Modélisation et résolution en programmation par contraintes de problèmes mixtes continu/discret de satisfaction de contraintes et d'optimisation

Nicolas Berger

▶ **To cite this version:**

Nicolas Berger. Modélisation et résolution en programmation par contraintes de problèmes mixtes continu/discret de satisfaction de contraintes et d'optimisation. Modélisation et simulation. Université de Nantes, 2010. Français. NNT : . tel-00560963

HAL Id: tel-00560963

https://theses.hal.science/tel-00560963

Submitted on 31 Jan 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Centrale de Nantes Université de Nantes École des Mines de Nantes

ÉCOLE DOCTORALE STIM
« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE MATHÉMATIQUES »

N° attribué par la bibliothèque

Année 2010

Modélisation et résolution en programmation par contraintes de problèmes mixtes continu/discret de satisfaction de contraintes et d'optimisation

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE NANTES

Discipline : INFORMATIQUE

présentée et soutenue publiquement par

Nicolas BERGER

le Jeudi 7 Octobre 2010

au LINA

devant le jury ci-dessous

Président	:	Pr. Frédéric SAUBION	LERIA – Université d'Angers
Rapporteurs	:	Arnaud LALLOUET, Professeur	GREYC – Université de Caen
		Gilles TROMBETTONI, Maître de Conférences, HDR	INRIA – Université de Nice-Sophia
Examinateur	:	Arnaud GOTLIEB, Chargé de recherche	INRIA Rennes

Directeur de thèse : Laurent GRANVILLIERS, Professeur
Encadrant de thèse : Frédéric GOUALARD, Maître de Conférences
Laboratoire : LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE
UMR CNRS 6241 – 2, rue de la Houssinière – BP 92208
44322 NANTES CEDEX 3

N° ED : .....

MODÉLISATION ET RÉSOLUTION EN PROGRAMMATION PAR CONTRAINTES DE PROBLÈMES MIXTES CONTINU/DISCRET DE SATISFACTION DE CONTRAINTES ET D'OPTIMISATION

Modeling and Solving Mixed Continuous/Discrete Constraint Satisfaction and Optimisation Problems

Nicolas BERGER

favet neptunus eunti

Université de Nantes

Nicolas BERGER

Modélisation et résolution en programmation par contraintes de problèmes mixtes continu/discret de satisfaction de contraintes et d'optimisation

xv+146 p.

Ce document a été préparé avec LaTeX2$_\varepsilon$ et la classe `these-IRIN` version 0.92 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe `these-IRIN` est disponible à l'adresse :

`http ://login.irin.sciences.univ-nantes.fr/`

Impression : BERGER-PhD-thesis.tex – 25/10/2010 – 11 :27

Révision pour la classe : $ Id : these-IRIN.cls,v 1.3 2000/11/19 18 :30 :42 fred Exp

Tous sous le ciel, connaissant le beau comme le beau : voici le laid !
Tous connaissant le bien comme le bien : voici le mal !
C'est ainsi que l'être et le non-être naissent l'un de l'autre,
Que le difficile et le facile s'accomplissent l'un par l'autre,
Que mutuellement le long et le court se délimitent,
Le haut et le bas se règlent,
Le ton et le son s'accordent,
L'avant et l'après s'enchaînent.

C'est pourquoi le sage s'en tient à la pratique du non-agir.
Il enseigne sans parler.
Tous les êtres agissent et il ne leur refuse pas son aide.
Il produit sans s'approprier,
Travaille sans rien attendre,
Accomplit des oeuvres méritoires sans s'attacher,
Et, justement parce qu'il ne s'y attache pas,
Elles subsistent.

— Lao-Tseu, De la Voie et de sa vertu, II.

REMERCIEMENTS

Par où commencer ? Par le début, ce ne sera certes pas une erreur... Voire même avant le début ? Pourquoi pas, en effet, puisque mon premier merci sera pour Marc CHRISTIE, grâce à qui j'ai pu me mettre dans l'idée que j'avais effectivement la possibilité de *poursuivre en thèse* alors que je n'étais encore qu'en fin de licence. Et c'est aussi grâce à lui que j'ai découvert la programmation par contraintes et que je me suis acclimaté à ce cher laboratoire, à travers de nombreux stages sous sa direction et dans une ambiance de travail à la fois sérieuse et détendue. Enfin, je lui dois également d'avoir candidaté à cette *bourse sur critères universitaires*, de près de 4000 euros, que j'ai reçue en récompense de mes résultats en première année de Master et qui, accessoirement, m'a beaucoup aidé à *recoller mes morceaux* après les efforts intenses et continus que j'avais fournis pour obtenir des résultats satisfaisants cette année-là. Merci Marc, et bonne route à toi.

J'ai ensuite envie de remercier Frédéric GOUALARD, co-encadrant ma thèse mais aussi déjà présent dès les premiers instants, aux côtés de Marc, pour *m'occuper les bras* deux étés de suite alors que j'avais à cœur de me faire une place de thésard au LINA. Avec Laurent GRANVILLIERS, qui a été mon directeur de recherche ces quatre (cinq !) dernières années, que j'ai rencontré pendant ma première année de Master, au moment de fournir le support théorique à mes premières connaissances, toutes expérimentales, dans cette discipline quasi-mathématique qu'est la programmation par contraintes, ils forment le cœur, si j'ose dire, de ces scientifiques vrais, intellectuellement aussi précis qu'exigeants, qui ne sont pas là pour se perdre dans des apparences administratives mais qui nourrissent au contraire une curiosité toujours plus vive à l'égard du domaine de connaissance extrêmement pointu qui est le leur. J'ai eu beaucoup de chance de tomber sur eux. J'ai d'ailleurs envie d'ajouter tout de suite Alexandre GOLDSZTEJN et Christophe JERMANN, les deux autres *permanents* de l'équipe, comme on les appelle, qui avec Frédéric et Laurent ont formé le corps de l'équipe de recherche au sein de laquelle j'ai effectué mes quatre ans de thèse. Merci à vous quatre, et bonne continuation.

Mais je serais injuste de limiter mes remerciements aux seuls permanents de cet équipe, puisque j'ai bien sûr passé également beaucoup de temps en compagnie des doctorants du bureau 218, j'ai nommé d'abord ceux de la *première saison*, Jean-Marie NORMAND, Thomas DOUILLARD et Ricardo SOTO. Puis est venu pour eux le temps d'intégrer leur propre *spin-off*, les uns devenus docteurs, l(es) autre(s) non... Merci encore à eux, quoi qu'il en soit, pour leur aide précieuse au quotidien, que ce soit sur le plan technique mais aussi et surtout social, humain. Par chance, ils ont ensuite été remplacés par d'autres doctorants mais non des moindres, j'ai nommé Marie PELLEAU, Aurélien MÉREL et récemment Lauriane MILAN, qui ont toujours rendu plus vivables les conditions *atmosphériques* parfois très étouffantes dans lesquelles baigne le bureau à l'approche des dates importantes. À ce sujet j'ai envie d'ajouter des doctorants d'autres bureaux, d'autres couloirs, d'autres sphères, mais dont la présence a été tout aussi précieuse pour pouvoir apprécier un quotidien qui manquait parfois cruellement de sens, je pense à Benoît GUÉDAS, Guillaume PINOT et JACQUENOT, Sébastian PEÑA ou encore Fabien POULARD et Anthony PRZYBYLSKI, sans parler de Thomas CERQUEUS et VINCENT, Ahmed ZIANI, j'en oublie forcément mais qu'ils sachent tous tout le bien que je pense d'eux. Pour finir avec les doctorants, une inévitable mais méritée *mention spéciale du jury* pour Matthieu VERNIER, Prajol SHRESTHA et les affinités évidentes qui me relient à l'un comme à l'autre sur

un grand nombre de sujets, donnant toujours plus de sens aux moments que nous passons (et passerons encore !) ensemble. Dans tous les cas, un grand merci à tous et à toutes.

Comment ne pas penser maintenant à mes parents Brigitte et Jean-Pierre, mon frère Antoine, ainsi que ma soeur Marie, son bien-aimé Jean-François et leur adorable petit Ruben ? Mais aussi tous les membres de l'accueillante et très nombreuse famille à laquelle j'ai la chance d'appartenir. Sans vous, je n'y serais sûrement jamais arrivé aussi *facilement*, même si j'ai eu fort à faire ces (bientôt) dix dernières années. Je sais très bien que tout le monde n'a pas cette chance en ce monde et qu'en plus de toutes les épreuves que j'ai eu à traverser, d'autres moins chanceux que moi étaient seuls et ont dû en plus lutter pour se vêtir, se nourrir et avoir un toit. Rien de cela pour moi qui ai atterri en thèse sans jamais avoir eu à me préoccuper de ce genre de problèmes qui auraient facilement pu empoisonner davantage un quotidien déjà difficilement respirable... Mais c'est également le moment d'adresser un énorme merci à ce cher *dragon* d'Erwan, que j'ai rencontré chemin faisant et qui, bien que nous ne soyons pas *d'un même sang*, m'a lui aussi apporté le soutien et l'affection indéfectibles qui m'ont aidé à garder la tête hors de l'eau quand je manquais de couler sous les assauts des vagues, toujours plus profondes et puissantes, de cette quête de sens, vitale, qui m'anime au quotidien et, pourquoi ne pas l'espérer, pour encore très longtemps.

Pour finir, un énorme merci à Lao-Tseu, ainsi qu'à ses nombreux traducteurs. Pour sa rigueur et sa discipline, mais aussi pour l'admiration et la profondeur que m'inspirent de longue date ses écrits, qui m'ont plus que jamais été d'une aide précieuse pour garder le cap tout au long de la rédaction de ce manuscrit, ultime étape de mon doctorat. Oui, merci beaucoup, *Vieux-Maître*[1] !!

[1] Traduction française communément admise pour *Lao-Tseu*.

SOMMAIRE

I

INTRODUCTION

Les contraintes apparaissent spontanément dans nombre de situations auxquelles sont confrontés les êtres humains. Elles formalisent d'une manière transparente et naturelle les dépendances qui existent dans le monde physique ou ses représentations mathématiques abstraites [Bar99]. Ce sont des restrictions de l'ensemble des possibles, des éléments de connaissance par lequel le champ des possibles se réduit : les contraintes sont un moyen très générique de représenter les régularités qui gouvernent notre monde, que ce soit au niveau informationnel, physique, biologique ou social [Dec03]. Elles matérialisent une information qui existe indépendamment d'elles, et fixent les règles de ce qui est possible ou non. Les contraintes peuvent ainsi porter sur des objets, des personnes, des grandeurs physiques ou des abstractions mathématiques :

- *la somme des angles d'un triangle vaut 180 degrés* ;
- *une personne ne peut pas se trouver dans plusieurs lieux simultanément* ;
- *Suzanne ne peut pas être mariée à la fois avec Jacques et avec Bernard* ;
- *chaque ligne d'une grille de sudoku contient exactement une fois chacun des chiffres de 1 à 9 ;*
- etc.

Les contraintes ont quelques intéressantes propriétés qui leur sont caractéristiques [Bar]. Elles sont *non-directionnelles*, c'est-à-dire que les variables sur lesquelles une contrainte porte sont tantôt l'origine, tantôt la destination de l'information que celle-ci véhicule. Une autre de leur propriété est qu'elles sont *déclaratives* : elles spécifient une relation qui doit être vérifiée, sans pour autant définir la manière d'atteindre ce but. Elles sont également *additives*, i.e. l'ordre de définition des contraintes ne compte pas, seule la conjonction finale des contraintes posées est importante.

Étant donné un ensemble de contraintes, portant chacune sur un sous-ensemble donné d'un certain nombre de variables auxquelles on cherche à affecter une valeur, une question centrale est de savoir s'il existe une possibilité de satisfaire simultanément toutes ces contraintes et, le cas échéant, quelles valeurs on peut affecter à chaque variable pour être sûr de satisfaire toutes les contraintes. L'exemple 1.1 montre un problème bien connu[1] qui se base sur la notion de contraintes.

Exemple 1.1. *Cinq maisons sont alignées, de couleurs différentes. Dans chaque maison vit un homme de nationalité différente. Chaque personne boit une boisson différente, chaque personne fume un type de cigarettes différent, chaque personne élève un animal différent. Pour chaque maison, on cherche à connaître sa couleur, la nationalité de la personne qui y vit ainsi que sa boisson préférée, son type de cigarettes et son animal. Les contraintes sont les suivantes :*

1. *L'Anglais vit dans la maison rouge.*
2. *L'Espagnol possède un chien.*
3. *On boit du café dans la maison verte.*

[1] Ce problème est parfois attribué à Einstein ou à Lewis Carroll, et est aussi connu sous le nom de *problème du zèbre*.

4. *L'Ukrainien boit du thé.*
5. *La maison verte est à droite de la maison blanche.*
6. *Le fumeur de Old Gold possède des escargots.*
7. *On fume des Kools dans la maison jaune.*
8. *On boit du lait dans la maison du milieu.*
9. *Le Norvégien vit dans la première maison.*
10. *La personne qui fume des Chesterfield vit dans une maison à côté de celle où il y a un renard.*
11. *On fume des Kools dans une maison à côté de celle où il y a un cheval.*
12. *Le fumeur de Lucky Strike boit du jus d'orange.*
13. *Le Japonais fume des Parliament.*
14. *Le Norvégien vit à côté de la maison bleue.*

Une solution à ce problème est donnée par la table suivante :

Maison	*1*	*2*	*3*	*4*	*5*
Couleur	*jaune*	*bleu*	*rouge*	*blanc*	*vert*
Nationalité	*Norvégien*	*Ukrainien*	*Anglais*	*Espagnol*	*Japonais*
Boisson	*eau*	*thé*	*lait*	*jus d'orange*	*café*
Cigarettes	*Kools*	*Chesterfield*	*Old Gold*	*Lucky Strike*	*Parliament*
Animal	*renard*	*cheval*	*escargot*	*chien*	*zèbre*

Résolution de problèmes sous contraintes

Cette problématique est au cœur de plusieurs disciplines en informatique. La recherche locale et les algorithmes génétiques, la programmation mathématique, la programmation par contraintes sont autant de branches de l'informatique qui ont vocation à résoudre des problèmes construits autour de la notion de contrainte. Chacune de ses disciplines dispose d'un formalisme particulier et propose des méthodes différentes pour tenter de résoudre de manière générique le problème de la satisfaction des contraintes.

La recherche locale s'intéresse à la résolution de problèmes principalement combinatoires, qui impliquent de grouper, d'ordonner ou d'affecter un ensemble discret, fini, d'objets en satisfaisant un certain nombres de contraintes données [HS05]. Pour ce faire, la méthode appliquée consiste à modifier une solution existante, souvent générée de manière aléatoire, pour tenter d'améliorer la qualité de cette solution en se basant sur différentes heuristiques d'exploration. Cette recherche de qualité est cependant locale, c'est-à-dire qu'elle ne peut pas garantir une optimalité globale au vu de toutes les solutions potentielles existantes. Mais elle représente souvent la seule façon d'obtenir une solution pour les instances de problèmes de grande taille et très difficiles [HT06]. Les algorithmes génétiques sont également une méthode de recherche et d'optimisation, qui se base quant à elle sur le modèle de l'évolution naturelle [ROM09]. L'ensemble des solutions potentielles est représenté sous la forme d'une population d'individus, dont on va simuler l'évolution du génome au moyen de croisements et de mutations. Ces méthodes sont elles aussi adaptées aux problèmes difficiles dont on cherche une estimation de la meilleure solution [Tsa92].

La notion de *qualité* d'une solution, que l'on vient d'évoquer au paragraphe précédent, est également au cœur des problèmes sous contraintes que résout la programmation mathématique. Formalisée par une fonction sur les variables du problème, la recherche de son optimalité est au cœur des méthodes de résolution mises en oeuvre par cette autre discipline. Les méthodes employées sont par ailleurs spécifiques

au type de problème considéré et on trouve autant de sous-disciplines que de familles de problèmes : par exemple, la programmation en nombre entiers (IP), la programmation linéaire (LP), etc.

La programmation par contraintes est également une discipline dont le but est, comme son nom l'indique, de résoudre des problèmes posés sous la forme de contraintes. C'est cette branche de l'informatique, dédiée à la résolution de tels problèmes, qui nous intéresse ici particulièrement. Son objectif est la résolution exacte de problèmes sous contraintes dans lesquels la notion de qualité des solutions n'est pas obligatoirement présente.

1.1 La programmation par contraintes

La programmation par contraintes est un paradigme puissant pour résoudre efficacement des problèmes qui émergent dans de nombreux domaines de l'activité humaine, comme par exemple l'ordonnancement, la planification, les tournées de véhicules, la configuration, les réseaux et la bioinformatique [RvBW06]. Selon Freuder, l'un des pionniers de cette discipline, la programmation par contraintes est même l'une des approches les plus réussies à ce jour en informatique du Saint-Graal de la programmation : *l'utilisateur définit le problème, l'ordinateur le résout*[2]. Quoi qu'il en soit, c'est en tout cas une branche interdisciplinaire de l'informatique qui fait appel à des techniques issues de différents champs de connaissance, notamment l'intelligence artificielle, la logique, les bases de données, les langages de programmation et la recherche opérationnelle [Fre97].

Les premières idées dans ce domaine ont fait leur apparition dans les années 1960 et 70, à propos de la résolution de problèmes d'intelligence artificielle comme l'étiquetage de scène [Wal75], qui consiste à interpréter les lignes de dessins en 2D pour reconnaître les objets d'une scène en 3D, ou encore le dessin interactif [Sut63] au travers d'applications graphiques autorisant l'utilisation de figures géométriquement contraintes. Puis la principale étape fut la combinaison des contraintes avec la programmation logique pour donner la programmation logique à contraintes [Gal85, JL87] ; l'idée majeure derrière la programmation logique —et la programmation déclarative en général— étant en effet que l'utilisateur décrit ce qui doit être résolu et non pas comment le résoudre [Bar99, Apt03].

Parallèlement, les travaux fondateurs de Moore et de ses successeurs sur l'arithmétique des intervalles [Moo66] ont permis une amélioration très significative des méthodes de calcul numérique et de résolution des systèmes réels linéaires ou non-linéaires par ordinateur en s'affranchissant, grâce à la notion d'intervalle, des erreurs de calculs inhérentes à l'approximation en machine des nombres réels par les nombres flottants. Ce progrès dans la gestion par l'ordinateur du domaine réel a ouvert la voie à Cleary qui est alors parvenu à intégrer une arithmétique relationnelle au cadre de résolution de la programmation logique à contraintes, permettant ainsi pour la première fois d'exprimer des contraintes entre variables réelles [Cle87]. Une étape majeure fut ensuite la conception et le développement de Numerica [VHMD97], combinant un langage de modélisation adapté aux contraintes non linéaires et différentes avancées algorithmiques qui permirent une accélération conséquente du traitement des problèmes [BG06].

Aujourd'hui, la programmation par contraintes est un domaine qui progresse rapidement [RvBW06]. De nouveaux résultats de recherche sont constamment publiés et de nouveaux sujets d'investigation sont encore en train d'émerger. Les grandes directions des recherches actuelles concernent par exemple la résolution de problèmes à contraintes utilisant des quantificateurs, la notion d'explication des échecs

[2] « *Constraint programming is one of the closest approaches computer science has yet made to the Holy Grail of programming :* the user states the problem, the computer solves it. »[Fre97]

de résolution ou plus largement les interactions avec l'utilisateur, ou bien encore la mise au point de techniques d'apprentissage automatique.

1.2 Problématique de la thèse

La programmation par contraintes est un domaine de l'informatique qui a émergé dans les années 1980. Initialement dédiée à la résolution de problèmes d'intelligence artificielle à variables entières, *discrètes*, c'est dans les années 1990 que les chercheurs se sont attaqués aux problèmes sous contraintes à variables réelles, *continues*, en intégrant les travaux antérieurs de MOORE et ses successeurs à propos de l'approximation efficace du calcul sur les réels par une arithmétique des intervalles à bornes flottantes. Cependant, les problèmes mixtes, qui utilisent à la fois des variables entières et des variables réelles, n'ont été que très peu considérés jusqu'ici, du point de vue de la programmation par contraintes [Gel98, GF03]. L'état actuel des connaissances implique de résoudre un problème mixte soit en relaxant le problème pour résoudre un problème purement continu, soit en le discrétisant pour appliquer les méthodes discrètes.

Cette thèse se place du point de vue de la résolution de problèmes de satisfaction de contraintes continues. Les travaux réalisés doivent permettre d'avancer dans deux directions orthogonales :

- modifier le cadre de résolution de problèmes de satisfaction de contraintes continues pour permettre de formuler et de traiter d'une manière optimale les problèmes impliquant des variables et contraintes discrètes, en améliorant à la fois les capacités en termes de modélisation discrète des outils dédiés au continu et leurs capacités à résoudre les modèles ainsi formulés de la manière la plus efficace possible ;
- intégrer de manière efficace la notion d'objectif au schéma classique de résolution de la programmation par contraintes continues, qui historiquement s'intéresse à la satisfation des contraintes plutôt qu'à l'optimisation d'un objectif, mais dont certaines caractéristiques sont à la fois très intéressantes en même temps qu'indispensables à la résolution rigoureuse de problèmes d'optimisation non linéaire globale, impliquant des variables continues.

Concernant la modélisation des problèmes, il faut ainsi disposer d'un langage suffisamment expressif pour mélanger à la fois des contraintes discrètes et notamment les contraintes globales —e.g. `alldifferent` pour exprimer une différence de valeur entre variables ou encore `atmost` pour signifier qu'au plus p variables parmi n peuvent avoir une valeur donnée— et des contraintes non linéaires continues comme $x^2 + y^2 \leq a^2$ ou toute relation entre des grandeurs à valeurs réelles. Concernant la résolution, il faut étendre la programmation par contraintes et en particulier les techniques de consistance locale, qui permettent d'accélérer le processus en éliminant les valeurs détectées comme n'étant pas des solutions, et les heuristiques de branchement dont le rôle est de guider l'examen méthodique de toutes les solutions candidates. Enfin, il faut aussi intégrer ces nouvelles approches dans un cadre générique d'optimisation afin de se positionner dans le domaine de la résolution de problèmes d'optimisation mixtes (MINLP), où l'objectif est de calculer la valeur optimale que peut atteindre une certaine grandeur en fonction de tout ou partie des paramètres du problème.

1.3 Nos contributions

Cette section présente brièvement les principales contributions à cette problématique auxquelles nous avons pu participer à travers les travaux de recherche que nous avons réalisés au cours de ce doctorat :

- Nous avons eu l'opportunité de faire l'application des techniques de programmation par contraintes pour la modélisation et la résolution d'un problème de conception en robotique pour lequel la PPC n'avait jamais été mise à contribution jusqu'ici. Une fois que nous sommes parvenus à modéliser le problème, nous avons eu à modifier le cadre classique de résolution d'un solveur de contraintes continues destiné à la satisfaction de contraintes uniquement, pour y intégrer la notion de recherche rigoureuse d'optimum. Des expériences nous ont permis de mettre en évidence que notre approche était compétitive par rapport à d'autres outils, à la pointe des techniques d'aujourd'hui en matière d'optimisation.

- Nous avons mis en oeuvre une collaboration de deux solveurs, l'un spécialement dédié aux problèmes discrets, l'autre conçu en particulier pour les problèmes continus. L'idée directrice qui nous a poussé à entreprendre cette réalisation est extrêmement naturelle : il semble évidemment très prometteur de pouvoir profiter des avantages de chacun des outils dans leur champ de prédilection respectif. Les techniques orientées continu sont relativement peu efficaces pour traiter la partie discrète des problèmes mixtes et, réciproquement, les techniques orientées discret ne sont pas efficaces pour traiter la partie continue de ces problèmes. En pratique, cette collaboration nous permet d'abord de faciliter la phase de modélisation en offrant la possibilité de s'abstraire d'un mécanisme de modélisation rigide et difficile à utiliser. En outre, cette collaboration nous permet de résoudre plus efficacement certains types de problèmes mixtes.

- Nous avons ensuite cherché à faire un usage optimal d'une contrainte globale entière dans un solveur continu. Pour ce faire, nous avons effectué une comparaison des différentes modélisations possibles de la contrainte globale discrète `alldifferent`, ainsi que des techniques de filtrage qui sont dédiées à chacune d'entre elles. Cette étude expérimentale nous a conduit à résoudre des problèmes d'optimisation issus du monde réel et à implémenter une variété d'algorithmes de filtrage dans un solveur de contraintes continues. Il ressort de nos expériences qu'en ce qui concerne la résolution de ces problèmes difficiles du monde réel, même s'il reste avantageux, surtout en termes de robustesse, d'opter pour une modélisation globale de la contrainte et non son éclatement en une clique de contraintes binaires, il n'est pas intéressant d'opter pour un algorithme de filtrage trop complexe comme c'est le cas de l'algorithme de filtrage de type consistance de bornes le plus répandu, même dans sa version la plus optimisée.

- Nous avons également réfléchi à une spécialisation des techniques de filtrage basées sur l'arithmétique des intervalles, dédiée au cas des contraintes arithmétiques discrètes et mixtes. Réécrivant dans un premier temps de nouvelles définitions à certaines opérations de l'arithmétique des intervalles, en les adaptant au contexte particulier d'intervalles d'entiers, et réfléchissant à une prise en compte plus tôt au cours du calcul sur les intervalles des contraintes d'intégralité sur les variables entières, nous avons pu dans un deuxième temps implémenter ces nouvelles idées dans un solveur de contraintes continues. Cela nous a ainsi permis d'observer l'impact significatif que peuvent avoir ces optimisations naturelles du calcul, d'une part sur l'efficacité théorique des méthodes de filtrage basées sur cette arithmétique des intervalles, et d'autre part en pratique sur les temps effectifs de résolution des problèmes.

1.4 Organisation du document

La suite de ce manuscrit est découpée en six chapitres. Chacun de ces chapitres correspond bien évidemment à une thématique particulière. Nous donnons à présent un bref aperçu de ce que contient chacun d'entre eux :

II – État de l'art

(a) présentation des éléments essentiels de modélisation des problèmes qui nous intéressent en programmation par contraintes et en programmation mathématique également,

(b) présentation des techniques majeures de résolution de ces problèmes par la programmation par contraintes ;

III – Modélisation et résolution d'un problème de conception en robotique

(a) présentation du contexte du problème à résoudre,

(b) description détaillée du processus de modélisation du problème,

(c) transformation du cadre classique de résolution de problèmes de satisfaction de contraintes en un cadre de résolution dédié aux problèmes d'optimisation sous contraintes,

(d) réalisation d'une série d'expériences sur un ensemble d'instances du problème modélisé ;

IV – Collaboration de deux solveurs discret et continu

(a) présentation détaillée des outils permettant la modélisation et la résolution des problèmes, qu'il s'agisse de bibliothèques ou de logiciels,

(b) description du modèle de collaboration utilisé et de son implémentation,

(c) réalisation d'expériences sur une variété de problèmes mixtes de différents types ;

V – Utilisation de la contrainte discrète Alldifferent dans un solveur continu

(a) mise en avant des motivations qui nous ont poussés à entreprendre ces travaux ;

(b) modélisation de problèmes d'optimisation du monde réel en utilisant la contrainte discrète globale *alldifferent*,

(c) présentation des différentes possibilités de modélisation de la contrainte *alldifferent* et des différents algorithmes de filtrage possibles,

(d) réalisation d'expériences destinées à comparer les efficacités respectives des différentes modélisations et algorithmes de filtrage associés ;

VI – Spécialisation du filtrage basé sur l'arithmétique des intervalles

(a) description des travaux de APT et ZOETEWEIJ sur l'arithmétique des intervalles d'entiers ;

(b) présentation détaillée de modifications à apporter au cadre de filtrage dédié aux contraintes continues afin de traiter plus précisément les domaines entiers ;

(d) description de l'implémentation de ces techniques dans le solveur à intervalles *RealPaver* ;

(e) mise en évidence expérimentale des effets sur la résolution de problèmes mixtes des nouvelles techniques mises en oeuvre.

VII – Conclusion et perspectives

(a) synthèse des différents points abordés dans chacuns des chapitres ;

(b) ouverture sur des travaux futurs, perspectives de recherches.

II

ÉTAT DE L'ART

Renoncer à étudier soulage l'inquiétude :
Entre acquiescer et consentir, la nuance est bien petite !
Mais combien diffèrent le bien et le mal.

— LAO-TSEU, De la Voie et de sa vertu, XX.

La programmation par contraintes est un paradigme générique de résolution de problèmes dont l'une des principales caractéristiques est de distinguer deux grandes tâches principales :

- modéliser le problème à résoudre : cette première tâche consiste à formaliser le problème à l'aide de concepts dédiés, dans le but de le reformuler de manière plus abstraite ;

- résoudre le modèle ainsi obtenu : cette seconde étape vise à appliquer des techniques de résolution qui vont permettre de trouver une (des) solution(s) s'il en existe.

L'objet du présent chapitre est de présenter en détails chacune de ces deux étapes. Dans la première section, on s'intéressera donc à la modélisation des problèmes en programmation par contraintes, ainsi qu'en programmation mathématique. La seconde section sera quant à elle consacrée aux techniques de résolution de ces problèmes par la programmation par contraintes, qu'il s'agisse de méthodes dédiées aux problèmes discrets, continus ou mixtes.

2.1 Modélisation des problèmes

Dans cette première section, nous introduisons les principales notions utiles à la modélisation des problèmes à résoudre : du point de vue de la programmation par contraintes dans un premier temps, puis du point de vue de la programmation mathématique.

2.1.1 En programmation par contraintes

Cette sous-section introduit le formalisme utilisé en programmation par contraintes pour modéliser les problèmes à résoudre. Quelques exemples choisis dans la littérature seront ensuite présentés de façon à éclaircir le cas échéant les notions décrites.

2.1.1.1 Problème de satisfaction de contraintes

Au sein du cadre générique de résolution offert par la programmation par contraintes, le problème à résoudre est modélisé en termes de problème de satisfaction de contraintes [FM06] :

Définition 2.1 (CSP)**.** Un *problème de satisfaction de contraintes (CSP)* est un triplet ($\mathcal{X}$,$\mathcal{D}$,C) avec :
- $\mathcal{X}$ un ensemble $\{X_1, X_2, \ldots, X_n\}$ de *variables* ;
- $\mathcal{D}$ un ensemble $\{D_1, D_2, \ldots, D_n\}$ de *domaines* : $X_i \in D_i$, $i = 1, \ldots, n$;
- C un ensemble $\{C_1, C_2, \ldots, C_t\}$ de *contraintes* où chaque C_i définit un sous-ensemble du produit cartésien des domaines des variables sur lesquelles elle porte : $C_i(X_{i_1}, \ldots, X_{i_k}) \subseteq D_{i_1} \times \cdots \times D_{i_k}$.

La notion de domaine désigne l'ensemble de valeurs que peut potentiellement prendre une variable. Cette notion générique fait référence à des ensembles de natures potentiellement très différentes :
- un ensemble de valeurs symboliques : on peut vouloir représenter des couleurs, e.g. $D = \{rouge, bleu, vert\}$, ou encore des jours de la semaine, par exemple $D = \{lundi, mercredi, samedi\}$;
- un ensemble d'entiers non contigus : il est possible d'utiliser un ensemble d'entiers quelconques, e.g. $D = \{5, 8, 12\}$;
- un intervalle d'entiers : on peut également définir un ensemble d'entiers contigus, en compréhension, ne spécifiant que les bornes inférieure et supérieure de l'intervalle, par exemple $D = \{1, \ldots, 10\}$;
- un intervalle de réels[1] : de la même façon, on peut définir en compréhension un ensemble de réels, en ne déclarant que ses bornes inférieure et supérieure, e.g. $D = [-\pi, \pi]$.

Chaque variable possède ainsi un domaine initial qui contraint la valeur qu'on peut lui affecter. Si sa valeur est à chercher dans $\mathbb{N}$, dans $\mathbb{Z}$ ou dans tout ensemble discret de valeurs, on parlera de variable discrète. Si sa valeur est à trouver dans $\mathbb{R}$ ou dans tout sous-ensemble continu de valeurs, on parlera de variable continue. En fonction du type des variables du problème, on pourra donc distinguer trois grandes catégories de problèmes de satisfaction de contraintes : selon qu'il ne contient que des variables discrètes, que des variables continues, ou bien à la fois des variables discrètes et continues, on dira d'un CSP qu'il est *discret*, *continu* ou *mixte*.

Une fois qu'on a défini pour chaque variable son domaine initial, on peut poser des contraintes les reliant les unes aux autres, restreignant les combinaisons de valeurs autorisées. Là aussi il existe des contraintes de différentes natures :

[1] On reviendra dans la section suivante de ce chapitre sur l'utilisation en machine de cette notion d'intervalle de réels.

- contrainte *en extension* : étant donnée un sous-ensemble de variables, on définit explicitement la liste des tuples autorisés, par exemple $(a, b, c) \in \{(1, 2, 3), (4, 5, 6)\}$;
- contrainte *en intension* : une expression arithmétique définit une relation entre plusieurs variables, e.g. $(x - 5)^2 + (y - 2)^2 \leq 1$;
- contrainte *dynamique*[2] : on ajoute une précondition qui doit être vérifiée avant de pouvoir considérer la contrainte ;
- contrainte dite *globale* : on peut utiliser une relation prédéfinie, à la signification précise, comme par exemple[3] $alldif([a, b, c])$ qui impose à des variables d'un sous-ensemble de $\mathcal{X}$ de devoir prendre chacune une valeur différente de celle des autres ;
- contrainte *par morceaux* : on associe différentes contraintes à différentes parties du domaine des variables sur lesquelles elles sont définies ;
- contrainte *douce* : on autorise une telle contrainte à être violée par une solution potentielle, ce qui est utile dans le cas où le problème posé est trop contraint pour avoir une solution.

Définition 2.2 (Solution)**.** Une *solution* d'un CSP est un n-uplet $S = (s_1, s_2, \ldots, s_n)$, avec $s_i \in D_i \ \forall i$, pour lequel toute contrainte $C_j \in C$ est *satisfaite* : aucune contrainte $C_j(X_{j_1}, \ldots, X_{j_k})$ n'est violée quand chacune des variables X_{j_i} sur lesquelles elle porte se voit affecter sa valeur correspondante s_{j_i} dans la solution.

Selon les situations, résoudre un CSP peut consister à trouver une, toutes, ou la meilleure des solutions. Dans ce dernier cas, on doit d'ailleurs fournir un moyen d'évaluer la qualité d'une solution, moyen qui prend très souvent la forme d'une expression arithmétique définie sur un sous-ensemble des variables du problème et qu'on va utiliser pour quantifier la qualité d'une solution donnée.

2.1.1.2 Exemples

À présent, nous allons examiner quelques exemples de CSP, purement théoriques ou au contraire extrêmement concrets, pour comprendre en pratique comment on peut se servir de cette notion pour formaliser des problèmes.

Exemple 2.2. [Mac77] *Soit le CSP discret défini par :*
- $\mathcal{X} = \{X_1, X_2, X_3, X_4, X_5\}$,
- $\mathcal{D} = \{D_1, D_2, D_3, D_4, D_5\}$,
  avec $D_1 = D_2 = \{a, b, c\}$ *et* $D_3 = D_4 = D_5 = \{a, b\}$,
- $C = \{C_{13}, C_{23}, C_{34}, C_{35}, C_{45}\}$,
  avec $C_{13} = C_{23} = \{(b, a), (c, a), (c, b)\}$, $C_{34} = C_{35} = \{(a, b)\}$ *et* $C_{45} = \{(b, a)\}$.

Dans l'exemple ci-dessus, une contrainte C_{ij} entre deux variables X_i et X_j est définie en extension par l'ensemble des couples (u, v) qui la satisfont, le premier (resp. deuxième) élément du couple étant une valeur du domaine de X_i (resp. X_j). Remarquons qu'une autre façon d'exprimer en extension une contrainte est d'expliciter des couples non pas de valeurs, mais de couples variable-valeur e.g. $\{(X_i, u), (X_j, v)\}$.

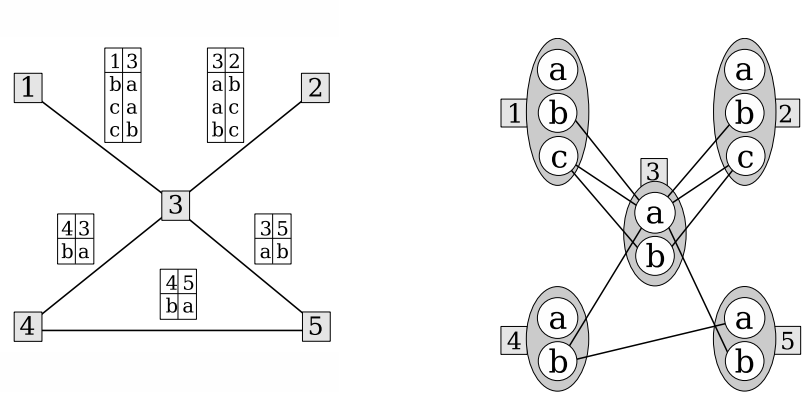
Ce problème utilise des variables à domaine symbolique et des contraintes en extension définissant explicitement les combinaisons de valeurs autorisées. En outre, ce problème n'a aucune solution. En effet, on constate que les paires de valeurs autorisées par les contraintes ne permettent pas de trouver une affectation de l'ensemble des variables du problème telle que toutes les contraintes soient satisfaites

[2] Ce type de contrainte est également appelée contrainte *conditionnelle*.

[3] Il existe plusieurs centaines de contraintes globales correspondant à des usages et des applications diverses et variées. Elles sont exhaustivement répertoriées dans [BCDP07] par exemple.

simultanément, e.g. il n'y a aucune valeur pour X_5 qui soit simultanément compatible avec les valeurs autorisées pour X_3 et X_4 par les contraintes C_{35} et C_{45}. Cet état de fait est d'ailleurs plus facilement visible si l'on considère une représentation graphique du problème :

Exemple 2.3. [Mac77] *Deux représentations possibles pour le CSP de l'exemple précédent :*

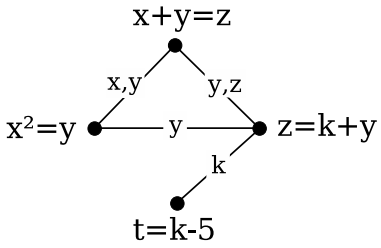


En effet, considérant par exemple le schéma de droite, où les sommets représentent les variables du problème et les combinaisons de valeurs autorisées sont exprimées par un arc reliant ces valeurs dans le domaine des variables concernées, on s'aperçoit qu'aucune valeur de X_5 n'est reliée à la fois à une valeur de X_3 et à une valeur de X_4. Le schéma de gauche, où les tables que portent les arcs sont les paires de valeurs autorisées pour les variables reliées, représente quant à lui une autre forme de visualisation possible pour le CSP, qui pourra s'avérer plus adéquate dans certaines circonstances, en particulier dans le cas où les domaines des variables du CSP comporteraient un grand nombre de valeurs. Avant de poursuivre, notons à propos de ce problème purement discret que les contraintes qui le définissent, ici données en extension sous la forme d'une liste de tuples valides, peuvent également dans le cas présent être exprimées en intension sous la forme $C_{xy} : x > y$, en utilisant l'ordre lexicographique.

L'exemple suivant est, quant à lui, purement continu et montre une autre représentation graphique possible pour un CSP :

Exemple 2.4. [Gou00] *Soit le CSP continu défini par :*

- $\mathcal{X} = \{x, y, z, k\}$,
- $\mathcal{D} = \{D_x, D_y, D_z, D_k\}$
  avec $D_x = [3, 4]$, $D_y = [4.5, 8]$, $D_z = [-7, 9]$
  et $D_k = [0, 12]$,
- $C = \begin{cases} x^2 = y \\ x + y = z \\ z = k + y \\ t = k - 5 \end{cases}$.

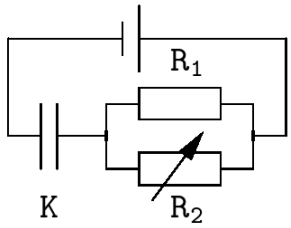


Les contraintes de ce CSP continu sont les sommets de son graphe représentatif, et un arc étiqueté par un ensemble de variables relie deux contraintes partageant ces variables. En outre, ce CSP n'admet aucune solution. Sans entrer dès maintenant dans les détails à propos du filtrage des domaines, qui sera abordé en profondeur dans la suite de ce chapitre, on peut d'ores-et-déjà constater que le domaine $D_x = [3, 4]$ de x ne contient aucun réel tel qu'il existe un y compatible au sens de la contrainte $x^2 = y$: intuitivement, on comprend bien que les valeurs compatibles de y se trouvent dans le domaine $[9, 16]$, dont l'intersection avec $D_y = [4.5, 8]$ est vide.

Pour finir, voici un problème avec une signification plus concrète, qui de plus s'avère être un CSP mixte i.e. faisant intervenir à la fois des variables discrètes et des variables continues :

Exemple 2.5. [HSG01] *Soit un circuit électrique composé d'une résistance* R_1 *de* $0.1\,M\Omega$ *connectée en parallèle à une résistance variable* R_2 *de valeur comprise entre* $0.1\,M\Omega$ *et* $0.4\,M\Omega$*. Un condensateur K est connecté en série à ces deux résistances. On dispose d'un kit de composants électroniques ne contenant que les condensateurs de valeurs* $1\,\mu F$*,* $2.5\,\mu F$*,* $5\,\mu F$*,* $10\,\mu F$*,* $20\,\mu F$ *et* $50\,\mu F$*. On cherche à ce que le temps de charge du condensateur soit compris entre* 0.5 *s et* 1 *s, i.e. que la tension aux bornes du condensateur atteigne 99% de sa valeur entre* 0.5 *s et* 1 *s. Le CSP correspondant est :*

- $\mathcal{X} = \{t, R_2, K\}$,
- $\mathcal{D} = \{D_t, D_{R_2}, D_K\}$
  avec $D_t = [0.5, 1]$, $D_{R_2} = [0.1 \times 10^6, 0.4 \times 10^6]$,
  et $D_K = \{1{\times}10^{-6}, 2.5{\times}10^{-6}, 5{\times}10^{-6}, 10{\times}10^{-6}, 20{\times}10^{-6}, 50{\times}10^{-6}\}$,
- $C = \{C_{ohm}, C_{charge}\}$
  avec C_{ohm} *:* $\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2}$ *la loi d'Ohm pour des résistances en parallèle,*
  et C_{charge} *:* $1 - exp(\frac{-t}{R \cdot K}) = 0.99$ *la loi de charge d'un condensateur en série.*



La seule possibilité pour le condensateur cherché ici est de $K = 10\ \mu F$. Il existe cependant une infinité de couples de valeurs réelles pour (t, R_2) qui satisfont toutes les contraintes du problème pour cette valeur de K. Dans ces conditions, on peut imaginer pertinent de formuler différemment le problème, par exemple en ajoutant l'objectif de minimiser R. Nous allons maintenant voir, dans la suite de cette section, que cette notion d'objectif est prépondérante en programmation mathématique.

2.1.2 En programmation mathématique

Le terme de *programmation* fait ici référence à la description de la planification et de l'ordonnancement d'activités au sein de grandes organisations, dans les années 1940 initialement. L'idée était, d'une part, d'exhiber des liens mathématiques entre des grandeurs variables représentant des quantités à produire ou des niveaux d'activité ; d'autre part, de trouver un moyen d'exprimer un objectif à maximiser ou minimiser en fonction des valeurs de ces grandeurs. La *programmation mathématique* permet de décrire la minimisation ou la maximisation d'une fonction-objectif définie sur des variables soumises à des contraintes [FGK02].

Définition 2.3 (Problème d'optimisation sous contraintes)**.** Un *problème d'optimisation sous contraintes* a la forme générique suivante :

$$
\begin{array}{lll}
\min & f(x_{k_1}, \ldots, x_{k_{n_f}}), & k_1, \ldots, k_{n_f} \in \mathbb{N}, \\
\text{s.c.} & g_i(x_{i_1}, \ldots, x_{i_{n_i}}) = 0, & i, i_1, \ldots, i_{n_i} \in \mathbb{N}, \\
 & h_j(x_{j_1}, \ldots, x_{j_{n_j}}) \leq 0, & j, j_1, \ldots, j_{n_j} \in \mathbb{N}, \\
 & \underline{x_m} \leq x_m \leq \overline{x_m}, & m \in \mathbb{N}, \\
 & \underline{x_p} \in \mathbb{R}, & p \in P \subseteq \mathbb{N}, \\
 & x_q \in \mathbb{Z}, & q \in Q \subseteq \mathbb{N}, P \cap Q = \varnothing.
\end{array}
$$

Un problème d'optimisation consiste à trouver les valeurs des variables qui minimisent la valeur correspondante de la fonction objectif. En outre, cette minimisation peut faire intervenir ou non des contraintes à satisfaire. Certaines variables peuvent être entières, d'autres réelles, et à l'instar d'un CSP,

un problème d'optimisation peut être discret, continu ou mixte selon le type des variables qui le constituent.

Le langage de modélisation en usage pour exprimer des problèmes d'optimisation se veut le plus simple possible et utilise une syntaxe basée essentiellement sur celle des mathématiques. Notons quand même l'utilisation de primitives de modélisation plus complexes issues de la programmation par contraintes, i.e. les contraintes globales, comme une tendance apparue récemment [Hoo07].

Considérons à présent quelques exemples de problèmes d'optimisation sous contraintes :

Exemple 2.6. [FGK02] *Une petite acierie doit décider de l'allocation pour la semaine prochaine (40 heures au total) de son laminoir. Celui-ci traite des plaques d'acier non finies et peut produire deux types de produits semi-finis, des bandes et des rouleaux. Le laminoir peut produire 200 bandes ou 140 rouleaux par heure, sachant qu'une bande rapporte $ 25 et un rouleau $ 30. Enfin, il ne faut pas produire davantage que le total des commandes programmées, à savoir 6000 bandes et 4000 rouleaux. Le problème d'optimisation correspondant à la question de déterminer les quantités respectives à produire pour maximiser le profit est le suivant :*

$$\begin{array}{ll} \max & 25x + 30y \\ \text{s.c.} & \frac{1}{200}x + \frac{1}{140}y \leq 40, \\ & 0 \leq x \leq 6000, \\ & 0 \leq y \leq 4000, \\ & x, y \in \mathbb{N}. \end{array}$$

Ce problème est purement discret, toutes les variables étant des variables entières. L'application de la méthode de résolution nous permet de trouver la solution optimale à ce problème comme correspondant à $x = 6000$ et $y = 1400$, pour une valeur de l'objectif de 192000.

Le problème suivant est quant à lui un problème d'optimisation mixte, impliquant donc à la fois des variables réelles et des variables entières :

Exemple 2.7. [DGDG95] *Un ressort hélicoïdal de compression doit être mis au point en minimisant son volume. Les grandeurs dont dépend ce volume sont x_1 le nombre de spires, variable entière ; x_2 le diamètre du fil, variable réelle avec seulement 42 valeurs possibles*[4] *; enfin, x_3 le diamètre moyen des spires, variable réelle. Le problème d'optimisation correspondant est le suivant :*

$$\begin{array}{ll} \min & 0.25\pi^2 x_2^2 x_3 (x_1 + 2) \\ \text{s.c.} & S - \frac{8KP_{max}x_3}{\pi x_2^3} \geq 0, \\ & l_{max} - \frac{P_{max}}{K} - 1.05(x_1 + 2)x_2 \geq 0, \\ & x_2 - d_{min} \geq 0, \\ & D_{max} - (x_2 + x_3) \geq 0, \\ & \frac{D}{d} - 3 \geq 0, \\ & \delta_{pm} - \delta_p \geq 0, \\ & \frac{P_{max} - P}{k} - \delta_w \geq 0, \\ & x_1 \in \mathbb{N}, \\ & x_2 \in \{0.0095, \ldots, 0.5000\}, \\ & x_3 \in \mathbb{R}. \end{array}$$

[4]Les valeurs possibles pour ce diamètre sont les suivantes (en pouces) : 0.0095, 0.0104, 0.0118, 0.0128, 0.0132, 0.0140, 0.0150, 0.0162, 0.0173, 0.0180, 0.0200, 0.0230, 0.0250, 0.0280, 0.0320, 0.0350, 0.0410, 0.0470, 0.0540, 0.0630, 0.0720, 0.0800, 0.0920, 0.1050, 0.1200, 0.1350, 0.1480, 0.1620, 0.1770, 0.1920, 0.2070, 0.2250, 0.2440, 0.2630, 0.2830, 0.3070, 0.3310, 0.3620, 0.3940, 0.4375, 0.5000.

