1.5	1.08
1x13,1x10,1x4,25x2,25x1					587		839	199	991	189
Ponts(eq)	30	128	33.4	33.4	1.89	**0.59**	6.85	8.19	0.79	**0.71**
1x14,6x2,4x1			20399	20399	357		783	231	307	231
Ponts(geo)	38	128	44.1	44.1	2.6	**0.16**	2.01	0.31	1.45	**0.39**
13x2,12x1			18363	18363	685		6711	767	6711	767
Sierp3(geo)	124	198	>3600	>3600	77.5	**0.62**	49.0	**1.38**	52.5	1.77
44x2,36x1					1727		84169	1513	84169	1513
Star(eq)	46	128	>3600	>3600	4.9	**0.05**	35.6	**0.12**	44.0	0.26
3x6,3x4,8x2					283		44195	263	44023	263
Tangent(eq)	28	128	77	77	2.1	**0.08**	1.74	**0.08**	1.87	0.14
1x4,10x2,4x1			390903	390903	753		12027	255	12235	255
Tangent(geo)	42	128	–	–	7.38	**0.08**	0.80	0.19	0.80	**0.19**
2x4,11x2,12x1					859		1415	251	1407	251
Tetra(eq)	30	256	1281	1281	12.3	**0.63**	33.6	1.06	13.57	**0.76**
1x9,4x3,1x2,7x1			607389	607389	1713		4619	483	2243	483
Sierp3(eq)	see Section J.6.2					>5000	see Section J.6.2			

Experimental protocol

Every Box-k based strategy has been tuned with 6 different sets of parameter values : $\tau_{\rho_{io}}$ is 0.01, 0.2 or 0.8 (0.01 is always the best value on decomposed systems) ; the precision ρ_{propag} used in the HC4 propagation is 1% or 10% ; All the other parameters have been empirically fixed : the precision ρ_{propag} in the Box-k propagation is always 10% ; the maximum number τ_{leaves} of leaves inside a subsystem tree is 10 ; the number of slices of 3BCID in Box-k(3BCID) is 10. To be fair, the parameters of the competitor algorithms have been tuned so that 8 trials have been performed for Box and HC4, and 16 trials have been run for 3BCID. For all the tests, the Newton ceil (size of maximum diameter under which interval Newton is run) is 10, and the same variable order is used in a round-robin strategy (except for IBB and for Box-k with multisplit).

The subsystems given to our Box-k propagation are defined automatically. The irreducible blocks

produced by the IBB decomposition simply become the well-constrained subsystems handled by Box-k-Revise.

Results

Strategies based on HC4, Box and 3BCID followed by interval Newton are not competitive at all with Box-k and IBB on the tested decomposed systems. The comparison of Box-k against IBB is very positive because the CPU times reported for IBB are really the best that have never been obtained with any variant of this dedicated algorithm. Also, no timeout is reached by Box-k+multisplit and IBB is on average only twice faster than Box-k(3BCID) (at most 6 on Latham). As expected, the results confirm that multisplit is always relevant for decomposed benchmarks. For the benchmark Sierp3(eq) (the fractal Sierpinski at level 3 handled by an equational decomposition), an equational decomposition makes appear a large irreducible 50×50 block of distance constraints. This renders IBB unefficient on it (timeout).

J.6.2 Experiments on structured systems

Tab. J.2 – Results on structured benchmarks. The same protocol as above has been followed, except that the solving strategy is more sophisticated. Between two bisections, the propagation with subsystems follows a 3BCID contraction and an interval Newton. The four Box-k columns report the results obtained by the S-kB-Revise variant. The results obtained by Box-k-Revise are generally worse and appear, with multisplit only, in the last two columns.

Benchmark	n	#sols	**HC4**	**Box**	**3BCID**	**Box-k(HC4)**		**Box-k(3BCID)**		**Box-k-Revise**	
							msplit		**msplit**	HC4	3BCID
Bratu	60	2	58	626	48.7	47.0	**33.0**	135	126	86.4	96.2
29x3			15653	13707	79	39	17	43	25	125	129
Brent	10	1015	1383	127	**17.0**	28.5	20.2	44.9	31.0	20.8	34.9
2x5			7285095	42191	9849	2975	4444	4585	1309	5215	4969
BroydenBand	20	1	>3600	0.17	**0.11**	0.45	0.15	0.91	0.31	0.30	0.28
1x6,3x5				1	21	4	19	17	3	7	3
BroydenTri	30	2	1765	**0.16**	0.25	0.22	0.24	0.39	0.29	0.19	0.23
6x5			42860473	63	25	11	19	9	3	19	17
Reactors	30	120	>3600	>3600	288	340	315	81.4	**67.5**	250	194
3x10					39253	14576	10247	1038	788	35867	21465
Reactors2	10	24	>3600	>3600	28.8	**9.5**	12.3	10.4	12.2	9.93	11.9
2x5					128359	4908	10850	4344	5802	5597	5353
Sierp3Bis(eq)	83	6	>3600	>3600	4917	>3600	>3600	>3600	**389**	>3600	4503
1x14,6x6,15x2,3x1					44803				218		122409
Trigexp1	30	1	>3600	13	**0.08**	**0.08**	**0.08**	**0.08**	0.09	0.08	0.08
6x5				27	1	1	1	1	1	1	1
Trigexp2	11	0	1554	>3600	83.7	81.2	85.7	105	83.0	**80.6**	82.1
2x4,2x3			2116259		16687	15771	16755	3797	2379	15771	11795

Nine *structured* systems appear in Table J.2. They are scalable chains of constraints of reasonable arity [204]. They are denoted *structured* because they are not sufficiently sparse to be decomposed

by an equational decomposition, i.e., the system contains only one irreducible block, thus making IBB pointless. A brief and manual analysis of the constraint graph of every benchmark has led us to define a few well-constrained subsystems of reasonable size (between 2 and 10). In the same way, we have replaced the 50×50 block in Sierp3(eq) by 6×6 and 2×2 Box-k subsystems.

Standard strategies based on HC4 or Box followed by interval Newton are generally not competitive with Bok-k on the tested benchmarks. The solving strategy based on S-kB-Revise with subContractor=3BCID (column Box-k(3BCID)) appears to be a robust hybrid algorithm that is never far behind 3BCID and is sometimes clearly better. The gain w.r.t. 3BCID falls indeed between 0.7 and 12. The small number of boxes highlights the additional filtering power brought by well-constrained subsystems. Again, multisplit is often the best option.

The success of Box-k on Sierp3Bis(eq) has led us to try a particular version of IBB in which the inter-block filtering [222] is performed by 3BCID. Although this variant seldom shows a good performance, it can solve Sierp3(eq) in 330 seconds.

J.6.3 Benefits of sophisticated features

Tables J.3 has finally been added to show the individual benefits brought by two features : the user parameter $\tau_{\rho_{io}}$ driving the procedure ProcessLeaf ? and the backtrackable list of leaves used to reuse the job achieved inside the subsystems.

TAB. J.3 – Benefits of the backtrackable data structure (BT) and of $\tau_{\rho_{io}}$ in the Box-k-based strategy. Setting $\tau_{\rho_{io}} = \infty$ means that subsystem leaves will be always processed in the revise procedure.

	Chair	**Latham**	**Ponts(eq)**	**Ponts(geo)**	**Sierp3(geo)**	**Star**	**Tan(eq)**	**Tan(geo)**	**Tetra**
BT, $\tau_{\rho_{io}}$	0.52	1.08	0.71	0.31	1.38	0.12	0.08	0.19	0.76
¬ BT, $\tau_{\rho_{io}}$	10.8	4.61	1.51	1.27	23.9	2.34	0.71	1.58	2.13
BT, $\tau_{\rho_{io}} = \infty$	23.4	4.71	2.60	1.00	23.8	1.67	1.09	1.81	3.57
¬ BT, $\tau_{\rho_{io}} = \infty$	24.2	6.60	2.80	1.11	23.9	2.40	1.15	1.82	3.54
	Bratu	**Brent**	**BroyB.**	**BroyT.**	**Sierp3B(eq)**	**Reac.**	**Reac.2**	**Trigexp1**	**Trigexp2**
BT, $\tau_{\rho_{io}}$	33.0	20.2	0.15	0.24	389	67	12.2	0.08	83
¬ BT, $\tau_{\rho_{io}}$	33.2	21.0	0.14	0.23	411	97	12.0	0.07	85
BT, $\tau_{\rho_{io}} = \infty$	33.9	23.8	0.38	0.28	519	164	13.1	0.10	103
¬ BT, $\tau_{\rho_{io}} = \infty$	33.0	28.7	0.40	0.38	533	401	18.7	0.07	148

Every cell reports the best result (CPU time in second) among both sub-contractors. Multisplit is allowed in all the tests. The first line of results corresponds to the implemented and sophisticated revise procedure ; the next ones correspond to simpler versions for which at least one of the two advanced features has been removed.

Three main observations can be drawn. First, when a significant gain is brought by the features on a given system, then this system is efficiently handled against competitors in Tables J.1 and J.2. Second, $\tau_{\rho_{io}}$ seems to have a better impact on performance than the backtrackable list, but the difference is slight. Third, several systems are only slightly improved by one of both features,

whereas the gain is significant when both are added together. This is true for most of the IBB benchmarks. On these systems, between 2 bisections in the search tree, it often occurs that a job inside *several* subsystems leads to identify atomic boxes (some others are not fully explored thanks to $\tau_{\rho_{io}}$). Although we multisplit only one of these subsystems, the job on the others is saved in the backtrackable list.

J.7 Conclusion

We have proposed a new type of filtering algorithms handling $k \times k$ well-constrained subsystems in an NCSP. $k \times k$ interval Newton calls and selected bisections inside such subsystems are useful to better contract decomposed and structured NCSPs. In addition, the local trees built inside subsystems allow a solving strategy to learn choice points bisecting several variable domains simultaneously.

Solving strategies based on Box-k propagations and multisplit have mainly three parameters : the choice between Box-k-Revise and S-kB-Revise (although Box-k-Revise seems better suited only for decomposed systems), the choice of sub-contractor (although 3BCID seems to be often a good choice), and $\tau\rho_{io}$. This last parameter appears to be finally the most important one.

On decomposed and structured systems, our first experiments suggest that our new solving strategies are more efficient than standard general-purpose strategies based on HC4, Box or 3BCID (with interval Newton). Box-k+multisplit can be viewed as a generalization of IBB. It can also solve large decomposed NCSPs with relatively small blocks in less than one second, but can also handle structured NCSPs that IBB cannot treat.

Subsystems have been automatically added in the decomposed systems, but have been manually added in the structured ones, as global constraints. In this paper, we have validated the fact that handling subsystems could bring additional contraction and relevant multi-dimensional choice points. The next step is to automatically select a relevant set of subsystems. We believe that an adaptation of maximum-matching machinery or other graph-based algorithms along with a criterion similar to ρ_{io} could lead to efficient heuristics.

Acknowledgments

We thank the referees for their helpful comments.

Annexe K

IDWalk : A Candidate List Strategy with a Simple Diversification Device

Article [231] : paru dans les actes du congrès CP, Constraint programming, en 2004

Auteurs : Bertrand Neveu, Gilles Trombettoni, Fred Glover

Abstract

This paper presents a new optimization metaheuristic called IDWalk (Intensification/Diversification Walk) that offers advantages for combining simplicity with effectiveness. In addition to the number S of moves, IDWalk uses only one parameter Max which is the maximum number of candidate neighbors studied in every move. This candidate list strategy manages the Max candidates so as to obtain a good tradeoff between intensification and diversification.

A procedure has also been designed to tune the parameters automatically. We made experiments on several hard combinatorial optimization problems, and IDWalk compares favorably with correspondingly simple instances of leading metaheuristics, notably tabu search, simulated annealing and Metropolis. Thus, among algorithmic variants that are designed to be easy to program and implement, IDWalk has the potential to become an interesting alternative to such recognized approaches.

Our automatic tuning tool has also allowed us to compare several variants of IDWalk and tabu search to analyze which devices (parameters) have the greatest impact on the computation time. A surprising result shows that the specific diversification mechanism embedded in IDWalk is very significant, which motivates examination of additional instances in this new class of "dynamic" candidate list strategies.

K.1 Introduction

Local search is widely used in combinatorial optimization because it often yields a good solution in reasonable time. Among the huge number of metaheuristics that have been designed during the last decades, only a few can obtain a good performance on most problems while managing a small number of parameters.

The goal of our work was to obtain a new computationally effective metaheuristic by performing a study of the most intrinsic phase of the search process, the phase that examines a list of candidates (neighbors) for the next move. This study has led us to design a new, simple and very promising *candidate list strategy (CLS)* to provide a metaheuristic that implements local search devices in the neighborhood exploration phase.

Several CLS procedures have been designed in the past, particularly in connection with tabu search [99]. The IDWalk (Intensification/Diversification Walk) metaheuristic presented in this paper can be viewed as an extension of the AspirationPlus CLS approach [99] that is endowed with a simple and efficient diversification mechanism, called SpareNeighbor below, to exit from local minima.

Roughly, IDWalk performs S moves and returns the best solution found during the walk. Every time IDWalk selects a move, it examines at most Max candidate neighbors by selecting them randomly one by one. If the cost of a neighbor x' is *less than or equal to* the cost of the current solution x, then x' is chosen for the next move (rudimentary intensification effort). If no neighbor has been accepted among the Max examined, then one of these candidates, with a cost worse than the one of x, is chosen for the next move (rudimentary diversification device). Two variants perform this simple diversification process by setting a specific value to a parameter called SpareNeighbor. In the first variant ID(any), where SpareNeighbor is set to any, *any* previously rejected candidate is randomly selected (among the Max visited neighbors). In the second variant ID(best), where SpareNeighbor is set to best, a *best* (or rather less worsening, in terms of cost) previously rejected candidate is selected.

The first part of the paper introduces the IDWalk candidate list strategy. Section K.2 gives a detailed description of the two variants of IDWalk. Performed on a large sample of benchmarks, IDWalk compares very favorably with correspondingly simple instances of leading metaheuristics, notably tabu search, simulated annealing [169] and Metropolis [60].

The second part of this paper tries to understand the role of key intensification and diversification parameters in the optimization process. Section K.3 uses tabu search, several variants of IDWalk, and our automatic tuning tool to learn more about the impact of parameters on the computation time. Two CLS devices are studied along with the tabu list. This first analysis performed on numerous instances from different problem classes reveals that the SpareNeighbor diversification device used by IDWalk and tabu search has generally a crucial impact on performance.

K.2 Description of ID Walk and comparison with leading metaheuristics

This section describes the two main variants of ID Walk, introduces a straightforward tool used to tune automatically easy to program metaheuristics and reports the experimental results performed on a large sample of problems.

K.2.1 Description of ID Walk

Without loss of generality, the following pseudo-code description assumes that ID Walk solves a combinatorial *minimization* problem.

Algorithm ID Walk *(S : number of moves,* Max *: number of neighbors,* SpareNeighbor *: type of diversification)*
  Start with a configuration x
  BestConfiguration $\leftarrow x$
  for i **from** *1* **to** S **do**
    Candidate $\leftarrow 1$
    RejectedCandidates $\leftarrow \emptyset$
    Accepted ? $\leftarrow$ *false*
    while *(*Candidate $\leq$ Max*)* **and** *(***not** Accepted ?*)* **do**
      $x' \leftarrow$ Select (Neighborhood(x), any)
      if cost*(x')* $\leq$ cost*(x)* **then**
        Accepted ? $\leftarrow$ *true*
      else
        RejectedCandidates $\leftarrow$ RejectedCandidates $\cup \{x'\}$
      end
      Candidate $\leftarrow$ Candidate $+1$
    end
    if Accepted ? **then**
      $x \leftarrow x'$
    else
      $x \leftarrow$ Select (RejectedCandidates, SpareNeighbor)
    end
    if cost*(x)* $<$ cost(BestConfiguration) **then** BestConfiguration $\leftarrow x$
  end.
  return BestConfiguration
end.

The move selection is the main contribution of ID Walk and Max is a simple parameter for imposing a ratio between intensification and diversification efforts :

– First, the parameter is often useful to limit the number of neighbors visited in problems with large neighborhoods, to avoid an exhaustive search.

– Second, Max must be sufficiently large to allow the search to pursue better solutions in an aggressive way (intensification).
– Third, Max must be sufficiently small to allow the search to exit from local minima (diversification).

We have designed two variants of ID Walk that embedd two different ways for exiting from local minima, and thus two degrees of diversification. These variants differ only on the way a candidate is chosen when none of them has been accepted (in the while loop), that is, they differ on the SpareNeighbor parameter.

The variant ID(any)

ID(any) (Intensification/Diversification Walk with *Any* "spare" neighbor) corresponds to the algorithm ID Walk called with SpareNeighbor equal to any. In this case, the Select function chooses **any** neighbor among the Max previously rejected candidates. This neighbor is randomly selected.

The variant ID(best)

ID(best) (Intensification/Diversification Walk with *Best* "spare" neighbor) corresponds to the algorithm ID Walk called with SpareNeighbor equal to best. In this case, the Select function chooses a **best** neighbor (i.e., with a lowest cost for the objective function) among the Max rejected candidates.

Note that a variant of tabu search also uses a parameter SpareNeighbor set to best. The behavior of the TS used in this paper is similar to the one of ID(best) in case all the studied candidates have not been accepted because they are all tabu and do not meet the aspiration criterion : instead of getting stuck, TS and ID(best) move to the best neighbor, in terms of cost. (More common variants of TS select a neighbor that has least recently or least frequently been selected in the past, breaking ties by reference to cost.)

K.2.2 Automatic parameter tuning procedure

We have implemented a straightforward procedure for tuning the two parameters of ID Walk. In accordance with experimental observations, we have assumed, somewhat naively, that for a given walk length S, there exists one value for Max that maximizes the performance, i.e., that gives the best average cost of the solution. We suspected however that the best value of Max depends on S, so that the implemented procedure for tuning Max is called every time the number of moves is increased. The principle of the automatic tuning procedure is the following :

1. $S \leftarrow S_0$ (the walk length S is initialized)
2. Until a maximum running time is exceeded :
   (a) Tune Max by running the algorithm ID Walk on reduced walk lengths S/K. Let N_i be the value found for the parameter.

(b) Run the algorithm ID Walk(S, N_i,SpareNeighbor).

(c) $S \leftarrow F \times S$

In the experiments presented below, $S_0 = 10^6$ moves, $K = 50$, and we have chosen an increasing factor $F = 4$. Note that we restart from scratch (i.e., from a new configuration) when moving from S_j to S_{j+1}. Only the lattest value of Max is reused.

Thus, every phase i, performed with a given walk length S, includes a step (a) tuning Max and a solving step (b) keeping Max constant. Runs in steps (a) and (b) are performed with a given number of trials (e.g., 10 trials). In the tuning step (a), $P = 10$ different parameter values are tried for N_i in a dichotomous way. The number of moves of our tuning procedure is then : $\sum_{i=1}^{maxiter} S_0(1 + P/K)F^i$

The tuning step (a) is perfomed as follows. Starting from an initial value for Max (depending on the metaheuristic), Max is divided by 2 or multiplied by 2 until a minimum is reached, in terms of cost. The value of Max is then refined in a dichotomous way.

Our automatic tuning procedure is also applied to other algorithms with one parameter such as Metropolis and simulated annealing with a linear temperature decrease. In this case, the (initial) temperature replaces the parameter Max in the above description.

This tuning procedure has also been extended to tune algorithms with two parameters (in addition to the number S of moves), such as the tabu search and more sophisticated variants of ID Walk that will be introduced in Section K.3.

K.2.3 Experiments and problems solved

We have performed experiments on 21 instances issued from 5 categories of problems, generally encoded as weighted MAX-CSPs problems with two different neighborhoods, which yields in fact 35 instances. Graph coloring instances are proposed in the DIMACS challenge [214]. We have also tested CELAR frequency assignment problems [76][1], a combinatorial game, called Spatially-balanced Latin Square, and random Constraint Satisfaction Problems (CSPs).

Several principles are followed in this paper concerning the experimental part. First, we compare metaheuristics that have at most two parameters. Indeed, the simple versions of the leading metaheuristics have only a few parameters and the procedure described above can tune them automatically. Second, for the sake of simplicity, we have not tested algorithms including restart mechanisms. This would make our automatic tuning procedure more complicated. More important, the restart device, although often useful, is in a sense orthogonal to the CLS mechanisms studied in this article that are applied during the move operation. Third, no clever heuristics have been used for generating the first configuration that is generally randomly produced, or only incorporates straightforward considerations[2]. In addition, for three among the five categories of tested problems, two different neighborhoods with specific degrees of intensification are used.

[1]Thanks to the "Centre d'électronique de l'Armement".

[2]For the latin square problem, a line contains the n different symbols (in any order) ; for the car sequencing, the initial assembly line contains the n cars (in any order).

Random CSPs

We have used the generator of random uniform binary CSPs designed by Bessière [32] to generate 30 CSP instances with two different densities. All are satisfiable instances placed before the complexity peak. Ten (resp. twenty) instances in the first (resp. second) category have 1000 (resp. 500) binary constraints, 1000 variables with a domain size 15, and tightness 50 (resp. 88). A tightness 50 means that 50 tuples over 225 (15×15) do not satisfy the constraints.

These constraint satisfaction instances are handled as optimization MAX-CSPs : the number of violated constraints is minimized during the search and a solution is given by a configuration with cost 0.

The usual definition of neighborhood used for CSPs is chosen here : a new configuration x' is a neighbor of the current configuration x if both have the same values, except for one variable v which takes different values in both configurations. More precisely, we define two different neighborhoods :

– (`VarConflict`) Configurations x and x' are neighbors iff v belongs to a violated constraint.
– (`Minton`) Following the Min-conflict heuristics proposed by Minton et al. [210], v belongs to a violated constraint, and the new value of v in configuration x' is different than the old value and produces the lowest number of conflicts.

