Domaines abstraits en programmation par contraintes

Pelleau Marie

▶ **To cite this version:**

Pelleau Marie. Domaines abstraits en programmation par contraintes. Informatique [cs]. Université de Nantes (FR), 2012. Français. ⟨NNT : ⟩. ⟨tel-03299271⟩

HAL Id: tel-03299271

https://theses.hal.science/tel-03299271v1

Submitted on 26 Jul 2021

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.

HAL Authorization

UNIVERSITÉ DE NANTES

UFR DES SCIENCES ET TECHNIQUES

SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DE MATHÉMATIQUES

Année 2016

N° attribué par la bibliothèque

Domaines abstraits en programmation par contraintes

THÈSE DE DOCTORAT
Discipline : Informatique
Spécialité : Programmation par contraintes

Présentée
et soutenue publiquement par

Marie PELLEAU

le 29 novembre 2012 au LINA, devant le jury ci-dessous

Président et rapporteur	:	Pr. Pedro BARAHONA, Professeur	Universidade Nova de Lisboa
Rapporteur	:	Dr. Xavier RIVAL, Chargé de Recherche	Inria
Examinateurs	:	Dr. Benoit BAUDRY, Chercheur	Inria

Directeur de thèse : Pr. Frédéric BENHAMOU
Co-directeur de thèse : Pr. Pascal VAN HENTENRYCK
Co-encadrant de thèse : Dr. Charlotte TRUCHET
Laboratoire : **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

DOMAINES ABSTRAITS EN PROGRAMMATION PAR CONTRAINTES

Abstract Domains in Constraint Programming

Marie PELLEAU

⋈

favet neptunus eunti

Université de Nantes

Marie PELLEAU

Domaines abstraits en programmation par contraintes

ix+132 p.

Ce document a été préparé avec LaTeX2e et la classe `these-LINA` version v. 1.30 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe `these-LINA` est disponible à l'adresse :

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

Impression : sommaire.tex – 15/2/2016 – 15:39

Révision pour la classe : these-LINA.cls,v 1.30 2005/08/16 17:01:41 mancheron Exp

Résumé

La programmation par contraintes permet de formaliser et résoudre des problèmes fortement combinatoires, dont le temps de calcul évolue en pratique exponentiellement. Les méthodes développées aujourd'hui résolvent efficacement de nombreux problèmes industriels de grande taille dans des solveurs génériques. Cependant, les solveurs restent dédiés à un seul type de variables : réelles ou entières, et résoudre des problèmes mixtes discrets-continus suppose des transformations *ad hoc*. Dans un autre domaine, l'interprétation abstraite permet de prouver des propriétés sur des programmes, en étudiant une abstraction de leur sémantique concrète, constituée des traces des variables au cours d'une exécution. Plusieurs représentations de ces abstractions, appelées domaines abstraits, ont été proposées. Traitées de façon générique dans les analyseurs, elles peuvent mélanger les types entiers, réels et booléens, ou encore représenter des relations entre variables. Dans cette thèse, nous définissons des domaines abstraits pour la programmation par contraintes, afin de construire une méthode de résolution traitant indifféremment les entiers et les réels. Cette généralisation permet d'étudier des domaines relationnels, comme les octogones déjà utilisés en interprétation abstraite. En exploitant l'information spécifique aux octogones pour guider la recherche de solutions, nous obtenons de bonnes performances sur les problèmes continus. Dans un deuxième temps, nous définissons notre méthode générique avec des outils d'interprétation abstraite, pour intégrer les domaines abstraits existants. Notre prototype, AbSolute, peut ainsi résoudre des problèmes mixtes et utiliser les domaines relationnels implémentés.

Mots-clés : Programmation par contraintes, Interprétation abstraite, Domaine abstrait, Octogone, Continu-discret.

Abstract

Constraint Programming aims at solving hard combinatorial problems, with a computation time increasing in practice exponentially. The methods are today efficient enough to solve large industrial problems, in a generic framework. However, solvers are dedicated to a single variable type: integer or real. Solving mixed problems relies on *ad hoc* transformations. In another field, Abstract Interpretation offers tools to prove program properties, by studying an abstraction of their concrete semantics, that is, the set of possible values of the variables during an execution. Various representations for these abstractions have been proposed. They are called abstract domains. Abstract domains can mix any type of variables, and even represent relations between the variables. In this PhD dissertation, we define abstract domains for Constraint Programming, so as to build a generic solving method, dealing with both integer and real variables. We can also study the octagons abstract domain, already defined in Abstract Interpretation. Guiding the search by the octagonal relations, we obtain good results on a continuous benchmark. In a second part, we define our solving method using Abstract Interpretation techniques, in order to include existing abstract domains. Our solver, AbSolute, is able to solve mixed problems and use relational domains.

Keywords: Constraint Programming, Abstract Interpretation, Abstract Domain, Octagon, Continuous-Discrete.

Table des matières

CHAPITRE 1

Introduction

Les avancées récentes de l'informatique sont incontestables. Certaines sont visibles, d'autres moins connues du grand public : aujourd'hui, on sait résoudre rapidement de nombreux problèmes connus pour être difficiles (demandant un temps de calcul important). Par exemple, on sait placer automatiquement, en quelques dizaines de secondes, des milliers d'objets de formes diverses dans un nombre minimum de containers, et en respectant des contraintes spécifiques : accessibilité des marchandises, non-écrasement, *etc.* [Beldiceanu *et al.*, 2007]. La programmation par contraintes permet de formaliser de tels problèmes en utilisant des contraintes qui décrivent un résultat à atteindre (accessibilité de certains objets par exemple). Ces contraintes viennent avec des algorithmes efficaces pour résoudre les problèmes fortement combinatoires.

Dans un autre domaine, la sémantique, l'interprétation abstraite (IA) attaque un problème insoluble dans le cas général : la correction des programmes. Grâce à de solides outils théoriques établis dès sa création (théorèmes de points fixes), l'interprétation abstraite parvient à prouver des propriétés sur des programmes. Dans cette discipline aussi, l'efficacité des méthodes permet des applications impressionnantes : les outils d'interprétation abstraite ont par exemple réussi à prouver qu'il n'y avait pas d'erreurs de débordement dans les commandes de vol de l'Airbus A380, qui contiennent près de 500 000 lignes de code.

Cette thèse se situe à l'interface entre la programmation par contraintes et l'interprétation abstraite, deux domaines de l'informatique aux problématiques a priori assez différentes. En programmation par contraintes, le but est en général d'obtenir de bons temps de calcul pour des problèmes en général NP, ou d'étendre les outils pour pouvoir traiter plus de problèmes. En interprétation abstraite, le but est d'analyser de très grands programmes en capturant un maximum de propriétés. Pourtant, on trouve dans ces deux disciplines une préoccupation commune : cerner un espace impossible ou difficile (en temps de calcul) à calculer exactement : l'ensemble des solutions, en programmation par contraintes, la sémantique du programme en interprétation abstraite. Il s'agit alors de calculer des sur-approximations pertinentes de cet espace. La programmation par contraintes propose des méthodes permettant d'encadrer très précisément cet espace (consistance et propagation), mais toujours avec des sur-approximations cartésiennes (boîtes dans $\mathbb{R}^n$ ou $\mathbb{Z}^n$). L'interprétation abstraite utilise des sur-approximations souvent moins précises mais pas seulement cartésiennes : elles peuvent avoir des formes beaucoup plus variées (boîtes mais aussi octogones, ellipsoïdes, *etc.*), ce qui lui permet de capturer des propriétés plus fines.

Dans cette thèse, nous exploitons les similitudes de ces méthodes de sur-approximation pour intégrer des outils d'interprétation abstraite aux méthodes de programmation par contraintes. Nous redéfinissons les outils de la programmation par contraintes à partir de notions de l'interprétation abstraite (domaines abstraits). Ce n'est pas qu'un exercice intellectuel. En effet, en généralisant la description des sur-approximations, on profite en programmation par contraintes d'un gain important en expressivité. En particulier, on traite ainsi uniformément les problèmes à variables réelles

et à variables entières, ce qui n'est pas le cas actuellement. Nous développons également le cas des octogones, en montrant qu'il est possible d'exploiter les relations capturées par ce domaine particulier pour résoudre plus efficacement les problèmes continus. Enfin, nous réalisons le travail inverse : nous définissons la programmation par contraintes comme une opération abstraite en interprétation abstraite, et développons un solveur capable de traiter en pratique tous les domaines abstraits.

1.1 Contexte

Comme mentionné précédemment, la programmation par contraintes et l'interprétation abstraite ont une préoccupation commune : calculer efficacement et exactement une approximation d'un espace impossible ou difficile. Cependant, les enjeux et les problématiques de ces deux domaines sont différents, de même que leurs champs d'applications.

1.1.1 La programmation par contraintes

La programmation par contraintes, dont les origines remontent à 1974 [Montanari, 1974], repose sur la formalisation des problèmes comme une conjonction de formules logiques du premier ordre, les contraintes. Une contrainte définit une relation entre les variables d'un problème : par exemple, deux objets placés dans un même container ont une intersection géométrique vide, ou bien un objet lourd doit être placé sous un objet fragile. Il s'agit donc de programmation déclarative. La programmation par contraintes offre des méthodes de résolution génériques efficaces pour de nombreux problèmes combinatoires. Les applications académiques et industrielles sont variées : problèmes d'ordonnancement [Grimes et Hebrard, 2011, Hermenier *et al.*, 2011], de conception de tables de substitution en cryptographie [Ramamoorthy *et al.*, 2011], de calcul d'emplois du temps [Stølevik *et al.*, 2011], de prédiction de la structure secondaire d'un ARN en biologie [Perriquet et Barahona, 2009], de conception de réseaux de fibre optique [Pelleau *et al.*, 2009], ou encore d'harmonisation automatique en musique [Pachet et Roy, 2001].

L'une des limitations à l'expressivité des méthodes de programmation par contraintes est qu'elles sont dédiées à la nature du problème à résoudre : les solveurs utilisés pour des problèmes à variables discrètes sont fondamentalement différents des techniques dédiées aux problèmes à variables continues. D'une certaine façon, la sémantique même du problème est différente selon que l'on traite des problèmes discrets ou des problèmes continus.

Or, nombre de problèmes industriels sont mixtes : ils contiennent des variables entières et des variables réelles. C'est le cas par exemple d'un problème consistant à réparer rapidement un réseau électrique après une catastrophe naturelle [Simon *et al.*, 2012], pour rétablir l'électricité au plus vite dans les zones touchées. Dans ce problème, on cherche à établir un plan d'action, et à déterminer les trajets à effectuer par les équipes de réparation intervenant sur le réseau. Certaines des variables sont discrètes, par exemple, à chaque équipement (générateur, ligne) est associée une variable booléenne indiquant s'il est opérationnel ou non. D'autres sont réelles, comme la puissance électrique sur une ligne. Un autre exemple d'application est celui de la conception de la topologie d'un réseau de transmission *multicast* [Chi *et al.*, 2008] : on souhaite concevoir un réseau qui soit fiable, c'est-à-dire continuant de fonctionner quand un de ses composants est défectueux, de sorte que toutes les communications des utilisateurs puissent transiter avec le moins de délai possible. Ici encore, certaines variables sont entières, le nombre de lignes dans le réseau, d'autres sont continues, comme le flot d'information transitant en moyenne sur le réseau.

La convergence des contraintes discrètes et continues en programmation par contraintes est à la fois un besoin industriel et un challenge scientifique.

1.1.2 L'interprétation abstraite

Les fondements de l'interprétation abstraite ont été établis en 1976 par Cousot et Cousot [Cousot et Cousot, 1976]. L'interprétation abstraite est une théorie de l'approximation de sémantiques [Cousot et Cousot, 1977b] dont l'une des applications est la preuve de programmes. Le but est de vérifier et prouver qu'un programme ne contient pas de *bugs*, c'est-à-dire d'erreurs à l'exécution. Les enjeux industriels sont importants. En effet, de nombreux bugs ont marqué l'histoire, comme le bug de l'an 2000, dû à une erreur de conception des systèmes. Le 1er janvier 2000, certains systèmes affichaient la date du 1er janvier 1900. Ce bug devrait se reproduire le 19 janvier 2038 sur certains systèmes UNIX [Robinson, 1999]. Un autre exemple de bug est celui du tristement célèbre vol inaugural de la fusée Ariane 5, qui, suite à une erreur dans le système de navigation, a causé la destruction de la fusée 40 secondes seulement après son décollage.

Tous les jours de nouveaux logiciels sont développés, correspondant à des milliers ou des millions de lignes de code. Tester ou vérifier ces programmes à la main demanderait un temps considérable. La correction de programmes ne pouvant être prouvée de façon générique, l'interprétation abstraite met en place des méthodes permettant d'analyser automatiquement certaines propriétés d'un programme. Les analyseurs reposent sur des opérations sur les sémantiques des programmes, c'est-à-dire l'ensemble des valeurs que peuvent prendre les variables du programme au cours de son exécution. En calculant des sur-approximations de ces sémantiques, un analyseur peut par exemple prouver que les variables ne prendront pas de valeurs au delà des plages autorisées (*overflow*).

De nombreux analyseurs sont développés et utilisés pour divers domaines d'applications comme l'aérospatiale [Lacan *et al.*, 1998, Souyris et Delmas, 2007], la radiothérapie [analyse le code des installations nucléaire,] ou encore la physique des particules [analyse le code du LHC,].

1.2 Présentation de la problématique

Cette thèse s'intéresse aux méthodes de résolution de programmation par contraintes dites complètes, qui permettent de trouver l'ensemble des solutions ou de prouver que celui-ci est vide, le cas échéant. Ces méthodes reposent sur un parcours exhaustif de l'espace de toutes les valeurs possibles, aussi appelé espace de recherche. Elles sont accélérées par des opérations restreignant l'espace à visiter (consistance et propagation). Les méthodes existantes sont dédiées à un certain type de variable, discret ou continu. Face à un problème mixte, contenant à la fois des variables discrètes et continues, la programmation par contraintes n'offre pas de réelle solution et les techniques disponibles sont souvent limitées. Généralement, les variables sont artificiellement transformées afin qu'elles soient toutes discrètes comme dans le solveur Choco [Team, 2010], ou toutes continues comme dans le solveur RealPaver [Granvilliers et Benhamou, 2006].

En interprétation abstraite les programmes analysés contiennent souvent, si ce n'est toujours, différents types de variables. Les théories d'interprétation abstraite intègrent indifféremment de nombreux types de domaines, et ont permis de concevoir des analyseurs traitant uniformément des variables discrètes et continues.

Nous proposons de nous inspirer des travaux de l'interprétation abstraite portant sur les différents types de domaines afin d'offrir de nouvelles méthodes de résolution en programmation par contraintes permettant notamment d'approximer avec des formes variées et de résoudre des problèmes mixtes.

1.3 Organisation de ce manuscrit

Ce manuscrit est organisé comme suit. Le chapitre 2 donne les notions de l'interprétation abstraite et de la programmation par contraintes nécessaires à la compréhension de nos travaux ainsi qu'une analyse des similitudes et différences entre ces deux domaines. En se basant sur les points communs identifiés entre la programmation par contraintes et l'interprétation abstraite, nous définissons des domaines abstraits pour la programmation par contraintes, avec une résolution basée sur ces domaines abstraits, chapitre 3. L'utilisation en programmation par contraintes d'un exemple de domaine abstrait existant en interprétation abstraite, les octogones, est détaillée chapitre 4. Le chapitre 5, donne les détails d'implémentation de la méthode de résolution présentée chapitre 3 pour les octogones. Finalement, le chapitre 6 redéfinit les notions de programmation par contraintes à l'aide des techniques et outils disponibles en interprétation abstraite, permettant de définir une méthode de résolution dite abstraite. Une implémentation prototype est enfin présentée ainsi que de premiers résultats expérimentaux.

1.4 Nos contributions

Les travaux de cette thèse ont pour but d'apporter de nouvelles techniques de résolution à la programmation par contraintes. On distingue deux parties dans les travaux de cette thèse. Dans la première, les domaines abstraits de l'interprétation abstraite sont définis en programmation par contraintes ainsi que différents opérateurs nécessaires à la résolution. Ces nouvelles définitions nous permettent de définir un cadre de résolution uniforme ne dépendant plus, ni du type des variables, ni de la représentation des valeurs des variables. Un exemple de méthode de résolution utilisant le domaine abstrait des octogones et respectant ce cadre est implémenté au sein d'un solveur continu Ibex [Chabert et Jaulin, 2009], et testé sur des exemples de problèmes continus.

Dans la seconde, les différents opérateurs de programmation par contraintes nécessaires à la résolution sont définis en interprétation abstraite, nous permettant de définir une méthode de résolution à l'aide des opérateurs existant en interprétation abstraite. Cette méthode fut ensuite implémentée au-dessus d'Apron [Jeannet et Miné, 2009], une bibliothèque de domaines abstraits.

La plupart des résultats théoriques et pratiques des chapitres 3, 4, 5 et 6 font l'objet de publications dans des conférences ou journaux [Truchet *et al.*, 2010], [Pelleau *et al.*, 2011], [Pelleau *et al.*, b] (accepté avec corrections), [Pelleau *et al.*, a] (soumis).

Notations

Nous donnons ici, toutes les notations qui seront utilisées tout au long de ce manuscrit.

Interprétation abstraite

$\mathcal{D}$	un domaine de manière générale
$\mathcal{D}^\flat$	domaine concret
$\mathcal{D}^\sharp$	domaine abstrait
f	une fonction
$f^\flat$	une fonction concrète
$f^\sharp$	une fonction abstraite
α	une fonction d'abstraction
γ	une fonction de concrétisation
$[a, b]$	un intervalle à bornes flottantes
$[\![a, b]\!]$	un intervalle d'entiers
$\triangledown^\sharp$	un opérateur d'élargissement
$\triangle^\sharp$	un opérateur de rétrécissement
$\bot, \top$	le plus petit/grand élément
ρ_i	un opérateur de clôture inférieure pour un test complexe C_i

Programmation par contraintes

$v_1 \dots v_n$	les variables d'un CSP
$\hat{D}_1 \dots \hat{D}_n$	les domaines initiaux d'un CSP
$C_1 \dots C_p$	les contraintes d'un CSP
$\hat{D}$	l'espace de recherche initial d'un CSP
D	l'espace de recherche
S	l'ensemble des solutions
S_C	l'ensemble des solutions pour la contrainte C
$\mathbb{S}$	l'ensemble des produits cartésiens de sous-ensembles finis d'entiers
$\mathbb{IB}$	l'ensemble des boîte entières
$\mathbb{B}$	l'ensemble des boîtes
ρ_C	le propagateur de la contrainte C
$\mathbb{C}_C^E$	pour un ensemble non vide E et une contrainte C, l'ensemble des éléments de E contenant les solutions de C
$\mathbf{C}_C^E$	le plus petit élément de $\mathbb{C}_C^E$ s'il existe
$\oplus$	un opérateur de coupe
τ	une fonction de précision

Les octogones

$\mathbb{O}$	le treillis des octogones
B	la base canonique de $\mathbb{F}^n$

$B_\alpha^{i,j}$	avec $i, j \in [\![1, n]\!]$, la (i, j)-base tournée obtenue par rotation de α dans le plan défini par i, j de la base canonique
$\mathbf{B}_{\mathbf{O}}^{i,j}$	pour $\mathbf{O}$ un octogone, la boîte dans la base $B_\alpha^{i,j}$
$C^{i,j}$	pour une contrainte C, la (i, j)-contrainte tournée
$\oplus_o$	l'opérateur de coupe octogonal
τ_o	la fonction de précision octogonale
α_o	la fonction d'abstraction octogonale
γ_o	la fonction de concrétisation octogonale

Les domaines abstraits

$\mathcal{S}^\sharp$	le domaine abstrait des produits cartésiens de sous-ensembles finis d'entiers
α_a	la fonction d'abstraction de $\mathcal{D}^\flat$ vers $\mathcal{S}^\sharp$
γ_a	la fonction de concrétisation de $\mathcal{S}^\sharp$ vers $\mathcal{D}^\flat$
τ_a	la fonction de précision de $\mathcal{S}^\sharp$
$\oplus_a$	l'opérateur de coupe de $\mathcal{S}^\sharp$
$\mathcal{I}^\sharp$	le domaine abstrait des boîtes entières
α_b	la fonction d'abstraction de $\mathcal{D}^\flat$ vers $\mathcal{I}^\sharp$
γ_b	la fonction de concrétisation de $\mathcal{I}^\sharp$ vers $\mathcal{D}^\flat$
τ_b	la fonction de précision de $\mathcal{I}^\sharp$
$\oplus_b$	l'opérateur de coupe de $\mathcal{I}^\sharp$
$\mathcal{B}^\sharp$	le domaine abstrait des boîtes
α_h	la fonction d'abstraction de $\mathcal{D}^\flat$ vers $\mathcal{B}^\sharp$
γ_h	la fonction de concrétisation de $\mathcal{B}^\sharp$ vers $\mathcal{D}^\flat$
τ_h	la fonction de précision de $\mathcal{B}^\sharp$
$\oplus_h$	l'opérateur de coupe de $\mathcal{B}^\sharp$
$\mathcal{O}^\sharp$	le domaine abstrait des octogones
α_o	la fonction d'abstraction de $\mathcal{D}^\flat$ vers $\mathcal{O}^\sharp$
γ_o	la fonction de concrétisation de $\mathcal{O}^\sharp$ vers $\mathcal{D}^\flat$
τ_o	la fonction de précision de $\mathcal{O}^\sharp$
$\oplus_o$	l'opérateur de coupe de $\mathcal{O}^\sharp$
$\mathcal{P}^\sharp$	le domaine abstrait des polyèdres
τ_p	la fonction de précision de $\mathcal{P}^\sharp$
$\oplus_p$	l'opérateur de coupe de $\mathcal{P}^\sharp$
$\mathcal{M}^\sharp$	le domaine abstrait des boîtes mixtes
α_m	la fonction d'abstraction de $\mathcal{D}^\flat$ vers $\mathcal{M}^\sharp$
γ_m	la fonction de concrétisation de $\mathcal{M}^\sharp$ vers $\mathcal{D}^\flat$
τ_m	la fonction de précision de $\mathcal{M}^\sharp$
$\oplus_m$	l'opérateur de coupe de $\mathcal{M}^\sharp$
$\mathcal{E}^\sharp$	une complétion disjonctive
π	un opérateur de sélection

CHAPITRE 2

État de l'art

Dans ce chapitre nous présentons les notions sur lesquelles repose l'interprétation abstraite ainsi que les principes de la programmation par contraintes. Nous ne faisons pas une présentation exhaustive de ces deux domaines, mais détaillons uniquement les notions nécessaires à la compréhension de ce manuscrit. Les notions abordées comprennent notamment celles d'ensemble partiellement ordonné, de treillis et de point fixe, qui sont à la base des théories sous-jacentes à ces deux domaines, mais aussi les outils qui seront utilisés par la suite, tels que les opérateurs de rétrécissement et d'élargissement en interprétation abstraite ou la consistance et l'opérateur de coupe en programmation par contraintes. Ce chapitre présente aussi une analyse des similitudes entre l'interprétation abstraite et la programmation par contraintes sur lesquelles reposent les travaux de cette thèse.