Les lecteurs intéressés par une description plus approfondie de ce modèle et en particulier la signification physique des différentes contraintes utilisées pourront se reporter à [KK94]. Quant à la solution optimale pour ce problème, on obtient $x_1 = 10$, $x_2 = 0.283$ et $x_3 = 1.180$, correspondant à une valeur de l'objectif de 2.798.

2.1.3 Généralisation

Nous pouvons à présent généraliser la notion de problème de satisfaction de contraintes et celles de problème d'optimisation sous contraintes en une notion plus générale dont l'un et l'autre sont des instances particulières :

Définition 2.4 (CSOP)**.** Un *problème de satisfaction de contraintes et d'optimisation (CSOP)* est un quadruplet ($\mathcal{X}$,$\mathcal{D}$,$\mathcal{C}$, f) avec :

- $\mathcal{X}$ un ensemble $\{X_1, X_2, \ldots, X_n\}$ de *variables*,
- $\mathcal{D}$ un ensemble $\{D_1, D_2, \ldots, D_n\}$ de *domaines* tels que $X_i \in D_i$, $i = 1 \ldots n$,
- C un ensemble $\{C_1, C_2, \ldots, C_t\}$ de *contraintes* où chaque C_i est une relation portant sur un sous-ensemble de $\mathcal{X}$, $i = 1 \ldots t$,
- f une fonction-objectif, définie sur un sous-ensemble de $\mathcal{X}$, éventuellement constante.

C'est à l'étude et au développement de méthodes de résolution dédiées à ce type spécifique de problèmes que se consacre cette thèse, dans le cas particulier des CSOP qui impliquent à la fois des variables discrètes et des variables continues.

2.2 Techniques de résolution

Cette section introduit en détails le cadre générique de résolution mis en oeuvre par la programmation par contraintes. Les deux thèmes majeurs en sont :

- le filtrage des domaines ;
- l'exploration de l'espace de recherche.

2.2.1 Filtrage des domaines

Comme on a pu le constater brièvement dans la section précédente et notamment à propos de l'exemple 2.4, les contraintes permettent de filtrer les domaines des variables sur lesquelles elles portent. En effet, il est fait mention dès 1977, dans [Mac77], d'une façon dont le processus de résolution des CSP peut être accéléré en tenant compte des contraintes de manière plus précise. Mackworth y explique comment gagner du temps en faisant précéder la résolution proprement dite par une phase de réduction des domaines. En définissant certaines propriétés que respectent les solutions d'un problème donné, il est possible de vider les domaines des valeurs qui ne les respectent pas et ainsi d'accélérer[5] le processus de résolution.

Notion de consistance

Les pionniers de cette technique sont Waltz, Montanari, Mackworth, et Freuder dans [Wal72], [Mon74], [Mac77], et [Fre78]. Si c'est à Mackworth qu'on doit la notion de *consistance* pour désigner

[5]Cela sera détaillé au début de la sous-section suivante à propos de l'exploration de l'espace de recherche.

la propriété qui sert à distinguer les valeurs qu'on garde et celles qu'on élimine, il ne fait que poser un mot sur une notion qu'ont déjà définie précédemment WALTZ et MONTANARI. Ce sont les mêmes FREUDER et MACKWORTH qui reviennent en 1985 dans [MF85] sur la complexité de leurs propres algorithmes et de leurs évolutions. Dans [MH86], on trouvera ensuite une nouvelle version de leur algorithme, puis notamment dans [HDmT92] et dans [Bes94].

En effet, depuis son apparition cette notion de consistance a été utilisée sous de nombreuses déclinaisons. En fonction de leur efficacité à réduire les domaines et de leur rapidité à le faire, ces différentes évolutions du constat initial de MACKWORTH se sont transformées, ont été améliorées, revues et corrigées les unes après les autres et sont plus que jamais en usage aujourd'hui.

Les techniques de filtrage varient selon qu'il s'agit d'un CSP discret, continu ou mixte. Dans cette sous-section, nous passons en revue les différentes méthodes mises en application pour chaque type de CSP.

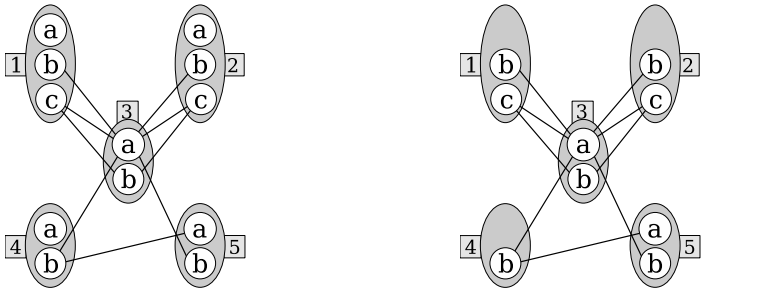
2.2.1.1 Domaines discrets

La consistance d'arc[6] [Mac77] permet de vérifier que toutes les valeurs d'un domaine disposent d'au moins une valeur dans les domaines des autres variables pour lesquelles une contrainte donnée est satisfaite :

Définition 2.5 (Consistance d'arc)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$, le domaine D d'une variable V est *arc-consistant* par rapport à une contrainte C ssi pour toute valeur v de D, il existe au moins une combinaison de valeurs pour les autres variables de C telle que C est satisfaite —une telle combinaison de valeurs est appelée *support*. Un CSP est *arc-consistant* ssi tous ses domaines le sont par rapport à toutes ses contraintes.

On peut retirer d'un domaine non arc-consistant ses valeurs sans support puisqu'elles ne participeront à aucune solution. En effet, pour ces valeurs *inconsistantes* il existe au moins une variable qui ne possède aucune valeur compatible au sens d'une contrainte donnée. C'est d'ailleurs l'idée de base du filtrage par consistance d'arc : retirer de tous les domaines toutes les valeurs qui ne sont pas arc-consistantes. Cependant, c'est un mécanisme algorithmiquement coûteux, même si de nombreuses améliorations de l'algorithme initial ont vu le jour les unes après les autres, notamment dans [Mac77], [MF85], [MH86], [HDmT92], [Bes94], et rassemblées récemment dans [Bes06].

Exemple 2.8. [Mac77] *Le CSP représenté à l'exemple 2.2, après filtrage par consistance d'arc :*



(i) CSP initial *(ii) CSP après filtrage par AC*

[6]Ce terme de consistance d'arc a pour origine le raisonnement sur la représentation du CSP sous forme de graphe, telle qu'introduite dans la précédente section. En réalité, il s'agit bel-et-bien d'un anglicisme, consacré par la communauté francophone de la programmation par contraintes, alors qu'une traduction fidèle du terme original *arc consistency* serait plutôt *cohérence d'arc* [Heu06].

Les valeurs a *des variables X_1, X_2 et X_4 ne sont compatibles avec aucune valeur de X_3 ni de X_5, et ont donc été supprimées.*

L'algorithme de filtrage par consistance d'arc dans sa version optimale $AC7$ a dans le pire des cas une complexité temporelle en $O(ed^2)$ et spatiale en $O(ed)$, avec e le nombre de contraintes et d la taille du plus grand domaine [BFR99].

Parallèlement, Montanari a défini dans [Mon74] une propriété très forte des solutions des réseaux de relations. C'est cette propriété, que Mackworth a rebaptisé ensuite *consistance de chemin*. Elle est bien plus forte que la consistance d'arc car elle permet cette fois de retirer des domaines des n-uplets de valeurs incompatibles, qui n'apparaissent dans aucune solution. Mais elle est en revanche plus difficile à implémenter et coûteuse à calculer.

Définition 2.6 (Consistance de chemin)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$, la paire de domaines (D_α, D_ω) des variables V_α et V_ω est *chemin-consistante* par rapport à un ensemble ordonné $\mathcal{E} = \{C_1, \ldots, C_m\}$ de contraintes C_i portant chacune sur un ensemble $\mathcal{V}_i$ de variables tel que $V_\alpha \in \mathcal{V}_1$, $V_\omega \in \mathcal{V}_m$ et $\mathcal{V}_i \cap \mathcal{V}_{i+1} \neq \emptyset \; \forall i < m$ ssi pour toute paire de valeurs de (D_α, D_ω) qui satisfait toutes les contraintes de $\mathcal{E}$ portant sur $\mathcal{V}_\alpha \cup \mathcal{V}_\omega$, il existe au moins une combinaison de valeurs des variables des $\mathcal{V}_i$ telle que toute contrainte de $\mathcal{E}$ est satisfaite.

À l'instar de la consistance d'arc (AC), la consistance de chemin (PC) permet de filtrer les domaines pour en éliminer les valeurs inconsistantes qui ne peuvent participer à aucune solution. Ce mécanisme est toutefois encore plus coûteux que le précédent. Son algorithme initial a lui aussi été revu et corrigé de nombreuses fois. Les deux algorithmes ont évolué de concert et, aujourd'hui, l'algorithme $PC5$ a dans le pire des cas une complexité temporelle en $O(n^3d^3)$ et spatiale en $O(n^3d^2)$, avec d la taille du plus grand domaine et n le nombre de variables [Sin96].

En outre, Montanari a démontré l'équivalence entre la consistance de chemin et la 2-consistance de chemin i.e. le cas particulier de la consistance de chemin quand les contraintes de $\mathcal{F}$ ne portent que sur une variable hormis V_α et V_ω. Cela signifie par exemple qu'il n'est pas nécessaire de considérer la consistance des chemins de longueur supérieure à 2 dans la représentation sous forme de graphe d'un CSP où chaque sommet matérialise une variable.

Exemple 2.9. [Bes06] *Soit un CSP $P = (\mathcal{X}, \mathcal{D}, C)$ avec $\mathcal{X} = \{X_1, X_2, X_3\}$, $\mathcal{D} = \{D_1, D_2, D_3\}$ où $D_1 = D_2 = D_3 = \{1, 2\}$, et $C = \{C_1, C_2\}$ où $C_1 : X_1 \neq X_2$ et $C_2 : X_2 \neq X_3$. P n'est pas chemin-consistant puisque il n'y a aucune valeur de X_2 qui satisfasse les contraintes de C pour les combinaisons de valeurs $(X_1, X_3) \in \{(1, 2), (2, 1)\}$. En revanche, le CSP $P' = (\mathcal{X}, \mathcal{D}, C \cup \{C_3 : X_1 = X_3\})$ est chemin-consistant.*

De nombreuses évolutions de ces deux consistances ont ensuite vu le jour. On trouve aujourd'hui un grand nombre de consistances discrètes différentes, chacune étant dotée d'une puissance de filtrage relativement plus forte ou plus faible que les autres, et permettant de trouver le bonne équilibre selon les cas entre puissance de filtrage et rapidité d'exécution[7].

2.2.1.2 Domaines continus

En présence de contraintes sur les réels, les techniques de filtrage[8] de domaines discrets que nous venons de présenter ne sont pas applicables en l'état. Elles doivent être retravaillées pour s'adapter à ce type de domaine, et notamment à l'impossibilité d'énumérer en pratique un domaine continu.

[7] L'Annexe A contient un complément d'information à ce propos qui n'est pas en rapport direct avec l'objet de cette thèse.

[8] Pour un domaine continu, on utilisera également le terme de *réduction* ou de *contraction*.

Représentation des nombres réels

Un problème fondamental de l'informatique est la représentation en machine des nombres réels, représentation par essence approchée puisqu'une quantité finie de bits ne permet de représenter qu'une quantité finie de nombres. Un nombre flottant f est défini de manière normalisée comme suit [IEE85] :

$$f = (-1)^s \cdot m \cdot 2^E$$

où s est le bit de signe, m la mantisse et E l'exposant. On définit ainsi l'ensemble $\mathbb{F}$ des nombres flottants. Remarquons l'inclusion $\mathbb{F} \subset \mathbb{R}$. Dans ces conditions, il n'y a pas d'autre choix que d'arrondir chaque réel au flottant le plus proche, malgré les erreurs d'approximation que cela peut engendrer. L'exemple 2.10 montre les divergences qu'on peut observer en pratique entre le calcul sur les réels d'une part et le calcul sur les flottants d'autre part, en fonction du nombre de bits utilisés pour représenter les nombres en machine.

Exemple 2.10. [Rum88], [Che07] *La fonction de* RUMP *a pour expression :*

$$f(x,y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + \frac{x}{2y}$$

La valeur exacte de l'évaluation de $f(77617, 33096)$ *est* $-54767/66192$*, soit environ -0.8273960599. La table ci-après contient les résultats obtenus pour* $f(77617, 33096)$ *en fonction du nombre de bits utilisés pour représenter les nombres, au moyen de l'outil* MuPad 2.5.3 *sur un processeur Intel Pentium M :*

Précision	Valeur
7	-2.47588×10^{27}
16	$-5.764075226 \times 10^{17}$
24	-134217728.0
32	-0.84375
64	$-0.8273960599\ldots$

Comme on vient de le voir, un nombre réel n'est pas forcément représentable en machine, ce qui introduit des erreurs d'évaluation dans les calculs. Cependant, il est possible de borner la valeur d'un réel par deux flottants et d'effectuer les calculs sur de tels intervalles pour pallier le problème des erreurs d'arrondis, grâce à l'arithmétique des intervalles.

Arithmétique des intervalles

C'est l'idée qui sous-tend les travaux fondateurs de CLEARY, utilisant le moteur de *Prolog* pour réaliser des encadrements précis de variables contraintes par des prédicats arithmétiques comme `mul(X,Y,Z)` ou `add(X,Y,Z)`. En intégrant ainsi les travaux réalisés auparavant par MOORE sur l'arithmétique des intervalles [Moo66], CLEARY étend aux domaines continus la notion de filtrage par consistance des domaines discrets, en tant qu'opération de réduction de la largeur de l'intervalle qui définit le domaine de la variable[9]. A l'origine, l'arithmétique des intervalles a été introduite par MOORE dans les années 1960. L'idée principale en est de remplacer les calculs peu fiables sur les nombres flottants par des calculs fiables sur des intervalles, i.e. des paires de nombres flottants. La plupart du travail réalisé à cet époque dans ce

[9] [CTN04] puis [CTN05] amènent à conclure que si ce n'est pas contre-productif de chercher à exploiter la notion de *trou* dans les intervalles, le gain n'est pas intéressant devant la complexification des algorithmes que cela entraîne ; [BG06] rappelle quant à lui le manque d'efficacité en pratique du raisonnement sur les unions d'intervalles.

domaine a été consacrée à l'étude des propriétés mathématiques de structures algébriques basées sur la notion d'intervalles, à l'élaboration d'algorithmes de calcul d'approximations fiables et précises des solutions faisables ou optimales de systèmes de contraintes linéaires et non-linéaires et à l'étude de la convergence de ces algorithmes [BVH97]. Ainsi, depuis [Cle87], [OV90] et l'intégration efficace dans les langages de programmation logique comme *Prolog* du calcul sur les intervalles mis au point quelques décennies plus tôt, il est possible de raisonner sur des CSP dont les domaines des variables ne sont pas discrets et énumérés, mais continus et seulement exprimés comme un encadrement par une borne inférieure et une borne supérieure définies comme des nombres flottants.

Définition 2.7 (Intervalle à bornes flottantes)**.** Étant donné l'ensemble $\mathbb{F} \cup \{-\infty, +\infty\}$ des flottants auquel on a ajouté $-\infty$ et $+\infty$, et la relation $\leq$ sur cet ensemble, l'*intervalle à bornes flottantes* $[a, b]$ est l'ensemble des nombres réels compris entre a et b :

$$[a, b] = \{x \in \mathbb{R} \ : \ a \leq x \leq b\} \ \forall a, b \in \mathbb{F} \cup \{-\infty, +\infty\}$$

On trouve dans [Lho93] les bases pour l'adaptation effective aux domaines continus des techniques discrètes de filtrage par consistance, avec par exemple la définition d'une consistance d'arc aux bornes des domaines. Puis, dans [BMVH94], l'arithmétique des intervalles —telle que définie par Moore dans [Moo66]— est mise en lumière pour exhiber les principales caractéristiques utiles des intervalles à bornes flottantes et de l'ensemble $\mathbb{I}$ de ces intervalles.

Définition 2.8 (Enveloppe)**.** Soit r un sous-ensemble de $\mathbb{R}$. L'*enveloppe* de r, notée $\square(r)$, est le plus petit intervalle à bornes flottantes contenant r :

$$\begin{array}{rrcl} \forall\, a \in \mathbb{R}, & \square(a) & = & [\lfloor a \rfloor_{\mathbb{F}}, \lceil a \rceil^{\mathbb{F}}] \\ \forall\, a \leq b,\ a, b \in \mathbb{R}, & \square([a, b]) & = & [\lfloor a \rfloor_{\mathbb{F}}, \lceil b \rceil^{\mathbb{F}}] \\ \forall\, a_1, \ldots, a_n \in \mathbb{R}, & \square(\{a_1, \ldots, a_n\}) & = & \square([\min(a_1, \ldots, a_n), \max(a_1, \ldots, a_n)]) \end{array}$$

où $\lceil a \rceil^{\mathbb{F}}$ (resp. $\lfloor a \rfloor_{\mathbb{F}}$) est le plus petit (resp. grand) élément de $\mathbb{F} \cup \{-\infty, +\infty\}$ plus grand (resp. petit) ou égal à a.

Étant donnée une opération arithmétique élémentaire $\circ \in \{+, -, \times, /\}$ et deux intervalles I_1, I_2, l'opération correspondante sur les intervalles est définie de la manière suivante :

$$I_1 \ \circ \ I_2 = \ \square\{x \ \circ \ y \ : \ x \in I_1, \ y \in I_2\}$$

Ce qui nous donne en pratique les règles de calcul suivantes, pour deux intervalles $[a, b]$ et $[c, d]$:

- $[a, b] + [c, d] = \square([a + b, c + d])$
- $[a, b] - [c, d] = \square([a - d, b - c])$
- $[a, b] \times [c, d] = \square([\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)])$
- $[a, b] \ / \ [c, d] = \square([\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)]),\ 0 \notin [c, d]$

En outre, l'arithmétique des intervalles nous permet de la même façon de définir une extension aux intervalles des fonctions réelles :

Définition 2.9 (Extension aux intervalles)**.** Étant donnée une fonction réelle $f : \mathbb{R}^n \to \mathbb{R}$, la fonction sur les intervalles $F : \mathbb{I}^n \to \mathbb{I}$ est une *extension aux intervalles* de f ssi $\forall I \in \mathbb{I}^n$, $\{f(x) : x \in I\} \subseteq F(I)$.

Notons qu'il existe de nombreuses extensions aux intervalles, basées sur des transformations symboliques ou des relaxations numériques : les formes de Bernstein, les formes de Horner et l'extension de Taylor par exemple [Gou00], [BG06]. La Figure 2.1 donne un aperçu de la différence observable selon l'extension utilisée pour l'évaluation d'une fonction.

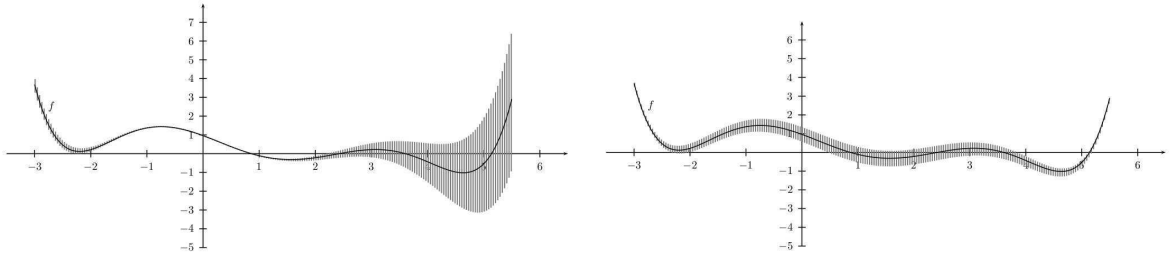


Figure 2.1 – [Gou00] Évaluation de $f(x) = (x^6/4 - 1,9x^5 + 0,25x^4 + 20x^3 - 17x^2 - 55,6x + 48)/50$ en forme naturelle (gauche), en forme de BERNSTEIN (droite), pour une largeur d'intervalle unitaire de 0,02.

Adaptation de la notion de consistance

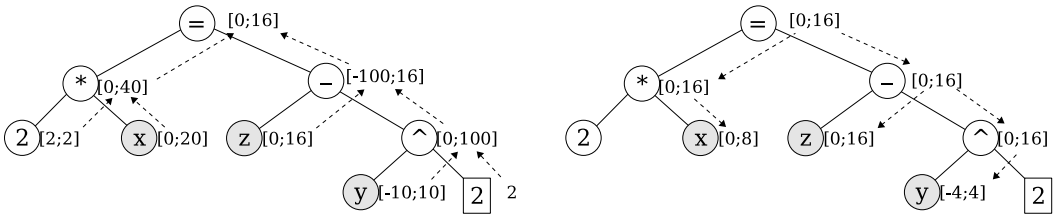
Du point de vue de la résolution de contraintes, il est dès lors possible de raisonner sur les domaines continus à la place de raisonner sur les domaines discrets, et d'utiliser l'arithmétique des intervalles et notamment l'extension aux intervalles des fonctions réelles pour définir de nouvelles techniques de filtrage par consistance, dédiées aux domaines continus cette fois.

Définition 2.10 (Consistance d'enveloppe)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$, un domaine $D_i \in \mathcal{D}$ est *enveloppe-consistant* par rapport à une contrainte $C \in C$ ssi $D_i = \square(D_i \cap \{a_i \in \mathbb{R} \ : \ \exists a_1 \in D_1 \ldots \exists a_{i-1} \in D_{i-1} \exists a_{i+1} \in D_{i+1} \ldots \exists a_n \in D_n : C(a_1, a_2, \ldots, a_n)\})$.

La consistance d'enveloppe, aussi appelée consistance 2B [Lho93], est une relaxation de la consistance d'arc. Il s'agit d'une propriété locale, liée aux bornes du domaine d'une variable au niveau d'une seule contrainte : une contrainte c est 2B consistante si pour n'importe quelle variable x de c, il y a des valeurs dans les domaines de toutes les autres variables de c pour lesquelles c est satisfaite quand x vaut la borne inférieure (resp. supérieure) de son domaine [CDR99]. Mais à cause des erreurs d'arrondis inhérentes au calcul avec des nombres flottants, obtenir l'enveloppe d'un ensemble de réels est une tâche difficile en elle-même, difficulté que l'algorithme de filtrage dédié *HC3* parvient en partie à contourner en rendant 2B consistant le système qu'on aura obtenu en décomposant les contraintes du problème en contraintes aussi simples que possible[10] : par exemple, la contrainte $x + y \cdot z = t$ pourra être décomposée en deux contraintes $y \cdot z = \alpha$ et $x + \alpha = t$, introduisant une nouvelle variable α, et le filtrage sera réalisé sur le système obtenu ainsi par décomposition [BGGP99]. Enfin, l'algorithme *HC4*, très proche de *HC3*, est capable d'aboutir au même résultat mais sans avoir pour cela recours à l'introduction de telles variables auxiliaires —l'exemple 2.11 ci-après montre comment cet algorithme *HC4* utilise une contrainte pour réduire les domaines des variables de cette dernière [Gou00]. Considérant une expression arithmétique de la contrainte sous forme d'arbre où les feuilles sont des variables ou des constantes, les noeuds étant des opérateurs arithmétiques et la racine un symbole de relation, le filtrage d'un pavé par *HC4* est réalisé en deux phases, l'évaluation ascendante et la propagation descendante :

Exemple 2.11. [Gou00] *Filtrage par l'algorithme* HC4 *pour la contrainte* $2x = z - y^2$ *avec les domaines* $D_x = [0, 20]$, $D_y = [-10, 10]$ *et* $D_z = [0, 16]$ *:*

[10] On utilisera le terme de contraintes *primitives*.



(i) Evaluation ascendante *(ii) Propagation descendante*

La première phase se résume à faire remonter *les valeurs des feuilles jusqu'à la racine en utilisant simplement les règles du calcul arithmétique de* Moore *sur les intervalles ; la seconde phase consiste à faire* redescendre *vers les feuilles la valeur inférée pour la racine à partir de la relation représentée et de la valeur avérée de ses opérandes, grâce à l'inversion des opérations arithmétiques.*

Cependant, l'évaluation d'expressions arithmétiques sur des intervalles perd en précision à mesure qu'augmente le nombre d'occurrences d'une même variable dans une contrainte, et si l'algorithme *HC*4 est capable de réduire efficacement les domaines dans des contraintes où chaque variable n'apparaît qu'une seule fois, ce n'est plus le cas quand certaines variables ont plusieurs occurrences dans une même contrainte [BG06]. En effet, si on considère par exemple la contrainte $x + x = 0$ avec $x \in [-1, 1]$, la règle de réduction que nous donne l'arithmétique des intervalles est $D_x := D_x \cap (-D_x)$, qui reste complètement sans effet ici. Une deuxième consistance dédiée aux domaines continus a donc été mise au point pour combler ce problème de l'inefficacité du filtrage par consistance d'enveloppe pour les contraintes où des variables ont plusieurs occurrences, la consistance de pavé [BMVH94]. Sa définition[11] est la suivante :

Définition 2.11 (Consistance de pavé)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$, un domaine $D_i \in \mathcal{D}$ est *pavé-consistant* par rapport à une contrainte $C_j \in C$ de la forme $f_j(x_{j_1}, \ldots, x_{j_n}) = 0$ ssi $\forall i$, $D_i = \square(\{a_i \in D_i \ : \ 0 \in F_j(D_1, D_2, \ldots, D_{i-1}, \square(\{a_i\}), D_{i+1}, \ldots, D_m)\})$, où F_j est une extension aux intervalles de f.

Notons que dans le cas où la variable considérée n'apparaît qu'une seule fois dans la contrainte, la consistance d'enveloppe est équivalente et moins coûteuse à calculer que la consistance de pavé [CDR99]. En outre, la consistance de pavé est plus faible que la consistance d'arc, comme le montre l'exemple suivant :

Exemple 2.12. [HMK97] *Soit la contrainte* $x_1 + x_2 - x_1 = 0$ *et les domaines* $D_1 = D_2 = [-1, 1]$. D_2 *n'est pas arc-consistant puisqu'il n'existe aucune valeur* $n_1 \in D_1$ *telle que* $n_1 + 1 - n_1 = 0$. *Pourtant* D_2 *est pavé-consistant puisque* $([-1, 1] + [-1, -1^+] - [-1, 1]) \cap [0, 0]$ *et* $([-1, 1] + [1^-, 1] - [-1, 1]) \cap [0, 0]$ *ne sont pas vides.*

De plus, la consistance de pavé est une relaxation de la consistance d'enveloppe dont le principe est de remplacer le test de satisfaction de contraintes sur les réels par une procédure de réfutation sur les intervalles, et pour atteindre la consistance de pavé, l'algorithme de filtrage *BC3* cherche à trouver des valeurs consistantes en bordure de D_i [BMV94]. Comme le montre en pratique l'exemple 2.13, l'implémentation standard de cet algorithme utilise une procédure de recherche dichotomique :

Exemple 2.13. [BG06] *Soit* $y - x^2 = 0$ *une contrainte avec* $D_x = [0, 1]$ *et* $D_y = [0, 4]$. *La borne inférieure de* D_y *est pavé-consistante puisque* 0 *appartient à l'intervalle* $0 - D_x^2 = [-1, 0]$. *À l'inverse, la borne supérieure est inconsistante. Le domaine de y peut alors être itérativement divisé en 2 pour y trouver la valeur consistante la plus grande :*

[11] Cette définition ne fait référence qu'à des contraintes d'égalité, mais elle peut être ensuite facilement étendue aux inégalités en considérant que $f(x) \leq 0 \Leftrightarrow f = z,\ z \in [-\infty, 0]$ et $f(x) \geq 0 \Leftrightarrow f = z,\ z \in [0, +\infty]$.

$$
\begin{array}{llllll}
[2,4] & - & D_x^2 & = & [1,4] \not\ni 0 & \rightarrow \quad [2,4] \textit{ est éliminé} \\
[0,2] & - & D_x^2 & = & [-1,2] & \\
[1,2] & - & D_x^2 & = & [0,2] & \\
[1.5,2] & - & D_x^2 & = & [0.5,2] \not\ni 0 & \rightarrow \quad [1.5,2] \textit{ est éliminé} \\
\ldots & & & & \ldots & \ldots
\end{array}
$$

Finalement, on obtient le nouvel intervalle $D_y = [0, 1]$.

Enfin, l'algorithme de filtrage *BC4* améliore l'algorithme *BC3* en prenant en compte à la fois le fait que *HC4* est efficace en ce qui concerne les contraintes sans aucune variable apparaissant plusieurs fois, mais aussi à l'inverse la caractéristique de *BC3* de ne pas être sensible négativement aux occurrences multiples de variables [BGGP99]. L'idée est d'appliquer l'algorithme *HC4* sur chacune des contraintes jusqu'à stabilité[12] du domaine des variables à occurrence unique, avant d'appeler l'algorithme *BC3* seulement pour les contraintes où apparaissent plusieurs fois certaines variables, et uniquement pour réduire le domaine de ces dernières [Gou00]. L'exemple 2.14 montre l'impact positif d'une telle stratégie, en comparant les filtrages par *HC4*, *BC3* et *BC4* pour des contraintes avec et sans occurrences multiples de variables.

Exemple 2.14. [BG06] *Soient* $(x, y) \in [-10, 10]$ *et la contrainte* $c : x^2 + y^2 = 1$*, ainsi qu'une contrainte équivalente à c,* $c' : xx + y^2 = 1$*. La table suivante montre le résultat du filtrage en utilisant soit l'algorithme HC4, soit l'algorithme BC3, soit l'algorithme BC4. Dans chaque cas, on donne les domaines obtenus après réduction et le nombre de calculs sur les intervalles qui ont été effectués.*

	HC4		*BC3*		*BC4*	
c	$[-1,1]^2$	*15*	$[-1,1]^2$	*1178*	$[-1,1]^2$	*15*
c'	$[-10,10]^2$	*16*	$[-1,1] \times [-\sqrt{2}, \sqrt{2}]$	*1238*	$[-1,1] \times [-\sqrt{2}, \sqrt{2}]$	*625*

2.2.1.3 Utilisation des contraintes mixtes

Quand il s'agit de filtrer les domaines, une façon générique de traiter les contraintes mixtes est de plonger les variables entières dans le domaine des intervalles et de traiter les variables discrètes comme des variables continues. Après chaque filtrage, on s'assure alors que les domaines de ces variables continues particulières soient tronquées de façon à obtenir l'intervalle à bornes entières le plus large possible.

Les seuls travaux qu'on peut trouver dans la littérature concernant des techniques de filtrage dédiées à l'utilisation des contraintes mixtes sont ceux de GELLE et FALTINGS [Gel98], [GF03]. On y distingue trois types de contraintes mixtes, associant à chacune une façon différente de filtrer :

- les contraintes de type catalogue qui associent une variable discrète et une variable continue, e.g. $(x, y) \in \{(1, [0, 100]), (2, [0, 1000])\}$ (C_{M1}) ;
- les contraintes numériques qui utilisent un opérateur de discrétisation, par exemple $n = \lceil x/y \rceil$ (C_{M2}) ;
- les autres contraintes numériques faisant intervenir à la fois variables discrètes et continues, comme $v = \frac{4}{3}\pi R^3$ (C_{M3}).

Pour les contraintes mixtes du premier type, l'idée est de les transformer en contraintes discrètes, grâce à un opérateur qui associe à chaque grandeur intervalle une grandeur discrète qui la représente, dans le but de pouvoir appliquer les techniques dédiées aux domaines discrets. En ce qui concerne les contraintes

[12] Cette notion d'itération du filtrage ou *propagation* sera du reste abordée en détails plus loin dans cette section.

mixtes du deuxième type, qui utilisent des fonctions de transtypage réel vers entier, elles sont approximées par des contraintes numériques, e.g. on remplace $i = \lceil r \rceil$ par $r \leq i$ et $i \leq r + 1$, avant d'appliquer les méthodes dédiées aux contraintes continues. Enfin, concernant les contraintes du troisième type, elles sont traitées comme des contraintes continues dans lesquels les variables discrètes ont été au préalable temporairement instanciées au point-médian de leur domaine courant. On reviendra sur ce principe en fin de chapitre pour davantage d'explications —cf p.29.

2.2.1.4 Propagation

Quand un domaine a été réduit, la question se pose des répercussions sur les autres domaines avec lesquels sont partagées des contraintes. Ceci est brièvement illustré par l'exemple introductif suivant :

Exemple 2.15. [Gra05] *Soient les domaines $D_x = [0, 5]$ et $D_y = [0, 4]$, et soient deux contraintes $y = x+1$ (c_1) et $y = x - 1$ (c_2). L'utilisation de c_1 pour un filtrage par consistance d'enveloppe provoque les réductions :*
- $D_y = [0, 4] \cap [0, 5] + 1 = [1, 4]$
- $D_x = [0, 5] \cap [0, 4] - 1 = [0, 3]$

Puis l'utilisation de c_2 provoque les réductions :
- $D_y = [1, 4] \cap [0, 3] - 1 = [1, 2]$
- $D_x = [0, 3] \cap [1, 4] + 1 = [2, 3]$

Enfin, l'utilisation à nouveau de c_1 entraîne la réduction :
- $D_y = [1, 2] \cap [2, 3] + 1 = \varnothing$

Le problème n'a aucune solution.

Ainsi, certaines réductions peuvent être très avantageusement enchaînées car si leur effet seul n'est pas très important, l'effet conjugué de toutes peut avoir une capacité filtrante bien plus grande. Ce mécanisme d'itération du filtrage pose cependant un certain nombre de questions. Par exemple, l'itération va-t-elle se terminer ? Les filtrages de domaines finissent-ils obligatoirement par ne plus rien ôter ? Comment optimiser l'enchaînement des différents filtrages ? etc. Le cadre général des itérations chaotiques va permettre de répondre à nombre de ces questions.

Cadre formel

A l'image de [GH88], [MR91] et [Fal94] notamment, nombre d'études théoriques ont déjà été réalisées pour fournir à la propagation un cadre formel, avec théorèmes et propriétés prouvés, de façon à pouvoir se doter d'un mécanisme réellement contrôlable et efficace. [Apt00] et [Bes06] sont eux aussi le reflet d'une démarche de formalisation dont le but est de mieux connaître et maîtriser les tenants et aboutissants de la propagation[13].

Définition 2.12 (Itération)**.** Soit un ordre partiel $(D, \sqsubseteq)$ avec un plus petit élément $\perp$ et un ensemble fini de fonction $\mathcal{F} = \{f_1, \ldots, f_k\}$ sur D. Une *itération de $\mathcal{F}$* est une séquence infinie de valeurs $d_0, d_1, \ldots$ définie inductivement par

$$\begin{cases} d_0 &= \perp \\ d_j &= f_{i_j}(d_{j-1}), \quad j \neq 0 \end{cases}$$

où i_j est un élément de [1..k]. Une séquence croissante $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \ldots$ *finit par se stabiliser à d* si $d_{i+1} = d_i$ pour un certain $j \geq 0$ avec $i \geq j$.

[13] La suite de cette section est tirée essentiellement de [Apt00].

Ce formalisme s'illustre ici de la façon suivante : l'ordre partiel s'applique aux CSP au regard d'une consistance donnée, le plus petit élément $\perp$ étant le CSP initial et les d_i étant obtenus par filtrages successifs, et les fonctions sont les algorithmes de filtrage de domaine. Par la suite ne seront étudiées que les itérations de fonctions ayant certaines propriétés :

Définition 2.13 (Monotonie)**.** Soit un ordre partiel $(D, \sqsubseteq)$ avec un plus petit élément $\perp$ et une fonction f sur D. f est *monotone* si $x \sqsubseteq y$ implique $f(x) \sqsubseteq f(y)$ $\forall x$ $\forall y$.

Le rôle de la monotonie est éclairé par la simple observation suivante :

Lemme 2.1 (Stabilisation)**.** *Soit un ordre partiel $(D, \sqsubseteq)$ avec un plus petit élément $\perp$ et $\mathcal{F}$ un ensemble fini de fonctions monotones sur D, et supposons qu'une itération de $\mathcal{F}$ finit par se stabiliser en un point fixe d commun aux fonctions de $\mathcal{F}$. Alors d est le plus petit point fixe commun aux fonctions de $\mathcal{F}$.*
Preuve. *Considérons e un point fixe commun aux fonctions de $\mathcal{F}$ et prouvons que $d \sqsubseteq e$. Soit $d_0, d_1, \ldots$ l'itération en question. Pour un certain $j \geq 0$, on a $d_i = d$ pour $i \geq j$. Il suffit de prouver par induction sur i que $d_i \sqsubseteq d$. Il est évident que c'est vrai pour $i = 0$ puisque $d_0 = \perp$. Supposons à présent que c'est vrai pour un certain $i \geq 0$. On a $d_{i+1} = f_j(d_i)$ pour un certain $j \in [1..k]$. Du fait de l'hypothèse d'induction et de la monotonicité, il s'ensuit que $f_j(d_i) \sqsubseteq f_j(e)$, et donc que $d_{i+1} \sqsubseteq e$ puisque e est un point fixe de f_j.* □

Fixons à présent un ordre partiel $(D, \sqsubseteq)$ avec un plus petit élément $\perp$, et un ensemble fini de fonction $\mathcal{F} = \{f_1, \ldots, f_k\}$ sur D. On s'intéresse à calculer le plus petit point fixe commun aux fonctions de $\mathcal{F}$. C'est dans ce but qu'on étudie l'Algorithme 2.1, inspiré par un algorithme similaire tiré de [MR99].

ALG. 2.1 : Propager($\mathcal{F}$,x)

```
1 G := F ;
2 d := domaine(x) ;
3 tant que G ≠ ∅ faire
4 |   choisir g ∈ G ;
5 |   e := g(d) ;
6 |   G := G ∪ update(G, F, g, x, d, e) ;
7 |   d := e ;
8 fin
```

Dans cet algorithme, l'ensemble de fonctions *update* est tel que :
(i) $d = e \Rightarrow update(\mathcal{G}, \mathcal{F}, g, x, d, e) = \mathcal{G} - \{g\}$,
(ii) $\forall f \in \mathcal{F} : f(e) \neq e \Rightarrow f \in update(\mathcal{G}, \mathcal{F}, g, x, d, e)$.

Intuitivement, (i) signifie qu'il devra être impossible à la fonction de choix de retourner indéfiniment une fonction qui ne réduit pas le domaine, et (ii) que l'invariant de boucle de l'algorithme est que toute fonction $h \in \mathcal{F} - \mathcal{G}$ est telle que $h(d) = d$. On garantit de cette façon une réduction optimale.

Théorème 2.1 (Propagation)**.** *L'algorithme de propagation est tel que :*
(i) Toute exécution qui termine calcule dans d le plus petit point fixe commun des fonctions de $\mathcal{F}$.
(ii) Toute exécution termine.
Preuve. *(i) Montrons que le point fixe calculé est effectivement le plus petit des points fixes. Soit α un point fixe commun. Prouvons par induction que $\alpha \in d$ à chaque pas de l'algorithme. C'est vrai pour le*

premier pas. Pour le k-ième pas, on a $\alpha \in d_k$ *par hypothèse et* $g(\alpha) \in g(d_k)$ *car g est monotone. Comme* $\alpha \in g(\alpha)$ *et* $d_{k+1} = g(d_k)$*, il vient* $\alpha \in d_{k+1}$*.*
(ii) Soit l'ordre produit de $(D, \sqsupset)$ *et* $(\mathcal{N}, <)$ *défini sur les éléments de* $D \times \mathcal{N}$ *par*

$$(d_1, n_1) < (d_2, n_2) \Leftrightarrow d_1 \sqsupset d_2 \vee (d_1 = d_2 \wedge n_1 < n_2)$$

où on utilise $\sqsupset$ *l'ordre inverse défini par* $d_1 \sqsupset d_2$ *ssi* $d_2 \sqsubseteq d_1 \vee d_2 \neq d_1$*. Les couples* $(x, Card(\mathcal{G}))$ *calculés par l'itération sont strictement décroissants au sens de cet ordre produit, puisqu'à chaque fois soit le domaine est réduit, soit le nombre d'éléments dans la liste* $\mathcal{G}$ *diminue. Or par hypothèse* $(D, \sqsubseteq)$ *est fini donc* $(D, \sqsupset)$ *est bien fondé, ce qui implique la terminaison.* □

Ainsi, une forme possible de l'algorithme *update*, respectant les spécifications ci-dessus, peut être celle de l'Algorithme 2.2 :

ALG. 2.2 : Update($\mathcal{G}, \mathcal{F}, g, x, d, e$)

```
1 si d=e alors
2 |   retourner G − {g}
3 sinon
4 |   retourner G ∪ {f ∈ F : x ∈ variables(f)}
5 fin
```

Ajoutons que la fonction *update* décrite ci-dessus peut être améliorée en tenant en compte de l'idempotence des fonctions de filtrage :

Définition 2.14 (Idempotence)**.** Soit une fonction f sur un domaine D. f est *idempotente* si $f(f(x)) = f(x)\ \forall x \in D$.

Il ne sera donc pas judicieux d'ajouter à l'ensemble $\mathcal{G}$ des fonctions de filtrage idempotentes puisque, quelle que soit la réduction qu'elles viennent de réaliser, les rappeler sur le domaine qu'elles ont elles-mêmes réduit ne peut avoir aucun effet supplémentaire.

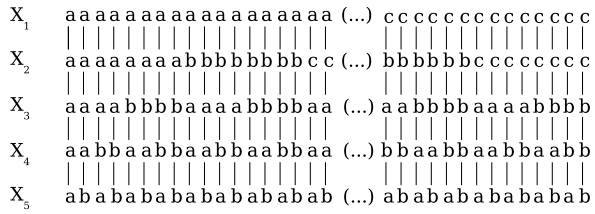
2.2.2 Exploration de l'espace de recherche

Envisageons à présent l'étape suivante du processus de résolution. Une fois les domaines réduits par filtrage, il est loin d'être garanti que chaque variable n'a plus qu'une seule valeur dans son domaine, s'il s'agit d'une variable discrète, ou que l'intervalle a été réduit jusqu'à avoir la largeur demandée, s'il s'agit d'une variable continue —ni même que son domaine a été rendu vide par le filtrage dans le cas d'un CSP sans solution. Non seulement les consistances vues plus haut ne sont pas efficaces de la même façon pour tous les problèmes, mais surtout elles sont loin de garantir de filtrer suffisamment pour obtenir des domaines ne contenant que des solutions. Aussi doit-on se doter d'un autre mécanisme que le filtrage pour tester toutes les combinaisons possibles de valeurs consistantes de façon à pouvoir garantir de n'avoir perdu aucune solution.

2.2.2.1 CSP discrets

Historiquement, l'exploration exhaustive de l'espace de recherche précède l'existence des techniques de filtrage par consistance. Aujourd'hui, les deux ne vont pas l'un sans l'autre, l'objectif étant de trouver un bon compromis entre puissance du filtrage et complexité de l'exploration. L'idée de base en matière de stratégie de recherche des solutions est de générer une affectation des variables du problème, la tester, générer une autre affectation, la tester, etc. : c'est la technique dite de *generate-and-test*.

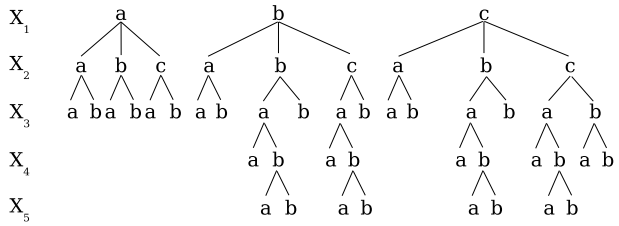
Exemple 2.16. [Mac77] *Trace du generate-and-test pour le CSP de l'exemple 2.2 :*



Toutes les combinaisons de valeurs sont générées et testées, intégralement, une par une.

Cette technique très naïve est facilement améliorable en ce que chaque combinaison testée est évidemment très similaire à la précédente. C'est de ce constat qu'est né le mécanisme de *backtracking* — *retour-arrière* en français dans le texte— qui reprend la combinaison précédemment testée pour l'étendre de manière incrémentale, tout en vérifiant à chaque étape la satisfaction des contraintes :

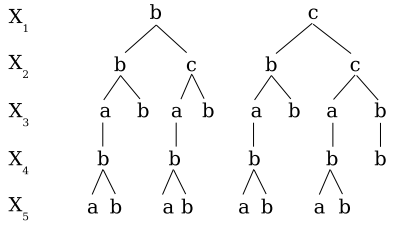
Exemple 2.17. [Mac77] *Trace du backtracking pour le CSP de l'exemple 2.2 :*



A chaque étage de l'arbre, on affecte une nouvelle valeur à une variable, on teste la satisfaction des contraintes, puis on continue en profondeur avec la variable suivante, jusqu'à ce qu'il n'y ait plus de variable à affecter et tant que les contraintes peuvent être satisfaites.

Le produit du déroulement de cet algorithme de recherche est appelé *arbre de recherche*. On peut en outre se rendre compte du gain obtenu par filtrage des domaines en observant l'arbre de recherche correspondant au même CSP que ci-dessus, mais lorsqu'un filtrage par consistance d'arc a été au préalable appliqué aux domaines initiaux :

Exemple 2.18. *Arbre de recherche pour le même CSP mais après filtrage par consistance d'arc :*



En effet, il n'y a plus que 2 valeurs à tester dans les domaines de X_1 et X_2, une seule dans le domaine de X_4.

Les deux paramètres qui régissent principalement le déroulement du backtracking concernent la méthode de choix de variable et celle de choix de l'ordre d'instanciation de ces valeurs. Des stratégies de choix de variables communément employées en matière de CSP discrets sont celle dite du *fail-first*, où l'on choisit en priorité la variable de plus petit domaine, ou bien celle du *most-constrained*, où l'on préfère choisir d'abord la variable la plus contrainte. L'idée qui gouverne ces choix est de chercher à éviter d'explorer une longue branche de l'arbre se soldant par un échec. Pour cela, on privilégie les variables qui sont les plus sujettes à provoquer des inconsistances pendant le filtrage. En ce qui concerne l'affectation d'une valeur à la variable choisie, la règle générale consiste simplement à créer une branche pour chaque valeur du domaine. Dans certains cas cependant, on peut définir un ordre sur les valeurs à affecter, selon leur sémantique dans le problème considéré [Heu06]. On peut également limiter le nombre de branches créées en un noeud, en n'énumérant qu'un nombre réduit de valeurs du domaines et en conservant les autres dans une dernière branche.