Graph coloring instances

We have selected three graph coloring instances from the two most difficult categories in the catalogue : the `le450_15c` with 450 nodes and 16680 edges, the `le450_25c` with 450 nodes and 17425 edges, and the more dense `flat300_28` instance with 300 nodes and 21695 edges. All instances are embedded with specially constructed best solutions having, respectively, 15, 25 and 28 colors.

In this paper, graph coloring instances are encoded as MAX-CSP : variables are the vertices in the graph to be colored ; the number d of colors with which the graph must be colored yields domains ranging from 1 to d ; vertices linked by an edge must be colored with different colors : the corresponding variables must take different values. Coloring a graph in d colors amounts in minimizing the number of violated constraints and finding a solution with cost 0.

The two neighborhoods `VarConflict` and `Minton` defined above are used.

CELAR frequency assignment instances

We have also selected the three most difficult instances of radio link frequency assignment [76] : `celar6`, `celar7` and `celar8`. These instances are realistic since they have all been built from different sub-parts of a real problem. The `celar6` has 200 variables and 1322 constraints ; the `celar7` has 400 variables and 2865 constraints ; the `celar8` has 916 variables and 5744 constraints.

The variables are the frequencies to be assigned a value which belong to a predefined set of

allowed frequencies (domain size about 40). The constraints are of the form $|x_i - x_j| = \delta$ or $|x_i - x_j| > \delta$. Our encoding is standard and creates only the even variables in the CSP along with only the inequalities[3].

The objective function to be minimized is a weighted sum of violated constraints. Note that the weights of the constraints in `celar7` belong to the set $\{1, 10^2, 10^4, 10^6\}$, making this instance highly challenging. In addition to these problems, we have solved the `celar9` and `celar10` instances which have the same type of constraints and also unary soft constraints which assign some variables to given values. All the instances are encoded with the `VarConflict` neighborhood.

Spatially-balanced Latin Square

The latin square problem consists in placing r different symbols (values) in each row of a $r \times r$ square (i.e., grid or matrix) such that every value appears only once in each row and in each column. We tried an encoding where the latin square constraint on a row is satisfied and a specific neighborhood : swap in a row two values which take part in a conflict in a latin square column constraint. A simple descent algorithm (with allowed plateaus) can quickly find a solution for a latin square of size 100. This suggests that there are no local minima.

The *spatially-balanced* latin square problem [107] must also solve additional constraints on every value pair : the average distance between the columns of two values in each row must be equal to $(r+1)/3$. The problem is challenging for both exact and heuristic methods. An exact method can only solve the problem for sizes up to 8 and 9. A simple descent algorithm could not solve them. As shown in the experiments below, `TS` and `ID(best)` can solve them easily.

Car sequencing

The car sequencing problem deals with determining a sequence of cars on an assembly line so that predefined constraints are met. We consider here the nine harder instances available in the benchmark library CSPLib [98]. In these instances, every car must be built with predefined options. The permutation of the n cars on the assembly line must respect the following constraints : consider every option o_i ; for any sequence of $q(o_i)$ consecutive cars on the line, *at most* $p(o_i)$ of them must require option o_i, where $p(o_i)$ and $q(o_i)$ are two integers associated to o_i. A neighbor is obtained by simply permuting two cars on the assembly line. Two neighborhoods have been implemented :

– (`NoConflict`) Any two cars can be permuted.
– (`''VarConflict''`) Two cars c_1 and c_2 are permuted such that c_2 is randomly chosen while c_1 violates the requirement of an option o_i, that is, c_1 belongs to a sub-sequence of length $q(o_i)$ containing more than $p(o_i)$ cars with o_i.

[3] A bijection exists between odd and even variables. A simple propagation of the equalities allows us to deduce the values of the odd variables.

K.2.4 Compared optimization metaheuristics

We have compared IDWalk with correspondingly simple versions of leading optimization metaheuristics that manage only a few parameters. All algorithms have been developed within the same software system [223]. Our platform INCOP is implemented in C++ and the tests have been performed on a PentiumIII 935 Mhz with a Linux operating system. All algorithms belong to a hierarchy of classes that share code, so that sound comparisons can be made between them.

Our Metropolis algorithm is standard. It starts with a random configuration and a walk of length S is performed as follows. A neighbor is accepted if its cost is lower than or equal to the current configuration. A neighbor with a cost higher than the current configuration is accepted with a probability function of a constant temperature. When no neighbor is accepted, the current configuration is not changed. Our simulated annealing SA approach follows the same schema, with a temperature decreasing during the search. It has been implemented with a linear decrease from an initial temperature (given as parameter) to 0.

Our simple TS variant is implemented as follows : a tabu list of recently executed moves avoids coming back to previous configurations. The aspiration criterion is applied when a configuration is found that is better than the current best cost. The two parameters of this algorithm are the tabu list length (which is fixed) and the size of the examined neighborhood. The best neighbor which is not tabu is selected.

K.2.5 Results

This section reports the comparisons between ID(any), ID(best), simulated annealing (SA), Metropolis and tabu search (TS) on the presented problems. 20 among the 35 instances make no significant difference between the tested algorithms and the corresponding results are thus reported in Appendix K.5.

Note that the goal of these experiments is to compare simple versions of the leading metaheuristics implemented in the same software architecture. We do not compare with the best metaheuristics on every tested instance. In particular, ad-hoc metaheuristics obtain sometimes better results than our general-purpose algorithms do (see below). However, due to the efficient implementation of our library INCOP and due to the advances provided by IDWalk, very good results are often observed. More precisely :

- As shown below, ID(any) is excellent on CELAR instances and is competitive with state-of-the-art algorithms [172, 297, 175, 63]. The only slightly better general-purpose metaheuristic is GWW_idw, a more sophisticated population-based algorithm with four parameters [224].
- Several ad-hoc algorithms obtain very good results on the 3 tested graph coloring instances [214, 71, 91]. However, the results obtained by ID(best) and TS are impressive on le450_15c. Also, our TS, and our SA with more time [223], can color for the first time flat_300_28_0 in 30 colors.
- ID(best) and TS obtain even better results than the complicated variants of SA used by the designers of the balanced latin square problem [107].
- On car sequencing problems, we obtain competitive results as compared to the local search approach implemented in the COMET library [206] and the ant colony optimization approaches

described in [108] (although the lattest seems faster on the easiest car sequencing instances).

CELAR instances

Tab. K.1 – Comparisons between algorithms on CELAR instances. The first column contains the best bound ever found for the instance (not proven for celar7 and celar8). The second column reports the time per trial in minutes. For the other columns, each cell contains the average cost (left) on 10 or 20 trials, and the best cost (right). The numbers are reported minus the value of the best known bound, i.e., 0 means that the bound has been obtained.

	Bound	**T**	ID(any)		ID(best)		Metropolis		SA		TS	
celar6	3389	14	**58**	**0**	470	304	1659	517	778	150	389	227
celar7	343592	6	**29742**	**406**	$8.6\,10^5$	487406	$5.6\,10^6$	$2.5\,10^6$	$9\,10^5$	113301	$9\,10^5$	376787
celar8	262	50	29	5	131	73	108	38	**19**	**2**	84	38
celar9	15571	3	**0**	**0**	801	671	2188	416	69	0	644	249
celar10	31516	1	**0**	**0**	323	0	59	0	0	0	0	0

The results show that ID(any) is clearly superior to others. The only exception concerns celar8 for which SA is better than ID(any). The following remarks highlight the excellent performance of ID(any) :

- ID(any) can reach the best kwown bound for all the instances. With more available time, the best bound 262 is reached for celar8 and bounds less than 343600 can be obtained on the challenging celar7 that has a very chahuted landscape (with constraint violation weights ranging from 1 to 10^6).
- Only a few ad-hoc algorithms can obtain such results on celar6 and celar7 [172, 297], while all the tested algorithms are general-purpose.
- The excellent result on celar9 (10 sucesses on 10 trials) is in fact obtained in 7 s, instead of 3 min for others. The excellent result on celar10 is in fact obtained in 1 s, instead of resp. 47 s and 34 s for SA and TS.

Graph coloring instances

TS obtains generally the best results, especially on le450_25c. It can even color le450_25c once in 800 s with the VarConflict neighborhood.

ID(any) and ID(best) also obtain good results, especially on le450_15c.

Spatially-balanced latin square instances

These tests show that ID(best) and TS clearly dominate the others.

TAB. K.2 – Comparisons between algorithms on graph coloring instances. For `le450_15c`, a cell contains the average time required per trial in seconds and the number of times (on 10 trials) the graph can be colored in 15 colors (into parentheses). For `le450_25c`, a cell contains the average cost (left) and the best cost (right) among the ten trials, obtained in 800 seconds per trial. The numbers are reported minus 25, i.e., 0 means that the graph has been colored in 25 colors.

	Neighborhood	**#col**	ID(any)	ID(best)	Metropolis	SA	TS
le450_15c	VarConflict	15	**99 (10)**	151 (8)	220 (0)	82 (3)	112 (10)
le450_15c	Minton	15	27 (10)	8 (10)	108 (10)	74 (6)	**3 (10)**
le450_25c	VarConflict	25	3.3 2	3.6 2	4.1 3	5.9 2	**2.3 0**
le450_25c	Minton	25	3.2 3	3.5 2	3.2 2	3.8 2	**2.6 1**

TAB. K.3 – Comparisons between algorithms on spatially-balanced latin square instances. Each cell contains the average time in seconds per trial (over 10 trials). For `blatsq8`, all the algorithms always find a solution (10/10). For `blatsq9`, the number of successes (between 0 to 10) is indicated into parentheses.

	ID(any)	ID(best)	Metrop.	SA	TS
blatsq8	23	**1.5**	10	15	**2.8**
blatsq9	998 (6)	**5 (10)**	26 (10)	46 (10)	**9 (10)**

Car sequencing instances

Table K.4 collapses the results obtained on the two most difficult instances of car sequencing (in the CSPLib) : `pb10-93` and `pb16-81`.

The reader can first notice that the results obtained with the more "aggressive" neighborhood are better for all the metaheuristics. The trend is confirmed on the other instances in appendix, although this is not systematic.

On these instances, `ID(best)` give the best results (twice) or is only twice slower than the best one, that is Metropolis or `TS`. `ID(any)` and `SA` are less effective.

Summary

On the 15 instances tested above (some of them being encoded with two different neighborhoods), we can conclude that :

- `ID(any)` dominates others on 4 `CELAR` and 1 graph coloring instances.
- `ID(best)` dominates others on 1 spatially-balanced latin square instance and 2 car sequencing instances. It is also generally good when `TS` is the best.
- Metropolis dominates others on only 1 car sequencing instance and is sometimes very bad.

TAB. K.4 – Comparisons between algorithms on car sequencing instances. Each cell contains the average time in seconds per trial (over 10 trials) and the number of successes into parentheses (between 0 to 10).

	Neighborhood	ID(any)	ID(best)	Metrop.	SA	TS
pb10-93	NoConflict	759 (0)	1842 (6)	**737 (6)**	697 (0)	5902 (4)
pb10-93	VarConflict	1330 (1)	**442 (10)**	509 (7)	709 (4)	1400 (9)
pb16-81	NoConflict	2450 (8)	**499 (10)**	945 (10)	592 (9)	580 (9)
pb16-81	VarConflict	603 (2)	188 (10)	677 (10)	1039 (9)	**99 (10)**

– SA dominates others only on celar8 and is sometimes very bad.
– TS dominates others on 3 graph coloring instances, 1 spatially-balanced latin square instance and 1 car sequencing instance.

As a result, TS gives the best results on these instances, although it is bad on some CELAR problems, especially celar7.

We should highlight the excellent results obtained by the "best" metaheuristic among ID(any) and ID(best) on all the instances : one version of ID Walk is the best for 8 over the 15 tested instances, and is very efficient on 5 others (generally ID(best)). They are only clearly dominated by TS on the graph coloring instance le450_25c (with the 2 implemented neighborhoods).

K.2.6 Using the automatic tuning tool in our experiments

Our tuning tool has allowed us to perform the large number of tests gathered above. The robustness of the tuning process depends on the tested problem and metaheuristic. Car sequencing instances seem more difficult to be tuned automatically. Also, the tool is less reliable when applied with SA and metaheuristics with two parameters (TS and more sophisticated variants of ID Walk), so that a final manual tuning was sometimes necessary to obtain reliable results. The complexity times reported above do not include the tuning time. However, note that more than 80% of them have been obtained automatically. Especially, Table K.5 reports the overall time spent to obtain the results of ID(best) and ID(any) on the 15 instances above. This underlines that all the results, except 1, have been obtained automatically.

For readers who wish to investigate ID_Walk on their own, Table K.6 gathers the values selected for Max in our experiments.

K.3 Variants

Several variants of ID Walk have been designed to better understand the role of different devices on performance. Section K.3.1 describes these variants and Section K.3.2 perform some experiments that lead to significant results.

TAB. K.5 – Total time (tuning+solving) in minutes spent on the 15 instances by IDWalk. (N), (V) and (M) denote the different neighborhoods, resp., NoConflict, VarConflict, Minton.

Instance	celar6	celar7	celar8	celar9	celar10	blatsq8	blatsq9	pb10-93(N)
ID(any)	manual	147	666	36	2	7	311	142
ID(best)	414	200	702	45	51	2.5	4.5	524
Instance	pb10-93(V)	pb16-81(N)	pb16-81(V)	le_15c(V)	le_15c(M)	le_25c(V)	le_25c(M)	
ID(any)	295	611	164	117	24	429	223	
ID(best)	89	374	75	67	4	251	186	

TAB. K.6 – Values computed for the Max parameter by the automatic tuning tool (except for celar6).

Instance	celar6	celar7	celar8	celar9	celar10	blatsq8	blatsq9	pb10-93(N)
ID(any)	125	125	120	256	225	175	212	2800
ID(best)	15	7	29	45	16	125	71	1110
Instance	pb10-93(V)	pb16-81(N)	pb16-81(V)	le_15c(V)	le_15c(M)	le_25c(V)	le_25c(M)	
ID(any)	1200	900	562	40	4	100	6	
ID(best)	468	579	179	20	3	93	6	

K.3.1 Description of variants

In addition to the number S of moves, the variants ID(a,g) and ID(b,g) have only one parameter (like ID(any) or ID(best)), while ID(a,t) and ID(a,m) have two (like TS).

Variant ID(a,t) (ID(any) with a tabu list)

ID(a,t) is ID(any) endowed with a tabu list of fixed length. One of the Max neighbor is accepted iff its cost is better than or equal to the current cost and is not tabu.

Variant ID(a,g) ("Greedy" variant of ID(any))

At every move, ID(a,g) examines the Max candidates : it selects the best neighbor among the Max candidates if one of them improves or keeps the current cost ; otherwise it randomly selects *any* of them.

Remark : This variant is allowed by the original move procedure[4] implemented in the INCOP library [223]. More precisely, INCOP allows the user to define a *minimum number* Min *of neighbors* that are visited at every move, among which the best accepted candidate is returned. Without going into details, Min is set to 0 (or 1) in the variants above and is set to Max in the "greedy" variants.

[4]The Min parameter is also used in the Aspiration Plus strategy.

Variant `ID(b,g)` ("Greedy" variant of `ID(best)`)

`ID(b,g)` selects the best neighbor among the `Max` candidates (`Min=Max`). `ID(b,g)` is similar to a `TS` with no tabu list (or a tabu list of null length).

Other variants could be envisaged. In particular, many well known devices could enrich `ID Walk`, such as strategic oscillation (i.e., making `Max` vary with time). However, the aim of the next section is to compare the impact on performance of the following three mechanisms :

– the `Min` parameter,
– the `SpareNeighbor` diversification device,
– the tabu list.

K.3.2 First comparison between local search devices

Tab. K.7 – Measuring the impact of `Min`, `SpareNeighbor` and the tabu list on performance. Every cell has the same content as described in the previous tables (only the average cost appears for `celar7`). The last column `p-q` gives the length p of the `TS` tabu list and the length `q` of the `ID(a,t)` tabu list.

Instance	Neigh.	ID(a)	ID(a,t)	ID(a,g)	ID(b)	ID(b,g)	TS	p-q
celar6	VarC	58 0	60 0	96 0	470 304	408 308	389 227	1-6
celar7	VarC	$3\,10^4$	$4\,10^4$	$4.8\,10^4$	$8.6\,10^5$	$8\,10^5$	$9\,10^5$	50-48
celar8	VarC	29 5	37 13	38 16	131 73	91 54	84 38	2-45
celar9	VarC	0 0	0 0	0 0	801 671	36 313	644 249	15-12
celar10	VarC	0 0	0 0	0 0	323 0	0 0	0 0	2-5
le_15c	VarC	99 (10)	18 (10)	92 (10)	151 (8)	152 (6)	112 (10)	72-56
le_15c	Mint.	27 (10)	1 (10)	4 (10)	8 (10)	14 (10)	3 (10)	45-10
le_25c	VarC	3.3 2	3.3 2	3.7 3	3.6 2	2.8 1	2.3 0	2-4
le_25c	Mint.	3.2 3	2.4 1	4.1 2	3.5 2	2.8 2	2.6 1	2-4
blatsq8	VarC	99	171	84	2	4	4	0-2
blatsq9	VarC	1410(5)	1581(5)	972(1)	40(10)	16(10)	16(10)	0-3
pb10-93	NoC	759(0)	4301(0)	5979(0)	1842(6)	1698(2)	5902(4)	1-1
pb10-93	VarC	1330(1)	5381(0)	1457(0)	442(10)	1264(10)	1400(9)	1-1
pb16-81	NoC	2450(8)	894(2)	1763(0)	499(10)	1182(10)	580(9)	1-2
pb16-81	VarC	603(2)	890(0)	862(1)	188(10)	236(10)	99(10)	1-4

There is no need to go into details to discover a significant result in the local search field. The impact of the `SpareNeighbor` parameter on performance is highly crucial, while it is unused in most metaheuritics and implicit (and fixed to `best`) in a simple form of tabu search. The result is clear on three categories of problems (among five) : CELAR, latin square and car sequencing. Therefore we believe that this diversification device should be studied more carefully in the

future and incorporated in more metaheuristics. This surprising result also explains the success of ID(any) and ID(best) (in disjoint cases especially).

On the opposite, we can observe that the impact of Min is very weak.

We can finally observe that the tabu list is very effective for graph coloring instances, but the effect on the other categories of problems is not clear.

Note that all the metaheuristics have a good behavior on the uniform random CSP instances. The results are thus reported in Appendix K.5.

To sum up, 1 category of problems does not discriminate the tested devices, 1 category takes advantage on the tabu list, and 3 categories are well handled by this new SpareNeighbor diversification device.

Impact of parameter SpareNeighbor

Table K.7 has been arranged so that columns on the left side correspond to metaheuristics with SpareNeighbor=any, while columns on the right side correspond to metaheuristics with SpareNeighbor=best. The impact of parameter SpareNeighbor is very significant on CELAR, latin square and car sequencing problems, for which several orders of magnitude can sometimes be gained by choosing any (for CELAR) or best (for latin square and car sequencing).

On car sequencing instances, we can notice that a good performance is obtained by setting SpareNeighbor to best and by using a VarConflict neighborhood. Both trends indicate that the notion of intensification is very significant.

Impact of the tabu list

The observations are especially based on the comparison between ID(b,g) and TS since ID(b,g) can be viewed as TS with a null tabu list. The comparison between ID(any) and ID(a,t) is informative as well. The interest of the tabu list is not clear on CELAR and car sequencing problems. The impact of the tabu list seems null on latin square when SpareNeighbor is set to best since the automatic tuning procedure selects a list of length 0. It is even slightly counter-productive when SpareNeighbor = any. On the opposite, the gain in performance of the tabu list is quite clear on graph coloring for which ID(a,t) and our TS variant obtain even better results than ID(any) and ID(b,g) resp.

Weak impact of parameter Min

The reader should first understand that the parameter Min set to Max allows a more aggressive search but is generally more costly since all the neighbors are necessarily examined.

The observations are especially based on the comparison between ID(any) (Min=0) and ID(a,g) (Min=Max) on one hand, and ID(best) and ID(b,g) on the other hand. First, the impact on performance of setting Min to 0 or Max seems negligible, except for 4 instances (among 15+15) :

celar7, le450_15c (VarConflict), pb10-93 (VarConflict) and pb16-81 (NoConflict). Second, it is generally better to select a null value for Min, probably because a large value is more costly. Third, we also made experiments with another variant of ID(any) where Min can be tuned between 0 and Max. This variant did not pay off, so that the results are not reported in the paper.

This analysis suggests to not pay a great attention to this parameter and thus to favor a null value for Min in metaheuritics.

K.4 Conclusion

We have presented a very promising candidate list strategy. Its performance has been highlighted on 3 over the 5 categories of problems tested in this paper. Moreover, a first analysis has underlined the significance of the SpareNeighbor diversification device that is ignored by most of the metaheuristics.

All the metaheuristics compared in this paper have two points in common with IDWalk. They are simple and have a limited number of parameters. Moreover, they use a specific mechanism to exit from local minima.

Our study could be extended by analyzing the impact of random restart mechanisms. In particular, it would allow us to compare IDWalk with the GSAT and the WALKSAT [264] algorithms used for solving the well-known SAT problem (satisfiability of logical propositional formula). Note that WALKSAT is equipped with specific intensification and diversification devices.