In this chapter we present the notions upon which Abstract Interpretation is based and the principles of Constraint Programming. We do not do an exhaustive presentation of both areas, but rather give the notions needed for the understanding of this thesis. The concepts discussed include those of partially ordered set, lattice and fixpoint, which are at the basis of the underlying theories in both fields, and also the in place tools, such as narrowing and widening operators in Abstract Interpretation or consistency and splitting operator in Constraint Programming. This chapter also presents a analysis of the similitudes between Abstract Interpretation and Constraint Programming upon which rely the works presented in this thesis.

2.1 Interprétation Abstraite

Les fondements de l'interprétation abstraite ont été introduits en 1976 par Patrick Cousot et Radhia Cousot [Cousot et Cousot, 1976]. Dans cette section nous faisons une introduction à l'interprétation abstraite en nous concentrant uniquement sur les aspects qui nous intéressent. Pour une présentation plus exhaustive voir entre autres [Cousot et Cousot, 1992a, Cousot et Cousot, 1977a].

2.1.1 Introduction à l'Interprétation Abstraite

Une des applications de l'interprétation abstraite (IA) est de prouver de façon automatique qu'il n'existe pas certains types de bugs dans un programme, c'est-à-dire qu'il n'y ait pas d'erreurs au moment de l'exécution du programme. Regardons directement un exemple.

Exemple 2.1.1 – Soit le programme suivant :

1: real x, y
2: $x \leftarrow 2$
3: $y \leftarrow 5$
4: $y \leftarrow y * (x - 2)$
5: $y \leftarrow x/y$

Faisons l'historique d'exécution de ce programme.

ligne	x	y	
1	?	?	
2	2	?	
3	2	5	
4	2	0	
5	2	NaN	Erreur : division par zéro

L'historique d'exécution permet, sur des exemples jouets comme celui-ci, de détecter rapidement si le programme contient des erreurs. Cependant, les programmes de la vie réelle sont bien plus complexes et bien plus grands en termes de lignes de code, il est impossible de tester toutes les situations réelles, et enfin, d'après le théorème de l'arrêt, il est indécidable dans le cas général de prouver la seule terminaison. Aujourd'hui l'informatique est omniprésente et des programmes critiques peuvent contenir de milliers voire millions de lignes de codes [Havelund *et al.*,]. Les erreurs d'exécution, dans ces programmes, se traduisent directement en coûts importants. Par exemple, en 1996, l'autodestruction de la fusée Ariane 5 était due à un dépassement de capacité des entiers [de la Commission d'enquête Ariane 501,]. Ou encore, en 1991, des missiles Patriot américains n'ont pas réussi à détruire un Scud ennemi causant la mort de 28 soldats et ce uniquement à cause d'une erreur d'arrondi qui s'est propagée tout au long des calculs [Defense,].

Il faut donc s'assurer que ce genre de programmes n'aient pas d'erreurs à l'exécution. De plus, cette vérification doit être effectuée dans une période de temps raisonnable et sans avoir à exécuter le programme. En effet, envoyer des sondes dans l'espace juste pour vérifier si le programme est correct, au sens où il ne contient pas d'erreurs d'exécution, n'est pas une solution viable d'un point de vue économique et écologique.

C'est là que l'interprétation abstraite entre en jeu. L'une de ses applications est de vérifier qu'un programme est correct au moment de la compilation et donc avant qu'il ne soit exécuté. L'idée est d'étudier les valeurs pouvant être prises par les variables tout au long du programme.

On appelle *sémantique* l'ensemble de ces valeurs et *spécification* l'ensemble des comportements souhaités tels que ne jamais diviser par zéro. Si la sémantique du programme respecte toutes les spécifications données, on peut affirmer que le programme est correct.

Une application importante de l'interprétation abstraite est la conception d'analyseurs statiques de programmes qui soient corrects et terminent. Un analyseur est dit correct si celui-ci répond qu'un programme ne contient pas d'erreurs uniquement quand ce programme n'en contient pas. Il existe de nombreux analyseurs statiques, on distinguera deux types d'analyseurs, les analyseurs corrects tels qu'Astrée [Bertrane *et al.*, 2010] et Polyspace [Analyser,], et les analyseurs non-corrects tels que Coverity [Coverity,]. Ces analyseurs ont tous des applications industrielles. Par exemple, Astrée a permis de prouver l'absence d'erreurs d'exécution dans le logiciel de commande de vol du système *fly-by-wire* de l'Airbus A340, et plus récemment d'analyser le code des commandes électriques pour l'Airbus A380 [Alberganti, 2005, Souyris et Delmas, 2007]. Polyspace a permis d'analyser le programme de vol de la fusée Ariane 502 [Lacan *et al.*, 1998] et de vérifier les logiciels de sécurité d'installations nucléaires [analyse le code des installations nucléaire,]. Coverity a quant à lui été utilisé pour vérifier le code de la sonde Curiosity [analyse le code de Curiosity,] et celui du grand collisionneur d'hadrons (LHC) [analyse le code du LHC,], l'accélérateur de particules ayant permis la découverte d'une particule qui semble être le boson de Higgs.

Calculer la sémantique réelle, qu'on appelle *sémantique concrète*, est très coûteux et indécidable dans le cas général. En effet, le théorème de Rice établit que toute propriété non triviale des programmes qui s'énonce uniquement sur les entrées-sorties des programmes est indécidable. Les analyseurs calculent donc une approximation de la sémantique concrète, la *sémantique abstraite*. La première étape est d'associer à chaque instruction du programme à analyser une fonction modifiant l'ensemble des valeurs possibles pour les variables en accord avec l'instruction considérée. Le programme devient donc une composition de ces fonctions et l'ensemble des comportements observables du programme correspond à un point fixe de cette composition de fonctions sur la sémantique concrète. La seconde étape est de définir un domaine abstrait afin de limiter l'expressivité en ne gardant qu'un sous-ensemble de propriétés des variables du programme. Un domaine abstrait est une structure de données représentable en machine utilisée pour représenter certaines propriétés du programme. De plus, les domaines abstraits sont fournis avec des algorithmes efficaces pour le calcul des différents opérateurs de la sémantique concrète ainsi que des opérateurs permettant de calculer un point fixe en un temps fini. L'analyseur observe toujours un sur-ensemble des comportements du programme, ainsi toutes les propriétés trouvées par l'analyseur sont vérifiées pour le programme, cependant, il peut en omettre.

Exemple 2.1.2 – Prenons le programme suivant :

1: real x, y
2: $x \leftarrow random(1, 10)$
3: **si** $x \bmod 2 = 0$ **alors**
4: $\quad y \leftarrow x/2$
5: **sinon**
6: $\quad y \leftarrow x - 1$
7: **fin si**

La variable x étant comprise entre 1 et 10 (instruction 2), on peut, par exemple, déduire que la variable y est comprise entre 0 et 8. Cependant, le fait que pour toute exécution, $y < x$ est

une propriété omise par le domaine abstrait des intervalles, mais pas par le domaine abstrait des polyèdres.

Nous présentons dans les sous-sections suivantes les notions théoriques sur lesquelles repose l'interprétation abstraite.

2.1.2 Présentation générale

La théorie sous-jacente de l'interprétation abstraite utilise les notions de point fixe et treillis. Ces notions sont rappelées dans cette section.

2.1.2.1 Treillis

Les treillis sont une notion bien connue en informatique. Ici ils sont utilisés pour exprimer des opérations sur les domaines abstraits et nécessitent certaines propriétés, telles qu'être complets et clos.

Définition 2.1.1 (Poset)**.** Une relation $\sqsubseteq$ sur un ensemble non vide $\mathcal{D}$ est un *ordre partiel* (po) si et seulement si elle est réflexive, antisymétrique et transitive. Un ensemble muni d'un ordre partiel est appelé *ensemble partiellement ordonné*(poset). S'ils existent, on notera $\bot$ le plus petit élément et $\top$ le plus grand élément de $\mathcal{D}$.

Exemple 2.1.3 – Soit $\mathbb{F}$ l'ensemble des nombres à virgule flottante selon la norme IEEE [Goldberg, 1991]. Pour $a, b \in \mathbb{F}$, on peut définir $[a, b] = \{x \in \mathbb{R}, a \leq x \leq b\}$ l'intervalle réel délimité par les flottants a et b, et $\mathbb{I} = \{[a, b], a, b \in \mathbb{F}\}$ l'ensemble des intervalles. Étant donné un intervalle $I \in \mathbb{I}$, on note $\underline{I}$ (resp. $\overline{I}$) sa borne inférieure (resp. supérieure) et pour tout point x, $\underline{x}$ son approximation flottante inférieure (resp. $\overline{x}$, supérieure).

Soit $\mathbb{I}^n$ l'ensemble des produits cartésiens de n intervalles. L'ensemble $\mathbb{I}^n$ muni de la relation $\subseteq$ est un ensemble partiellement ordonné.

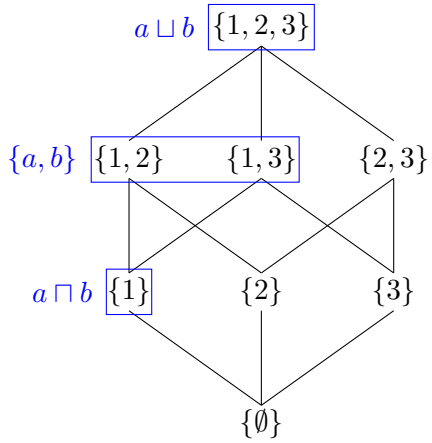
Remarque 2.1.1 – Notons que la relation d'inclusion dans les ensembles $\subseteq$ est un ordre partiel. Donc pour tout ensemble E non vide, $\mathcal{P}(E)$ muni de cette relation est un ensemble partiellement ordonné.

Définition 2.1.2 (Treillis)**.** Un ensemble partiellement ordonné $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap)$ est un *treillis* si et seulement si pour $a, b \in \mathcal{D}$, le couple $\{a, b\}$ admet une plus petite borne supérieure (*least upper bound*, *lub*) notée $a \sqcup b$, et une plus grande borne inférieure (*greatest lower bound*, *glb*) notée $a \sqcap b$. Un treillis est dit *complet* si et seulement si tout sous-ensemble admet à la fois une *lub* et une *glb*.

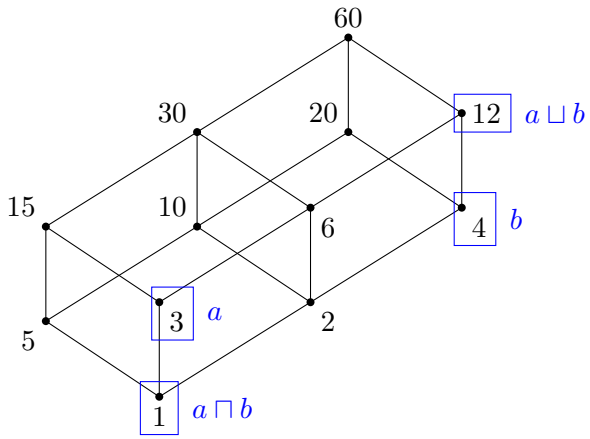
Dans un treillis, tout sous-ensemble fini possède une *lub* et une *glb*. Dans un treillis complet, tout sous-ensemble possède une *lub* et une *glb*, même quand ce sous-ensemble n'est pas fini. Un treillis complet possède donc un plus grand élément, noté $\top$, et un plus petit élément, noté $\bot$.

Remarque 2.1.2 – Notons que tout treillis fini est automatiquement complet

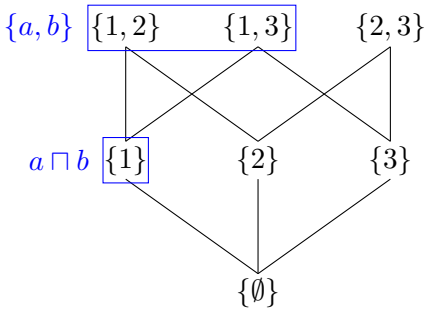
La figure 2.1 donne des exemples d'ensembles partiellement ordonnés représentés à l'aide de diagrammes de Hasse. Le premier (figure 2.1(a)) correspond à l'ensemble des parties de $\{1, 2, 3\}$ avec la relation d'inclusion des ensembles $\subseteq$. Cet ensemble partiellement ordonné est fini et possède un plus petit élément $\{\emptyset\}$ et un plus grand élément $\{1, 2, 3\}$, c'est donc un treillis complet.



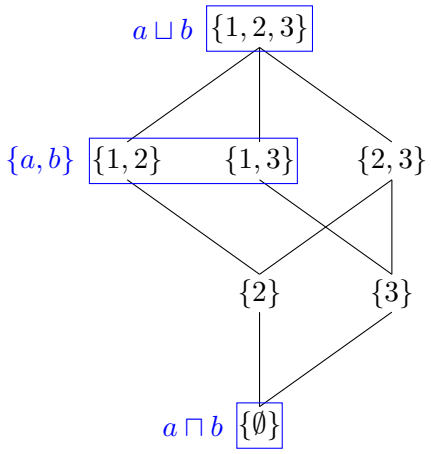
(a) $(\mathcal{P}(\{1,2,3\}), \subseteq, \cup, \cap)$



(b) Ensemble des diviseurs de 60 ordonné par la relation de divisibilité



(c) $(\mathcal{P}(\{1,2,3\}) \setminus \{1,2,3\}, \subseteq, \cup, \cap)$



(d) $(\mathcal{P}(\{1,2,3\}) \setminus \{1\}, \subseteq, \cup, \cap)$

Figure 2.1 – Exemples d'ensembles partiellement ordonnés représentés à l'aide de diagrammes de Hasse.

Par exemple, le couple $\{\{1,2\},\{1,3\}\}$ a un plus petit élément $\{1,2\} \cap \{1,3\} = \{1\}$ et un plus grand élément $\{1,2\} \cup \{1,3\} = \{1,2,3\}$.

Le deuxième (figure 2.1(b)) correspond à l'ensemble des diviseurs de 60 : $\{1,2,3,5,6,10,12,15,20,30,60\}$ muni de la relation de divisibilité. De même, cet ensemble partiellement ordonné est fini et possède un plus petit élément 1 et un plus grand élément 60, c'est donc aussi un treillis complet. Le couple $\{3,4\}$ a un plus petit élément 1 (leur plus grand commun diviseur) et un plus grand élément 12 (leur plus petit commun multiple).

En revanche, le troisième (figure 2.1(c)) n'est pas un treillis. En effet, le couple $\{\{1,2\},\{1,3\}\}$ ne possède pas de plus grand élément. De même, si on retire l'élément $\{\emptyset\}$ au treillis figure 2.1(a), l'ensemble partiellement ordonné obtenu n'est plus un treillis. Cependant, on peut retirer n'importe quel élément qui ne soit ni le plus petit ni le plus grand élément du treillis et celui-ci conserve sa propriété de treillis (figure 2.1(d)).

Exemple 2.1.4 – Il est facilement vérifié que l'ensemble partiellement ordonné $(\mathbb{I}^n, \subseteq, \cup, \cap)$ avec le plus petit élément $\bot = \emptyset$ et le plus grand élément $\top = \mathbb{F}^n$ est un treillis complet. Soient $I = I_1 \times \cdots \times I_n$ et $I' = I'_1 \times \cdots \times I'_n$ deux éléments quelconque de $\mathbb{I}^n$. Le couple $\{I, I'\}$ admet une *glb*

$$I \cap I' = [\max(\underline{I_1}, \underline{I'_1}), \min(\overline{I_1}, \overline{I'_1})] \times \cdots \times [\max(\underline{I_n}, \underline{I'_n}), \min(\overline{I_n}, \overline{I'_n})]$$

et une *lub*

$$I \cup I' = [\min(\underline{I_1}, \underline{I'_1}), \max(\overline{I_1}, \overline{I'_1})] \times \cdots \times [\min(\underline{I_n}, \underline{I'_n}), \max(\overline{I_n}, \overline{I'_n})]$$

On en déduit que tout sous-ensemble admet une *lub* et une *glb* et donc que $(\mathbb{I}^n, \subseteq, \cup, \cap)$ est un treillis. De plus comme ce treillis est fini, on en déduit que $(\mathbb{I}^n, \subseteq, \cup, \cap)$ est un treillis complet.

Les treillis sont les ensembles de base sur lesquels reposent les domaines abstraits en interprétation abstraite. Une caractéristique importante des domaines abstraits est qu'ils peuvent être liés par une correspondance de Galois. Notons qu'il n'existe pas toujours de correspondance de Galois (remarque 2.1.5). Les correspondances de Galois ont été appliquées à la sémantique par Cousot et Cousot dans [Cousot et Cousot, 1977a] de la façon suivante.

Définition 2.1.3 (Correspondance de Galois)**.** Soient $\mathcal{D}_1$ et $\mathcal{D}_2$ deux ensembles partiellement ordonnés, une correspondance de Galois est définie par deux morphismes, une abstraction $\alpha : \mathcal{D}_1 \to \mathcal{D}_2$ et une concrétisation $\gamma : \mathcal{D}_2 \to \mathcal{D}_1$ tels que :

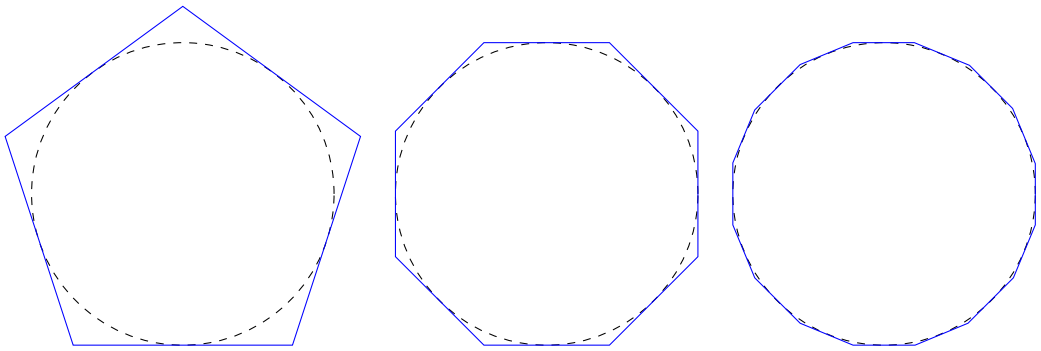
$$\forall X_1 \in \mathcal{D}_1, X_2 \in \mathcal{D}_2, \alpha(X_1) \sqsubseteq X_2 \Longleftrightarrow X_1 \sqsubseteq \gamma(X_2)$$

Une correspondance de Galois sera généralement représentée de la manière suivante :

$$\mathcal{D}_1 \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}_2$$

Remarque 2.1.3 – Une conséquence importante de cette définition est que les fonctions α et γ sont croissantes pour l'ordre $\sqsubseteq$ [Cousot et Cousot, 1992a], c'est-à-dire :

$$\forall X_1, Y_1 \in \mathcal{D}_1, X_1 \sqsubseteq Y_1 \Rightarrow \alpha(X_1) \sqsubseteq \alpha(Y_1), \text{ et}$$
$$\forall X_2, Y_2 \in \mathcal{D}_2, X_2 \sqsubseteq Y_2 \Rightarrow \gamma(X_2) \sqsubseteq \gamma(Y_2)$$



(a) (b) (c)

Figure 2.2 – Approximations d'un disque

Remarque 2.1.4 – Il découle de cette définition que $(\alpha \circ \gamma)(X_2) \sqsubseteq X_2$ et $X_1 \sqsubseteq (\gamma \circ \alpha)(X_1)$. On dit que X_2 est une approximation (ou abstraction) correcte de X_1.

Remarque 2.1.5 – Notons qu'il n'existe pas forcément d'abstraction, par exemple le domaine abstrait des polyèdres ne possède pas d'abstraction. En effet, il existe une infinité d'approximations d'un cercle à l'aide d'un polyèdre. De ce fait, il n'existe donc pas de correspondance de Galois pour les polyèdres.

La figure 2.2 montre trois différentes approximations d'un cercle avec des polyèdres. Étant donné qu'il existe une infinité de points sur le cercle, il existe une infinité de tangentes au cercle, et donc il faudrait potentiellement un polyèdre avec une infinité de côtés afin d'approximer exactement le disque.

Les correspondances de Galois sont utilisées en programmation par contraintes bien qu'elles ne soient pas nommées, notamment lors de la résolution de problèmes continus. En effet, les intervalles à bornes réelles, ne pouvant pas être représentés en machine, sont approximés par des intervalles à bornes flottantes. Le passage d'une représentation à l'autre forme une correspondance de Galois comme montré dans l'exemple ci-dessous.

Exemple 2.1.5 – Soit $\mathbb{J}$ l'ensemble des intervalles à bornes réelles. Étant donné deux ensembles partiellement ordonnés $(\mathbb{I}^n, \subset)$ et $(\mathbb{J}^n, \subset)$, il existe une correspondance de Galois

$$\mathbb{J}^n \underset{\alpha_{\mathbb{I}}}{\overset{\gamma_{\mathbb{I}}}{\leftrightarrows}} \mathbb{I}^n$$
$$\alpha_{\mathbb{I}}([x_1, y_1] \times \cdots \times [x_n, y_n]) = [\underline{x_1}, \overline{y_1}] \times \cdots \times [\underline{x_n}, \overline{y_n}]$$
$$\gamma_{\mathbb{I}}([a_1, b_1] \times \cdots \times [a_n, b_n]) = [a_1, b_1] \times \cdots \times [a_n, b_n]$$

Dans cet exemple, la fonction d'abstraction $\alpha_{\mathbb{I}}$ transforme un produit cartésien d'intervalles à bornes réelles en un produit cartésien d'intervalles à bornes flottantes en approximant chacune des bornes réelles par le flottant le plus proche utilisant la direction adéquate. La concrétisation quant à elle est directe étant donné qu'un nombre flottant est aussi un nombre réel.