On peut citer [DP88] comme exemple d'alternative au backtracking : une utilisation judicieuse d'une consistance d'arc directionnelle alliée avec un bon choix de variable permet de résoudre certains CSP sans retour-arrière. D'autres se sont également employés à circonscrire le besoin de backtrack aux plus petites portions possibles du CSP, on peut citer notamment les méthodes de consistance adaptative utilisant des coupe-cycles [Dec92]. Enfin, de nombreuses améliorations de l'algorithme initial ont vu le jour depuis son apparition dans les années 1970 et sont présentées dans l'Annexe A.

2.2.2.2 CSP continus

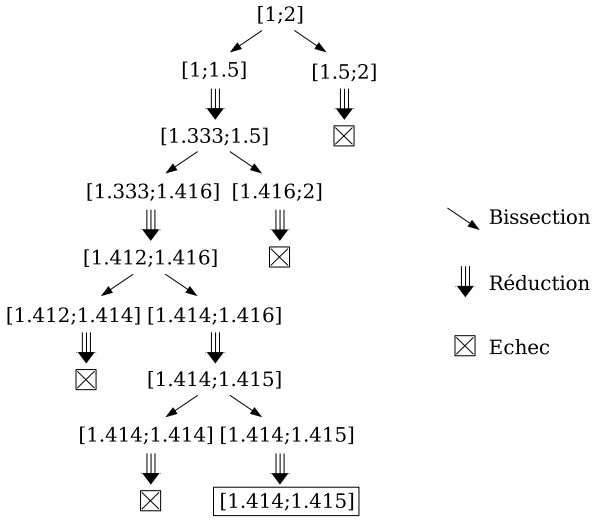
Dans le cas où les domaines sont continus, il paraît à juste titre très coûteux d'énumérer chacun des intervalles de largeur minimale qu'il contient, si bien que l'algorithme générique de recherche par backtracking doit être modifié pour s'adapter aux domaines continus.

Branch-and-prune

L'idée de l'algorithme d'exploration dédié aux domaines continus est de procéder à une *bissection* plutôt qu'à une *énumération* pour parcourir l'espace de recherche : à chaque noeud, on déclenche la phase de propagation des réductions, puis lorsque les domaines ne peuvent plus être significativement réduits, alors on choisira une variable dont le domaine va être coupé en deux parties. Chaque moitié sera ensuite explorée dans une branche de l'arbre, de la même façon qu'avec un domaine discret chaque branche correspond à une valeur du domaine initial —les autres domaines étant simplement recopiés à l'identique. Cet algorithme dit de *branch-and-prune* que nous donnons ici —cf Algorithme 2.3— est celui donné par Van Hentenryck dans [HMK97], évolution des travaux fondateurs de Cleary vis-à-vis de la résolution de problèmes à variables réelles [Cle87] mais aussi très nettement dans la continuité des premières idées de Moore sur le sujet [Moo79].

À l'inverse du cas d'un CSP discret, une heuristique de choix de variables en usage en ce qui concerne la résolution d'un CSP continu va donner la préférence à la variable dont le domaine est le plus large. Ceci peut s'expliquer par le fait qu'un domaine large signifie qu'on a du mal à le réduire grâce aux contraintes, et que le couper ainsi en deux permet de le réduire malgré l'inefficacité potentielle des contraintes. Notons par ailleurs qu'une autre heuristique largement en usage consiste simplement à choisir les variables les unes après les autres, en tourniquet —*round-robin* en anglais.

Exemple 2.19. [Cle87] *Arbre de résolution pour le CSP de contrainte*[14] $X \cdot X = 2$ *avec* $X \in [1, 2]$ *:*



L'intervalle de départ est successivement réduit et coupé en deux jusqu'à ce qu'on obtienne un intervalle de largeur minimale.

ALG. 2.3 : BranchAndPrune(*Constraint*[] C, *Box* B_0)

```
1  B ← filtrer(C, B0)
2  si non estVide(B) alors
3  |  si estAssezFin(B) alors
4  |  |  retourner {B}
5  |  sinon
6  |  |  (B1, B2) ← split(B)
7  |  |  retourner BranchAndPrune(C, B1) ∪ BranchAndPrune(C, B2)
8  |  fin
9  sinon
10 |  retourner ∅
11 fin
```

Ajoutons que la résolution d'un problème de satisfaction de contraintes doté d'une fonction-objectif requiert de maintenir à jour des bornes fiables pour cette dernière [Han92]. De cette manière, on peut

[14] Cet exemple est tiré des travaux fondateurs de CLEARY [Cle87], où le langage de modélisation n'englobe pas encore la notion de puissance d'une variable.

utiliser la fonction-objectif de la même manière que les autres contraintes pour filtrer les domaines. En outre, de nouvelles bornes peuvent être obtenues de manière rigoureuse dans les régions satisfaisant les contraintes et on utilise à cet effet des algorithmes de preuve d'existence [Kea96], l'idée étant de mettre à jour l'encadrement de la valeur optimale connue à chaque fois qu'on découvre un point qui satisfait toutes les contraintes du problème et dont la valeur de la fonction-objectif est meilleure que celle qu'on avait pu obtenir jusque là. Enfin, l'ordre de traitement des pavés ainsi que la stratégie de branchement sont deux paramètres importants pour l'efficacité du processus de résolution, puisqu'en effet le but est de trouver le plus rapidement possible les pavés qui améliorent le plus l'encadrement de la fonction-objectif et ainsi filtrer le plus efficacement possible l'espace de recherche [CR97]. Ce point sera abordé plus en profondeur au chapitre suivant.

2.2.2.3 CSP mixtes

Après avoir étudié en détails la notion de stratégie d'exploration, dédiée d'abord aux domaines discrets puis aux domaines continus, intéressons-nous à présent à la question d'une stratégie d'exploration dédiée à la résolution de CSP mixtes.

Une façon générique de résoudre un CSP mixte est d'exploiter les similarités dans l'exploration de l'espace de recherche entre CSP discrets et continus :

- les CSP continus sont résolus en utilisant des méthodes basées sur les intervalles, créant par bissection un arbre de domaines continus qui sont réduits par filtrage grâce à l'arithmétique des intervalles ;
- les CSP discrets sont résolus par énumération de combinaisons de valeurs, déclarées consistantes, de chacune des variables du problème.

Dans le cas général, explorer l'espace de recherche d'un CSP mixte va donc consister simplement à énumérer les variables discrètes et bissecter les variables continues, de façon à produire un arbre de recherche mixte.

Approches dédiées en programmation par contraintes

Dans la littérature, seuls Gelle et Faltings ont proposé par ailleurs une nouvelle combinaison des deux méthodes discrète et continue, destinée à résoudre en particulier les problèmes mixtes sous contraints dotés de larges ensembles de solutions contiguës [Gel98], [GF03]. Par bissection est généré un arbre d'intervalles duquel un arbre des points-médians est obtenu, arbre des point-médians qui va permettre une énumération systématique de tous les domaines. Concrètement, la façon dont sont obtenus ces points-médians va dépendre du type de domaine considéré :

- un domaine discret comme $\{a, b, d\}$, issu par filtrage d'un domaine $\{a, b, c, d\}$, sera représenté, en utilisant l'ordre lexicographique par exemple, par la liste d'intervalles $[1, 2], [4, 4]$ —le point médian d'un intervalle d'entiers $[u, v]$ pouvant par la suite être obtenu par le calcul $\lfloor (u + v)/2 \rfloor$;
- un domaine continu, réprésenté par un intervalle $[c, d]$, $c, d \in \mathbb{R}$, sera identifié par un point-médian calculé grâce à la formule $(c + d)/2$.

L'algorithme explore l'espace de recherche de la façon suivante. D'abord, on vérifie la consistance d'arc d'une affectation des variables aux points-médians correspondants aux domaines courants. Si ce test échoue, on obtiendra de nouveaux points-médians à tester en utilisant successivement l'intervalle à gauche du point-médian puis l'intervalle à sa droite, puis le suivant dans la liste éventuelle, jusqu'à obtenir des intervalles d'une largeur suffisamment fine. Le choix de la prochaine variable à backtracker est effectué en privilégiant d'abord les variables discrètes intervenant dans des contraintes mixtes, puis

les autres variables discrètes et enfin les variables continues. En outre, l'utilisation d'un filtrage de type *forward checking* permet d'améliorer les performances obtenues, spécialement quand il s'agit d'un problème dont les solutions sont réparties en un faible nombre de régions contiguës.

Méthodes issues de la programmation mathématique

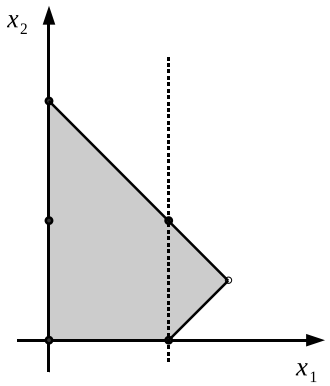
Pour conclure cet état de l'art, il nous faut à présent faire mention de techniques de résolution mises au point en programmation mathématique et dédiées à la résolution de problèmes mixtes sous contraintes, avec objectif à optimiser (MINLP).

Un aspect important de l'optimisation globale en programmation mathématique est l'utilisation de *relaxations linéaires*, qui permettent de trouver une borne inférieure à la valeur de la fonction-objectif à minimiser et ainsi rendre possible d'écarter des pavés pour lesquels cette borne inférieure est supérieure à l'évaluation f^{best} de la meilleure solution trouvée jusqu'ici [Neu04]. Cette phase de linéarisation est réalisée en deux étapes, comme par exemple dans un outil récent comme *Couenne* [BLL*09] :

- une phase de reformulation a lieu en début de résolution et consiste à reformuler le problème en un problème équivalent mais symboliquement plus simple grâce à l'adjonction de variables auxiliaires ;
- la phase de linéarisation proprement dite est quant à elle réalisée à chaque noeud de la résolution et va permettre d'obtenir, à partir du problème reformulé, un système d'inéquations linéaires qui encadre l'ensemble admissible.

En outre, les *plans de coupe*, aussi appelés *coupes*, ou *inégalités valides*, sont des inégalités linéaires qui peuvent être inférées à partir d'un ensemble de contraintes entières ou mixtes, inégalités qui sont ensuite ajoutées à l'ensemble de contraintes du problème [Hoo06]. D'une part, ces coupes vont permettre de couper l'exploration de solutions non entières d'une relaxation linéaire, et ainsi faire gagner cette dernière en précision. D'autre part, elles vont également permettre de réduire le backtracking, en excluant des valeurs qui ne sont pas admissibles pour les variables. L'exemple suivant illustre cette dualité :

Exemple 2.20. [Hoo06] *Ensemble admissible (zone grisée) de la relaxation linéaire du système* $x_1 + x_2 \leq 2$, $x_1 - x_2 \leq 0$, *avec les domaines* $x_i \in \{0, 1, 2\}$, *et un plan de coupe* $x_1 \leq 1$ *(droite en pointillés) :*



Comme le montre la figure ci-dessus, $x_1 \leq 1$ *est une inégalité valide parce qu'elle n'ôte aucun point entier de l'ensemble admissible. Cependant, il s'agit également d'une coupe pour la relaxation linéaire du problème, qu'elle renforce.*

III

MODÉLISATION ET RÉSOLUTION D'UN PROBLÈME DE CONCEPTION EN ROBOTIQUE

Celui qui en toutes choses suit la Voie et règle ses principes sur la Voie,
Parce qu'il aspire à l'union suprême,
La Voie l'accueille avec joie.
Aussi sa conduite, ses projets, ses oeuvres ou ses abstentions,
Ont-ils d'heureux résultats.

— LAO-TSEU, De la Voie et de sa vertu, XXIII.

Dans ce chapitre, nous modélisons et résolvons le problème de la prédiction des erreurs de position et de rotation de l'organe terminal d'un système robotique mécanique, aussi appelées *erreurs de pose*, en fonction du jeu dans ses articulations ou, plus précisément, du jeu introduit dans les articulations par des imperfections de fabrication.

Après avoir montré comment le problème peut être exprimé sous la forme de deux problèmes continus d'optimisation sous contraintes, nous montrons comment le classique cadre continu de résolution en programmation par contraintes peut être modifié pour nous permettre de gérer rigoureusement et globalement ces problèmes d'optimisation difficiles. En particulier, nous présentons des expériences préliminaires dans lesquelles notre optimiseur est très compétitif comparé aux méthodes les plus efficaces présentées dans la littérature, tout en fournissant des résultats plus robustes.

3.1 Contexte

La précision d'un système robotique mécanique est un aspect crucial pour l'exécution d'opérations qui demandent de l'exactitude. Cependant, cette précision peut être affectée négativement par des erreurs de positionnement et/ou d'orientation, aussi appelées erreurs de pose, de l'organe terminal du manipulateur. Une principale source d'erreurs de pose réside dans le jeu qu'il y a dans les articulations, qui introduit des degrés de liberté supplémentaires de déplacement entre les éléments pairés par les jointures du manipulateur. Il apparaît en outre que réduire ces déplacements augmente à la fois les coûts de production et la difficulté d'assemblage du mécanisme. Actuellement, cette problématique est d'ailleurs une tendance majeure de recherche en robotique [FS98, Inn02, MZL09]. Une façon de traiter le jeu dans les articulations est de prévoir son impact sur les erreurs de pose. Dans ce but, les déplacements impliqués ainsi qu'un ensemble de paramètres du système peuvent être modélisés par deux problèmes d'optimisation sous contraintes, dont le maximum global va caractériser l'erreur de pose maximum. Il est donc obligatoire d'obtenir un encadrement rigoureux de ce maximum global, ce qui élimine la possibilité d'utiliser des optimiseurs locaux. Cette modélisation peut être vue comme un ensemble de variables prenant leurs valeurs dans des domaines continus, accompagné d'un ensemble de contraintes et d'une fonction-objectif. Il s'avère qu'un tel modèle est difficile à résoudre expérimentalement : les temps de résolution peuvent augmenter de manière exponentielle avec la taille du problème, qui dans ce cas varie avec le nombre d'articulations du manipulateur.

S'il existe des travaux de recherche traitant de la modélisation du jeu dans les articulations et la prévision des erreurs dues à ce phénomène [FS98, Inn02], peu d'entre eux ont pour objet les techniques de résolution [MZL09], et aucun ne fait appel à la programmation par contraintes. Dans ce chapitre, nous investiguons donc l'usage de la programmation par contraintes pour résoudre de tels problèmes. En particulier, nous combinons l'algorithme classique de branch-and-bound avec l'arithmétique des intervalles et les techniques de filtrage associées. L'algorithme de branch-and-bound nous permet de gérer la partie optimisation du problème tandis que les temps de résolution peuvent être diminués grâce à ces techniques de filtrage. En outre, la fiabilité du processus est garantie par l'utilisation de l'arithmétique des intervalles, ce qui est intrinsèquement requis par l'application qui nous intéresse ici.

3.2 Modélisation du problème

Dans le cadre des travaux que nous présentons dans ce chapitre, on considère des manipulateurs sériels composés de n pivots, n liaisons et un organe terminal. La Figure 3.1 représente un tel manipulateur doté de deux articulations, nommé manipulateur *RR*, tandis que la Figure 3.2 montre un pivot affecté par du jeu. Nous considérerons ici que ce jeu dans les articulations apparaît lors d'erreurs de production. En conséquence, le problème d'optimisation à résoudre aura pour but de trouver l'erreur maximum en position et en rotation de l'organe terminal d'un manipulateur étant donnée une configuration de ce dernier.

3.2.1 Organe terminal sans jeu dans les articulations

Afin de décrire de manière unique l'architecture du manipulateur, i.e. le placement et l'orientation relatives des axes des articulations, nous utilisons la nomenclature Denavit-Hartenberg [DH55]. Dans ce but, les liaisons sont numérotées de 0 à n , la $j^{\text{ème}}$ articulation étant définie comme celle reliant la $(j-1)^{\text{ème}}$ liaison avec la $j^{\text{ème}}$. Le manipulateur est ainsi considéré comme étant composé de $n+1$ liaisons

et n articulations, où la liaison 0 est la base fixe et la liaison n l'organe terminal. Ensuite, on définit des repères $\mathcal{F}_j$ d'origine O_j et d'axes X_j, Y_j, Z_j, le repère $\mathcal{F}_j$ étant associé à la $(j-1)^{\text{ème}}$ liaison pour j variant de 1 à $n+1$. Le torseur suivant transforme $\mathcal{F}_j$ en $\mathcal{F}_{j+1}$:

$$\mathbf{S}_j = \begin{bmatrix} \mathbf{R}_j & \mathbf{t}_j \\ \mathbf{0}_3^T & 1 \end{bmatrix}, \tag{3.1}$$

où $\mathbf{R}_j$ est une matrice de rotation 3×3, $\mathbf{t}_j \in \mathbb{R}^3$ pointe de l'origine de $\mathcal{F}_j$ vers $\mathcal{F}_{j+1}$, et $\mathbf{0}_3$ est le vecteur zéro à trois dimensions. De plus, $\mathbf{S}_j$ peut s'exprimer de la façon suivante :

$$\mathbf{S}_j = \begin{bmatrix} \cos\theta_j & -\sin\theta_j \cos\alpha_j & \sin\theta_j \sin\alpha_j & a_j \cos\theta_j \\ \sin\theta_j & \cos\theta_j \cos\alpha_j & -\cos\theta_j \sin\alpha_j & a_j \sin\theta_j \\ 0 & \sin\alpha_j & \cos\alpha_j & b_j \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

où α_j, a_j, b_j et θ_j représentent respectivement la torsion de la liaison, sa longueur, son décalage et l'angle avec le pivot. Remarquons que a_1, b_1, a_2 et b_2 sont dépeints par la Figure 3.1 pour le manipulateur correspondant où α_1 et α_2 sont par contre nuls puisque les axes des pivots sont parallèles.

À condition que les articulations soient parfaitement rigides dans toutes les directions sauf une, qu'il n'y ait pas de jeu dans les articulations, que les liaisons soient parfaitement rigides et que la géométrie du manipulateur robotique soit connue avec exactitude, la pose de l'organe terminal en fonction du repère fixe $\mathcal{F}_0$ s'exprime ainsi :

$$\mathbf{P} = \prod_{j=1}^{n} \mathbf{S}_j, \tag{3.3}$$

Cependant, nous devons inclure de petites erreurs dans l'équation (3.3) si nous voulons prendre en compte le jeu dans les articulations.

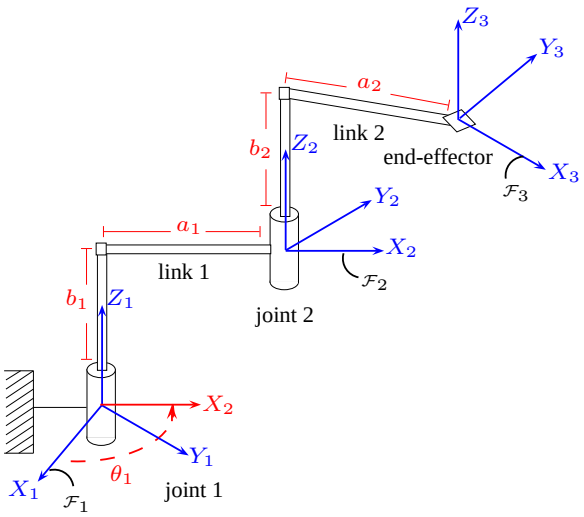


Figure 3.1 – [BSG*10] Manipulateur sériel composé de deux liaisons pivots.

3.2.2 Modélisation des erreurs dues au jeu dans les articulations

Prenant en compte le jeu dans les articulations, le repère $\mathcal{F}_j$ associé à la liaison $j-1$ est transformé en $\mathcal{F}'_j$. À condition que l'erreur soit petite, cette erreur sur la pose de l'articulation j par rapport à l'articulation $j-1$ peut être représentée par le torseur de faible déplacement décrit dans l'équation (3.4) :

$$\delta\mathbf{s}_j \equiv \begin{bmatrix}\delta\mathbf{r}_j \\ \delta\mathbf{t}_j\end{bmatrix} \in \mathbb{R}^6, \qquad (3.4) \qquad\qquad \delta\mathbf{S}_j = \begin{bmatrix}\delta\mathbf{R}_j & \delta\mathbf{t}_j \\ \mathbf{0}_3^T & 0\end{bmatrix}, \qquad (3.5)$$

où $\delta\mathbf{r}_j \in \mathbb{R}^3$ représente la faible rotation changeant $\mathcal{F}_j$ en $\mathcal{F}'_j$, tandis que $\delta\mathbf{t}_j \in \mathbb{R}^3$ pointe de l'origine de $\mathcal{F}_j$ vers celle de $\mathcal{F}'_j$. Il sera utile de représenter $\delta\mathbf{s}_j$ comme la matrice 4×4 donnée par l'équation (3.5) dans laquelle $\delta\mathbf{R}_j \equiv \partial(\delta\mathbf{r}_j \times \mathbf{x})/\partial\mathbf{x}$ est la matrice produit vectoriel de $\delta\mathbf{r}_j$. Intuitivement, la meilleure façon de modéliser le jeu dans les articulations dans une jointure est de borner inférieurement et supérieurement les erreurs associées. Si on considère ces deux bornes comme identiques, cela fait en général apparaître 6 paramètres pour borner le torseur d'erreur $\delta\mathbf{s}_j$. En conséquence, les bornes de l'erreur s'écrivent de la manière suivante :

$$\delta r_{j,X}^2 + \delta r_{j,Y}^2 \le \delta\beta_{j,XY}^2, \qquad (3.6) \qquad\qquad \delta t_{j,X}^2 + \delta t_{j,Y}^2 \le \delta b_{j,XY}^2, \qquad (3.8)$$

$$\delta r_{j,Z}^2 \le \delta\beta_{j,Z}^2, \qquad (3.7) \qquad\qquad \delta t_{j,Z}^2 \le \delta b_{j,Z}^2, \qquad (3.9)$$

où $\delta\mathbf{r}_j \equiv \left[\delta r_{j,X}\, \delta r_{j,Y}\, \delta r_{j,Z}\right]^T$ et $\delta\mathbf{t}_j \equiv \left[\delta t_{j,X}\, \delta t_{j,Y}\, \delta t_{j,Z}\right]^T$.

Du fait du jeu dans les articulations, le repère $\mathcal{F}_{n+1}$ associé à l'organe terminal est transformé en $\mathcal{F}'_{n+1}$. D'après [MLS94], le déplacement amenant le repère $\mathcal{F}_j$ en $\mathcal{F}'_j$ est donné par la matrice exponentielle de $\delta\mathbf{S}_j$, $e^{\delta\mathbf{S}_j}$. Il en résulte que le torseur représentant la pose de l'organe terminal décalé peut être calculé tout au long de la chaîne cinématique par :

$$\mathbf{P}' = \prod_{j=1}^{n} e^{\delta\mathbf{S}_j}\mathbf{S}_j, \qquad (3.10)$$

où le torseur $\mathbf{P}'$ change le repère $\mathcal{F}_1$ en $\mathcal{F}'_{n+1}$ avec prise en compte des erreurs.

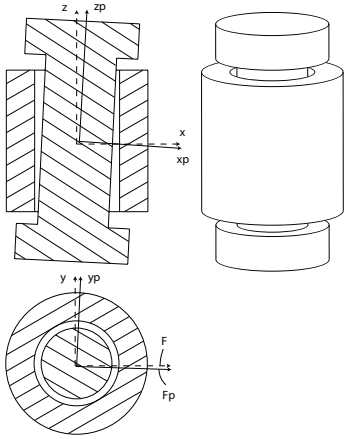


Figure 3.2 – [BSG*10] Liaison pivot affectée par du jeu.

Afin de mesurer l'erreur sur la pose de la plate-forme en mouvement, on calcule le torseur $\Delta\mathbf{P}$ qui transforme la pose nominale $\mathcal{F}_{n+1}$ en la pose décalée $\mathcal{F}'_{n+1}$ tout au long de la chaîne cinématique :

$$\Delta\mathbf{P} = \mathbf{P}^{-1}\mathbf{P}' = \prod_{j=n}^{1} \mathbf{S}_j^{-1} \prod_{j=1}^{n} \left(e^{\delta\mathbf{S}_j}\mathbf{S}_j\right). \tag{3.11}$$

D'après [CC09], il s'avère que $\Delta\mathbf{P}$ peut aussi bien être représenté de manière vectorielle comme un torseur de faible déplacement $\delta\mathbf{p}$, de la façon suivante :

$$\delta\mathbf{p} = \sum_{j=1}^{n} \left(\prod_{k=n}^{j} \mathrm{adj}(\mathbf{S}_k)^{-1} \right) \delta\mathbf{s}_j, \quad \text{avec} \quad \mathrm{adj}(\mathbf{S}_j) \equiv \begin{bmatrix} \mathbf{R}_j & \mathbf{O}_{3\times3} \\ \mathbf{T}_j\mathbf{R}_j & \mathbf{R}_j \end{bmatrix}$$

qui est la transformation adjointe du torseur $\mathbf{S}_j$ et $\mathbf{T}_j \equiv \partial(\mathbf{t}_j \times \mathbf{x})/\partial\mathbf{x}$ la matrice produit vectoriel de $\mathbf{t}_j$.

3.2.3 Erreur de pose maximum de l'organe terminal

Exprimons $\delta\mathbf{p}$ comme $[\delta\mathbf{p}_r\, \delta\mathbf{p}_t]^T$ avec $\delta\mathbf{p}_r \equiv [\delta p_{r,X}\, \delta p_{r,Y}\, \delta p_{r,Z}]^T$ et $\delta\mathbf{p}_t \equiv [\delta p_{t,X}\, \delta p_{t,Y}\, \delta p_{t,Z}]^T$ caractérisant respectivement les erreurs de rotation et de translation de l'organe terminal du manipulateur. Afin de trouver les erreurs maximum de pose de l'organe terminal pour une configuration donnée du manipulateur, nous devons résoudre deux problèmes d'optimisation sous contraintes :

$$\begin{aligned} \max \quad & \|\delta\mathbf{p}_r\|_2, \\ \text{s.c.} \quad & \delta r_{j,X}^2 + \delta r_{j,Y}^2 - \delta\beta_{j,XY}^2 \le 0, \\ & \delta r_{j,Z}^2 - \delta\beta_{j,Z}^2 \le 0, \\ & j = 1, \ldots, n \end{aligned} \tag{3.12}$$

$$\begin{aligned} \max \quad & \|\delta\mathbf{p}_t\|_2, \\ \text{s.c.} \quad & \delta r_{j,X}^2 + \delta r_{j,Y}^2 - \delta\beta_{j,XY}^2 \le 0, \\ & \delta r_{j,Z}^2 - \delta\beta_{j,Z}^2 \le 0, \\ & \delta t_{j,X}^2 + \delta t_{j,Y}^2 - \delta b_{j,XY}^2 \le 0, \\ & \delta t_{j,Z}^2 - \delta b_{j,Z}^2 \le 0, \\ & j = 1, \ldots, n \end{aligned} \tag{3.13}$$

où $\|.\|_2$ est la norme de rang 2. L'erreur maximum en rotation due au jeu dans les articulations est obtenue en résolvant le problème (3.12). De la même façon, le déplacement maximum dû au jeu est obtenu en résolvant le problème (3.13). Notons que les contraintes de ces problèmes sont définies par les équations (3.6)–(3.9). En outre, il apparaît que ces problèmes sont des problèmes quadratiques non convexes quadratiquement contraints (QCQP).

3.3 Modification du cadre de résolution classique

Les problèmes de satisfaction de contraintes sont habituellement résolus en utilisant un algorithme de branch-and-prune. L'algorithme 3.3 est un exemple basique de branch-and-prune. Ses entrées sont un ensemble de contraintes et un pavé de domaines initiaux. Il alterne pruning (l.4) et branching (l.5) pour produire un ensemble de pavés qui couvre précisément l'ensemble des solutions : grâce à la propriété satisfaite par la fonction $contract_C$ de ne perdre aucune solution potentielle grâce aux arrondis corrects de l'arithmétique des intervalles, cet algorithme maintient la propriété $\mathbf{x} \in B \wedge (\forall c \in C, c(\mathbf{x})) \Rightarrow \mathbf{x} \in \mathcal{L}$ c'est-à-dire qu'il est impossible de passer à côté d'une solution. Le critère d'arrêt est généralement la

ALG. 3.3 : BranchAndPrune(*ensemble de contraintes* $C = \{c_1, \ldots, c_m\}$, *pavé* B_0)

```
1  L ← {B_0}
2  tant que L ≠ ∅ et non critère_arrêt faire
3      (B, L) ← extract(L)
4      B ← contract_C(B)
5      si splitable(B) alors
6          {B', B''} ← split(B)
7          L ← L ∪ {B', B''}
8      fin
9  fin
10 retourner L
```

taille des pavés dans $\mathcal{L}$, l'algorithme s'arrêtant quand ils ont atteint une largeur inférieure à une précision donnée. *RealPaver* [GB06] implémente un algorithme de ce type et nous nous en sommes servi de point de départ.

Un algorithme de branch-and-prune peut être transformé en branch-and-bound capable de traiter des problèmes de minimisation —les problèmes de maximisation étant traités d'une manière similaire. L'algorithme 3.4 donne un exemple d'un tel algorithme de branch-and-bound. La fonction de coût est une entrée supplémentaire, grâce à laquelle peut être maintenue une borne supérieure du minimum global dans la variable m, initialisée à $+\infty$ en début de la recherche (l.2). Cette borne supérieure est utilisée périodiquement pour rejeter les parties de l'espace de recherche dont le coût lui est supérieur, et ce simplement en ajoutant la contrainte $f(\mathbf{x}) \leq m$ à l'ensemble des contraintes du problème (l.5). Finalement, la borne supérieure est mise à jour à la ligne 6 (cf Algorithme 3.5), en cherchant des points faisables à l'inté-

ALG. 3.4 : BranchAndBound(*ensemble de contraintes* $C = \{c_1, \ldots, c_m\}$, *pavé* B_0, *fonction* f)

```
1  L ← {B_0}
2  m ← +∞
3  tant que L ≠ ∅ et non critère_arrêt faire
4      (B, L) ← extract(L)
5      B ← contract_{C∪{f(X)≤m}}(B)
6      (B, m) ← update(C, B, f, m)
7      si splitable(B) alors
8          {B', B''} ← split(B)
9          L ← L ∪ {B', B''}
10     fin
11 fin
12 retourner (m, L)
```

ALG. 3.5 : Update(*ensemble de contraintes C*, *pavé B*, *fonction f*, *réel m*)

```
1  pour i ∈ {1, ..., N} faire
2  |  b ← random(B)
3  |  si solution(b, C) alors
4  |  |  n ← f(b)
5  |  |  si n < m alors
6  |  |  |  m ← n
7  |  |  fin
8  |  fin
9  fin
10 retourner(B, m)
```

rieur du pavé courant puis en évaluant la fonction de coût sur de tels points une fois trouvés. Dans notre implémentation actuelle, des points sont générés aléatoirement dans le pavé courant et les contraintes sont rigoureusement vérifiées en utilisant l'arithmétique des intervalles avant de mettre à jour la borne supérieure. Remarquons que si des algorithmes de branch-and-bound plus élaborés font généralement appel à de la recherche locale pour trouver de bons points faisables, les expériences que nous présentons à la section suivante, et que nous avons réalisées en utilisant cette simple méthode de génération aléatoire de points, montrent déjà de bonnes performances. L'algorithme de branch-and-bound maintient un encadrement du minimum global qui converge vers celui-ci à condition que des points faisables soient trouvés pendant la recherche, ce qui est garanti dans le cas de contraintes d'inégalité qui ne sont pas singulières.

Une implémentation efficace de l'algorithme de branch-and-bound requiert que la liste des pavés $\mathcal{L}$ soit gérée avec attention : il faut la maintenir triée par rapport à la borne inférieure de l'objectif évalué sur chaque pavé. Ainsi, à chaque fois qu'un pavé B est inséré dans $\mathcal{L}$, l'évaluation aux intervalles $f(B)$ est calculée et la borne inférieure de cette évaluation est utilisée pour maintenir la liste triée. Ensuite, à la ligne 4 on extrait le premier pavé de $\mathcal{L}$ de telle sorte que les régions les plus prometteuses de l'espace de recherche soient explorées les premières, ayant pour effet des améliorations drastiques. Un autre avantage de maintenir $\mathcal{L}$ triée est que le premier pavé de la liste contient la valeur la plus basse de borne inférieure de l'objectif parmi tous les pavés. En conséquence, on peut utiliser cette valeur et mesurer sa distance à la borne supérieure courante m pour arrêter l'algorithme quand l'encadrement du minimum global a atteint la précision souhaitée.

L'algorithme de branch-and-bound que nous venons de décrire a été implementé dans *RealPaver*. C'est cette implémentation que nous avons utilisée pour réaliser les expériences que nous présentons dans la section suivante.

3.4 Expériences

Nous avons effectué un ensemble d'expériences pour analyser les performances de notre approche pour résoudre le problème d'erreur de pose. Nous l'avons comparée avec *GAMS/BARON* [GAM] et *ECL^{i}PS^e* [WNS97]. *GAMS/BARON* est un système de programmation mathématique largement utilisé

de résolution de problèmes d'optimisation[1]. Quant à *ECL^{i}PS^e*, c'est un des rares systèmes de programmation par contraintes capables de résoudre des problèmes continus d'optimisation.

Nous avons testé 8 modèles, 4 sur l'erreur maximum en translation (T) et 4 sur l'erreur maximum en rotation (R). Pour chaque type de modèle, nous considérons à chaque fois des manipulateurs ayant de 2 à 5 joints. La Table 3.1 présente les résultats obtenus. Les colonnes 1 et 2 montrent le nombre de joints et le type de problème. Les colonnes 3 à 5 donnent des informations pertinentes quant à l'instance considérée (nombre de variables, de contraintes, d'opérations arithmétiques dans la fonction-objectif). Les colonnes 5 à 10 contiennent les temps de résolution observés en utilisant *GAMS/BARON*, *RealPaver*, et *ECL^{i}PS^e*. Nos expériences ont été réalisées sur un Pentium D à 3 Ghz avec 2 Go de RAM. Le temps de résolution présenté ici est à chaque fois le meilleur temps obtenu sur 5 lancements.

#joints	Type	Problem Size			GAMS	RP	ECL^{i}PS^e
		#var	#ctr	#op			
2	T	12	8	28	0.08	0.004	>60
2	R	6	4	18	0.124	0.004	>60
3	T	18	12	135	0.124	0.008	t.o.
3	R	9	6	90	0.952	0.004	t.o.
4	T	24	16	374	0.144	0.152	t.o.
4	R	12	8	205	2.584	0.02	t.o.
5	T	30	20	1073	0.708	>60	t.o.
5	R	15	10	480	9.241	0.26	t.o.

Table 3.1 – Temps de résolution (secondes)

Les résultats montrent que notre approche est plus rapide dans presque tous les cas. Pour les instances de taille plus réduites comme 2R, 2T, 3R et 3T, *RealPaver* offre des performances remarquables et peut être jusqu'à 100 fois plus rapide que *GAMS/BARON*. Un telle rapidité de convergence peut être expliquée en partie par l'efficacité des techniques de filtrage, basées ici sur la consistance d'enveloppe et bien optimisées ; mais surtout, par le fait qu'il n'y a pas besoin de calculs vraiment très précis pour ce problème particulier. En fait, il n'est pas utile d'atteindre une précision de l'ordre de 10^{-8}, du fait principalement de la limitation physique des outils de conception qui ne sont pas capable d'atteindre une précision aussi grande.

Pour des instances plus grandes comme 4R et 5R, *RealPaver* reste nettement plus rapide. Cependant, à mesure que la taille du problème augmente, et en particulier dès que le nombre de variables excède 20, la convergence cesse d'être instantanée. Ceci est un phénomène commun qu'on peut expliquer à la fois par le caractère exponentiel par rapport au nombre de variables de la complexité de ce type d'algorithme de résolution, et aussi au fait que la fonction-objectif elle-même croît exponentiellement avec le nombre de variables sur lesquelles elle porte. Néanmoins, nous estimons que les temps de résolution restent raisonnables au vu de la complexité et de la taille des problèmes ainsi que des techniques utilisées[2].

Cependant, les temps de résolution ne sont pas le seul point à examiner dans cette situation. Au contraire, il est également important de discuter la fiabilité des solutions calculées par *BARON*. En effet, il a déjà été mentionné dans [LMR07] que celles calculées par *BARON* n'étaient pas fiables en général.

[1]Dans notre version du logiciel, la résolution des problèmes linéaires (resp. non linéaires) est effectuée par *CPLEX* (resp. *MINOS*).

[2]Des expériences ultérieures ont en outre montré que notre approche permettait en l'état de résoudre l'instance 5T en un temps de l'ordre de la seconde, à condition que les points tirés au sort par l'algorithme soit tels qu'ils permettent de mettre à jour l'objectif de façon significative

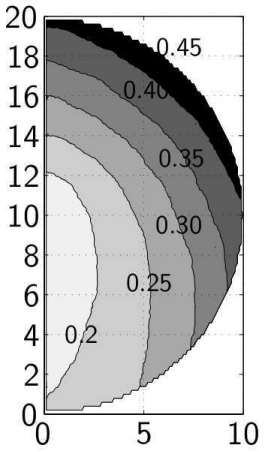


Figure 3.3 – Isocontours de l'erreur maximum de position dans l'espace de travail.

Nous avons pu vérifier ici, en utilisant *RealPaver*, soit que l'encadrement optimal calculé par *BARON* n'était pas faisable, soit qu'à l'inverse cet encadrement faisable n'était pas optimal.

Pour finir, remarquons le fait que bien que *RealPaver* et ECL^iPS^e utilisent des techniques de programmation par contraintes, le second est complètement dépassé même pour les plus petites de nos instances. Ce n'est pas surprenant cependant, étant donné que le premier a été conçu pour résoudre des problèmes continus non linéaires alors qu'ECL^iPS^e est un outil plus générique et extensible.

3.5 Conclusion

Dans ce premier chapitre, nous avons montré comment utiliser la programmation par contraintes pour résoudre le problème du calcul de l'erreur de pose maximum d'un système robotique mécanique due au jeu dans les articulations. Nous avons commencé par modéliser le problème comme un CSP continu doté d'une fonction-objectif, avant de montrer comment transformer le schéma *branch-and-prune* de résolution qu'emploie classiquement la programmation par contraintes, en un algorithme *branch-and-bound* dédié à la résolution rigoureuse de problèmes d'optimisation sous contraintes. Enfin, nous avons présenté des résultats expérimentaux, obtenus à partir de plusieurs instances du problème initial, et montré que notre approche était très compétitive par rapport à des outils à la pointe des techniques d'aujourd'hui.

Il est important de noter que cette approche n'est pas spécifique à la robotique mais est applicable à n'importe quel problème où sont requis des calculs rigoureux, gage de fiabilité des résultats numériques obtenus. De plus, des travaux futurs devraient permettre de mettre en oeuvre d'autres critères de filtrage utilisant la fonction-objectif et d'ainsi accélérer encore la convergence de l'algorithme d'optimisation, permettant de traiter efficacement des problèmes de plus grande taille. Par ailleurs, nous avons déjà commencé à explorer des pistes pour l'amélioration des performances de notre optimiseur global sous contraintes d'inégalités, et notamment l'utilisation de la contrainte additionnelle $f'(B) = 0$, qui permet sur des exemples triviaux d'effectuer des réductions de domaine drastiques, mais qui pour le problème traité ici ne s'est pas montrée d'une utilité particulière. D'autres expériences devraient nous permettre de mieux comprendre cet état de fait.

Notons également qu'au niveau de l'exploitation des résultats du point de vue de la robotique et non de l'informatique cette fois, l'enjeu sera pour nous de faire l'application de la méthode que nous avons

décrite ici afin de calculer l'erreur de pose maximum pour un échantillon de points de l'espace de travail du robot. Cela nous permettra de produire la figure dite des *isocontours* de l'erreur maximum en fonction de la position du robot dans l'espace de travail, d'une nature similaire à celle présentée en Figure 3.3, et qui permettra de mieux connaître l'impact du jeu dans les articulations sur l'erreur de pose du robot. Il pourra être intéressant au passage de comparer les performances de nos algorithmes avec celles des algorithmes génétiques habituellement utilisés à cet effet. Enfin, d'autres travaux sont déjà en cours pour également prendre en compte les imperfections d'usinage des pièces du robot.

IV

COLLABORATION DE DEUX SOLVEURS DISCRET ET CONTINU

Il ne peut pas se maintenir debout,
Celui qui se dresse sur la pointe des pieds.
Il ne peut pas marcher,
Celui qui tient écartées ses jambes.

— LAO-TSEU, De la Voie et de sa vertu, XXIV.

Dans ce chapitre, nous mettons en oeuvre une collaboration de deux solveurs de contraintes, l'un dédié aux contraintes discrètes, l'autre dédié aux contraintes continues. Notre idée directrice est de partir des outils existants pour voir ce qui manque vraiment à chacun pour traiter efficacement les problèmes mixtes, que ce soit sur le plan de la modélisation ou bien des techniques de résolution, et d'interfacer les outils entre eux pour que l'un comble les lacunes de l'autre.

Nous commençons par une présentation détaillée des capacités de modélisation des principaux outils existants, dans le but de choisir lesquels faire collaborer. Ensuite, nous décrivons la manière dont nous avons pu mettre en oeuvre la collaboration entre ces outils. Finalement, nous présentons les résultats expérimentaux que nous avons obtenus sur un ensemble de problèmes de test.

4.1 Présentation des outils existants

Cette section fait un tour d'horizon des principaux outils de résolution non commerciaux disponibles, les outils commerciaux étant en effet d'un intérêt moindre pour nous ici puisque nous voulons avoir accès au code source des logiciels que nous allons utiliser. Nous avons choisi de classer les outils en trois catégories selon la forme dans laquelle s'expriment les modèles :

- des outils boîte-noire qui prennent en entrée un fichier contenant le modèle écrit dans un langage dédié uniquement à la modélisation ;
- des interpréteurs CLP, qui travaillent avec des modèles définis sous la forme de programmes écrits en langage déclaratif de type *Prolog* étendu ;
- des bibliothèques d'objets et de fonctions dédiés aux CSP écrits dans un langage à objets type C++ ou Java, qui permettent d'exprimer les modèles sous la forme de programmes compilables dans ce langage.

Nous ne cherchons pas à être exhaustifs face au grand nombre d'outils non commerciaux disponibles et ne référençons plutôt que les outils les plus connus, dans chacun des différents panels correspondant à chaque type d'outil existant. Par contre, nous nous efforcerons de l'être concernant la mise-en-avant des propriétés de chacun, au regard de notre problématique de résoudre des problèmes mixtes, pour proposer un aperçu le plus complet possible des différentes possibilités offertes par chacun en termes de modélisation. On trouvera dans la Table 4.1 un condensé de l'ensemble de nos observations à ce sujet.

Méthode utilisée

Pour mieux cerner les capacités qu'offre chacun des outils, nous proposons d'étudier la possibilité de modéliser trois CSP de trois types différents : discret, continu, mixte. Le premier est un CSP purement discret bien connu dans la communauté de la programmation par contraintes, il s'agit de résoudre l'équation *SEND+MORE=MONEY* où chaque lettre correspond à un chiffre distinct :

Exemple 4.1. *Le problème* SEND+MORE=MONEY *est modélisé à l'aide du CSP discret suivant :*

$$\left(\begin{array}{lcl} \mathcal{X} & = & \{S, E, N, D, M, O, R, Y\} \\ \mathcal{D} & = & \left\{D_S = D_E = D_N = D_D = D_M = D_O = D_R = D_Y = \{0, \ldots, 9\}\right\} \\ C & = & \left\{\begin{array}{l} S \neq 0, \\ M \neq 0, \\ 1000S + 100E + 10N + D + 1000M + 100O + 10R + E \\ = 10000M + 1000O + 100N + 10E + Y \end{array}\right\} \end{array}\right.$$

Le deuxième modèle que nous avons cherché à résoudre est le CSP continu qui correspond à la recherche de la configuration 3D d'une molécule de cyclohexane, décrite par un système de trois équations non linéaires :

Exemple 4.2. *Le problème* cyclohexane *est modélisé à l'aide du CSP continu suivant :*

$$\left(\begin{array}{lcl} \mathcal{X} & = & \{x, y, z\} \\ \mathcal{D} & = & \left\{D_x = D_y = D_z = [-1 \times 10^8, 1 \times 10^8]\right\} \\ C & = & \left\{\begin{array}{rcl} y^2(1+z^2) + z(z-24y) & = & -13, \\ x^2(1+y^2) + y(y-24x) & = & -13, \\ z^2(1+x^2) + x(x-24z) & = & -13 \end{array}\right\} \end{array}\right.$$

Quant au troisième, il s'agira d'un CSP mixte, dont l'objet est le déplacement dans son espace de travail d'un bras de robot en fonction de ses caractéristiques :

Exemple 4.3. *Le problème* robot *est modélisé à l'aide du CSP mixte suivant :*

$$\left(\begin{array}{lcl} \mathcal{X} & = & \{i, j, a, b, \alpha, \beta, x, y, d\} \\ \mathcal{D} & = & \left\{D_i = D_j = \{2, \ldots, 5\}, D_a = D_b = [2, 8], D_x = [0, 10], D_y = [0, 6], D_\alpha = D_\beta = [0, \pi],\right. \\ & & \left. D_d = [1.99, 2.01]\right\} \\ C & = & \left\{\begin{array}{c} a \cdot sin(\alpha) = b \cdot sin(\beta - \alpha) + y, \\ a \cdot cos(\alpha) = x - (b \cdot cos(\beta - \alpha)), \\ a \cdot cos(\alpha) \leq 10, \\ a \cdot sin(\alpha) \leq 6, \\ (x-8)^2 + (y-4)^2 \leq 4, \\ a = (i-1)d, \\ b = (j-1)d \end{array}\right\} \end{array}\right.$$

4.1.1 Modélisation en langage dédié

Nous commençons cette présentation détaillée des outils existants par des logiciels qui résolvent des problèmes exprimés dans un langage dédié à la modélisation, *Minion* et *RealPaver*.

Minion

Minion est un solveur de contraintes de type boîte-noire dont le langage de modélisation utilisé par défaut est basé sur la notion de représentation matricielle d'un CSP pour pouvoir définir très simplement des contraintes sur les lignes, colonnes ou plans [GJM06].

Plus récemment, il a été interfacé avec le langage haut niveau de modélisation de problèmes combinatoires *Essence* [FHJ*05] —un sous-ensemble de celui-ci nommé *Essence'* pour être exact— au moyen du traducteur de modèles *Taylor* [GMR07]. C'est sous cette dernière forme que nous donnons ici le modèle du problème discret $SEND + MORE = MONEY$ (cf Prog. 4.1).

Comme on peut s'en rendre compte à la lecture du modèle, le langage de modélisation *Essence'* utilisé pour exprimer des modèles à résoudre par *Minion* est intuitif dans son utilisation et n'est pas difficile à comprendre. On s'aperçoit ainsi que *Minion* permet la déclaration de contraintes globales et de contraintes arithmétiques sur des variables entières.

Cependant, le solveur ne permet pas la déclaration de variables de type réel. Dans l'optique de résoudre des problèmes mixtes, il faut donc y incorporer la notion de domaine de type intervalle à bornes flottantes, ainsi que tout ce qui permet une gestion efficace de ce type de domaine : arithmétique des

intervalles, possibilité d'utiliser des expressions arithmétiques sur des variables réelles pour filtrer les domaines, bissection de domaines et heuristique de branchement associée.