IDWalk can be viewed as an instance of the AspirationPlus strategy, where parameters Min and Plus (see [99]) are set to 0, and where the aspiration level can be adjusted *dynamically* during the search : the aspiration level (threshold) for IDWalk always begins at the value of the current solution, but when none of the Max candidates qualify, the aspiration level is increased to the value of "any" candidate (SpareNeighbor=any) or of the "best" one (SpareNeighbor=best). Since the value of Min is not important (with "static" aspiration criteria) and since we have exhibited a significant and efficient instance of a dynamic AspirationPlus strategy, this paper strongly suggests the relevance of investigating additional dynamic forms in this novel and promising class of strategies.

In particular, the SpareNeighbor parameter can be generalized to take a value k between 1 (any) and Max (best), thus selecting the "best of k randomly chosen moves". Another variant would select any of the k best candidates.

Acknowledgments

Thanks to Pascal Van Hentenryck for useful discussions on preliminary works.

K.5 Results over less discriminating benchmarks

Tab. K.8 – Comparing metaheuristics on the 20 remaining instances : 4 random CSPs, 2 graph coloring instances, and 14 car sequencing instances. Every cell has the same content as described in the previous tables. For random CSPs, the time includes the tuning step (see Section K.2.2) and a run is interrupted as soon as a solution is found.

Instance	Neigh.	ID(a)	ID(a,t)	ID(a,g)	ID(a,m)	ID(b)	ID(b,g)	TS	Metr.	SA
csp1	VarC	91	110	228	88	165	–	121	200	105
csp1	Mint.	127	69	197	253	77	–	64	99	172
csp2	VarC	86	61	211	206	115	–	118	101	245
csp2	Mint.	49	76	126	161	98	–	67	42	43
flat_28	VarC	5.1 4	5.7 4	4.7 3	5.7 5	5.4 5	4.7 3	5 3	5.5 3	6.5 5
flat_28	Mint.	4.5 4	4.9 4	5 4	4.4 3	5.3 4	5 4	5.1 0	4.3 3	5.5 4
pb4-72	NoC	49	32	379	49	96	173	130	40	57
pb4-72	VarC	93	118	143	132	15	41	29	30	126
pb6-76	NoC	0.2	0.1	0.7	1.2	0.2	0.5	0.4	0.2	0.4
pb6-76	VarC	0.1	0.4	0.2	0.15	0.1	0.4	0.4	0.3	1.0
pb19-71	NoC	4	11	28	9	6	14	31	5	10
pb19-71	VarC	3	4	9	3	2	4	4	5	7
pb21-90	NoC	12	22	40	12	6	14	13	10	4
pb21-90	VarC	5	4	13	2	2	5	4	3	9
pb26-82	NoC	22	107	466	70	55	290	150	22	25
pb26-82	VarC	177	291	96	57	15	28	22	25	141
pb36-92	NoC	59	107	866	71	51	103	86	76	241
pb36-92	VarC	64	50	146	43	9	18	23	16	30
pb41-66	NoC	4	5	33	4	7	20	24	8	9
pb41-66	VarC	1.4	1.6	7.1	1.4	0.7	3.2	1.7	0.7	0.7

Annexe L

Incremental Move for Strip Packing Based on Maximal Holes

Article [229] : publié dans la revue IJAIT (International Journal on Artificial Intelligence Tools) en 2008

Auteurs : Bertrand Neveu, Gilles Trombettoni, Ignacio Araya, Maria-Cristina Riff

Abstract

When handling 2D packing problems, numerous incomplete and complete algorithms maintain a so-called bottom-left (BL) property : no rectangle placed in a container can be moved more left or bottom. While it is easy to make a rectangle BL when it is added in a container, it is more expensive to maintain all the placed pieces BL when a rectangle is removed. This prevents researchers from designing incremental moves for metaheuristics or efficient complete optimization algorithms. This paper investigates the possibility of violating the BL property. Instead, we propose to maintain the set of *maximal holes*, which allows incremental additions and removals of rectangles.

To validate our alternative approach, we have designed an incremental move, maintaining maximal holes, for the strip packing problem, a variant of the famous 2D bin-packing. We have also implemented a metaheuristic, *with no user-defined parameter*, using this move and standard greedy heuristics. We have finally designed two variants of this incomplete method. In the first variant, a better first layout is provided by a hyperheuristic proposed by some of the authors. In the second variant, a fast repacking procedure recovering the BL property is occasionally called during the local search.

Experimental results show that the approach is competitive with the best known incomplete algorithms.

L.1 Introduction

Packing problems consist in placing pieces in containers, such that the pieces do not intersect. Specific variants differ in the considered dimension (1D, 2D or 3D), in the type of pieces, or in additional constraints : for cutting applications, whether the (2D) container is guillotinable or not ; whether the objects can rotate, and so on. The 2D strip packing problem studied in this paper finds the best way for placing rectangles of given heights and widths, without overlapping, into a strip of given width and infinite height. The goal is to minimize the required height. We also study the variant that allows the rotation of rectangles with an angle of 90 degrees.

Packing problems have numerous practical applications. Strip packing occurs for instance in the cutting of rolls of paper or metal. In 3D, solving packing problems helps transporting a volume of goods in containers. The most interesting packing problems are all NP-hard, leading to the design of complete combinatorial algorithms, incomplete greedy heuristics, metaheuristics or genetic algorithms.

To limit the combinatorial explosion, most algorithms maintain the *Bottom-Left (BL) property, that is, a layout where the bottom and left segments of every rectangle touch the container or another rectangle*. First, the BL property lowers the number of possible locations for rectangles. Second, it can be proven that any solution of a 2D packing problem can be transformed into a solution respecting the BL property with a simple repacking procedure. However, when a rectangle is removed from the container, this repacking procedure is not local to the removed rectangle and to its neighbors, but modifies the whole layout in the worst case (e.g., when the removed rectangle is placed on the bottom-left corner of the container).

After a brief survey of existing algorithms in Section L.2, we present in Section L.3 how to add/remove one rectangle in/from a container. These operations are original in that they incrementally maintain a set of so-called *maximal holes* without necessarily recovering the BL property. These operations are generic and can be applied to any 2D packing problem. The second part of this paper experimentally shows that it is possible to design algorithms that, although they do not always respect the BL property, do not "fragment" the container, i.e., do not provide a bad layout with a lot of small holes between rectangles. Section L.4 introduces a new and incremental move for 2D strip packing that maintains the set of maximal holes during the addition and removal of rectangles. This move leads to a new incomplete algorithm for strip packing with **no** user-defined parameter. Section L.5 presents variants of this method that is improved in two ways. First, the metaheuristic starts with a better initial configuration obtained by a hyper-heuristic (proposed by some of the authors). Second, when the walk cannot improve the current solution, the metaheuristic launches a repacking procedure that recovers the BL property. The experiments presented in Section L.6 show the interest of these new methods based on our move.

L.2 Existing algorithms for strip packing

A lot of researchers have proposed different algorithms to handle bin packing so that we focus on strip packing in this section. In the last few years, the interest in strip packing has increased, hence the proposal of new approaches and the improvement of existing strategies.

Exact approaches are in general limited to small instances (Ref. [184]). Although not competitive with incomplete approaches, the branch and bound algorithm proposed by Martello et al. (Ref. [192]) is interesting and can solve some instances of up to 200 rectangles. Their algorithm computes good bounds obtained by geometrical considerations and a relaxation of the problem.

E. Hopper's thesis (Ref. [134]) exhaustively describes existing incomplete algorithms for strip packing. We just provide an overview of these heuristics ranging from simple greedy (constructive) algorithms to complex metaheuristics or genetic algorithms.

Bottom Left Fill (BLF) (see Ref. [55]) is a generalization of the first greedy heuristic proposed by Baker et al. (Ref. [20]) in 1980. BLF handles the rectangles in a predefined order, e.g., by decreasing width, height or surface. A rectangle R is placed in the first location that can contain R. The locations (i.e., corners or holes) are sorted according to their ordinate in the strip as first criterion and according to their abcissa as second criterion, so that an added rectangle is positioned on the strip as far down and to the left as possible. Therefore, the built layout always respects the BL property. Contrarily to the algorithm presented in this paper, many metaheuristics consider a move that exchanges two rectangles in the order followed by BLF. This is the case of the hybrid tabu search / genetic algorithm designed by Iori et al. (Ref. [142]).

Hopper presented in Ref. [135] an improved strategy of BLF called BLD, where the objects are ordered using various criteria (e.g., height, width, perimeter) and the algorithm selects the best result obtained. Lesh et al. in Ref.[185] have improved the BLD heuristic. Their BLD^* strategy repeats greedy placements with a specific randomized ordering until a time limit is reached.

The Best-Fit (BF) greedy heuristic proposed by Burke et al. in Ref. [46] adopts in a sense a dual strategy while also respecting the Bottom[-Left] property. At each step, a most bottom location in the partial solution is considered, and the rectangle fitting best into it is selected, if any. In Ref. [47], Burke et al. improve their approach by using a metaheuristic phase (implemented by a tabu search, a simulated annealing or a genetic algorithm) for repairing the last part of the solution obtained by BF.

Finally, two last approaches must be mentioned and will constitute our main competitors. Bortfeldt proposes in Ref. [40] a very sophisticated genetic algorithm directly working with the geometry of the layout. The best algorithm for handling strip packing with rectangles of fixed orientation is a reactive GRASP algorithm (Ref. [6]) working as follows : all the rectangles are first placed on the strip with a randomized (and improved) BF greedy heuristic. Some rectangles on the top of the strip are then removed and placed again with the greedy algorithm (in a different order). Several such steps are performed with an increasing portion of rectangles, adopting a variable neighborhood search strategy.

L.3 Maintaining "maximal holes"

The key idea behind our approach is to incrementally maintain a set of *maximal holes* when rectangles are added or removed.

Definition 23 *(Maximal hole) Let us consider a container C partially filled with a set S of rectangles. A maximal hole H (w.r.t. C and S) is an empty rectangular surface in C such that :*

- *H does not intersect any rectangle in S (i.e., H is a "hole" in the container),*
- *H is maximal, i.e., there exists no hole H' such that H is included inside H' (notation : the inclusion of a rectangle H inside a rectangle H' will be denoted by $H \subset H'$)*[1].

Fig. L.1 shows three examples with resp. 2, 2 and 4 maximal holes (from left to right). The maximal hole in grey corresponds to the most bottom-left one.

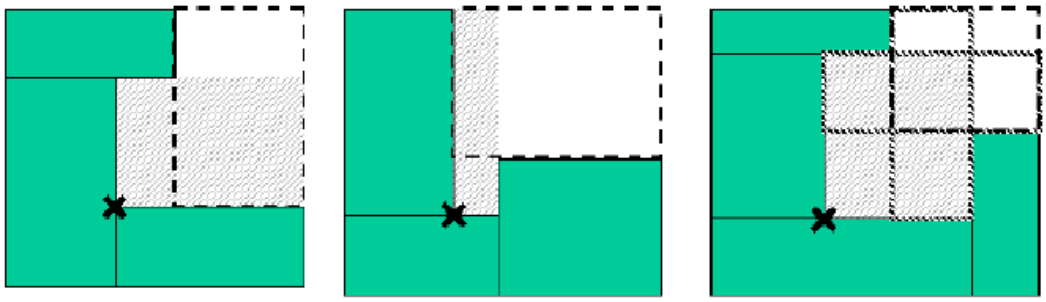


FIG. L.1 – Examples of maximal holes

Most of existing algorithms can use such a set of maximal holes for implementing their atomic operations. In particular, the BLF and BF greedy heuristics introduced above can implement the possible locations (into which the rectangles are added) as the set of maximal holes.

However, the interest is even greater. We claim that it is possible to design algorithms whose number of maximal holes remains small in practice during the search, even though the rectangles are removed, violating the BL property. The idea is the following. Thanks to the set of maximal holes, when a rectangle is removed, we do not modify the partial solution to make the rectangles BL again. Instead, we just update the set of maximal holes. Thus, a rectangle R placed in the future in the container will be BL (if it exactly covers a maximal hole)[2]. The main interest is to still limit the set of possible locations for the rectangles (to the maximal holes) while preserving the incrementality after a rectangle removal. In a sense, the good results obtained on strip packing problems by the metaheuristic proposed in this article experimentally validate this claim.

[1] This property implies that the bottom and left segments of H touch the container or rectangles in S.

[2] We have done simply no effort in the metaheuristic presented hereafter to locally improve the layout when a (small) rectangle is added in a (large) hole so that the rectangle R is not BL. We let the evaluation of the objective function do the selection between neighbor candidates.

Atomic operations between two holes

Although other modelings are possible, a rectangle or a rectangular hole R is represented by four coordinates : $R.x_L$, $R.y_B$, $R.x_R$, $R.y_T$: $(R.x_L, R.y_B)$ represents the bottom-left corner of R while $(R.x_R, R.y_T)$ is the top-right corner.

The incremental additions and removals of rectangles into/from a container are based on two operations between rectangles and rectangular holes. The **Minus**(H, R) operation between a hole H and a rectangle R is used when a rectangle is added in a container. It returns the set of maximal holes that remain when (the newly added) R intersects H. A simple computation of the newly created maximal holes (`Holes`) is performed as follows :

1. `Holes` $\leftarrow \emptyset$
2. `If` $R.x_R < H.x_R$ `then Holes` $\leftarrow$ `Holes` $\cup \{(R.x_R, H.y_B, H.x_R, H.y_T)\}$ `EndIf`
3. `If` $R.y_T < H.y_T$ `then Holes` $\leftarrow$ `Holes` $\cup \{(H.x_L, R.y_T, H.x_R, H.y_T)\}$ `EndIf`
4. `If` $H.x_L < R.x_L$ `then Holes` $\leftarrow$ `Holes` $\cup \{(H.x_L, H.y_B, R.x_L, H.y_T)\}$ `EndIf`
5. `If` $H.y_B < R.y_B$ `then Holes` $\leftarrow$ `Holes` $\cup \{(H.x_L, H.y_B, H.x_R, R.y_B)\}$ `EndIf`

`Minus`(H, R) may create less than four holes because the tested conditions are generally not fulfilled simultaneously.

The second operation **Plus**(H_1, H_2) holds between two rectangular maximal holes. If H_1 and H_2 intersect or are contiguous, `Plus` returns at most the following two new maximal holes :

– ($\max(H_1.x_L, H_2.x_L)$, $\min(H_1.y_B, H_2.y_B)$, $\min(H_1.x_R, H_2.x_R)$, $\max(H_1.y_T, H_2.y_T)$)
– ($\min(H_1.x_L, H_2.x_L)$, $\max(H_1.y_B, H_2.y_B)$, $\max(H_1.x_R, H_2.x_R)$, $\min(H_1.y_T, H_2.y_T)$)

Once again, a returned "degenerate" hole reduced to a single segment will not be considered.

Addition and removal of rectangles

Based on these operations, we present the two procedures used in most algorithms handling any 2D packing problem : `AddRectangle` and `RemoveRectangle`.

`AddRectangle`(in R, in/out C, in/out S) updates the set S of maximal holes when the rectangle R is added into the container (C is the set of rectangles placed in the container. At the beginning, S is reduced to the initial empty rectangular container.) `AddRectangle` mainly applies the `Minus` operator to R and to the holes in S that intersect R, as follows :

1. Add R in C.
2. For every hole in S intersecting R, add in a set `setH` the holes returned by `Minus`(H, R).
3. Filter `setH` by preserving only the *maximal* holes.

Remarks :

– The case may occur that two holes H_1 and H_2, each created by two different calls to `Minus`, verify $H_1 \subset H_2$. This justifies the third step.
– (Correction) A proof by contradiction helps us to understand that a newly created hole needs not be "merged" with a contiguous hole H' which does not intersect R to (recursively) build a larger hole. Otherwise indeed, it would mean that H' was not maximal. This point has a significant and positive impact on the efficiency of the procedure that visits only holes intersecting R (and not their "neighbors").

The procedure `RemoveRectangle` is a bit more complex. `RemoveRectangle` (in R, in/out C, in/out S) updates the set S of maximal holes when the rectangle R is removed from the container. It replaces R by a hole H and applies the `Plus` operation on H and its contiguous holes (if any). A fixed-point process is applied to ensure completeness. The detailed pseudo-code is described hereafter.

```
Algorithm RemoveRectangle (in R, in/out C, in/out S)
    Remove R from C ; Add R in S
    Initialize a list Holes with holes in S that are contiguous to R
    Initialize a list HolePairs with pairs (R, H) such that H is in Holes
    while HolePairs is not empty do
        Select (hole1, hole2) in Holepairs
        Remove (hole1, hole2) from HolePairs
        newHoles ← Plus (hole1, hole2) /* newHoles contains at most 2 holes */
        for every newHole in newHoles do
            for every hole in Holes do
                /* Ensure maximality */
                if newHole ⊂ hole then
                    delete newHole from newHoles ; break
                else
                    if hole ⊂ newHole then
                        delete hole from Holes
                    end
                end
            end
            if newHole ∈ newHoles then
                Add newHole to Holes
            else
                Add to HolePairs all the pairs (newHole, H) such that H is in Holes
            end
        end
    end
end.
```

Proposition 15 *(Termination and correction of* `RemoveRectangle`*) Let R be a rectangle to be removed from a container, let S be the corresponding set of maximal holes.*

A call to `RemoveRectangle` *terminates and updates S with the set of all the maximal holes of the container.*

Proof. (sketch ; the full proof requires an induction)

The termination is based on several points :

- Like for `AddRectangle`, it is not necessary to visit holes that are not pushed initially in `HolePairs` (i.e., the procedure visits only the neighbors of R).
- Because the `Plus` operation does not return more than two holes, the number of holes in `Holes` never increases during the execution of `RemoveRectangle`.
- The items above and the definition of `Plus` imply that the union of all the holes created during the execution of `RemoveRectangle` does not change. In other words, the "surface" covered by all the considered holes (R and neighbor holes) is constant during the execution of `RemoveRectangle`.
- The `Plus`(H_1, H_2) operation generates at most two holes H'_1, H'_2 that are larger than or equal to the input holes.

These points explain why a fixed-point is reached. The correction is ensured by the exhaustive application of the `Plus` operation to every possible pair. □

The `RemoveRectangle` procedure can be used by a classical 2D packing algorithm satisfying the BL property : when a rectangle is removed (or placed elsewhere in the container), the rectangles already placed in the container must be moved to recover the BL property, trying to limit the "fragmentation" of the container. However, this article explores the possibility of doing nothing special after a rectangle removal. Such an approach is described hereafter for solving strip packing.

L.4 An incremental move for strip packing

The strip packing is a variant of the 2D bin packing problem. A set of rectangles must be positioned in *one* container, called *strip*, which is a rectangular area. The strip has a fixed width dimension and a variable height. The goal is to place all the rectangles on the strip with no overlapping, using a minimum height of the container.

As said in the introduction, once we work in more than one dimension, the objects placed in the container are very dependent on each other and it is very difficult to incrementally repair the current solution. This explains why existing metaheuristics or genetic algorithms, allowing the search to escape from local minima, are not endowed with a low-cost "move". Most of the approaches use a classical greedy heuristic, e.g., BLF or BF. A widespread move consists, for instance, in exchanging two rectangles in the order in which the greedy heuristic will handle the rectangles. We understand that exchanging two rectangles i and j in the order implies, in the worst case, to position again all the rectangles after i in the order.

To handle strip packing, our metaheuristic uses a move based on the "geometry" of the rectangles on the strip. This move makes an intensive use of the incremental `AddRectangle` and `RemoveRectangle` procedures. It removes one rectangle R on the top of the layout and places it inside the strip. More precisely, the new location for R is a maximal hole or a placed rectangle. The rectangles of the layout that intersect R in its new location are placed again on the strip with a greedy heuristic such as Best-Fit Decreasing (`BF`). More precisely, a move is implemented as follows :

1. Take one rectangle R the top side of which is the highest on the strip (the case may occur that several rectangles are candidates).
2. Select R', which is one rectangle on the strip or one maximal hole, such that when R is placed in the bottom-left corner of R' then :
   - R remains inside the strip,
   - the new position of R is strictly lower than its previous position.
3. Consider the set S of rectangles that would intersect R in its new position. The rectangles in S must be placed elsewhere. First remove them from the strip with calls to `RemoveRectangle`.
4. Place R in the new position selected at step 2.
5. Place again the rectangles in S with the greedy heuristic G.

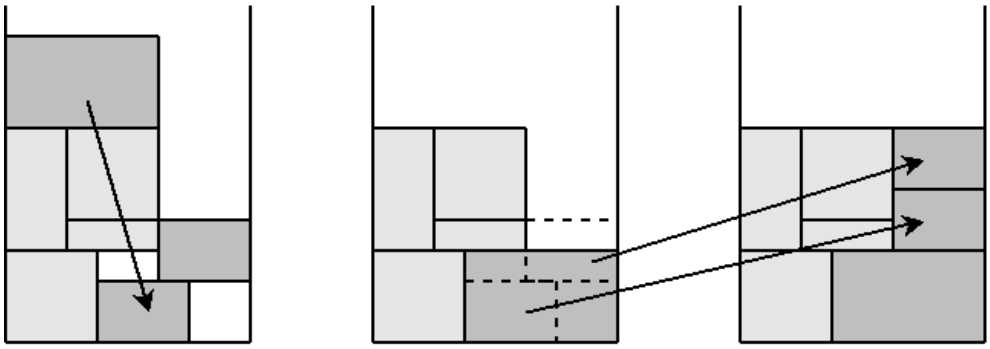


FIG. L.2 – One complete move

The steps 1 and 2 above pursue better solutions in an aggressive way (intensification). A similar move has been mentioned in Ref.[6] while it has not been used in their final heuristic.