2.1.2.2 Concret/Abstrait

Le domaine concret, noté $\mathcal{D}^\flat$, correspond aux valeurs que peuvent prendre les variables tout au long du programme, $\mathcal{D}^\flat = \mathcal{P}(V)$ pour V un ensemble. Calculer le domaine concret peut être

indécidable et une approximation est acceptée. L'approximation du domaine concret est appelée domaine abstrait et est notée $\mathcal{D}^\sharp$. Si une correspondance de Galois existe entre le domaine concret et le domaine abstrait : $\mathcal{D}^\flat \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$, toute fonction concrète $f^\flat$ dans $\mathcal{D}^\flat$ admet une abstraction $f^\sharp$ dans $\mathcal{D}^\sharp$ telle que

$$\forall X^\sharp \in \mathcal{D}^\sharp, (\alpha \circ f^\flat \circ \gamma)(X^\sharp) \sqsubseteq f^\sharp(X^\sharp)$$

Ceci est une conséquence de la remarque 2.1.4. De plus, la fonction abstraite $f^\sharp$ est dite optimale si et seulement si $\alpha \circ f^\flat \circ \gamma = f^\sharp$.

Par la suite on note $\mathcal{D}$ (resp. f) quand on parle d'un domaine (resp. d'une fonction) de manière générale (qu'il soit concret ou abstrait). On notera $\mathcal{D}^\flat$ et $f^\flat$ le domaine et une fonction concrète, et $\mathcal{D}^\sharp$ et $f^\sharp$ le domaine et une fonction abstraite.

2.1.2.3 Fonction de transfert

Afin d'analyser un programme, chaque ligne de code est analysée. Pour cela, chaque instruction du programme est associée à une fonction, appelée fonction de transfert, qui modifie les valeurs possibles pour les variables.

Définition 2.1.4 (Fonction de transfert)**.** Soit C une ligne de code à analyser. Une fonction de transfert $F : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ retourne, en fonction d'un ensemble d'états de départ, un ensemble d'environnements correspondant à tous les états accessibles après avoir exécuté C. On notera $\{\!|C|\!\}$ la fonction de transfert de l'instruction C et $\{\!|C|\!\}X$ lorsqu'elle est appliquée à l'ensemble d'environnements X.

Exemple 2.1.6 – Considérons une affectation $x \leftarrow expr$ où x est une variable et $expr$ une expression quelconque. La fonction de transfert $\{\!|x \leftarrow expr|\!\}$ modifie uniquement les valeurs possibles pour la variable x dans l'ensemble d'environnements de départ.

Soient deux variables x et y dont les valeurs possibles sont comprises dans $[-10, 10]$, la fonction de transfert $\{\!|x \leftarrow random(1, 10)|\!\}$ modifie seulement les valeurs possibles pour x. On a maintenant x dans $[1, 10]$ et y dans $[-10, 10]$.

Exemple 2.1.7 – Considérons maintenant une expression booléenne, la fonction de transfert ne conserve que les environnements satisfaisant l'expression booléenne.

Soit deux variables x et y dont les valeurs possibles sont comprises dans $[-10, 10]$, après l'expression booléenne $x \leq 0$ la fonction de transfert $\{\!|x \leq 0|\!\}$ filtre les valeurs de x afin de satisfaire la condition. On a maintenant x dans $[-10, 0]$ et y dans $[-10, 10]$.

Remarque 2.1.6 – Par la suite, toute fonction de transferts, F, est supposée croissante.

2.1.2.4 Point fixe

L'interprétation abstraite repose aussi sur la notion de point fixe.

Définition 2.1.5 (Point fixe)**.** Soit F une fonction, on appelle point fixe de F, un élément X tel que $F(X) = X$. On note $\mathrm{lfp}_X F$ un plus petit point fixe de F plus grand que X, et $\mathrm{gfp}_X F$ un plus grand point fixe de F plus petit que X.

Remarque 2.1.7 – Notons que, quand F est croissante, si le plus grand ou le plus petit point fixe existe, alors il est unique.

Chaque instruction du programme est associée à une fonction de transfert, le programme est donc associé à une composition de ces fonctions. Prouver que le programme est correct revient à calculer le plus petit point fixe de cette composition de fonctions. Le calcul du point fixe est nécessaire pour l'analyse des boucles par exemple. Les fonctions associées aux instructions de la boucle sont appliquées plusieurs fois, jusqu'à ce que le point fixe soit atteint.

Plusieurs schémas itératifs sont possibles. Soit $(X_1, \ldots, X_n)$ l'ensemble des ensembles d'environnements, X_i correspond à l'ensemble d'environnements à l'instruction i. On note X_i^j l'ensemble d'environnements à l'instruction i et à l'itération j. Soit F_i la fonction de transfert de l'instruction i. Le schéma itératif le plus courant est celui des itérations Jacobi calculant la valeur de X_i^j en fonction des ensembles d'environnements de l'itération précédente :

$$X_i^j = F_i(X_1^{j-1}, \ldots, X_n^{j-1})$$

Un autre schéma itératif est celui des itérations de Gauss-Seidel calculant la valeur de X_i^j en fonction des ensembles d'environnements déjà calculés à l'itération courante et des ensembles d'environnements de l'itération précédente :

$$X_i^j = F_i(X_1^j, \ldots, X_{i-1}^j, X_i^{j-1}, X_{i+1}^{j-1}, \ldots X_n^{j-1})$$

Dans les exemples de cette section, nous utiliserons les itérations de Gauss-Seidel.

Exemple 2.1.8 – Soit le programme suivant :

1: int x, y
2: $x \leftarrow 0$
3: $y \leftarrow x$
4: **tant que** $x < 10$ **faire**
5: $\quad y \leftarrow 2x$
6: $\quad x \leftarrow x + 1$
7: **fin tant que**

On a :
$X_1 = \top$
$X_2 = \{\!|x \leftarrow 0|\!\}X_1$
$X_3 = \{\!|y \leftarrow x|\!\}X_2$
$X_4 = X_3 \cup X_6$
$X_{4'} = \{\!|x < 10|\!\}X_4$
$X_5 = \{\!|y \leftarrow 2x|\!\}X_{4'}$
$X_6 = \{\!|x \leftarrow x + 1|\!\}X_5$
$X_7 = \{\!|x \geq 10|\!\}X_4$

où $X_{4'}$ est l'ensemble d'environnements si on rentre dans la boucle **tant que**. Tous les ensembles d'environnements sont initialisés à $\bot = \emptyset$, à l'exception du premier qui lui est initialisé à $\top = \mathbb{Z}^2$.

En appliquant les fonctions de transfert une première fois on obtient :

$X_1 = \top$
$X_2 = \{x = 0, y \in \mathbb{Z}\}$
$X_3 = \{x = y = 0\}$
$X_4 = \{x = y = 0\}$
$X_{4'} = \{x = y = 0\}$
$X_5 = \{x = y = 0\}$
$X_6 = \{x = 1, y = 0\}$
$X_7 = \bot$

En appliquant une deuxième fois les fonctions de transfert on obtient :

$X_1 = \top$
$X_2 = \{x = 0, y \in \mathbb{Z}\}$
$X_3 = \{x = y = 0\}$
$X_4 = \{x \in [\![0, 1]\!], y = 0\}$
$X_{4'} = \{x \in [\![0, 1]\!], y = 0\}$
$X_5 = \{x \in [\![0, 1]\!], y \in [\![0, 2]\!]\}$
$X_6 = \{x \in [\![1, 2]\!], y \in [\![0, 2]\!]\}$
$X_7 = \bot$

où $[\![a, b]\!] = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$ est l'intervalle des entiers entre a et b.

Les ensembles d'environnements X_2 et X_3 n'ont pas été modifiés et dépendent d'ensembles d'environnements qui n'ont pas non plus été modifiés, on dit alors qu'ils ont atteint leur point fixe. Contrairement à l'ensemble d'environnements X_7 qui n'a pas été modifié mais qui dépend d'ensembles d'environnements qui ont été modifiés. Les autres fonctions de transfert sont appliquées jusqu'à ce que le point fixe soit atteint. Après douze itérations, le point fixe est atteint, on a alors :

$X_1 = \top$
$X_2 = \{x = 0, y \in \mathbb{Z}\}$
$X_3 = \{x = y = 0\}$
$X_4 = \{x \in [\![0, 10]\!], y \in [\![0, 18]\!]\}$
$X_{4'} = \{x \in [\![0, 9]\!], y \in [\![0, 18]\!]\}$
$X_5 = \{x \in [\![0, 9]\!], y \in [\![0, 18]\!]\}$
$X_6 = \{x \in [\![1, 10]\!], y \in [\![0, 18]\!]\}$
$X_7 = \{x = 10, y \in [\![0, 18]\!]\}$

On peut donc affirmer que tout au long de ce programme, la variable x vaut entre 0 et 10 et la variable y vaut entre 0 et 18. De plus, après l'exécution de ce programme, la variable x vaut 10 et que la variable y vaut entre 0 et 18. Ce résultat est une approximation des valeurs possibles, en effet, à la fin de l'exécution de ce programme, y vaut toujours 18.

Analyser un programme revient à sur-approximer la valeur de l'environnement en tout point du programme. En partant du plus petit élément $\bot$, l'application successive des fonctions de transfert permet de calculer un point fixe. Ce point fixe correspond au plus petit point fixe de la composition des fonctions de transfert plus grand que $\bot$. On en déduit qu'en interprétation abstraite, calculer le plus petit point fixe $\mathrm{lfp}_\bot F$ avec $F : \mathcal{D} \to \mathcal{D}$ la composition des fonctions de transfert et $\mathcal{D}$ un ensemble partiellement ordonné ou treillis complet, permet d'analyser un programme.

Cependant, le plus petit point fixe peut ne pas être atteignable lorsque que son calcul est indécidable. Dans ce cas, une approximation de celui-ci est calculée. De plus, le calcul du point fixe peut converger très lentement, notamment lors de l'analyse d'une boucle. Afin d'accélérer le calcul de l'approximation du point fixe un opérateur d'élargissement (*widening*) est utilisé.

Définition 2.1.6 (Élargissement (*widening*))**.** L'opérateur binaire abstrait $\triangledown^\sharp$ de $\mathcal{D}^\sharp \times \mathcal{D}^\sharp$ dans $\mathcal{D}^\sharp$ est un élargissement si et seulement si :

1. $\forall X^\sharp, Y^\sharp \in \mathcal{D}^\sharp, (X^\sharp \triangledown^\sharp Y^\sharp) \sqsupseteq^\sharp X^\sharp, Y^\sharp$, et
2. pour toute chaîne $(X_i^\sharp)_{i\in\mathbb{N}}$, la chaîne croissante $(Y_i^\sharp)_{i\in\mathbb{N}}$ définie par :

$$\begin{cases} Y_0^\sharp &= X_0^\sharp \\ Y_{i+1}^\sharp &= Y_i^\sharp \triangledown^\sharp X_{i+1}^\sharp \end{cases}$$

est stable après un nombre fini d'itérations, *i.e.,* $\exists K$ tel que $Y_{K+1}^\sharp = Y_K^\sharp$.

L'opérateur d'élargissement permet de calculer un plus petit point fixe lfp. Il calcule, en fonction de plusieurs itérations précédentes, une approximation d'un plus petit point fixe tout en restant au-dessus. Cet opérateur effectue des itérations croissantes, c'est-à-dire des itérations qui font croître l'ensemble des valeurs possibles pour les variables.

Remarque 2.1.8 – Pour tout ensemble partiellement ordonné, l'opérateur d'élargissement permet de calculer une approximation du plus petit point fixe en un nombre fini d'itérations. En particulier, quand l'ensemble partiellement ordonné considéré possède une chaîne infinie croissante, un opérateur d'élargissement est nécessaire afin d'atteindre une approximation du plus petit point fixe en un nombre fini d'itérations. Il assure ainsi la terminaison.

Remarque 2.1.9 – Notons que si l'opérateur d'élargissement et les fonctions de transferts sont corrects, alors la correction est assurée.

Remarque 2.1.10 – Si le résultat obtenu avec l'élargissement respecte les spécifications données, alors le programme est correct. En effet, si un ensemble d'environnements X respecte les spécifications alors un ensemble d'environnements plus petit $X' \sqsubseteq X$ les respecte aussi. Cependant, nous ne pouvons rien affirmer si le résultat obtenu avec l'élargissement ne respecte pas les spécifications.

Exemple 2.1.9 – Reprenons le programme de l'exemple 2.1.8. Utilisons cette fois-ci un opérateur d'élargissement pour l'analyse de la boucle. On redéfinit donc l'environnement correspondant à la condition de la boucle :

$$X_4 = X_4 \triangledown^\sharp (X_3 \cup X_6)$$

avec $\triangledown^\sharp$ un opérateur d'élargissement tel que :

$$[\![a,b]\!] \triangledown^\sharp [\![c,d]\!] = \left[\!\!\left[\begin{cases} a & \text{si } a \leq c \\ -\infty & \text{sinon} \end{cases}, \begin{cases} b & \text{si } b \geq d \\ +\infty & \text{sinon} \end{cases} \right]\!\!\right] \text{ et,}$$
$$X \triangledown^\sharp \bot = X \text{ et } \bot \triangledown^\sharp X = X$$

Cet opérateur met la borne supérieure des valeurs possibles pour une variable x à $+\infty$ si entre deux itérations l'intervalle de la variable x croît vers $+\infty$. Inversement, la borne inférieure est mise à $-\infty$ si l'intervalle de la variable décroît vers $-\infty$.

Regardons l'évolution de l'environnement X_4. À la première itération, on a :

$$\begin{aligned} X_4 &= \bot \triangledown^\sharp \{x = y = 0\} \\ &= \{x = y = 0\} \end{aligned}$$

Ce qui correspond au résultat obtenu après la première itération dans l'exemple 2.1.8. Après la deuxième itération, on obtient :

$$X_4 = \{x = y = 0\} \nabla^\sharp \{x \in [\![0,1]\!], y = 0\}$$
$$= \{x \in [\![0,+\infty]\!], y = 0\}$$

L'opérateur d'élargissement déduit de l'itération précédente que x croît dans la boucle et donc modifie sa borne supérieure. De même, à l'itération suivante, l'opérateur d'élargissement déduit que y croît dans la boucle et modifie sa borne supérieure. Après deux autres itérations, le point fixe est atteint, on a alors :

$$X_1 = \top$$
$$X_2 = \{x = 0, y \in \mathbb{Z}\}$$
$$X_3 = \{x = y = 0\}$$
$$X_4 = \{x \in [\![0,+\infty]\!], y \in [\![0,+\infty]\!]\}$$
$$X_{4'} = \{x \in [\![0,9]\!], y \in [\![0,+\infty]\!]\}$$
$$X_5 = \{x \in [\![0,9]\!], y \in [\![0,18]\!]\}$$
$$X_6 = \{x \in [\![1,10]\!], y \in [\![0,18]\!]\}$$
$$X_7 = \{x \in [\![10,+\infty]\!], y \in [\![0,+\infty]\!]\}$$

On peut donc affirmer qu'après l'exécution de l'instruction 7, la variable x est supérieure ou égale à 10 et que la variable y est positive, ce qui est moins précis que le résultat obtenu dans l'exemple 2.1.8. Cependant, seulement quatre itérations sont nécessaires pour atteindre le point fixe, ce qui est trois fois moins que sans opérateur d'élargissement.

Remarque 2.1.11 – Notons que le nombre d'itérations nécessaires avec l'opérateur d'élargissement ne dépend plus des constantes du programme. En remplaçant la constante 10 par 100 ou 1000, les itérations sans élargissement vont converger de plus en plus lentement, alors que celles avec élargissement ne seront pas affectées.

Cet opérateur peut générer une très grande sur-approximation du point fixe, comme montré dans l'exemple 2.1.9 où la seule information que l'on obtient pour y est qu'il est positif alors que dans l'exemple 2.1.8 on a pu inférer qu'il était inférieur ou égal à 18. De même, la variable x est supérieure à 10 alors que dans l'exemple 2.1.8 elle valait exactement 10. Afin de raffiner cette approximation, un opérateur de rétrécissement (*narrowing*) peut être utilisé.

Définition 2.1.7 (Rétrécissement (*narrowing*))**.** L'opérateur binaire abstrait $\triangle^\sharp$ de $\mathcal{D}^\sharp \times \mathcal{D}^\sharp$ dans $\mathcal{D}^\sharp$ est un rétrécissement si et seulement si :

1. $\forall X^\sharp, Y^\sharp \in \mathcal{D}^\sharp, (X^\sharp \sqcap^\sharp Y^\sharp) \sqsubseteq^\sharp (X^\sharp \triangle^\sharp Y^\sharp) \sqsubseteq^\sharp X^\sharp$, et
2. pour toute chaîne $(X_i^\sharp)_{i\in\mathbb{N}}$, la chaîne $(Y_i^\sharp)_{i\in\mathbb{N}}$ définie par :

$$\begin{cases} Y_0^\sharp &= X_0^\sharp \\ Y_{i+1}^\sharp &= Y_i^\sharp \triangle^\sharp X_{i+1}^\sharp \end{cases}$$

est stable après un nombre fini d'itérations, *i.e.,* $\exists K$ tel que $Y_{K+1}^\sharp = Y_K^\sharp$.

Tout comme l'élargissement, le rétrécissement calcule en fonction des itérations précédentes une approximation d'un point fixe, cependant le point fixe calculé n'est pas forcément le plus petit point fixe lfp. En effet, l'opérateur de rétrécissement calcule une approximation d'un point fixe (gfp ou lfp) tout en restant au-dessus. Contrairement à l'élargissement, le rétrécissement effectue des itérations décroissantes, c'est-à-dire des itérations qui font décroître l'ensemble des valeurs possibles pour les variables.

Exemple 2.1.10 – Reprenons le programme de l'exemple 2.1.8. Dans l'exemple précédent (exemple 2.1.9), nous avions utilisé un opérateur d'élargissement afin d'accélérer le calcul du point fixe, ce qui a généré une sur-approximation. Partant du résultat obtenu dans l'exemple 2.1.9, nous utilisons un opérateur de rétrécissement afin de raffiner ce résultat. L'environnement correspondant à la condition de la boucle est donc redéfini :

$$X_4 = X_4 \vartriangle^\sharp (X_3 \cup X_6)$$

avec $\vartriangle^\sharp$ un opérateur d'élargissement tel que :

$$[\![a,b]\!] \vartriangle^\sharp [\![c,d]\!] = \left[\!\!\left[\begin{cases} c & \text{si } a = -\infty \\ a & \text{sinon} \end{cases}, \begin{cases} d & \text{si } b = +\infty \\ b & \text{sinon} \end{cases} \right]\!\!\right] \text{ et,}$$
$$X \vartriangle^\sharp \bot = \bot \text{ et } \bot \vartriangle^\sharp X = \bot$$

Cet opérateur permet de réduire uniquement les bornes infinies. Les bornes peuvent donc être réduites au plus une seule fois.

Regardons l'évolution de l'environnement X_4, à la première itération, on a :

$$\begin{aligned} X_4 &= \{x \in [\![0,+\infty]\!], y \in [\![0,+\infty]\!]\} \vartriangle^\sharp \{x \in [\![0,10]\!], y \in [\![0,18]\!]\} \\ &= \{x \in [\![0,10]\!], y \in [\![0,18]\!]\} \end{aligned}$$

Ce qui correspond au résultat obtenu après la onzième itération dans l'exemple 2.1.8. Après seulement deux itérations, le point fixe est atteint et est le même que dans l'exemple 2.1.8. En utilisant les opérateurs d'élargissement et de rétrécissement, on atteint le point fixe en six itérations, ce qui est deux fois moins que quand ils ne sont pas utilisés. Notons que dans cet exemple le point fixe atteint est le même dans les deux cas. Dans le cas général, si un point fixe est atteint en un nombre fini d'itérations sans utiliser les opérateurs d'élargissement et de rétrécissement alors celui-ci peut être plus précis que celui obtenu en utilisant ces opérateurs.

De plus, notons que partant de l'exemple 2.1.9, appliquer les fonctions de transfert telles que définies dans l'exemple 2.1.8 aurait généré des itérations décroissantes et ainsi permis de réduire les environnements sans utiliser l'opérateur de rétrécissement.

Un schéma d'approximation du point fixe utilisant un opérateur d'élargissement et de rétrécissement est donné figure 2.3. En partant du plus petit élément $\bot$, deux itérations croissantes sont effectuées, l'ensemble des valeurs possibles pour les variables croît. Puis l'opérateur d'élargissement $\triangledown^\sharp$ est utilisé, faisant passer l'environnement au-dessus du plus petit point fixe lfp. Notons que l'élargissement peut aussi faire passer l'environnement au-dessus du plus grand point fixe gfp. L'opérateur de rétrécissement $\vartriangle^\sharp$ peut ensuite être utilisé afin de raffiner l'approximation obtenue. Cette itération décroissante raffine l'approximation tout en restant au-dessus du point fixe considéré, qui est ici le plus petit point fixe lfp. En résumé, l'opérateur d'élargissement permet, en partant d'un environnement en-dessous d'un point fixe, de passer au-dessus en agrandissant l'ensemble des valeurs des variables, alors que l'opérateur de rétrécissement, étant donné un environnement au-dessus d'un point fixe, le réduit tout en restant au-dessus de ce point fixe.