PROG. 4.1 : Modèle *Minion* pour le problème discret $SEND + MORE = MONEY$.

```
1  language ESSENCE' 1.b.a
2    letting DOMAIN be domain int(0..9)
3    find S,E,N,D,M,O,R,Y  : DOMAIN
4    such that
5      S  != 0,
6      M  != 0,
7
8        1000*S+100*E+10*N+D
9      + 1000*M+100*O+10*R+E
10     = 10000*M+1000*O+100*N+10*E+Y,
11
12     alldifferent([S,E,N,D,M,O,R,Y])
```

- **l. 1** : entête du modèle avec la version du langage utilisé
- **l. 2** : déclaration d'un domaine entier $\{0, \ldots, 9\}$
- **l. 3** : déclaration des variables associées aux digits, à valeur dans le domaine défini ci-dessus
- **l. 5-6** : ajout des contraintes empêchant S et M de valoir 0
- **l. 8-10** : ajout de la contrainte arithmétique spécifiant la relation de somme entre les digits
- **l. 12** : ajout d'une contrainte globale `alldifferent` sur les variables du problème

RealPaver

RealPaver [GB06] est lui aussi un outil de résolution qui traite des modèles exprimés dans un langage de haut niveau, dédié à la modélisation de problèmes de satisfaction de contraintes.

Comme un peut s'en apercevoir avec le Programme 4.2, et comme le signifie en outre le nom de *RealPaver*, c'est un solveur de contraintes capable de paver des domaines réels étant données des équations/inéquations restreignant les valeurs admissibles pour les variables : utilisant la notion d'intervalle à bornes flottantes, il lui est possible d'effectuer des calculs corrects sur les réels.

Quant au Programme 4.3, il représente la formulation du problème discret $SEND + MORE = MONEY$. Comme on peut le voir dans cet exemple, *RealPaver* ne dispose pas de contraintes globales ni d'un opérateur de différence. S'il est tout de même possible d'exprimer ces contraintes en utilisant l'opérateur de valeur absolue et en contraignant son résultat pour disposer d'une contrainte de différence, et d'utiliser une clique de telles contraintes pour modéliser un `alldifferent`[1], on s'aperçoit ici que c'est le manque de contraintes globales, discrètes, qui fait le point faible de *RealPaver* dans l'optique de résoudre efficacement des problèmes mixtes.

Finalement, le Programme 4.4 montre la modélisation du problème mixte du positionnement d'un bras de robot. Comme dans l'exemple précédent, on constate que *RealPaver* est capable de gérer l'intégralité de tout ou partie de ses variables et permet de déclarer des variables discrètes dont le domaine est un intervalle d'entiers.

Ainsi, *RealPaver* est déjà capable de traiter des problèmes continus, discrets ou mixtes. Cependant, l'absence de contraintes globales est sa principale faiblesse au vu de notre problématique, que ce soit en terme d'aisance de modélisation ou d'efficacité de résolution.

[1] Ce point sera d'ailleurs abordé plus en profondeur au chapitre suivant.

PROG. 4.2 : Modèle *RealPaver* pour le problème continu *Cyclohexane*.

```
1  problem cyclohexane
2     num
3        p := 1.0e-8;
4     var
5        x : real / p ~ [-1.0e8,1.0e8],
6        y : real / p ~ [-1.0e8,1.0e8],
7        z : real / p ~ [-1.0e8,1.0e8];
8     ctr
9        0 = 13 + y^2*(1+z^2) + z*(z - 24*y),
10       0 = 13 + z^2*(1+x^2) + x*(x - 24*z),
11       0 = 13 + x^2*(1+y^2) + y*(y - 24*x);
12    solve *;
13 end
14
```

- **l. 1** : entête du modèle
- **l. 2-7** : déclaration d'une constante p et de variables réelles pour lesquelles on cherche à atteindre la précision p
- **l. 8-11** : ajout des contraintes de configuration de la molécule
- **l. 12** : pas de variables auxiliaires, on lance donc la résolution sur toutes les variables puisque *solve* $*$ est identique à *solve* x, y, z

PROG. 4.3 : Modèle *RealPaver* pour le problème discret $SEND + MORE = MONEY$.

```
1  problem send_more_money
2    var
3      S : int ~ [0,9],E : int ~ [0,9],
4      N : int ~ [0,9],D : int ~ [0,9],
5      M : int ~ [0,9],O : int ~ [0,9],
6      R : int ~ [0,9],Y : int ~ [0,9];
7
8    ctr
9      S>=1,M>=1,
10
11     abs(S-E)>=1,abs(S-N)>=1,abs(S-D)>=1,
12     abs(S-M)>=1,abs(S-O)>=1,abs(S-R)>=1,
13     abs(S-Y)>=1,abs(E-N)>=1,abs(E-D)>=1,
14     abs(E-M)>=1,abs(E-O)>=1,abs(E-R)>=1,
15     abs(E-Y)>=1,abs(N-D)>=1,abs(N-M)>=1,
16     abs(N-O)>=1,abs(N-R)>=1,abs(N-Y)>=1,
17     abs(D-M)>=1,abs(D-O)>=1,abs(D-R)>=1,
18     abs(D-Y)>=1,abs(M-O)>=1,abs(M-R)>=1,
19     abs(M-Y)>=1,abs(O-R)>=1,abs(O-Y)>=1,
20     abs(R-Y)>=1,
21
22       1000*S + 100*E + 10*N + D
23   +  1000*M + 100*O + 10*R + E
24   = 10000*M + 1000*O + 100*N + 10*E + Y;
25
26   solve *;
27 end
```

- **l. 1** : entête du modèle
- **l. 2-6** : déclaration de variables entières
- **l. 9** : ajout des contraintes empêchant S et M de valoir 0
- **l. 11-20** : ajout d'une clique de contraintes binaires pour que toute paire de variables prenne un couple de valeurs différentes
- **l. 22-24** : ajout de la contrainte arithmétique spécifiant la relation de somme entre les digits
- **l. 26** : pas de variables auxiliaires, on lance la résolution sur toutes les variables

PROG. 4.4 : Modèle *RealPaver* pour le problème mixte du bras de robot.

```
1  problem arm
2
3    num
4      Tx :=10.0, Ty :=6.0,
5      x0 :=8, y0 :=4;
6
7    var
8      d :real~[1.99,2.01],
9      a :real~[2.0,8.0], b :real~[2.0,8.0],
10     alpha :real~[0,Pi], beta :real~[0,Pi],
11     x :real~[0,Tx], y :real~[0,Ty],
12     i :int~[2,5], j :int~[2,5];
13
14   ctr
15     a*sin(alpha)=b*sin(beta-alpha)+y,
16     a*cos(alpha)=x-(b*cos(beta-alpha)),
17     a*cos(alpha)<=Tx,
18     a*sin(alpha)<=Ty,
19     (x-x0)^2+(y-y0)^2<=4,
20     a=(i-1)*d,
21     b=(j-1)*d;
22
23   solve *;
24
25 end
```

- **l. 1** : entête du modèle ;
- **l. 3-5** : déclarations de constantes ;
- **l. 8-12** : déclarations des variables du modèle, certaines réelles et d'autres entières, en utilisant les constantes définies en tête de modèle ou prédéfinies comme `Pi` ;
- **l. 15-21** : ajout des contraintes du modèle ;
- **l. 23** : pas de variables auxiliaires, on lance la résolution sur toutes les variables.

4.1.2 Modélisation en langage déclaratif

Nous continuons notre tour d'horizon des logiciels de résolution de problèmes avec une section dans laquelle nous présentons des outils qui traitent des modèles exprimés sous la forme de programmes écrits dans un langage déclaratif : *ECL^{i}PS^e* et *CHR*.

ECL^{i}PS^e

ECL^{i}PS^e est un langage de programmation logique à contraintes dont l'origine remonte au début des années 1990 [AW07]. Aussi, exprimer un modèle pour *ECL^{i}PS^e* consiste à écrire un prédicat en langage déclaratif, semblable à n'importe quel autre prédicat ou presque.

Le Programme 4.5 correspond au modèle discret $SEND + MORE = MONEY$. Le langage utilisé ici est très proche d'un langage de modélisation dédié même si cela reste la syntaxe d'un langage de la famille *Prolog* qui a été élargie, pour accueillir l'expression de contraintes arithmétiques notamment. *ECL^{i}PS^e* dispose en outre d'un catalogue de contraintes globales comme cet `alldifferent` qu'on utilise ici pour contraindre les différents digits à prendre des valeurs distinctes.

Mais *ECL^{i}PS^e* offre également la possibilité de déclarer des variables réelles et des contraintes continues, comme le montre le Programme 4.6 : le symbole *$* permet de faire la distinction entre domaine discret et domaine continu, entre contrainte discrète et contrainte continue.

Ainsi, il est tout à fait possible d'exprimer et de résoudre grâce à *ECL^{i}PS^e* des problèmes mixtes. Le Programme 4.7 montre d'ailleurs l'exemple du problème mixte du bras de robot. Même si dans ce cas précis n'interviennent pas simultanément contraintes arithmétiques continues et contraintes globales

PROG. 4.5 : Modèle ECL^iPS^e pour le problème discret $SEND + MORE = MONEY$.

```
1  sendmore(Digits) :-
2      Digits = [S,E,N,D,M,O,R,Y],
3      Digits :: [0..9],
4
5      S #\= 0,
6      M #\= 0,
7
8              1000*S+100*E+10*N+D
9          + 1000*M+100*O+10*R+E
10     #= 10000*M+1000*O+100*N+10*E+Y,
11
12     alldifferent(Digits),
13
14     labeling(Digits).
```

- **l. 1** : entête du modèle
- **l. 2-3** : déclarations des variables, entières, et de leurs domaines initiaux
- **l. 5-6** : ajout des contraintes empêchant S et M de valoir 0
- **l. 8-10** : ajout de la contrainte arithmétique spécifiant la relation de somme entre les digits
- **l. 12** : ajout d'une contrainte globale `alldifferent` sur les variables du problème
- **l. 14** : lancement de l'exploration de l'espace de recherche en énumérant les domaines des variables

discrètes, il est tout à fait possible de modéliser de tels problèmes et la force d'ECL^iPS^e vis-à-vis de notre problématique est bel-et-bien l'étendue de ses capacités de modélisation.

Cependant, si le moteur de résolution d'ECL^iPS^e est parfaitement opérationnel en ce qui concerne le traitement des problèmes discrets, il n'en est pas de même quant à la résolution de problèmes continus ou mixtes. Comme nos expériences le mettront en lumière plus loin dans ce chapitre, l'efficacité d'ECL^iPS^e est très rapidement mise à mal par la présence de variables continues dans le modèle[2].

PROG. 4.6 : Modèle ECL^iPS^e pour le problème continu *Cyclohexane*.

```
1  cyclohexane(Vars) :-
2
3      Vars = [X,Y,Z],
4      Vars $:: [-1e8..1e8],
5
6      13+Y^2*(1+Z^2)+Z*(Z-24*Y)$=0,
7      13+Z^2*(1+X^2)+X*(X-24*Z)$=0,
8      13+X^2*(1+Y^2)+Y*(Y-24*X)$=0,
9
10     locate([X,Y,Z],[X,Y,Z],1e-6,lin).
```

- **l. 1** : entête du modèle
- **l. 2-3** : déclarations des variables, réelles (opérateur $ ::) et de leurs domaines initiaux
- **l. 5-7** : ajout des contraintes du modèle
- **l. 10** : lancement de l'exploration de l'espace de recherche en bissectant les domaines jusqu'à une largeur finale de 1×10^{-6}

[2]Il s'agit en effet d'un programme que son ancienneté a rendu très complet, mais qui, bien que fournissant de très bons résultats en terme de correction de la solution, est hélas loin d'être optimal quant à la rapidité d'obtention de cette solution, en particulier en ce qui concerne les problèmes continus ou mixtes.

PROG. 4.7 : Modèle ECL^iPS^e pour le problème mixte du bras de robot.

```
1  arm(Vars) :-
2
3      Tx is 10.0, Ty is 6.0,
4      X0 is 8, Y0 is 4,
5      D is 1.99__2.01,
6
7      Vars = [A,B,Alpha,Beta,X,Y,I,J],
8      A $::[2.0..8.0], B $::[2.0..8.0],
9      Alpha $::[0.. pi], Beta $::[0.. pi],
10     X $::[0..Tx], Y $::[0..Ty],
11     I ::[2..5], J ::[2..5] ;
12
13     A * sin(Alpha)
14         $= B * sin(Beta - Alpha) + Y,
15     A * cos(Alpha)
16         $= X - (B * cos(Beta - Alpha)),
17     A * cos(Alpha) $=< Tx,
18     A * sin(Alpha) $=< Ty,
19     (X - X0)^2 + (Y - Y0)^2 $=< 4,
20     A $= (I - 1) * D,
21     B $= (J - 1) * D,
22
23     locate([A,B,Alpha,Beta,X,Y,I,J],
24         [A,B,Alpha,Beta,X,Y,I,J],1e-6,lin).
```

- **l. 1** : entête du modèle
- **l. 3-5** : déclarations de constantes
- **l. 7-11** : déclarations des variables, réelles ou entières, et de leurs domaines initiaux
- **l. 13-21** : ajout des contraintes du modèle
- **l. 23-24** : lancement de l'exploration de l'espace de recherche en bissectant les domaines jusqu'à une largeur finale de 1×10^{-6}

CHR

L'autre outil que nous présentons dans cette catégorie des outils de résolution traitant des modèles exprimés en langage déclaratif est *Constraint Handling Rules (CHR)*. Il existe des implémentations de ce langage dans différents langages hôtes comme *Java*, *Prolog* et même *Haskell*[3]. Difficile à classer, *CHR* est un langage à écrire des solveurs plutôt qu'un solveur à proprement parler [Rai08].

Le Programme 4.8 correspondant à un modèle *CHR* pour résoudre le problème discret $SEND + MORE = MONEY$. La syntaxe déclarative rappelle fortement *Prolog*. En fait, le langage *CHR* ne sert pas tant à exprimer le modèle qu'à écrire le solveur pour le résoudre, un solveur par type de modèle pour ainsi dire.

Si *CHR* est capable en théorie de traiter les contraintes globales et les domaines autres que discret — cette flexibilité est même une des raisons d'être de *CHR*— il faut d'abord trouver un solveur adéquat et à défaut l'écrire. On trouve par exemple dans [Rai08] une méthode de génération automatique de solveurs *CHR* pour les contraintes globales. Ainsi, au regard de notre problématique de la résolution efficace des problèmes mixtes, l'essentiel du travail à réaliser ici consisterait à faire un état de l'art en profondeur du millier d'articles traitant de CHR référencés à ce jour, et à fusionner tout ce qui a été déjà fait en *CHR* avant de compléter les manques grâce à d'autres outils le cas échéant[4].

[3] Pour les besoins de ce comparatif, nous nous sommes basés sur l'implémentation fournie avec *YAP Prolog* [SCDRA].

[4] Nos travaux sont antérieurs à la sortie du livre [Frü09] qui constitue à n'en pas douter un parfait point de départ dans cette direction.

PROG. 4.8 : Modèle *CHR* pour le problème discret $SEND + MORE = MONEY$.

```
1  sum(1000,S,100,E,1,H1),
2  sum(1,H1,10,N,1,H2),
3  sum(1,H2,1,D,1,H3),
4  sum(1,H3,1000,M,1,H4),
5  sum(1,H4,100,O,1,H5),
6  sum(1,H5,10,R,1,H6),
7  sum(1,H6,1,E,1,Z),
8  sum(10000,M,1000,O,1,G1),
9  sum(1,G1,100,N,1,G2),
10 sum(1,G2,10,E,1,G3),
11 sum(1,G3,1,Y,1,Z).
12
13
14 neq(S,0),neq(M,0),
15
16 neq(S,E), neq(S,N), neq(S,D), neq(S,M),
17 neq(S,O), neq(S,R), neq(S,Y), neq(E,N),
18 neq(E,D), neq(E,M), neq(E,O), neq(E,R),
19 neq(E,Y), neq(N,D), neq(N,M), neq(N,O),
20 neq(N,R), neq(N,Y), neq(D,M), neq(D,O),
21 neq(D,R), neq(D,Y), neq(M,O), neq(M,R),
22 neq(M,Y), neq(O,R), neq(O,Y), neq(R,Y),
23
24 fd(S,0,9),fd(E,0,9),fd(N,0,9),fd(D,0,9),
25 fd(M,0,9),fd(O,0,9),fd(R,0,9),fd(Y,0,9).
```

- **l. 1-11** : ajout de la contrainte arithmétique spécifiant la relation de somme entre les digits
- **l. 14** : ajout des contraintes empêchant S et M de valoir 0
- **l. 16-22** : ajout d'une clique de contraintes binaires pour que toute paire de variables prenne un couple de valeurs différentes
- **l. 24-25** : lancement de l'exploration de l'espace de recherche en donnant les domaines initiaux des variables

4.1.3 Modélisation en langage orienté-objet

Il existe enfin d'autres outils de résolution de CSP qui eux sont fournis sous la forme de bibliothèques d'objets et de fonctions, écrits dans un langage générique de programmation orientée objet, et qui requièrent de la part de l'utilisateur, pour résoudre un problème de contraintes donné, d'écrire un programme et de le compiler comme un programme traditionnel. Les outils de cette catégorie que nous présenterons ici sont *Gecode* et *Choco*.

Gecode

Commençons avec *Gecode*, une bibliothèque de résolution de problèmes sous contraintes écrite en C++. Tout comme *Minion*, son efficacité à résoudre des problèmes, même imposants en taille, n'est plus à démontrer [Tac09].

Le Programme 4.9 donne un bon aperçu de ce à quoi ressemble un modèle au format tel que le traite *Gecode*. Il s'agit ainsi d'une classe *C++* et d'une fonction *main*. Dans la première, les variables et les contraintes sont déclarées et instanciées, et la résolution paramétrée. Dans la deuxième, on lance la méthode de résolution associée à la classe qu'on vient de définir.

Pour autant efficace qu'il soit toutefois, *Gecode* ne propose pas le type de variable réel. Il n'est donc pas possible de déclarer de domaines réels ni de modéliser des contraintes continues ou mixtes. Dans l'optique de résoudre des problèmes mixtes, il faudrait donc y inclure la notion d'intervalle à bornes flottantes, de même que tout ce qui permet de gérer efficament ce type de domaine, comme l'arithmétique des intervalles, la possibilité de filtrer les domaines en utilisant des expressions arithmétiques pour filtrer les domaines, le mécanisme de bissection de domaines et l'heuristique de branchement associée.

PROG. 4.9 : Modèle *Gecode* pour le problème discret $SEND + MORE = MONEY$.

```
 1 class Money : public Example
 2 {
 3 protected :
 4   static const int nl = 8;
 5   IntVarArray le;
 6
 7 public :
 8   Money(const Options& opt) : le(this,nl,0,9)
 9   {
10     IntVar
11       s(le[0]), e(le[1]), n(le[2]), d(le[3]),
12       m(le[4]), o(le[5]), r(le[6]), y(le[7]);
13
14     rel(this, s, IRT_NQ, 0);
15     rel(this, m, IRT_NQ, 0);
16
17     post(this,       1000*s+100*e+10*n+d
18       +              1000*m+100*o+10*r+e
19       == 10000*m+1000*o+100*n+10*e+y,
20         opt.icl);
21
22     distinct(this,le,opt.icl);
23
24     branch(this,le,BVAR_SIZE_MIN,BVAL_MIN);
25
26   }
27 };
28
29 int
30 main(int argc, char** argv)
31 {
32   Options opt("SEND+MORE=MONEY");
33   opt.solutions = 0;
34   opt.iterations = 20000;
35   opt.parse(argc,argv);
36   Example::run<Money,DFS>(opt);
37   return 0;
38 }
39
```

- **l. 1-8** : déclaration d'une classe C++ dédiée au problème, et instanciation des paramètres hérités comme le nombre de variables *nl*
- **l. 10-12** : déclaration des variables
- **l. 14-15** : ajout des contraintes empêchant S et M de valoir 0
- **l. 17-20** : ajout de la contrainte arithmétique spécifiant la relation de somme entre les digits
- **l. 22** : ajout d'une contrainte de type `alldifferent` portant sur l'ensemble des digits
- **l. 24** : définition de la stratégie d'exploration
- **l. 29-38** : initialisation et lancement de la résolution

Choco

Choco est une bibliothèque écrite en Java qui permet la modélisation et la résolution de problèmes sous contraintes. L'implémentation a été faite avec deux objectifs : d'une part offrir une architecture logicielle modulaire qui accepte aisément des extensions, et d'autre part, mettre en oeuvre une gestion optimisée des événements de propagation [Lab00][5].

Le Programme 4.10 correspond à la traduction du modèle $SEND+MORE = MONEY$. Remarquons que la déclaration d'expressions arithmétiques pour les contraintes se fait de manière préfixée, à l'aide de méthodes de la classe `Problem` : le langage Java ne permet pas de surcharger ses opérateurs et il est donc impossible d'adopter une syntaxe plus naturelle comme le permet par exemple *Gecode*, écrit lui en C++.

En outre, CHOCO permet la modélisation de problèmes continus en proposant d'instancier des variables de type réel comme on peut le voir dans le Programme 4.11. On s'aperçoit même finalement

[5] Les présents travaux ont été réalisés avant la récente refonte de la bibliothèque présentée dans [RJL*08] et se basent sur la version 1.2.06 qui lui est antérieure.

que *Choco* est déjà, dans l'état actuel, tout à fait capable de traiter des problèmes mixtes comme notre problème du bras de robot (cf Prog. 4.12).

Cependant, ses limites se trouvent principalement dans les opérations arithmétiques proposées (peu de non-linéaire, erreurs de calculs dans les fonctions trigonométriques) et les types de contraintes numériques disponibles (égalité seulement). En outre, le langage de modélisation proposé est difficile à utiliser et les modèles obtenus sont peu lisibles, comme on peut déjà s'en apercevoir à travers les quelques exemples présentés ici.

PROG. 4.10 : Modèle *Choco* pour le problème discret $SEND + MORE = MONEY$.

```
1  public class Send
2  {
3
4    public static void main(String[] args)
5    {
6
7      Problem pb = new Problem();
8
9      IntDomainVar S = pb.makeEnumIntVar("S",0,9);
10     IntDomainVar E = pb.makeEnumIntVar("E",0,9);
11     IntDomainVar N = pb.makeEnumIntVar("N",0,9);
12     IntDomainVar D = pb.makeEnumIntVar("D",0,9);
13     IntDomainVar M = pb.makeEnumIntVar("M",0,9);
14     IntDomainVar O = pb.makeEnumIntVar("O",0,9);
15     IntDomainVar R = pb.makeEnumIntVar("R",0,9);
16     IntDomainVar Y = pb.makeEnumIntVar("Y",0,9);
17
18     IntDomainVar SEND
19         = pb.makeBoundIntVar("SEND",0,1000000);
20     IntDomainVar MORE
21         = pb.makeBoundIntVar("MORE",0,1000000);
22     IntDomainVar MONEY
23         = pb.makeBoundIntVar("MONEY",0,1000000);
24
25
26     IntVar[] letters = {S, E, N, D, M, O, R, Y};
27     pb.post(pb.alldifferent(letters));
28
29     pb.post(pb.neq(S, 0)) ;
30     pb.post(pb.neq(M, 0)) ;
31
32     pb.post(pb.eq(SEND,
33       pb.scalar(new int[]{1000,100,10,1},
34         new IntDomainVar[]{S, E, N, D})));
35
36     pb.post(pb.eq(MORE,
37       pb.scalar(new int[]{1000,100,10,1},
38         new IntDomainVar[]{M, O, R, E})));
39
40     pb.post(pb.eq(MONEY,
41       pb.scalar(new int[]{10000,1000,100,10,1},
42         new IntDomainVar[]{M, O, N, E, Y})));
43
44     pb.post(pb.eq(0,pb.minus(
45                       pb.plus(SEND, MORE),
46                       MONEY)));
47
48     pb.solve(false);
49   }
50 }
```

- **l. 1-7** : déclaration d'une classe Java et de sa fonction `main` dédiée à la résolution du problème
- **l. 9-16** : déclaration des variables correspondant aux différents digits, de type entier, ayant leur domaine énuméré, de domaine initial $\{0, \ldots, 9\}$
- **l. 18-23** : déclaration des variables correspondant à chaque ligne de la somme, de type entier, ayant leur domaine borné, de domaine initial $[0, 10^6]$
- **l. 26-27** : ajout d'une contrainte `alldifferent` portant sur l'ensemble des digits
- **l. 29-30** : ajout des contraintes $S \neq 0$ et $M \neq 0$
- **l. 32-42** : ajout des contraintes qui relient chaque variable correspondant à un des trois termes de la somme aux digits qui le constitue, e.g. $SEND = S \times 1000 + E \times 100 + N \times 10 + D$
- **l. 44-46** : ajout de la contrainte $SEND + MORE = MONEY$
- **l. 38** : lancement de la résolution

PROG. 4.11 : Modèle *Choco* pour le problème continu *Cyclohexane*.

```
1  public class Cyclohexane
2  {
3
4    public static void main(String[] args)
5    {
6
7      Problem pb = new Problem();
8      pb.setPrecision(1e-8);
9
10     RealVar x = pb.makeRealVar("x",-1.0e8,1.0e8);
11     RealVar y = pb.makeRealVar("y",-1.0e8,1.0e8);
12     RealVar z = pb.makeRealVar("z",-1.0e8,1.0e8);
13
14     RealExp exp1 = pb.plus(pb.mult(pb.power(y,2),
15       pb.plus(pb.cst(1.0),pb.power(z,2))),
16       pb.mult(z,pb.minus(z,pb.mult(pb.cst(24),y))));
17
18     RealExp exp2 = pb.plus(pb.mult(pb.power(z,2),
19       pb.plus(pb.cst(1.0),pb.power(x,2))),
20       pb.mult(x,pb.minus(x,pb.mult(pb.cst(24),z))));
21
22     RealExp exp3 = pb.plus(pb.mult(pb.power(x,2),
23       pb.plus(pb.cst(1.0),pb.power(y,2))),
24       pb.mult(y,pb.minus(y,pb.mult(pb.cst(24),x))));
25
26     Equation eq1
27       = (Equation) pb.eq(exp1, pb.cst(-13));
28     eq1.addBoxedVar(y);eq1.addBoxedVar(z);
29     pb.post(eq1);
30
31     Equation eq2
32       = (Equation) pb.eq(exp2, pb.cst(-13));
33     eq2.addBoxedVar(x);eq2.addBoxedVar(z);
34     pb.post(eq2);
35
36     Equation eq3
37       = (Equation) pb.eq(exp3, pb.cst(-13));
38     eq3.addBoxedVar(x);eq3.addBoxedVar(y);
39     pb.post(eq3);
40
41     Solver solver = pb.getSolver();
42     solver.setFirstSolution(false);
43     solver.generateSearchSolver(pb);
44     solver.addGoal(new AssignInterval(
45             new CyclicRealVarSelector(pb),
46             new RealIncreasingDomain()));
47
48     solver.launch();
49   }
50 }
```

- **l. 1-7** : déclaration d'une classe Java et de sa fonction `main` dédiée à la résolution du problème
- **l. 8** : paramétrage de la précision à atteindre sur le domaine des variables réelles
- **l. 10-12** : déclaration des variables du modèle, de type réel, de domaine initial $[-1 \times 10^8, 1 \times 10^8]$
- **l. 14-24** : constructions des expressions arithmétiques qui forment le corps des contraintes, e.g. $y^2(1+z^2)+z(z-24y)$
- **l. 26-39** : ajout des contraintes du modèle en utilisant les expressions instanciées ci-dessus, en spécifiant grâce à la fonction `addBoxedVar` le fait que les variables devront être réduite par consistance de pavé
- **l.41-46** : configuration du solveur associé au modèle sur la façon dont prendre en compte les variables réelles dans la stratégie d'exploration
- **l. 47** : lancement de la résolution

PROG. 4.12 : Modèle *Choco* pour le problème mixte du bras de robot.

```
1  public class Arm
2  {
3    public static void main(String[] args)
4    {
5      Problem pb = new Problem();
6
7      RealVar a = pb.makeRealVar("a",2.0,8.0);
8      RealVar b = pb.makeRealVar("b",2.0,8.0);
9      RealVar alpha
10         = pb.makeRealVar("alpha",0.0,Math.PI);
11     RealVar beta
12         = pb.makeRealVar("beta", 0.0,Math.PI);
13     RealVar x = pb.makeRealVar("x",0.0,10.0);
14     RealVar y = pb.makeRealVar("y",0.0,8.0);
15
16     IntDomainVar i = pb.makeBoundIntVar("i",1,4);
17     IntDomainVar j = pb.makeBoundIntVar("j",1,4);
18     RealVar ir = pb.makeRealVar("i'",1.0,4.0);
19     RealVar jr = pb.makeRealVar("j'",1.0,4.0);
20     pb.post(new MixedEqXY(ir,i));
21     pb.post(new MixedEqXY(jr,j));
22
23     RealExp exp1 = pb.minus(y,pb.plus(pb.mult(a,
24       pb.sin(alpha)),pb.mult(b,pb.sin(
25       pb.minus(alpha, beta)))));
26
27     RealExp exp2 = pb.minus(x,pb.plus(pb.mult(a,
28       pb.cos(alpha)),pb.mult(b,pb.cos(
29       pb.minus(alpha,beta)))));
30
31     RealExp circle = pb.plus(pb.power(pb.minus(x,
32       pb.cst(8.0)),2),pb.power(pb.minus(y,
33       pb.cst(4.0)),2));
34
35     pb.post(pb.eq(exp1,0.0));
36     pb.post(pb.eq(exp2,0.0));
37     pb.post(pb.leq(circle, 4.0));
38
39     pb.post(pb.eq(alpha, pb.around(Math.PI / 6)));
40     pb.post(pb.leq(pb.mult(a,pb.cos(alpha)),10.0));
41     pb.post(pb.leq(pb.mult(a,pb.sin(alpha)),8.0));
42
43     pb.post(pb.eq(pb.minus(a, pb.mult(ir,
44      new RealIntervalConstant(1.99, 2.01))),0.0));
45     pb.post(pb.eq(pb.minus(b, pb.mult(jr,
46      new RealIntervalConstant(1.99, 2.01))),0.0));
47
48     Solver solver = pb.getSolver();
49     solver.setFirstSolution(true);
50     solver.generateSearchSolver(pb);
51     solver.addGoal(
52         new AssignInterval(
53         new CyclicRealVarSelector(pb),
54         new RealIncreasingDomain()));
55
56     solver.launch();
57   }
58 }
```

- **l. 1-5** : déclaration d'une classe Java et de sa fonction `main` dédiée à la résolution du problème
- **l. 7-14** : déclaration des variables continues du modèle, de type réel, avec leurs domaines initiaux respectifs
- **l. 16-17** : déclaration des variables discrètes du modèle, ayant leurs domaine bornés, de type entier, avec leurs domaines initiaux respectifs
- **l. 18-19** : déclaration de variables continues correspondant aux variables discrètes du modèle
- **l. 20-21** : ajout de contraintes reliant chaque variable discrète à sa variable continue équivalente
- **l. 23-33** : constructions des expressions arithmétiques qui forment le corps des contraintes, e.g. $(x-8)^2+(y-4)^2 \leq 4$
- **l. 35-37** : ajout des contraintes du modèle en utilisant les expressions instanciées ci-dessus
- **l. 39-46** : ajout des autres contraintes du modèle en utilisant les variables continues homologues des variables discrètes
- **l. 48-54** : configuration du solveur associé au modèle sur la façon dont prendre compte les variables réelles dans la stratégie d'exploration
- **l. 56** : lancement de la résolution

4.1.4 Synthèse

La Table 4.1 récapitule l'ensemble de ce qui a été montré des capacités de modélisation[6] de chacun des solveurs. Quant à la Figure 4.1, elle représente sous une forme plus synthétique ce qui manque à chaque outil pour résoudre efficacement les problèmes mixtes.

Au regard de nos observations, nous avons choisi en conséquence de faire collaborer *Choco* et *RealPaver*. En effet, il ne manque quasiment rien au premier pour être complètement opérationnel sur les problème mixtes et c'est bel-et-bien le deuxième, *RealPaver*, qui est capable de lui apporter le peu qui lui manque le plus efficacement[7] parmi les outils à notre disposition. Ainsi, nous allons confier à *Choco* le rôle principal puisqu'il est en effet déjà très complet, et nous baser sur *RealPaver* pour combler sa principale faiblesse : la gestion de la partie continue ou mixte du modèle.

			Minion	RealPaver	ECL^{i}PS^e	CHR	Gecode	Choco
domaine	booléen		+	+	+	+	+	+
	intervalle d'entiers		+	+	+	+	+	+
	ensemble creux d'entiers		+	+	+	+	+	+
	intervalle à bornes flottantes		-	+	+	+	-	+
	union d'intervalles		-	+	+	?	+	-
variable	booléenne		+	+	+	+	+	+
	entière		+	+	+	+	+	+
	réelle		-	+	+	+	-	+
contrainte	arithmétique	entière	+	+	+	+	+	+
		réelle	-	+	+	?	-	+
	globale		+	-	+	+	+	+
	en extension		+	+	+	+	+	+
	dynamique		+	+	+	?	-	+
divers	opérations arithmétiques	linéaires	+	+	+	+	+	+
		non linéaires	+	+	+	?	-	+
	constante	bool	+	+	+	+	+	+
		int	+	+	+	+	+	+
		float	-	+	-	?	-	+
		intervalle d'entiers	-	-	+	?	-	-
		intervalle à bornes flottantes	-	-	+	-	-	+

Table 4.1 – Récapitulatif des capacités de modélisation des solveurs comparés.

[6] Tous les concepts non triviaux qui y figurent ont déjà été présentés plus tôt dans ce chapitre ou au début du précédent.

[7] Un second choix aurait pu être d'optimiser le code d'*ECL^{i}PS^e*. Mais s'agissant d'une tâche très difficile à évaluer, nous lui avons préféré un objectif plus clair.

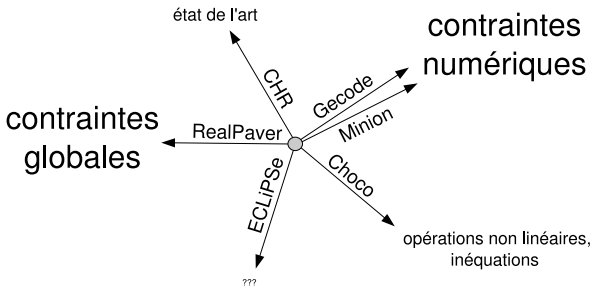


Figure 4.1 – Schéma de synthèse, la taille de la police utilisée est proportionnelle à la quantité estimée de travail à fournir pour compléter l'outil en question : *Choco*, *CHR* et ECL^iPS^e sont déjà assez bien dotés, *RealPaver*, *Minion* et *Gecode* sont quant à eux plus spécifiques à un type de problème.

4.2 Mise-en-oeuvre de la collaboration

Dans cette section, nous décrivons la façon dont nous avons réalisé la collaboration entre *Choco*, un solveur dédié aux contraintes discrètes, et *RealPaver*, un solveur dédié aux contraintes continues.

4.2.1 Solveurs utilisés

Commençons par décrire dans les grandes lignes l'architecture de chacun des deux solveurs que nous avons fait collaborer.

RealPaver

RealPaver est un outil logiciel pour la modélisation et la résolution de systèmes de contraintes non linéaires. La version 1.0 que nous avons utilisée est utilisable sous la forme d'une bibliothèque C/C++ avec une API qui lui permet de s'intégrer facilement dans d'autres applications.

L'implémentation du parseur, des expressions arithmétiques ainsi que leur évaluation en utilisant l'arithmétique des intervalles et la gestion en mémoire des boîtes sont réalisées en C. Ainsi, que ce soit par exemple le type des variables ou des noeuds des arbres représentant les opérations arithmétiques, cela n'est pas matérialisé par des classes particulières d'objets mais par des valeurs spécifiques pour un champ donné d'une structure de donnée générique. À l'inverse, les opérateurs de réduction et tout ce qui concerne la propagation mais aussi les stratégies d'exploration, ainsi que les autres structures de données haut-niveau du solveur, sont réalisés en C++ et utilisent le concept d'héritage pour se spécialiser le cas échéant. Les différentes possibilités de configuration du solveur sont ainsi exprimées sous la forme d'une hiérarchie de classes au sein de laquelle on peut choisir celle qui correspond au paramétrage désiré.

Choco

Choco est une bibliothèque Java initialement dédiée à la résolution de problèmes discrets. Puis elle a été étendue aux problèmes continus ou mixtes par l'adjonction du type de variable réel ainsi que du type de domaine intervalle à bornes flottantes et des règles d'évaluation des expressions arithmétiques se basant sur l'arithmétique des intervalles.

Contrairement à *RealPaver*, *Choco* est foncièrement et intégralement orienté objet. Ainsi, une hiérarchie de contraintes abstraites les répartit selon leur genre —globale, arithmétique, booléenne, etc.— et le type des variables sur lesquelles elles portent —entière, réelle, booléenne, ensembliste, etc. Idem pour tout ce qui concerne les stratégies d'exploration et toutes les structures de données haut-niveau du solveur qui sont représentées par une hiérarchie ad hoc. En outre, un mécanisme basé sur la notion d'événement déclenche automatiquement la méthode adéquate comme par exemple lors d'un changement de borne inférieure qui entraîne la réévaluation de toutes les contraintes utilisant la variable dont le domaine a été modifié.

4.2.2 Principe de la collaboration

Le principe de la collaboration est donné à la Figure 4.2. Ainsi, la boucle principale de l'exploration de l'espace de recherche est confiée à *Choco* qui fait appel à *RealPaver* simplement comme une nouvelle sorte de contrainte, portant sur toute la partie du modèle nécessitant d'être évaluée grâce à l'arithmétique des intervalles : la phase de modélisation du problème intègre désormais la possibilité de spécifier un nouveau type de contrainte, `RPIneqPropag`, qui porte sur un ensemble d'équations et/ou d'inéquations qu'on va transmettre à *RealPaver* sous la forme de chaînes de caractères. À partir de ces contraintes, ce dernier va instancier un propagateur de la même façon qu'il l'aurait fait habituellement à partir d'un fichier texte contenant un modèle exprimé dans son langage dédié. Puis, à chaque fois qu'a lieu l'itération

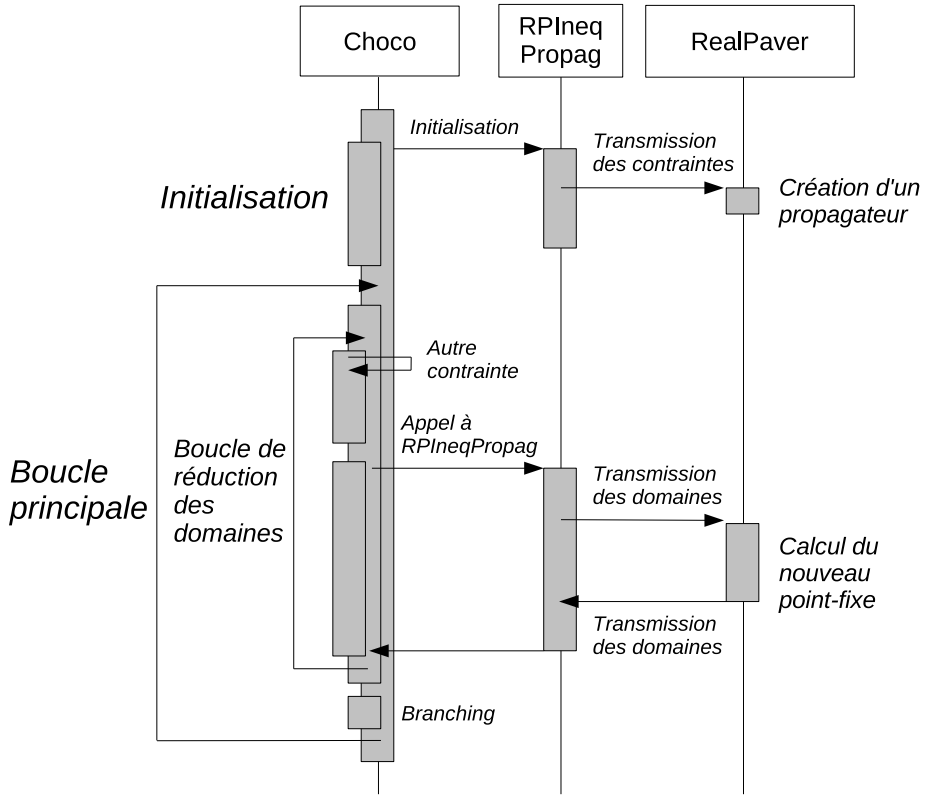


Figure 4.2 – Principe de la collaboration.

des réductions de domaines pendant l'exploration de l'espace de recherche par *Choco*, après par exemple qu'une ou des variables sur lesquelles ce propagateur porte aient vu leurs domaines respectifs réduits par l'application du filtrage d'une autre contrainte *Choco*, la méthode de calcul de point du fixe du propagateur est appelée avec en paramètre la liste des domaines —tel qu'actuellement réduits— des variables sur lesquelles il porte. Ces domaines sont recopiés dans *RealPaver* avant d'être réduits grâce aux algorithmes de filtrage de ce dernier. Enfin, quand le point-fixe a été atteint, les nouveaux domaines sont retransmis à *Choco* afin de poursuivre la résolution au point où elle en était rendue, à savoir ou bien mener à son terme la propagation des réductions de domaine, ou bien choisir une variable pour énumérer/bissecter son domaine et ainsi poursuivre l'exploration de l'espace de recherche.

4.2.3 Détails d'implémentation

Il peut être intéressant à présent de fournir quelques détails concernant l'implémentation de la collaboration proprement dite, en particulier à propos de ce qui concerne l'interfaçage de méthode Java avec des méthodes C++ via la bibliothèque *JNI*.

Java Native Interface (JNI)

Java Native Interface (JNI) est une bibliothèque qui permet la communication entre des méthodes écrites en C++ et des méthodes écrites en Java. Ainsi, on peut très bien faire appel à du code Java depuis C++ ou, réciproquement, appeler des fonctions C++ depuis une méthode écrite en Java.

En pratique, l'appel depuis un programme Java de méthodes écrites en C++ se réalise comme le montre le Programme 4.13. D'abord, on importe comme module statique la bibliothèque préalablement compilée des fonctions faisant appel à *JNI*. Puis on déclare explicitement la signature des fonctions qu'on compte utiliser. Logiquement, l'appel à l'une de ces fonctions se fait ensuite aussi simplement que l'appel à toute méthode statique, à savoir en ayant préfixé le nom de la méthode par `RPIneqPropag`.

A l'inverse, du point de vue C++ les choses sont un peu plus complexes comme le montre le Programme 4.14. En fait, la création des entêtes des fonctions C++ appelées par Java via *JNI* est réalisée automatiquement au moyen d'un exécutable qui prend en entrée le(s) fichier(s) Java dans le(s)quel(s) sont déclarées les signatures des fonctions utilisées. Ensuite, l'appel à une méthode ou l'utilisation d'un objet, d'une classe Java est réalisé via un ensemble de méthodes C++ dédiées, permettant de faire usage de tout ce dont on disposerait en Java mais alourdissant très fortement la syntaxe du langage comme on peut peut-être déjà s'en rendre compte dans le Programme 4.14.

Ainsi, si *JNI* est beaucoup plus simple à utiliser du côté Java que du côté C++, la bibliothèque permet d'utiliser du code Java dans du code C++ et réciproquement. Elle nous a concrètement permis de faire travailler de concert deux programmes écrits dans chacun de ces deux langages. Mais au delà de l'utilisation de cette bibliothèque, voyons à présent les autres caractéristiques de l'implémentation de la collaboration que nous avons réalisée.

Du point de vue de Choco/Java

Comme on vient juste de le voir, l'utilisation de *JNI* depuis un programme Java est quasiment transparente et n'est source d'aucune complication notable. En outre, la contrainte `RPIneqPropag` que nous avons implémentée dans *Choco* hérite d'un type contrainte abstrait, `AbstractLargeRealIntConstraint`, de façon à être réveillée automatiquement par les événements de modification de domaine. Du reste, c'est

PROG. 4.13 : Déclaration en Java des fonctions C++ appelables via JNI.

```
1  public class RPIneqPropag extends AbstractLargeRealIntConstraint
2  {
3
4    static{System.loadLibrary("RPIneqPropag"); }
5    public static native void parseVarRP(int n, String s, Var[] v) ;
6    public static native void parseCtrRP(String s) ;
7    public static native void propagateRP(Var[] v) ;
8    public static native void statRP() ;
9
10   (...)
11
12 }
```

- **l. 1** : déclaration de la classe `RPIneqPropag`, avec héritage à partir de la classe `AbstractLargeRealIntConstraint`
- **l. 4** : chargement statique de la bibliothèque homonyme de fonctions `RPIneqPropag`
- **l. 5-8** : déclarations des entêtes des fonctions à utiliser dans la bibliothèque

l'essentiel du travail à fournir —du point de vue de *Choco*— pour s'insérer correctement dans le processus de résolution. Elle est dotée d'une méthode `propagate` dans laquelle on va concrètement faire appel à *RealPaver* via des méthodes C++ dédiées que nous avons implémentées.

Mis à part cela, il suffit d'étendre le langage de modélisation de façon à pouvoir faire appel à une méthode d'ajout de contrainte au propagateur et faire le lien, à l'aller comme au retour, entre les domaines traités par *Choco* et ceux retournés par *RealPaver*.

Du point de vue de RealPaver/C++

Quant à l'implémentation en C++ de la partie de la collaboration faisant appel à des méthodes Java, l'essentiel a déjà été abordé plus avant dans cette section. En effet, l'utilisation de *JNI* a tendance à rendre compliquée la plus élémentaire des opérations comme par exemple la recopie des domaines que rend très compliquée de prime abord l'utilisation des primitives *JNI* de tranfert de données Java/C++.

Cela mis-à-part, l'essentiel de l'effort à fournir ici a consisté simplement à comprendre comment instancier puis lancer un propagateur en utilisant les bons objets et méthodes fournis par l'API de *RealPaver* : on fait appel à un objet de type `rp_propagator`, héritant du type opérateur de réduction `rp_operator` et disposant d'une méthode `apply` prenant en paramètre un pavé de domaines et réduisant ce dernier autant que faire se peut.

PROG. 4.14 : Extrait d'une fonction C++ faisant appel à des éléments écrits en Java.

```
1  JNIEXPORT void JNICALL Java_choco_real_constraint_RPIneqPropag_parseVarRP
2    (JNIEnv * env, jclass, jint n, jstring str, jobjectArray vars)
3  {
4
5    jobject  dom = env->GetObjectArrayElement(vars,0) ;
6    jclass cls = env->GetObjectClass(dom) ;
7    jmethodID fid_str = env->GetMethodID(cls, "toString", "()Ljava/lang/String ;") ;
8
9    for (int i=0; i<rp_problem_nvar(problem); ++i)
10   {
11      dom = env->GetObjectArrayElement(vars,i) ;
12      const char* cStr = env->GetStringUTFChars((jstring)env->CallObjectMethod(dom,fid_str),0)
13
14      (...)
15
16   }
17 }
```

- **l. 1-2** : entête de la fonction, généré automatiquement par l'exécutable `jni` à partir d'un fichier java faisant usage de la bibliothèque (cf Prog. 4.13)
- **l. 5** : récupération du premier élément du tableau `vars`
- **l. 6** : instanciation d'un objet représentant la classe de cet objet
- **l. 7** : instanciation d'un objet représentant la méthode `toString` de cette classe
- **l. 9** : boucle de traitement de chacun des éléments du tableau `vars`
- **l. 11** : récupération du i-ème élément du tableau `vars`
- **l. 12** : appel à la méthode `toString` de cet élément et stockage dans une chaîne de la valeur retournée

4.3 Expériences

Dans cette section nous décrivons les expériences que nous avons réalisées pour évaluer le comportement et les performances de la collaboration que nous avons mise en oeuvre.