The evaluation of the objective function (to be minimized) could be the (one-dimensional) height h of the layout. However, to break ties, we have also considered the number u of units filled by rectangles on the highest line of the layout. The finer two-dimensional objective function is thus equal to $w \times (h-1) + u$, where w is the width of the strip.

In the strip packing variant where rectangles can rotate with an angle of 90 degrees, a move is modified as follows. The step 2 (resp. 4) above must also select an orientation for the rectangle R, with a probability 0.5 for both possibilities.

Time complexity

It appears that the notion of maximal holes, designed by *maximal areas*, has been introduced independently by El Hayek et al. in Ref. [77] for 2D bin packing problems. They have not detailed operators like `Minus`, `Plus`, `AddRectangle`, `RemoveRectangle`. However, they have proven that the maximum number a of maximal areas managed during a packing procedure is $O(n^2)$, where n is the number of rectangles to be placed.

Our operators `Minus`(H, R) and `Plus`(H_1, H_2) are $O(1)$ because they return at most respectively four and two new maximal holes. Therefore, our procedure `AddRectangle` is $O(a_1)$, where a_1 is the number of holes intersecting R ($a_1 \leq a$). Our procedure `RemoveRectangle` is $O(a_2^2)$, where a_2 is the number of holes contiguous to R, i.e., a_2 is the size of the list `Holes` in the pseudo-code ($a_2 \leq a$).

We report in Table L.1 some statistics from experiments on Hopper and Turton's instances (every class contains three instances ; see Section L.6). They show that our procedures using maximal holes have a low time complexity in practice. This justifies why our move is claimed to be *incremental*.

The last five columns report statistics performed during the whole sequence of moves run to handle the different instances. The statistics first show (column 3) that the maximum number a of maximal holes grows linearly and is close to the number n of rectangles. Second, the number of displaced rectangles in one move (column 5) remains always very small on average. Finally, the last two columns concern `RemoveRectangle`. They report the maximum and average number a_2 of contiguous holes managed by `RemoveRectangle`. Note that a_2 remains small on average.

Tab. L.1 – Statistics on Hopper and Turton's instances. #rectangles : number of rectangles of the instance ; maximal holes : maximum number of maximal holes during the search ; max rect. (resp. aver. rect.) : maximum (resp. average) number of displaced rectangles during one move ; max (resp. aver.) cont. holes : maximum (resp. average) number of contiguous holes considered by `RemoveRectangle`.

Class	#rectangles	maximal holes	max rect.	aver. rect.	max cont. holes	aver. cont. holes
C1	17	16	08	1.9	10.00	2.95
C2	25	19	10	1.6	12.66	3.01
C3	29	26	11	2.0	18.00	3.76
C4	49	45	16	2.5	17.33	3.85
C5	73	55	18	2.4	21.00	3.73
C6	97	78	19	2.6	22.66	3.90
C7	197	145	25	2.8	30.33	4.21

Selected metaheuristic and greedy heuristics

We have designed an incomplete method able to be specialized with number of greedy heuristics and metaheuristics (performing a sequence of the moves described above). The method adopts the following scheme :

1. a greedy heuristic first computes an initial layout,
2. a metaheuristic, driven by an automatic tuning procedure, then repairs the first solution.

We have tried the main metaheuristics available in the `INCOP C++` library (Ref. [223]) developed by the first author : tabu search, simulated annealing (and a Metropolis variant), and `ID(best)` (Ref. [231]) which is a simple variant of `ID Walk` with only one parameter (`MaxNeighbors`). The simulated annealing has been discarded because it yields the worst performance on strip packing. Tabu search (with two parameters), `ID Walk` with two parameters and `ID(best)` gave a good performance, and we have chosen `ID(best)` for its simplicity. In particular, the automatic tuning procedure provided by `INCOP` is more robust when it tunes only one parameter.

`ID(best)` is a *candidate list strategy* that uses one parameter `MaxNeighbors` to perform one move from a configuration x to a configuration x', as follows :

1. `ID(best)` picks randomly neighbor candidates one by one and evaluates them. The *first* neighbor x' with a cost better than or equal to the cost of x is accepted.
2. If `MaxNeighbors` neighbors have been rejected, then the *best* neighbor among them (with a cost strictly worse than the cost of x) is selected.

`ID(best)` has a common behavior with a variant of tabu search once all the candidates have been rejected (item 2 above). However, the differences are the absence of tabu list and another policy for selecting a neighbor x' (step 1 above).

A description of the automatic tuning procedure can be found in Ref. [231]. Every trial is independent from the others and is interrupted when a maximal amount of CPU time is exceeded. A trial is a succession of one automatic *tuning step*, where the parameter `MaxNeighbors` is tuned in a dichotomic way on short walks, and one *exploring step* where the parameter is kept fixed during a long walk. After one tuning step and one exploring step, the trial is continued with a larger number of moves.

The same policy has been followed for all the tested strip packing benchmarks. In a same trial, the first tuning step runs 24 walks (with different values for the parameter) of 200 moves each ; the first exploring step is a walk made of 10000 moves ; the second tuning step runs 24 walks of 800 moves each ; the second exploring step is a walk made of 40000 moves, and so on. It turns out that the tuning time represents about 30% of the global time.

The initial value of `MaxNeighbors` has been empirically set to the number of rectangles to be positioned. For the strip packing variant allowing the rotation of rectangles, the initial value is the number of rectangles multiplied by 2.

Although simple, the automatic tuning procedure is robust (i.e., the tuned value converges and produces a good configuration) in a majority of trials.

Our algorithm works with any of the standard greedy heuristics : BF or BLF, considering one among the four possible orders among rectangles : largest Width first (w), largest Height first (h), largest Surface first (s), largest Perimeter (p), providing eight possible combinations : `BFw`, `BFh`, `BFs`, `BFp`, `BLFw`, `BLFh`, `BLFs`, `BLFp`[3].
The initial layout is provided by a greedy heuristic randomly picked among the eight ones.
During the moves performed by `ID(best)`, the choice of greedy heuristic has been directly incorporated into the neighborhood : in addition to the choice of rectangle R picked on the top of the layout and to the location in which the rectangle R will be moved, for placing again on the strip the displaced rectangles, one of the eight greedy heuristics is chosen at random.

This randomization presents two advantages. First, in our understanding, some biases are avoided. For instance, it is well-known that, when handling 2D packing with allowed rotation of rectangles, a bias introduced by the BF greedy heuristic is to place a lot of rectangles "vertically" on the right side of the strip. Second, it avoids the user to choose among the (eight) available greedy heuristics.

Thus, used with `ID(best)` and its automatically tuned parameter, the method proposed in this paper has simply *no* user-defined parameter.

L.5 Variants of our incomplete algorithm

We have added two features to our metaheuristic (denoted by `IDW` below). The first one consists in replacing the greedy heuristic used for the first layout with a hyper-heuristic (`HH`) recently proposed by some of the authors in Ref. [8] (the approach is denoted by `HH+IDW` below). The second feature consists in "rePacking" the layout at times for recovering the BL property (the corresponding approaches are denoted by `IDW+P` and `HH+IDW+P` below).

L.5.1 First layout obtained by a hyperheuristic

The hyperheuristic framework manages a set of low-level heuristics and tries to find a way to apply them. In Ref. [8], the authors have designed a hyperheuristic for handling strip packing instances. The hyperheuristic builds a sequence of greedy heuristics. Each element of the sequence places a given number of rectangles on the strip with the corresponding greedy heuristic. The hyperheuristic performs a hill-climbing on the sequence by applying moves that add, remove or replace elements (i.e., greedy heuristics) in the sequence.

Four greedy heuristics are used inside the hyperheuristic : the standard BLF and BF heuristics mentioned above ; the recursive heuristic HR which is also used for problems respecting the guillotine cut constraint (Ref. [310]) ; the BFDH* heuristic used by Bortfeldt for generating the initial configurations in its genetic algorithm (Ref. [40]).

Our metaheuristic has been extended by using this hyperheuristic to generate the first layout, yielding the `HH+IDW` variant.

[3] On the tested instances, among the eight combinations, some greedy heuristics are better than some others *on average*, but none of them is always strictly dominated by one of the others.

L.5.2 Repacking procedure

The idea behind our metaheuristic was to not recover the BL property every time a rectangle is removed during the moves performed by local search. Instead, the set of maximal holes is incrementally recomputed. However, this is not contradictory with calls to a *repacking procedure* (recovering the BL procedure), provided that the time spent by the repacking procedure is dominated by that of the local search.

The repacking procedure is a simple loop on all the rectangles of the current layout handled in a bottom-left order. At the i^{th} iteration :

- the rectangles from 1 to $i-1$ in the order verify the BL property,
- the i^{th} rectangle is removed from the layout and placed again by a BLF-like greedy heuristic (i.e., in the most bottom-left maximal hole of the current layout), making BL the rectangles from 1 to i in the order.

The main difference with the BLF greedy heuristic is that all the rectangles are considered to be on the layout when the i^{th} rectangle is handled : the rectangles from 1 to $i-1$ in the order, but also the other ones. At the end of this loop, all the rectangles thus verify the BL property.

It is straightforward to prove by induction that the layout obtained by such a repacking procedure is better than or equal to the initial layout. Indeed, every rectangle does not move or is displaced in a most bottom-left position.

We have decided to call the repacking procedure so occasionally that the time spent is dominated by the local search. We also wanted the local search to run the repacking procedure in an adaptive way. That is why the repacking procedure is launched when the lattest move led the current solution to a significantly worse cost, i.e., an increase of 1 of the upper line of the layout. In our experiments below, we have evaluated the part of the repacking procedure in the overall CPU time. This ratio is comprised between 3% and 20% according to the considered instance. A ratio of 20% occurs when the selected neighborhood is very small (i.e., `MaxNeighbors` is small) and a lot of accepted moves in the walk increase the cost.

Interleaving this rePacking procedure with our metaheuristic leads to the `IDW+P` variant. Adding as a preprocessing a call to `HH` leads to the `HH+IDW+P` variant.

L.6 Experiments

We have performed experiments on five series enclosing 547 benchmarks. The 21 zero-waste instances by Hopper and Turton [135] are classified into 7 classes of increasing strip width. The corresponding results are reported in Tables L.2 and L.5. Tables L.3 and L.6 show the results obtained on the 13 *gcut* instances by Beasley [25], the 3 *cgcut* instances, and the 10 *beng* instances by Bengtsson [26]. The results obtained on the 12 *ngcut* instances also proposed by Beasley are not reported because they are all optimally solved by our metaheuristic (in less than 3 seconds) and by competitors. Note that the `gcut` instances have a reasonable number of rectangles but have a wide strip ranging from 250 to 3000 units.

Tables L.4 and L.7 include the results obtained on the 500 instances proposed by Martello and Vigo [193], Berkey and Wang [30]. This huge number of instances are classified into 10 classes, themselves subdivided into 5 series of 10 instances each. The classes define different strip widths ranging from 10 to 300. The 5 series define instances with resp. 20, 40, 60, 80 or 100 rectangles. Finally, Table L.8 reports the results obtained on the Hopper and Turton instances (Ref. [135]) for a variant of strip packing where rectangles can rotate with an angle of 90 degrees.

Competitors

Not all the presented competitors have tested the five presented benchmark series. Also, they have adopted slightly different experimental conditions.

The hybrid tabu/genetic algorithm is run by Iori during 300 seconds on a Pentium III at 800 Mhz (Ref. [142]). The BLD* algorithm is run by Lesh et al on a Pentium at 2 Ghz (Ref. [185]). The two presented results correspond to time limits of respectively 60 seconds and 3600 seconds.

The results presented for Burke et al. correspond to their BF heuristic enhanced with tabu search, simulated annealing or a genetic algorithm. They run their heuristic 10 times with a time limit of 60 seconds per run on a Pentium IV at 2 Ghz. Bortfeldt's genetic algorithm is run 10 times on every instance with an average time per run of 160 seconds on a Pentium at 2 Ghz. The GRASP algorithm (Ref. [6]) is also run 10 times on every instance with a time limit of 60 seconds on Pentium IV Mobile at 2 Ghz.

Experimental conditions

For every instance, our metaheuristic (`IDW`, `HH+IDW`, `IDW+P` or `HH+IDW+P`) spends a total time of 1000 seconds on a Pentium IV at 2.66 Ghz. This time corresponds to 10 runs of 100 seconds each. The same amount of CPU time is allocated to `HH`. When `HH` computes the first layout used by `IDW` (resp. `IDW+P`), 10 seconds are allocated to `HH` while 90 seconds are used for `IDW` (resp. `IDW+P`).

For all the heuristics, we report the best bound obtained. The average bounds are not reported in the tests reported in Section L.6.1 because they are not always available in the literature and are indeed sometimes meaningless for certain algorithms. However, the average costs appear in Section L.6.2 that recalls the results of the best competitors.

Although all the algorithms have been performed neither on the same computers nor in exactly the same amount of CPU time, we think that the comparison between competitors is rather fair. Indeed, if the machines and the allowed times were normalized, the (ratio) differences between competitors would not exceed a factor 2. It is not sufficient to bring a significant gap in terms of computed bound on the tested instances of this NP-hard strip packing problem.

L.6.1 Comparison of our metaheuristic with competitors

Table L.2

Every class of the Hopper and Turton's instances contains 3 zero-waste instances with a given width (column Width), a given number of rectangles (N), and a given optimum - the ordinate of the top side of a highest rectangle in the strip - obtained by construction (*Opt.*). The cells report the average percentage deviation from optimum. The reported results for Burke's algorithm is their best tested metaheuristic : BF + simulated annealing. The reported results for Lesh et al's algorithm were obtained in 3600 seconds.

GRASP outperforms the other algorithms, especially on the largest classes 6 and 7. `ID Walk` is generally better than other competitors (except GRASP). Bortfeldt's approach behaves well on class 7.

Tab. L.2 – Comparison on Hopper and Turton instances.

Class	Width	N	Opt.	`IDW`	Iori	Lesh	Burke	Bortfeldt	GRASP
C1	20	16–17	20	0.00	1.59	–	0.00	1.59	0.00
C2	40	25	15	0.00	2.08	–	6.25	2.08	0.00
C3	60	28–29	30	2.15	2.15	–	3.23	3.23	1.08
C4	60	49	60	1.64	4.75	–	1.64	2.70	1.64
C5	60	73	90	1.81	3.92	2.17	1.46	1.46	1.10
C6	80	97	120	1.37	4.00	1.64	1.37	1.64	0.83
C7	160	196.3	240	1.77	–	–	1.77	1.23	1.23
# optimal solutions				7/21	5/18	0/6	3/21	4/21	8/21

Table L.3

The column LB yields Lower Bound computations of the optima (which are not necessarily reached). The following columns report the bound of the best solution computed by the corresponding algorithm. We report the bound of the best solution obtained by Lesh et al.'s algorithm in resp. 60 and 3600 seconds. The benchmarks `gcut09'...gcut13'` (Ref. [6]) are variants of the benchmarks `gcut09...gcut13` in which the rectangles have been rotated with an angle of 90 degrees (i.e., the width and the height of every rectangle have been exchanged).

On the three presented categories, note that GRASP is generally better than or similar to `ID Walk` (except on *gcut09'*) which is itself better than Iori's algorithm and Lesh's approach in 3600 seconds.

Tab. L.3 – Comparison on *beng*, *cgcut* and *gcut* instances.

Instance	Width	N	LB	IDW	Iori	Lesh 60	Lesh 3600	GRASP
beng01	25	20	30	30	31	–	–	30
beng02	25	40	57	58	58	–	–	57
beng03	25	60	84	85	86	–	–	84
beng04	25	80	107	108	110	–	–	107
beng05	25	100	134	135	136	–	–	134
beng06	40	40	36	36	37	–	–	36
beng07	40	80	67	68	69	–	–	67
beng08	40	120	101	102	–	–	–	101
beng09	40	160	126	126	–	–	–	126
beng10	40	200	156	156	–	–	–	156
cgcut01	10	16	23	23	23	–	–	23
cgcut02	70	23	63	65	65	–	–	65
cgcut03	70	62	636	675	676	–	–	661
gcut01	250	10	1016	1016	1016	1016	1016	1016
gcut02	250	20	1133	1194	1207	1211	1195	1191
gcut03	250	30	1803	1803	1803	1803	1803	1803
gcut04	250	50	2934	3030	3130	3072	3054	3002
gcut05	500	10	1172	1273	1273	1273	1273	1273
gcut06	500	20	2514	2686	2675	2682	2656	2627
gcut07	500	30	4641	4697	4758	4795	4754	4693
gcut08	500	50	5703	5960	6240	6181	6081	5908
gcut09'	1000	10	2022	2241	–	–	–	2256
gcut10'	1000	20	5356	6399	–	–	–	6393
gcut11'	1000	30	6537	7736	–	–	–	7736
gcut12'	1000	50	12522	13172	–	–	–	13172
gcut13'	3000	32	4772	5055	–	–	–	5009

Table L.4

The cells in Table L.4 include the percentage deviation between the (not necessarily reached) lower bound and a value $v : v$ is an average value over the 50 best solution costs obtained for the 50 instances in a given class.

From best to worst, the order between competitors is GRASP, `ID Walk`[4], Bortfeldt's algorithm, Lesh's algorithm, Iori's algorithm.

[4]The tests reported in this table correspond to an old version of our metaheuristic, where `ID Walk` is run 20 times per instance, with two specified greedy heuristics : 10 trials with `BFw`, and 10 trials with `BLFw` or `BFh`, depending of the considered instance. However, on the same instances, the results of the variants presented in Table L.7 have been obtained with the described version, where the selected greedy heuristics are no more defined by the user.

Tab. L.4 – Comparison on the 500 instances proposed by Martello, Vigo, Berkey, Wang.

Class	Width	IDW	Iori	Lesh 60	Lesh 3600	Bortfeldt	GRASP
01	10	0.67	0.64	0.81	0.68	0.75	0.63
02	30	0.58	1.78	1.12	0.42	0.88	0.10
03	40	2.16	3.05	2.71	2.23	2.52	1.73
04	100	3.47	5.08	4.41	3.54	3.19	2.02
05	100	2.20	3.15	2.85	2.43	2.59	2.05
06	300	4.86	5.99	6.45	5.13	4.96	3.08
07	100	1.12	1.16	1.17	1.12	1.19	1.10
08	100	4.19	6.16	5.99	4.93	3.85	3.57
09	100	0.07	0.07	0.07	0.07	0.07	0.07
10	100	3.12	4.67	4.11	3.48	3.05	2.93
Overall		2.24%	3.17%	2.97%	2.40%	2.31%	1.73%

L.6.2 Results obtained by variants of our metaheuristic

Tables L.5, L.6 and L.7 report a comparison between different variants of our metaheuristic (IDW) : HH+IDW, IDW+P and HH+IDW+P. The tables also show the best and average times of HH and of our best competitor : GRASP.

These tables mainly underline the good performance obtained by a hybridization between HH and IDW which nearly reaches (and sometimes exceeds) the performance of GRASP.

Tab. L.5 – Comparison on Hopper and Turton's instances.

Class	N	Opt.	IDW		HH+IDW		IDW+P		HH+IDW+P		HH		GRASP	
			best	aver.	best	aver.	best	aver.	best	aver.	best	aver.	best	aver.
C1	16–17	20	0.0	0.79	0.0	0.48	0.0	0.16	0.0	0.95	1.59	2.38	0.0	0.0
C2	25	15	0.0	3.12	0.0	1.87	0.0	1.90	0.0	1.87	0.0	1.46	0.0	0.0
C3	28–29	30	2.15	3.10	1.08	2.58	2.15	2.90	2.15	2.26	2.15	2.80	1.08	1.08
C4	49	60	1.64	2.81	1.64	2.43	2.17	2.70	1.64	2.49	2.70	3.22	1.64	1.64
C5	73	90	1.81	2.73	1.10	1.78	1.81	2.52	1.10	1.67	1.10	2.24	1.10	1.10
C6	97	120	1.37	2.49	1.10	1.75	1.37	2.25	1.10	1.48	1.64	1.93	0.83	1.56
C7	196.3	240	1.77	2.74	1.10	1.42	1.67	2.53	1.10	1.34	1.10	1.40	1.23	1.36

L.6.3 Results on Hopper and Turton's instances with rotation

Table L.8 reports the result of the variant of strip packing where rectangles can rotate with an angle of 90 degrees.

TAB. L.6 – Comparison on *beng*, *cgcut* and *gcut* instances.