De nombreux travaux [Bagnara *et al.*, 2005a, Bagnara *et al.*, 2005b, Bagnara *et al.*, 2006, Simon et King, 2006, D'Silva, 2006, Cortesi, 2008, Monniaux, 2009, Simon et Chen, 2010] se sont intéressés au calcul fin de l'élargissement alors que le rétrécissement reste moins étudié [Cousot et Cousot, 1992b, Moreno-Navarro *et al.*, 1993, Alpuente *et al.*, 1993, Hickey et Ju, 1997, Cortesi et Zanioli, 2011]. Il existe même des domaines abstraits pour lesquels il n'existe pas de rétrécissement, les polyèdres en sont un exemple. Cette différence d'intérêt

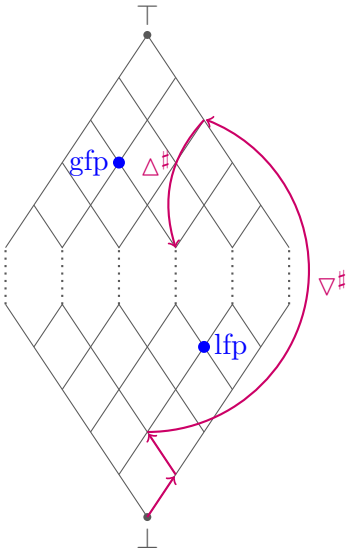


Figure 2.3 – Schéma d'approximation d'un point fixe utilisant un élargissement et rétrécissement.

entre l'élargissement et le rétrécissement peut être due à trois raisons : premièrement le rétrécissement n'est pas nécessaire pour assurer la correction contrairement à l'élargissement qui est quant à lui indispensable. En effet, l'opérateur d'élargissement permet de calculer une approximation du plus petit point fixe en un nombre fini d'itérations et ce même quand l'ensemble partiellement ordonné considéré possède une chaîne infinie croissante. Ainsi si le résultat obtenu avec l'élargissement respecte les spécifications données, alors le programme considéré est correct. Cependant, si le résultat obtenu avec l'élargissement ne respecte pas les spécifications données, cela ne veut pas forcément dire que le programme ne les respecte pas, mais peut être dû à une trop grande sur-approximation. De ce fait, il est nécessaire que l'opérateur d'élargissement soit correctement conçu afin de minimiser la sur-approximation et ainsi éviter les faux négatifs.

Deuxièmement, appliquer les fonctions de transfert sans opérateur d'élargissement ni d'opérateur de rétrécissement permet d'effectuer un certain nombre d'itérations décroissantes et est parfois suffisant pour améliorer la précision après un élargissement [Bertrane *et al.*, 2010]. C'est le cas pour le programme utilisé pour illustrer cette section (exemple 2.1.8). Après l'élargissement, l'application des fonctions de transfert sans élargissement permet de réduire les environnements et d'obtenir une approximation du plus petit point fixe plus précise. Dans cet exemple, le point fixe atteint est le même que celui obtenu avec l'opérateur de rétrécissement. De plus, la terminaison est garantie car le nombre d'itérations décroissantes est fini et généralement fixé à l'avance.

Finalement, quand cette technique n'est pas suffisante, le rétrécissement n'aide pas, en pratique, à améliorer la précision et des méthodes autres que les itérations décroissantes doivent être considérées telles que celle présentée par Halbwachs et Henry dans [Halbwachs et Henry, 2012]. Après une série d'itérations croissantes et décroissantes, le point fixe est atteint, cependant la précision atteinte n'est pas satisfaisante. Dans cet article les auteurs proposent de relancer une série d'itérations en ne conservant qu'une partie du résultat précédent, c'est-à-dire de ne conserver pour chacun des environnements les valeurs possibles que pour un sous-ensemble de variables uniquement. Les expériences effectuées montrent une réduction de la sur-approximation de l'opérateur d'élargissement.

Cependant, quand la sur-approximation est due à des fonctions de transfert imprécises, une autre solution est celle des itérations locales proposée par Granger en 1992 [Granger, 1992]. Une série d'itérations décroissantes est appliquée plusieurs fois. Cette méthode est présentée plus en détails dans la section suivante.

2.1.2.5 Itérations locales

La fonction de transfert abstraite $F^\sharp \sqsupseteq (\alpha \circ F^\flat \circ \gamma)$ ne permet pas toujours de calculer efficacement le plus petit domaine abstrait englobant l'expression considérée, et cela même quand elle est optimale $(\gamma \circ F^\sharp \circ \alpha) = F^\flat$. Afin de calculer efficacement le résultat des fonctions de transfert, des opérateurs de clôture inférieure sont généralement utilisés [Granger, 1992].

Définition 2.1.8 (Opérateur de clôture inférieure)**.** Un opérateur $\rho : \mathcal{D} \to \mathcal{D}$ est un opérateur de clôture inférieure si et seulement si ρ est :

1. croissant, $\forall X, Y \in \mathcal{D}, X \sqsubseteq Y \Rightarrow \rho(X) \sqsubseteq \rho(Y)$
2. contractant, $\forall X \in \mathcal{D}, \rho(X) \sqsubseteq X$, et
3. idempotant, $\forall X \in \mathcal{D}, \rho(X) = (\rho \circ \rho)X$.

Les travaux de Granger [Granger, 1992] montrent qu'en itérant plusieurs fois les opérateurs de clôture, il est possible d'améliorer les calculs du plus petit point fixe. En effet, étant donnée une abstraction correcte $\rho^\sharp$ de $\rho^\flat$, la limite $Y^\sharp_\delta$ d'une séquence de l'opérateur de rétrécissement, $Y^\sharp_0 = X^\sharp$, $Y^\sharp_{i+1} = Y^\sharp_i \triangle \rho^\sharp(Y^\sharp_i)$ est une abstraction de $(\rho^\flat \circ \gamma)(X^\sharp)$. Notons cependant que si $\rho^\sharp$ n'est pas une abstraction optimale de $\rho^\flat$, $Y^\sharp_\delta$ peut être significativement plus précis que $\rho^\sharp(X^\sharp)$. Une application pertinente est celle de l'analyse d'une conjonction de tests complexes $C_1 \wedge \cdots \wedge C_p$ où chaque test atomique C_i a une modélisation abstraite $\rho^\sharp_i$. Notons que généralement, $\rho^\sharp = \rho^\sharp_1 \circ \cdots \circ \rho^\sharp_p$ n'est pas optimal et cela même quand chaque $\rho^\sharp_i$ est optimal.

Les opérateurs de clôture peuvent être reformulés comme un point fixe. Ceci permet d'unifier l'utilisation des rétrécissements et fait apparaître une similarité dans les calculs itératifs. Étant donné un élément X, ρ calcule le plus grand point fixe plus petit que X, c'est-à-dire $\rho(X) = \mathrm{gfp}_X\, \rho$. Les itérations locales peuvent être utilisées à tout moment de l'analyse et pas uniquement après un élargissement.

Étant donné un programme à analyser, un domaine abstrait est choisi afin de représenter au mieux les propriétés du programme. Il existe différents types de domaines abstraits. Une courte présentation des domaines abstraits est donnée dans la prochaine sous-section.

2.1.2.6 Domaines abstraits

Les domaines abstraits jouent un rôle central en interprétation abstraite. Du fait de l'importance des propriétés et variables numériques dans un programme, de nombreux domaines abstraits numériques sont développés. Les intervalles [Cousot et Cousot, 1977a] et les polyèdres [Cousot et Halbwachs, 1978] font partie des principaux domaines abstraits numériques. Ces dernières années ont vu le développement d'un grand nombre de nouveaux domaines abstraits capturant d'autres propriétés, tels que les octogones [Miné, 2006], les ellipsoïdes [Feret, 2004], les octaèdres [Clarisó et Cortadella, 2004] ou encore les variétés [Rodriguez-Carbonell et Kapur, 2004]. De plus, des bibliothèques de support pour les domaines abstraits telles qu'Apron [Jeannet et Miné, 2009] ont été développées. Ces nouveaux

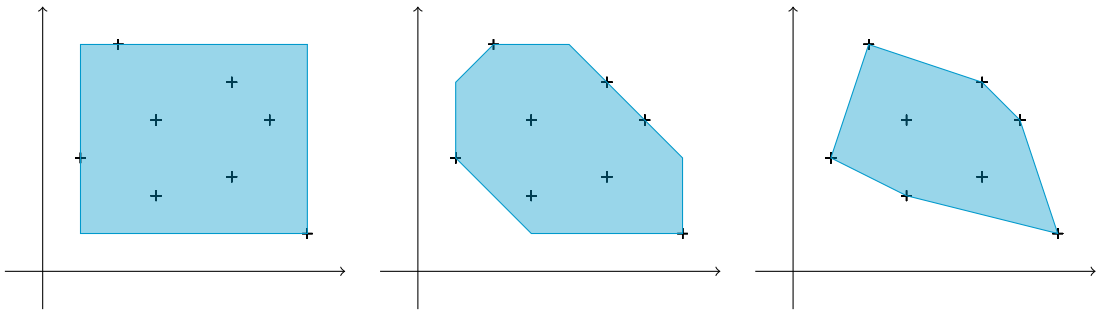
domaines peuvent manipuler toutes sortes de variables numériques, dont les entiers mathématiques, les rationnels, les réels, les entiers et les flottants en machine [Miné, 2012]. Ils peuvent exprimer des relations entre différents types de variables [Miné, 2004], et entre les variables numériques et booléennes [Bertrane *et al.*, 2010]. En outre, des combinateurs génériques de domaines peuvent être appliqués afin de construire de nouveaux domaines à partir des domaines abstraits existants. Les complétions disjonctives [Cousot et Cousot, 1992a], les produits réduits [Cousot et Cousot, 1979, Cousot *et al.*, 2007, Cousot *et al.*, 2011] et le partitionnement [Bourdoncle, 1992, Rival et Mauborgne, 2007] en font partie. Ces domaines abstraits sont des briques des analyseurs statiques, qui généralement utilisent simultanément différents domaines abstraits. Il est crucial de choisir (ou de concevoir) avec soin les domaines abstraits pour un problème donné. Ceci est généralement fait à la main en se basant sur les propriétés qui doivent être déduites.

Ces diverses représentations sont regroupées suivant leur niveau d'expressivité. Plus les propriétés exprimables par un domaine abstrait sont complexes, et plus celui-ci sera précis. Les propriétés exprimables par un domaine abstrait correspondent aux liens entre les différentes variables que le domaine abstrait peut représenter. En d'autres termes, plus les liens représentables entre les variables sont complexes et plus le domaine abstrait est précis. Cette précision vient généralement au prix d'un certain coût en terme de temps de calcul. On distingue trois grandes familles : les domaines abstraits non-relationnels, relationnels et faiblement relationnels. Dans la première famille, les propriétés sont exprimées sur une seule variable, les domaines abstraits de cette famille sont donc les moins expressifs. Afin de représenter plusieurs variables, on utilisera un produit cartésien du domaine choisi. Le domaine le plus connu dans cette famille est celui des intervalles [Cousot et Cousot, 1976] où à chaque variable correspond un intervalle de valeurs possibles pour cette variable.

Dans les deux autres familles de domaines abstraits, on peut, comme leur nom l'indique, avoir des relations entre les variables. Les domaines relationnels sont très expressifs et existent en de nombreuses variétés. On y trouve les polyèdres [Cousot et Halbwachs, 1978], qui expriment des relations linéaires entre les variables, et les ellipsoïdes [Feret, 2004], qui expriment des polynômes du second degré de type ellipses. La dernière famille, les domaines faiblement relationnels, se situe entre les deux familles précédemment nommées. Elle est composée de domaines abstraits offrant un compromis entre la précision et le temps de calcul. Ces domaines faiblement relationnels permettent de représenter certaines des propriétés exprimables possibles entre variables. Ils ont été introduits en 2004 par Miné [Miné, 2004]. On trouve dans cette famille, entre autres, les zones exprimant des inégalités de la forme $v_1 - v_2 \leq c$ avec v_1 et v_2 des variables et c une constante, et les octogones qui expriment des inégalités de la forme $\pm v_1 \pm v_2 \leq c$.

La figure 2.4 montre pour un même ensemble de points un exemple de domaine abstrait dans chacune des trois catégories citées précédemment. Le premier correspond au domaine non-relationnel des intervalles (2.4(a)), le deuxième au domaine faiblement relationnel des octogones (2.4(b)) et le dernier au domaine relationnel des polyèdres (2.4(c)). On voit bien que plus les propriétés représentables par le domaine abstrait sont complexes et plus celui-ci est précis. Les polyèdres sont plus précis que les octogones, qui sont eux plus précis que les intervalles.

Un domaine abstrait correspond à un ensemble calculable en machine $\mathcal{D}^\sharp$ muni de l'ordre partiel $\sqsubset^\sharp$ permettant de manipuler des éléments calculables en machine. De ce fait tous les opérateurs dans $\mathcal{D}^\flat$ doivent avoir une abstraction dans $\mathcal{D}^\sharp$. Ces différents opérateurs sont listés ci-dessous.



(a) Intervalles (b) Octogones (c) Polyèdres

Figure 2.4 – Différents domaines abstraits représentant le même ensemble de points

Opérations sur les domaines abstraits :

- une fonction de concrétisation $\gamma : \mathcal{D}^\sharp \to \mathcal{D}^\flat$, et si elle existe une fonction d'abstraction $\alpha : \mathcal{D}^\flat \to \mathcal{D}^\sharp$ formant une correspondance de Galois $\mathcal{D}^\flat \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$,
- un plus petit élément $\bot^\sharp$ et un plus grand élément $\top^\sharp$ tels que $\gamma(\bot^\sharp) = \emptyset$ et $\gamma(\top^\sharp) = V$ avec $\mathcal{D}^\flat = \mathcal{P}(V)$,
- des algorithmes efficaces pour le calcul des fonctions de transfert,
- des algorithmes efficaces pour le calcul de l'union $\cup^\sharp$ et l'intersection $\cap^\sharp$,
- des algorithmes efficaces pour le calcul de l'élargissement $\triangledown^\sharp$ si $\mathcal{D}^\sharp$ a une chaîne infinie croissante, et
- s'il existe et que $\mathcal{D}^\sharp$ possède une chaîne infinie décroissante, des algorithmes efficaces pour l'opérateur de rétrécissement $\triangle^\sharp$.

Notons qu'il peut ne pas exister de fonction d'abstraction et d'opérateur de rétrécissement. De plus, bien qu'assez rare, il peut ne pas exister d'opérateur d'intersection. Comme précisé remarque 2.1.5, le domaine abstrait des polyèdres ne possède pas de fonction d'abstraction et donc pas de correspondance de Galois. De plus, les polyèdres ne possèdent pas non plus d'opérateur de rétrécissement. Bien qu'ils puissent ne pas exister, ces différents opérateurs sont utiles de façon à n'avoir qu'une seule représentation possible pour un domaine concret donné (abstraction) et améliorer la sur-approximation lors du calcul du point fixe (rétrécissement).

2.1.3 Conclusion

Les sections précédentes présentent succinctement l'interprétation abstraite, les principales définitions et notions nécessaire par la suite sont données. Pour une présentation plus détaillée nous conseillons au lecteur de lire par exemple [Cousot et Cousot, 1977a].

En résumé, l'interprétation abstraite analyse automatiquement les programmes afin de certifier que ceux-ci ne contiennent pas d'erreurs. Cette analyse repose sur le calcul d'une sémantique du programme, c'est-à-dire l'ensemble de toutes les valeurs possibles des variables du programme. Calculer directement la sémantique concrète peut prendre beaucoup de temps voire être impossible, cette sémantique est donc abstraite. En d'autres termes seules certaines caractéristiques seront conservées afin de simplifier et ainsi accélérer les calculs, et ce tout en restant correct. Pour chacun des opérateurs existants dans le concret, un opérateur abstrait est défini et des algorithmes efficaces sont mis en place. La sémantique abstraite est approximée à l'aide d'un domaine abs-

trait. Il existe de très nombreux domaines abstraits en interprétation abstraite, offrant différents compromis entre temps de calcul et précision.

Après ce tour d'horizon de la théorie générale et des techniques en interprétation abstraite, la section suivante présente succinctement la programmation par contraintes, ses principes, challenges scientifiques et techniques de résolution.

2.2 Programmation par Contraintes

La programmation par contraintes (PPC) fut introduite en 1974 par Montanari [Montanari, 1974], elle repose sur l'idée que les problèmes combinatoires peuvent être exprimés comme la conjonction de formules de la logique de premier ordre, appelées contraintes. Un problème est dit combinatoire lorsqu'il contient un très grand nombre de combinaisons de valeurs possibles. Prenons l'exemple de la résolution d'une grille de Sudoku : chaque case ne contenant pas déjà une valeur peut valoir entre 1 et 9. Le nombre de toutes les grilles possibles, correspondant à l'énumération de toutes les valeurs possibles pour toutes les cases vides, est très grand. Si seulement 10 cases sont vides, ce nombre vaut déjà plus de 3 millions (il est égal à 9^k où k est le nombre de cases vides, car pour chacune des cases vides, on a 9 valeurs possibles). C'est un problème combinatoire. Énumérer toutes ces grilles afin de vérifier si elles correspondent à une solution prendrait trop de temps. La programmation par contraintes met en place des techniques afin de résoudre efficacement ce type de problèmes combinatoires. Comme décrit par E. Freuder, en programmation par contraintes, l'utilisateur spécifie le problème et l'ordinateur le résout :

> *"Constraint Programming represents 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."*
>
> – Eugene C. Freuder [Freuder, 1997].

Afin de définir le problème, l'utilisateur définit les contraintes de celui-ci, c'est-à-dire les spécifications du problème. Une contrainte représente une relation combinatoire spécifique du problème [Rossi *et al.*, 2006]. Toujours avec l'exemple du Sudoku, le fait que chaque chiffre compris entre 1 et 9 apparaisse exactement une fois dans chaque ligne, est une contrainte. Chaque contrainte vient avec des opérateurs *ad hoc* exploitant la structure exprimée par la contrainte de façon à réduire la combinatoire. Les contraintes sont ensuite combinées dans des algorithmes de résolution. La plupart des efforts de recherche en programmation par contraintes se concentre sur la définition et l'amélioration des contraintes [Fages et Lorca, 2011], le développement d'algorithmes efficaces [Bessiere *et al.*, 2011, Petit *et al.*, 2011], ou le paramétrage fin des solveurs [Ansótegui *et al.*, 2009, Arbelaez *et al.*, 2009]. Il existe de très nombreuses contraintes globales [Bessière et van Hentenryck, 2003, des contraintes globales,] et chacune vient avec un algorithme réduisant sa combinatoire.

La programmation par contraintes offre des techniques efficaces pour la résolution combinatoire avec de nombreuses applications de la vie réelle, parmi lesquelles on trouve l'ordonnancement [Grimes et Hebrard, 2011, Hermenier *et al.*, 2011], la cryptographie [Ramamoorthy *et al.*, 2011], la conception d'emploi du temps [Stølevik *et al.*, 2011], la musique [Truchet et Assayag, 2011], la biologie [Perriquet et Barahona, 2009], ou encore la conception de réseaux à fibres optiques [Pelleau *et al.*, 2009].

Cependant il existe encore des limitations à l'utilisation de la programmation par contraintes, une des plus importantes étant le manque d'algorithmes de résolution capable de traiter à

la fois les variables discrètes et continues. En 2009, des travaux [Berger et Granvilliers, 2009, Chabert *et al.*, 2009] ont proposé des méthodes afin de résoudre des problèmes mixtes, cependant les techniques utilisées transforment les variables afin de n'avoir que des variables discrètes ou continues.

Finalement, la programmation par contraintes n'offre qu'un choix limité pour les représentations des variables qui sont des produits cartésiens d'un certain ensemble de base, entier ou réel.

Dans la suite nous présentons les bases de la programmation par contraintes nécessaires à la compréhension de ce manuscrit. Nous nous concentrerons uniquement sur les aspects qui nous intéressent. Pour une présentation plus exhaustive, voir [Rossi *et al.*, 2006].

2.2.1 Principes

En programmation par contraintes, les problèmes sont modélisés dans un format spécifique, celui des Problèmes de Satisfaction de Contraintes (*Constraint Satisfaction Problem*, CSP). Les variables peuvent être soit entières soit réelles.

Définition 2.2.1 (Problème de satisfaction de contraintes)**.** Un problème de satisfaction de contraintes (CSP) est défini par un ensemble de variables $(v_1 \dots v_n)$ prenant leurs valeurs dans les domaines $(\hat{D}_1 \dots \hat{D}_n)$ et un ensemble de contraintes $(C_1 \dots C_p)$. Une contrainte est une relation sur un sous-ensemble de variables.

Le domaine D_i correspond à l'ensemble des valeurs possibles pour la variable v_i. L'ensemble de toutes les affectations possibles pour les variables $D = D_1 \times \cdots \times D_n$ est appelé *espace de recherche*. L'espace de recherche étant modifié tout au long de la résolution, on notera l'espace de recherche initial $\hat{D} = \hat{D}_1 \times \cdots \times \hat{D}_n$. Les problèmes peuvent être soit discrets ($\hat{D} \subseteq \mathbb{Z}^n$) soit continus ($\hat{D} \subseteq \mathbb{R}^n$). Les domaines sont quant à eux toujours bornés dans $\mathbb{R}$ ou $\mathbb{Z}$.

Les solutions du CSP sont les éléments de $\hat{D}$ satisfaisant les contraintes. On note S l'ensemble des solutions, $S = \{(s_1 \dots s_n) \in \hat{D} \,|\, \forall i \in [\![1, p]\!],\ C_i(s_1 \dots s_n)\}$. Pour une contrainte C, on notera $S_C = \{(s_1 \dots s_n) \in \hat{D} \,|\, C(s_1 \dots s_n)\}$ l'ensemble des solutions de C.

Exemple 2.2.1 – Considérons la grille de Sudoku de 4×4 suivante :

	3	1	
			4
1	2		
			1

Un modélisation possible est d'associer à chaque case une variable comme suit :

v_1	v_2	v_3	v_4
v_5	v_6	v_7	v_8
v_9	v_{10}	v_{11}	v_{12}
v_{13}	v_{14}	v_{15}	v_{16}

Chaque variable a une valeur entre 1 et 4, on a donc $\hat{D}_1 = \hat{D}_2 = \cdots = \hat{D}_{16} = [\![1, 4]\!]$. Pour spécifier qu'une case a une valeur fixée, on peut soit modifier son domaine $\hat{D}_3 = \{1\}$, soit ajouter la contrainte $v_3 = 1$. Dans un Sudoku, chaque ligne, chaque colonne et chaque bloc doit contenir exactement une fois chaque chiffre. Pour la première ligne on a donc les contraintes suivantes : $v_1 \neq v_2, v_1 \neq v_3, v_1 \neq v_4, v_2 \neq v_3, v_2 \neq v_4,$ et $v_3 \neq v_4$. Spécifier que toutes les variables d'un ensemble sont différentes peut être réalisé à l'aide de la contrainte globale $\mathrm{alldifferent}$: pour la première ligne on peut donc remplacer les six contraintes listées précédemment par la contrainte $\mathrm{alldifferent}(v_1, v_2, v_3, v_4)$.