4.3.1 Description des problèmes

Nous avons utilisé des problèmes de différentes sources et de différentes natures. En ce qui concerne les problèmes d'optimisation, nous avons adopté pour les résoudre la stratégie qui consiste à transformer la recherche de maximum/minimum en une contrainte d'encadrement étant donné une valeur connue de cet optimum. De nombreux problèmes de la base de données *CSPlib* sont en effet construit en utilisant ce principe [GW99].

MINLPlib

MINLPlib est une collection de problèmes d'optimisation mixtes, c'est-à-dire qui font intervenir à la fois des variables discrètes —parmi lesquelles certaines peuvent être booléennes et d'autres entières— et des variables continues [BDM03]. Une petite moitié des problèmes que nous utilisons dans cette section provient de cette bibliothèque :

- `synthes3` : un problème simplifié de synthèse de procédé qui vise à déterminer la structure optimale et les conditions d'opérations d'un procédé qui doit satisfaire des spécifications de conception données [DG86] ;
- `st_miqp5` : un problème de programmation mixte quadratique indéfinie *MIQP* [TS02] ;
- `ex1263` : un problème d'optimisation de découpe de papier cherchant à optimiser le profit en minimisant les pertes de matière première [HWPS98].

Placement d'antennes

Un autre problème consiste à placer des antennes sur une surface rectangulaire de façon à maximiser la distance entre les antennes, étant données des contraintes de distance minimimum et maximum possibles entre deux antennes et matérialisant des limitations techniques[8].

En outre, nous avons considéré une contrainte supplémentaire qui consiste à devoir placer certaines antennes à des positions discrètes et les autres à des positions quelconques. Nous testons deux instances de ce problème pour sa version à 4 antennes, l'une où une seule des antennes doit être placée à des coordonnées entières, l'autre où trois des antennes le doivent.

Fractions

Nous avons également testé le comportement de la collaboration sur des instances d'un problème faisant intervenir non seulement des contraintes arithmétiques mais aussi des contraintes globales discrètes. Il s'agit d'une variante du problème classique de YOSHIGAHARA qui consiste à résoudre l'équation suivante :

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1$$

où chaque lettre est associée à un digit distinct et différent de zéro. Nous avons également étendu ce problème en une deuxième instance, plus difficile à résoudre, en ajoutant des variables J, K, L et un quatrième terme $\frac{J}{KL}$ à l'équation. En outre, afin de garder le problème consistant, nous avons étendu les domaines initiaux jusqu'à 12 au lieu de 9 i.e. les digits s'expriment en base 13 au lieu de la base 10. Notons enfin que pour chacune des instances nous avons également posé des contraintes de façon à briser les symétries, e.g. $\frac{A}{BC} \leq \frac{D}{EF}$ et $4\frac{A}{BC} \geq 1$.

4.3.2 Résultats

Les résultats expérimentaux sont présentés dans la Table 4.2. Tous les calculs ont été réalisés sur un Intel Xeon 3 Ghz avec 16 Go de RAM. Nous avons comparé la collaboration des solveurs *Choco 1.2.06* et *RealPaver 1.0* avec les outils *RealPaver 1.0* seul, *Choco 1.2.06* seul et *ECL^{i}PS^e 5.10*. Chaque colonne correspond à un outil différent `RP`, `COOP`, `Choco` et `ECL`$^{\texttt{i}}$`PS`$^{\texttt{e}}$. Une colonne `#var` renseigne pour chaque instance le rapport du nombre de variables discrètes sur le nombre de variables continues. Les

[8] Les contraintes d'allocation de fréquences ne sont pas prises en compte ici.

temps de résolution en secondes sont donnés sur la première ligne et le nombre de branchements sur la deuxième, exprimés en milliers pour les variables discrètes et pour les variables continues. Un nombre de branchements valant ∅ signifie que ce nombre est sans objet pour le problème et le type de variable donnés. Un point d'interrogation marque un temps de résolution dépassant les 10000 secondes. Une barre horizontale représente une impossibilité de modéliser le problème avec l'outil donné.

Constatons pour commencer que la collaboration des solveurs est la plus efficace pour traiter les instances du problème `fractions`, à savoir les problèmes faisant intervenir à la fois des contraintes arithmétiques nécessitant d'être évaluées grâce à l'arithmétique des intervalles, déléguées à RealPaver, et des contraintes globales traitées par les algorithmes dédiés et state-of-the-art implémentés dans Choco, pour filtrer ici le `alldifferent`. Ce type de problème mixte est donc logiquement celui qui est à même de profiter le plus de ce type de collaboration de solveurs et matérialise le mieux le bien-fondé de l'idée directrice de ces travaux consistant à utiliser les forces de chacun des outils pour combler les faiblesses de l'autre. En pratique, on peut observer ici que la collaboration est quinze fois plus rapide que *RealPaver* seul, ce qui s'explique aisément par le fait que ce dernier utilise une clique de diségalités binaires pour représenter le `alldifferent` et perd ainsi le double avantage d'une information globale à toutes les variables traitée qui plus est de façon optimale par les algorithmes de filtrage de *Choco*. En outre, la collaboration est huit fois plus rapide que *Choco* seul, qui pèche de par sa capacité nettement moins bonne à utiliser les contraintes arithmétiques pour filtrer les domaines et doit donc avoir recours à un nombre de branchements deux fois plus élevé. Enfin, remarquons qu'elle est quatre fois plus rapide qu'*ECL^{i}PS^e* et que cela semble dû à l'efficacité de *RealPaver* pour le calcul sur les intervalles, puisqu'on observe le même nombre de branchements mais pour un temps de résolution bien supérieur.

En outre, la coopération que nous avons mise en oeuvre est largement plus rapide que *Choco* seul dans la très grande majorité des cas. En effet, *RealPaver* est nettement plus efficace que *Choco* pour traiter les contraintes arithmétiques, ce qui peut très certainement s'expliquer par le fait que *RealPaver*

pb	#var	RP	COOP	Choco	ECL^{i}PS^e
synthes3	8/9	0.85 $\frac{0^+}{0.63}$	1.12 $\frac{0^+}{0.29}$	—	0.88 $\frac{0^+}{0.39}$
st_miqp5	2/6	3.1 $\frac{0.2}{4.5}$	5.9 $\frac{0.37}{4.9}$	24.7 $\frac{0.35}{3.9}$	132.96 $\frac{0.47}{35.6}$
ex1263	59/20	66.6 $\frac{84.4}{\varnothing}$	160.0 $\frac{75.6}{\varnothing}$	94.3 $\frac{117.5}{\varnothing}$	21.73 $\frac{169.2}{\varnothing}$
antennes_41	2/6	?	?	?	?
antennes_43	6/2	260.9 $\frac{274.4}{102.8}$	381.3 $\frac{275.1}{108.3}$	1862.4 $\frac{174.6}{66.1}$	839.41 $\frac{146.0}{186.7}$
fractions_10	9/0	2.0 $\frac{6.7}{\varnothing}$	0.91 $\frac{1.3}{\varnothing}$	2.5 $\frac{2.0}{\varnothing}$	1.42 $\frac{0.86}{\varnothing}$
fractions_13	12/0	3351.6 $\frac{667.2}{\varnothing}$	220.9 $\frac{484.7}{\varnothing}$	1707.0 $\frac{1033.7}{\varnothing}$	848.2 $\frac{485.0}{\varnothing}$

Table 4.2 – Temps de résolution en secondes (première ligne) et nombre de branchement sur entier/réel en milliers (deuxième ligne).

a été conçu dans ce but, autour d'un cœur optimisé en C et dédié au calcul sur les intervalles à bornes flottantes, alors que *Choco*, codé en Java, a simplement été étendu après sa conception. L'exception à cette tendance générale, qu'on observe pour `ex1263`, tient au nombre élevé de variables utilisées et montre une faiblesse de notre modèle de collaboration : le temps passé à recopier les domaines d'un solveur à l'autre solveur. Pour cet exemple, on a en effet relevé un temps de transfert s'élevant à plus de 50 secondes. De plus, on voit bien dans cet exemple notamment que *Choco*, moins efficace pour réduire les domaines, doit effectuer plus de branchements.

À l'opposé, *RealPaver* est plus rapide que la collaboration dans la majorité des cas —toutes les instances des problèmes qui ne font pas intervenir de contraintes globales. Le gain procuré par la stratégie d'exploration de *Choco*, énumérant intégralement les domaines discrets, n'est pas compensé par les temps de transfert des domaines. Mais ces temps de transfert, s'il est indéniable qu'ils participent à la différence observée, n'expliquent pas toute la différence et on doit aussi considérer le fait que le point fixe est calculé dans la coopération sur toutes les contraintes à chaque fois, à chaque modification d'une variable, alors que dans *RealPaver* ce rappel des contraintes est plus souple et seules les contraintes impliquant les variables modifiées sont propagées. Ainsi la différence est expliquée à la fois par les temps de transfert des domaines mais aussi par le caractère plus lourd, moins fin, du calcul du point fixe par *Choco* à l'aide de la contrainte utilisant *RealPaver*.

Notons pour finir que ECL^iPS^e apparaît ici bien plus performant sur les problèmes avec un rapport élevé du nombre de variables discrètes sur le nombre de variables continues, comme le montre `ex1263` par opposition à `st_miqp5` par exemple : dans une situation du genre du premier cas, le solveur est très compétitif pour résoudre un problème où les trois quarts des variables sont discrètes —binaires qui plus est— alors que dans le second cas, il est largement dépassé par au moins l'un des autres outils, sinon une majorité d'entre eux.

4.4 Conclusion

Dans ce chapitre nous avons présenté nos travaux concernant la réalisation d'une collaboration entre deux solveurs, l'un dédié aux domaines discrets, l'autre dédié aux domaines continus. Après avoir étudié en détails les capacités de modélisation en termes de problèmes mixtes des principaux outils non commerciaux disponibles actuellement, nous avons entrepris d'utiliser les capacités de propagation sur les contraintes arithmétiques d'un solveur continu pour améliorer les capacités dans ce domaine d'un solveur discret. Nous avons alors pu mesurer l'efficacité de la collaboration que nous avons mise en oeuvre en la comparant à différents outils vis-à-vis de la résolution d'une sélection de problèmes mixtes. À travers ces travaux, nos principales contributions sont les suivantes :

- Nous avons mis au point une plateforme logicielle doté d'un langage de modélisation facilitant l'expression des problèmes mixtes en fusionnant les hautes capacités d'expression de contraintes arithmétiques du langage de modélisation dédié d'un solveur continu, avec les nombreuses primitives de modélisation de contraintes globales d'un solveur discret, améliorant en cela les capacités de chacun des deux outils pris individuellement.

- Nous avons également mis en évidence la nette supériorité en termes d'efficacité de résolution d'une collaboration d'outils pour résoudre les problèmes mixtes qui font intervenir à la fois des contraintes arithmétiques, nécessitant une évaluation par l'arithmétique des intervalles, et des contraintes globales, requérant des algorithmes de filtrage dédiés et optimisés.

En outre, des travaux futurs pourraient permettre de profiter encore davantage des forces d'une telle collaboration en tentant, sinon de s'en affranchir, au moins d'en améliorer les points faibles majeurs que sont actuellement les temps de recopie de domaines d'un solveur à l'autre et le manque de souplesse du propagateur utilisé pour calculer un point-fixe des domaines à partir des contraintes arithmétiques du modèle. En ce qui concerne le premier point, on peut en effet concevoir une symbiose plus profonde entre les deux outils, qui pourrait amener par exemple à proposer une gestion plus intelligente de la recopie des domaines, au moins dans le cas particulier où il n'y a pu se produire aucune réduction et où la recopie en retour est inutile. Quant au deuxième point, on peut également envisager une collaboration plus en profondeur qui permettrait une sélection plus fine des opérateurs de réduction à rappeler en cas de modification de domaines, par exemple en offrant la possibilité de n'effectuer le calcul de point-fixe que sur un sous-ensemble des contraintes arithmétiques déléguées au solveur continu.

V

UTILISATION DE LA CONTRAINTE DISCRÈTE ALLDIFFERENT DANS UN SOLVEUR CONTINU

Lorsque l'oeuvre utile est accomplie et que point la renommée,
Que la personne s'efface : voici la Voie.

— LAO-TSEU, De la Voie et de sa vertu, IX.

Ce chapitre présente une étude expérimentale investigant le champ des techniques de programmation par contraintes dédiées à la résolution de problèmes mixtes, un champ où peu de travaux ont déjà été réalisés [Gel98], [GF03]. Dans ce chapitre, nous nous intéressons à la recherche d'une manière simple de permettre aux solveurs continus de gérer efficacement les contraintes globales et de tirer avantage de cette force pour résoudre des problèmes mixtes. En particulier, nous étudions la possibilité d'utiliser la contrainte `alldifferent` pour modéliser des problèmes mixtes du monde réel et les résoudre dans un solveur continu. Cette contrainte est en effet l'une des plus utilisées et des mieux étudiées.

Le chapitre est organisé comme suit. D'abord, nous présentons ce qui nous motive à aller dans cette direction. Ensuite, nous montrons comment utiliser la contrainte `alldifferent` pour la modélisation de quelques problèmes mixtes du monde réel. Nous montrons alors les solutions possibles pour la formulation de la contrainte et son filtrage dans un solveur continu. Enfin, nous présentons les résultats des expériences que nous avons réalisées pour comparer leur comportement, avant de conclure le chapitre.

5.1 Motivations

Les problèmes d'optimisation non linéaire mixtes-entiers (MINLP) sont des problèmes d'optimisation sous contraintes qui font intervenir à la fois des variables discrètes et des variables continues, dotés d'une fonction-objectif éventuellement non convexe et de contraintes qui peuvent être non linéaires. Des problèmes de ce genre apparaissent dans de nombreux domaines d'application comme par exemple la bioinformatique, l'optimisation de portefeuilles d'actions et le design en ingénierie. La plupart de ces méthodes de résolution sont basées sur le cadre classique du branch-and-bound et ont recours à des techniques de différentes sortes, comme les relaxations linéaires, les méthodes de coupes ou encore la reformulation symbolique.

Récemment, une tendance majeure a émergé dans le champ de la recherche opérationnelle, la convergence avec la programmation par contraintes. Elle est dédiée à la recherche de moyens pour la première de tirer avantage de méthodes issues de la seconde [Hoo07]. En fait, il se trouve qu'intégrer la programmation par contraintes et la recherche opérationnelle est une tendance très naturelle puisqu'elles ont des buts très similaires. L'utilisation de l'arithmétique des intervalles en programmation par contraintes est par exemple un moyen puissant de résoudre les MINLP de façon rigoureuse et complète, i.e. de les résoudre avec les deux intéressantes caractéristiques de calculer des solutions garanties et de ne perdre aucune solution. Cela devient très intéressant quand on considère le fait qu'un outil de dernière génération bien connu comme *BARON* n'est pas digne de confiance en général [BSG*10], [LMR07].

Mais ce qui nous intéresse particulièrement ici est l'utilisation de contraintes globales en recherche opérationnelle. Les contraintes globales sont de puissantes primitives de modélisation qui sont très largement utilisées dans le domaine de la programmation par contraintes. Les utilisateurs disposent d'un catalogue de plusieurs centaines de contraintes préconstruites parmi laquelle ils n'ont qu'à choisir celle qui convient le mieux au problème qu'ils veulent modéliser [BCDP07]. En outre, chaque contrainte globale a son propre algorithme de filtrage dédié de façon à ce que le processus de résolution s'adapte le mieux au problème modélisé et rejette facilement de larges parties de l'espace de recherche. L'utilisation de contraintes globales en recherche opérationnelle a ainsi deux grands avantages [Hoo07] :

- rendre les modèles plus faciles à construire et à mettre à jour ;
- révéler la structure du problème au solveur pour permettre de le résoudre plus efficacement.

5.2 Modéliser des MINLP avec la contrainte AllDifferent

Dans cette section, nous donnons trois exemples de MINLP qui peuvent être utilement modélisés en utilisant la contrainte discrète `alldifferent`. Ce sont des problèmes du monde réel qu'on peut trouver dans la littérature. Notre but ici est seulement de montrer comment utiliser la contrainte `alldifferent` pour une modélisation à la fois plus facile et plus concise. Les lecteurs intéressés qui auraient besoin de plus d'informations à propos de la modélisation complète des problèmes pourront se référer aux articles correspondants.

5.2.1 Exemples

Ordonnancement robuste à une machine

Pour commencer, nous présentons la formulation en programmation mixte-entière de MONTEMANNI pour le problème d'ordonnancement robuste à une machine avec des données intervalles qu'on peut

trouver dans [Mon07], où l'on considère pour chaque tâche un intervalle de temps de traitements possibles. L'optimisation est réalisée selon un critère de robustesse, i.e. on cherche l'ordonnancement qui se comporte le mieux dans le pire des cas.

Chacune de N tâches données doit se voir affecter un ordre de traitement distinct, ce qui en programmation non linéaire mixte-entière est normalement modélisé en utilisant des variables binaires, e.g. x_{ik} est vraie quand la tâche i est traitée en $k^{\text{ème}}$ position, et sous la contrainte qu'à chaque tâche (resp. position) soit affectée exactement une position (resp. tâche) :

$$\begin{cases} \sum_{i=1}^{N} x_{ik} = 1, \\ \sum_{k=1}^{N} x_{ik} = 1. \end{cases}$$

Ce schéma correspond parfaitement à la signification de la contrainte globale `alldifferent`. Étant donné un ensemble de N variables entières t_i prenant chacune sa valeur dans $\{1, \ldots, N\}$, les deux contraintes ci-dessus peuvent être reformulées de la manière suivante :

$$\text{alldifferent}([t_1, \ldots, t_N]).$$

Le modèle que nous fournissons ici pour le problème d'ordonnancement robuste résulte d'une technique de reformulation bien connue qui exploite la propriété d'unimodularité du sous-problème de maximisation [Law76] :

$$\min \sum_{i=1}^{N} \eta_i + \sum_{k=1}^{N} \tau_k$$

s.c.

$$\begin{aligned}
&(1) && \text{alldifferent}(t_1, \ldots, t_N), \\
&(2) && t_i = \sum_{k=1}^{N} x_{ik} \cdot k, \\
&(3) && \sum_{k=1}^{N} x_{ik} = 1, \\
&(4) && \eta_i + \tau_k \geq \overline{p_i} \sum_{j=1}^{k} (k-j) x_{ij} + \underline{p_i} \sum_{j=k}^{N} (k-j) x_{ij}, \\
& && x_{ik} \in \{0, 1\}, \\
& && t_i \in \{1, \ldots, N\}, \\
& && \eta_i \in \mathbb{R}, \\
& && \tau_k \in \mathbb{R}.
\end{aligned}$$

La contrainte (1) exprime que chaque tâche doit avoir un ordre de traitement différent ; (2) transmet cette valeur aux variables binaires, ce que requiert la contrainte (4). La contrainte (3) empêche cette transmission d'impliquer deux variables binaires x_{ik} simultanément vraies pour un même i. Enfin, (4) fait simplement le lien entre les variables dont on maximise la somme et l'expression arithmétique du coût de regret pour le pire des scénarios associé à cet ordonnancement. De plus amples détails sont disponibles dans [Mon07].

Ordonnancement de lot économique avec décroissance de la performance

Nous introduisons à présent un modèle en programmation mathématique pour le problème de l'ordonnancement de lot économique (ELSP) avec décroissance de la performance, dont l'objectif est de minimiser le coût total de production —composé des coûts d'initialisation et de stockage— par un ordonnancement cyclique d'un ensemble donné de produits pour lesquels il y a une demande continue et fixée [CLPP05].

Comme dans l'exemple précédent, le `alldifferent` est une façon naturelle d'exprimer cette idée d'ordonnancement qui d'habitude est exprimée dans ce type de modèles par des contraintes booléennes :

$$\begin{cases} \sum_{i=1}^{N} Z_{ij} = 1, \\ \sum_{j=1}^{N} Z_{ij} = 1. \end{cases}$$

Ici aussi, étant donné un ensemble de N variables entières n_i prenant leur valeur dans $\{1, \ldots, N\}$, les deux contraintes ci-dessus peuvent être reformulées en utilisant la contrainte globale `alldifferent` :

$$\text{alldifferent}([n_1, \ldots, n_N]).$$

Nous donnons à présent un aperçu du modèle complet :

$$\min \frac{1}{Tc} \left[\begin{array}{l} \sum_i Cinvf_i \Big(\frac{1}{2}(G_i\gamma_i - d_i)TP_i^2 + \frac{G_i\alpha_i}{\beta_i} TP_i \\ + \frac{G_i\alpha_i}{\beta_i^2}(e^{-\beta_i TP_i} - 1) + \frac{1}{2} d_i (Tc - TP_i)^2 \Big) \\ + \sum Cf_i G_i TP_i + \sum_i \sum_j Ctr_{ij} Z_{ij} \end{array} \right]$$

s.c.

$$\begin{aligned}
&(1) && \text{alldifferent}(n_1, \ldots, n_N), \\
&(2) && n_i = \sum_j Z_{ij} \cdot j, \\
&(3) && \sum_j Z_{ij} = 1, \\
&(4) && d_i Tc \leq G_i\gamma_i TP_i + G_i \tfrac{\alpha_i}{\beta_i}(1 - e^{-\beta_i TP_i}), \\
&(5) && Tc = \sum_i TP_i + \sum_i \sum_j \tau_{ij} Z_{ij} + \sum_i \sum_j sl_{ij}, \\
&(6) && -(1 - Z_{ij})\overline{Tc} \leq TS_j - (TS_i + TP_i + \tau_{ij} + sl_{ij}), \\
&(7) && (1 - Z_{ij})\overline{Tc} \geq TS_j - (TS_i + TP_i + \tau_{ij} + sl_{ij}), \\
& && Tc_i, TP_i, TS_i, d_i, \tau_{ij}, sl_{ij} \in \mathbb{R}. \\
& && n_i \in \{1, \ldots, N\}, \\
& && Z_{ij} \in \{0, 1\},
\end{aligned}$$

La fonction objectif à optimiser est composée de la somme des coûts par cycle du stockage, des matières premières et de l'entretien. Comme dans l'exemple précédent, la contrainte (1) exprime que chaque tâche doit avoir un ordre de traitement distinct, (2) transmet cette valeur aux variables binaires comme le nécessite la contrainte (5). La contrainte (3) empêche cette transmission de faire intervenir

deux variables Z_{ij} simultanément vraies pour un i donné. La contrainte (4) exprime que pour chaque produit, on impose que la quantité produite soit supérieure à la demande pendant le cycle, tandis que la contrainte (5) impose que la durée du cycle soit supérieure à la somme des temps de traitement et de transition entre produits. Pour finir, les contraintes (6) et (7) déclarent que chaque produit ne peut commencer à être produit qu'après le précédent traitement suivi d'une réinitialisation de la chaîne de production pour le bon produit. De plus amples détails sont disponibles dans [CLPP05].

Motif optimal de rechargement d'un réacteur nucléaire

Le modèle que nous présentons à présent est une version simplifiée mais réaliste du problème du motif optimal de rechargement d'un réacteur nucléaire [QGH*99]. Il fait également partie de la collection *MINLPlib* de modèles MINLP académiques et industriels [BDM03].

Ce problème consiste à trouver le motif optimal de rechargement en combustible du cœur d'un réacteur nucléaire. Dans le modèle que nous considérons ici, l'objectif est de maximiser en fin de cycle le rapport du taux de neutrons produits sur le taux de neutrons perdus —ce qu'on appelle la *réactivité* du cœur— tout en satisfaisant des contraintes de sécurité, et sachant que les propriétés de chaque barre de combustible dépendent de sa place dans le réacteur au cycle précédent.

La trajectoire d'un cycle à l'autre des barres dans le cœur du réacteur peut être représentée par une matrice de la forme suivante :

$$\begin{array}{ccccccc} 4 & \to & 7 & \to & 3 & \to & 1 \\ 2 & \to & 8 & \to & 10 & \to & 12 \\ 5 & \to & 11 & \to & 6 & \to & 9 \end{array}$$

Ci-dessus on peut voir une trajectoire possible, d'une durée de 4 cycles, pour un réacteur de 12 noeuds et dans lequel 3 barres de combustible —soit un quart du nombre total de barres— sont retirées du cœur à chaque fin de cycle. Chaque nombre y réfère à un noeud différent du réacteur. Selon cette matrice, les barres placées aux noeuds 1, 12 et 9 sont retirées à chaque rechargement pendant que des barres fraîches sont insérées dans les noeuds 4, 2 et 5. La barre au noeud 7 est déplacée au noeud 3, la barre au noeud 10 est déplacée au noeud 12, etc.

Comme précédemment, l'approche MINLP classique pour modéliser l'affectation d'un noeud à une cellule distincte de la matrice et réciproquement est d'utiliser des variables binaires : $x_{i,l,m}$ est valuée à *vrai* lorsque que la barre au noeud i est affectée à la colonne l et à la ligne m dans la matrice de trajectoire. On doit ensuite contraindre ces variables de manière à ce qu'une exactement soit mise à *vrai* pour un i donné puisque chaque noeud doit apparaître dans la matrice une fois et une seule. Réciproquement, une variable et une seule doit être mise à *vrai* pour une position donné (l, m) de la matrice puisque chaque cellule doit se voir affecter exactement un noeud. Ainsi on a :

$$\begin{cases} \displaystyle\sum_{i=1}^{N} x_{i,l,m} = 1, \\ \displaystyle\sum_{l=1}^{L}\sum_{m=1}^{M} x_{i,l,m} = 1. \end{cases}$$

Il s'avère que le problème d'affectation des noeuds du réacteur aux cellules de la matrice de trajectoire peut s'exprimer au moyen d'une contrainte `alldifferent`. En effet, considérons la numérotation suivante des cellules :

1	2	3	4
5	6	7	8
9	10	11	12

Ainsi, à chaque noeud i doit être affecté un numéro distinct n_i, compris dans $\{1, \ldots, 12\}$, ce qui une fois encore correspond parfaitement avec la signification de la contrainte `alldifferent`. La contrainte d'affectation de la matrice de trajectoire est donc simplement modélisée comme suit :

$$\text{alldifferent}([n_1, \ldots, n_N]).$$

Nous donnons à présent un aperçu du modèle complet du problème de motif optimal de rechargement du réacteur nucléaire où chaque cycle est divisé en un nombre discret d'étapes :

$$\max_{x,k^{\infty},R,k^{eff}} k_T^{eff}$$

s.c.

$$
\begin{aligned}
&(1) && \text{alldifferent}(n_1, \ldots, n_N),\\
&(2) && n_i = \sum_{l,m} x_{i,l,m} \cdot (l + (m-1) \cdot L),\\
&(3) && \sum_l \sum_m x_{i,l,m} = 1,\\
&(4) && k_{i,t}^{\infty} R_{i,t} \leq \frac{f^{lim}}{N}\\
&(5) && k_{i,1}^{\infty} = \sum_{l>1} \sum_m x_{i,l,m} \sum_j x_{j,l-1,m}\, k_{j,T}^{\infty}\\
& && \quad +(\sum_m x_{i,1,m}) k^{fresh},\\
&(6) && k_{i,t+1}^{\infty} = k_{i,t}^{\infty} - (\alpha P_c \Delta_t) k_{i,t}^{\infty} R_{i,t},\\
&(7) && \sum_i k_{i,t}^{\infty} R_{i,t} = 1\\
&(8) && k_T^{eff}\, R_{i,j} = \sum_j G_{i,j} k_{j,T}^{\infty} R_{j,t},\\
& && k_t^{eff} \geq 0,\\
& && k_{i,t}^{\infty} \geq 0,\\
& && R_{i,t} \geq 0,\\
& && n_i \in \{1, \ldots, N\},\\
& && x_{i,l,m} \in \{0, 1\},
\end{aligned}
$$

Les contraintes (1) à (3) modélisent le problème d'affectation de la matrice de trajectoire : (1) exprime que chaque noeud doit y avoir une position distincte tandis que (2) transfère cette position aux variables binaires pour que la contrainte (5) puisse accéder à cette information et sache quels noeuds se sont vus affecter une barre fraîche ; (3) empêche ce transfert d'impliquer deux variables $x_{i,l,m}$ mises à *vrai* pour un même i donné. Les contraintes (4) à (7) décrivent le comportement des barres de combustible en termes de lois physiques : (4) requiert que la puissance maximum par noeud ne soit pas trop grande, pour des raisons évidentes de sécurité ; (5) définit la valeur initiale du coefficient de multiplication de puissance de chaque noeud selon qu'il contient ou non une barre fraîche ; (6) décrit la décroissance de la réactivité locale ; (7) sert à normaliser la puissance en chaque noeud. Enfin, la contrainte (8) exprime la réactivité moyenne du cœur en fonction de la réactivité locale à chaque noeud et de la probabilité G_{ij} qu'un neutron émis au noeud i soit absorbé au noeud j.

5.2.2 Synthèse

Ces trois exemples nous ont permis de bien comprendre la façon dont on peut utiliser la contrainte discrète `alldifferent` pour modéliser des problèmes MINLP issus du monde réel. L'idée est d'être capable de répérer si une partie du modèle ne correspond pas à un problème d'affectation et/ou d'identifier, dans les modèles de problèmes de type ordonnancement/affectation sous contraintes, s'il est possible de trouver une partie avec la forme générique suivante :

$$\begin{cases} \sum_{i_1=1}^{N_{i_1}} \dots \sum_{i_p=1}^{N_{i_p}} & x_{i_1,\dots,i_p,j_1,\dots,j_q} = 1, \\ \sum_{j_1=1}^{N_{j_1}} \dots \sum_{j_q=1}^{N_{j_q}} & x_{i_1,\dots,i_p,j_1,\dots,j_q} = 1, \end{cases}$$

où $N_{i_1} \cdot \ldots \cdot N_{i_p} = N_{j_1} \cdot \ldots \cdot N_{j_q} = M$. En présence d'un motif de contraintes du type ci-dessus, on peut le remplacer par une contrainte `alldifferent` portant sur un ensemble de variables n_k prenant chacune sa valeur dans $\{1, \dots, M\}$:

$$\text{alldifferent}([\text{n}_1, \dots, \text{n}_\text{M}])$$

5.3 Traiter le AllDifferent dans un solveur continu

Dans cette section, nous passons en revue les différentes façons dont on peut traiter la contrainte `alldifferent`, soit en la transformant en un ensemble de contraintes connues du solveur, soit en implémentant des algorithmes de filtrages dédiés.

5.3.1 Reformulations de la contrainte

5.3.1.1 Clique de contraintes binaires

La première et la plus simple des façons de modéliser une diségalité entre un certain nombre de variables est d'utiliser une clique de contraintes binaires, reliant chaque variable avec les autres par autant de relations binaires de façon à ce que toutes les paires de variables ne puissent recevoir la même valeur. Dans le cas qui nous concerne, cette relation binaire peut être ou bien une diségalité ou bien une contrainte de distance.

Une formulation naïve de la contrainte `alldifferent` sur n variables se base ainsi sur une clique de diségalités entre paires de variables, ajoutant de cette façon au modèle un nombre de nouvelles contraintes en $O(n^2)$. Notons que quand le solveur ne dispose pas de contrainte de diségalité, on peut utiliser à la place une contrainte de distance supérieure ou égale à 1 entre deux variables :

$$\forall i,\ j,\ 1 \leq i < j \leq n,\ \{|x_i - x_j| \geq 1\}$$

où la valeur absolue peut également être remplacée par le carré $(x_i - x_j)^2$. De cette manière, il reste possible d'exprimer un modèle faisant appel à la contrainte `alldifferent` dans un solveur qui n'implémente pas de contrainte de diségalité. Si de nombreuses opportunités de filtrage sont perdues dans une telle reformulation, cela ne requiert du solveur que de fournir une contrainte de diségalité ou un opérateur de valeur absolue.

5.3.1.2 Formulation MILP

Du point de vue de la programmation mathématique, où d'ordinaire il n'y pas de mot-clé `alldifferent` dans le langage de modélisation, et comme on a pu s'en apercevoir dans la section précédente, on exprime la contrainte `alldifferent` en utilisant des variables booléennes [Hoo07] :

$$\begin{cases} \sum_{j=1}^{m} y_{ij} = 1, & i = 1, \ldots, n \\ \sum_{i=1}^{n} y_{ij} \leq 1, & j = 1, \ldots, m \end{cases}$$

Une variable y_{ij} valant *vrai* signifie que la $i^{\text{ème}}$ variable prend pour valeur la $j^{\text{ème}}$ valeur de son domaine. De nombreux problèmes d'affectation utilisent ce genre de modélisation. Quand il y a autant de valeurs dans le domaine que de variables à contraindre par le `alldifferent`, on remplace par un égal l'inférieur-ou-égal dans la seconde équation :

$$\begin{cases} \sum_{j=1}^{n} y_{ij} = 1, & i = 1, \ldots, n \\ \sum_{i=1}^{n} y_{ij} = 1, & j = 1, \ldots, n \end{cases}$$

Pour finir, remarquons qu'il est possible au besoin de faire le lien pour un i donné entre les variables booléennes y_{ij} et la valeur entière affectée à la variable x_i. Considérant par exemple $x_i \in \{v_1, \ldots, v_n\}$, on a :

$$x_i = \sum_{j=1}^{m} v_j y_{ij}, \quad i = 1, \ldots, n$$

5.3.2 Implémentation de filtrages dédiés

5.3.2.1 Relaxation convexe d'enveloppe

On peut renforcer la modélisation sous forme de clique décrite ci-dessus en ayant recours à la relaxation convexe d'enveloppe de la contrainte `alldifferent` ainsi que la définit HOOKER dans [Hoo02] pour les domaines intervalles d'entiers :

$$\begin{cases} \sum_{j=1}^{n} x_j = \dfrac{n(n+1)}{2} & (1) \\ \sum_{j \in J} x_j \geq \dfrac{|J|(|J|+1)}{2}, \ \forall J \subset \{1, \ldots, n\} \, avec \, |J| < n & (2) \end{cases}$$

à condition que $x_j \in [1, n]$. Si on a $x_j \in [a, b]$ avec $a \neq 1$, il suffit de considérer la translation $y_j = x_j - a + 1 \in [1, b - a + 1]$.

Cependant, l'ensemble complet de contraintes de la relaxation décrite par la formule ci-dessus contient un nombre de contraintes qui croît exponentiellement avec le nombre de variables utilisées par le modèle, si bien qu'il devient rapidement impossible d'utiliser concrètement l'ensemble de contraintes ainsi décrit.

Heureusement, on peut savoir quelles sont les contraintes de la relaxation qui sont le plus susceptibles de pouvoir filtrer les domaines étant donné un ensemble de domaines à réduire :

$$\begin{cases} \sum_{i=1}^{k} x_i \geq \frac{k(k+1)}{2} \ \forall\ k < N \ tel\ que \ \sum_{i=1}^{k} \underline{x_i} < \frac{k(k+1)}{2} \\ \sum_{i=k}^{N} x_i \leq \frac{k(k+1)}{2} \ \forall\ k > 1 \ tel\ que \ \sum_{i=k}^{N} \overline{x_i} > \frac{N(N+1)}{2} - \frac{k(k-1)}{2} \end{cases}$$

à condition de classer les variables en ordre croissant (resp. décroissant) de la borne inférieure (resp. supérieure) de leur domaine i.e. $\underline{x_i} \leq \underline{x_j}\ \forall i \leq j$ (resp. $\overline{x_i} \geq \overline{x_j}\ \forall i \geq j$).

La formule ci-dessus s'implémente facilement en un algorithme de filtrage de bornes, en se servant de la première ligne pour filtrer les bornes inférieures et de la seconde pour filtrer les bornes supérieures

ALG. 5.1 : ReviseLowerBound(*Box B*)

```
1  B' ← sort(B)
2  sum ← 0
3  pour k ← 1 à |B'| − 1 faire
4      si |B'_k| = |B'_{k+1}| = 1 et B'_k = B'_{k+1} alors
5          retourner false
6      fin
7      sum ← sum + B'_k
8      si sum < k(k+1)/2 alors
9          B' ← reduce(B', Σ_{i=1}^{k} x_i ≥ k(k+1)/2)
10         si empty(B') alors
11             retourner false
12         fin
13     fin
14 fin
15 retourner true
```

- **l. 1** : tri des domaines par ordre croissant de borne inférieure
- **l. 2** : initialisation de l'accumulateur
- **l. 3** : début de la boucle de traitement des domaines
- **l. 4-6** : test pour savoir si deux variables sont instanciées à la même valeur, échec de la fonction le cas échéant
- **l. 7** : mise-à-jour de l'accumulateur
- **l. 8** : test pour savoir si on satisfait la condition d'application du filtrage
- **l. 9** : filtrage des domaines en utilisant la contrainte d'diségalité sur les k premiers domaines
- **l. 10-12** : échec de la fonction si le domaine obtenu est vide

ALG. 5.2 : Reduce($Box\ B$)

1 $B \leftarrow reduce\left(B, \sum_{i=1}^{N} x_i = \frac{N(N+1)}{2}\right)$
2 **si non** $empty(B)$ **alors**
3 | retourner $ReviseLowerBound(B)$ **et** $ReviseUpperBound(B)$
4 **fin**

- **l. 1** : filtrage en utilisant la contrainte de somme sur toutes les variables
- **l. 2-4** : tentative de mise-à-jour des bornes inférieures et supérieures

des domaines des variables. Nous ne donnons ici que l'algorithme pour filtrer les bornes inférieures —cf Algo. 5.1— puisque l'autre est trivial à obtenir par symétrie. Réduire un pavé consiste alors à utiliser en même temps l'égalité (1) de la formule de la relaxation, et les inégalités de la forme ci-dessus, que nous qualifierons d'*ad hoc*, comme le montre l'Algorithme 5.2.

Pour finir, signalons que nous voulons également tester expérimentalement une deuxième façon d'utiliser cette relaxation dont nous venons de décrire l'usage communément pratiqué. Dans la section suivante, nous comparerons deux versions du filtrage :

- en utilisant l'égalité (1) et les inégalités ad hoc ;
- en utilisant l'égalité (1) seule.

5.3.2.2 Contrainte de cardinalité de domaine

Nous présentons à présent l'opérateur # de cardinalité de domaine, qui s'applique sur une union d'intervalles d'entiers et retourne le nombre de valeurs distinctes qu'elle contient. À présent supposons qu'une contrainte `alldifferent` tienne pour un ensemble donné de variables $x_1, \ldots, x_n$. On a :

$$\#(D_{x_1} \cup \cdots \cup D_{x_n}) \geq n$$

où D_{x_i} est le domaine de x_i pour tout i. Ceci est en fait une condition nécessaire bien connue, basée sur la notion d'intervalle de HALL —notion qu'on détaillera davantage dans la sous-section suivante— en particulier l'intervalle de HALL associé à tout l'ensemble de variables [Hal35].

L'algorithme de filtrage dédié que nous avons implémenté est présenté comme l'Algorithme 5.3. En premier lieu, si l'opérateur de cardinalité de domaine calcule un entier inférieur à n le nombre de variables, alors la contrainte ne tient pas et retourne une inconsistance. Deuxièmement, quand une variable est instanciée à une valeur, cette valeur est retirée des domaines de toutes les variables pour lesquelles il s'agit d'une valeur en borne de domaine. Cette technique correspond d'ailleurs au traitement classique d'une contrainte de diségalité : étant donnés $x \neq y$ et $x = a$ alors a est retiré du domaine de y.

ALG. 5.3 : Reduce($Box\ B$)

```
1  u ← ∅
2  pour k ← 1 à |B| faire
3      u = u ∪ B_k
4      si |B_k| = 1 alors
5          pour l ∈ {1, ..., k − 1, k + 1, ..., |B|} faire
6              B_l = B_l \ B_k
7              si empty(B_l) alors
8                  retourner false
9              fin
10         fin
11     fin
12     si |u| < |B| alors
13         retourner false
14     sinon
15         retourner true
16     fin
17 fin
```

- **l. 1** : initialisation de l'union des domaines
- **l. 2** : début de la boucle de traitement des domaines
- **l. 3** : ajout du domaine courant à l'union
- **l. 4-6** : dans le cas où le domaine courant est réduit à une valeur, celle-ci est retirée des autres domaines (à condition qu'elle soit une borne pour ces derniers)
- **l. 7-9** : échec de la fonction si un domaine se retrouve vide
- **l. 12-16** : test du nombre de valeurs contenues dans l'union des domaines, ce qui détermine l'échec ou la réussite de la fonction

5.3.2.3 Consistance de bornes

Nous pouvons également implémenter un algorithme dédié de filtrage par consistance de bornes comme celui donné dans [LOQTvB03], ou bien encore un algorithme classique de matching comme celui décrit dans [Rég94]. Par ailleurs, dans le cadre d'une étude expérimentale utilisant un solveur travaillant sur des intervalles, nous nous focalisons sur le cas des domaines intervalles d'entiers sans trous. Puisque nous savons déjà que l'algorithme de matching de RÉGIN n'est pas le plus efficace sur ce type de domaines [vH01], nous nous intéresserons plutôt à l'algorithme de consistance de bornes donné dans [LOQTvB03], i.e. la version la plus récente et la plus efficace des évolutions de la version initiale de l'algorithme bien connu de PUGET [Pug98].

L'idée clé sur laquelle se base cet algorithme est la recherche d'intervalles de HALL, c'est-à-dire des intervalles dont la taille est égale au nombre de variables dont ils peuvent contenir le domaine. Par exemple, étant données quatre variables contraintes par une contrainte `alldifferent`, e.g. $x_1 \in \{1, 2\}$, $x_2 \in \{2, 3\}$, $x_3 \in \{1, 2, 3\}$ et $x_4 \in \{1, \ldots, 5\}$, l'intervalle d'entiers $I_H = \{1,\ 2,\ 3\}$ est un intervalle de HALL

ALG. 5.4 : UpdateLowerBounds(*Box B*)

```
1  pour i ← 1 à nb + 1 faire
2      t[i] ← h[i] ← i − 1
3      d[i] ← bounds[i] − bounds[i − 1]
4  fin
5  pour i ← 0 à niv − 1 faire
6      x ← maxsorted[i].minrank
7      y ← maxsorted[i].maxrank
8      z ← pathmax(t, x + 1)
9      j ← t[z]
10     d[z] ← d[z] − 1
11     si d[z] = 0 alors
12         t[z] ← z + 1
13         z ← pathmax(t, t[z])
14         t[z] ← j
15     fin
16     pathset(t, x + 1, z, z)
17     si d[z] < bounds[z] − bounds[y] alors
18         retourner false
19     fin
20     si h[x] > x alors
21         w ← pathmax(h, h[x])
22         B_k ← maxsorted[i].min ← bounds[w]
23         pathset(h, x, w, w)
24     fin
25     si d[z] = bounds[z] − bounds[y] alors
26         pathset(h, h[y], j − 1, y)
27         h[y] ← j − 1
28     fin
29 fin
30 retourner true
```

- **l. 1-4** : initialisation des structures de données
- **l. 5** : début de la boucle de traitement des domaines
- **l. 6-9** : récupération des informations de l'intervalle courant
- **l. 10-16** : traitement de l'intervalle courant
- **l. 17-19** : échec détecté
- **l. 20-24** : mise-à-jour de la borne inférieure du domaine courant
- **l. 25-28** : détection d'un nouvel intervalle de HALL

par rapport à l'ensemble de variables $\{x_1, x_2, x_3\}$ et il est impossible pour la variable x_4 de se voir affecter une des valeurs contenue dans I_H : il n'y a en effet pas d'autre choix que celui d'affecter ces valeurs à x_1, x_2 et x_3 —peu importe quelle variable prend quelle valeur— et ces valeurs peuvent alors être retirées du domaine initial de x_4 qui est ainsi réduit à $\{4, 5\}$.

L'algorithme complet pour le filtrage des bornes inférieures est donné ici bien qu'il soit un peu difficile à comprendre de prime abord. Il s'agit de l'Algorithme 5.4 et il se dérive symétriquement en un algorithme de filtrage des bornes supérieures que par contre nous ne donnerons pas ici. Ces deux algorithmes font usage d'un certain nombre de structures de données dédiées, triées, qui ont déjà été optimisées plusieurs fois depuis la première parution de l'algorithme dans [Pug98]. De plus, nous n'expliquerons pas son fonctionnement en détails ici car cela prendrait une place importante et là n'est pas le but de cette section. Cela nous paraît par contre important de montrer cet algorithme ici, de façon à ce que chacun puisse le comparer avec les précédents algorithmes présentés dans cette section et puisse réaliser qu'il a un degré beaucoup plus élevé de complexité dans son principe, en particulier quand on le compare à l'Algorithme 5.3. En conséquence, des lecteurs intéressés qui auraient besoin de plus d'explications ne devront pas hésiter à se référer à l'abondante littérature couvrant ce sujet, à commencer par [Pug98], [LOQTvB03] et [vH01].

5.4 Expériences

Notre but est d'étudier l'impact sur la résolution du choix d'une modélisation et d'une méthode de filtrage pour le `alldifferent`. Comme on l'a déjà mentionné à plusieurs reprises, peu ou pas de travaux ont déjà été réalisés dans ce domaine de la programmation par contraintes pour les problèmes mixtes, faisant intervenir à la fois des contraintes globales, discrètes, et des contraintes non linéaires, continues.

5.4.1 Description des problèmes

Nous avons effectué des expériences en utilisant trois sortes de problèmes :

- des problèmes académiques : des puzzles bien connus, purement discrets ;
- des problèmes artificiels : des problèmes que nous avons créé à des fins de test, allant du mixte jusqu'au purement discret ;
- des MINLP issus du monde réel : des problèmes mixtes que nous avons trouvé dans la littérature[1] et qui peuvent utilement être modélisés avec la contrainte `alldifferent` .

La Table 5.1 résume les caractéristiques principales des instances que nous avons résolues de chacun de ces types de problèmes.

5.4.1.1 Problèmes académiques

Nous avons réalisé une partie de nos expériences sur des problèmes académiques bien connus, purement discrets, faisant usage de la contrainte `alldifferent`. Bien qu'il ne s'agisse pas à proprement parler de problèmes mixtes puisque toutes les variables y sont entières, il est déjà intéressant d'observer le comportement des méthodes que nous comparons dans ce cas particulier de problèmes discrets,

[1] Il s'agit bien entendu des problèmes que nous avons présentés au début de ce chapitre.

name	#var	#sol
sevvoth	9/0	1
fractions	12/0	68
magic-square	16/0	7040
queens-10	30/0	724
queens-11	33/0	2680
queens-12	36/0	14200
int-frac-12	12/0	24
int-frac-14	14/0	49
int-frac-16	16/0	1068
mixed-sin-5	5/5	121
mixed-sin-6	6/6	746
mixed-sin-7	7/7	5041
hard-mixed-sin-4	4/4	24
hard-mixed-sin-5	5/5	832
hard-mixed-sin-6	6/6	18228
nuclear-A	90/57	1
nuclear-B	156/50	1
sched-9	90/19	1
sched-10	110/21	1
sched-11	132/23	1
ELSP-8	64/74	1
ELSP-9	81/92	1
ELSP-10	100/112	1

Table 5.1 – Pour chaque problème, #var donne le nombre variables discrètes/continues dans le modèle utilisant la contrainte alldifferent , et #sol représente le nombre de solutions au problème.

puisqu'en effet ils font intervenir à la fois des contraintes globales et des contraintes arithmétiques nécessitant une évaluation aux intervalles. Cette sous-section présente brièvement les problèmes que nous avons utilisés à cet effet.