Instance	Width	N	LB	IDW		HH+IDW		IDW+P		HH+IDW+P		HH		GRASP	
				best	aver.	best	aver.	best	aver.	best	aver.	best	aver.	best	aver.
beng01	25	20	30	30	30.7	30	30.8	30	30.5	30	30.4	30	30.5	30	30.0
beng02	25	40	57	58	58.0	58	58.0	58	58.1	58	58.0	57	57.0	57	57.0
beng03	25	60	84	85	85.3	84	84.6	84	85.2	84	84.5	85	85.0	84	84.0
beng04	25	80	107	108	108.9	108	108.1	108	108.9	107	107.9	108	108.6	107	107.0
beng05	25	100	134	135	135.8	134	134.0	134	135.2	134	134.0	134	134.0	134	134.0
beng06	40	40	36	36	36.0	36	36.0	36	36.0	36	36.0	36	36.0	36	36.0
beng07	40	80	67	68	68.0	67	67.8	68	68.0	67	67.3	67	67.9	67	67.0
beng08	40	120	101	102	102.4	101	101.0	102	102.4	101	101.0	101	101.0	101	101.0
beng09	40	160	126	126	126.5	126	126.0	126	126.6	126	126.0	126	126.0	126	126.0
beng10	40	200	156	156	157.2	156	156.0	156	157.2	156	156.0	156	156.0	156	156.0
cgcut01	10	16	23	23	23.0	23	23.0	23	23.0	23	23.0	23	23.0	23	23.0
cgcut02	70	23	63	65	65.5	65	65.0	65	65.5	65	65.0	65	65.0	65	65.0
cgcut03	70	62	636	675	680.2	663	667.8	671	677.3	662	667.9	662	665.5	661	661.0
gcut01	250	10	1016	1016	1016.0	1016	1016.0	1016	1016.0	1016	1016.0	1016	1016.0	1016	1016.0
gcut02	250	20	1133	1194	1214.4	1196	1205.8	1204	1209.0	1195	1203.9	1196	1206.1	1191	1191.0
gcut03	250	30	1803	1803	1808.8	1803	1803.0	1803	1803.0	1803	1803.0	1803	1803.0	1803	1803.0
gcut04	250	50	2934	3030	3100.5	3011	3028.9	3031	3075.6	3020	3032.0	3019	3025.8	3002	3002.0
gcut05	500	10	1172	1273	1273.0	1273	1273.0	1273	1273.0	1273	1273.0	1284	1287.0	1273	1273.0
gcut06	500	20	2514	2686	2706.9	2644	2659.6	2646	2668.2	2639	2651.1	2644	2654.2	2627	2627.0
gcut07	500	30	4641	4697	4769.4	4702	4703.5	4694	4737.1	4704	4710.1	4694	4705.0	4693	4693.0
gcut08	500	50	5703	5960	6061.3	5915	5957.5	5891	5977.6	5895	5947.7	5922	5951.5	5908	5912.2
gcut09	1000	10	2022	2317	2317.0	2317	2317.0	2317	2317.0	2317	2317.0	2317	2317.0	–	–
gcut10	1000	20	5356	5973	6049.4	5965	5983.1	5970	5990.9	5969	5972.0	5965	5995.7	–	–
gcut11	1000	30	6537	7066	7139.5	6980	7043.6	7043	7111.7	6966	6994.1	6973	7029.7	–	–
gcut12	1000	50	12522	14690	14762	14690	14690	14690	14690	14690	14690	14690	14690	–	–
gcut13	3000	32	4772	4998	5063.5	4944	4986.8	4994	5012.7	4914	4975.7	4945	5019.0	–	–
gcut09'	1000	10	2022	2241	2248.5	2254	2257.2	2241	2248.6	2241	2251.3	2290	2291.3	2256	2256.0
gcut10'	1000	20	5356	6399	6470.7	6399	6408.0	6399	6427.0	6422	6427.3	6402	6426.3	6393	6393.0
gcut11'	1000	30	6537	7736	7749.2	7736	7736.0	7736	7736.0	7736	7736.0	7736	7736.0	7736	7736.0
gcut12'	1000	50	12522	13172	13646	13172	13217	13172	13523	13172	13213	13172	13183	13172	13172
gcut13'	3000	32	4772	5055	5104.5	5037	5078.7	5061	5084.0	5028	5075.4	5028	5070.2	5009	5009.5

The first results were reported by Hopper and Turton (column `HT`) themselves in 2000 (Refs. [134, 135]). Note that `IDW` can find the 3 optima of the class 3 (i.e., an average deviation of 0.00) and one optimum of the class 5 (0.74) in 1000 seconds per run when it is manually tuned with the `BFs` greedy heuristic.

L.6.4 Synthesis

Iori's algorithm is generally the worst one on the tested instances. It is better than `IDW` only on `gcut06` and on the class 1 by Martello/Vigo. None of these results hold against `HH+IDW[+P]`. This provides an experimental evidence (by contradiction) that a good approach for handling strip packing should be based on the geometry of the layout.

Tab. L.7 – Comparison on the 500 instances proposed by Martello, Vigo, Berkey, Wang.

Class	Width	IDW	HH+IDW	HH+IDW+P	HH	GRASP
		best	best	best	best	best
01	10	0.67	0.64	0.65	0.72	**0.63**
02	30	0.58	0.25	0.16	0.34	**0.10**
03	40	2.16	**1.71**	1.72	1.72	1.73
04	100	3.47	2.44	2.40	2.60	**2.02**
05	100	2.20	2.08	2.09	**2.05**	**2.05**
06	300	4.86	3.71	3.72	3.80	**3.08**
07	100	1.12	1.13	1.13	1.13	**1.10**
08	100	4.19	3.81	3.68	3.74	**3.57**
09	100	0.07	0.07	0.07	0.07	0.07
10	100	3.12	2.80	**2.78**	2.80	2.93
Overall		2.24%	1.86%	1.84%	1.90%	**1.73%**

Lesh's algorithm behaves sometimes well but does not improve its solution a lot when spending more time (e.g., from 60 s to 3600 s). It is better than IDW only on gcut06 and on the class 2 by Martello/Vigo. This highlights the interest of a metaheuristic able to escape from local minima.

Burke's algorithm applied to Hopper and Turton's instances seems competitive with IDW only on large instances, while it is not competitive with HH+IDW[+P]. The same conclusion can be drawn when comparing Bortfeldt's approach, IDW and HH+IDW[+P] (on the problem with no allowed rotation of rectangles).

Thus, on strip packing with rectangles of fixed orientation, IDW generally outperforms the other competitors, but it is generally worse than (or equal to) GRASP (except for gcut09'), which highlights the interest of a sophisticated and randomized greedy heuristic (based on BF). If we compare GRASP and HH+IDW+P, both algorithms obtain similar best solutions. The difference between both is small. Both are close to each other on Hopper and Turton's instances, beng instances and cgcut instances. GRASP remains slightly better on gcut instances. Note that HH+IDW+P outperforms GRASP on the Hopper and Turton's class 7 (thanks to HH), on gcut09' (thanks to IDW), on gcut08 (thanks to IDW+P), and on classes 3 and 10 by Martello et al. (thanks to HH).

We must observe the very good results obtained by HH. However, the approach fails on certain instances that are generally easy for other algorithms, e.g., on Hopper and Turton's class 1 or on gcut05. This could come from a current lack of the approach that may be unable to reach any point in the search space. The hybridization between IDW and HH is particularly beneficial since the hybrid version is always competitive and outperforms sometimes IDW and HH individually (see for instance the class 6 by Hopper and Turton, beng03, beng04, gcut11, gcut13).

On the variant with non-fixed orientation of rectangles, it is difficult to evaluate our approach due to the lack of competitors. IDW seems to behave well. It is far above Hopper and Turton's al-

Tab. L.8 – Comparison on Hopper and Turton's instances with non-fixed orientation of rectangles.

Class	N	Opt.	IDW		HH+IDW		IDW+P		HH+IDW+P		HH		HT	Bortfeldt
			best	aver.	best	aver.	best	aver.	best	aver.	best	aver.	best	best
C1	16–17	20	0.00	1.17	0.00	1.50	1.67	1.67	0.00	0.17	1.67	2.50	4	1.70
C2	25	15	0.00	4.44	0.00	0.22	0.00	3.11	0.00	0.44	0.00	0.00	6	0.00
C3	28–29	30	3.33	3.33	2.22	3.00	2.22	3.00	2.22	2.44	2.22	3.00	5	2.22
C4	49	60	1.67	2.28	1.67	1.72	1.67	1.94	1.67	1.72	1.67	2.50	3	0.00
C5	73	90	1.48	2.15	1.11	1.19	1.48	2.07	1.11	1.15	1.11	1.22	3	0.00
C6	97	120	1.67	2.14	0.83	1.11	1.67	1.83	0.83	1.14	0.83	1.56	3	0.33
C7	196.3	240	2.08	2.54	0.83	1.19	1.80	2.28	0.83	1.19	0.69	1.33	4	0.33
Overall			1.46		0.95		1.50		0.95		1.17		4	0.654

gorithm but it is below Bortfeldt's algorithm on large instances. However, the difference between Bortfeldt's algorithm and `HH+IDW` is small. The good behavior of Bortfeldt's algorithm might be explained by its postprocessing phase performed on non-guillotine instances.

Interest of using maximal holes

Two points highlight the interest of the maximal holes : the good performance obtained by our approach and the slight advantage given to the repacking procedure.

First, it is worthwhile to underline that the good behavior of our method is mainly due to our incremental move. Our move makes an intensive use of `AddRectangle` and `RemoveRectangle` whose time complexities are closely related to the maximal holes.

Second, the interest of the repacking procedure is not significant. `IDW+P` almost always outperforms `IDW` if we compare the average bounds, but the comparison on the best bounds is not so clear. Also, the difference is even less significant when `HH` is used to compute the first layout. In other terms, `HH+IDW` and `HH+IDW+P` obtain very similar results. This provides an experimental evidence that maintaining the bottom-left property during the search is not crucial.

L.7 Conclusion

The contribution described in this paper is twofold. First, we have proposed incremental operators to maintain a set of maximal holes during the addition/removal of rectangles into/from a container for any 2D packing problem. We have suggested to relax the BL property which is respected by most of complete and incomplete algorithms. Second, we have designed a metaheuristic for handling 2D strip packing, endowed with an incremental move based on the geometry of the layout, and maintaining the set of maximal holes. In particular, this metaheuristic has no user-defined parameter and no greedy heuristic to be specified. This metaheuristic behaves well

on the tested benchmarks.

We have designed more efficient variants of this metaheuristic that start with a better first layout provided by a hyperheuristic (`HH`). These variants are really competitive with state-of-the-art algorithms. This hybridization is particularly beneficial since, although `HH` often shows a good performance, `HH` cannot efficiently handle certain instances on which `IDW` and `HH+IDW` behave well.

The good performance obtained by GRASP, Bortfeldt's algorithm, `HH` or our metaheuristic yields an experimental evidence that the best methods for handling strip packing :

– exploit the geometry of the layout,
– make use of several well-known greedy heuristics or of a sophisticated one.

It turns out that all the efficient approaches (except ours) implement improved greedy heuristics : Bortfeldt's algorithm uses the BFDH* heuristic while GRASP uses a very sophisticated BF-like heuristic. Also, the hyperheuristic approach tries to better exploit the best greedy heuristics known for strip packing. This suggests that our method could be improved by using more sophisticated greedy heuristics.

The greedy heuristic proposed by Chen and Huan in Ref. [56] will be studied in a future work. Although their heuristic does not handle strip packing but 2D rectangle packing (i.e., 2D bin packing with a unique bin), they obtain impressive results on Hopper and Turton's instances. One reason is that they determine in advance the height of the container, using the fact that these instances are zero-waste. However, the good performance could also be due to a great attention paid by their heuristic when selecting the next rectangle to be placed in the container. Such greedy algorithms can also benefit from maintaining the set of maximal holes.

Annexe M

Strip Packing Based on Local Search and a Randomized Best-Fit

Article [226] : paru au workshop BPPC du congrès CPAIOR, en 2008

Auteurs : Bertrand Neveu, Gilles Trombettoni

Abstract

We present an incomplete algorithm with no user-defined parameter for handling the strip-packing problem, a variant of the famous 2D bin-packing. The performance of our approach is due to several devices. We propose a move, based on the geometry of the layout, which is made incremental by maintaining the set of *maximal holes*. For escaping from local minima, the *Intensification Diversification Walk* (`ID Walk`) metaheuristic is used. `ID Walk` manages only one parameter that is automatically tuned by our tool.

We focus here on the greedy heuristics used to perform the moves and to compute the first layout before running the metaheuristic. In particular, we propose a variant of the well-known Best-fit (decreasing) (BF), called `RBF`, in which the criterion (i.e., height, width, perimeter, surface) changes every time a hole is selected. This simple way to randomize the most efficient greedy strategy is a key for obtaining good bounds while diversifying the layouts.

This paper provides an experimental evidence that a local search approach can be competitive with the best known incomplete algorithms.

M.1 The problem

Packing problems have numerous practical applications and the most interesting ones are all NP-hard, leading to the design of complete combinatorial algorithms, incomplete greedy heuristics, metaheuristics or genetic algorithms.

Packing problems consist in placing pieces in containers, such that the pieces do not intersect. Specific variants differ in the considered dimension (1D, 2D or 3D), in the type of pieces, or in additional constraints (e.g., for cutting applications, whether the (2D) container is guillotinable or not). The strip-packing is a variant of the 2D bin packing problem. A set of rectangles must be positioned in *one* container, called *strip*, which is a rectangular area. The strip has a fixed width dimension and a variable height. The goal is to place all the rectangles on the strip with no overlapping, using a minimum height of the container. We also study the variant that allows the rotation of rectangles with an angle of 90 degrees. The reader will refer to [227] and [134] for a description of incomplete methods handling strip-packing.

M.2 Our approach

We have designed an incomplete algorithm divided into two main phases :

1. a greedy heuristic produces a first layout of the rectangles on the strip,
2. a local search procedure, called `ID Walk`, driven by an automatic tuning procedure, repairs the first solution.

The efficiency of our method reaches that of state-of-the-art incomplete algorithms. To our knowledge, no other existing local search method has proven such a performance.

Incremental operations and maximal holes

To limit the combinatorial explosion, most algorithms maintain the *Bottom-Left (BL) property, that is, a layout where the bottom and left segments of every rectangle touch the container or another rectangle*. (Some algorithms maintain only a *bottom property* where only the bottom segment touchs. We refer by B[L] property both properties.) First, the B[L] property lowers the number of possible locations for rectangles. Second, it can be proven that any solution of a 2D packing problem can be transformed into a solution respecting the BL property with a simple repacking procedure. However, maintaining the B[L] property after a rectangle removal is not local to the removed rectangle and to its neighbors, but modifies the whole layout in the worst case (e.g., when the rectangle placed on the bottom-left corner of the container is removed).

We have proposed in [228] and [227] an alternative approach that does not respect the B[L] property but maintains instead the set of *maximal holes*.

Definition 24 *(Maximal hole) Let us consider a container C partially filled with a set S of rectangles. A maximal hole H (w.r.t. C and S) is an empty rectangular surface in C such that :*

– *H does not intersect any rectangle in S (i.e., H is a "hole" in the container),*
– *H is maximal, i.e., there is no hole H' such that H is included inside H'*[1].

Fig. M.1 shows three examples with resp. 2, 2 and 4 maximal holes (from left to right). The maximal hole in grey corresponds to the most bottom-left one.

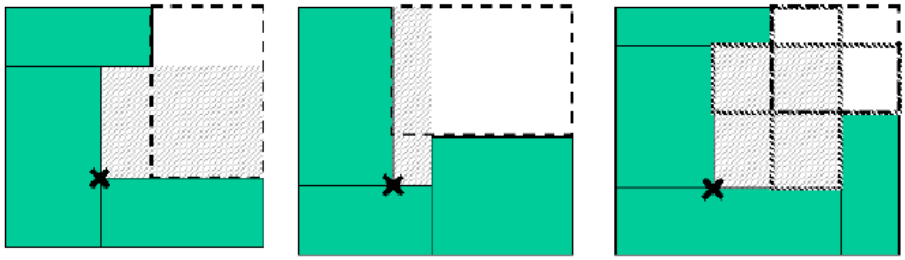


FIG. M.1 – Examples of maximal holes

It appears that the notion of maximal holes, designed by *maximal areas*, has been introduced independently by El Hayek et al. in [77] for 2D bin packing problems. They have proven that the maximum number of maximal areas managed during a packing procedure is $O(n^2)$, where n is the number of rectangles to be placed. We have detailed in [227] atomic operations that can be performed in constant time on rectangles and maximal holes. **Minus**(H, R) adds/modifies at most 4 maximal holes when one rectangle R is added in a container. A second operation **Plus**(H_1, H_2) holds between two rectangular maximal holes. If H_1 and H_2 intersect or are contiguous, `Plus` returns at most two new maximal holes.

Based on these operations, we have designed two procedures `AddRectangle` and `RemoveRectangle` for adding/removing one rectangle in/from a container [227]. They can be used in most algorithms handling 2D packing problems.

Incremental move for strip packing

To handle strip packing, our metaheuristic uses a move based on the "geometry" of the rectangles on the strip. This move makes an intensive use of the incremental `AddRectangle` and `RemoveRectangle` procedures. It removes one rectangle R on the top of the layout and places it inside the strip. More precisely, the new location for R is a maximal hole or a placed rectangle. The rectangles of the layout that intersect R in its new location are placed again on the strip with a greedy heuristic such as Best-Fit Decreasing (`BF`).

We report in Table 1 of Ref. [227] statistics that show that, in practice, during the local search, the number of possible locations for a rectangle grows in a linear way w.r.t the number of rectangles, and that the number of displaced rectangles in one move remains always very small on average.

[1]This property implies that the bottom and left segments of H touch the container or rectangles in S.

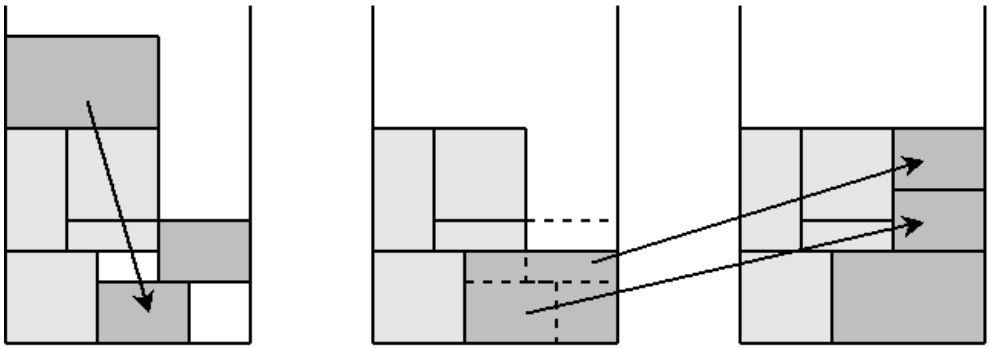


FIG. M.2 – One complete move

Selected local search procedure

We have selected a fine two-dimensional objective function equal to $W \times H + n$, where W is the width of the strip, H is the ordinate of the top side of a highest rectangle on the strip *minus one*, and n is the number of units filled by rectangles on the highest line of the strip.

We have implemented our incomplete algorithm in the `INCOP C++` library developed by the first author (Ref. [223]). This algorithm could be specialized with number of greedy heuristics and metaheuristics. We have tried the main metaheuristics available in `INCOP` and have selected `ID Walk` (i.e., the `ID(best)` variant) for its efficiency and its simplicity [231]. In particular, the automatic tuning procedure provided by `INCOP` is more robust when it tunes only one parameter. A description of the automatic tuning procedure can be found in Refs. [231] and [227]. Note that the tuning time represents about 30% of the global time. `ID(best)` is a *candidate list strategy* that uses one parameter `MaxNeighbors` to perform one move from a configuration x to a configuration x', as follows :

1. `ID(best)` picks randomly neighbor candidates one by one and evaluates them. The *first* neighbor x' with a cost better than or equal to the cost of x is accepted.
2. If `MaxNeighbors` neighbors have been rejected, then the *best* neighbor among them (with a cost strictly worse than the cost of x) is selected.

`ID(best)` has a common behavior with a variant of tabu search once all the candidates have been rejected (item 2 above). However, the differences are the absence of tabu list and another policy for selecting a neighbor x' (step 1 above).

Finally, during the local search, a *repacking procedure* is launched occasionally to recover the BL property. It is run only when the latest move leaves the current solution at a significantly worse cost, i.e., an increase of 1 of the upper line of the layout. The repacking procedure is a variant of the famous Bottom-Left-Fill greedy heuristic [55]. It performs a simple loop on the rectangles in a bottom-left order and repacks them on the current layout. The idea behind our metaheuristic was to *not* recover the BL property every time a rectangle is removed during the moves performed by local search. Instead, the set of maximal holes is incrementally recomputed at every move and the repacking procedure is called so occasionally that the time spent is dominated by the

local search. The impact of the repacking procedure on performance is positive but slight.

Overall, used with `ID Walk`, its automatically tuned parameter, and with the repacking procedure presented above, our metaheuristic has simply *no* user-defined parameter.

M.3 Greedy heuristics

Our latest advances concern the choice of greedy heuristics. It is used to compute a first layout. Also, this allows our move to put again displaced rectangles into the strip. The impact of the greedy heuristics on the efficiency is significant.

Our algorithm works with a simple version of the Best-Fit Decreasing greedy heuristic (BF) [46] :

The rectangles are initially sorted in decreasing order, according to one of the following criteria : largest Width first (w), largest Height first (h), largest Perimeter first (p), largest Surface first (s). A more original criterion, called largest Max first (m), selects the rectangle one dimension of which (width *or* height) is the largest among all the dimensions of the others. The criterion m stastistically implies to place first the rectangles that are more difficult to place. In other words, like the criterion p, this strategy differs the placement of the rectangles that are small in both dimensions and are thus easy to place.

Then, following a bottom-left order, `BF` tries to fill every hole. (A significant property is that only the holes on the top of the current layout - having an infinite height - must be considered.) At each iteration, the lists are traversed to place one of the rectangles on the strip, as follows :

– The list is traversed a first time to select the first rectangle (if any) that *fits* exactly the width of the hole or one of the two neighbor heights.
– If no such rectangle is selected, the list is traversed a second time to select the first rectangle (if any) with a width less than that of the hole.

If no rectangle can fit inside the hole, this hole will never been filled and the next hole is considered.

M.3.1 A simple randomized Best Fit

We introduce in this paper a Randomized variant of `BF` denoted by `RBF`. `RBF` accepts as parameter a list of sorting criteria. For instance, `RBF(w,p)` admits the two criteria *largest Width first* and *largest Perimeter first*. `RBF` follows the scheme of `BF` except that the chosen criterion changes from an iteration (filling a hole) to another, the criterion being picked randomly in the list. Although more studies must be performed to validate it, `RBF` is simple and intensifies well the search. Also, successive calls to `RBF` produce different layouts.