On obtient donc un CSP avec seize variables $(v_1 \dots v_{16})$, prenant leurs valeurs dans les domaines $\hat{D}_2 = \{3\}, \hat{D}_3 = \hat{D}_9 = \hat{D}_{16} = \{1\}, \hat{D}_8 = \{4\}, \hat{D}_{10} = \{2\}$, tous les autres étant égaux à $[\![1, 4]\!]$, et douze contraintes, $(C_1 \dots C_4)$ correspondent aux lignes, $(C_5 \dots C_8)$ aux colonnes et $(C_9 \dots C_{12})$ aux blocs :

$C_1 : \mathrm{alldifferent}(v_1, v_2, v_3, v_4)$	$C_5 : \mathrm{alldifferent}(v_1, v_5, v_9, v_{13})$	$C_9 : \mathrm{alldifferent}(v_1, v_2, v_5, v_6)$
$C_2 : \mathrm{alldifferent}(v_5, v_6, v_7, v_8)$	$C_6 : \mathrm{alldifferent}(v_2, v_6, v_{10}, v_{14})$	$C_{10} : \mathrm{alldifferent}(v_3, v_4, v_7, v_8)$
$C_3 : \mathrm{alldifferent}(v_9, v_{10}, v_{11}, v_{12})$	$C_7 : \mathrm{alldifferent}(v_3, v_7, v_{11}, v_{15})$	$C_{11} : \mathrm{alldifferent}(v_9, v_{10}, v_{13}, v_{14})$
$C_4 : \mathrm{alldifferent}(v_{13}, v_{14}, v_{15}, v_{16})$	$C_8 : \mathrm{alldifferent}(v_4, v_8, v_{12}, v_{16})$	$C_{12} : \mathrm{alldifferent}(v_{11}, v_{12}, v_{15}, v_{16})$

Les solutions de ce CSP correspondent bien aux grilles de Sudoku correctement remplies. Pour cet exemple il existe une unique solution :

4	3	1	2
2	1	3	4
1	2	4	3
3	4	2	1

2.2.1.1 Représentation des domaines

En fonction du type des variables, le domaine peut être stocké comme un intervalle ou un ensemble de points. Dans le cas d'une variable entière v_i, le domaine peut être représenté comme un sous-ensemble de points de D_i ou comme un sous-intervalle de D_i.

Définition 2.2.2 (Produit cartésien entier)**.** Soient $v_1, \dots, v_n$ des variables de domaines discrets finis $\hat{D}_1, \dots, \hat{D}_n$. On appelle *produit cartésien entier* tout produit cartésien d'ensembles d'entiers dans $\hat{D}$. Les produits cartésiens entiers de $\hat{D}$ forment un treillis fini pour l'inclusion :

$$\mathbb{S} = \left\{ \prod_i X_i \,|\, \forall i, X_i \subseteq \hat{D}_i \right\}$$

Définition 2.2.3 (Boîte entière)**.** Soient $v_1, \dots, v_n$ des variables de domaines discrets finis $\hat{D}_1, \dots, \hat{D}_n$. On appelle *boîte entière* tout produit cartésien d'intervalles entiers dans $\hat{D}$. Les boîtes entières de $\hat{D}$ forment un treillis fini pour l'inclusion :

$$\mathbb{IB} = \left\{ \prod_i [\![a_i, b_i]\!] \,|\, \forall i, [\![a_i, b_i]\!] \subseteq \hat{D}_i, a_i \leq b_i \right\} \cup \emptyset$$

Pour les variables réelles, les réels n'étant pas représentables en machine, leurs domaines sont représentés par un intervalle réel à bornes flottantes.

Définition 2.2.4 (Boîte)**.** Soient $v_1, \ldots, v_n$ des variables de domaines bornés continus $\hat{D}_1, \ldots, \hat{D}_n \in \mathbb{I}$. On appelle *boîte* tout produit cartésien d'intervalles à bornes flottantes dans $\hat{D}$. Les boîtes de $\hat{D}$ forment un treillis fini pour l'inclusion :

$$\mathbb{B} = \left\{ \prod_i I_i \,|\, \forall i, I_i \in \mathbb{I}, I_i \subseteq \hat{D}_i \right\} \cup \emptyset$$

2.2.1.2 Satisfaction d'une contrainte

Pour les variables entières, étant donnée une instanciation des variables, une contrainte répond *vrai* si cette affectation des variables satisfait la contrainte et *faux* sinon.

Exemple 2.2.2 – Soient v_1 et v_2 deux variables entières de domaines $D_1 = D_2 = [\![0, 2]\!]$. Soit $C : v_1 + v_2 \leq 3$ une contrainte. Étant donné l'affectation de la variable v_1 à 2 et de la variable v_2 à 0, la contrainte C répond vrai, c'est-à-dire, $C(2, 0)$ est vrai. En revanche, $C(2, 2)$ est faux.

Dans le cas des domaines réels, une caractéristique importante est que les contraintes peuvent répondre :

— *vrai*, si la boîte ne contient que des solutions ;
— *faux*, si la boîte ne contient aucune solution ;
— et *peut-être*, quand on ne peut déterminer si la boîte contient ou non des solutions.

Ces différentes réponses sont dues à l'arithmétique des intervalles [Moore, 1966]. Afin de savoir si une contrainte est satisfaite, chaque variable est remplacée par son domaine puis la contrainte est évaluée avec les règles de calcul de l'arithmétique des intervalles.

Soient $I_1, I_2 \in \mathbb{I}$ deux intervalles tels que $I_1 = [a_1, b_1]$ et $I_2 = [a_2, b_2]$, on a entre autres les formules suivantes :

$$I_1 + I_2 = \left[\underline{a_1 + a_2},\ \overline{b_1 + b_2} \right]$$

$$I_1 - I_2 = \left[\underline{a_1 - b_2},\ \overline{b_1 - a_2} \right]$$

$$I_1 \times I_2 = \left[\begin{array}{l} \min\left(\underline{a_1 \times a_2}, \underline{a_1 \times b_2}, \underline{b_1 \times a_2}, \underline{b_1 \times b_2}\right) \\ \quad \max\left(\overline{a_1 \times a_2}, \overline{a_1 \times b_2}, \overline{b_1 \times a_2}, \overline{b_1 \times b_2}\right) \end{array} \right]$$

$$I_1 / I_2 = I_1 \times [1/a_2, 1/b_2] \text{ si } 0 \notin I_2$$

$$I_1^2 = \begin{cases} \left[\min\left(\underline{a_1^2}, \underline{b_1^2}\right), \max\left(\overline{a_1^2}, \overline{b_1^2}\right)\right] & \text{si } 0 \notin I_1 \\ \left[0, \max\left(\overline{a_1^2}, \overline{b_1^2}\right)\right] & \text{sinon} \end{cases}$$

Exemple 2.2.3 – Soient v_1 et v_2 deux variables continues de domaines $D_1 = [0, 2]$ et $D_2 = [0, 2]$. Soient les trois contraintes suivantes :

$C_1 : v_1 + v_2 \leq 6$
$C_2 : v_1 - v_2 \geq 4$
$C_3 : v_1 - v_2 = 0$

La première contrainte C_1 répond *vrai*, en effet, en remplaçant v_1 et v_2 par leur domaine on obtient $[0, 2] + [0, 2] = [0, 4]$ et donc $[0, 4] \leq 6$ ce qui est vrai étant donné que tout point

de l'intervalle $[0, 4]$ est inférieur ou égal à 6. La deuxième contrainte C_2 répond *faux*, en effet $[0, 2] - [0, 2] = [-2, 2]$ et la condition $[-2, 2] \geq 4$ est toujours fausse. La dernière répond quant à elle *peut-être* : on a $[-2, 2] \leq 0$, la seule chose que l'on peut déduire est qu'il existe peut-être des valeurs de v_1 et v_2 pour lesquelles la contrainte est satisfaite.

2.2.1.3 Solutions, approximations

Étant donné un problème, on cherche à le résoudre, c'est-à-dire à trouver les solutions de ce problème. Dans le cas des variables discrètes, une solution est une instanciation de toutes les variables, ce qui correspond à avoir une seule valeur dans chacun des domaines. Dans le cas des variables continues, on considérera comme solution une boîte contenant uniquement des points solutions ou étant assez petite. Le fait qu'une solution soit contenue dans les boîtes solutions de petite taille n'est pas certain. Le fait qu'elle soit petite réduit les chances qu'elle ne contienne pas de solution mais ne les supprime pas.

Résoudre un problème en programmation par contraintes revient à calculer l'ensemble des solutions ou une approximation de cet ensemble. Dans le cas des variables discrètes, énumérer l'ensemble des solutions est possible mais peut être coûteux. Selon les applications, quand un problème a un très grand nombre de solutions, on ne voudra pas forcément toutes les énumérer. Par exemple, considérons $x \in \mathbb{Z}$ une variable entière, les contraintes $x \geq 0, x \leq 999$ ont un très grand nombre de solutions (1000), et on peut ne souhaiter qu'une ou qu'un sous-ensemble de solutions. Pour les variables continues, l'ensemble de solutions peut être infini et l'énumération n'est alors pas possible. Il est, par exemple, impossible d'énumérer les réels compris entre 0 et 1. De plus, pour un ensemble de solutions fini, il est peu probable que les solutions réelles soient représentables en machine. Dans ce cas, une approximation de l'ensemble des solutions est acceptée.

Définition 2.2.5 (Approximation)**.** Une *approximation complète* (resp. *approximation correcte*), de l'ensemble de solutions est une union de séquences de domaines $D_1 \dots D_n$ telle que $D_i \subseteq \hat{D}_i$ et $S \subseteq \bigcup(D_1 \times \cdots \times D_n)$ (resp. $\bigcup(D_1 \times \cdots \times D_n) \subseteq S$).

La correction garantit que les éléments sont bien des solutions, alors que la complétude garantit qu'aucune solution n'est perdue. En pratique, un solveur de contraintes sur les domaines finis doit être complet et correct, ce qui revient à calculer les solutions. Pour les domaines continus, le fait que les réels ne soient pas représentables en machine complique les choses, et on retirera soit la correction (la plupart du temps) soit la complétude. Dans le premier cas, la résolution retourne des boîtes pouvant contenir des points qui ne sont pas des solutions [Benhamou *et al.*, 1999], on parle alors d'approximation extérieure ou *sur-approximation*. Cette approximation est la plus courante. Pour la plupart des problèmes on cherche à connaître toutes les solutions, et plus important encore on veut être sur de n'en perdre aucune.

Dans le second, toutes les boîtes retournées contiennent uniquement des points solutions [Collavizza *et al.*, 1999, Christie *et al.*, 2006], dans ce cas, on parle d'approximation intérieure ou *sous-approximation*. Ce type d'approximation apparaît pour les problèmes où l'on veut être sûr que tous les points soient bien des solutions, par exemple lors du contrôle d'un bras de robot en chirurgie [Abdallah *et al.*, 1996], on veut s'assurer que le bras du robot reste dans la zone d'opération. Un autre exemple est celui du contrôle d'une caméra [Benhamou *et al.*, 2004]. Dans ce problème on cherche à déterminer les mouvements qui doivent être effectués par une caméra

(déplacements, zoom, *etc.*) afin de réaliser une animation à partir d'une scène et d'un type de plan spécifié par l'utilisateur.

Remarque 2.2.1 – Les notions de correction et complétude sont différentes en interprétation abstraite et en programmation par contraintes. Afin d'éviter toute ambiguïté, on utilisera le terme sur-approximation pour une approximation PPC-complète IA-correcte et sous-approximation pour une approximation PPC-correcte IA-complète.

Afin de résoudre un problème, c'est-à-dire calculer l'ensemble des solutions (ou une approximation dans le cas des réels) à partir du CSP, les méthodes de résolutions PPC-complètes alternent deux phases : la propagation et l'exploration.

2.2.2 Propagation

Dans un premier temps les algorithmes de propagation essaient de réduire les domaines en se basant sur les contraintes. Les valeurs des domaines ne pouvant faire partie d'une solution sont supprimées des domaines. Ces valeurs sont dites inconsistantes.

2.2.2.1 Consistance d'une seule contrainte

Étant donné une contrainte C et des domaines, la consistance supprime des domaines les valeurs inconsistantes pour la contrainte C. Différentes variantes de consistance ont été proposées telles que la consistance d'arc généralisée (*generalized arc consistency*, aussi connue sous le nom d'*hyper-arc consistency* ou *domains consistency*) [Mackworth, 1977b], la consistance de chemin (*path-consistency*) [Montanari, 1974], la k-consistance (k-*consistency*) [Freuder, 1978, Freuder, 1982] ou encore la consistance de borne (*bound consistency*, aussi appelée *interval consistency*) [Dechter, 2003, van Hentenryck *et al.*, 1995, Apt, 2003, Schulte et Stuckey, 2005]. Elles diffèrent suivant le type des domaines et la "force" de celles-ci, qui s'évalue en fonction du nombre de valeurs inconsistantes supprimées des domaines. Plus une consistance est forte, plus elle est coûteuse en termes de complexité en temps et complexité mémoire. Nous reviendrons sur ce point par la suite. Ces consistances reposent sur la notion de support.

Définition 2.2.6 (Support)**.** Soient $v_1 \dots v_n$ des variables de domaines discrets finis $D_1 \dots D_n, D_i \subseteq \hat{D}_i$, et C une contrainte. La valeur $x_i \in D_i$ a un *support* si et seulement si $\forall j \in [\![1, n]\!], j \neq i, \exists x_j \in D_j$ tel que $C(x_1, \dots, x_n)$ soit vrai.

Les consistances les plus courantes sont les suivantes.

Définition 2.2.7 (Consistance d'arc généralisée)**.** Soient $v_1 \dots v_n$ des variables de domaines discrets finis $D_1 \dots D_n, D_i \subseteq \hat{D}_i$, et C une contrainte. Les domaines sont dits *arc-consistants généralisés* (GAC) pour C si et seulement si $\forall i \in [\![1, n]\!], \forall v \in D_i$, v a un support.

La consistance d'arc généralisée conserve uniquement, pour chaque variable, les valeurs pour lesquelles il existe une solution pour la contrainte C, c'est-a-dire, les valeurs ayant un support.

Remarque 2.2.2 – Cette consistance est aussi connue sous le nom de

Exemple 2.2.4 – Considérons deux variables (v_1, v_2) de domaines discrets $D_1 = D_2 = \{-1, 0, 1, 2, 3, 4\}$ et la contrainte $v_1 = 2v_2 + 2$. Les domaines arc-consistants pour ce CSP sont $D_1 = \{0, 2, 4\}$ et $D_2 = \{-1, 0, 1\}$.

Les valeurs -1 et 1 ont été supprimées du domaine de v_1 car il n'existe pas de valeurs pour v_2 telles que la contrainte soit satisfaite. De même, si $v_2 \geq 2$ on en déduit que $v_1 \geq 6$, or la valeur maximale pour v_1 est 4. Les valeurs supérieures ou égales à 2 peuvent donc être supprimée du domaine de v_2.

La consistance d'arc généralisée pour une contrainte binaire C a une complexité en temps dans le pire des cas de $O(d^2)$ [Mohr et Henderson, 1986], où d est la taille maximale des domaines. Notons que cette complexité ne prend pas en compte la complexité de la contrainte. Pour chaque variable toutes les valeurs doivent être testées afin de s'assurer qu'elles aient bien un support. Pour les contraintes de plus de deux variables, vérifier que des domaines soient arc-consistants est NP-complet, et cela même quand les contraintes sont linéaires [Choi *et al.*, 2006]. Il existe, cependant, des algorithmes spécifiques pour certains types de contraintes. Par exemple, il existe un algorithme dédié pour différentes variantes de la contrainte globale alldifferent [Sellmann, 2002, Hoeve, 2004, Gent *et al.*, 2008].

D'autre consistances pour les variables discrètes ont été définies, telles que la consistance de borne basée sur la représentation des variables discrètes à l'aide d'intervalles entiers. Elle est définie comme suit.

Définition 2.2.8 (Consistance de borne)**.** Soient $v_1 \dots v_n$ des variables de domaines discrets finis $D_1 \dots D_n$, $D_i \subseteq \hat{D}_i$, et C une contrainte. Les domaines sont dits *borne-consistants* (BC) pour C si et seulement si $\forall i \in [\![1, n]\!]$, D_i est un intervalle entier, $D_i = [\![a_i, b_i]\!]$, et a_i et b_i ont un support.

Cette consistance est moins forte que la consistance d'arc généralisée, en effet elle vérifie uniquement si les bornes des domaines possèdent un support.

Remarque 2.2.3 – Il existe différentes définitions de la consistance de borne [Choi *et al.*, 2006] (bound(D)-consistency, bound(Z)-consistency, bound(R)-consistency ou encore range consistency).

Exemple 2.2.5 – Considérons deux variables (v_1, v_2) de domaine discret $D_1 = D_2 = [\![-1, 4]\!]$ et la contrainte $v_1 = 2v_2 + 2$. Les domaines borne-consistants pour ce CSP sont $D_1 = [\![0, 4]\!]$ et $D_2 = [\![-1, 1]\!]$.

Les domaines borne-consistants et arc-consistants pour D_2 pour la variable v_2 sont égaux (-1, 0, et 1) bien que la représentation soit différente (intervalle entier ou ensemble d'entiers). Au contraire, le domaine borne-consistant $D_1 = [\![0, 4]\!]$ pour la variable v_1 a une représentation plus compacte que le domaine arc-consistant $D_1 = \{0, 2, 4\}$ mais contient plus de valeurs possibles pour v_1.

L'intuition nous fait penser que la complexité de la consistance de borne pour une contrainte binaire est dans le pire des cas en $O(d^2)$ avec d la taille maximale des domaines. En effet, dans le pire des cas la consistance de borne laisse une seule valeur dans chacun des domaines, et donc vérifie si il existe un support pour chacune des valeurs des domaines. Pour toute contrainte, la complexité de la consistance de borne est dans le pire des cas $O(n^2)$ où n est le nombre de variables [Zhang et Yap, 2000]. Tout comme la consistance d'arc généralisée, vérifier si des domaines sont borne-consistants est un problème NP-complet dans le cas général [Schulte et Stuckey, 2005, Choi *et al.*, 2006]. Notons qu'il existe tout de même différents travaux proposant des algorithmes efficaces pour le calcul de la consistance de borne pour certaines contraintes, telles que la contrainte globale alldifferent [Puget, 1998, Mehlhorn et Thiel, 2000, López-Ortiz *et al.*, 2003] et

la contrainte globale gcc [Katriel et Thiel, 2003, Quimper *et al.*, 2003] ou encore les contraintes sur des ensembles [van Hentenryck *et al.*, 2008].

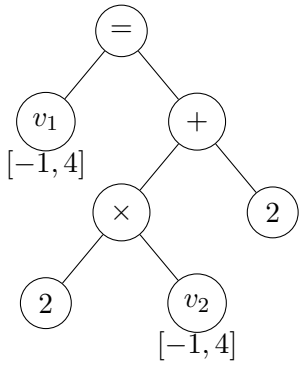
Il existe aussi différentes consistances pour les domaines de variables continues comme la consistance d'enveloppe (ou Hull-consistance, *Hull-consistency*), la consistance de boîte (*box-consistency*) ou encore une extension de la consistance d'arc généralisée pour les contraintes continues [Benhamou *et al.*, 1994]. Nous ne présentons ici que la consistance d'enveloppe telle que définie dans [Benhamou et Older, 1997].

Définition 2.2.9 (Hull-Consistance)**.** Soient $v_1 \dots v_n$ des variables de domaines continus représentés par les intervalles $D_1 \dots D_n \in \mathbb{I}, D_i \subseteq \hat{D}_i$, et C une contrainte. Les domaines $D_1 \dots D_n$ sont dits *Hull-consistant* (HC) pour C si et seulement si $D_1 \times \cdots \times D_n$ est la plus petite boîte à bornes flottantes contenant les solutions de C dans $D_1 \times \cdots \times D_n$.

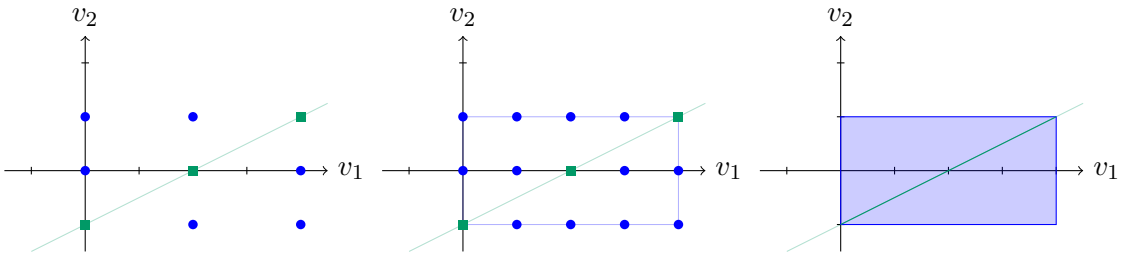
Exemple 2.2.6 – Considérons deux variables (v_1, v_2) de domaine continu $D_1 = D_2 = [-1, 4]$ et la contrainte $v_1 = 2v_2 + 2$. Les domaines hull-consistants pour ce CSP sont $D_1 = [0, 4]$ et $D_2 = [-1, 1]$.

Il existe différents algorithmes permettant de calculer pour une contrainte la consistance d'enveloppe. En 1999, Benhamou *et al.* dans [Benhamou *et al.*, 1999] proposent l'algorithme HC4-Revise se basant sur la représentation des contraintes sous forme d'arbre afin de calculer la consistance d'enveloppe en un temps linéaire $O(e)$ avec e le nombre d'opérateurs unaires ou binaires de la contrainte. Récemment, un nouvel algorithme Mohc-Revise [Araya *et al.*, 2010] étudie la monotonie de la contrainte afin de calculer la consistance d'enveloppe.

Exemple 2.2.7 – Reprenons le CSP de l'exemple 2.2.6. La contrainte $v_1 = 2v_2 + 2$ est représentée à l'aide de l'arbre de syntaxe suivant :

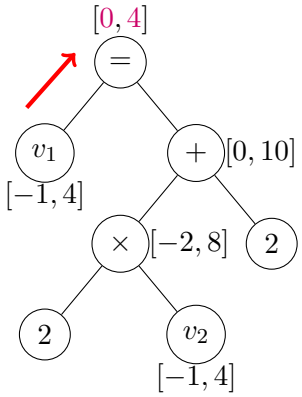


Les nœuds correspondant à des variables sont étiquetés par le domaine de la variable correspondante. L'algorithme HC4-Revise parcourt une première fois l'arbre des feuilles vers la racine, en utilisant l'arithmétique des intervalles [Moore, 1966] afin d'étiqueter chacun des nœuds par l'intervalle des valeurs possibles. Par exemple, le nœud $(\times)$ est étiqueté $2 \times [-1, 4] = [-2, 8]$. L'étiquette de la racine est calculée à partir des nœuds fils mais aussi en fonction du type de la contrainte. Dans cet exemple, la contrainte étant une égalité, l'étiquette de la racine est égale à l'intersection des intervalles de ses nœuds fils.