Sevvoth

Le problème *Sevvoth* [Kei] est un problème du type *SEND+MORE=MONEY*, dans lequel les mots à additionner sont ceux du poème ci-après, le terme résultant étant le mot *SEVVOTH*[2] lui-même. Comme dans tous les problèmes de ce genre, un chiffre est associé à chaque lettre et tous doivent être distincts les uns des autres —ce qu'on modélise bien sûr au moyen d'une contrainte alldifferent. On ajoute alors la contrainte arithmétique qui représente la somme globale, chaque mot *MOT* générant un terme comme $M * 100 + O * 10 + T$. Le poème utilisé est le suivant :

[2]Il s'agit du nom d'une île mythique quelque part en Mer du Nord.

Ten herons rest near North Sea shore
As tan terns soar to enter there.
As herons nest on stones at shore,
Three stars are seen ; tern snores are near !

Fractions

Le problème *fractions* est une extension du problème classique de Yoshigahara qui vise à résoudre l'équation suivante :

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1$$

où chaque lettre est associée à un digit distinct et différent de zéro. Pour compliquer le problème, nous avons comme au chapitre précédent ajouté trois variables J, K, L et un quatrième terme $\frac{J}{KL}$ à l'équation. Pour garder le problème consistant, nous avons également dû étendre les domaines initiaux de façon à ce que chaque digit s'exprime en base 13. Nous avons également imposé les contraintes suivantes, de façon à casser les symétries et limiter le nombre de solutions :

$$\begin{cases} \frac{A}{BC} \leq \frac{D}{EF} \leq \frac{G}{HI} \leq \frac{J}{KL}, \\ 4 * \frac{A}{BC} \geq 1, \\ 4 * \frac{J}{KL} \leq 1. \end{cases}$$

Carré magique

Le problème du *carré magique* est un puzzle mathématique bien connu où l'on doit remplir une grille de dimensions $n \times n$ avec les n^2 premiers nombres entiers de façon à ce que la somme des cases de chaque ligne, colonne ou diagonale soit égale à n^2. Brièvement, les variables du problèmes sont bien sûr les n^2 cases de la grille, de domaine initial $\{1, \ldots, n^2\}$, soumises à une contrainte `alldifferent` pour qu'elles prennent toutes des valeurs deux à deux distinctes ; les contraintes restantes sont des contraintes arithmétiques matérialisant mathématiquement le fait que la somme des cases de chaque ligne, colonne ou dialogue doit être égale à n^2.

N reines

Pour finir, le problème des *N reines* est le casse-tête bien connu qui vise comme sont nom l'indique à placer n reines sur un échiquier de dimensions $n \times n$ de façon à satisfaire la contrainte que deux reines ne peuvent pas être prises l'une par l'autre, i.e. toute paire de reines ne peut pas être sur une seule ligne, une seule colonne ou une seule diagonale. Une modélisation possible consiste à affecter à chaque reine une variable R_i représentant sa ligne, la reine R_i se trouvant dans la $i^{\text{ème}}$ colonne. On ajoute alors trois contraintes `alldifferent` :

- une sur R_i, $\forall i$: deux reines ne peuvent pas être sur la même ligne ;
- une sur $R_i + i$, $\forall i$: deux reines ne peuvent pas être sur la même diagonale NO-SE ;
- une sur $R_i - i$, $\forall i$: deux reines ne peuvent pas être sur la même diagonale NE-SO.

5.4.1.2 Problèmes artificiels

Nous avons également conçu trois autres problèmes, dimensionnables et avec des caractéristiques distinctes. Avec ces problèmes dont nous pouvons faire varier la taille, nous cherchons notamment à jeter un regard plus précis sur l'impact de la taille sur le comportement des méthodes comparées.

Le premier de ces problèmes, *int-frac*, est un problème purement discret doté d'une structure inspirée du problème *fractions* :

$$\begin{cases} \text{alldifferent}(x_1, \ldots, x_n), \\ \dfrac{x_1}{x_2} + \ldots + \dfrac{x_{n-1}}{x_n} = \dfrac{1}{2} + \ldots + \dfrac{n-1}{n}, \\ x_i + 1 \leq x_{i+2},\ 1 \leq i \leq n,\ i \text{ odd}, \\ x_i \in \{1, \ldots, n\},\ 1 \leq i \leq n. \end{cases}$$

Le deuxième problème, *mixed-sin*, est un problème mixte qui fait appel à la fois à des contraintes linéaires et non linéaires :

$$\begin{cases} \text{alldifferent}(p_1, \ldots, p_n), \\ x_i p_i \ + \ \displaystyle\sum_{j \neq i} x_j = i,\ 1 \leq i \leq n, \\ \sin(\pi x_i) = 0,\ 1 \leq i \leq n, \\ p_i \in \{1, \ldots, n\},\ 1 \leq i \leq n, \\ x_i \in [-100, 100],\ 1 \leq i \leq n. \end{cases}$$

Quant au troisième problème, *hard-mixed-sin*, il s'agit d'un problème mixte inspiré du précédent mais dans lequel on a durci les contraintes non linéaires :

$$\begin{cases} \text{alldifferent}(p_1, \ldots, p_n), \\ \sin(x_i p_i) \ + \ \displaystyle\sum_{j \neq i} x_j = i,\ 1 \leq i \leq n, \\ \sin(\pi x_i) = 0,\ 1 \leq i \leq n, \\ p_i \in \{1, \ldots, n\},\ 1 \leq i \leq n, \\ x_i \in [-10, 10],\ 1 \leq i \leq n. \end{cases}$$

5.4.1.3 MINLP issus du monde réel

Enfin, nous avons résolu plusieurs instances des problèmes issus du monde réel et que nous avons déjà décrits dans la Section 5.2.1 :

- trois du problème d'ordonnancement robuste, *scheduling*, avec respectivement 9, 10 et 11 tâches ;
- trois du problème d'ordonnancement de lot économique, *ELSP*, avec respectivement 8, 9 et 10 produits ;
- deux du problème de rechargement de réacteur nucléaire, *nuclear*, la première avec une matrice de trajectoire de taille 3×3 où chaque cycle est divisé en 3 étapes, la seconde avec une matrice de trajectoire de taille 4×3 où chaque cycle est seulement divisé en 2 étapes.

5.4.2 Résultats

Les résultats expérimentaux sont présentés dans les Tables 5.2, 5.3 et 5.4. Toutes les expériences ont été conduites sur un Intel Xeon à 3 GHz avec 16 Go de RAM, en utilisant le solveur *RealPaver 1.0* [GB06] étendu pour l'occasion de façon à pouvoir traiter des contraintes `alldifferent` de chacune des façons que nous avons décrites en détails à la Section 5.3. Les six premières colonnes correspondent aux différentes méthodes ou combinaisons de méthodes que nous avons voulu comparer : `MILP` pour la

formulation MILP de la contrainte ; $\neq$ pour la formulation en utilisant seulement les contraintes de diségalités ; pour ces deux formulations, une colonne étiquetée $+\geq\leq$ (resp. +SUM) indique l'usage supplémentaire d'une relaxation convexe d'enveloppe utilisant les inégalités ad hoc (resp. utilisant l'égalité-somme seule). Les deux dernières colonnes CARD et ALLDIF correspondent naturellement à la formulation de la contrainte utilisant l'opérateur de cardinalité et à la formulation utilisant l'algorithme dédié de filtrage par consistance de borne. Les temps de résolution en secondes sont donnés sur la première ligne, le nombre de branchements sur la seconde et exprimé en milliers. Un point d'interrogation indique une méthode qui n'a pas pu résoudre le problème donné en un temps raisonnable, c'est-à-dire inférieur à 10000 secondes. Un nombre de branchements valant ∅ signifie qu'il est impossible de brancher sur ce type de variables pour le problème considéré.

Pour commencer, comparant les relaxations SUM et $\geq\leq$ on peut s'apercevoir que la première est plus efficace que la seconde dans presque tous les cas et quel que soit le type de problème. À chaque fois, on constate que le nombre de branchements est à peu près le même, montrant que la contrainte de somme sur toutes les variables réalise quasiment la totalité du travail de filtrage. Ce constat est d'ailleurs assez bien reflété par le graphique présenté en Figure 5.1, qui montre bien l'impact qu'a cette contrainte sur le temps de résolution quelles que soient les contraintes qu'on lui rajoute dans la relaxation. Ainsi, l'effort additionnel fourni par les inégalités ad hoc ne génère pas suffisamment de rejets de valeurs pour valoir la peine d'être réalisé dans le cas général, et peut même aller jusqu'à faire accroître significativement les temps de résolution —jusqu'à 200% par rapport au temps standard MILP ou $\neq$. Un contre-exemple existe cependant, quand on résout les instances du problème purement discret *int-frac* en utilisant la formulation MILP : dans ce cas, utiliser les inégalités ad hoc aide à réduire le temps de résolution d'un dixième et le nombre de branchement d'un tiers. Une explication pour ce phénomène peut être qu'il existe des valeurs du domaine qui sont retirées du domaine par les contraintes $\neq$ mais pas par les contraintes MILP, et que

Pb	MILP	MILP +≥≤	MILP +SUM	≠	≠ +≥≤	≠ +SUM	CARD	ALLDIF
sevvoth	16.6 4.7 ∅	10.7 2.1 ∅	9.7 2.1 ∅	14.3 5.5 ∅	10.0 3.0 ∅	8.5 3.0 ∅	5.4 2.1 ∅	5.1 1.7 ∅
fractions	1308.9 590.0 ∅	1074.3 291.1 ∅	917.8 292.6 ∅	366.0 538.1 ∅	193.0 163.0 ∅	125.8 163.7 ∅	20.3 87.4 ∅	15.7 68.6 ∅
m-square	?	?	?	2134.4 1859.8 ∅	5989.7 1856.0 ∅	2227.4 1859.8 ∅	71.9 155.5 ∅	49.4 98.0 ∅
queens-10	?	1536.2 163.0 ∅	?	94.3 49.8 ∅	116.1 27.7 ∅	59.1 27.9 ∅	10.1 18.2 ∅	5.8 8.0 ∅
queens-11	?	?	?	675.9 310.1 ∅	735.7 142.2 ∅	362.5 143.1 ∅	52.9 91.4 ∅	24.9 36.9 ∅
queens-12	?	?	?	5200.1 2039.2 ∅	4805.6 782.6 ∅	2289.1 785.3 ∅	260.3 377.9 ∅	133.5 187.7 ∅

Table 5.2 – Temps de résolution en secondes (première ligne) et nombre de branchements sur variable entière/réelle en milliers (seconde ligne).

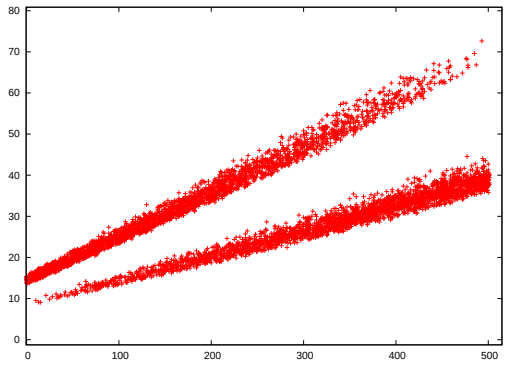


Figure 5.1 – Temps de résolution en fonction du nombre de contraintes d'une relaxation dont les contraintes ont été tirées aléatoirement, pour le problème *Sevvoth*. Le groupe des points correspondant aux meilleurs temps de résolution quelque soit la taille correspond uniquement à des relaxations utilisant la contrainte de somme sur toutes les variables.

ces valeurs peuvent aussi être retirées par les inégalités mais pas par la contrainte d'égalité sur toutes les variables. En effet, pour le problème considéré on n'observe pas ce phénomène quand on utilise la formulation $\neq$ pour laquelle les valeurs filtrées par les inégalités semblent être déjà filtrées par les diségalités, rendant les inéquations inutiles.

Concernant à présent les modélisations spécialisées `CARD` et `ALLDIF`, dans le cas particulier des problèmes purement discrets on s'aperçoit sans surprise qu'il y a une amélioration des temps de résolution d'un facteur allant jusqu'à 10 (resp. 100) provenant de l'utilisation de ces formulations et des algorithmes de filtrage dédiés en comparaison avec la formulation $\neq$ (resp. `MILP`) —cf Table 5.2. De plus, dans ce cas on s'aperçoit qu'il y a seulement un facteur inférieur à 2 entre `CARD` et `ALLDIF` en termes de temps de résolution, tandis que dans le cas plus général des problèmes mixtes —cf Tables 5.3 et 5.4— et plus particulièrement en ce qui concerne les problèmes issus du monde réel, il n'y a aucune différence significative entre `CARD` et `ALLDIF`. Une première raison à cela peut être trouvée dans le fait qu'une grosse partie du travail de filtrage a été réalisée par le processus d'exploration, au moyen des branchements, et non par les algorithmes de filtrage eux-mêmes, signifiant finalement que plus le filtrage est simple et plus la résolution est rapide : on s'aperçoit en effet qu'il y a une différence à peine significative vis-à-vis de la formulation $\neq$ pour les problèmes issus du monde réel. Mais il apparaît également dans ce dernier cas que 90 % du travail de résolution est passé à traiter la partie continue/mixte des modèles. En effet, le rapport entre les nombres de branchements sur variables entières et sur variables réelles nous prouve qu'un dixième seulement du temps de résolution est consacré à l'instanciation des variables entières, d'autant plus que la stratégie de branchement que nous avons utilisée consiste à toujours choisir d'abord les variables entières. Ceci explique également pourquoi on n'observe pas ici une différence plus important avec la formulation `MILP`, nettement moins performante pourtant sur les problèmes discrets ou les problèmes mixtes artificiels, les seconds ayant une partie continue d'un volume bien inférieur à celle des problèmes issus du monde réel.

Ainsi, nos expériences montrent que la formulation `CARD` est une bonne approche pour étendre un solveur fonctionnant sur les intervalles de façon à gérer plus efficacement les problèmes mixtes faisant usage de la contrainte `alldifferent` :

- le concept de cardinalité d'ensemble est une notion mathématique très utilisée, e.g. il y a déjà un mot-clé `card` dans le langage de modélisation *AMPL* ;

- l'algorithme de filtrage que nous avons implémenté est vraiment très simple à mettre en oeuvre ;
- cet algorithme peut drastiquement réduire les temps de résolution face à une modélisation classique `MILP` ou même $\neq$, même dans le cas particulier de problèmes purement discrets ;
- il est très compétitif par rapport à l'algorithme state-of-the-art de filtrage de la contrainte `alldifferent`, spécialement quand il s'agit de problèmes mixtes issus du monde réels et dotés d'une partie continue/mixte très importante.

D'autre part, nos expériences montrent clairement une nouvelle utilisation possible pour la relaxation convexe d'enveloppe, qui non seulement se comporte aussi bien que l'utilisation standard dans le cas général des problèmes mixtes non linéaires, mais qui en plus est très significativement meilleure en termes de performances dans certains cas. Trouver quelles inégalités sont les plus à propos parmi toutes celles que contient l'ensemble de contraintes de la relaxation, de taille exponentielle en nombre de variables, ne semble pas valoir l'effort au vu du gain de performance. Utiliser la contrainte de somme sur toutes les variables est suffisant pour atteindre bien plus rapidement une puissance de filtrage au moins équivalente dans presque la totalité des cas.

Pb	MILP	MILP +$\geq\leq$	MILP +SUM	$\neq$	$\neq$ +$\geq\leq$	$\neq$ +SUM	CARD	ALLDIF
int-frac-12	817.8 394.6 ∅	222.2 80.9 ∅	252.6 115.0 ∅	33.2 51.3 ∅	16.8 14.7 ∅	9.4 14.7 ∅	1.7 8.7 ∅	1.0 6.0 ∅
int-frac-14	?	6989.5 1824.4 ∅	7878.3 2771.4 ∅	996.0 1160.2 ∅	416.0 263.3 ∅	225.3 263.5 ∅	29.9 149.1 ∅	19.1 102.0 ∅
int-frac-16	?	?	?	?	?	6189.1 5607.6 ∅	725.7 3024.5 ∅	441.8 2082.7 ∅
mixed-sin-5	158.7 0.1 271.2	167.8 0.1 271.2	162.0 0.1 271.2	9.3 7.9 11.3	8.9 6.9 9.3	8.1 6.9 9.3	6.1 6.3 8.4	5.8 5.3 8.4
mixed-sin-6	6209.6 0.7 8143.2	6612.4 0.7 8143.5	6320.8 0.7 8143.5	197.0 123.0 184.4	148.1 83.7 118.7	138.9 83.7 118.7	99.5 72.6 105.0	96.0 59.1 105.1
mixed-sin-7	?	?	?	?	8866.2 3617.3 6638.9	8085.6 3621.0 6639.3	5530.5 3108.1 5806.6	5290.3 2558.0 5792.3
hard-mixed-sin-4	4.5 0.0^{+} 9.7	4.5 0.0^{+} 9.7	4.3 0.0^{+} 9.7	0.9 0.3 1.6	0.8 0.2 1.4	0.9 0.2 1.4	0.8 0.2 1.4	0.8 0.1 1.4
hard-mixed-sin-5	542.3 0.1 956.5	649.3 0.1 956.5	554.2 0.1 956.5	90.2 25.1 126.3	91.8 12.1 108.8	79.6 12.1 108.8	69.3 10.3 106.8	67.9 8.3 106.7

Table 5.3 – Temps de résolution en secondes (première ligne) et nombre de branchements sur variable entière/réelle en milliers (seconde ligne).

Pb	MILP	MILP +≥≤	MILP +SUM	≠	≠ +≥≤	≠ +SUM	CARD	ALLDIF
nuclear-A	74.3 $\frac{0.0^+}{6.4}$	74.0 $\frac{0.0^+}{6.4}$	74.3 $\frac{0.0^+}{6.4}$	75.6 $\frac{0.0^+}{6.4}$	75.3 $\frac{0.0^+}{6.4}$	73.5 $\frac{0.0^+}{6.4}$	73.2 $\frac{0.0^+}{6.4}$	73.2 $\frac{0.0^+}{6.4}$
nuclear-B	624.6 $\frac{0.1}{12.6}$	628.3 $\frac{0.1}{12.6}$	633.1 $\frac{0.1}{12.6}$	626.7 $\frac{0.1}{12.6}$	631.8 $\frac{0.1}{12.6}$	631.4 $\frac{0.1}{12.6}$	626.6 $\frac{0.1}{12.6}$	624.9 $\frac{0.1}{12.6}$
schedu-9	850.9 $\frac{9.1}{160.5}$	1230.6 $\frac{0.0^+}{261.7}$	586.8 $\frac{8.9}{108.8}$	9.9 $\frac{0.0^+}{2.0}$	9.8 $\frac{0.0^+}{2.0}$	10.1 $\frac{0.0^+}{2.0}$	9.3 $\frac{0.0^+}{2.0}$	9.2 $\frac{0.0^+}{2.0}$
schedu-10	1444.9 $\frac{63.6}{165.4}$	2545.5 $\frac{0.0^+}{377.7}$	1136.6 $\frac{63.0}{120.7}$	43.1 $\frac{0.0^+}{6.6}$	42.9 $\frac{0.0^+}{6.6}$	42.9 $\frac{0.0^+}{6.6}$	40.3 $\frac{0.0^+}{6.6}$	40.1 $\frac{0.0^+}{6.6}$
schedu-11	?	?	?	231.7 $\frac{0.0^+}{25.5}$	227.2 $\frac{0.0^+}{25.5}$	229.6 $\frac{0.0^+}{25.5}$	215.9 $\frac{0.0^+}{25.5}$	213.9 $\frac{0.0^+}{25.5}$
ELSP-8	210.9 $\frac{0.0^+}{45.5}$	212.1 $\frac{0.0^+}{45.5}$	210.1 $\frac{0.0^+}{45.5}$	9.3 $\frac{0.0^+}{2.0}$	9.6 $\frac{0.0^+}{2.0}$	9.5 $\frac{0.0^+}{2.0}$	9.0 $\frac{0.0^+}{2.0}$	9.0 $\frac{0.0^+}{2.0}$
ELSP-9	164.8 $\frac{0.0^+}{25.5}$	163.8 $\frac{0.0^+}{25.5}$	160.7 $\frac{0.0^+}{25.5}$	75.0 $\frac{0.1}{11.3}$	75.4 $\frac{0.1}{11.3}$	75.7 $\frac{0.1}{11.3}$	70.7 $\frac{0.1}{11.3}$	71.4 $\frac{0.1}{11.3}$
ELSP-10	1057.6 $\frac{0.1}{126.0}$	1066.9 $\frac{0.1}{126.0}$	1048.3 $\frac{0.1}{126.0}$	1240.6 $\frac{0.1}{143.4}$	1223.4 $\frac{0.1}{143.4}$	1242.6 $\frac{0.1}{143.4}$	1171.6 $\frac{0.1}{143.4}$	1167.4 $\frac{0.1}{143.4}$

Table 5.4 – Temps de résolution en secondes (première ligne) et nombre de branchements sur variable entière/réelle en milliers (seconde ligne).

5.5 Conclusion

Dans ce chapitre, nous avons présenté une étude expérimentale à propos de la résolution dans un solveur continu de problèmes mixtes utilisant la contrainte discrète `alldifferent` . Nous avons d'abord montré comment faire usage de cette contrainte pour modéliser une certaine catégorie de problème MINLP issus du monde réel et qu'on peut trouver dans la littérature. Nous avons aussi passé en revue les principales façons dont on peut traiter une contrainte `alldifferent` dans un solveur continu, basé sur la notion d'intervalle, en reformulant la contrainte ou bien en implémentant des algorithmes de filtrage dédiés. Finalement, nous avons réalisé des expériences sur différents types de problèmes de façon à étudier l'impact du choix de la méthode de traitement sur la résolution proprement dite. À travers ces travaux, nos principales contributions tiennent en deux points :

- Nous avons montré qu'un algorithme de filtrage dédié mais très simple à implémenter, comme celui que nous avons réalisé pour la contrainte CARD, était très compétitif par rapport au très optimisé algorithme state-of-the-art de filtrage en ce qui concerne la résolution de problèmes mixtes très contraints issus du monde réel, tout en restant relativement compétitif dans le cas particulier de problèmes purement discret, ce qui n'est pas du tout le cas des formulations ≠ et surtout MILP, cette dernière étant pourtant la formulation d'usage ;

- Nous avons découvert que la relaxation convexe d'enveloppe telle qu'en usage actuellement pouvait être dépassée en termes de performances dans notre solveur à intervalles par un ensemble de contraintes beaucoup plus facile à construire et à utiliser, et qui pouvait permettre une résolution jusqu'à deux fois plus rapide pour résoudre certaines instances des problèmes mixtes issus du monde réel.

De futurs travaux pourraient permettre de s'interroger quant à la généralisation de nos résultats à d'autres problèmes mixtes utilisant la contrainte `alldifferent`. En outre, il pourrait être intéressant d'étudier plus précisément l'impact de la constriction du problème sur le comportement de chaque traitement : soit en trouvant des problèmes mixtes issus du monde réel et ayant un rapport plus équilibré entre le nombre de contraintes discrètes et le nombre de contraintes réelles, soit même en créant un générateur d'instances dans lequel on pourrait faire évoluer ce rapport. Enfin, rappelons qu'à l'heure actuelle aucun solveur de la platforme *AMPL* n'est en mesure de traiter le mot-clé `card` qui pourtant est une primitive de ce langage. Au vu de nos résultats, il semble en effet prometteur d'y implémenter une méthode de filtrage associée qui pourrait permettre de modéliser simplement et efficacement une contrainte `alldifferent`.

VI

SPÉCIALISATION AUX DOMAINES D'ENTIERS DU FILTRAGE BASÉ SUR L'ARITHMÉTIQUE DES INTERVALLES

La perfection accomplie semble incomplète,
Mais elle sert sans s'user ;
La grande plénitude paraît vide,
Mais elle donne sans s'épuiser.

— LAO-TSEU, De la Voie et de sa vertu, XXXXV.

Ce chapitre est dédié à la spécialisation aux contraintes mixtes des techniques de filtrage de domaines basées sur l'arithmétique des intervalles. Après la présentation des travaux de APT et ZOETEWEIJ à propos de l'utilisation d'une arithmétique des intervalles d'entiers pour la résolution de problèmes purement discrets, nous montrons comment le cadre classique de résolution dédié aux problèmes réels peut être modifié pour mieux prendre en compte la notion d'intégralité des variables discrètes. Nous donnons ensuite quelques précisions à propos de l'implémentation de ces modifications dans un solveur à intervalles, puis nous réalisons des expériences pour en étudier le comportement en pratique.

6.1 Travaux en rapport

Nous commençons ce chapitre par présenter les travaux de APT et ZOETEWEIJ, parus d'abord dans [AZ03], puis dans [AZ07]. Ces travaux ont pour objet la propagation de contraintes arithmétiques discrètes, i.e. des contraintes arithmétiques ne faisant intervenir que des variables entières. Dans ce but, ils introduisent une arithmétique des intervalles d'entiers, dédiées au filtrage de contraintes arithmétiques discrètes uniquement. En particulier, ils proposent une définition forte de la division illustrée par l'exemple 6.4 et la figure 6.1. Mais un algorithme mettant en oeuvre cette division s'avèrera trop coûteux pour être implémenté et testé en pratique par eux.

Exemple 6.4. [AZ07] *Soient x et y deux variables entières de domaines respectifs $d_x = [155, 161]$ et $d_y = [9, 11]$.*

- *la division dite* faible *de x par y est égale à $[\lceil \min(A) \rceil, \lfloor \max(A) \rfloor] = [15, 17]$ avec $A = \{155/9, 155/11, 161/9, 161/11\}$;*
- *en revanche, la division dite* forte *de x par y est égale à $[16, 16]$: il n'existe aucune autre valeur z entière pour laquelle on peut trouver une valeur de x et de y satisfaisant la contrainte $z = x/y$.*

Après avoir introduit plusieurs schémas de décomposition des contraintes en contraintes plus petites, ils réalisent des expériences pour comparer ces différentes approches de propagation de contraintes arithmétiques discrètes qu'ils proposent. Pour APT et ZOETEWEIJ, les conclusions principales de ces travaux sont ainsi les suivantes :

- l'implémentation de l'élévation à la puissance au moyen de multiplications donne de plus faibles réductions de domaines, si bien qu'il est avantageux que la contrainte $x = y^n$ soit considérée comme atomique ;
- l'introduction de variables auxiliaires est bénéfique car d'une part cela renforce la propagation, d'autre part cela évite de réévaluer des sous-expressions dont les domaines des variables n'ont pas changé ;
- les résultats expérimentaux obtenus montrent que les approches de décomposition partielle et totale sont plus efficaces que l'approche directe en terme de nombre d'opérations arithmétiques effectuées, à condition de séquencer correctement l'ordre de traitement des contraintes partielles.

Les travaux présentés dans ce chapitre prolongent ceux de APT et ZOETEWEIJ dans la direction d'une meilleure connaissance des avantages en pratique de l'arithmétique des intervalles d'entiers par rapport à l'arithmétique des intervalles de réels. En particulier, les auteurs n'ont pas été intéressés par une compa-

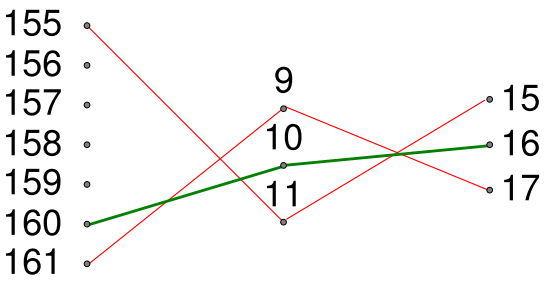


Figure 6.1 – Comparaison entre division faible (rouge) et forte (vert)

raison du comportement ni en théorie ni en pratique des méthodes de résolution basées sur l'arithmétique des intervalles réels et celles basées sur l'arithmétique des intervalles d'entiers. D'autre part, nous élargissons le spectre des opérations étendues aux intervalles d'entiers et enfin, notre approche n'est pas limitée aux problèmes purement discrets : nous nous intéressons aux problèmes mixtes.

6.2 Opérations arithmétiques sur intervalles d'entiers

Ainsi, l'arithmétique des intervalles peut être améliorée en prenant en compte certaines caractéristiques des nombres entiers. Il nous est alors possible de donner de nouvelles définitions à certaines opérations arithmétiques qui vont alors permettre d'effectuer des calculs plus précis et donc d'obtenir des intervalles moins large. Ceci aura pour conséquence d'accélérer drastiquement le processus de résolution grâce à une diminution du nombre nécessaire de calculs sur les intervalles.

6.2.1 Nouvelles définitions

Nous commençons par redéfinir les opérations de logarithme et d'inverse sur les intervalles en prenant en compte certaines intéressantes propriétés sur les entiers de ces fonctions.

6.2.1.1 Logarithme

Soit x une variable entière. Supposons que $d_x = [a, b]$ contient strictement 0, i.e. $a < 0 < b$. Comme le montre la Figure 6.2, il n'existe aucun nombre réel inférieur à 0 qui puisse être obtenu par $log(x)$ quand x est un entier. En effet, les seuls nombres dont le logarithme est un nombre réel inférieur à 0 sont des nombres réels compris entre 0 et 1. Ainsi, effectuer l'opération de logarithme sur des entiers n'offre aucune possibilité d'obtenir un nombre inférieur à 0. Ce qui nous donne l'encadrement suivant :

$$\log(x) \in [0, \log(b)] \tag{6.1}$$

Comparons cette nouvelle définition pour l'opération de logarithme sur les entiers avec la définition originale de cette opération telle qu'utilisée sur les réels. Quand la contrainte d'intégralité du domaine

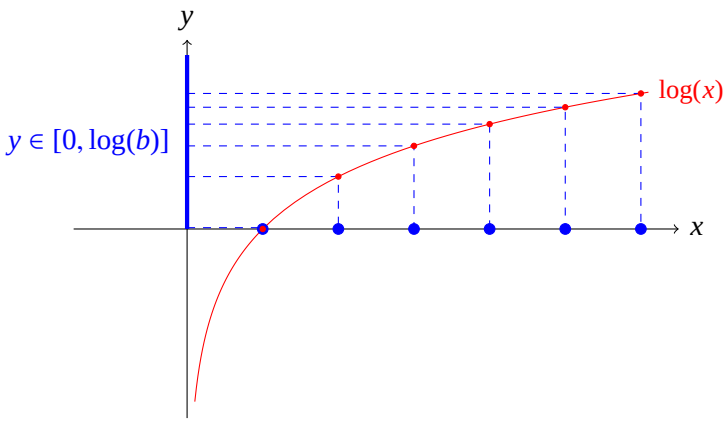


Figure 6.2 – $y = \log(x)$, $x \in \{a, \ldots, b\}$ avec $a < 0 < b$

est relâchée et que le calcul est effectué sur un intervalle de réels, c'est la définition suivante qui nous donne le résultat de l'encadrement :

$$\log(x) \in [-\infty, \log(b)] \tag{6.2}$$

Ainsi, puisque la fonction $log(x)$, qui tend vers $-\infty$ en 0, ne peut atteindre aucune valeur inférieure à 0 quand x est un entier, on peut redéfinir l'opération correspondante sur les intervalles quand on traite une variable dont le domaine est celui des entiers. Utiliser la définition (6.1) nous permettra dans ce cas d'effectuer des évaluations plus précises des expressions arithmétiques, et cela sans perdre de solution potentielle, puisque la contrainte $y = log(x)$ avec $x \in \mathbb{N}$ n'admet aucune solution quand $d_y \cap [0, +\infty] = \varnothing$.

6.2.1.2 Inverse

À présent, soit x une variable entière et supposons à nouveau que $d_x = [a, b]$ contienne strictement 0, i.e. $a < 0 < b$. D'une façon similaire à ce que nous avons observé pour le logarithme, on s'aperçoit qu'il n'existe aucun nombre réel inférieur à -1 ou supérieur à 1 qui puisse être obtenu par $1/x$ quand x est un entier. En effet, les seuls nombres dont l'inverse est un nombre réel inférieur à -1 ou supérieur à 1 sont des nombres réels compris entre -1 et 1 (voir Figure 6.3). Aussi, effectuer l'opération d'inverse sur des entiers n'a aucune chance de retourner un nombre inférieur à -1 ou supérieur à 1. En conséquence, on a l'encadrement suivant :

$$1/x \in [-1, 1/a] \cup [1/b, 1] \tag{6.3}$$

Comme précédemment pour le logarithme, comparons cette nouvelle définition de l'opération d'inverse sur les entiers avec la définition originale de cette opération qu'on utilise sur les réels. Lorsque qu'on relâche la contrainte d'intégralité et qu'on effectue le calcul sur un intervalle de réels, le résultat de l'encadrement nous est donné par la définition suivante :

$$1/x \in [-\infty, 1/a] \cup [1/b, +\infty] \tag{6.4}$$

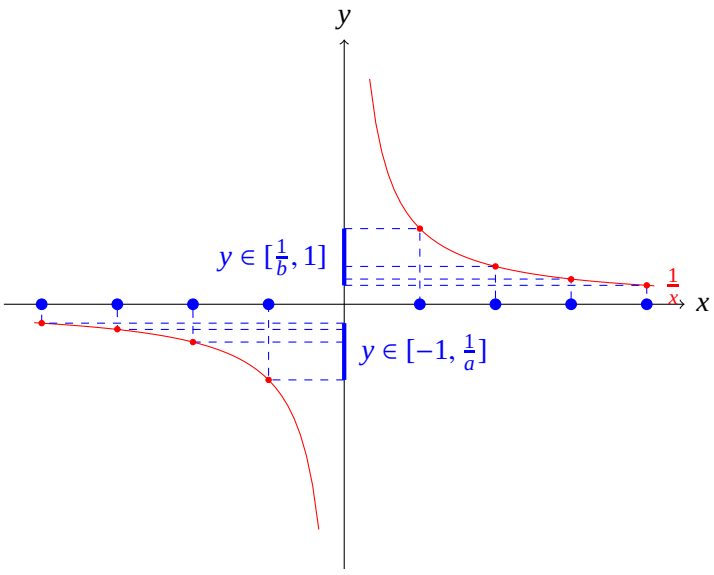


Figure 6.3 – $y = \frac{1}{x}$, $x \in \{a, \ldots, b\}$ avec $a < 0 < b$.

Il est clair que la définition (6.3) permet, en resserrant les bornes des valeurs possibles, d'obtenir un encadrement plus précis que la définition (6.4). En fait, il n'y a aucun nombre réel dans $[-\infty, -1) \cup (1, +\infty]$ qui puisse être le résultat de $1/x$ quand x est un entier car la fonction $1/x$, qui tend vers $-\infty$ en 0^- et vers $+\infty$ en 0^+, ne peut atteindre aucune valeur inférieure à -1 ou supérieure à 1 quand x est un entier. Aussi, on peut redéfinir l'opération correspondante sur les intervalles quand une traite une variable dont le domaine est celui des entiers. L'utilisation de la définition (6.3) va donc permettre d'effectuer le cas échéant des évaluations plus fines des expressions arithmétiques. Aucune solution potentielle ne sera perdue puisque la contrainte $y = 1/x$ avec $x \in \mathbb{N}$ n'admet aucune solution quand $d_y \cap ([-1, 1/a] \cup [1/b, 1]) = \varnothing$.

6.2.2 Application

Notre but ici n'est pas de présenter une arithmétique des intervalles complètement étendue aux problèmes mixtes. Cependant, il nous faut remarquer que la division est particulièrement utile dans le contexte de la satisfaction de contraintes arithmétiques. En outre, elle est fortement mise à contribution au cours de la résolution de problèmes polynomiaux. Comme nous allons pouvoir le constater d'ici la fin de cette section, l'utilisation d'une définition plus précise du calcul sur les intervalles va permettre d'améliorer le filtrage d'une manière très significative.

Soit $x \in [-10, 10]$ une variable entière et soit $y \in [-10, 10]$ une variable réelle. Considérons le système suivant :

$$\begin{cases} x \cdot y &= 1 \\ x + y &= 2 \end{cases}$$

Utilisation de la définition originale

En utilisant le calcul donné par la définition originale (6.4) de l'opération sur les réels, la première équation reste inutile et ne permet pas de réduire les domaines de x et y. En effet, en ce qui nous concerne ici, $x \cdot y = 1$ implique $x = 1/y$ et $y = 1/x$. D'où le calcul suivant, identique pour d_x et d_y :

$$\begin{array}{lcl} d_y & := & \square(d_y \cap (1/d_x)) \\ d_y & := & \square([-10, 10] \cap ([-\infty, -1/10] \cup [1/10, +\infty])) \\ d_y & := & \square([-10, -1/10] \cup [1/10, 10]) \\ d_y & := & [-10, 10] \end{array}$$

En utilisant la définition originale, il est donc impossible de réduire le domaine de la variable y au moyen de la première contrainte. En revanche, la deuxième contrainte permet d'obtenir la réduction suivante, identique pour d_x et d_y :

$$\begin{array}{lcl} d_x & := & \square(d_x \cap (2 - d_y)) \\ d_x & := & \square([-10, 10] \cap [-8, 12]) \\ d_x & := & [-8, 10] \end{array}$$

Mais cette réduction du domaine de la variable x est trop faible pour avoir une incidence en retour sur le domaine de y en utilisant la définition (6.4). Les domaines obtenus en fin de propagation restent donc $d_x = d_y = [-8, 10]$.

Utilisation de la nouvelle définition

Considérons à présent la nouvelle définition. On constate que la première contrainte permet cette fois de réduire le domaine de y :

$$\begin{array}{lcl} d_y & := & \square(d_y \cap (1/d_x)) \\ d_y & := & \square([-10,10] \cap ([-1,-1/10] \cup [1/10,1])) \\ d_y & := & \square([-1,-1/10] \cup [1/10,1]) \\ d_y & := & [-1,1] \end{array}$$

En conséquence, l’utilisation de la deuxième contrainte entraîne une réduction bien plus importante cette fois-ci que dans le cas précédent :

$$\begin{array}{lcl} d_x & := & \square(d_x \cap (2 - d_y)) \\ d_x & := & \square([-10,10] \cap [1,3]) \\ d_x & := & [1,3] \end{array}$$

Grâce à l’utilisation de la nouvelle définition, la réduction est maintenant suffisamment pour pouvoir être propagée en ayant des répercussions. Réinjectant la valeur obtenue dans la première contrainte, il vient en effet :

$$d_y := \square(d_y \cap (1/d_x)) := [\tfrac{1}{3}, 1]$$

Enfin, en propageant à nouveau dans la deuxième contrainte[1] puis dans la première, on obtient finalement les réductions suivantes :

$$\begin{array}{lcl} d_x & := & \square(d_x \cap (2 - d_y)) := 1 \\ d_y & := & \square(d_y \cap (1/d_x)) := 1 \end{array}$$

De par l’utilisation de la nouvelle définition, on constate que les domaines obtenus en fin de propagation sont cette fois l’unique solution du système. Ainsi, la prise en compte du caractère entier des variables pendant l’évaluation aux intervalles nous permet d’obtenir un filtrage bien plus fort.

6.3 Gestion des contraintes d’intégralité

Dans cette section, nous commençons par présenter la façon communément pratiquée de prendre en compte les contraintes d’intégralité dans un solveur à intervalles. Ensuite, nous introduisons une nouvelle façon de prendre en compte ces contraintes, plus tôt au cours du processus de propagation.

6.3.1 Troncature aux bornes entières

Certains solveurs à intervalles sont capables de prendre en compte de façon correcte le fait qu’une variable soit entière et sont ainsi capables d’appliquer des contraintes d’intégralité, i.e. des contraintes qui stipulent que certaines variables ne peuvent prendre qu’une valeur entière, au cours de la propagation des réductions de domaines grâce aux contraintes du modèle. L’idée derrière le filtrage mis en oeuvre pour respecter l’intégralité d’une variable est la suivante : étant donnée une contrainte C portant sur

[1] Le domaine de d_x calculé par cette réduction est d’abord $[1, \frac{5}{3}]$ qui est alors tronqué en $[1,1]$ parce que x est une variable entière. Ce mécanisme de troncature sera l’objet de la section suivante.

au moins une variable entière x, de domaine $[a, b]$ enveloppe ou pavé-consistant, il n'est pas possible de trouver une valeur $v \in [a, \lceil a \rceil]$ (resp. $[\lfloor b \rfloor, b]$) telle que $C(\ldots, v, \ldots)$ est satisfaite. Ainsi, une façon de faire couramment mise en oeuvre est d'implémenter une procédure de filtrage dédiée à la contrainte d'intégralité, qui a pour but de tronquer un domaine à ses bornes entières de la manière suivante :

$$d_x \quad := \quad \square(\text{int}(d_x))$$

où *int* est l'opérateur qui tronque à ses bornes entières un intervalle à bornes flottantes, i.e.

$$\text{int}([a, b]) \quad := \quad \begin{cases} [\lceil a \rceil, \lfloor b \rfloor] & \text{si } a \leq b + 1 \\ \varnothing & \text{sinon} \end{cases}$$

En outre, cette procédure de filtrage va être appliquée régulièrement au cours de la propagation des réductions de domaines, permettant ainsi de maintenir les domaines des variables entières à des intervalles à bornes entières. De cette façon, un solveur à intervalles va pouvoir en pratique satisfaire les contraintes d'intégralité des variables qui le nécessitent.

Soit x une variable entière et $x = f(y)$ une contrainte utilisée pour le filtrage. L'implémentation classique de l'opération de troncature, destinée à la satisfaction des contraintes d'intégralité que nous venons de décrire, peut s'exprimer formellement de la façon suivante :

$$\begin{array}{lll} d_x & := & \square(d_x \cap F(d_y)) \\ d_x & := & \square(\text{int}(d_x)) \end{array} \tag{6.5}$$

où le traitement de la contrainte d'intégralité est réalisé après le filtrage du domaine. Mais nous allons voir à présent qu'il est en fait possible d'effectuer la troncature des domaines de variables entières directement à l'intérieur des opérateurs de filtrage associées aux contraintes arithmétiques elles-mêmes, obtenant de cette manière une amélioration radicale des opérations de filtrage.

6.3.2 Amélioration du mécanisme

Le traitement classique de la contrainte d'intégralité qu'on vient de présenter est ainsi réalisé de la façon suivante au cours de la propagation, en deux temps :

1. Les domaines de toutes les variables sont filtrés par les contraintes du modèle ;
2. Les domaines des variables entières modifiés sont tronqués à leurs bornes entières.

Mais le post-traitement qui permet de satisfaire la contrainte d'intégralité peut aussi être réalisé directement au moment du filtrage utilisant une contrainte arithmétique, réunissant les deux étapes ci-dessus en une seule, de la manière suivante :

$$d_x \quad := \quad \square(\text{int}(d_x \cap F(d_y))) \tag{6.6}$$

où la troncature aux bornes entières est réalisée avant le calcul de l'enveloppe de $d_x \cap F(d_y)$. L'exemple 6.5 va d'ailleurs nous donner un aperçu de combien il peut être intéressant de prendre ainsi en compte la contrainte d'intégralité le plus tôt possible au cours des calculs sur les intervalles.

Exemple 6.5. *Soit la contrainte $y = x^2$ où x est une variable entière avec $D_x = [-2, 2]$ et y une variable réelle avec $D_y = [2, 3]$. Le domaine de x est calculé au moyen de l'opération inverse du carré, appliquée*

à y et à son domaine. Il s'ensuit que x doit appartenir à l'intervalle $[-\sqrt{3}, -\sqrt{2}] \cup [\sqrt{2}, \sqrt{3}]$. *Appliquant la méthode (6.5), on obtient :*

$$\begin{array}{lcl} d_x & := & \Box(\text{int}(\Box([-\sqrt{3}, -\sqrt{2}] \cup [\sqrt{2}\ \sqrt{3}])) \\ d_x & := & \Box(\text{int}([-1.732\ldots, 1.732\ldots]) \\ d_x & := & \Box([-1, 1])) \\ d_x & := & [-1, 1] \end{array}$$

Cependant, si à la place on applique plutôt la méthode (6.6), on obtient :

$$\begin{array}{lcl} d_x & := & \Box(\text{int}([-\sqrt{3}, -\sqrt{2}] \cup [\sqrt{2}, \sqrt{3}]) \\ d_x & := & \Box(\varnothing) \\ d_x & := & \varnothing \end{array}$$

L'exemple précédent met en évidence le gain potentiel qu'on peut attendre de cette nouvelle manière de prendre en compte les contraintes d'intégralité des variables. Posons la proposition suivante :

Proposition 6.1. *Soient* d_x^1 *et* d_x^2 *les domaines obtenus respectivement en utilisant les méthodes de calcul (6.5) et (6.6). Alors on a* $d_x^2 \subseteq d_x^1$.

Preuve. Il est immédiat de s'apercevoir que

$$d_x \cap F(d_y) \subseteq \Box(d_x \cap F(d_y))$$

d'après est une propriété élémentaire de l'opération d'enveloppe. Il vient ensuite naturellement que

$$\text{int}(d_x \cap F(d_y)) \subseteq \text{int}(\Box(d_x \cap F(d_y)))$$

puisqu'on sait que $A \subseteq B \Rightarrow \text{int}(A) \subseteq \text{int}(B)$. Finalement on obtient

$$\Box(\text{int}(d_x \cap F(d_y))) \subseteq \Box(\text{int}(\Box(d_x \cap F(d_y))))$$

en utilisant de nouveau les propriétés de l'opération d'enveloppe. □

Ainsi, remplacer l'implémentation classique du filtrage utilisant la contrainte d'intégralité comme un post-traitement, par une version optimisée du filtrage qui intègre directement cette contrainte, permet d'obtenir des réductions de domaine plus fortes. En conséquence, chaque opération de contraction du domaine d'une variable entière doit être capable de calculer la troncature aux bornes entières de son nouveau domaine avant de retourner l'enveloppe de ce dernier.

6.4 Implémentation

Dans cette section, nous donnons des détails à propos de l'implémentation que nous avons pu réaliser des améliorations du filtrage dédiées aux contraintes arithmétiques mixtes, décrites dans les sections précédentes.

Plate-forme utilisée

Nous avons de nouveau utilisé le solveur *RealPaver* [GB06] pour effectuer l'implémentation des optimisations du filtrage. Ainsi qu'on a pu s'en apercevoir précédemment, il s'agit d'un solveur de contraintes continues, linéaires ou non linéaires, doté d'une API C++ construite autour d'un cœur en C. Il peut ainsi être utilisé comme une application indépendante ou bien encapsulé dans une autre application. Nous rappelons brièvement quelques unes de ses principales caractéristiques :

- les modèles peuvent être exprimés en utilisant un langage de modélisation dédié (voir Fig. 6.1) ;
- les domaines en entrée peuvent être des intervalles ou unions d'intervalles ;
- les variables déclarées peuvent être réelles ou entières ;
- les contraintes arithmétiques linéaires et non linéaires sont autorisées ;
- les autres contraintes à disposition incluent les contraintes en extension, les contraintes conditionnelles et les contraintes par morceaux.

Comme on l'a déjà partiellement vu au Chapitre 3, la représentation des domaines, les structures arborescentes utilisées pour représenter les expressions arithmétiques et les évaluer, ainsi que le parsing des modèles, sont écrits en C ; les opérateurs de réduction de domaines, la propagation de contraintes, les heuristiques de sélection de variable et de découpage de domaine sont implémentés en C++.

Les opérateurs de réduction de domaine implémentés incluent le filtrage par consistance d'enveloppe, de pavé et 3B. Il y aussi de multiples heuristiques de choix de variables : priorité aux variables les plus contraintes, priorité aux domaines entiers les plus fins (resp. réels les plus larges), priorité aux variables de décision, tourniquet, etc. Enfin, les stratégies de découpage de domaine implémentés incluent la bissection et l'énumération.