M.3.2 Initial layout

The first layout is simply obtained by successive calls to `BF` or `RBF`, following two possible strategies. The first one simply calls `RBF` iteratively until the time limit is reached.

The second strategy, designed here by `RR`, accepts as parameter a list G of greedy heuristics. The method applies iteratively one greedy heuristic picked in G (in a round robin - RR - manner) until the time limit is reached. The list G can include `BF` and `RBF` greedy heuristics together. In the experiments reported hereafter, the greedy heuristic designed by `RR10` is a round-robin with 5 different RBFs and 5 BFs.

M.3.3 Greedy heuristics used during the local search

Experiments (not reported here) have shown that several choices of greedy heuristics give similar results when they put again displaced rectangles into the strip. The reason is probably that only a small number of rectangles are displaced at every move. In our latest version, we use `RBF(p,s,w,h)`.

M.4 Experiments

We have performed experiments on five series enclosing 552 benchmarks. The 21 zero-waste instances by Hopper and Turton [135], the 13 *gcut* instances by Beasley [25], the 3 *cgcut* instances, the 10 *beng* instances by Bengtsson [26], and the 500 instances proposed by Martello and Vigo [193], Berkey and Wang [30].

The hybrid tabu/genetic algorithm is run by Iori during 300 seconds on a `Pentium III 800 Mhz` (Ref. [142]). The BLD* algorithm is run by Lesh et al on a `Pentium 2 GHz` (Ref. [185]). The two presented results correspond to time limits of respectively 60 seconds and 3600 seconds. The results presented for Burke et al. correspond to their BF heuristic enhanced with simulated annealing. They run their heuristic 10 times with a time limit of 60 seconds per run on a `Pentium IV 2 GHz`. Bortfeldt's genetic algorithm is run 10 times on every instance with an average time per run of 160 seconds on a `Pentium 2 GHz`. The GRASP algorithm (Ref. [6]) is the best known approach to handle strip packing. It is also run 10 times on every instance with a time limit of 60 seconds on `Pentium IV Mobile 2 GHz`. The hyperheuristic by Araya et al. [8] runs 10 times with a time limit of 100 seconds per run on a `Pentium IV 3 GHz`.

Our metaheuristic (`IDW`) is also run 10 times with a time limit of 100 seconds per trial on a `Pentium IV 3 GHz` : the first 10 seconds are used by the greedy heuristic `RR10` for computing the first layout while 90 seconds are spent in the local search. When greedy heuristics (`RR10`, `RR4`, `RBF`) are run alone, they perform 10 trials on a `Pentium IV 3 GHz`. Every trial runs the greedy heuristic iteratively until the time limit of 100 seconds is reached.

The cells in tables M.1, M.2 and M.4 report the average percentage deviation from optimum.

M.4.1 Synthesis

Iori's, Lesh's, Burke's and Bortfeld's algorithms are not competitive with the others. `HH` gives very good results but is generally slightly outperformed by GRASP. Our metaheuristic is competitive with the state-of-the-art GRASP. However, our approach is simpler and can handle the variant

allowing the rotation of rectangles with an angle of 90 degrees. On this variant, it is difficult to evaluate our approach due to the lack of competitors. `IDW` seems to behave well while remaining behind Bortfeld's algorithm.

As a conclusion, most of the underlying concepts can be applied to several 2D packing problems.

Acknowledgments

Special thanks to Ignacio Araya and Maria-Cristina Riff for the collaboration on previous works about strip packing.

Tab. M.1 – Comparison on the zero-waste instances by Hopper and Turton. N is the number of rectangles and Opt. is the optimum.

Class	Width	N	Opt.	`IDW`	`RR10`	Iori	Lesh 3600	Burke	Bortfeldt	HH	GRASP
C1	20	1617	20	**0.00**	1.59	1.59	-	**0.00**	1.59	1.59	**0.00**
C2	40	25	15	**0.00**	**0.00**	2.08	-	6.25	2.08	**0.00**	**0.00**
C3	60	2829	30	2.15	2.15	2.15	-	3.23	3.23	2.15	**1.08**
C4	60	49	60	**1.09**	1.64	4.75	-	1.64	2.70	2.70	1.64
C5	60	73	90	**0.73**	1.10	3.92	2.17	1.46	1.46	1.10	1.10
C6	80	97	120	**0.83**	**0.83**	4.00	1.64	1.37	1.64	1.64	**0.83**
C7	160	196.3	240	**0.41**	0.69	-	-	1.77	1.23	1.10	1.23
# optimal solutions				**9/21**	6/21	5/18	0/6	3/21	4/21	6/21	8/21

Tab. M.2 – Comparison on Hopper and Turton's instances with non-fixed orientation of rectangles. The column HT report the results obtained by the authors themselves. GRASP cannot handle this variant of strip-packing.

Class	N	Opt.	`IDW`		`RR10`		`HH`		HT	Bortfeld
			best	aver.	best	aver.	best	aver.	best	best
C1	1617	20	**0.00**	**0.00**	**0.00**	**0.00**	1.67	2.50	4	1.70
C2	25	15	0.00	0.00	0.00	0.00	0.00	0.00	6	0.00
C3	2829	30	**2.22**	3.11	3.33	3.33	**2.22**	3.00	5	**2.22**
C4	49	60	1.11	1.61	1.67	1.67	1.67	2.50	3	**0.00**
C5	73	90	0.74	1.07	1.11	1.11	1.11	1.22	3	**0.00**
C6	97	120	0.83	0.83	0.83	0.83	0.83	1.56	3	**0.28**
C7	196-197	240	0.42	0.57	0.42	0.75	0.69	1.33	4	**0.28**
Overall			0.76		1.05		1.17		4	**0.64**

TAB. M.3 – Comparison on *beng*, *cgcut* and *gcut* instances. The benchmarks `gcut09'...gcut13'` (Ref. [6]) are variants of the benchmarks `gcut09...gcut13` in which the rectangles have been rotated with an angle of 90 degrees (i.e., the width and the height of every rectangle have been exchanged). The column LB yields a Lower Bound of the optimum (which is not necessarily reached). The following columns report the bound of the best solution computed by the corresponding algorithm.

Instance	Width	N	LB	IDW	RR10	Iori	Lesh 60	Lesh 3600	HH	GRASP
beng01	25	20	30	30	31	31	–	–	30	30
beng02	25	40	57	57	57	58	–	–	58	57
beng03	25	60	84	84	84	86	–	–	85	84
beng04	25	80	107	107	107	110	–	–	108	107
beng05	25	100	134	134	134	136	–	–	134	134
beng06	40	40	36	36	36	37	–	–	36	36
beng07	40	80	67	67	67	69	–	–	67	67
beng08	40	120	101	101	101	–	–	–	101	101
beng09	40	160	126	126	126	–	–	–	126	126
beng10	40	200	156	156	156	–	–	–	156	156
cgcut01	10	16	23	23	65	23	–	–	23	23
cgcut02	70	23	63	65	65	65	–	–	65	65
cgcut03	70	62	636	**658**	663	676	–	–	662	661
gcut01	250	10	1016	1016	1016	1016	1016	1016	1016	1016
gcut02	250	20	1133	**1187**	1204	1207	1211	1195	1196	1191
gcut03	250	30	1803	1803	1803	1803	1803	1803	1803	1803
gcut04	250	50	2934	3026	3025	3130	3072	3054	3019	**3002**
gcut05	500	10	1172	1273	1273	1273	1273	1273	1273	1273
gcut06	500	20	2514	2639	2637	2675	2682	2656	2644	**2627**
gcut07	500	30	4641	4701	4701	4758	4795	4754	4694	**4693**
gcut08	500	50	5703	5913	5935	6240	6181	6081	5922	**5908**
gcut09'	1000	10	2022	**2241**	2303	–	–	–	2290	2256
gcut10'	1000	20	5356	6399	6413	–	–	–	6402	**6393**
gcut11'	1000	30	6537	7736	7736	–	–	–	7736	7736
gcut12'	1000	50	12522	13184	13174	–	–	–	**13172**	**13172**
gcut13'	3000	32	4772	**5007**	5041	–	–	–	5028	5009

Tab. M.4 – Comparison on the 500 instances proposed by Martello, Vigo, Berkey, Wang. The cells include the percentage deviation between the (not necessarily reached) lower bound and a value v : v is an average value over the 50 best solution costs obtained for the 50 instances in a given class.

Class	Width	IDW	RR10	Iori	Lesh 60	Lesh 3600	Bortfeldt	HH	GRASP
01	10	**0.64**	0.77	**0.64**	0.81	0.68	0.75	0.72	**0.63**
02	30	**0.10**	0.35	1.78	1.12	0.42	0.88	0.34	**0.10**
03	40	1.82	2.02	3.05	2.71	2.23	2.52	**1.72**	**1.73**
04	100	**1.80**	2.03	5.08	4.41	3.54	3.19	2.60	2.02
05	100	2.15	2.24	3.15	2.85	2.43	2.59	**2.05**	**2.05**
06	300	3.41	3.72	5.99	6.45	5.13	4.96	3.80	**3.08**
07	100	1.19	1.27	1.16	1.17	1.12	1.19	1.13	**1.10**
08	100	3.77	4.03	6.16	5.99	4.93	3.85	3.74	**3.57**
09	100	0.07	0.13	0.07	0.07	0.07	0.07	0.07	0.07
10	100	2.91	3.13	4.67	4.11	3.48	3.05	2.93	**2.80**
Overall		1.79%	1.97%	3.17%	2.97%	2.40%	2.31%	1.90%	**1.73%**

TAB. M.5 – Comparison between our BF-based greedy heuristics on a sample of 62 benchmarks. All the results correspond to 10 trials of 100 seconds each, on a same computer `Pentium IV 3 GHz`. The instances named C201 and the following ones correspond to first subclasses of the 500 instances presented above. `RR4` is our round-robin greedy heuristics with the four standard BFs : `BF(h)`, `BF(w)`, `BF(s)`, `BF(p)`. `RBF` generally outperforms `RR4`, justifying the interest of our simple randomization. `RR10` is slightly better than `RBF`.

Instance	RR10		RR4		RBF		Instance	RR10		RR4		RBF	
	best	average	best	average	best	average	best	best	average	best	aver.	best	aver.
HT C1-P1	20	20	21	21	20	20	beng1	31	31	31	31	31	31
C1-P2	21	21	21	21	21	21	beng2	57	57.3	57	57.9	57	57.3
C1-P3	20	20	21	21	20	20	beng6	36	36.2	37	37	36	36.1
C2-P1	15	15	16	16	15	15	beng7	67	67	67	67	67	67.3
C2-P2	15	15.3	16	16	15	15.3	C201	22	22	23	23	22	22
C2-P3	15	15	15	15	15	15	C311	233	233.9	234	234	233	233.8
C3-P1	31	31	31	31	31	31	C341	727	728.9	727	729.4	726	729.9
C3-P2	31	31	31	31	31	31	C401	72	72.6	73	73	72	72
C3-P3	30	30	31	31	30	30	C411	92	92.5	93	93	92	92.2
C4-P1	61	61	61	61.4	61	61	C421	220	220.4	220	220.5	219	220.5
C4-P2	61	61	61	61.5	61	61	C431	246	246.7	246	246.5	246	246.7
C4-P3	61	61	61	61	61	61	C441	290	290.3	290	290.9	290	290.8
C5-P1	91	91	91	91	91	91	C511	744	746.1	754	754	744	745.8
C5-P2	91	91	91	91	91	91	C521	1864	1865	1864	1864.4	1874	1879.1
C5-P3	91	91	91	91	91	91	C541	2320	2330	2320	2328.8	2367	2374.4
C6-P1	121	121	121	121	121	121	C601	188	188	192	192	188	188
C6-P2	121	121	121	121	121	121	C611	240	241.7	242	242.2	240	241
C6-P3	121	121	121	121	121	121	C621	590	591.9	590	592.2	592	594.6
C7-P1	242	242	242	242.2	242	242.3	C631	654	658.7	657	658.4	657	660
C7-P2	241	241.7	241	242	241	241.9	C641	772	775.4	774	776.2	775	776.8
C7-P3	241	241.9	242	242	242	242							
cgcut03	663	663	667	670.8			C731	1997	1997	1997	1997	2064	2064
gcut2	1204	1204	1204	1204	1204	1204	C741	2664	2664	2664	2664	2708	2708
gcut4	3025	3039	3068	3068	3032	3052.9	C801	500	501.5	511	511	500	501.4
gcut5	1273	1273	1295	1295	1273	1273	C811	1014	1018.1	1018	1018.6	1008	1013.3
gcut6	2637	2643.9	2695	2695	2644	2644	C821	1517	1523.2	1510	1523.3	1505	1525.1
gcut7	4701	4712.1	4745	4745	4701	4701.8	C831	1898	1907.9	1883	1904	1903	1912.9
gcut8	5935	5950.9	5940	5942.2	6010	6042.4	C841	2333	2343.5	2326	2338.3	2352	2361.6
gcut9'	2303	2303	2303	2303	2303	2303	C911	1915	1915	1915	1915	1940	1940
gcut10	6413	6413	6434	6434	6393	6393	C1011	742	742.6	749	749	743	743.9
gcut12'	13174	13197.9	13312	13323.4	13404	13495.8	C1021	1064	1072.2	1066	1071.5	1071	1074
gcut13'	5041	5058.9	5077	5084.4	4996	5031.1							

Bibliographie

[1] A. Aggoun and N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling and Placement Problems. *Mathematical and Computer Modelling*, 17 :57–73, 1993.

[2] S. Ait Aoudia. *Modélisation géométrique par contraintes : quelques méthodes de résolution.* Ph.d. thesis, Ecoles des Mines de Saint-Etienne, 1994.

[3] S. Ait-Aoudia, R. Jegou, and D. Michelucci. Reduction of Constraint Systems. In *Compugraphic*, 1993.

[4] B. Aldefeld. Variations of Geometries Based on a Geometric-reasoning Method. *Computer Aided Design*, 20(3) :117–126, 1988.

[5] E. Allgower and K. Georg. *Introduction to Numerical Continuation Methods.* SIAM, Classics in Applied Mathematics Series, No. 45, 2003.

[6] R. Alvarez-Valdes, F. Parreño, and J.M. Tamarit. Reactive GRASP for the Strip Packing Problem. *Computers and Operations Research*, 35 :1065–1083, 2008.

[7] I. Araya. *Exploiting Common Subexpressions and Monotonicity of Functions for Filtering Algorithms over Intervals.* PhD thesis, University of Nice–Sophia, 2010.

[8] I. Araya, B. Neveu, and M.-C. Riff. An Efficient Hyperheuristic for Strip Packing Problems. In *Adaptive and Multilevel Metaheuristics*, volume 136 of *Studies on Computational Intelligence*. Springer, 2008.

[9] I. Araya, B. Neveu, and G. Trombettoni. Exploiting Common Subexpressions in Numerical CSPs. In *Proc. CP, Constraint Programming, LNCS 5202*, pages 342–357, 2008.

[10] I. Araya, B. Neveu, and G. Trombettoni. A New Monotonicity-Based Interval Extension Using Occurrence Grouping. In *IntCP, int. WS on interval analysis, constraint propagation, applications, at CP conference*, pages 51–64, 2009.

[11] I. Araya, B. Neveu, and G. Trombettoni. An Interval Constraint Propagation Algorithm Exploiting Monotonicity. In *IntCP, int. WS on interval analysis, constraint propagation, applications, at CP conference*, pages 65–83, 2009.

[12] I. Araya, G. Trombettoni, and B. Neveu. Filtering Numerical CSPs Using Well-Constrained Subsystems. In *Proc. CP, Constraint Programming, LNCS 5732*, pages 158–172, 2009.

[13] I. Araya, G. Trombettoni, and B. Neveu. Exploiting Monotonicity in Interval Constraint Propagation. In *Proc. AAAI (to appear).* AAAI Press, 2010.

[14] Archimède. On the measurement of the circle. In *Thomas L. Heath (ed.), The Works of Archimedes, Cambridge University Press, 1897; Dover edition, 1953*, pages 91–98, Before 212 BC !

[15] D. Avis and K. Fukuda. Reverse Search Enumeration. *Discrete Applied Mathematics*, 6 :21–46, 1996.

[16] A. Baharev, T. Achterberg, and E Rév. Computation of an Extractive Distillation Column with Affine Arithmetic. *AIChE Journal*, 55(7) :1695–1704, 2009.

[17] A. Baharev and E. Rév. A Complete Nonlinear System Solver Using Affine Arithmetic. In *IntCP, int. WS on interval analysis, constraint propagation, applications, at CP conference*, pages 17–33, 2009.

[18] A. Baharev and E. Rev. `http://reliablecomputing.eu/benchmarks.html`, 2009.

[19] J.-C. Bajard and J.-M. Muller. *Calcul et arithmétique des ordinateurs*. Hermès, Lavoisier, 2004.

[20] B.S. Baker, E.G. Coffman, and R.L. Rivest. Orthogonal Packings in 2D. *SIAM Journal on Computing*, 9 :846–855, 1980.

[21] R. Barták and R. Erben. A new Algorithm for Singleton Arc Consistency. In *Proc. FLAIRS*, 2004.

[22] H. Batnini. *Contraintes Globales et Techniques de Résolution pour les CSPs Continus*. PhD thesis, Université de Nice-Sophia Antipolis, november 2005.

[23] H. Batnini, C. Michel, and M. Rueher. Mind the Gaps : A New Splitting Strategy for Consistency Techniques. In *Proc. CP, Constraint Programming, LNCS 3709*, pages 77–91, 2005.

[24] P.-L. Bazin. A Parametric Scene Reduction Algorithm from Geometric Relations. In *Vision Geometry IX, SPIE*, 2000.

[25] J.E. Beasley. Algorithms for Unconstrained Two-Dimensional Guillotine Cutting. *J. of the operational research society*, 33 :49–64, 1985.

[26] B.E. Bengtsson. Packing Rectangular Pieces – A Heuristic Approach. *The computer journal*, 25 :353–357, 1982.

[27] F. Benhamou and F. Goualard. Universally Quantified Interval Constraints. In *Proc. CP, Constraint Programming, LNCS 1894*, pages 67–82, 2004.

[28] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proc. ICLP, int. conference on logic programming*, pages 230–244, 1999.

[29] F. Benhamou and L. Granvilliers. *Continuous and Interval Constraints*. Elsevier, 2006. chapitre 16 du livre : Handbook of Constraint Programming.

[30] J.O. Berkey and P.Y. Wang. Two-Dimensional Finite Bin Packing Algorithms. *Journal of the operational research society*, 38 :423–429, 1987.

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

[32] C. Bessière. Random Uniform CSP Generators. 2002. `http://www.lirmm.fr/~bessiere/generator.html`.

[33] C. Bliek. Generalizing Dynamic and Partial Order Backtracking. In *Proc. AAAI*, pages 319–325, 1998.

[34] C. Bliek, B. Neveu, and G. Trombettoni. Using Graph Decomposition for Solving Continuous CSPs. In *Proc. CP, Constraint Programming, LNCS 1520*, pages 102–116, 1998.

[35] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer Verlag, New York, Inc., 1998.

[36] L. Blum, M. Shub, and M. Smale. On a Theory of Computation over the Real Numbers ; NP-completeness, Recursive Functions and Universal Machines. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 387–397, 1988.

[37] D. Bondyfalat and S. Bougnoux. Imposing Euclidean Constraints During Self-Calibration Processes. In *SMILE workshop on 3D structure from multiple images of large-scale environments, LNCS 1506*, pages 224–235, 1998.

[38] D. Bondyfalat, B. Mourrain, and T. Papadopoulo. An Application of Automatic Theorem Proving in Computer Vision. In *Proc. ADG, WS on Automatic Deduction in Geometry*, pages 207–231, 1999.

[39] A. Borning. ThingLab *: A Constraint–Oriented Simulation Laboratory*. PhD thesis, Stanford University, 1979.

[40] A. Bortfeldt. A Genetic Algorithm for the Two-Dimensional Strip Packing Problem with Rectangular Pieces. *European J. of Operational Research*, 172 :814–837, 2006.

[41] W. Bouma, I. Fudos, C. M. Hoffmann, J. Cai, and R. Paige. Geometric Constraint Solver. *Computer Aided Design*, 27(6) :487–501, 1995.

[42] S. Bouveret, S. de Givry, F. Heras, J. Larrosa, E. Rollon, M. Sanchez, T. Schiex, G. Verfaillie, and M. Zytnicki. Max-CSP Competition 2007 : Toolbar/Toulbar2 Solver Brief Description. In *Proc. International CSP Solver Competition*, pages 19–21, 2008.

[43] D. P. Brown. *Calculus and Mathematica*. Addison-Wesley, 1991.

[44] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3) :293–318, 1992.

[45] B. Buchberger. Gröbner Bases : an Algorithmic Method in Polynomial Ideal Theory. *Multidimensional Systems Theory*, pages 184–232, 1985.

[46] E. Burke, G. Kendall, and G. Whitwell. A New Placement Heuristic for the Orthogonal Stock Cutting Problem. *Operations Research*, 52 :697–707, 2004.

[47] E. Burke, G. Kendall, and G. Whitwell. Metaheuristic Enhancements of the Best-Fit Heuristic for the Orthogonal Stock Cutting Problem. *Submitted to INFORMS*, 2006.

[48] M. Ceberio and L. Granvilliers. Solving Nonlinear Equations by Abstraction, Gaussian Elimination, and Interval Methods. In *Workshop FROCOS*, 2002.

[49] G. Chabert. *Techniques d'intervalles pour la résolution de systèmes d'intervalles*. PhD thesis, Université de Nice–Sophia, 2007.

[50] G. Chabert. `www.ibex-lib.org`, 2009.