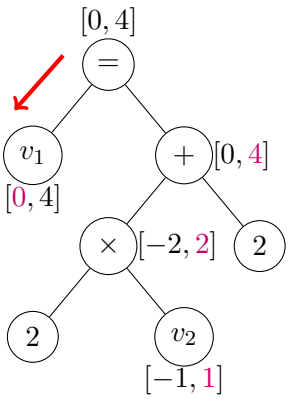


(a) GAC (b) BC (c) HC

Figure 2.5 – Différentes consistances pour la contrainte $v_1 = 2(v_2 + 1)$.



Puis un deuxième parcours de la racine vers les feuilles est effectué. Il permet de communiquer aux nœuds de l'arbre l'information obtenue à la racine, et ainsi modifier les valeurs possibles à chaque nœud. Le nœud $(\times)$ est désormais étiqueté $[0, 4] - 2 = [-2, 2]$.



L'application de cet algorithme nous permet de retrouver le même résultat que dans l'exemple 2.2.6 ($D_1 = [0, 4], D_2 = [-1, 1]$). Notons cependant que ce n'est pas toujours le cas.

La figure 2.5 illustre les consistances obtenues dans les exemples 2.2.4, 2.2.5 et 2.2.6. Dans les figures 2.5(a) et 2.5(b) les points sont ceux obtenus par le produit cartésien des domaines consistants. Les points solutions sont représentés par des carrés verts, et sont au nombre de 3. En

effectuant le produit cartésien des domaines arc-consistants (figure 2.5(a)), on obtient 9 points, et pour les domaines borne-consistants (figure 2.5(b)), on obtient 15 points. On peut en déduire que, plus la consistance choisie est forte et plus le ratio du nombre de points solutions sur le nombre total de points est élevé. La figure 2.5(c) montre quant à elle la boîte hull-consistante, les solutions sont là aussi en vert et correspondent à la diagonale dans la boîte. On peut voir que dans cette boîte, il y a plus de points non-solutions que de points solutions.

Pour une contrainte C donnée, un algorithme qui prend en entrée les domaines et retire les valeurs inconsistantes est appelé un *propagateur*, noté ρ_C. L'algorithme HC4-Revise est un propagateur pour la consistance d'enveloppe. Les propagateurs retournent toujours une sur-approximation de S. Il arrive qu'ils calculent exactement les domaines consistants, mais comme cela peut être très coûteux ou impossible, en général, ils peuvent retourner des sur-approximations des domaines consistants.

Pour plusieurs contraintes, les propagateurs de chaque contrainte sont appliqués itérativement jusqu'à ce qu'un point fixe soit atteint. On parle alors de boucle de propagation.

2.2.2.2 Boucle de propagation

Les représentations des domaines, présentées précédemment (définitions 2.2.2, 2.2.3, 2.2.4), formant des treillis complets pour l'inclusion, il est suffisant de calculer la consistance pour chacune des contraintes jusqu'à ce qu'un point fixe soit atteint. Il a été montré par Apt en 1999 dans [Apt, 1999] ou par Benhamou en 1996 dans [Benhamou, 1996] que l'ordre dans lequel les consistances sont appliquées n'importe pas. En effet, les treillis étant complets, tout sous-ensemble a un unique plus petit élément : le point fixe consistant.

Cependant l'ordre dans lequel les consistances, et donc les propagateurs, sont appliquées influe sur la vitesse de convergence, c'est-à-dire le nombre d'itérations nécessaire afin d'atteindre le point fixe. Différentes stratégies de propagation sont possibles. La stratégie naïve est d'exécuter à chaque itération les propagateurs de toutes les contraintes. Une autre est d'exécuter les propagateurs des contraintes contenant au moins une variable dont le domaine a été modifié à l'itération précédente. Cette stratégie est celle généralement utilisée. Trouver un algorithme efficace calculant les domaines consistants pour une conjonction de contraintes dans le cas général est un vrai challenge scientifique.

Par la suite, le terme "complexité" est utilisé pour désigner la complexité dans le pire des cas.

Depuis les années 1970, de nombreux algorithmes ont été développés pour la consistance d'arc généralisée pour les contraintes binaires (AC) ou pour toutes les contraintes (GAC), chacun améliorant la complexité des algorithmes précédents. Le premier algorithme AC1 décrit par Mackworth en 1977 [Mackworth, 1977a] permet de calculer les domaines arc-consistants pour des contraintes binaires. Il vérifie pour chaque contrainte (p), pour chacune des variables (n), et pour chaque valeur possible (au plus d) pour les variables si elle possède un support (d^2). Cet algorithme a une complexité en temps de $O(d^3np)$ où d est la taille maximale des domaines, n est le nombre de variables et p le nombre de contraintes. Cet algorithme est très simple et une version améliorée, appelée AC3 est donnée dans le même article [Mackworth, 1977a]. AC3 utilise une liste afin de ne propager que les contraintes dont au moins une variable a été modifiée. AC3 a une complexité en temps de $O(pd^3)$ et une complexité en mémoire de $O(p)$. La même année une version étendue de AC3 n'étant plus restreinte au contraintes binaires est présentée, GAC3 [Mackworth, 1977b]. Cet algorithme a une complexité en temps de $O(pk^3d^{k+1})$ et une complexité en mémoire de $O(pk)$,

avec k l'arité maximale des contraintes, c'est-à-dire le nombres maximal de variables pouvant être contenues dans les contraintes.

Les algorithmes AC4, proposé par Mohr et Henderson en 1986 [Mohr et Henderson, 1986], et GAC4, proposé par Mohr et Masini en 1988 [Mohr et Masini, 1988], ont une complexité en temps dans le pire des cas optimale [Bessière, 2006]. AC4 a une complexité en temps et en mémoire de $O(pd^2)$ et GAC4 a une complexité en temps de $O(pkd^k)$.

De très nombreux algorithmes ont depuis été conçus : AC5 [van Hentenryck *et al.*, 1992], AC6 [Bessière, 1994], AC7 [Bessière *et al.*, 1999], AC2001 [Bessière et Régin, 2001], AC3.2 [Lecoutre *et al.*, 2003] pour n'en nommer que quelques uns, rivalisant d'ingéniosité afin de réduire en pratique la complexité en mémoire et surtout le temps de calcul. Une comparaison et la complexité de certains de ces algorithmes ainsi que celles données précédemment peuvent être trouvées dans [Mackworth et Freuder, 1982] et [Bessière, 2006].

Alors qu'il y a eu de nombreux articles et algorithmes sur le calcul efficace de la consistance d'arc généralisée, la consistance de borne a fait l'objet de moins de travaux. De plus, il existe différentes définitions de la consistance de borne [Choi *et al.*, 2006] (bound(D)-consistency, bound(R)-consistency ou encore range consistency) et celle considérée ici a changé plusieurs fois de noms au cours des années.

Dans le cas des variables continues, l'algorithme HC3 proposé en 1995 [Benhamou, 1995] permet de calculer une approximation de la consistance d'enveloppe pour une conjonction de contraintes. Puis en 1999 [Benhamou *et al.*, 1999], l'algorithme HC4 est proposé. La boucle de propagation appelle l'algorithme HC4-Revise, vu précédemment, pour chaque contrainte contenant au moins une des variables dont le domaine a été modifié. Plus récemment, un algorithme Mohc étudiant la monotonie des contraintes à propager a été développé [Araya *et al.*, 2010, Araya *et al.*, 2012].

2.2.3 Exploration

Généralement, la propagation ne suffit pas pour calculer les solutions. Durant la deuxième étape du processus de résolution, des hypothèses sur les valeurs des variables sont faites. Dans le cas des variables entières, des valeurs sont itérativement données aux variables jusqu'à obtenir un succès (une solution est trouvée) ou un échec (une contrainte fausse, un domaine vide). Pour les variables réelles, l'idée est la même à ceci près que les domaines sont coupés en deux sous-domaines plus petits. Ce mécanisme d'exploration introduit la notion de *point de choix* dans le processus de résolution.

2.2.4 Schéma de résolution

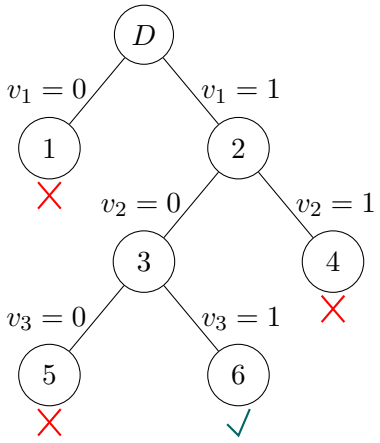
Le processus de résolution alterne les phases de réduction et de choix. En tout point de la résolution, la propagation est d'abord effectuée afin de supprimer le plus tôt des domaines les valeurs inconsistantes et ainsi éviter les calculs inutiles. Ainsi, la consistance est maintenue tout au long de la résolution. Trois cas sont alors possibles :

1. une solution est trouvée, elle est alors stockée ;
2. aucune solution n'est possible, c'est-à-dire qu'au moins un des domaines est vide ou qu'une contrainte est fausse ;
3. ou aucun des deux cas précédents.

Dans les deux premiers cas, le processus retourne à un point de choix antérieur afin d'effectuer une nouvelle hypothèse. Plusieurs cas sont alors possibles, retourner au dernier point de choix, appelé *backtracking*, ou dans le cas d'un échec, retourner au point de choix responsable de l'échec, dans ce cas on parle de *backjumping* [Dechter, 1990]. Dans tous les cas, si l'espace de recherche n'est pas encore totalement exploré, un choix est effectué, une variable et une valeur sont choisies dans le cas discret ou un domaine à couper dans le cas continu. Ce choix est pris en compte en instanciant la variable ou coupant le domaine choisi, puis on revient à la première étape : la propagation est effectuée en ce nouveau point de la résolution. Ainsi, la consistance est maintenue tout au long de la résolution.

Ce mécanisme de résolution correspond à un arbre de recherche dans lequel les nœuds correspondent au point de choix, et les arcs correspondent à une affectation d'une variable ou à une coupe d'un domaine.

Exemple 2.2.8 – Considérons un CSP sur des variables booléennes v_1, v_2, v_3 de domaines $D_1 = D_2 = D_3 = \{0, 1\}$ et avec la contrainte : v_1 et non v_2 et v_3. On a l'arbre de recherche suivant :



où $D = D_1 \times D_2 \times D_3$. En partant de l'espace de recherche initial, un premier choix est fait : $v_1 = 0$. Après cette affectation (nœud 1) la contrainte est fausse, l'exploration est arrêtée dans cette branche et retourne à la racine où un autre choix est fait : $v_1 = 1$ (nœud 2). L'exploration continue jusqu'à ce que tous les nœuds soient visités.

Dans cet exemple, les croix rouges correspondent à des échecs et la marque verte correspond à une solution.

Le schéma général d'un solveur discret utilisant la consistance d'arc généralisée est donné en exemple (algorithme 2.1). De même, un solveur continu est donné algorithme 2.2. Il utilise la consistance d'enveloppe. Dans ces exemples, la résolution s'arrête quand :

- toutes les solutions sont trouvées, c'est-à-dire :
  - toutes les solutions ont été énumérées dans le cas discret ;
  - les boîtes calculées contiennent uniquement des solutions ou sont plus petite qu'une certaine précision, dans le cas continu.
- il a été prouvé qu'il n'existe aucune solution, en d'autres termes, l'espace de recherche a été entièrement exploré sans succès.

int $j \leftarrow 0$ */* j indique la profondeur de l'arbre de recherche */*
int op[] */* à une profondeur j, stocke l'indice de la variable sur laquelle on fait des hypothèses, initialisé uniformément à 0 */*
int width[]*/* à une profondeur j, stocke la largeur de l'arbre déjà explorée, initialisé uniformément à 0 */*
produit cartésien entier $e \in \mathbb{S}$
liste de produits cartésiens entiers sols */* stocke les solutions */*
$e \leftarrow D$ */* initialisation avec les domaines initiaux */*
répéter
  $e \leftarrow$ consistance d'arc généralisée(e)
  width[j]+ +
  si e est une solution **alors**
    / succès */*
    sols $\leftarrow$ sols $\cup\, e$
  fin si
  si $e = \emptyset$ ou e est une solution **alors**
    / retour au dernier point backtrackable */*
    tant que $j \geq 0$ et width[j] $\geq |D_{op[j]}|$ **faire**
      width[j] $\leftarrow 0$
      j– –
    fin tant que
  sinon
    / nouvelles hypothèses */*
    choisir une variable v_i à instancier
    op[j] $\leftarrow i$
    j+ +
  fin si
  si $j \geq 0$ **alors**
    affecter $v_{op[j-1]}$ à la (width[j] + 1)-eme valeur possible */* point backtrackable */*
  fin si
jusqu'à $j < 0$

Algorithme 2.1: Un solveur discret, utilisant les produits cartésiens entiers pour représenter les domaines.

Notons qu'à la fois dans le cas discret et le cas continu, le processus de résolution peut être modifié de façon à s'arrêter dés que la première solution est trouvée.

Dans ces deux méthodes de résolution, les critères de choix de la variable à instancier ou du domaine à couper ne sont pas donnés explicitement. Ceci vient du fait qu'il n'existe pas une unique façon de choisir le domaine à couper ou la variable à instancier et que cela dépend souvent du problème à résoudre. La prochaine section décrit certaines des stratégies de choix ou stratégies d'exploration existantes.

```
liste de boîtes sols ← ∅                                   /* stocke les solutions */
queue de boîtes toExplore ← ∅                              /* stocke les boîtes à explorer */
boîte b ← D                                                /* initialisation avec les domaines initiaux */
push b dans toExplore
tant que toExplore ≠ ∅ faire
  b ← pop(toExplore)
  b ← Hull-consistence(b)
  si b ≠ ∅ alors
    si b ne contient que des solutions ou b est assez petit alors
      /* succès */
      sols ← sols ∪ b
    sinon
      /* nouvelles hypothèses */
      coupe b en b1 et b2 en coupant la suivant l'une de ses dimensions
      push b1 et b2 dans toExplore
    fin si
  fin si
fin tant que
```

Algorithme 2.2: Un solveur classique en continu.

2.2.5 Stratégies d'exploration

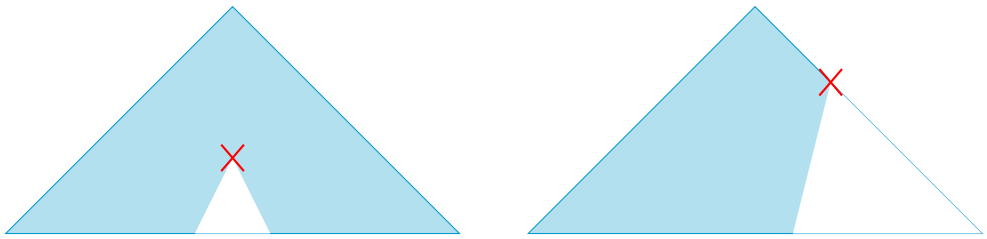
De nombreuses stratégies de choix ont été développées afin de déterminer l'ordre selon lequel les variables doivent être instanciées ou les domaines doivent être coupés. Selon les choix effectués, les performances peuvent être très différentes. Dans le cas discret, la plus connue est peut-être celle présentée en 1979 par Haralick et Elliott [Haralick et Elliott, 1979], *first-fail*, choisissant d'instancier la variable ayant le plus petit domaine. L'idée est d'échouer le plus tôt possible en instanciant en premier la variable pour laquelle très peu de valeurs sont possibles. L'intuition est la suivante : plus tôt on échoue et plus gros est le sous-arbre coupé dans l'arbre de recherche.

"To succeed, try first where you are most likely to fail"
– Robert M. Haralick et Gordon L. Elliott [Haralick et Elliott, 1979].

La figure 2.6 montre que plus tôt l'échec apparaît dans l'arbre de recherche et plus gros est le sous-arbre coupé dans l'arbre de recherche. Dans la figure 2.6(a) l'échec arrive tard et seulement un petit sous-arbre dans l'arbre de recherche est coupé. En revanche dans la figure 2.6(b) l'échec arrive plus tôt et un plus gros sous-arbre dans l'arbre de recherche est coupé.

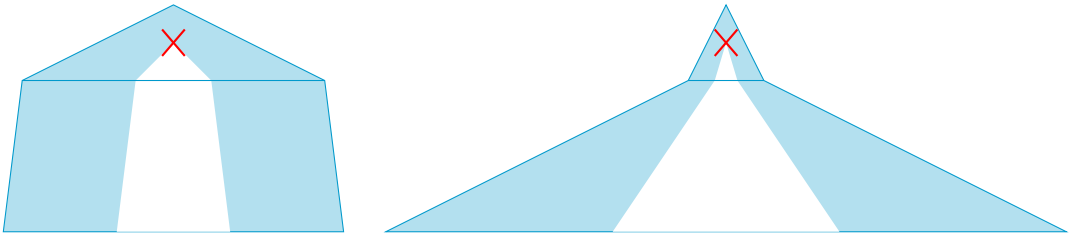
La figure 2.7, compare l'arbre de recherche obtenu avec la stratégie *first-fail* (figure 2.7(b)) à celui obtenu avec la stratégie instanciant les variables de grands domaines en premier. Notons que ces deux arbres ont la même surface. En cas d'échec, le sous-arbre coupé dans l'arbre de recherche est plus grand quand on utilise la stratégie *first-fail*.

Suivant cette idée, l'heuristique proposée dans [Brélaz, 1979] choisit la variable ayant le plus petit domaine (dom) et apparaissant dans le plus grand nombre de contraintes deg. En d'autres termes, la variable choisie est celle maximisant $\mathrm{dom} + \mathrm{deg}$. Puis en 1996 une autre stratégie est présentée [Bessière et Régin, 1996] choisissant la variable maximisant le ratio $\mathrm{dom}/\mathrm{deg}$. En 2004 [Boussemart *et al.*, 2004], une heuristique qui choisit la variable maximisant le ration $\mathrm{dom}/w\mathrm{deg}$



(a) (b)

Figure 2.6 – Comparaison du sous-arbre de l'arbre de recherche coupé, en fonction du moment où apparaît l'échec.



(a) Plus grands domaines d'abord (b) Plus petits domaines d'abord

Figure 2.7 – Comparaison de la stratégie qui consisterait à instancier les variables de grands domaines en premier (2.7(a)) à la stratégie *first-fail* (2.7(b)).

avec w un poids associé à chaque contrainte, est introduite. Les poids sont initialisés uniformément à 1 au début de la résolution, et à chaque fois qu'une contrainte est responsable d'un échec, son poids est augmenté. L'intuition est la suivante : en instanciant en priorité les variables faisant partie d'une contrainte difficile à satisfaire, plus tôt cette contrainte provoquera un échec et donc plus gros sera le sous-arbre coupé dans l'arbre de recherche. D'autres heuristiques ainsi qu'une comparaison de celles-ci sont données dans [Geelen, 1992, Gent *et al.*, 1996, van Beek, 2006].

Une fois la variable à instancier choisie, il ne reste qu'à choisir par quelle valeur l'instancier. Là aussi, de nombreuses stratégies ont été développées, choisissant la valeur maximisant le nombre de solutions possibles [Dechter et Pearl, 1987, Kask *et al.*, 2004], le produit de la taille des domaines (*promise*) [Ginsberg *et al.*, 1990], la somme de la taille des domaines (*min-conflicts*) [Frost et Dechter, 1995].

Dans le cas de variables continues, un domaine est généralement coupé en deux. Plusieurs heuristiques de choix du domaine à couper sont possibles, *largest-first* [Ratzet, 1994] choisissant le plus grand domaine, ainsi la taille des boîtes est réduite plus rapidement ; *round-robin* coupant les domaines les uns après les autres, permettant de s'assurer que tous les domaines sont coupés ; ou encore *Max-smear* [Hansen, 1992, Kearfott, 1996] choisissant de couper le domaine la fonction *smear* de la matrice jacobienne des contraintes (contenant les dérivées partielles de chacune des contraintes). Cette stratégie, revient à couper le domaine de la variable ayant la plus grande pente dans les contraintes.

2.2.6 Comparaison discret/continu

Il est important de remarquer que le schéma de résolution est significativement différent en fonction du type des variables (entier ou réel). En pratique, les solveurs sont dédiés à un certain type de variables. Il existe de nombreux solveurs discrets. Parmi les plus connus, on trouve Choco [Team, 2010], GeCode [Schulte, 2002], GnuProlog avec contraintes [Diaz *et al.*, 2012], Jacop [Wolinski *et al.*, 2004], Comet [van Hentenryck et Michel, 2005], Eclipse [Apt et Wallace, 2007] et Minion [Gent *et al.*, 2006]. Il existe bien moins de solveurs continus en programmation par contraintes. Les plus connus sont Declic [Benhamou *et al.*, 1997], Numerica [van Hentenryck *et al.*, 1997], RealPaver [Granvilliers et Benhamou, 2006] et Ibex [Chabert et Jaulin, 2009]. Prolog IV [Colmerauer, 1994] intègre à la fois un solveur discret et un solveur continu.

Des astuces d'implémentation sont nécessaires afin de résoudre des problèmes contenant à la fois des variables entières et des variables réelles. Par la suite, on emploiera les termes de "problème mixte" ou "problème continu-discret" pour désigner ce type de problèmes. Si le solveur choisi est discret, les variables réelles sont discrétisées, c'est-à-dire que les valeurs possibles pour les variables réelles sont énumérées pour un pas donné.

Exemple 2.2.9 – Soit x une variable réelle dont la valeur est comprise entre 0 et 0.5, et un pas de 0.1. Une fois discrétisée, son domaine est $\{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$.

Cette méthode offre la possibilité de traiter les problèmes mixtes mais dépend fortement du pas choisi. En effet, si le pas choisi est grand, des solutions pourraient être perdues mais le domaine des variables n'étant pas trop grand est plus facile à stocker. Au contraire si le pas est petit, les chance de perdre des solutions est moindre mais la taille des domaines obtenus est très grande et cela explose la combinatoire du problème. Cette méthode est utilisée entre autres dans le solveur discret Choco [Team, 2010].