Le plus gros du travail que nous avons réalisé ici concerne la partie bibliothèque de calcul sur les intervalles du logiciel. C'est là que nous avons modifié les méthodes existantes et rajouté de nouvelles

P . 6.1 : Modèle *RealPaver* pour un problème d'intersection entre un cercle et une parabole.

```
1  problem circle
2
3    num p  := 1.0e-8,
4          D  := 2.75,
5          x0  := 2,
6          y0  := 1,
7          R  := 1.25;
8
9
10   var  x  : real / p ~ [-oo,+oo],
11        y  : real / p ~ [-100,100];
12
13   ctr  (x-x0)^2 + (y-y0)^2 = R^2,
14        x^2 + y^2 = D^2;
15
16   solve *;
17
18 end
```

méthodes pour prendre en compte les nouvelles opérations arithmétiques et la gestion au plus tôt des contraintes d’intégralité.

6.4.1 Nouvelles opérations arithmétiques

Concernant l’implémentation des nouvelles opérations arithmétiques, le travail à effectuer dans *RealPaver* pour les prendre correctement en compte pendant le processus de résolution peut être décomposé en trois parties :

- ajouter les nouvelles opérations dédiées aux intervalles d’entiers dans la bibliothèque de calcul sur les intervalles ;
- modifier la méthode d’évaluation aux intervalles pour les calculs faisant intervenir des opérandes de domaine entier ;
- modifier la méthode de propagation descendante, utilisée par le filtrage par consistance d’enveloppe, pour les calculs faisant intervenir des opérandes de domaine entier.

L’ajout des nouvelles opérations dans la bibliothèque est quasi immédiat étant données les règles de calcul que nous avons données en Section 6.2. Nous avons ainsi implémenté deux nouvelles méthodes effectuant respectivement l’opération de logarithme et la division d’un intervalle d’entiers. En ce qui concerne ce dernier, remarquons qu’il n’y a aucun changement à faire quand il s’agit de traiter des domaines qui ne contiennent pas 0.

Étant donnés une expression arithmétique et un pavé correspondant aux domaines courants des variables, l’évaluation aux intervalles consiste à appliquer les règles de l’arithmétique des intervalles afin d’obtenir un intervalle correspondant à l’encadrement du résultat. Concernant l’implémentation de nos modifications, nous devons ici définir les bonnes méthodes à appeler e.g. c’est là que nous spécifions le fait que pour une variable entière dont on veut effectuer le logarithme, on fait appel à la nouvelle méthode.

Enfin, rappelons que la propagation descendante consiste à inférer de nouveaux domaines pour les variables, étant donnés leur expression arithmétique et un pavé correspondant aux domaines courants des autres variables. Aussi, nous avons dû modifier les appels aux méthodes de projection dans le cas des noeuds ayant pour fils une variable entière —et récursivement les noeuds ayant pour fils un noeud ayant pour fils une variable entière— de façon à faire appel aux bonnes opérations. Notons qu’il faut également modifier les appels aux méthodes de projection concernant l’opérateur de multiplication, puisque $x \cdot y = z$ implique $x = z/y$ et $y = x/z$ comme nous l’avons mis en évidence à la section précédente.

6.4.2 Gestion au plus tôt des contraintes d’intégralité

Concernant à présent la nouvelle stratégie de tronquer au plus tôt à leur bornes entières les domaines calculés, dès la phase de propagation descendante, il faut implémenter un nouvel opérateur capable de retourner l’enveloppe d’un domaine après seulement que celui-ci ait été tronqué à ses bornes entières. L’Algorithme 6.1 montre comment cela est possible dans le cas d’une union triée d’intervalles disjoints :

- on parcourt les intervalles en les tronquant un à un à leurs bornes entières, en commençant par celui de plus petite borne inférieure ;
- on met à jour la borne inférieure de l’enveloppe entière quand on obtient un intervalle non vide ;
- dès qu’on a trouvé une telle borne inférieure, on réitère pour trouver la borne supérieure, en commençant par l’intervalle de plus grande borne supérieure.

A . 6.1 : IntHull(*union of intervals* $U = \{i_1, \ldots, i_n\}$)

```
1  interval h ← ∅;
2  boolean found_inf ← found_sup ← false;
3  pour i ∈ {i_1, ..., i_n} et tant que non found_inf faire
4  |  interval j ← trunc_to_integer_bounds(i);
5  |  si non empty(j) alors
6  |  |  si non found_inf alors
7  |  |  |  inf(h) ← inf(j);
8  |  |  |  found_inf ← true;
9  |  |  fin
10 |  fin
11 fin
12 si found_inf alors
13 |  pour i ∈ {i_n, ..., i_1} et tant que non found_sup faire
14 |  |  interval j ← trunc_to_integer_bounds(i);
15 |  |  si non empty(j) alors
16 |  |  |  si non found_sup alors
17 |  |  |  |  sup(h) ← sup(j);
18 |  |  |  |  found_sup ← true;
19 |  |  |  fin
20 |  |  fin
21 |  fin
22 fin
23 retourner h;
```

- **l. 1** : initialisation de l'intervalle résultat
- **l. 2** : initialisation des booléens signifiant l'existence ou non d'une borne inférieure (resp. supérieure) entière
- **l. 3** : début de la boucle de parcours de chacun des intervalles de l'union
- **l. 4** : troncation à ses bornes entières de l'intervalle courant
- **l. 5-9** : mise-à-jour de la borne inférieure du résultat le cas échéant
- **l. 13** : boucle de parcours de chacun des intervalles de l'union
- **l. 14** : troncation à ses bornes entières de l'intervalle courant
- **l. 15-19** : mise-à-jour de la borne supérieure du résultat le cas échéant

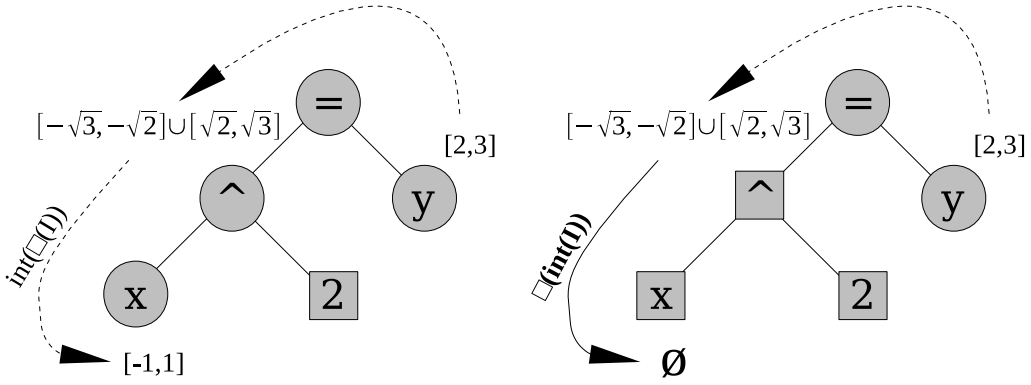


Figure 6.4 – Propagation descendante utilisant la méthode normale (gauche), utilisant la méthode de troncature au plus tôt (droite).

De cette façon, on peut effectivement calculer l'enveloppe du résultat de la division, par exemple, après seulement avoir replongé dans le domaine des entiers le résultat obtenu sur les réels. C'est cet algorithme que nous avons implémenté dans *RealPaver*.

L'étape suivante consiste à remplacer chaque appel à la méthode d'enveloppe standard, après une opération sur des variables entières capable de générer des trous dans les domaines, comme la division ou l'élévation à une puissance paire, par un appel à cette nouvelle méthode. Dans chaque opérateur de projection vers une variable entière capable de créer un trou dans le domaine et de retourner une union d'intervalles —e.g. élévation/racine avec puissance paire ou multiplication/division— il faut remplacer l'appel à la méthode standard par un appel à la nouvelle méthode. Nous avons ainsi défini de nouvelles fonctions de projection, qui sont appelées à la place des anciennes, dans le cas d'une projection vers une variable entière (voir Figure 6.4). De cette façon, on s'assure de ne plus surestimer les domaines qui ne contiennent aucun entier mais dont l'enveloppe en contient.

6.5 Expériences

Les expériences présentées ici ont été réalisées sur un Intel Xeon à 3 GHz avec 16 Go de RAM. Nous avons utilisé la version 1.0 de *RealPaver* ayant été modifiée de la façon que nous avons décrite dans la section précédente. Dans le but de mesurer les apports de nos optimisations du filtrage, nous avons utilisé deux problèmes mixtes dimensionnables que par simplicité nous nommerons problème A et problème B.

Problème A

Soit x un vecteur de variables réelles, de dimension n, et y un vecteur de variables entières, de dimension n également. Supposons que chaque variable ait pour domaine initial $[-10^8, 10^8]$ et considérons le système d'équations suivant :

$$\begin{cases} x_i y_i = 1, & i = 1, \ldots, n \\ \sum_{i=1}^{n} x_i^2 = 2n. \end{cases}$$

Les résultats expérimentaux pour ce problème sont présentés dans la Table 6.1 et les Figures 6.5 et 6.6. Nous avons mesuré le temps de résolution, en secondes, et le nombre de branchements, pour une dimension n du problème variant de 3 à 20. La première colonne correspond à la dimension de l'instance testée. La deuxième colonne montre les résultats obtenus avec l'implémentation standard. La troisième colonne présente quant à elle les résultats obtenus avec la nouvelle implémentation.

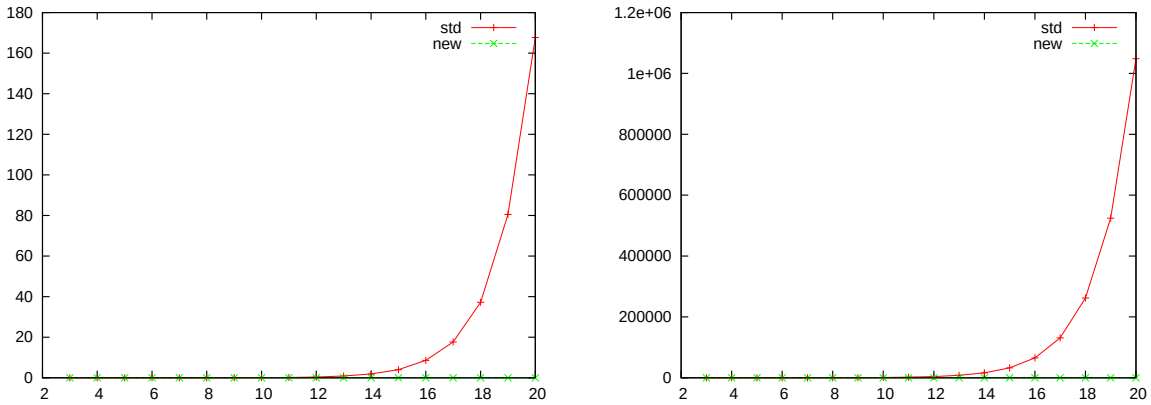


Figure 6.5 – Pour le Problème A, temps standard face à nouveau temps (gauche), nombre de découpages standard face à nouveau nombre de découpages (droite).

Pour ce problème, la nouvelle méthode se comporte mieux que la méthode standard quelle que soit la dimension du problème. De plus, le gain en temps et en nombre de branchements est exponentiel comme le montre la Figure 6.6. En effet, la méthode standard ne permet de réaliser que de faibles réductions de domaines en utilisant la première contrainte, si bien que pour prouver l'inconsistance du problème, la propagation de contraintes doit être combinée avec une technique de découpage de domaine. Cependant, cela requiert un nombre exponentiel de découpages, exactement 2^{n-1}. Mais avec la méthode

n	std	new
3	0.004 (7)	0^+ (0)
4	0.004 (15)	0^+ (0)
5	0.004 (31)	0^+ (0)
6	0.004 (63)	0^+ (0)
7	0.008 (127)	0^+ (0)
8	0.02 (255)	0^+ (0)

n	std	new
9	0.044 (511)	0^+ (0)
10	0.088 (1023)	0^+ (0)
11	0.188 (2047)	0^+ (0)
12	0.408 (4095)	0^+ (0)
13	0.88 (8191)	0^+ (0)
14	1.89 (16k)	0^+ (0)

n	std	new
15	4.024 (33k)	0^+ (0)
16	8.55 (65k)	0^+ (0)
17	17.68 (131k)	0^+ (0)
18	37.21 (262k)	0^+ (0)
19	80.53 (524k)	0^+ (0)
20	167.67 (1049k)	0^+ (0)

Table 6.1 – Temps de résolution et nombre de découpages pour le Problème A.

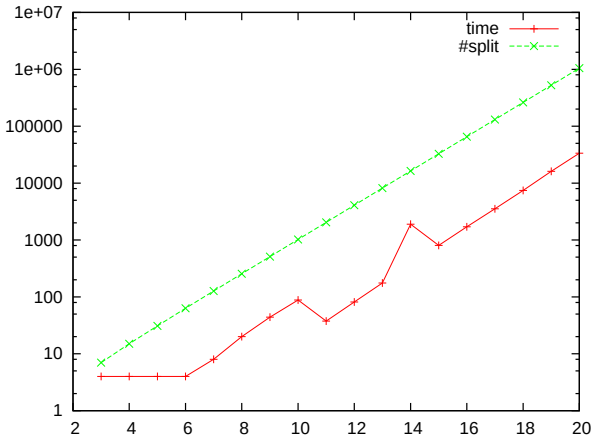


Figure 6.6 – Gains en temps et en nombre de découpages, en fonction du nombre de variables de l’instance résolue du Problème A (échelle logarithmique).

nouvellement implémentée, l’inconsistance peut être découverte en utilisant seulement l’algorithme de propagation de contraintes. L’explication en tient à ce que la nouvelle méthode de calcul de la division permet aux domaines d’être contractés par rapport aux contraintes $x_i y_i = 1$.

Ainsi, au vu de ce premier problème, les techniques que nous présentons dans ce chapitre, dédiée à un filtrage plus efficace des domaines entiers impliqués par des contraintes mixtes, semblent très prometteuses en pratique.

Problème B

Posons à présent le problème B. Soit x un vecteur de variables réelles, de dimension n, et y un vecteur de variables entières, de dimension n également. Supposons que chaque variable a pour domaine inital $[-10^8, 10^8]$ et considérons le système d’équations suivant :

$$\begin{cases} (x_i - 1)y_i &= 1, \quad i = 1, \ldots, n \\ \log(\prod_{i=1}^{n} y_i) &= 1. \end{cases}$$

Les résultats expérimentaux pour ce problème sont présentés dans la Table 6.2 et les Figures 6.7 et 6.8. À nouveau, nous avons mesuré le temps de résolution, en secondes, et le nombre de découpages, pour une dimension n du problème variant de 3 à 20. La première colonne correspond à la dimension de l’instance testée. La deuxième colonne montre les résultats obtenus avec l’implémentation standard. La troisième colonne présente quant à elle les résultats obtenus avec la nouvelle implémentation.

Comme dans le problème précédent, la nouvelle méthode se comporte mieux que la méthode standard quelle que soit la taille du problème. En outre, le gain est cette fois linéaire par rapport à la dimension du problème, en temps comme en nombre de branchements —cf Figure 6.8— i.e. la résolution du problème de dimension n avec la nouvelle méthode est n fois plus rapide et le nombre de branchements est n fois inférieur. Ceci s’explique par le fait que de nombreuses branches de l’arbre de recherche sont coupées après que la propagation de contraintes ait pu réussir à prouver l’inconsistance du pavé courant grâce aux nouvelles méthodes de filtrage que nous avons implémentées spécialement pour les domaines entiers. En

conséquence, ces branches inconsistantes ne sont pas explorées inutilement et la résolution s'en trouve accélérée d'autant.

Ainsi, au vu des résultats mesurés sur ce second problème, les techniques que nous présentons dans ce chapitre apparaissent de nouveau très prometteuses. Non seulement elles étaient déjà intéressantes d'un point de vue théorique, montrant comment l'arithmétique des intervalles peut être améliorée afin de gérer les domaines entiers d'une meilleure façon, avec plus de précision, mais il s'avère également qu'elles sont très efficaces d'un point de vue pratique pour les problèmes considérés, quel que soit leur dimension.

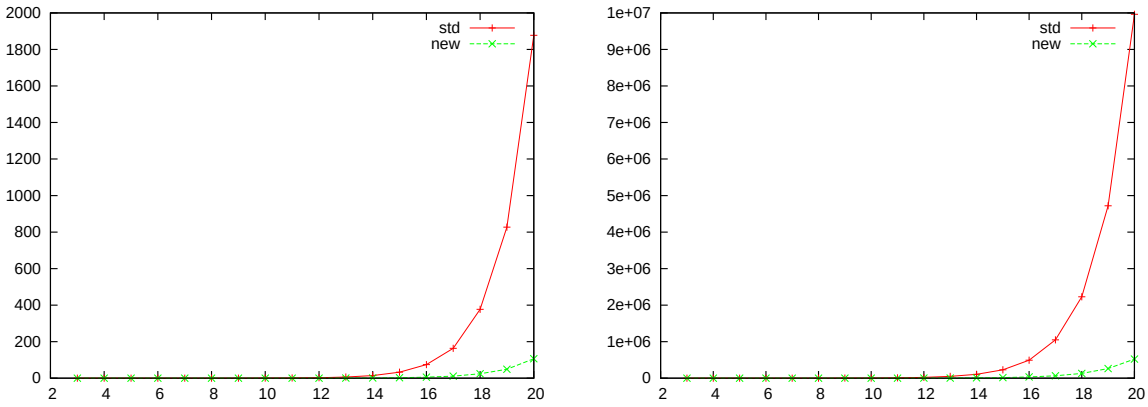


Figure 6.7 – Pour le Problème B, temps standard face à nouveau temps (gauche), nombre de découpages standard face à nouveau nombre de découpages (droite).

n	std	new
3	0 (7)	0 (3)
4	0.004 (23)	0.004 (7)
5	0.004 (63)	0.004 (15)
6	0.012 (159)	0.004 (31)
7	0.032 (383)	0.008 (63)
8	0.084 (895)	0.016 (127)

n	std	new
9	0.19 (2047)	0.028 (255)
10	0.47 (4607)	0.06 (511)
11	1.12 (10k)	0.12 (1023)
12	2.72 (22k)	0.26 (2047)
13	6.22 (49k)	0.56 (4095)
14	14.40 (106k)	1.19 (8191)

n	std	new
15	33.1 (229k)	2.58 (16k)
16	74.02 (491k)	5.32 (33k)
17	163.24 (1049k)	11.09 (65k)
18	377.1 (2228k)	23.92 (131k)
19	827.2 (4719k)	48.71 (262k)
20	1877.4 (9961k)	106.63 (524k)

Table 6.2 – Temps de résolution et nombre de découpages pour le Problème B.

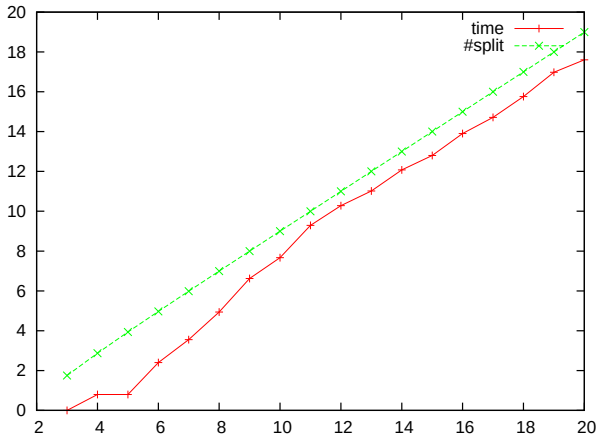


Figure 6.8 – Gains en temps et en nombre de découpages, en fonction du nombre de variables de l’instance résolue du Problème B.

6.6 Conclusion

Dans ce chapitre, nous avons présenté des améliorations pour les techniques de filtrage de domaine basées sur l’arithmétique des intervalles. Ces améliorations dédiées à une meilleure utilisation des contraintes mixtes, i.e. des contraintes qui font intervenir à la fois des variables discrètes et des variables continues. Prolongeant les travaux antérieurs de A et Z à propos d’une arithmétique des intervalles d’entiers, nous avons introduit de nouvelles règles de calcul sur les intervalles de certaines opérations arithmétiques. Celles-ci nous permettent de faire des encadrements plus fins du résultat de l’évaluation aux intervalles d’une expression arithmétique, en prenant en compte le caractère éventuellement entier des variables sur lesquelles portent ces opérations. Nous avons également mis en lumière une nouvelle façon de prendre en compte les contraintes d’intégralité au cours de la propagation de contraintes, plus tôt dans le processus de propagation. Nous avons ensuite montré comment nous avons pu implémenter ces améliorations dans un solveur à intervalles. Enfin, nous avons présenté des expériences dans le but d’observer le comportement de nos améliorations vis-à-vis de la résolution de problèmes mixtes de dimension variable. À travers ces travaux, nos contributions sont ainsi les suivantes :

- Nous donnons de nouvelles définitions pour des opérations typiques de l’arithmétique des intervalles, dans le cas particulier où leurs opérandes sont des variables entières, et montrons que cela permet, à travers un augmentation de la force du filtrage obtenu, d’accélérer de manière très significative la résolution des problèmes ;

- Nous proposons une nouvelle façon de gérer les contraintes d’intégralité des variables pendant le processus de résolution : prendre en compte ces contraintes le plus tôt possible au cours de la propagation, c’est-à-dire non pas comme un post-traitement au filtrage des domaines mais au moment même de celui-ci, permet d’accélérer de manière drastique la résolution des problèmes.

De futurs travaux pourraient être consacrés à mieux comprendre l’impact de chacune des améliorations du filtrage sur la résolution des problèmes, en réalisant des observations plus précises comme compter le nombre d’opérations arithmétiques effectuées, ou activer/désactiver chacune des optimisa-

tions, avant de comparer les résultats obtenus. En outre, il pourrait être intéressant de réaliser une étude plus exhaustive de l'arithmétique des intervalles d'entiers, dans le but de trouver éventuellement d'autres opérations arithmétiques qui pourraient bénéficier d'une telle spécialisation aux intervalles d'entiers, ou bien le cas échéant de prouver que cela n'est pas possible. Pour finir, il faudrait étudier expérimentalement l'impact des améliorations que nous présentons ici dans le cadre d'applications de la programmation par contraintes issues du monde réel.

VII

CONCLUSION ET PERSPECTIVES

Le présent chapitre conclut cette thèse consacrée à la modélisation et la résolution en programmation par contraintes de problèmes de satisfaction de contraintes et d'optimisation mixtes. Nous commencerons cette conclusion par une synthèse de nos travaux, chapitre par chapitre. Puis nous reviendrons sur les perspectives de recherche que nous avons pu mettre en lumière au cours des travaux que nous présentons dans cette thèse.

7.1 Synthèse de nos travaux

Cette section est consacrée à la récapitulation des points importants de chacun des travaux que nous avons présentés dans cette thèse :

- Modélisation et résolution d'un problème de conception en robotique ;
- Collaboration de deux solveurs discret et continu ;
- Utilisation de la contrainte `alldifferent` dans un solveur continu ;
- Spécialisation du filtrage basé sur l'arithmétique des intervalles.

7.1.1 Modélisation et résolution d'un problème de conception en robotique

Ce chapitre a été l'occasion de montrer comment faire usage de la programmation par contraintes pour résoudre efficacement et rigoureusement un problème d'optimisation sous contraintes tel que le problème du calcul de l'erreur de pose maximum d'un système robotique mécanique due au jeu dans ses articulations.

Dans un premier temps, nous avons modélisé le problème comme un CSP continu doté d'une fonction-objectif. Considérant comme idéal le système à modéliser, nous avons pour commencer décrit la position de l'organe effecteur en fonction des caractéristiques du robot. Ensuite, nous avons modélisé la notion de jeu dans les articulations avant de l'intégrer à ce modèle idéal que nous avions commencé par formuler. Nous avons finalement obtenu deux problèmes d'optimisation sous contraintes, l'un correspondant à l'erreur maximum en rotation, l'autre correspondant à l'erreur maximum en translation.

Dans un deuxième temps, nous avons alors montré comment transformer concrètement le schéma classique *branch-and-prune* de résolution tel qu'employé par la programmation par contraintes, en un algorithme de type *branch-and-bound* dédié à la résolution rigoureuse de problèmes d'optimisation sous contraintes. Nous avons mis en lumière une manière de mettre efficacement en oeuvre la gestion rigoureuse du calcul d'un encadrement optimal de la fonction-objectif au cours du processus de résolution classique réalisé en programmation par contraintes.

Finalement, nous avons présenté des résultats expérimentaux, obtenus à partir de plusieurs instances du problème initial. Cela nous a permis de mettre en évidence que notre approche était très compétitive

par rapport à des outils à la pointe des techniques d'aujourd'hui. D'autre part, nous avons pu constater par nous-mêmes une divergence assez significative dans les résultats obtenus par les différentes méthodes que nous comparions. Nous avons suggéré que cette divergence pouvait être mise sur le compte de la non-fiabilité des outils d'optimisation non basés sur l'arithmétique des intervalles, un état de fait déjà mentionné dans la littérature [LMR07].

Ainsi, nous avons eu l'opportunité de faire l'application des techniques de programmation par contraintes pour la modélisation et la résolution d'un problème de conception en robotique pour lequel la PPC n'avait jamais été mise à contribution jusqu'ici. L'intégration de méthodes orientées intervalle pour l'optimisation permet d'acquérir une robustesse des solutions que les méthodes numériques classiques ne permettent pas d'atteindre à cause des erreurs d'arrondis. Nos expériences ont par ailleurs montré qu'en ce qui concerne le problème considéré, ce gain en robustesse s'accompagnait d'un gain en efficacité de résolution, en temps de calcul.

7.1.2 Collaboration de deux solveurs discret et continu

Dans ce chapitre, nous avons présenté nos travaux concernant la réalisation d'une collaboration entre deux solveurs, l'un dédié aux contraintes discrètes, l'autre dédié aux contraintes continues. L'idée directrice sur laquelle nous avons entrepris cette réalisation est très naturelle et peut s'exprimer de la façon suivante : il semble particulièrement prometteur de pouvoir profiter des avantages de chacun des outils dans leur champ de prédilection respectif. Puisque les techniques orientées continu sont relativement peu efficaces pour traiter la partie discrète des problèmes mixtes et, réciproquement, les techniques orientées discret ne sont pas efficaces pour traiter la partie continue de ces problèmes, nous avons ainsi cherché à mettre en oeuvre une coopération de deux outils, le premier étant spécialement conçu pour résoudre les problèmes discrets, le second conçu en particulier pour la résolution de problèmes continus.

Dans ce but, nous sommes partis des outils existants pour voir ce qui manquait vraiment à chacun pour traiter efficacement les problèmes mixtes, que ce soit sur le plan de la modélisation ou bien des techniques de résolution. Nous avons donc commencé par effectuer une présentation détaillée des capacités, en terme de modélisation et de résolution de problèmes mixtes, des principaux outils non commerciaux disponibles actuellement, dans le but de choisir lesquels faire collaborer. Cette présentation détaillée des différents outils permettant la modélisation et la résolution de problèmes discrets ou continus, qu'ils soient fournis sous la forme de programmes en langage de modélisation dédié, en langage déclaratif type *Prolog* étendu ou bien directement en langage orienté objet, nous a permis de choisir quels outils interfacer dans l'idée que l'un puisse combler les lacunes de l'autre et réciproquement.

Nous avons alors entrepris d'utiliser les capacités de propagation sur les contraintes arithmétiques d'un solveur continu, *RealPaver* pour améliorer les capacités dans ce domaine d'un solveur discret, *Choco*. Nous avons décrit la manière dont nous avons pu mettre en oeuvre la collaboration entre ces outils, à savoir confier le rôle principal au second, l'exploration de l'espace de recherche et le filtrage des contraintes globales discrètes, tandis que le premier était mis à contribution pour filtrer efficacement les domaines en utilisant les contraintes arithmétiques continues et mixtes grâce à l'arithmétique des intervalles.

Finalement, nous avons présenté des résultats expérimentaux que nous avons obtenus sur un ensemble de problèmes de test. La réalisation d'expériences sur une variété de problèmes mixtes de différents types nous a permis de mesurer l'efficacité de la collaboration que nous avons mise en oeuvre en la comparant à l'utilisation des outils séparément vis-à-vis de la résolution d'une sélection de problèmes mixtes. En pratique, cette collaboration nous permet d'abord de faciliter la phase de modélisation en

offrant la possibilité de s'abstraire d'un mécanisme de modélisation rigide et difficile à utiliser. En outre, cette collaboration nous permet de résoudre plus efficacement certains types de problèmes mixtes. À travers ces travaux, nos principales contributions sont ainsi les suivantes

- Nous avons mis au point une plateforme logicielle doté d'un langage de modélisation facilitant l'expression des problèmes mixtes en fusionnant les hautes capacités d'expression de contraintes arithmétiques du langage de modélisation dédié d'un solveur continu, avec les nombreuses primitives de modélisation de contraintes globales d'un solveur discret, améliorant en cela les capacités de chacun des deux outils pris individuellement.

- Nous avons également mis en évidence la nette supériorité en termes d'efficacité de résolution d'une collaboration d'outils pour résoudre les problèmes mixtes qui font intervenir à la fois des contraintes arithmétiques, nécessitant une évaluation par l'arithmétique des intervalles, et des contraintes globales, requérant des algorithmes de filtrage dédiés et optimisés.

7.1.3 Utilisation de la contrainte Alldifferent dans un solveur continu

Dans ce deuxième chapitre, nous avons présenté une étude expérimentale à propos de la résolution dans un solveur continu de problèmes mixtes utilisant la contrainte discrète `alldifferent`. Cela nous a permis d'investiguer le champ des techniques de programmation par contraintes dédiées à la résolution de problèmes mixtes, un champ où très peu de travaux ont déjà été réalisés jusqu'ici. Dans ce chapitre, nous nous sommes intéressés à la recherche d'une manière simple de permettre aux solveurs continus de gérer efficacement les contraintes globales et de tirer avantage de cette force pour résoudre des problèmes mixtes. Cette étude expérimentale nous a conduit à résoudre des problèmes d'optimisation issus du monde réel et à implémenter une variété d'algorithmes de filtrage dans un solveur de contraintes continues. En particulier, nous avons étudié la possibilité d'utiliser la contrainte `alldifferent` pour modéliser des problèmes mixtes du monde réel et les résoudre dans un solveur continu. Cette contrainte est en effet l'une des plus utilisées et des mieux étudiées.

Nous avons d'abord présenté ce qui nous motivait à aller dans cette direction. En effet, l'utilisation de l'arithmétique des intervalles en programmation par contraintes est par exemple un moyen puissant de résoudre les MINLP de façon rigoureuse et complète, i.e. de les résoudre avec les deux intéressantes caractéristiques de calculer des solutions garanties et de ne perdre aucune solution. Ensuite, nous avons montré comment utiliser la contrainte `alldifferent` pour la modélisation de quelques problèmes mixtes du monde réel. Nous avons ainsi montré comment faire usage de cette contrainte globale discrète pour modéliser une certaine catégorie de problèmes MINLP, trouvés dans la littérature.

Nous avons alors mis en lumière les solutions possibles pour la formulation de la contrainte et son filtrage dans un solveur continu. Nous avons effectué une comparaison des différentes modélisations possibles de la contrainte globale entière `alldifferent`, ainsi que des techniques de filtrage qui sont dédiées à chacune d'entre elles. Nous avons ainsi passé en revue les principales façons dont on peut traiter une contrainte `alldifferent` dans un solveur continu, basé sur la notion d'intervalle, en reformulant la contrainte ou bien en implémentant des algorithmes de filtrage dédiés.

Finalement, nous avons réalisé des expériences sur différents types de problèmes, de façon à étudier l'impact du choix de la méthode de traitement sur la résolution proprement dite. Nous avons présenté les résultats des expériences que nous avons réalisées pour comparer les efficacités respectives des dif-

férentes modélisations et algorithmes de filtrage associés. Il ressort de nos expériences que même s'il reste très avantageux en général d'opter pour une modélisation globale de la contrainte plutôt que son éclatement en une clique de contraintes binaires, il n'est pas intéressant d'opter pour un algorithme de filtrage trop coûteux, comme c'est le cas de l'algorithme de filtrage de type consistance de bornes dans sa version *state-of-the-art*, en ce qui concerne la résolution des problèmes difficiles issus du monde réel. À travers ces travaux, nos principales contributions tiennent en deux points :

- Nous avons montré qu'un algorithme de filtrage global, dédié mais très simple à implémenter comme celui que nous avons réalisé pour la contrainte `CARD`, était très compétitif par rapport au très optimisé algorithme de filtrage en ce qui concerne la résolution de problèmes mixtes très contraints issus du monde réel, tout en restant relativement compétitif dans le cas particulier de problèmes purement discret, ce qui n'est pas du tout le cas des formulations $\neq$ et surtout `MILP`, cette dernière étant pourtant la formulation d'usage ;

- Nous avons découvert que la relaxation convexe d'enveloppe telle qu'en usage actuellement pouvait être dépassée en termes de performances dans notre solveur à intervalles par un ensemble de contraintes beaucoup plus facile à construire et à utiliser, et qui pouvait permettre une résolution jusqu'à deux fois plus rapide pour résoudre certaines instances des problèmes mixtes issus du monde réel.

7.1.4 Spécialisation du filtrage basé sur l'arithmétique des intervalles

Dans ce troisième chapitre, nous avons présenté des améliorations pour les techniques de filtrage de domaines basées sur l'arithmétique des intervalles. Ces améliorations sont dédiées à une meilleure utilisation des contraintes mixtes, i.e. les contraintes qui font intervenir à la fois des variables discrètes et des variables continues. Nous avons ainsi réfléchi à une spécialisation des techniques de filtrage basées sur l'arithmétique des intervalles, dédiée au cas des contraintes arithmétiques faisant intervenir des variables entières. Prolongeant les travaux antérieurs de APT et ZOETEWEIJ à propos d'une arithmétique des intervalles d'entiers, nous avons introduit de nouvelles règles de calcul sur les intervalles de certaines opérations arithmétiques.

Après avoir présenté les travaux de APT et ZOETEWEIJ à propos de l'utilisation d'une arithmétique des intervalles d'entiers pour la résolution de problèmes purement discrets, nous avons montré comment le cadre classique de résolution dédié aux problèmes réels pouvait être modifié pour mieux prendre en compte la notion d'intégralité des variables discrètes. Nous avons réécrit dans un premier temps de nouvelles définitions à certaines opérations de l'arithmétique des intervalles, en les adaptant au contexte particulier d'intervalles d'entiers. Celles-ci nous permettent de faire des encadrements plus fins du résultat de l'évaluation aux intervalles d'une expression arithmétique, en prenant en compte les caractéristiques des domaines entiers sur lesquelles portent ces opérations. Nous avons également mis en lumière une nouvelle façon de prendre en compte les contraintes d'intégralité au cours de la propagation de contraintes, plus tôt dans le processus de propagation.

Nous avons ensuite montré comment nous avons implémenté ces idées dans un solveur de contraintes continues. Après avoir donné quelques précisions à propos de l'implémentation des modifications à effectuer, nous avons réalisé des expériences pour en étudier le comportement en pratique. Nous avons alors pu présenter les expériences effectuées dans le but d'observer le comportement de nos améliorations vis-

à-vis de la résolution de problèmes mixtes de dimension variable. La mise en évidence expérimentale des effets sur la résolution de problèmes mixtes des nouvelles techniques mises en oeuvre nous a ainsi permis de constater l'impact drastique que peuvent avoir en pratique ces optimisations naturelles du calcul sur les temps effectifs de résolution des problèmes, au delà de l'efficacité théorique que nous avions déjà pu remarquer. Ainsi, nos contributions à travers ces travaux sont les suivantes :

- Nous donnons de nouvelles définitions pour des opérations typiques de l'arithmétique des intervalles, dans le cas particulier où leurs opérandes sont des variables entières, et montrons que cela permet, à travers un augmentation de la force du filtrage obtenu, d'accélérer de manière très significative la résolution des problèmes ;

- Nous proposons une nouvelle façon de gérer les contraintes d'intégralité des variables pendant le processus de résolution : prendre en compte ces contraintes le plus tôt possible au cours de la propagation, c'est-à-dire non pas comme un post-traitement au filtrage des domaines mais au moment même de celui-ci, permet d'accélérer de manière drastique la résolution des problèmes.

7.2 Perspectives de recherche

Dans cette dernière section, nous faisons le bilan des différentes perspectives de recherche dont nous pouvons faire état dans le cadre de cette fin de thèse, ayant bien évidemment pour objet les problèmes mixtes. Puis nous pourrons finalement élargir ces perspectives au problème plus général de la résolution rigoureuse des problèmes impliquant des variables continues.

7.2.1 Dans le domaine des problèmes mixtes

Nous commençons par récapituler les directions dans lesquelles les travaux que nous avons présentés ici nous semblent avantageusement améliorables.

Limitations de la collaboration de solveurs

Des travaux futurs pourraient permettre d'améliorer les points faibles majeurs d'une collaboration de solveurs telle que nous l'avons mise en oeuvre. En effet, on peut imaginer pouvoir profiter davantage des forces d'une telle collaboration en s'affranchissant de ses principaux inconvénients que sont pour nous les temps de recopie de domaines d'un solveur à l'autre, ainsi que le manque de souplesse du propagateur que nous avons utilisé pour calculer le point-fixe des domaines à partir des contraintes arithmétiques continues et mixtes du modèle :

- En ce qui concerne le premier point, on peut en effet concevoir une symbiose plus profonde entre les deux outils, qui pourrait amener par exemple à proposer une gestion plus intelligente de la recopie des domaines, au moins dans le cas particulier où il n'y a pu se produire aucune réduction et où la recopie en retour est inutile.

- Quant au deuxième point, on peut également envisager une collaboration plus en profondeur qui permettrait une sélection plus fine des opérateurs de réduction à rappeler en cas de modification de domaines, par exemple en offrant la possibilité de n'effectuer le calcul de point-fixe que sur un sous-ensemble des contraintes arithmétiques déléguées au solveur continu.

- Il est également intéressant d'envisager une collaboration de solveurs centrée cette fois sur le solveur continu. En effet, selon le type de problèmes il peut y avoir ou bien une majorité de contraintes arithmétiques, ou bien une majorité de contraintes globales discrètes. Dans le premier cas, il est sûrement avantageux de confier le rôle principal au solveur continu et de ne déléguer au solveur discret que le filtrage des contraintes continues.

Intégration de contraintes globales au cadre de résolution continu

De futurs travaux pourraient permettre de s'interroger quant à l'impact de la constriction du problème sur l'efficacité des différentes modélisations et filtrages mis en oeuvre pour les contraintes globales. En outre, la réutilisation des conclusions de nos travaux sur l'étude du `alldifferent` dans des outils largement en usage comme *AMPL* nous semble à la fois simple à mettre en oeuvre et prometteuse en pratique. Enfin, on peut imaginer la généralisation de notre approche aussi bien à d'autres problèmes mixtes utilisant la contrainte `alldifferent` qu'à des problèmes mixtes utilisant d'autres contraintes globales :

- Il semble important de trouver davantage de problèmes mixtes, issus du monde réel ou bien généré d'une façon ou d'une autre, mais ayant un rapport plus équilibré entre le nombre de contraintes discrètes et le nombre de contraintes continues, de façon à mieux connaître les comportements des différentes modélisations selon les caractéristiques des problèmes résolus.

- À l'heure actuelle, aucun solveur de la platforme de résolution *AMPL* n'est en mesure de traiter le mot-clé `card`, pourtant déjà une primitive de son langage de modélisation. Au vu de nos résultats, il semblerait très utile d'y implémenter une méthode de filtrage associée qui pourrait permettre de modéliser simplement et efficacement la contrainte `alldifferent`.

- Au delà des travaux que nous présentons ici sur la contrainte `alldifferent`, on peut tout à fait imaginer réaliser un travail similaire à propos d'autres contraintes globales. Ainsi, on pourrait même à terme se doter d'un ensemble minimal de primitives de type `card`, permettant d'exprimer un ensemble maximal de contraintes globales et implémentable facilement dans un solveur de contraintes continues.

Vers une arithmétique des intervalles mixtes

Nous voyons ici deux pistes principales pour de futures recherches sur le thème d'une meilleure maîtrise d'une arithmétique des intervalles mixte :

- De futurs travaux pourraient être consacrés réaliser une étude plus exhaustive de l'arithmétique des intervalles, dans le but de trouver éventuellement d'autres opérations arithmétiques qui pourraient

bénéficier d'une telle spécialisation aux intervalles d'entiers, ou bien le cas échéant de prouver que cela n'est pas possible.

- En outre, il pourrait être intéressant de mieux comprendre l'impact sur la résolution des problèmes de chacune des améliorations du filtrage que nous avons proposées ici, en réalisant des observations plus précises comme compter le nombre d'opérations arithmétiques effectuées, ou activer/désactiver chacune des optimisations, avant de comparer les résultats obtenus.

7.2.2 Dans le domaine des problèmes continus

Mais au-delà des problèmes mixtes, il nous semble intéressant, pour conclure cette thèse, d'évoquer des pistes dans le cadre plus général de la résolution de tout problème à variables continues, qu'il implique également ou non des variable discrètes. En effet, nous avons eu l'opportunité au cours de cette thèse de faire l'application des techniques de programmation par contraintes pour la modélisation et la résolution d'un problème de conception en robotique, pour lequel la programmation par contraintes n'avait jamais été mise à contribution jusqu'ici. Pour ce faire, nous avons eu à modifier le cadre classique de résolution d'un solveur de contraintes continues, destiné à la satisfaction de contraintes uniquement, pour y intégrer la notion de recherche rigoureuse d'optimum. En conclusion de ces travaux, nous sont apparues des directions de recherche intéressantes que nous partageons ici :

- Nous avons non seulement pu constater que pour le problème considéré, les méthodes de programmation par contraintes sont très compétitives par rapport aux méthodes de programmation mathématique, mais surtout que les résultats obtenus par l'une et l'autre méthode divergent sensiblement. Ainsi, il semble primordial d'élucider avec certitude la cause de cette divergence afin de chercher à corriger ce qui en est la source.

- De plus, des travaux futurs devraient permettre de mettre en oeuvre d'autres critères de filtrage utilisant la fonction-objectif, par exemple en se basant sur de la recherche locale, et d'ainsi accélérer encore la convergence de notre algorithme d'optimisation de façon à permettre de traiter efficacement des problèmes de taille plus importante.

- Notons également qu'au niveau de l'exploitation des résultats d'un point de vue purement robotique, nous avons encore à faire l'application de la méthode que nous avons décrite ici afin de tracer les isocontours de l'erreur maximum dans l'espace de travail, à partir du calcul de l'erreur de pose maximum pour un échantillon de points de l'espace de travail du robot et de façon à mieux connaître l'impact du jeu dans les articulations sur l'erreur de pose.

- En outre, d'autres travaux sont déjà en cours pour également prendre en compte les imperfections d'usinage des pièces du robot. L'enjeu est ici de chercher à compenser le jeu dans les articulations par ces imperfections et ainsi réduire drastiquement les coûts d'assemblage.

- Enfin, des membres de l'équipe de recherche en robotique de l'Institut de Recherche en Communications et Cybernétique de Nantes, avec laquelle nous avons collaboré à l'occasion de ces travaux, confirment le besoin réel qui existe pour l'intégration de méthodes de résolution efficaces et rigou-

reuses dans des outils de modélisation mathématique comme *MatLab*. Des travaux sont déjà en cours au sein de l'équipe Méthodes Ensemblistes pour l'Optimisation, du Laboratoire d'Informatique de Nantes Atlantique, au sein de laquelle a été réalisée cette thèse, à propos de l'inclusion de telles méthodes dans ces outils.

Quand les lettrés supérieurs ont entendu parler de la Voie,
Ils la pratiquent avec zèle.
Quand les lettrés du second ordre ont entendu parler de la Voie,
Tantôt ils la conservent, tantôt ils la perdent.
Quand les lettrés inférieurs ont entendu parler de la Voie,
Ils la tournent en dérision.
S'ils ne la tournaient pas en dérision,
Elle ne mériterait pas le nom de Voie !

C'est pourquoi les Anciens disaient :
Celui qui a l'intelligence de la Voie paraît enveloppé de ténèbres.
Celui qui est avancé dans la Voie ressemble à un homme arriéré.
Celui qui est à la hauteur de la Voie ressemble à un homme vulgaire.
L'homme d'une vertu supérieure est comme une vallée.
L'homme d'une grande pureté est comme couvert d'opprobre.
L'homme d'un mérite immense paraît frappé d'incapacité.
L'homme d'une vertu solide semble dénué d'activité.
L'homme simple et vrai semble vil et dégradé.

La Voie est un grand carré dont on ne voit pas les angles ;
Un grand vase qui semble loin d'être achevé ;
Une grande voix dont le son est imperceptible ;
Une grande image dont on n'aperçoit point la forme.
La Voie se cache et personne ne peut la nommer.
Elle sait prêter secours aux êtres,
Les guider vers la perfection.

— Lao-Tseu, De la Voie et de sa vertu, XLI.

Annexes

A

COMPLÉMENTS À L'ÉTAT DE L'ART

A.1 Filtrage de domaines

A.1.1 Domaines discrets

De nombreuses évolutions des consistances d'arc et de chemin ont vu le jour depuis l'apparition de ces dernières.

Généralisations

Dans [Fre85] a été élaborée une généralisation des consistances d'arc et de chemin, la notion de (i, j)-consistance. Grâce à ce nouveau formalisme, on peut voir la consistance d'arc comme étant la (1,1)-consistance, où toute valeur doit, pour être conservée, pouvoir être complétée par la valeur d'une variable adjacente dans le graphe. De même, la consistance de chemin est la (2,1)-consistance, où tout couple de valeurs doit pouvoir être complété par la valeur d'une variable adjacente.

Définition A.1 ((i,j)-consistance)**.** Un CSP discret $P = (\mathcal{X}, \mathcal{D}, C)$ est *(i,j)-consistant* ssi toute instanciation localement consistante de i variables peut être étendue à j variables.

FREUDER a auparavant élaboré dans [Fre78] la notion de k-consistance, une autre généralisation des consistances d'arc et de chemin. L'idée est de considérer les inconsistances par ensemble de k variables, et de pouvoir concevoir des consistances plus fortes que la consistance de chemin. De cette façon, la consistance d'arc est la 2-consistance, et la consistance de chemin la 3-consistance. Mais s'il est dans l'idée possible d'aller au delà de 3, on conçoit aisément que la complexité algorithmique d'un tel filtrage puisse le rende inutilisable en pratique.

Définition A.2 (k-consistance)**.** Un CSP discret $P = (\mathcal{X}, \mathcal{D}, C)$ est *k-consistant* ssi toute instanciation consistante de $k-1$ variables peut être étendue à k variables.

Définition A.3 (k-consistance forte)**.** Un CSP discret $P = (\mathcal{X}, \mathcal{D}, C)$ est *fortement k-consistant* ssi il est i-consistant pour tout $i \leq k$.

Si on peut dès lors parler de PC forte, quand un CSP est à la fois 2-consistant et 3-consistant, il reste que la PC est algorithmiquement prohibitive à assurer. Pour pallier cet inconvénient tout en cherchant à filtrer plus de valeur que l'AC, diverses consistances ont progressivement été proposées, des consistances intermédiaires entre AC et PC forte. Notons que ces nouvelles consistances sont dites *de domaine*, c'est-à-dire qu'il ne s'agira pas comme pour la PC de retirer simultanément d'un ensemble de variables des n-uplets inconsistants, mais uniquement de retirer du domaine d'une variable les valeurs ne respectant pas une certaine propriété.