[51] G. Chabert and A. Goldsztejn. Extension of the Hansen-Bliek Method to Right-Quantified Linear Systems. *Reliable Computing*, 13(4) :325–349, 2007.

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

[53] G. Chabert, G. Trombettoni, and B. Neveu. Box-Set Consistency for Interval-based Constraint Problems. In *SAC, Symposium on Applied Computing, ACM*, pages 1439–1443, 2005.

[54] G. Chabert, G. Trombettoni, and B. Neveu. IGC : une nouvelle consistance partielle pour les CSP continus. In *JFPC, journées francophones de programmation par contraintes*, pages 199–210, 2005.

[55] B. Chazelle. The Bottom Left Bin Packing Heuristic : An Efficient Implementation. *IEEE Transactions on Computers*, 32 :697–707, 1983.

[56] D. Chen and W. Huang. A Novel Quasi-Human Heuristic Algorithm for Two-Dimensional Rectangle Packing Problem. *International Journal of Computer Science and Network Security*, 6(12) :115–120, 2006.

[57] S. C. Chou. *Mechanical Theorem Proving*. Reidel Publishing Co., 1988.

[58] S.-C. Chou, X.-S. Gao, and J.-Z. Zhang. A Deductive Database Approach To Automated Geometry Theorem Proving and Discovering. *Journal of Automated Reasoning*, 25(3) :219–246, 2000.

[59] H. Collavizza, F. Delobel, and M. Rueher. Extending Consistent Domains of Numeric CSP. In *Proc. IJCAI*, pages 406–413, 1999.

[60] D. T. Connolly. An Improved Annealing Scheme for the QAP. *European Journal of Operational Research*, 46 :93–100, 1990.

[61] H. Crapo. On the Generic Rigidity of Plane Frameworks. Technical Report 1278, INRIA, 1990.

[62] J. Cruz and P. Barahona. Global Hull Consistency with Local Search for Continuous Constraint Solving. In *Proc. EPIA, LNAI 2258*, pages 349–362, 2001.

[63] S. de Givry, G. Verfaillie, and T. Schiex. Bounding the optimum of constraint optimization problems. In *Proc. CP97*, number 1330 in LNCS, 1997.

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

[65] R. Dechter. Enhancement Schemes for Constraint Processing : Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41(3) :273–312, 1990.

[66] T. Dimitriou. Characterizing the Space of Cliques in Random Graphs Using "Go With the Winners". In *Proc. AMAI*, 2002.

[67] T. Dimitriou and R. Impagliazzo. Towards an Analysis of Local Optimization Algorithms. In *Proc. STOC*, 1996.

[68] T. Dimitriou and R. Impagliazzo. Go With the Winners for Graph Bisection. In *Proc. SODA*, pages 510–520, 1998.

[69] F. Domes. GloptLab - a Configurable Framework for Solving Continuous, Algebraic CSPs. In *IntCP, int. WS on interval analysis, constraint propagation, applications, at CP conference*, pages 1–16, 2009.

[70] F. Domes and A. Neumaier. Quadratic Constraint Propagation. *to appear in Constraints*, 2009.

[71] R. Dorne and J.-K. Hao. Tabu Search for Graph Coloring, T-colorings and Set T-colorings. In *Meta-heuristics : Advances and Trends in Local Search Paradigms for Optimization*, pages 77–92. Kluwer Academic Publishers, 1998.

[72] S. Ducasse, M. Blay-Fornarino, and A.-M. Pinna-Déry. A Reflective Model for First Class Dependencies. In *OOPSLA, Int. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 265–280, 1995.

[73] J.-F. Dufourd, P. Mathis, and P. Schreck. Geometric Construction by Assembling Solved Subfigures. *Artificial Intelligence*, 99(1) :73–119, 1998.

[74] A. L. Dulmage and N. S. Mendelsohn. Covering of Bipartite Graphs. *Canad. J. Math.*, 10 :517–534, 1958.

[75] C. Durand. *Symbolic and Numerical Techniques For Constraint Solving*. PhD thesis, Purdue University, 1998.

[76] A. Eisenblätter and A. Koster. FAP web - A website about Frequency Assignment Problems. 2000. `http://fap.zib.de/`.

[77] J. El Hayek, A. Moukrim, and S. Negre. New Resolution Algorithm and Pretreatments for the Two-Dimensional Bin-packing Problem. *Computers & Operations Research*, 35(10) :3184–3201, 2008.

[78] E. Eremchenko and A. Ershov. Two New Decomposition Techniques in Geometric Constraint Solving. Research report Preprint number 11, LEDAS Company, 2004.

[79] C. Essert, P. Schreck, and J.-F. Dufourd. Sketch-based Pruning of a Solution Space within a Formal Geometric Constraint Solver. *Artificial Intelligence*, 124 :139–159, 2000.

[80] F. Fages and Julien Martin. From Rules to Constraint Programs with the Rules2CP Modelling Language. In *Recent Advances in Constraints, Springer-Verlag LNAI 5655*, pages 66–83, 2009.

[81] B. Faltings. Arc-consistency for Continuous Variables. *Artificial Intelligence*, 65, 1994.

[82] T.A. Feo and M.G.C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6 :109–133, 1995.

[83] P. Flajolet, P. Sipala, and J.-M. Steyaert. Analytic Variations on the Common Subexpression Problem. In *Proc. ALP, LNCS 443*, pages 220–334, 1990.

[84] S. Foufou, D. Michelucci, and J.-P. Jurzak. Numerical Decomposition of Geometric Constraints. In *Proc. SPM, Solid and Physical Modeling*, pages 143–151, 2005.

[85] B. Freeman-Benson, J. Maloney, and A. Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1) :54–63, 1990.

[86] C. Freuder, E. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, 29(1) :24–32, 1982.

[87] I. Fudos. *Constraint Solving for Computer Aided Design*. PhD thesis, Purdue University, 1995.

[88] I. Fudos and C. M. Hoffmann. Correctness Proof of a Geometric Constraint Solver. *International Journal of Computational Geometry and Applications*, 6 :405–420, 1996.

[89] I. Fudos and C. M. Hoffmann. A Graph-Constructive Approach to Solving Systems of Geometric Constraints. *ACM Transactions on Graphics*, 16(2) :179–216, 1997.

[90] C. Fuenfzig, D. Michelucci, and S. Foufou. Nonlinear Systems Solver in Floating-Point Arithmetic using LP Reduction. In *SIAM/ACM Joint Conference on Geometric and Physical Modeling*, 2009.

[91] P. Galinier and J.-K. Hao. Hybrid Evolutionary Algorithms for Graph Coloring. *Journal of Combinatorial Optimization*, 3(4) :379–397, 1999.

[92] M. Gangnet and B. Rosenberg. Constraint Programming and Graph Algorithms. In *Int. Symposium on Artificial Intelligence and Mathematics*, 1992.

[93] X.-S. Gao, K. Jiang, and C.-C. Zhu. Geometric Constraint Solving with Conics and Linkages. *Computer Aided Design*, 34(6) :421–433, 2002.

[94] X.-S. Gao and G.-F. Zhang. Geometric Constraint Solving Based on Connectivity of Graph. Technical Report 22, Academia Sinica, Beijing, China, december 2003.

[95] X.-S. Gao and G.-F. Zhang. Geometric Constraint Solving via C-tree Decomposition. In *Proc. SM, ACM symposium on Solid Modeling and Applications*, pages 45–55, 2003.

[96] M. R. Garey and D. S. Johnson. *Computers and Intractability. A guide to the theory of NP-completness*. San Francisco, W. H. Freeman, 1979.

[97] P. A. Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In *Proc. ECAI*, pages 31–35, 1992.

[98] I. Gent and T. Walsh. CSPLib : a benchmark library for constraints. In *Proc. CP, Constraint Programming*, 1999.

[99] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[100] A. Goldsztejn. *Définition et applications des extensions des fonctions réelles aux intervalles généralisés*. PhD thesis, Université de Nice–Sophia Antipolis, 2005.

[101] A. Goldsztejn. A Branch and Prune Algorithm for the Approximation of Non-Linear AE-Solution Sets. In *Proc. SAC, ACM*, pages 1650–1654, 2006.

[102] A. Goldsztejn. Modal Intervals Revisited Part I : A Generalized Interval Natural Extension. Technical Report hal-00294219, HAL, submitted to Reliable Computing.

[103] A. Goldsztejn. Modal Intervals Revisited Part II : A Generalized Interval Mean-Value Extension. Technical Report hal-00294222, HAL, submitted to Reliable Computing.

[104] A. Goldsztejn and F. Goualard. Box Consistency through Adaptive Shaving. In *Proc. ACM SAC*, pages 2049–2054, 2010.

[105] A. Goldsztejn and L. Granvilliers. A New Framework for Sharp and Efficient Resolution of NCSP with Manifolds of Solutions. In *Proc. CP, Constraint Programming, LNCS 5202*, pages 190–204, 2008.

[106] A. Goldsztejn, C. Michel, and M. Rueher. Efficient Handling of Universally Quantified Inequalities. *Constraints*, 14(1) :117–135, 2009.

[107] C. Gomes, M. Sellmann, C. van Es, and H. van Es. The Challenge of Generating Spatially Balanced Scientific Experiment Designs. In *Proc. of the first CPAIOR conference, LNCS 3011*, pages 387–394, 2004.

[108] J. Gottlieb, M. Puchta, and C. Solnon. A study of greedy, local search and ant colony optimization approaches for car sequencing problems. In *Proc. of the EvoCOP conference, LNCS 2611*, pages 246–257, 2003.

[109] M. Gouttefarde, J.-P. Merlet, and D. Daney. Wrench-feasible workspace of parallel cable-driven mechanisms. In *Proc. ICRA, IEEE Int. Conf. on Robotics and Automation*, 2007.

[110] C. Grandon and B. Neveu. A Specific Quantifier Elimination for Inner Box Test in Distance Constraints with Uncertainties. Technical Report 5883, INRIA, 2006.

[111] L. Granvilliers. `RealPaver` *User's Manual, version 0.3.* University of Nantes, 2003.

[112] L. Granvilliers and F. Benhamou. Algorithm 852 : Realpaver : An Interval Solver using Constraint Satisfaction Techniques. *ACM Transactions on Mathematical Software*, 32(1) :138–156, 2009.

[113] L. Granvilliers, E. Monfroy, and F. Benhamou. Symbolic-Interval Cooperation in Constraint Programming. In *Proc. ISSAC, ACM*, pages 150–166, 2001.

[114] J. Graver. *Counting on Frameworks : Mathematics to Aid the Design of Rigid Structures*, volume 25. Mathematical Association of America, 2002.

[115] J. Graver, B. Servatius, and H. Servatius. *Combinatorial Rigidity. Graduate Studies in Mathematics*. American Mathematical Society, 1993.

[116] E. Grossmann and J. S. Victor. Single and Multi-View Reconstruction of Structured Scenes. In *Proc. Asian Conference on Computer Vision*, pages 93–104, 2002.

[117] M. Gruebler. *Getriebelehre*. Springer, Berlin, 1917.

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

[119] E. Hansen and G. W. Walster. *Global Optimization using Interval Analysis*. CRC Press, 2^{nd} edition, 2003.

[120] R.I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2000.

[121] W. Harvey and P. J. Stuckey. Improving Linear Constraint Propagation by Changing Constraint Representation. *Constraints*, 7 :173–207, 2003.

[122] A. Heck. *Introduction to Maple*. Springer Verlag, 2003.

[123] G. Heindl, V. Kreinovich, and A.V. Lakeyev. Solving Linear Interval Systems is NP-Hard Even If We Exclude Overflow and Underflow. *Reliable Computing*, 4(4) :383–388, 1998.

[124] B. Hendrickson. Conditions for unique realizations. *SIAM journal of Computing*, 21(1) :65–84, 1992.

[125] L. Henneberg. *Die graphische Statik der starren Systeme.* , Leipzig, 1911.

[126] M. Hladik, D. Daney, and E. P. Tsigaridas. Bounds on Eigenvalues of Symmetric Interval Matrices. In *SWIM, WS on interval methods*, 2009.

[127] C. M. Hoffmann and R. Joan-Arinyo. Symbolic constraints in constructive geometric constraint solving. *Journal of Symbolic Computation*, 23 :287–299, 1997.

[128] C. M. Hoffmann and R. Joan-Arinyo. A Brief on Constraint Solving. *Computer-Aided Design and Applications*, 2005.

[129] C. M. Hoffmann, A. Lomonosov, and M. Sitharam. Finding Solvable Subsets of Constraint Graphs. In *Constraint Programming CP'97*, pages 463–477, 1997.

[130] C. M. Hoffmann, A. Lomonosov, and M. Sitharam. Decomposition of Geometric Constraints Part I : performance measures & Part II : new algorithms. *J. of Symbolic Computation*, 31(4), 2001.

[131] C. M. Hoffmann, M. Sitharam, and B. Yuan. Making Constraint Solvers More Usable : Overconstraint Problem. *Computer Aided Design*, 36(4) :377–399, 2004.

[132] J. E. Hopcroft and R. M. Karp. An $n^{2.5}$ Algorithm for Maximum Matching in Bipartite Graphs. *SIAM J. Computing*, 2(4) :225–231, 1973.

[133] J. E. Hopcroft and R. E. Tarjan. Dividing a Graph into Triconnected Components. *SIAM J. Computing*, 3 :135–158, 1973.

[134] E. Hopper. *Two-Dimensional Packing Utilising Evolutionary Algorithms and Other Meta-Heuristic Methods*. PhD. Thesis Cardiff University, 2000.

[135] E. Hopper and B. C. H. Turton. An Empirical Investigation on Metaheuristic and Heuristic Algorithms for a 2D Packing Problem. *European Journal of Operational Research*, 128 :34–57, 2001.

[136] W. Horner. A new Method of Solving Numerical Equations of all Orders, by Continuous Approximation. *Philos. Trans. Roy. Soc. London*, 109 :308–335, 1819.

[137] A. S. Householder. *Principles of Numerical Analysis*. McGraw-Hill, New York, NY, USA, 1953.

[138] C. Hsu and B. Brüderlin. A Degree-of-Freedom Graph Approach. In *Geometric Modeling : Theory And Practice*, pages 132–155. , 1997.

[139] S. Hudson. Incremental Attribute Evaluation : a Flexible Algorithm for Lazy Update. *ACM TOPLAS, Transactions on Programming Languages and Systems*, 13(3) :315–341, 1991.

[140] E. Hyvönen. Constraint Reasoning Based on Interval Arithmetic—The Tolerance Propagation Approach. *Artificial Intelligence*, 58 :71–112, 1992.

[141] ILOG, Av. Galliéni, Gentilly. *Ilog Solver V. 5, Users' Reference Manual*, 2000.

[142] M. Iori, S. Martello, and M. Monaci. *Metaheuristic Algorithms for the Strip Packing Problem*, pages 159–179. Kluwer Academic Publishers, 2003.

[143] M. Janssen, P. Van Hentenryck, and Y. Deville. A Constraint Satisfaction Approach for Enclosing Solutions to Parametric Ordinary Differential Equations. *SIAM Journal on Numerical Analysis*, 40(5) :1896–1939, 2002.

[144] L. Jaulin. Interval Constraint Propagation with Application to Bounded-error Estimation. *Automatica*, 36 :1547–1552, 2000.

[145] L. Jaulin. Localization of an Underwater Robot Using Interval Constraints Propagation. In *Proc. CP, Constraint Programming*, pages 244–255, 2006.

[146] L. Jaulin and G. Chabert. Hull Consistency Under Monotonicity. In *To appear in Proc. CP'09, LNCS 5732*, 2009.

[147] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer, 2001.

[148] L. Jaulin and E. Walter. Guaranteed Robust Nonlinear Minimax Estimation. *Transaction on Automatic Control, IEEE*, 47(11) :1857–1864, 2002.

[149] C. Jermann. *Résolution de contraintes géométriques par rigidification récursive et propagation d'intervalles*. Ph.d. thesis, UNSA, NICE, 2002.

[150] C. Jermann, B. Neveu, and G. Trombettoni. A New Structural Rigidity for Geometric Constraint Systems. In *ADG, int. WS on Automated Deduction in Geometry*, 2002.

[151] C. Jermann, B. Neveu, and G. Trombettoni. Algorithms for Identifying Rigid Subsystems in Geometric Constraint Systems. In *Proc. IJCAI*, pages 233–238, 2003.

[152] C. Jermann, B. Neveu, and G. Trombettoni. Inter-Block Backtracking : Exploiting the Structure in Continuous CSPs. In *COCOS, WS on Global Constrained Optimization and Constraint Satisfaction*, 2003.

[153] C. Jermann, B. Neveu, and G. Trombettoni. A New Structural Rigidity for Geometric Constraint Systems. In *ADG, Int. Workshop on Automated Deduction in Geometry, LNCS 2930*, pages 87–106, 2004.

[154] C. Jermann, B. Neveu, and G. Trombettoni. Algorithmes pour la détection de rigidités dans les csp géométriques. *JEDAI, journal électronique en intelligence artificielle*, 2, 2004.

[155] C. Jermann, B. Neveu, and G. Trombettoni. Inter-Block Backtracking : Exploiting the Structure in Continuous CSPs. In *Selected papers of COCOS, WS on Global Optimization and Constraint Satisfaction, LNCS 3478*, 2005.

[156] C. Jermann, G. Trombettoni, B. Neveu, and P. Mathis. Decomposition of Geometric Constraint Systems : a Survey. *IJCGA, Int. Journal of Computational Geometry and Applications*, 16(5) :379–414, 2006.

[157] C. Jermann, G. Trombettoni, B. Neveu, and M. Rueher. A Constraint Programming Approach for Solving Rigid Geometric Systems. In *Proc. CP, Constraint Programming*, volume 1894 of *LNCS*, pages 233–248, 2000.

[158] C. Jermann, G. Trombettoni, B. Neveu, and M. Rueher. A Constraint Programming Approach for Solving Rigid Geometric Systems. In *Proc. CP, Constraint Programming, LNCS 1894*, pages 233–248, 2000.

[159] C. Jermann, G. Trombettoni, B. Neveu, and M. Rueher. A New Heuristic to Identify Rigid Clusters. In *ADG, int. WS on Automated Deduction in Geometry*, pages 2–6, 2000.

[160] R. Joan-Arinyo and A. Soto. A correct Rule-Based Geometric Constraint Solver. *Computer and Graphics*, 5(21) :599–609, 1997.

[161] R. Joan-Arinyo, A. Soto-Riera, S. Vila-Marta, and J. Vilaplana-Pasto. Transforming an Under-constrained Geometric Constraint Problem into a Well-constrained one. In *Proc. SM, ACM symposium on Solid Modeling and Applications*, pages 33–44, New York, NY, USA, 2003.

[162] R. Joan-Arinyo, A. Soto-Riera, S. Vila-Marta, and J. Vilaplana-Pasto. Revisiting Decomposition Analysis of Geometric Constraint Graphs. *Computer Aided Design*, 36 :123–140, 2004.

[163] E. Kaucher. Interval Analysis in the Extended Interval Space. *Computing, Suppl.*, 2 :33–49, 1980.

[164] R. B. Kearfott. *Rigorous Global Search : Continuous Problems*. Kluwer, Dordrecht, 1996.

[165] R. B. Kearfott. Discussion and Empirical Comparisons of Linear Relaxations and Alternate Techniques in Validated Deterministic Global Optimization. *Journal of Optimization Methods and Software*, 21(5) :715–731, 2006.

[166] R. B. Kearfott, M. T. Nakao, A. Neumaier, S. Rump, S. Shary, and P. Van Hentenryck. Standardized Notation in Interval Analysis. In *Proc. International School-seminar on optimization methods and their applications*, 2005.

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

[168] M. Kieffer, L. Jaulin, E. Walter, and D. Meizel. Robust Autonomous Robot Localization Using Interval Analysis. *Reliable Computing*, 3(6) :337–361, 2000.

[169] S. Kirkpatrick, C. Gellat, and M. Vecchi. Optimization by simulated annealing. *Science*, 220 :671–680, 1983.

[170] A. Knoll, Y. Hijazi, A. Kensler, M. Schott, C. Hansen, and H. Hagen. Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. *Comput. Graph. Forum*, 28(1) :26–40, 2009.

[171] P. Koiran. Complexité et décidabilité pour les modèles de calcul algébriques et analogiques. *Habilitation à diriger des recherches, Université Claude Bernard–Lyon I*, 1999.

[172] A. Kolen. A Genetic Algorithm for Frequency Assignment. Technical report, Universiteit Maastricht, 1999.

[173] L. Kolev. An Improved Interval Linearization for Solving Non-Linear Problems. *Numerical Algorithms*, 37 :213–224, 2004.

[174] D. König. Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. In *Math Ann 77*, pages 453–465, 1916.

[175] A. Koster, C. Van Hoesel, and A. Kolen. Solving frequency assignment problems via tree-decomposition. Technical Report 99-011, Universiteit Maastricht, 1999.

[176] G. Kramer. *Solving Geometric Constraint Systems*. MIT Press, 1992.

[177] V. Kreinovich, A.V. Lakeyev, and S.I. Noskov. Optimal Solution of Interval Linear Systems is Intractable (NP-hard). *Interval Computations*, 1 :6–14, 1993.

[178] V. Kreinovich, A.V. Lakeyev, J. Rohn, and P.T. Kahl. *Computational Complexity and Feasibility of Data Processing and Interval Computations (Applied Optimization)*. Kluwer, 1997.

[179] E. Lahaye. Une méthode de résolution d'une catégorie d'équations transcendantes. *Compte-rendu des Séances de L'Académie des Sciences*, 198 :1840–1842, 1934.

[180] H. Lamure and D. Michelucci. Qualitative study of geometric constraints. In B. Bruderlin and D. Roller, editors, *Geometric Constraint Solving and Applications*, pages 234–258. Springer, 1998.