Si le solveur choisi est continu, des contraintes d'intégrité peuvent être ajoutées au problème. Ces nouvelles contraintes spécifient quelles variables sont entières et permettent ainsi un raffinement des bornes des domaines de ces variables. Lors de la propagation de ces contraintes, les bornes des domaines sont arrondies à l'entier le plus proche dans la direction adéquate. Cette méthode impose la consistance de borne pour les variables discrètes et ne permet pas d'utiliser la consistance d'arc généralisée. Elle a été développée par le solveur continu RealPaver [Granvilliers et Benhamou, 2006].

Une autre alternative est d'ajouter des contraintes globales discrètes à un solveur continu [Berger et Granvilliers, 2009] ou de créer des contraintes globales mixtes [Chabert *et al.*, 2009] permettant de traiter au sein d'une même contrainte des variables discrètes et continues. Ainsi, chaque variable bénéficie d'une consistance adaptée à son type (discret ou continu). Cependant cette méthode dépend du problème et demande de créer pour chaque problème les contraintes globales nécessaires ainsi qu'une consistance *ad hoc*.

2.2.7 Conclusion

La programmation par contraintes permet de résoudre efficacement des problèmes de satisfaction de contraintes. Bien que les méthodes de résolution ne dépendent pas du problème à résoudre, elles sont fortement dédiées au type des variables (discret ou continu) du problème. Cette spécialisation, bien que restrictive, rend les méthodes de résolutions efficaces. De nombreuses heuristiques

sont développées et utilisées lors de l'exploration afin d'améliorer les résultats obtenus tant au niveau de la qualité des solutions que du temps de résolution. Cependant, l'un des obstacles majeurs au développement de la programmation par contraintes et le manque d'outils et méthodes permettant de résoudre les problèmes mixtes (ayant à la fois des variables discrètes et réelles). Il n'existe pas, par exemple, de représentation permettant de mélanger dans un même CSP les entiers et les réels en conservant leurs types.

2.3 Synthèse

Dans les sections précédentes, nous avons introduit certaines notions d'interprétation abstraite et de programmation par contraintes, deux domaines de recherche qui semblent très éloignés. Pourtant, il existe des similitudes tant au niveau de la théorie sous-jacentes que dans les techniques utilisées. Cette section met en évidence certains de ces liens mais aussi certaines différences notables.

2.3.1 Liens entre l'interprétation abstraite et la programmation par contraintes

Il existe des liens entre la programmation par contraintes et l'interprétation abstraite, par exemple, ces deux domaines reposent sur le même cadre théorique (treillis et point fixe), mais en y regardant de plus près il y a aussi des différences notables. On présente ici certaines des similitudes et différences entre ces deux domaines.

Comme dit précédemment, la programmation par contraintes et l'interprétation abstraite ont le même cadre théorique, et plus important encore, le but principal est le même : calculer une sur-approximation d'un ensemble désiré, exprimé comme les solutions des contraintes en programmation par contraintes et comme les propriétés d'un programme en interprétation abstraite. Dans le cas particulier des CSP discrets, l'approximation doit être égale à l'ensemble désiré. La consistance et la propagation peuvent être vues comme un rétrécissement sur les domaines abstraits, car ils contribuent tous deux à réduire les domaines tout en restant au-dessus du point fixe. En interprétation abstraite, l'opérateur de rétrécissement peut être utilisé soit après un élargissement, ce qui garantit que les domaines abstraits restent plus grand que le point fixe voulu, soit dans le cas particulier des itérations locales. En programmation par contraintes, les domaines sont toujours réduits et la propagation est utilisée pour réduire les domaines le plus tôt possible. Notons que dans les deux cas, il est permis de ne pas atteindre le point fixe. Par exemple, en programmation par contraintes, pour certaines contraintes, telles que la contrainte `nvalue`, le calcul de la consistance est NP-difficile [Bessière *et al.*, 2004] et il est admis de calculer une sur-approximation des domaines consistants même pour des contraintes discrètes.

Remarque 2.3.1 – Concernant le cadre théorique, il faut souligner une autre différence notable entre la programmation par contraintes et l'interprétation abstraite. En interprétation abstraite, les théorèmes de points fixes sont fondamentaux et à la base de l'analyse des boucles, étape clef de l'analyse d'un programme. Nous ne les rappelons pas ici, car ils deviennent inutiles en programmation par contraintes : les treillis utilisés pour les consistances sont toujours finis. Ils sont construits à partir d'ensembles de départ bornés (domaines initiaux) et leurs éléments doivent être représentables en machine (entiers ou flottants), donc en nombre fini.

Les deux domaines reposent sur le calcul de points fixes ou de sur-approximations de points fixes. Cependant, les techniques utilisées pour les calculer sont significativement différentes. Pre-

mièrement, durant la résolution d'un problème donné, pour une succession de points de choix, les approximations en programmation par contraintes sont strictement décroissantes, excepté quand le point fixe est atteint, tandis qu'en interprétation abstraite les approximations peuvent augmenter, lors de l'approximation d'une boucle par exemple. La programmation par contraintes vise la complétude en améliorant les solutions en utilisant le raffinement, alors que l'interprétation abstraite embrasse généralement l'incomplétude.

Par ailleurs, la précision de l'analyseur statique est conditionnée par le choix du domaine abstrait et des opérateurs utilisés. Quand un raffinement est effectué, les domaines deviennent plus précis, ce qui peut être fait soit en relançant l'analyseur à partir des domaines obtenus [Clarke *et al.*, 2003], soit manuellement, ce qui nécessite de changer l'analyseur [Bertrane *et al.*, 2010]. Une autre façon d'améliorer la précision est d'ajouter des itérations locales [Granger, 1992]. En résumé, la précision d'un domaine abstrait n'est pas défini comme telle, elle est induite par les techniques choisies pour l'analyse. En revanche, la programmation par contraintes intègre pour les domaines continus une définition explicite de précision. Le choix du domaine abstrait dans un solveur de programmation par contraintes ne change pas sa précision (qui est fixe), mais son efficacité (la quantité de calcul nécessaire pour atteindre la précision souhaitée).

Une autre différence significative est que toutes les techniques de l'interprétation abstraite sont paramétrées par les domaines abstraits. Au contraire, les solveurs en programmation par contraintes dépendent fortement du type des variables (entières ou réelles) et surtout sont dédiés au domaine. Il n'existe pas de solveurs en programmation par contraintes pour lesquels les domaines seraient paramétrables, en particulier, il n'existe pas de solveurs mixtes. Section 2.2, nous avons vu qu'en programmation par contraintes, il existe seulement trois représentations pour les valeurs des variables en fonctions du type des variables. Au contraire, en interprétation abstraite, un important effort de recherche a été fait sur les domaines abstraits, conduisant à l'élaboration de nombreux domaines cartésiens ou relationnels tels que les polyèdres, octogones, ellipsoïdes pour n'en nommer que quelques-uns. Notons qu'ils peuvent dépendre du type des variables, comme les diagrammes binaires de décisions pour les variables booléennes [Bertrane *et al.*, 2010], mais plus important encore, ils peuvent analyser différents types de variables au sein du même analyseur.

2.3.2 Analyse

La programmation par contraintes permet de modéliser des problèmes très variés sous une même et unique forme, celle des CSP. Elle offre une grande variété de contraintes, notamment, dans le cas des variables discrètes, plus de trois cents contraintes globales qui permettent de décrire des relations complexes entre les variables et viennent avec des algorithmes de propagation efficaces. De manière générale, les algorithmes de propagation des contraintes sont aujourd'hui très efficaces. Nombres de travaux ont porté sur l'amélioration de la complexité et donc de l'efficacité de ces algorithmes. Un autre point fort de la programmation par contraintes est qu'elle offre des méthodes de résolution génériques, qui ne dépendent pas du problème à résoudre, et une même contrainte peut être utilisée dans différents problèmes. Afin d'améliorer les performances de ces méthodes de résolution, de très nombreuses heuristiques sont développées, telles que les heuristiques de choix de variables ou valeur pour l'instanciation, ou les heuristiques de choix de domaine à couper. Cependant, la programmation par contraintes n'offre que très peu de représentations pour les domaines, et tout comme les méthodes de résolution, ceux-ci sont restreint à un seul type de variables (discret ou continu).

L'interprétation abstraite quant à elle propose un très grand nombre de représentations, les domaines abstraits. Ceux-ci peuvent être de différentes formes (boîte, polyèdre, ellipse, ...) et ne sont pas définis pour un type particulier de variables. Ainsi, ils peuvent représenter aussi bien des variables discrètes que continues. En outre, l'utilisation de plusieurs domaines abstraits permet aux analyseurs de traiter efficacement des programmes de très grande taille, c'est-à-dire des programmes contenant un grand nombre de variables et de lignes de code. Un autre avantage des domaines abstraits est qu'ils viennent avec des algorithmes efficaces pour les opérateurs d'élargissement et de rétrécissement permettant de calculer en un petit nombre d'itérations une approximation du point fixe. Cependant cette approximation n'est pas toujours très précise et cela même après l'application de l'opérateur de rétrécissement ou des itérations locales.

En résumé, la programmation par contraintes n'offre que très peu de représentations pour les domaines mais des algorithmes efficaces, tandis qu'il existe, en interprétation abstraite, de très riches représentations pour les domaines et des algorithmes pour calculer les sur-approximations mais pas d'algorithme de résolution à proprement parler (au sens où la précision n'est pas paramétrable).

Il existe déjà des travaux à la frontière entre l'interprétation abstraite et le programmation par contraintes. Par exemple, la programmation par contraintes a été utilisée en vérification de programmes [Collavizza et Rueher, 2007], pour analyser des modèles caractéristiques et automatiquement générer des tests de configurations [Hervieu *et al.*, 2011], pour vérifier la modélisation de problèmes de satisfaction de contraintes [Lazaar *et al.*, 2012], ou encore pour améliorer les résultats d'un analyseur statique [Ponsini *et al.*, 2011].

Une autre approche pour la vérification de programmes est d'utiliser un solveur de satisfiabilité [King, 1969]. Ces dernières années ont vu des améliorations significatives dans les méthodes de satisfiabilité booléennes (SAT) et non-booléennes (SMT) [Kroening et Strichman, 2008], ainsi que dans leurs applications (par exemple le model-checker CBMC [checker CBMC,]). Par ailleurs, D'Silva et al. [D'Silva *et al.*, 2012] ont récemment proposé d'exprimer les algorithmes SAT dans le cadre de l'interprétation abstraite (à l'aide des points fixes et des abstractions), un moyen prometteur pour la pollinisation croisée entre ces domaines.

Les travaux présentés dans cette thèse sont similaires, sauf que nous nous situons à l'intersection entre l'interprétation abstraite et la programmation par contraintes. Il y a bien entendu des similitudes entre la résolution en programmation par contraintes et celle en SAT/SMT, cependant, à la fois le modèle et les méthodes de résolution diffèrent. Les méthodes de résolution en SAT et SMT utilisent un modèle basé sur les variables booléennes, pour lesquelles les algorithmes sont dédiés. La programmation par contraintes combine les contraintes sur tout type de variables, et parfois perd en efficacité pour gagner en expressivité.

CHAPITRE 3

L'interprétation abstraite pour les contraintes

Dans ce chapitre nous définissons de façon unifiée les différents composants de la résolution en programmation par contraintes. Ces nouvelles définitions permettent l'élaboration d'une seule et même méthode de résolution ne dépendant plus du type des variables. De plus, la définition des domaines abstraits de l'interprétation abstraite en programmation par contraintes donne désormais la possibilité d'utiliser des représentations qui ne soient plus cartésiennes.

In this chapter we give unified definitions for the different components of the resolution process in Constraint Programming. With these new definitions an unique resolution method can be defined for any abstract domain. This new resolution scheme does not depend on the variables type anymore. Moreover, the definition of the abstract domains in Constraint Programming gives the possibility to solve problems using non-Cartesian domains representations.

3.1 Introduction

En programmation par contraintes, les techniques de résolution dépendent fortement du type des variables, et sont même dédiées à un type de variables (entier ou réels). Si un problème contient à la fois des variables entières et réelles, il n'existe plus de méthodes de résolution. Deux solutions sont possibles afin de tout de même résoudre ce type de problèmes avec la programmation par contraintes : soit les entiers sont transformés en réels et des contraintes d'intégrité sont ajoutées au problème (la propagation de ces nouvelles contraintes permet un raffinement aux bornes des variables entières [Granvilliers et Benhamou, 2006]) ; soit les réels sont discrétisés, les valeurs possibles pour les variables réelles sont énumérées avec un pas donné [Team, 2010].

En regardant de plus près on voit que quelle que soit la méthode de résolution utilisée, celle-ci alterne phase de propagation et exploration. On peut même aller plus loin et dire que ces deux méthodes ne diffèrent que sur trois points : la représentation des domaines, la consistance utilisée et la manière de couper les domaines. L'opérateur de coupe utilisé lors de la résolution dépend fortement de la représentation choisie, en effet il faut connaître certaines caractéristiques de la représentation choisie, telle que sa taille, afin de pouvoir la couper en plus petits éléments. De même la consistance utilisée dépend fortement de la représentation choisie. En effet, on n'utilisera pas la consistance d'arc généralisée si les domaines sont représentés avec des intervalles entiers. Si le domaine est représenté avec des intervalles d'entiers, la consistance de bornes est utilisée. Si le produit cartésien entier est utilisé pour représenter le domaine, alors la consistance d'arc généralisée est plus adéquate. Si les domaines sont représentés avec des intervalles réels à bornes flottantes, on utilise la consistance d'enveloppe. Enfin, la consistance ne dépend pas uniquement de la représentation mais aussi des contraintes.

En s'inspirant des domaines abstraits en interprétation abstraite, on définit de la même façon les domaines abstraits en programmation par contraintes, comme un objet contenant, entre autres, une représentation calculable en machine, un opérateur de coupe et une fonction permettant de calculer la taille de la représentation. La consistance pourrait être ajoutée aux domaines abstraits, mais nous avons préféré la dissocier étant donné qu'elle ne dépend pas uniquement de la représentation des domaines. Grâce à cette définition des domaines abstraits en programmation par contraintes on obtient une méthode de résolution unique ne dépendant plus ni du type des variables, ni de la représentation des domaines et pour laquelle le domaine abstrait utilisé est un paramètre. De plus, nous ne sommes plus restreints aux représentations cartésiennes existantes, mais pouvons en définir de nouvelles de la même façon qu'en interprétation abstraite.

3.2 Composants unifiés

Nous définissons dans un premier temps toutes les briques nécessaires à l'élaboration d'une méthode de résolution unique, qui sont la consistance, l'opérateur de coupe et bien entendu les domaines abstraits pour la programmation par contraintes. Ces définitions reposent sur des notions d'ordre, de treillis et de point fixe.

3.2.1 Consistance et point fixe

Étant donnés un ensemble partiellement ordonné et une contrainte, on peut définir l'élément consistant comme le plus petit élément de l'ensemble partiellement ordonné s'il existe. De même pour une conjonction de contraintes, l'ensemble des plus petits éléments pour chacune des

contraintes forme un ensemble partiellement ordonné et l'élément consistant peut alors être défini comme le plus petit élément de cet ensemble. De ce constat découlent les définitions et propositions suivantes.

Soit E un sous-ensemble de $\mathcal{P}(\hat{D})$, avec $\hat{D}$ l'espace de recherche initial, muni de l'inclusion comme ordre partiel. Cet ensemble correspond à la représentation choisie pour les domaines. Notons E^f pour $E \backslash \emptyset$. Par la suite, nous restreindrons les définitions aux cas où E est clos par intersection, car cela est suffisant pour les cas classiques. Notons que si E est clos par intersection alors il possède une plus grande borne inférieure et forme un treillis dirigé.

Définition 3.2.1 (E-Consistance)**.** Soit C une contrainte. Un élément e est E-consistant pour C si et seulement si il est le plus petit élément de E contenant les solutions de C, S_C, c'est-à-dire, le plus petit élément de

$$\mathbb{C}_C^E = \{e' \in E, S_C \subseteq e'\}$$

Cette définition intègre les principales consistances définies section 2.2.2, qu'elles soient discrètes ou continues. Les propositions suivantes associent à chaque consistance existante son sous-ensemble correspondant. Ces propositions assurent que les principales consistances existantes soient bien incluses dans la définition 3.2.1.

Proposition 3.2.1. *Soit $\mathbb{S}$ l'ensemble des produits cartésiens de sous-ensembles finis d'entiers. La $\mathbb{S}$-consistance est la consistance d'arc généralisée (GAC) (Définition 2.2.7).*

Démonstration.

GAC $\Longrightarrow$ $\mathbb{S}$-consistant.

Nous prouvons d'abord que GAC$\Longrightarrow\mathbb{S}$-consistant. Soit $D = D_1 \times \cdots \times D_n$ GAC pour une contrainte C. D contient évidemment toutes les solutions, il reste donc à prouver que c'est le plus petit élément de $\mathbb{S}$. Soit $D' \in \mathbb{S}$ strictement plus petit que D, il existe donc un i tel que $D'_i \subset D_i$ et un $v \in D_i \setminus D'_i$. Puisque $v \in D_i$ et D est GAC, il existe aussi $x_j \in D_j$ pour $j \neq i$ tel que $(x_1 \dots v \dots x_n)$ soit une solution. Cette solution n'est pas dans D' et donc $D' \subset D$ perd des solutions. ◇

$\mathbb{S}$-consistant $\Longrightarrow$ GAC.

Nous prouvons maintenant que $\mathbb{S}$-consistant $\Longrightarrow$ GAC. Soit $D = D_1 \times \cdots \times D_n$ $\mathbb{S}$-consistant pour une contrainte C. Soient $i \in [\![1, n]\!]$ et $v \in D_i$. Supposons par l'absurde que pour chaque $x_j \in D_j$, $(x_1 \dots v \dots x_n)$ ne soit pas une solution. On peut construire l'ensemble $D_1 \times \cdots \times (D_i \setminus \{v\}) \times \cdots \times D_n$ strictement plus petit que D contenant toutes les solutions. Donc D n'est pas le plus petit élément. Il ne peut donc pas exister de tels i et v, et D est GAC. ◇

Un domaine est GAC si et seulement si il est $\mathbb{S}$-consistant.

□

Proposition 3.2.2. *Soit $\mathbb{IB}$ l'ensemble des boîtes entières (produits cartésiens d'intervalles finis d'entiers). La $\mathbb{IB}$-consistance est la consistance de bornes (BC) (Définition 2.2.8).*

Démonstration.

BC $\Longrightarrow$ $\mathbb{IB}$-consistant.

Nous prouvons d'abord que BC $\Longrightarrow$ $\mathbb{IB}$-consistant. Soit $D = D_1 \times \cdots \times D_n$ BC pour une contrainte C. D contient évidemment toutes les solutions, il reste donc à prouver que c'est le plus petit élément de $\mathbb{IB}$. Soit $D' \in \mathbb{IB}$ strictement plus petit que D, il existe donc un i tel que $D'_i \subset D_i$. Soit v une des bornes de D_i telle que $v \notin D'_i$. Comme D est BC, il existe $x_j \in D_j$ pour $j \neq i$ tel que $(x_1 \dots v \dots x_n)$ est une solution. Cette solution n'est pas dans D' et donc $D' \subset D$ perd des solutions. ◇

$\mathbb{IB}$-*consistant* $\Longrightarrow$ *BC*.

Prouvons maintenant que $\mathbb{IB}$-consistant $\Longrightarrow$ BC. Soit $D = D_1 \times \cdots \times D_n$ $\mathbb{IB}$-consistant pour une contrainte C. Soit $D_i = [\![a_i, b_i]\!]$. Supposons par l'absurde que pour chaque $x_j \in D_j$, $(x_1 \dots a_i \dots x_n)$ ne soit pas une solution. On peut construire l'ensemble $D_1 \times \cdots \times [\![a_i + 1, b_i]\!] \times \dots D_n$ strictement plus petit que D contenant toutes les solutions. Donc D n'est pas le plus petit élément. Il ne peut donc pas exister de tel D_i, et D est BC. ◇

Un domaine est donc BC si et seulement si il est $\mathbb{IB}$-consistant.

□

Proposition 3.2.3. *Soit $\mathbb{B}$ l'ensemble des boîtes. La $\mathbb{B}$-consistance est la consistance d'enveloppe ou Hull-consistance (HC) (Définition 2.2.9).*

Démonstration.

Hull-consistant $\Longrightarrow$ $\mathbb{B}$-*consistant*.

Prouvons d'abord que Hull-consistant $\Longrightarrow$ $\mathbb{B}$-consistant. Soit $D = D_1 \times \cdots \times D_n$ Hull-consistant pour une contrainte C. D contient bien entendu toutes les solutions, nous devons prouver qu'il est bien le plus petit des tels éléments de $\mathbb{B}$. Soit $D' \in \mathbb{B}$, strictement plus petit que D, alors il existe un i tel que $D'_i \subset D_i$. Soit $I = D_i \setminus D'_i$, comme $I \in D_i$ et D est Hull-consistant il existe aussi un $I_j \in D_j$ pour $j \neq i$ tel que $C(I_1, \dots, I, \dots, I_n)$. Cette solution n'est pas dans D' et donc tout $D' \subset D$ perd des solutions. ◇

$\mathbb{B}$-*consistant* $\Longrightarrow$ *Hull-consistant*.

Prouvons maintenant que $\mathbb{B}$-consistant $\Longrightarrow$ Hull-consistant. Soit $I \in \mathbb{B}$, $\mathbb{B}$-consistant pour une contrainte C. D'où I est le plus petit élément de $\mathbb{C}_C^{\mathbb{B}}$ ce qui correspond au plus petit produit cartésien d'intervalles à bornes flottantes contenant toutes les solutions pour C. Donc, par définition 2.2.9, I est Hull-consistant. ◇

Un domaine est Hull-consistant si et seulement si il est $\mathbb{B}$-consistant.

□

La définition 3.2.1 de la E-consistance généralise ainsi les principales consistances existantes. Pour tout $E \subseteq \mathcal{P}(\hat{D})$ clos par intersection, la consistance est bien définie. De cette définition de E-consistance découle la proposition suivante :

Proposition 3.2.4. *Si E est clos par intersection infinie, $\mathbb{C}_C^E$ est un treillis complet et il existe un unique élément E-consistant pour C dans E. S'il existe, cet élément est noté $\mathbf{C}_C^E$.*

Démonstration.