Consistances restreintes

Dans [Ber95], on trouve ainsi une première restriction de la PC, de façon à accélérer le filtrage tout en conservant un important pouvoir filtrant :

Définition A.4 (Consistance de chemin restreinte)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$, le domaine D d'une variable X est *chemin-consistant restreint* ssi il est arc-consistant et si, pour toute valeur u de D n'ayant qu'un seul support v dans le domaine d'une variable Y pour une contrainte C_{XY} donnée, chaque variable Z telle qu'il existe à la fois une contrainte C_{XZ} portant sur X et Z et une contrainte C_{YZ} portant sur Y et Z contient dans son domaine une valeur w telle que (u, w) et (v, w) satisfont respectivement C_{XZ} et C_{YZ} – i.e. la paire (u, v) est chemin-consistante.

Le terme anglo-saxon original est *restricted path consistency* (RPC). Cette consistance est destinée à être un intermédiaire entre AC et PC. En effet, la RPC est moins forte que la PC puisqu'elle ne s'intéresse qu'aux valeurs n'ayant qu'un seul support pour une contrainte, et non à toutes les valeurs comme c'est le cas pour la PC. Elle reste cependant plus forte que l'AC puisqu'elle considère des triplets de variables et non juste des couples.

On trouve aujourd'hui un algorithme de filtrage par RPC, dans le pire des cas de complexité temporelle en $O(en + ed^2 + cd^2)$ et spatiale en $O(ed + cd)$, avec d la taille du plus grand domaine, n le nombre de variables, e le nombre de contraintes et c le nombre de cliques[1] de trois sommets dans le graphe [DB97a].

Exemple A.6. [Ber95] *Filtrage par RPC :*

(i) CSP initial *(ii) CSP après filtrage par RPC*

En effet, la valeur c est l'unique support dans le domaine de X_j pour (X_i, a) et la paire $((X_i, a), (X_j, c))$ n'est pas chemin-consistante puisqu'il n'y a aucune valeur w dans le domaine de X_k telle que la combinaison $((X_i, a), (X_j, c), (X_k, w))$ satisfait toutes les contraintes. Une fois (X_i, a) retirée, les valeurs (X_k, e) et (X_j, d), non arc-consistantes, sont éliminées.

La définition de la RPC peut en outre être généralisée aux valeurs ayant non pas un seul support, mais au plus k supports :

Définition A.5 (Consistance de chemin k-restreinte)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$, le domaine D d'une variable X est *chemin-consistant k-restreint* ssi il est arc-consistant et si, pour toute valeur u de D ayant au plus k supports v_i dans le domaine d'une variable Y_i pour une contrainte C_{XY_i} donnée, chaque variable Z telle qu'il existe à la fois une contrainte C_{XZ} portant sur X et Z et une contrainte C_{Y_iZ} portant sur Y_i et Z contient dans son domaine une valeur w telle que (u, w) et (v_i, w) satisfont respectivement C_{XZ} et C_{Y_iZ} i.e. chaque paire (u, v_i) est chemin-consistante.

Il s'agit d'une extension de la RPC aux valeur ayant non pas 1 mais au plus k supports. Elle est donc logiquement plus forte que la RPC puisqu'elle examine plus de valeurs. Poursuivant les recherches dans

[1] Une clique est un graphe dans lequel tout sommet est relié par un arc à chacun des autres sommets.

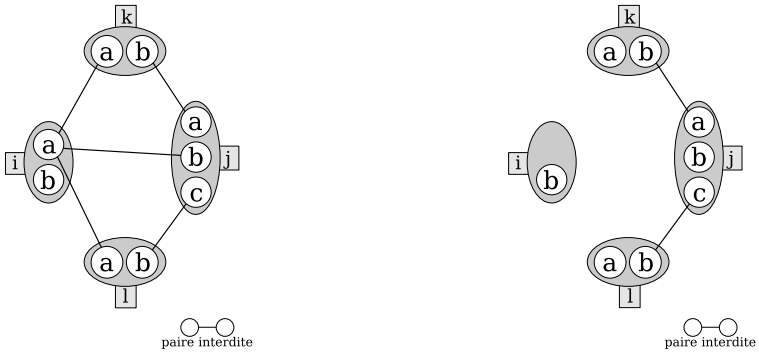
la direction du meilleur compromis entre filtrage et complexité, Debruyne propose dans [DB97a] une autre évolution de la RPC :

Définition A.6 (Consistance de chemin max-restreinte)**.** Etant donné un CSP discret $P = (\mathcal{X}, \mathcal{D}, C)$, le domaine D d'une variable X est *chemin-consistant max-restreint* ssi toute valeur de D possède un support chemin-consistant dans chaque domaine de variable Y pour laquelle il existe une contrainte portant à la fois sur X et sur Y.

La max-RPC est plus forte que la RPC, car ici on s'intéresse à toutes les valeurs —non pas juste à celles ayant un unique support— et plus forte également que la k-RPC, puisqu'on ne se restreint pas à examiner les valeurs qui satisfont un nombre de support d'au plus k, toutes les valeurs sont considérées. Pourtant, la max-RPC peut être plus facile à obtenir que n'importe quelle k-RPC : en effet, il n'y a pas à chercher dans le CSP quelles valeurs satisfont la condition d'avoir au plus k supports, ce qui pouvait s'avérer très coûteux quand k est grand.

Dans sa version optimale, l'algorithme a dans le pire des cas une complexité temporelle en $O(en + ed^2 + cd^3)$ et spatiale en $O(ed + cd)$, avec d la taille du plus grand domaine, n le nombre de variables, e le nombre de contraintes et c le nombre de 3-cliques dans le graphe [DB97a].

Exemple A.7. [Deb01] *Filtrage par max-RPC :*



(i) CSP initial *(ii) CSP après filtrage par max-RPC*

En effet, la valeur (X_i, a) n'a aucun support chemin-consistant.

Consistances inverses

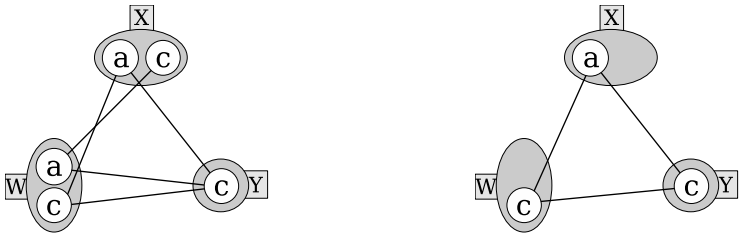
Toujours dans cette optique de trouver un compromis optimal entre complexité et efficacité de filtrage, dans [FE96] est élaborée la notion de consistance de chemin inverse :

Définition A.7 (Consistance de chemin inverse)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$, le domaine D d'une variable X est *chemin-consistant inverse* ssi toutes les valeurs des domaines de toutes deux variables Y et Z sont consistantes pour toute valeur de D par rapport à toute contrainte portant sur un sous-ensemble de $\{X, Y, Z\}$.

Selon le formalisme de Freuder, il s'agit de la (1,2)-consistance, toute valeur peut être étendue à tout couple de valeurs supplémentaire. On parle de PC *inverse* puisque la PC est la (2,1)-consistance, où tout couple de valeur peut être étendu à toute troisième valeur.

Dans sa version optimale, l'algorithme a dans le pire des cas une complexité temporelle en $O(en + ed^2 + cd^3)$ et spatiale en $O(ed + cd)$, avec d la taille du plus grand domaine, n le nombre de variables, e le nombre de contraintes et c le nombre de 3-cliques dans le graphe [Deb00].

Exemple A.8. [FE96] *Filtrage par PIC :*



(i) CSP initial *(ii) CSP après filtrage par PIC*

En effet, il n'y aucune combinaison satisfaisable de valeurs pour X et Y si W prend la valeur a, ni de combinaison satisfaisable de valeurs pour Y et W si X prend la valeur c.

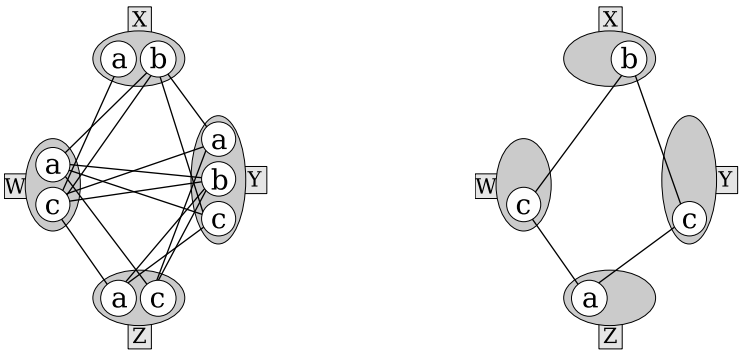
C'est également dans [FE96] que FREUDER développe la notion de consistance inverse de voisinage :

Définition A.8 (Consistance inverse de voisinage)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$, le domaine D d'une variable X est *inverse voisinage-consistant* ssi pour toute valeur v de D, il existe des valeurs w_i dans les domaines de chacune des variables Y_i pour lesquelles existe une contrainte C_j portant sur X et sur une des Y_i au moins, w_i telles que toutes les C_j sont satisfaites.

Le terme original est *neighborhood inverse consistency*. Les deux idées de bases sont la consistance inverse, développée par FREUDER dès 1985 sans être implémentée, et la consistance de voisinage, qui insiste sur le sous-problème que forme une valeur et les contraintes qu'elles imposent autour d'elle dans le graphe [FE96].

L'algorithme de filtrage par NIC a aujourd'hui dans le pire des cas une complexité temporelle en $O(g^2(n + ed)d^{g+1})$ et spatiale en $O(n)$, avec d la taille du plus grand domaine, n le nombre de variables, e le nombre de contraintes et g le degré maximum d'un noeud du graphe [FE96].

Exemple A.9. *[FE96] Filtrage par NIC :*



(i) CSP initial *(ii) CSP après filtrage par NIC*

En effet, la valeur (W, a) *n'est compatible avec aucune combinaison de valeurs satisfaisable de son voisinage* $\{X, Y, Z\}$*. De même pour* (X, a) *et son voisinage* $\{W, Y\}$*,* (Y, a) *et son voisinage* $\{X, W, Z\}$*, etc.*

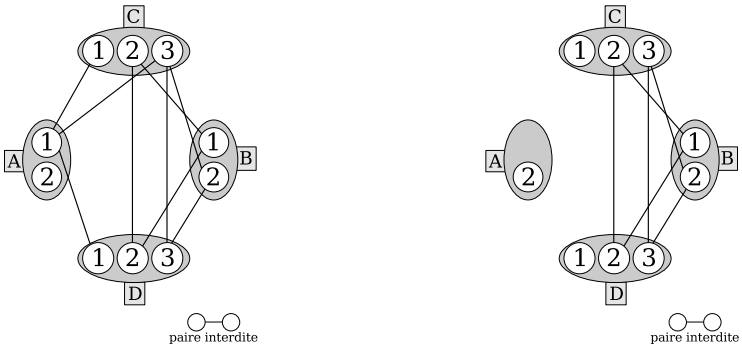
Enfin, pour terminer avec les consistances discrètes, citons [DB97b] et la consistance de singleton :

Définition A.9 (Consistance de singleton)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$ et une consistance discrète CONS, le domaine D d'une variable X est *singleton CONS-consistant* ssi D n'est pas vide et pour toute valeur v de D, le CSP dans lequel on a remplacé X par v n'est pas CONS-inconsistant.

Ici, on est sûr qu'il existe dans chaque domaine au moins une valeur qui participe avec v à une solution. En outre, en fonction de la consistance sur laquelle se base la consistance de singleton, on obtient une consistance différente : consistance d'arc singleton (SAC) pour la consistance d'arc, consistance de chemin restreinte singleton (SRPC) pour la RPC, etc.

L'algorithme de filtrage par SAC a dans le pire des cas une complexité temporelle en $O(en^2d^4)$ et spatiale en $O(ed)$, tandis que l'algorithme de filtrage par SRPC a lui dans le pire des cas une complexité temporelle en $O(en + n^2(e + c)d^4)$ et spatiale en $O(ed + cd)$ –avec d la taille du plus grand domaine, n le nombre de variables, e le nombre de contraintes et c le nombre de 3-cliques du graphe [DB97b].

Exemple A.10. [Deb05] *Filtrage par SRPC :*



(i) CSP initial *(ii) CSP après filtrage par SRPC*

En effet, la valeur $(A, 1)$ *a un unique support 2 dans le domaine de C. Le CSP dans lequel on a remplacé le domaine de A par* $\{1\}$ *est inconsistant pour la RPC : l'unique support de* $(A, 1)$ *est* $(C, 2)$*, et celui-ci n'est pas chemin-consistant car il n'y a aucune valeur u de B ni v de D telle que* $((A, 1), (B, u), (C, 2), (D, v))$ *est une solution.*

Synthèse

Pour conclure sur ce sujet, remarquons que ces consistances peuvent ainsi être mises en relation les unes par rapport aux autres en fonction de la quantité de valeurs qu'elles filtrent. Par exemple, RPC est plus forte qu'AC mais plus faible que PIC, ou encore SAC est plus forte que max-RPC. Le résultat de cette comparaison exhaustive est la figure A.1 page 123, extraite de [DB97b] et [Deb01], où est représenté

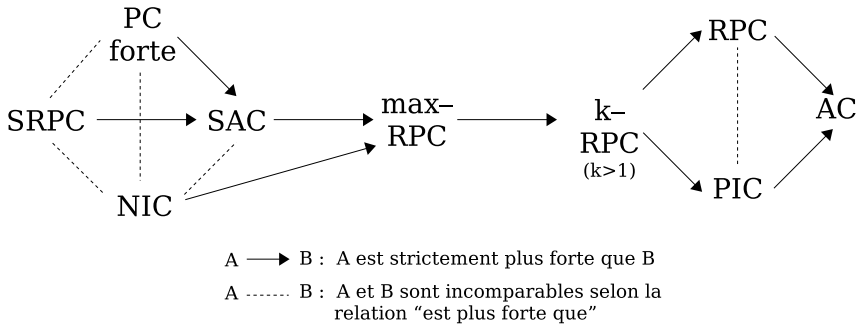


Figure A.1 – Relations entre les différentes consistances discrètes

cet ordre sur les consistances discrètes. On notera que certaines sont incomparables : cet ordre n'est pas total.

A.1.2 Domaines continus

Des consistances plus fortes que la consistance d'enveloppe et de pavé ont été mises au point pour pallier les faiblesses de ces dernières.

Consistances fortes

Une limitation de la consistance d'enveloppe, que partage également la consistance de pavé, est la localité de son application i.e. elle prend en compte les contraintes séparément. Or dès les premières tentatives d'adapter aux domaines continus les techniques de filtrage par consistance, Lhomme définit dans [Lho93] une consistance plus forte, capable de filtrer les domaines en utilisant toutes les contraintes à la fois :

Définition A.10 (Consistance 3B)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$, un domaine $D_i = [a, b]$ de $\mathcal{D}$ est *3B-consistant* ssi les CSP P' et P'' dans lesquels on a remplacé D_i respectivement par $[a_i, a_i^+]$ et $[b_i^-, b_i]$ ne sont pas inconsistants après filtrage par consistance d'enveloppe.

Il s'agit effectivement d'une consistance de plus grande force puisqu'elle garantit qu'un domaine ne puisse plus être réduit par aucune contrainte. De plus, il s'agit en fait d'une famille de consistances plutôt que d'une consistance, un peu à l'image de la consistance discrète de singleton décrite précédemment dans cette section.

Exemple A.11. [BG06] *Soient les contraintes* $x + y = 0$ *et* $x - y = 0$ *avec* $D_x = D_y = [-1, 1]$. *Ces domaines sont irréductibles au sens de la consistance d'enveloppe. Cependant, la consistance 3B permet de filtrer des valeurs. En effet, pour* $x = -1$ *par exemple, les contraintes impliquent à la fois* $y = 1$ *et* $y = -1$*, ce qui provoque le rejet de cette valeur de x.*

Notons en outre que la définition de la consistance 3B peut être réécrite pour n'importe quelle consistance, en l'occurrence la consistance de pavé, comme dans [VHMD97] :

Définition A.11 (Bound-consistance)**.** Etant donné un CSP $P = (\mathcal{X}, \mathcal{D}, C)$, un domaine $D_i = [a, b]$ de $\mathcal{D}$ est *Bound-consistant* ssi les CSP P' et P'' dans lesquels on a remplacé D_i respectivement par $[a, a^+]$ et $[b^-, b]$ ne sont pas inconsistants après filtrage par consistance de pavé.

Plus récemment a été introduite dans [CTN04] une nouvelle consistance ayant pour objet les domaines continus. Elle se base sur la notion de sous-ensemble arc-consistant d'un CSP, et est plus forte que l'AC.

Définition A.12 (Consistance d'ensemble de pavés)**.** Un CSP $P = (\mathcal{X}, \mathcal{D}, C)$ est *ensemble de pavés-consistant* ssi $\mathcal{D}$ est l'ensemble $\{B'\}$, $B' \in \mathcal{D}$, des pavés maximaux tels que $(\mathcal{X}, \{B'\}, C)$ est arc-consistant.

Cette nouvelle consistance est plus forte que l'AC, mais reste plus faible qu'une résolution globale du CSP i.e. elle laisse dans les domaines certaines valeurs qui ne sont pas des solutions du CSP. Ces deux points sont illustrés respectivement par les deux exemples A.12 et A.13 :

Exemple A.12. [CTN04] *Soient les domaines* $D_x = D_y = D_z = [-2, 2]$ *et* $D_w = [1, 4]$. *Soient les contraintes* $x^2 = w$, $x = y$, $y = z$ *et* $x = -z$. *La consistance d'arc est atteinte pour les domaines*

$D_x = D_y = D_z = [-2,-1] \cup [1,2]$ *et* $D_w = [1,4]$*, alors que le consistance d'ensemble de pavé élimine le pavé tout entier puisque, dans le domaine de x, ni* $[-2,-1]$ *ni* $[1,2]$ *n'appartiennent à un sous-pavé arc-consistant.*

Exemple A.13. [CTN04] *Soient les domaines* $D_x = D_y = D_z = [0,2]$*. Soient les contraintes* $x = y$*,* $x + z = 2$ *et* $y = z$*. Comme le pavé initial est arc-consistant, il est ensemble de pavés-consistant. Or la vraie solution est* $(1,1,1)$*.*

A.2 Exploration de l'espace de recherche

A.2.1 CSP discrets

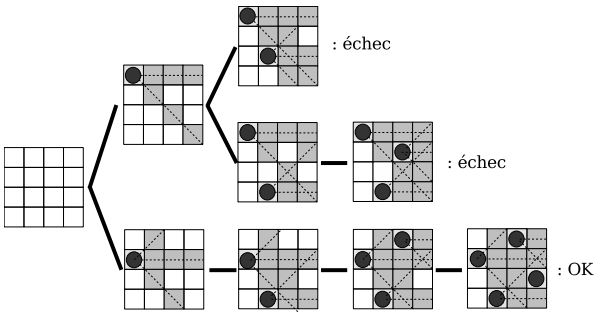
Intéressons-nous à présent aux améliorations du backtracking qui ont vu le jour depuis son apparition dans les années 1970. Outre l'apparition de la notion de consistance, destinée à contrer le phénomène de *trashing*[2] observé dans [Mac77], et dont l'efficacité est facilement constatable en comparant les arbres des exemples 2.22 et 2.23, on distingue trois catégories d'algorithmes de recherche résultant de l'évolution du backtracking : les algorithmes prospectifs, rétrospectifs et rétroprospectifs.

Algorithmes prospectifs

Les algorithmes prospectifs, tout d'abord, comme leur nom l'indique ont *la tête tournée vers le futur*. Il s'agit d'augmenter le niveau de filtrage à chaque noeud de façon à limiter les explorations infructueuses.

Un premier algorithme de ce type est le *forward checking* : après chaque instanciation, d'une variable x, on élimine du domaine de chaque variable y non encore instanciée les valeurs inconsistantes avec les contraintes contenant x et y [HE80].

Exemple A.14. [Bar] *Forward checking pour le problème des 4 reines*[3] *:*



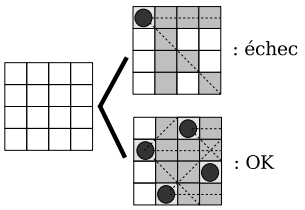
Ensuite est apparu dans [SF94] *MAC* pour *maintaining arc consistency*. Le CSP est initialement rendu arc-consistant et, à chaque nouvelle variable instanciée, on en propage les effets jusqu'à ce que le

[2]Ce terme désigne un comportement perfectible du backtracking qui consiste à traverser inutilement des portions de l'arbre : quand une valeur affectée à X_1 est incompatible avec toute valeur de X_3, chercher entre temps quelles valeurs de X_2 sont compatibles avec celle de X_1 est une perte de temps.

[3]Le problème des n reines est un classique de la programmation par contraintes : il s'agit de placer n reines sur un échiquier de telle sorte qu'aucune ne soit en position d'être prise par une autre.

CSP soit de nouveau arc-consistant : après chaque instanciation d'une variable, on vérifie, pour toutes les valeurs de chaque domaine de variable non encore instanciée, qu'il existe un support pour les contraintes qui la contiennent.

Exemple A.15. [Bar] *MAC pour le problème des 4 reines :*

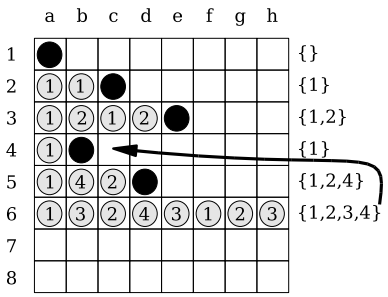


Algorithmes rétrospectifs

La deuxième catégorie d'algorithmes de recherche est celle des algorithmes rétrospectifs qui, eux, ont *la tête tournée vers le passé* et regardent dans les anciens échecs d'instanciation.

Un algorithme de ce type est par exemple le *conflict-directed backjumping (CBJ)* apparu dans [Pro93] : chaque échec sur le choix d'une valeur est expliqué par les précédents choix qui entrent en conflit, et en cas d'impossibilité d'affecter une variable –quand aucune valeur potentielle n'est consistante– on revient en arrière jusqu'à la variable la plus récemment instanciée dont la valeur provoque une incompatibilité.

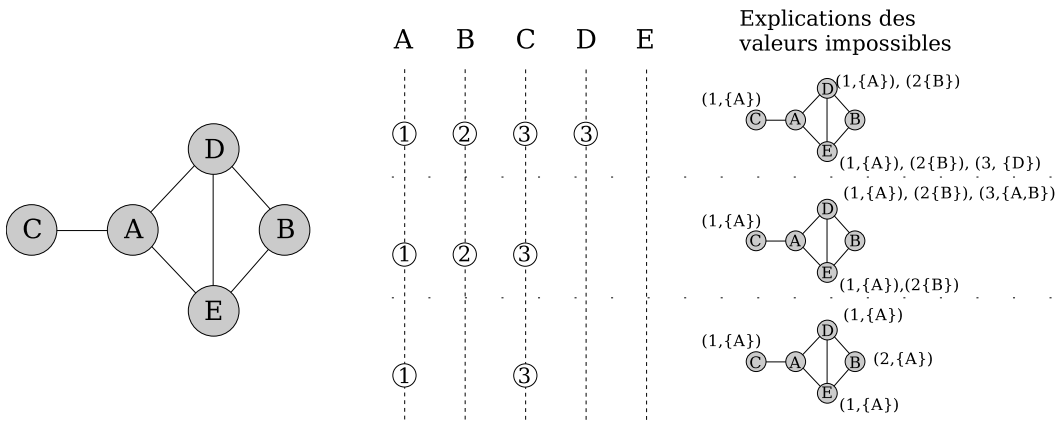
Exemple A.16. [Deb05] *CBJ pour le problème des 8 reines :*



On mémorise pour chaque reine la (les) ligne(s) de la (des) reine(s) qui bloque(nt) une (des) case(s). Ainsi, quand à la ligne 6 il n'y a plus d'alternative à tester, on revient en arrière jusqu'au choix le plus récent, c'est-à-dire 4.

Un autre exemple d'algorithme rétrospectif est le *dynamic backtracking (DBT)* [Gin93]. Comme pour le CBJ, l'idée est de mémoriser et d'utiliser les causes des échecs d'instanciation de valeurs. Mais cette fois, on ne revient que sur la valeur incompatible la plus récente et on ne remet pas en cause les autres variables déjà instanciées. Ainsi, il s'agit plus d'une réparation que d'un retour arrière.

Exemple A.17. [Gin93] *DBT pour une 3-coloration d'un graphe :*



Affectant 1 *à A,* 2 *à B,* 3 *à C puis* 3 *à D, il n'y a plus aucune valeur possible pour E. D étant la variable la plus récemment affectée, on revient sur sa valeur* 3*, rendue inconsistante du fait des valeurs de A et B. A nouveau il n'y a plus aucune valeur possible pour D –puisque* 3 *vient de devenir impossible à affecter– on revient donc sur B la variable explicative la plus récemment affectée, pour laquelle la valeur* 2 *devient inconsistante. Au prochain pas, on devra lui affecter une autre valeur sans revenir entre temps sur la valeur de C.*

Algorithmes rétro-prospectifs

Enfin, la troisième catégorie d'algorithme de recherche est celle des algorithmes rétro-prospectifs, qui conjuguent les propriétés des deux précédentes catégories. L'algorithme MAC-CBJ, par exemple, intègre au CBJ le maintien de la consistance d'arc, mais en pratique il est rarement plus efficace que MAC. Cela n'est pas le cas de l'algorithme MAC-DBT qui adjoint le maintien de la consistance d'arc au retour-arrière dynamique [JDB00] : cet algorithme se base sur la notion d'*explication* —au sens de cause d'une inconsistance— qui y semble très prometteuse en ce qui concerne l'amélioration de l'efficacité des stratégies de recherche.

BIBLIOGRAPHIE

[Apt00] APT K. R. : The role of commutativity in constraint propagation algorithms. *ACM Transactions on Programming Languages and Systems 22* (2000), 1002–1036.

[Apt03] APT K. : *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003.

[AW07] APT K. R., WALLACE M. : *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[AZ03] APT K. R., ZOETEWEIJ P. : A comparative study of arithmetic constraints on integer intervals. In *Apt, Fages, Rossi, Szeredi, Váncza (Eds.) Recent Advances in Constraints, LNAI 3010* (2003), Springer-Verlag, pp. 1–24.

[AZ07] APT K. R., ZOETEWEIJ P. : An analysis of arithmetic constraints on integer intervals. *Constraints 12*, 4 (2007), 429–468.

[Bar] BARTAK R. : Online guide to constraint programming (1998). http ://kti.mff.cuni.cz/ ~bartak/constraints/. Visité le 15 Mai 2010.

[Bar99] BARTAK R. : Constraint programming : In pursuit of the holy grail. In *Proceedings of WDS99 (invited lecture)* (1999), pp. 555–564.

[BCDP07] BELDICEANU N., CARLSSON M., DEMASSEY S., PETIT T. : Global constraint catalogue : Past, present and future. *Constraints 12*, 1 (2007), 21–62.

[BDM03] BUSSIECK M. R., DRUD A. S., MEERAUS A. : Minlplib - a collection of test models for mixed-integer nonlinear programming. *INFORMS Journal on Computing 15*, 1 (2003), 114–119.

[Ber95] BERLANDIER P. : Improving domain filtering using restricted path consistency. In *Proceedings of IEEE CAIA-95* (1995).

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

[Bes06] BESSIÈRE C. : Constraint propagation. In *Handbook of Constraint Programming* (2006), Elsevier.

[BFR99] BESSIERE C., FREUDER E. C., REGIN J. : Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence 107* (1999), 125–148.

[BG06] BENHAMOU F., GRANVILLIERS L. : Continuous and interval constraints. In *Handbook of Constraint Programming* (2006), Elsevier, pp. 592–624.

[BGGP99] BENHAMOU F., GOUALARD F., GRANVILLIERS L., PUGET J.-F. : Revising hull and box consistency. In *International Conference on Logic Programming (ICLP'99)* (1999).

[BLL*09] BELOTTI P., LEE J., LIBERTI L., MARGOT F., WACHTER A. : Branching and bounds tighteningtechniques for non-convex minlp. *Optimization Methods Software 24*, 4-5 (2009), 597–634.

[BMV94] Benhamou F., McAllester D., Van Hentenryck P. : Clp(intervals) revisited. In *ILPS'94 (International Symposium on Logic Programming)* (Ithaca, NY, USA, 1994), The MIT Press.

[BMVH94] Benhamou F., McAllester D., Van-Hentenryck P. : Clp(intervals) revisited. In *Proceedings of the International Symposium on Logic Programming* (1994), pp. 124–138.

[BSG*10] Berger N., Soto R., Goldsztejn A., Caro S., Cardou P. : Finding the maximal pose error in robotic mechanical systems using constraint programming. In *Proceedings of IEAAIE 2010* (2010).

[BVH97] Benhamou F., Van Hentenryck P. : Introduction to the special issue on interval constraints. *Constraints 2* (1997), 107–112.

[CC09] Cardou P., Caro S. : *The Kinematic Sensitivity of Robotic Manipulators to Manufacturing Errors*. Tech. rep., IRCCyN, Internal Report No RI2009_4, 2009.

[CDR99] Collavizza H., Delobel F., Rueher M. : Comparing Partial Consistencies. *Reliable Computing 5*, 3 (1999), 213–228.

[Che07] Chenouard R. : *Résolution par satisfaction de contraintes appliquée à l'aide à la décision en conception architecturale*. PhD thesis, École Nationale Supérieure d'Arts et Métiers de Bordeaux, 2007.

[Cle87] Cleary J. G. : Logical arithmetic. *Future Computing Systems 2*, 2 (1987), 125–147.

[CLPP05] Casas-Liza J., Pinto J. M., Papageorgiou L. G. : Mixed integer optimization for cyclic scheduling of multiproduct plants under exponential performance decay. *Chemical engineering research & design 83*, 10 (2005), 1208–1217.

[CR97] Csendes T., Ratz D. : Subdivision direction selection in interval methods for global optimization. *SIAM Journal on Numerical Analysis 34* (1997), 922–938.

[CTN04] Chabert G., Trombettoni G., Neveu B. : New light on arc consistency over continuous domains. In *International workshop on Constraint Propagation and Implementation. At CP conference* (2004).

[CTN05] Chabert G., Trombettoni G., Neveu B. : Box-set consistency for interval-based constraint problems. In *SAC* (2005), pp. 1439–1443.

[DB97a] Debruyne R., Bessière C. : From restricted path consistency to max-restricted path consistency. In *Proceedings of Third International Conference on Principles and Practice of Constraint Programming* (1997), pp. 312–326.

[DB97b] Debruyne R., Bessière C. : Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI-97* (1997), pp. 412–417.

[Deb00] Debruyne R. : A property of path inverse consistency leading to an optimal pic algorithm. In *Proceedings ECAI'00* (2000), pp. 88–92.

[Deb01] Debruyne R. : Domain filtering consistencies. *Journal of Artificial Intelligence Research 14* (2001), 205–230.

[Deb05] Debruyne R. : Cours de Master sur les réseaux de contraintes à domaines discrets, Faculté des sciences et techniques, Nantes, 2005.

[Dec92] Dechter R. : Constraint networks. In *Encyclopedia of Artificial Intelligence* (1992), Wiley and Sons.

[Dec03] DECHTER R. : *Constraint Processing*. Morgan Kaufman, 2003.

[DG86] DURAN M. A., GROSSMANN I. E. : An outer-approximation algorithm for a class of mixed-integer nonlinear programs. *Mathematical Programming 36*, 3 (1986), 307–339.

[DGDG95] DEB K., GOYAL M., DEB K., GOYAL M. : Optimizing engineering designs using a combined genetic search. In *Proceedings of the Sixth International Conference on Genetic Algorithms* (1995), pp. 521–528.

[DH55] DENAVIT J., HARTENBERG R. : A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices. *ASME Journal of Applied Mechanics 23* (1955), 215–221.

[DP88] DECHTER R., PEARL J. : Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence 34* (1988), 1–38.

[Fal94] FALTINGS B. : Arc-consistency for continuous variables. *Artificial Intelligence 65*, 2 (1994), 363–376.

[FE96] FREUDER E. C., ELFE C. D. : Neighborhood inverse consistency preprocessing. In *Proceedings of AAAI-96* (1996), pp. 202–208.

[FGK02] FOURER R., GAY D. M., KERNIGHAN B. W. : *AMPL : A Modeling Language for Mathematical Programming*. Duxbury Press, November 2002.

[FHJ*05] FRISCH A. M., HARVEY W., JEFFERSON C., MARTÍNEZ-HERNÁNDEZ B., MIGUEL I. : The essence of ESSENCE : A constraint language for specifying combinatorial problems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (2005), pp. 73–88.

[FM06] FREUDER E., MACKWORTH A. : Constraint satisfaction : An emerging paradigm. In *Handbook of Constraint Programming* (2006), Elsevier, pp. 13–27.

[Fre78] FREUDER E. C. : Synthesizing constraint expressions. *Communications of the ACM 21*, 11 (1978), 958–966.

[Fre85] FREUDER E. C. : A sufficient condition for backtrack-bounded search. *Journal of the ACM 32*, 4 (1985), 755–761.

[Fre97] FREUDER E. C. : In pursuit of the holy grail. *Constraints 2*, 1 (1997), 57–61.

[Frü09] FRÜHWIRTH T. : *Constraint Handling Rules*. Cambridge University Press, 2009.

[FS98] FOGARASY A., SMITH M. : The Influence of Manufacturing Tolerances on the Kinematic Performance of Mechanisms. In *Proceedings of the Institution of Mechanical Engineers - Part C : Mechanical Engineering Science* (1998), pp. 35–47.

[Gal85] GALLAIRE H. : Logic programming : Further developments. In *SLP* (1985), pp. 88–96.

[GAM] Gams. http ://www.gams.com/. Visité le 12 Septembre 2010.

[GB06] GRANVILLIERS L., BENHAMOU F. : Algorithm 852 : Realpaver : an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software 32*, 1 (2006), 138–156.

[Gel98] GELLE E. : *On the generation of locally consistent solution spaces in mixed dynamic constraint problems*. PhD thesis, Ecole polytechnique fédérale de Lausanne EPFL, Lausanne, 1998.

[GF03] GELLE E., FALTINGS B. : Solving mixed and conditional constraint satisfaction problems. *Constraints 8* (2003), 107–141.

[GH88] GÜSGEN H.-W., HERTZBERG J. : Some fundamental properties of local constraint propagation. *Artificial Intelligence 36* (1988), 237–247.

[Gin93] GINSBERG M. L. : Dynamic backtracking. *Journal of Artificial Intelligence Research 1*, 1 (1993), 25–46.

[GJM06] GENT I. P., JEFFERSON C., MIGUEL I. : Minion : A fast scalable constraint solver. In *Proceedings of ECAI 2006* (2006), pp. 98–102.

[GMR07] GENT I. P., MIGUEL I., RENDL A. : Tailoring solver-independent constraint models : A case study with ESSENCE' and Minion. In *Proceedings of the 7th International Symposium on Abstraction, Reformulation and Approximation* (2007), pp. 18–21.

[Gou00] GOUALARD F. : *Langages et Environnements en Programmation par Contraintes d'Intervalles*. PhD thesis, Ecole Doctorale : Sciences pour l'Ingénieur, Nantes, 2000.

[Gra05] GRANVILLIERS L. : Cours de Master sur les contraints numériques, Faculté des sciences et techniques, Nantes, 2005.

[GW99] GENT I., WALSH T. : *CSPLib : a benchmark library for constraints*. Tech. rep., Technical report APES-09-1999, 1999. Available from http ://csplib.cs.strath.ac.uk/. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).

[Hal35] HALL P. : On representatives of subsets. *Journal of the London Mathematical Society 10* (1935), 26–30.

[Han92] HANSEN E. : *Global Optimization Using Interval Analysis*. Marcel Dekker, New York, 1992.

[HDmT92] HENTENRYCK P. V., DEVILLE Y., MAN TENG C. : A generic arc-consistency algorithm and its specializations. *Artificial Intelligence 57* (1992), 291–321.

[HE80] HARALICK R., ELLIOTT G. : Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 (1980), 263–313.

[Heu06] HEUSCH M. : *Modélisation et résolution d'une application d'aide au déploiement d'antennes radio en programmation par contraintes sur le discret et le continu*. PhD thesis, UFR Sciences et Techniques, Université de Nantes, 2006.

[HMK97] HENTENRYCK P. V., MCALLESTER D., KAPUR D. : Solving polynomial systems using a branch and prune approch. *SIAM Journal on Numerical Analysis 34*, 2 (1997), 797–827.

[Hoo02] HOOKER J. N. : Logic, optimization and constraint programming. *INFORMS Journal on Computing 14* (2002), 295–321.

[Hoo06] HOOKER J. : Operations research methods in constraint programming. In *Handbook of Constraint Programming* (2006), Elsevier, pp. 521–564.

[Hoo07] HOOKER J. N. : *Integrated Methods for Optimization*, vol. 100 of *International Series in Operations Research & Management Science*. Springer, 2007.

[HS05] HOOS H. H., STÜTZLE T. : *Stochastic Local Search : Foundations and Applications*. Morgan Kaufmann Publishers, 2005.

[HSG01] HOFSTEDT P., SEIFERT D., GODEHARDT E. : A framework for cooperating solvers - a prototypic implementation. In *Proceedings of the Workshop on Cooperative Solvers in Constraint Programming - CoSolv. At the Seventh International Conference on Principles and Practice of Constraint Programming* (2001).

[HT06] Hoos H. H., Tsang E. : Local search methods. In *Handbook of Constraint Programming* (2006), Elsevier, pp. 135–167.

[HWPS98] Harjunkoski I., Westerlund T., Pörn R., Skrifvars H. : Different transformations for solving non-convex trim-loss problems by MINLP. *European Journal of Operational Research 105*, 3 (1998), 594–603.

[IEE85] IEEE Task P754 : *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, Aug. 12 1985.

[Inn02] Innocenti C. : Kinematic Clearance Sensitivity Analysis of Spatial Structures With Revolute Joints. *ASME Journal of Mechanical Design 124* (2002), 52–57.

[JDB00] Jussien N., Debruyne R., Boizumault P. : Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming* (2000), pp. 249–261.

[JL87] Jaffar J., Lassez J.-L. : Constraint logic programming. In *POPL* (1987), pp. 111–119.

[Kea96] Kearfott R. : *Rigorous Global Search : Continuous Problems*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1996.

[Kei] Keith M. : The alphametics page. http ://www.mathematik.uni-bielefeld.de/~sillke/puzzles/alphametic/alphametic-mike-keith.html. Visité le 20 avril 2010.

[KK94] Kannan B. K., Kramer S. N. : An augmented lagrange multiplier based method for mixed integer discrete continuous optimization and its applications to mechanical design. *ASME Journal of Mechanical Design* (1994), 116–318.

[Lab00] Laburthe F. : Choco : Implementing a cp kernel. In *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)* (2000).

[Law76] Lawler E. : *Combinatorial Optimization : Networks and Matroids*. Holt, Rinehart and Winston, New-York, 1976.

[Lho93] Lhomme O. : Consistency techniques for numeric csp. In *IJCAI 1993* (1993).

[LMR07] Lebbah Y., Michel C., Rueher M. : An Efficient and Safe Framework for Solving Optimization Problems. *Journal of Computational and Applied Mathematics 199*, 2 (2007), 372–377. Special Issue on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2004).

[LOQTvB03] López-Ortiz A., Quimper C.-G., Tromp J., van Beek P. : A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence* (2003), pp. 245–250.

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

[MF85] Mackworth A. K., Freuder E. C. : The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence 25* (1985), 65–74.

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

[MLS94] Murray R., Li Z., Sastry S. : *A Mathematical Introduction to Robotic Manipulation*. CRC, Boca Raton, Fl., 1994.

[Mon74] MONTANARI U. : Networks of constraints : Fundamental properties and application to picture processing. *Information Sciences 7*, 3 (1974), 95–132.

[Mon07] MONTEMANNI R. : A mixed integer programming formulation for the total flow time single machine robust scheduling problem with interval data. *Journal of Mathematical Modelling and Algorithms 6*, 2 (2007), 287–296.

[Moo66] MOORE R. E. : *Interval Analysis*. Prentice-Hall, 1966.

[Moo79] MOORE R. E. : *Methods and Applications of Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1979.

[MR91] MONTANARI U., ROSSI F. : Constraint relaxation may be perfect. *Artificial Intelligence 48* (1991), 143–170.

[MR99] MONFROY E., RETY J.-H. : Chaotic iteration for distributed constraint propagation. In *Proceedings of the 1999 ACM Symposium on Applied Computing* (1999), pp. 19–24.

[MZL09] MENG J., ZHANG D., LI Z. : Accuracy Analysis of Parallel Manipulators With Joint Clearance. *ASME Journal of Mechanical Design 131* (2009).

[Neu04] NEUMAIER A. : Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica 13* (2004), 271–369.

[OV90] OLDER W., VELLINO A. : Extending prolog with constraint arithmetic on real intervals. In *Canadian Conference on Computer & Electrical Engineering* (1990).

[Pro93] PROSSER P. : Domain filtering can degrade intelligent backtrack search. In *Proceedings IJCAI'93* (1993), pp. 262–267.

[Pug98] PUGET J.-F. : A fast algorithm for the bound consistency of alldiff constraints. In *AAAI/IAAI* (1998), pp. 359–366.

[QGH*99] QUIST A., GEEMERT R. V., HOOGENBOOM J., ILLEST T., KLERK E. D., ROOS C., TERKALY T. : Finding optimal nuclear reactor core reload patterns using nonlinear optimization and search heuristics. *Engineering Optimization 32*, 2 (1999), 143–176.

[Rai08] RAISER F. : Semi-automatic generation of CHR solvers for global constraints. In *CP '08 : Proceedings of the 14th international conference on Principles and Practice of Constraint Programming* (2008), pp. 588–592.

[Rég94] RÉGIN J.-C. : A filtering algorithm for constraints of difference in csps. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994), pp. 362–367.

[RJL*08] ROCHART G., JUSSIEN N., LORCA X., PRUDHOMME C., CAMBAZARD H., RICHAUD G., MENANA J., MALAPERT A. : Choco : an open source java constraint programming library. In *Proceedings of the 3rd International CSP Solver Competition* (2008), pp. 7–13.

[ROM09] RIOLO R., O'REILLY U.-M., MCCONAGHY T. : *Genetic Programming Theory and Practice VII*. Springer Publishing Company, Incorporated, 2009.

[Rum88] RUMP S. M. : Algorithms for verified inclusions—theory and practice. In *Reliability in computing : the role of interval methods in scientific computing* (San Diego, CA, USA, 1988), Academic Press Professional, Inc., pp. 109–126.

[RvBW06] ROSSI F., VAN BEEK P., WALSH T. : Introduction. In *Handbook of Constraint Programming* (2006), Elsevier, pp. 3–12.

[SCDRA] SANTOS-COSTA V., DAMAS L., REIS R., AZEVEDO R. : The Yap Prolog User Manual, 2000. http ://www.ncc.up.pt/~vsc/yap. Visité le 19 septembre 2010.

[SF94] SABIN D., FREUDER E. C. : Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of ECAI94* (1994).

[Sin96] SINGH M. : Path consistency revisited. *International Journal on Artificial Intelligence Tools 5*, 1-2 (1996), 127–141.

[Sut63] SUTHERLAND I. E. : *Sketchpad, A Man-Machine Graphical Communication System*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1963.

[Tac09] TACK G. : *Constraint Propagation – Models, Techniques, Implementation*. Doctoral dissertation, Saarland University, Jan. 2009.

[TS02] TAWARMALANI M., SAHINIDIS N. V. : *Convexification and Global Optimization in Continuous And*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[Tsa92] TSANG E. : Problem solving with genetic algorithms. *Science and Engineering Magazine, University of Essex Publication*, 6 (1992), 14–17.

[vH01] VAN HOEVE W.-J. : The all_different constraint : A survey. *CoRR cs.PL/0105015* (2001).

[VHMD97] VAN-HENTENRYCK P., MICHEL L., DEVILLE Y. : *Numerica : A Modeling Language for Global Optimization*. The MIT Press, Cambridge, MA, USA, 1997.

[Wal72] WALTZ D. L. : Generating semantic descriptions from drawings of scenes with shadows. In *The psychology of computer vision* (1972), Winston P., (Ed.), New York : McGraw-Hill, pp. 19–91.

[Wal75] WALTZ D. L. : Understanding line drawings of scenes with shadows. In *Psychology of Computer vision* (1975), New York : McGraw-Hill.

[WNS97] WALLACE M., NOVELLO S., SCHIMPF. J. : *ECLiPSe : A Platform for Constraint Logic Programming*. Tech. rep., IC-Parc, Imperial College, London, 1997.

LISTE DES TABLEAUX

TABLE DES FIGURES

TABLE DES MATIÈRES

Modélisation et résolution en programmation par contraintes de problèmes mixtes continu/discret de satisfaction de contraintes et d'optimisation

Nicolas BERGER

Résumé

Les contraintes sont un moyen générique de représenter les règles qui gouvernent notre monde. Étant donné un ensemble de contraintes, une question centrale est de savoir s'il existe une possibilité de toutes les satisfaire simultanément. Cette problématique est au cœur de la programmation par contraintes, un paradigme puissant pour résoudre efficacement des problèmes qui apparaissent dans de nombreux domaines de l'activité humaine. Initialement dédiée, dans les années 1980, à la résolution de problèmes d'intelligence artificielle à variables entières, c'est dans les années 1990 que la programmation par contraintes a été employée à la résolution de problèmes à variables réelles. Cependant, les problèmes mixtes —utilisant à la fois variables entières et réelles— n'ont été que très peu considérés jusqu'ici par la programmation par contraintes.

Dans cette thèse, nous nous plaçons du point de vue de la résolution de problèmes continus. Nous proposons et mettons en oeuvre différentes améliorations de ce cadre de résolution :

- Intégration de la notion de recherche rigoureuse d'optimum au cadre classique de résolution sans objectif, afin de modéliser et résoudre un problème de conception en robotique ;
- Collaboration de deux solveurs, l'un discret l'autre continu, plus efficace que chacun des outils pour résoudre les problèmes utilisant contraintes continues et contraintes discrètes ;
- Comparaison des différentes modélisations et filtrages possibles de la contrainte globale discrète `alldifferent`, permettant de l'utiliser dans un solveur dédié au continu ;
- Spécialisation des techniques de filtrage basées sur l'arithmétique des intervalles, augmentant la puissance de filtrage des contraintes arithmétiques discrètes et mixtes.

Mots-clés : programmation par contraintes, optimisation globale, collaboration de solveurs, contraintes globales, arithémtique des intervalles

Abstract

Constraints are a generic way of expressing regularities that rule our world. Given a set of constraints, an important question is to determine whether there exists a way of simultaneously satisfying them all. This problematic lies deep inside constraint programming, a powerful paradigm for efficiently solving problems that appear in many areas of human activity. During the 80's originally, it was dedicated to solving discrete problems in artificial intelligence. Then during the 90's, constraint programming was extended so as to solve continuous problems. However, mixed problems —with both discrete and continuous variables— have only been a very small concern so far.

In this thesis, we take the continuous solving framework point of view. We propose and implement several improvement of this framework :

- Integration of a rigorous global optimum search to the classical, without objective solving framework, in order to model and solve a conception problem in robotics ;
- Cooperation of two solvers, a discrete one and a continuous one, cooperation which is more efficient than both solvers at solving problems involving both discrete and continuous constraints ;
- Comparison of several ways for modeling and filtering the `alldifferent` discrete, global constraint within a solver dedicated to continuous problems ;
- Specialization of the filtering techniques that make use of interval arithmetic, so as to improve the filtering power of discrete and mixed discrete/continuous arithmetic constraints.

Keywords : constraint programming, global optimization, solver cooperation, global constraints, interval arithmetic