[181] R.S. Latham and A.E. Middleditch. Connectivity Analysis : A Tool For Processing Geometric Constraints. *Computer Aided Design*, 28(11) :917–928, 1996.

[182] Y. Lebbah. *Contribution à la résolution de contraintes par consistance forte*. Phd thesis, Université de Nantes, 1999.

[183] Y. Lebbah, C. Michel, M. Rueher, D. Daney, and J.P. Merlet. Efficient and Safe Global Constraints for Handling Numerical Constraint Systems. *SIAM Journal on Numerical Analysis*, 42(5) :2076–2097, 2005.

[184] N. Lesh, J. Marks, A. Mc. Mahon, and M. Mitzenmacher. Exhaustive Approaches to 2D Rectangular Perfect Packings. *Information Processing Letters*, 90 :7–14, 2004.

[185] N. Lesh, J. Marks, A. Mc. Mahon, and M. Mitzenmacher. New Heuristic and Interactive Approaches to 2D Strip Packing. *ACM J. of Exp. Algorithmics*, 10 :1–18, 2005.

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

[187] O. Lhomme. Quick Shaving. In *Proc. AAAI*, pages 411–415, 2005.

[188] A. Lomonosov. *Graph and Combinatorial Algorithms for Geometric Constraint Solving*. PhD thesis, University of Florida, 2004.

[189] L. Lovasz and Y. Yemini. On Generic Rigidity in the Plane. *SIAM J. Alg. Discrete Methods*, 3 :91–98, 1982.

[190] L. Luksan and J. Vleck. Sparse and Partially Separable Test Problems for Unconstrained and Equality Constrained Optimization. Technical Report 767, Institute of Computer Science, Academy of Sciences of the Czeck Republic, 1999.

[191] Y. Zhou M. Sitharam. A tractable, Approximate Characterization of Combinatorial Rigidity in 3D. In *ADG*, 2004.

[192] S. Martello, M. Monaci, and D. Vigo. An Exact Approach to the Strip Packing Problem. *INFORMS Journal of Computing*, 15 :310–319, 2003.

[193] S. Martello and D. Vigo. Exact Solution of the Two-Dimensional Finite Bin Packing Problem. *Management science*, 15 :310–319, 1998.

[194] W.A. Martin. Determining the Equivalence of Algebraic Expressions by Hash Coding. *J. ACM*, 18(4) :549–558, 1971.

[195] P. Massan Kuzo. *Des contraintes projectives en modélisation tridimensionnelle interactive*. Ph.d. thesis, Ecole des Mines de Nantes, 1999.

[196] P. Mathis. *Constructions géométriques sous contraintes en modélisation à base topologique*. PhD thesis, Université Louis Pasteur, 1997.

[197] P. Mathis, P. Schreck, and J.-F. Dufourd. YAMS : A Multi-Agent System for 2D Constraint Solving. In Beat Brüderlin and Dieter Roller, editors, *Geometric Constraint Solving and Applications*, pages 211–233. Springer, 1998.

[198] D. McAllester. Partial order backtracking. Research Note, Artificial Intelligence Laboratory, MIT, 1993. http ://eprints.kfupm.edu.sa/57156/1/57156.pdf.

[199] C. McGlone. Bundle Adjustment with Object Space Geometric Constraints for Site Modeling. In *Proc.* SPIE *Conf. on Integrating Photogrammetric Techniques with Scene Analysis and Machine Vision*, volume 2486, pages 25–36, 1995.

[200] P. McLauchlan, X. Shen, A. Manessis, P. Palmer, and A. Hilton. Surface Based Structure from Motion Using Feature Groupings. In *Proc. Asian Conference on Computer Vision*, pages 699–705, 2000.

[201] J.-P. Merlet. ALIAS : An Algorithms Library for Interval Analysis for Equation Systems. Technical Report 621, INRIA Sophia, 2000. `www-sop.inria.fr/coprin/logiciels/ALIAS/ALIAS.html`.

[202] J.-P. Merlet. Optimal Design for the Micro Parallel Robot MIPS. In *Proc. ICRA, International Conference on Robotics and Automation, IEEE*, pages 1149–1154, 2002.

[203] J.-P. Merlet. Interval Analysis and Robotics. In *Symp. of Robotics Research*, 2007.

[204] J.-P. Merlet. `www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html`, 2009.

[205] J.-P. Merlet and D. Daney. Dimensional Synthesis of Parallel Robots with a Guaranteed Given Accuracy over a Specific Workspace. In *Proc. ICRA, IEEE Int. Conf. on Robotics and Automation*, pages 942–947, 2005.

[206] L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. In *Proc. of the OOPSLA conference*, 2002.

[207] D. Michelucci and S. Foufou. The Witness Configuration Method. *Computer Aided Design*, 2005.

[208] D. Michelucci and H. Lamure. Résolution de contraintes géométriques par homotopie. In *Actes de AFIG*, 1994.

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

[210] S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing Conflict : A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, 58 :161–205, 1992.

[211] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs N.J., 1966.

[212] R.E. Moore, B. Kearfott, and M.J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009.

[213] J.-J. Moré, B. Garbow, and K. Hillstrom. Testing Unconstrained Optimization Software. *ACM Trans. Math. Software*, 7(1) :136–140, 1981.

[214] C. Morgenstern. Distributed Coloration Neighborhood Search. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability : Second DIMACS Implementation Challenge, 1993*, volume 26, pages 335–357. American Mathematical Society, 1996.

[215] B. Mourrain and J.-P. Pavone. Subdivision Methods for Solving Polynomial Equations. Technical Report RR-5658, INRIA, 2005.

[216] S. Muchnick. *Advanced Compiler Design and Implementation*. M. Kauffmann, 1997.

[217] P. Nataraj and M. Arounassalame. An Interval Newton Method based on the Bernstein Form for Bounding the Zeros of Polynomial Systems. In *Abstract at SCAN, the GAMM/IMACS int. Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations*, 2008.

[218] N. Nedialkov, K. Jackson, and J. Pryce. An Effective High-Order Interval Method for Validating Existence and Uniqueness of the Solution of an IVP for an ODE. *Reliable Computing*, 7(6) :449–465, 2001.

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

[220] A. Neumaier and O. Shcherbina. Safe Bounds in Linear and Mixed-Integer Programming. *Math. Programming A 99*, pages 283–296, 2004.

[221] B. Neveu, G. Chabert, and G. Trombettoni. When Interval Analysis helps Interblock Backtracking. In *Proc. CP, Constraint Programming, LNCS 4204*, pages 390–405, 2006.

[222] B. Neveu, C. Jermann, and G. Trombettoni. Inter-Block Backtracking : Exploiting the Structure in Continuous CSPs. In *COCOS, selected papers in the int. WS on Global Constrained Optimization and Constraints, LNCS 3478*, pages 15–30, 2005.

[223] B. Neveu and G. Trombettoni. INCOP : An Open Library for INcomplete Combinatorial OPtimization. In *Proc. Constraint Programming, LNCS 2833*, pages 909–913, 2003.

[224] B. Neveu and G. Trombettoni. When Local Search Goes with the Winners. In *Int. Workshop CPAIOR'2003*, pages 180–194, 2003.

[225] B. Neveu and G. Trombettoni. Hybridation de GWW avec de la recherche locale. *JEDAI, journal électronique en intelligence artificielle*, 3, 2004.

[226] B. Neveu and G. Trombettoni. Strip Packing Based on Local Search and a Randomized Best-Fit. In *BPPC, int. WS on bin packing and placement, at CPAIOR conference*, 2008.

[227] B. Neveu, G. Trombettoni, and I. Araya. Incremental Move for Strip-Packing. In *Proc. ICTAI, int. conference on tools with artificial intelligence, IEEE*, 2007.

[228] B. Neveu, G. Trombettoni, and I. Araya. Recherche locale pour la découpe de rectangles. In *Actes du congrès ROADEF*, pages 43–44, 2007.

[229] B. Neveu, G. Trombettoni, I. Araya, and M.-C. Riff. A Strip Packing Solving Method Using an Incremental Move Based on Maximal Holes. *IJAIT, Journal on Artificial Intelligence Tools*, 17(5) :881–901, 2008.

[230] B. Neveu, G. Trombettoni, and G. Chabert. Improving Inter-Block Backtracking with Interval Newton. *Constraints*, 14(4) :—, 2009.

[231] B. Neveu, G. Trombettoni, and F. Glover. ID Walk : A Candidate List Strategy with a Simple Diversification Device. In *Proc. CP, Constraint Programming, LNCS 3258*, pages 423–437, 2004.

[232] B. Neveu, G. Trombettoni, and F. Glover. IDW : un algorithme de recherche locale combinant intensification et diversification. In *ROADEF, congrès de l'association française de recherche opérationnelle et d'aide à la décision*, pages 288–289, 2005.

[233] J. J. Oung, M. Sitharam, B. Moro, and A. Arbree. FRONTIER : Fully Enabling Geometric Constraints for Feature Based Modeling and Assembly. In *Proc. of ACM Solid Modeling symposium*, 2001.

[234] J. Owen. Algebraic Solution For Geometry From Dimensional Constraints. In *Proc. SM, ACM Symposium on Solid Modeling and CAD/CAM Applications*, pages 397–407. ACM Press, 1991.

[235] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[236] B. T. Polyak. *Introduction to Optimization*. Optimization Software, 1987.

[237] A. Pothen and J. Chin-Fan. Computing the Block Triangular Form of a Sparse Matrix. *ACM Transactions on Mathematical Software*, 16(4) :303–324, 1990.

[238] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C*. Cambridge Univ. Press, 1988.

[239] P. Refalo. Impact-Based Search Strategies for Constraint Programming. In *Proc. CP, LNCS 3258*, pages 557–571, 2004.

[240] J.-C. Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique*. PhD thesis, LIRMM, Montpellier, France, 1995.

[241] J.C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proc. AAAI*, pages 362–367, 1994.

[242] J. Renegar. Recent Progress on the Complexity of the Decision Problem for the Reals. *DIMACS series in discrete mathematics and theoretical computer science*, 6 :287–308, 1991.

[243] J. Renegar. On the Computational Complexity and the First Order Theory of the Reals. *Journal of Symbolic Computation*, 13 :255–352, 1992.

[244] J. Rohn and V. Kreinovich. Computing Exact Componentwise Bounds on Solutions of Linear Systems with interval Data is NP-hard. *SIAM Journal on Matrix Analysis and Applications (SIMAX)*, 16 :415–420, 1995.

[245] M. Rueher, A. Goldsztejn, Y. Lebbah, and C. Michel. Capabilities of Constraint Programming in Rigorous Global Optimization. In *NOLTA, int. Symp. on Nonlinear Theory and its Applications*, 2008.

[246] S. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. http ://www.ti3.tu-harburg.de/rump/.

[247] S. Rump. Rigourous and Portable Standard Functions. *BIT*, 41(3) :540–562, 2001.

[248] H.-S. Ryoo and N. V. Sahinidis. A Branch-and-Reduce Approach to Global Optimization. *Journal of Global Optimization*, 8(2) :107–139, 1996.

[249] N. V. Sahinidis. Global Optimization and Constraint Satisfaction : The Branch-and-Reduce Approach. In *COCOS*, pages 1–16, 2002.

[250] N. V. Sahinidis and M. Twarmalani. *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming*. Kluwer, Dordrecht, 2002.

[251] S. Sahni. Computationally Related Problems. *SIAM Journal of Computing*, 3 :262–279, 1974.

[252] M. Sanchez, S. Bouveret, S. de Givry, F. Heras, P. Jégou, J. Larrosa, S. Ndiaye, E. Rollon, T. Schiex, C. Terrioux, G. Verfaillie, , and M. Zytnicki. Max-CSP Competition 2008 : Toulbar2 Solver Description. In *Proc. International CSP Solver Competition*, 2008.

[253] M. Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. PhD thesis, University of Washington, Seattle, 1994. Also available as Technical Report 94-09-10.

[254] H. Schichl and A. Neumaier. Interval Analysis on Directed Acyclic Graphs for Global Optimization. *Journal of Global Optimization*, 33(4) :541–562, 2005.

[255] E. Schramm and P. Schreck. Solving Geometric Constraints Invariant Modulo the Similarity Group. In *Proc. of the 2002 Int. Conference on Computational Science and its Applications, Part II*, pages 356–365. LNCS 2669 (Part III), 2003.

[256] P. Schreck. Implantation d'un système à base de connaissances pour les constructions géométriques. *Revue d'Intelligence Artificielle*, 8(3) :223–247, 1994.

[257] P. Schreck. Robustness in CAD Geometric Construction. In *Proc. of IV, int. Conference on Information Visualisation*, pages 111–116. IEEE, 2001.

[258] P. Schreck and P. Mathis. Geometrical Constraint System Decomposition : A Multi-group Approach. Technical report, Université de Strasbourg, France, 2005.

[259] M. Schub and S. Smale. Complexity of Bezout's theorem I : Geometric Aspects. *Journal of the American Mathematical Society*, 6 :459–501, 1993.

[260] M. Schub and S. Smale. Complexity of Bezout's theorem II : Volumes and Probabilities. *Computational Algebraic Geometry. Progress in Mathematics*, 109 :267–285, 1993.

[261] M. Schub and S. Smale. Complexity of Bezout's theorem III : Condition Number and Packing. *Journal of Complexity*, 9 :4–14, 1993.

[262] M. Schub and S. Smale. Complexity of Bezout's theorem V : Polynomial Time. *Theoretical Computer Science*, 133 :141–164, 1994.

[263] M. Schub and S. Smale. Complexity of Bezout's theorem IV : Probability of Success, Extensions. *SIAM Journal of Numerical Analysis*, 33(1) :128–148, 1996.

[264] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Cliques, Coloring, and Satisfiability : Second DIMACS Implementation Challenge. Theoretical Computer Science, vol. 26, AMS*, 2003.

[265] D. Serrano. *Constraint Management in Conceptual Design*. PhD thesis, MIT, 1987.

[266] B. Servatius and W. Whiteley. Constraining Plane Configurations in Computer-Aided Design : Combinatorics of Directions and Lengths. *SIAM J. on Discrete Mathematics*, 12(1) :136–153, 1999.

[267] H. Sherali and W. Adams. *Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems*. Kluwer Academic Publishers, 1999.

[268] H. Simonis. Sudoku as a Constraint Problem. In *CP WS on Modeling and Reformulating Constraint Satisfaction Problems*, pages 13–27, 2005.

[269] M. Sitharam. *Frontier, an Opensource GNU Geometric Constraint Solver : Version3 (2003) for General 2D and 3D systems*, 2003. http ://www.cise.ufl.edu/ sitharam.

[270] M. Sitharam. Combinatorial Approaches to Geometric Constraint Solving : Problems, Progress, Directions. In *AMS-DIMACS book on CAD and manufacturing*. 2005.

[271] Christine Solnon. Combining two Pheromone Structures for Solving the Car Sequencing Problem with Ant Colony Optimization. *European Journal of Operational Research*, 191(3) :1043–1055, 2008.

[272] S. Sorlin and C. Solnon. A Global Constraint for Graph Isomorphism Problems. In *Proc. CP-AI-OR, LNCS 3011*, pages 287–301, 2004.

[273] A. Sosnov. *Modélisation Géométrique par Séparation de Contraintes*. Thèse de doctorat, Université de Nantes, 2003.

[274] A. Sosnov and P. Macé. Rapid Algebraic Resolution of 3D Geometric Contraints and Control of their Consistency. In *ADG, WS on Automatic Deduction in Geometry*, 2002.

[275] P. Sturm and S.J. Maybank. A Method for Interactive 3D Reconstruction of Piecewise Planar Objects from Single Images. In *Proc. BMVC, British Machine Vision Conference*, pages 265–274, 1999.

[276] T. Sunaga. Theory of Interval Algebra and its Application to Numerical Analysis. *Research Association of Applied Geometry (RAAG) Memoirs*, 2 :29–46, 1958.

[277] G. Sunde. Specification of Shapes by Dimensions and Other Geometric Constraints. In *IFIP WG Geometric Modeling*, 1986.

[278] I. Sutherland. *Sketchpad : A Man-Machine Graphical Communication System*. PhD thesis, Department of Electrical Engineering, MIT, 1963.

[279] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972.

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

[281] G. Trombettoni. *Algorithmes de maintien de solution par propagation locale pour les systèmes de contraintes*. Thèse de doctorat, Université de Nice–Sophia Antipolis, 1997.

[282] G. Trombettoni. A Polynomial Time Local Propagation Algorithm for General Dataflow Constraint Problems. In *Proc. CP, Constraint Programming, LNCS 1520*, pages 432–446, 1998.

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

[284] G. Trombettoni and B. Neveu. Computational Complexity of Multi-way, Dataflow Constraint Problems. In *Proc. of IJCAI, Int. Joint Conference on Artificial Intelligence*, pages 358–363, 1997.

[285] G. Trombettoni and B. Neveu. Links for Boosting Predictable Interactive Constraint Systems. In *UICS, int. WS on User Interaction in Constraint Satisfaction, at CP conference*, 2001.

[286] G. Trombettoni, Y. Papegay, G. Chabert, and O. Pourtallier. A Box-Consistency Contraction Operator Based on Extremal Functions. In *Abstract at SCAN, the GAMM/IMACS int. Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations*, 2008.

[287] G. Trombettoni, Y. Papegay, G. Chabert, and O. Pourtallier. A Box-Consistency Contractor Based on Extremal Functions. In *Proc. CP, Constraint Programming (to appear).* Springer, LNCS, 2010.

[288] G. Trombettoni and M. Wilczkowiak. GPDOF : A Fast Algorithm to Decompose Underconstrained Geometric Constraints : Application to 3D Modeling. *IJCGA, Int. Journal of Computational Geometry and Applications*, 16(6) :479–511, 2006.

[289] W. Tucker. A Rigorous ODE Solver and Smale's 14th Problem. *Found. Comput. Math.*, 2 :53–117, 2002.

[290] J.R. Ullman. An Algorithm for Subgraph Isomorphism. *Journal ACM*, 23(1) :31–42, 1976.

[291] P. Van Hentenryck, D. McAllester, and D. Kapur. Solving Polynomial Systems Using a Branch and Prune Approach. *SIAM Journal on Numerical Analysis*, 34(2), 1997.

[292] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* The MIT press, 2005.

[293] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica : A Modeling Language for Global Optimization.* MIT Press, 1997.

[294] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, Implementation, and Evaluation of the Constraint Language `CC(FD)`. *J. Logic Programming*, 37(1–3) :139–164, 1994.

[295] B. Vander Zanden. An Incremental Algorithm for Satisfying Hierarchies of Multi-way, Dataflow Constraints. *ACM TOPLAS*, 18(1) :30–72, 1996.

[296] A. Verroust, F. Schonek, and D. Roller. Rule Oriented Method for Parametrized Computer Aided Design. *Computer Aided Design*, 24(6) :531–540, 1992.

[297] C. Voudouris and E. Tsang. Solving the radio link frequency assignment problem using guided local search. In *Nato Symposium on Frequency Assignment, Sharing and Conservation in Systems(AEROSPACE)*, 1998.

[298] X.-H. Vu, D. Sam-Haroud, and B. Faltings. Enhancing Numerical Constraint Propagation using Multiple Inclusion Representations. *Annals of Mathematics and Artificial Intelligence (to appear)*, 2008.

[299] X.-H. Vu, H. Schichl, and D. Sam-Haroud. Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems. In *Proc. ICTAI, IEEE*, pages 72–81, 2004.

[300] X.-H. Vu, H. Schichl, and D. Sam-Haroud. Interval Propagation and Search on Directed Acyclic Graphs for Numerical Constraint Solving. *Journal of Global Optimization (to appear)*, 2008.

[301] M. Warmus. Calculus of Approximations. *Bulletin de l'Académie Polonaise de Sciences*, 4(5) :253–257, 1956.

[302] W. Whiteley. Applications of the Geometry of Rigid Structures. In Henry Crapo, editor, *Computer Aided Geometric Reasoning*, pages 219–254. INRIA, 1987.

[303] M. Wilczkowiak. *3D Modelling from Images Using Geometric Constraints*. PhD thesis, Institut National polytechnique de Grenoble, France, 2004.

[304] M. Wilczkowiak, E. Boyer, and P. Sturm. 3D Modelling Using Geometric Constraints : A Parallelepiped Based Approach. In *ECCV'02, LNCS 2353*, pages 221–236, 2002.

[305] M. Wilczkowiak, P. Sturm, and E. Boyer. The Analysis of Ambiguous Solutions in Linear Systems and its Application to Computer Vision. In *Proc. BMVC, British Machine Vision Conference*, pages 53–62, 2003.

[306] M. Wilczkowiak, G. Trombettoni, C. Jermann, P. Sturm, and E. Boyer. Scene Modeling Based on Constraint System Decomposition Techniques. In *Proc. ICCV, International Conference on Computer Vision*, pages 1004–1010, 2003.

[307] W. Wu. Basic Principles Of Mechanical Theorem Proving In Elementary Geometries. *J. Automated Reasoning*, 2 :221–254, 1986.

[308] S.-M. Hu Y.-T. Li and J.-G. Sun. A Constructive Approach to Solving 3D Geometric Constraint Systems Using Dependence Analysis. *Computer Aided Design*, 34(2) :97–108, 2002.

[309] R. C. Young. The Algebra of Multi-valued Quantities. *Mathematische Annalen*, 104 :260–290, 1931.

[310] D. Zhang, Y. Kang, and A. Deng. A New Heuristic Recursive Algorithm for the Strip Packing Problem. *Computers and Operations Research*, 33 :2209–2217, 2006.