Posons $\mathbf{C}_C^E = \bigcap_{e \in E, S_C \subseteq e} e$.

Plus petit élément.

Prouvons dans un premier temps que $\mathbf{C}_C^E$ est le plus petit élément pour $\mathbb{C}_C^E$. Soit $A \in \mathbb{C}_C^E$ strictement plus petit que $\mathbf{C}_C^E$, on a $S_C \subseteq A$. Comme $\mathbf{C}_C^E$ est l'intersection de tous les éléments contenant les solutions, A ne peut être strictement plus petit que $\mathbf{C}_C^E$. On en déduit que $\mathbf{C}_C^E$ est le plus petit élément de $\mathbb{C}_C^E$. ◇

Unicité.

Prouvons maintenant que $\mathbf{C}_C^E$ est unique. Supposons par l'absurde qu'il existe un élément $B \in \mathbb{C}_C^E$ tel que B soit aussi le plus petit élément pour $\mathbb{C}_C^E$. Si un tel élément existe, alors il contient toutes les solutions, $S_C \subseteq B$. Comme $\mathbf{C}_C^E$ est l'intersection de tous les éléments contenant S_C, on a $\mathbf{C}_C^E \subseteq B$. On en déduit que $\mathbf{C}_C^E$ est unique. ◇

D'où, $\mathbf{C}_C^E$ est l'unique plus petit élément pour $\mathbb{C}_C^E$.

□

Remarque 3.2.1 – Notons que si E n'est pas clos par intersection, alors le plus petit élément pour $\mathbb{C}_C^E$ n'existe pas toujours. Par exemple, dans le cas particulier où C est un cercle et E l'ensemble des polyèdres convexes, il n'existe pas de plus petit polyèdre contenant les solutions étant donné que les polyèdres ne sont pas clos par intersection.

La définition 3.2.1 peut facilement s'étendre à une conjonction de contraintes de la façon suivante :

Définition 3.2.2 (E-consistance pour une conjonction de contraintes)**.** Soit $e \in E$, e est E-consistant pour $C_1 \dots C_p$ (ou un sous-ensemble) si et seulement si c'est le plus petit élément de $\mathbb{C}^E_{C_1 \wedge \dots \wedge C_p} = \{e' \in E, S_{C_1 \wedge \dots \wedge C_p} \subseteq e'\}$. S'il existe, il est noté $\mathbf{C}^E_{C_1 \wedge \dots \wedge C_p}$. S'il n'y a pas d'ambiguïté $\mathbb{C}^E_{C_1 \wedge \dots \wedge C_p}$ est noté $\mathbb{C}^E$ et son plus petit élément $\mathbf{C}^E$.

On utilisera de préférence cette définition dans le cas où E est clos par intersection.

Proposition 3.2.5. *Si E est clos par intersection, alors $\mathbf{C}^E$ existe et est unique. De plus, l'ensemble de tous les $\mathbf{C}^E_{C_{i_1} \wedge \dots \wedge C_{i_k}}$ pour $i_1 \dots i_k \in [\![1, p]\!]$ forme un treillis complet pour l'inclusion et $\mathbf{C}^E_{C_1 \wedge \dots \wedge C_p}$ est le plus petit élément.*

Démonstration.

Unicité.

L'unicité de $\mathbf{C}^E$ vient directement de la proposition 3.2.4. ◇

Treillis complet.

Prouvons d'abord que l'ensemble de tous les $\mathbf{C}^E_{C_{i_1} \wedge \dots \wedge C_{i_k}}$ pour $i_1 \dots i_k \in [\![1, p]\!]$ forme un treillis pour l'inclusion. Soit $\mathbf{C}^E_{C_i}$ et $\mathbf{C}^E_{C_j}$ pour $i, j \in [\![1, p]\!]$ deux éléments quelconque de l'ensemble. Le couple $\{\mathbf{C}^E_{C_i}, \mathbf{C}^E_{C_j}\}$ admet une *glb*

$$\mathbf{C}^E_{C_i} \cap \mathbf{C}^E_{C_j} = \bigcap_{e \in E, S_{C_i} \subset e \vee S_{C_j} \subset e} e$$

et une *lub*

$$\mathbf{C}^E_{C_i} \cup \mathbf{C}^E_{C_j} = \bigcap_{e \in E, S_{C_i} \subset e, S_{C_j} \subset e} e$$

Tout couple $\{\mathbf{C}^E_{C_i}, \mathbf{C}^E_{C_j}\}$ admet une *lub* et une *glb*, on en déduit que l'ensemble de tous les $\mathbf{C}^E_{C_{i_1} \wedge \dots \wedge C_{i_k}}$ où $i_1 \dots i_k \in [\![1, p]\!]$ forme un treillis. ◇

Plus petit élément.

Prouvons maintenant que $\mathbf{C}^E_{C_i \wedge C_j}$ est le plus petit élément pour $\mathbf{C}^E_{C_i}, \mathbf{C}^E_{C_j}$, en d'autres termes, $\mathbf{C}^E_{C_i \wedge C_j}$ est inclus dans $\mathbf{C}^E_{C_i} \cap \mathbf{C}^E_{C_j}$. Tout d'abord prouvons que $\mathbf{C}^E_{C_i \wedge C_j} \subseteq \mathbf{C}^E_{C_i}$. Comme $S_{C_i \wedge C_j} = \{(s_1 \dots s_n) \in D, C_i(s_1 \dots s_n) \wedge C_j(s_1 \dots s_n)\}$ et $S_{C_i} = \{(s_1 \dots s_n) \in D, C_i(s_1 \dots s_n)\}$, on a $S_{C_i \wedge C_j} \subseteq S_{C_i}$. On en déduit que $\bigcap_{e \subseteq E, S_{C_i \wedge C_j} \subseteq e} e \subseteq \bigcap_{e \subseteq E, S_{C_i} \subseteq e} e$. Donc $\mathbf{C}^E_{C_i \wedge C_j} \subseteq \mathbf{C}^E_{C_i}$. De la même façon nous prouvons que $\mathbf{C}^E_{C_i \wedge C_j} \subseteq \mathbf{C}^E_{C_j}$. Comme $\mathbf{C}^E_{C_i \wedge C_j} \subseteq \mathbf{C}^E_{C_i}$ et $\mathbf{C}^E_{C_i \wedge C_j} \subseteq \mathbf{C}^E_{C_j}$, on en déduit que $\mathbf{C}^E_{C_i \wedge C_j} \subseteq \mathbf{C}^E_{C_i} \cap \mathbf{C}^E_{C_j}$. Par conséquent $\mathbf{C}^E_{C_i \wedge C_j}$ est le plus petit élément pour $\mathbf{C}^E_{C_i}, \mathbf{C}^E_{C_j}$. ◇

□

Cette proposition assure que, si chaque contrainte d'un CSP vient avec un propagateur, il suffit d'appliquer ce propagateur de manière itérative, quel que soit l'ordre, jusqu'à ce que le point fixe soit atteint, pour obtenir la consistance de ce CSP. Une proposition similaire a déjà été définie pour les consistances existantes dans [Apt, 1999] ou [Benhamou, 1996].

Exemple 3.2.1 – Considérons le CSP pour les variables réelles v_1 et v_2 de domaines $D_1 = D_2 = [0, 5]$ et les contraintes

$C1 : 5v_1 + v_2 \geq 10$
$C2 : 2v_1 + 5v_2 \leq 20$
$C3 : 2v_2 - v_1 \geq 1$

La figure 3.1 montre le treillis des $\mathbb{B}$-consistances de ce CSP pour l'inclusion. Pour chacun des éléments de ce treillis, on a en rose foncé l'ensemble des solutions, en rose clair l'ensemble approximé par la $\mathbb{B}$-consistance représentée par une boîte verte. Les boîtes noires en pointillés servent juste à encadrer chacun des éléments. Pour les éléments correspondants à des intersections, les boîtes sont dessinées en tirets.

On voit bien que $\forall i_1, \dots, i_k \in [\![1, 3]\!], \mathbf{C}^{\mathbb{B}}_{C_{i_1}}, \dots, \mathbf{C}^{\mathbb{B}}_{C_{i_k}} \subseteq \mathbf{C}^{\mathbb{B}}_{C_{i_1}} \cap \dots \cap \mathbf{C}^{\mathbb{B}}_{C_{i_k}} \subseteq \mathbf{C}^{\mathbb{B}}_{C_{i_1} \wedge \dots \wedge C_{i_k}}$. On a, par exemple, $\mathbf{C}^{\mathbb{B}}_{C_1}, \mathbf{C}^{\mathbb{B}}_{C_3} \subseteq \mathbf{C}^{\mathbb{B}}_{C_1} \cap \mathbf{C}^{\mathbb{B}}_{C_3} \subseteq \mathbf{C}^{\mathbb{B}}_{C_1 \wedge C_3}$.

3.2.2 Opérateur de coupe

Une fois l'élément consistant calculé, nous devons encore l'explorer afin de trouver les solutions. Ceci est fait en coupant l'espace de recherche restant en plus petits éléments. Un élément est coupé par un opérateur de coupe tel que défini ci-dessous.

Définition 3.2.3 (Opérateur de coupe dans E)**.** Soit $(E, \subseteq)$ un ensemble partiellement ordonné. Un *opérateur de coupe* est un opérateur $\oplus : E \to \mathcal{P}(E)$ tel que $\forall e \in E$,

1. $|\oplus(e)|$ est fini, $\oplus(e) = \{e_1, \dots, e_k\}$,
2. $\bigcup_{i \in [\![1,k]\!]} e_i = e$,
3. $\forall i \in [\![1, k]\!], e \neq \emptyset \Longrightarrow e_i \neq \emptyset$,
4. $\exists i \in [\![1, k]\!], e_i = e \Longrightarrow e$ est le plus petit élément de E^f.

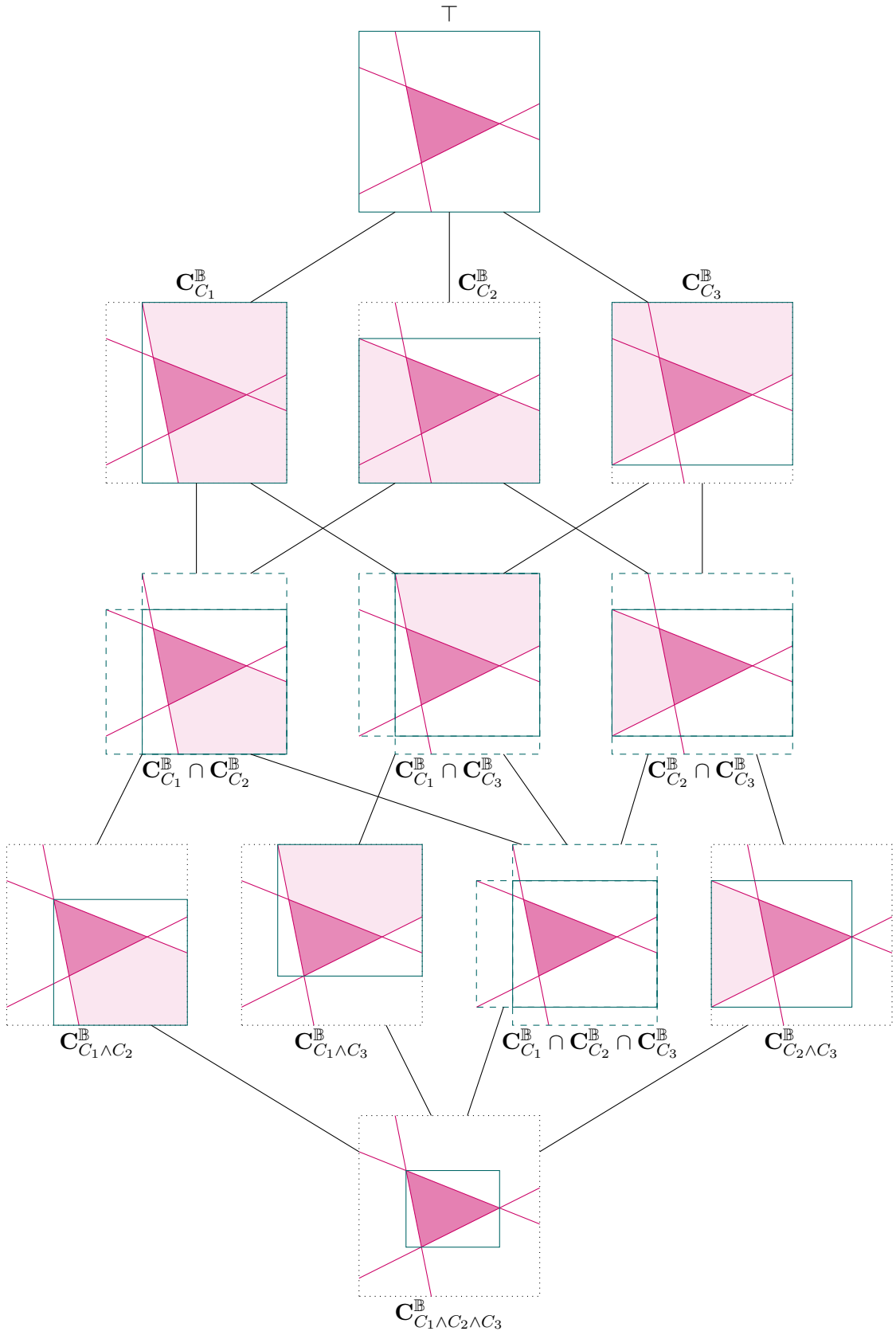


Figure 3.1 – Exemple de treillis fini pour l'inclusion, dont les éléments sont les $\mathbb{B}$-consistances pour le CSP donné dans l'exemple 3.2.1.

La première condition est nécessaire pour que la résolution termine. En effet, il est nécessaire que la largeur de l'arbre de recherche soit finie pour que l'arbre le soit. La deuxième condition assure que la résolution soit complète, c'est-à-dire que l'opérateur de coupe ne perde pas d'éléments. De plus, cette condition assure qu'aucun élément n'est ajouté. La troisième condition interdit à l'opérateur de coupe de retourner un élément vide. La dernière condition assure que l'opérateur coupe effectivement : il est interdit de conserver le même domaine.

Remarque 3.2.2 – Il est important de noter que la définition implique que si e n'est pas le plus petit élément de E^f alors $\forall i \in [\![1, k]\!], e_i \subsetneq e$.

Remarque 3.2.3 – Notons aussi que cette définition de l'opérateur de coupe n'exige pas que $\oplus(e)$ soit une partition de e. Ceci est nécessaire pour pouvoir inclure la coupe sur les intervalles à bornes flottantes et n'affecte en aucun cas la complétude de la résolution. Comme sur la figure 3.2, un opérateur de coupe est appliqué à l'élément de gauche (3.2(a)), donnant les deux éléments de droite (3.2(b)). L'élément de départ est bien inclus dans l'union des éléments obtenus par l'opérateur de coupe, et l'intersection de ces éléments est non nulle.

Nous montrons ci-dessous que l'instanciation discrète et la bissection en continu sont incluses dans la définition. Nous utiliserons les notations suivantes pour les domaines cartésiens : soit $\oplus_1 : E_1 \to \mathcal{P}(E_1)$ un opérateur pour un ensemble partiellement ordonné E_1. Soit E_2 un autre ensemble partiellement ordonné et Id la fonction identité pour E_2. On note $\oplus_1 \times Id$ l'opérateur sur $E_1 \times E_2$ tel que $\oplus_1 \times Id(e_1, e_2) = \cup_{e \in \oplus_1(e_1)} e \times e_2$. On notera aussi Id^i pour le produit cartésien de i fois Id.

Exemple 3.2.2 – L'instanciation d'une variable discrète est un opérateur de coupe sur $\mathcal{P}(\mathbb{Z})$: $\oplus_{\mathbb{Z}}(d) = \cup_{v \in d}\{v\}$. Pour chaque $i \in [\![1, n]\!]$, l'opérateur $\oplus_{\mathbb{Z}^n,i}(d) = Id^{i-1} \times \oplus_{\mathbb{Z}} \times Id^{n-i-1}$, avec $\oplus_{\mathbb{Z}}$, à la $i$ème place, est un opérateur de coupe. Cela revient à choisir une variable v_i et une valeur v dans D_i.

Exemple 3.2.3 – L'opérateur de coupe continu est défini sur $\mathcal{P}(\mathbb{I})$: $\oplus_{\mathbb{I}}(I) = \{I^{\vdash}, I^{\dashv}\}$ avec $I^{\vdash}$ et $I^{\dashv}$ deux sous-intervalles non vide de I tels que $I^{\vdash} \cup I^{\dashv} = I$ et ce quelle que soit la manière de gérer les erreurs d'arrondi sur les flottants. L'opérateur de coupe le plus courant pour les CSP continus, pour $I = [a, b]$, retourne $I^{\vdash} = [a, h]$ et $I^{\dashv} = [h, b]$ avec $h \in \mathbb{F}, h = \frac{a+b}{2}$ arrondi suivant la direction adéquate. Afin d'assurer la terminaison de l'opérateur de coupe, celui-ci s'arrête quand a et b sont deux flottants consécutifs.

Exemple 3.2.4 – L'opérateur de coupe usuel pour un produit cartésien d'intervalles est défini dans $\mathcal{P}(\mathbb{B})$. Soit $I \in \mathbb{B}, D = I_1 \times \cdots \times I_n$, l'opérateur choisit d'abord un intervalle $I_i, i \in [\![1, n]\!]$. Puis cet intervalle est coupé avec $\oplus_{\mathbb{I}}$. On a donc :

$$\oplus_{\mathbb{B}}(I) = \{I_1 \times \cdots \times I_i^{\vdash} \times \cdots \times I_n, I_1 \times \cdots \times I_i^{\dashv} \times \cdots \times I_n\}$$

avec $\{I_i^{\vdash}, I_i^{\dashv}\} = \oplus_{\mathbb{I}}(I_i)$. L'opérateur de coupe continu le plus courant en programmation par contraintes, coupe généralement le domaine réalisant le maximum de $\max(\overline{I_i} - \underline{I_i})$.

Par la suite, nous écrirons, pour tout E, $\oplus_E$ l'opérateur de coupe dans E.

Grâce aux définitions génériques de la consistance et de l'opérateur de coupe, nous pouvons maintenant définir les domaines abstraits en programmation par contraintes.

3.2.3 Domaines abstraits

Notre but étant de définir un solveur générique totalement indépendant de la représentation des domaines, nous définissons dans cette section les domaines abstraits pour la programmation par contraintes. Ceux-ci sont définis de sorte qu'ils aient toutes les propriétés requises pour être utilisés lors de la résolution.

Définition 3.2.4 (Domaine abstrait pour la programmation par contraintes)**.** Un *domaine abstrait* est défini par :

- un treillis complet E ;
- une concrétisation $\gamma : E \to D$, et une abstraction $\alpha : D \to E$ formant une correspondance de Galois entre E et l'espace de recherche ;
- une forme normale représentable en machine ;
- une séquence d'opérateurs de coupe sur E ;
- une fonction de précision strictement croissante $\tau : E \to \mathbb{R}^+$ telle que $\tau(e) = 0 \Leftrightarrow e = \emptyset$.

La correspondance de Galois relie le domaine abstrait à une représentation en machine des domaines du CSP. Cependant, comme pour les domaines abstraits en interprétation abstraite, elle peut ne pas exister. La forme normale assure qu'il existe une représentation unique pour chaque domaine abstrait, et que ceux-ci soient entres autres comparables. La fonction de précision τ correspond à une mesure du domaine abstrait. C'est un artifice technique pour exprimer la précision du domaine abstrait. Elle est utilisée dans la condition d'arrêt de la résolution.

Les domaines abstraits peuvent être définis indépendamment des domaines d'un CSP. Ils sont destinés à déterminer la forme de la représentation des domaines. Ils peuvent, bien entendu, être cartésiens, mais cela n'est plus obligatoire. Notons que nous n'avons pas défini les propagateurs comme partie intégrante d'un domaine abstrait. Les propagateurs dépendant à la fois de la forme de la représentation choisie et des contraintes, nous avons choisi de les définir à part de façon *ad hoc*.

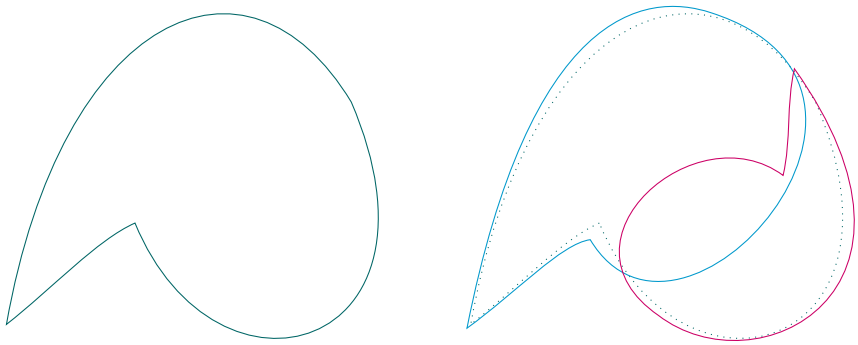
Grâce à cette définition, on peut par exemple définir le domaine abstrait Shadok [*] figure 3.2. On suppose bien entendu que l'on puisse le calculer, qu'il soit représentable en machine, et qu'il existe une consistance Shadok permettant de calculer, si il existe, le plus petit Shadok contenant les solutions. La figure 3.2(a) montre ce à quoi ressemblerait un Shadok en forme normale. Le résultat d'un possible opérateur de coupe est donné figure 3.2(b).

Les domaines abstraits étant désormais définis pour la programmation par contraintes, une méthode de résolution basée sur les domaines abstraits peut être définie.

3.3 Résolution unifiée

Grâce aux définitions d'une consistance (définition 3.2.1) et d'un opérateur de coupe (définition 3.2.3) ne dépendant plus de la représentation des domaines, et grâce à celle des domaines abstraits en programmation par contraintes, il est désormais possible de définir une résolution unifiée, basée sur les domaines abstraits. Cette méthode de résolution alterne toujours des phases de propagation et d'exploration mais n'est plus dédiée à un certain type de représentation.

*. http://www.lesshadoks.com/



(a) un Shadok (b) un exemple de coupe d'un Shadok

Figure 3.2 – On pourrait définir un domaine abstrait Shadok. Un Shadok en forme normale est donné (3.2(a)) ainsi que le résultat d'un possible opérateur de coupe (3.2(b))

liste de domaines abstraits sols $\leftarrow \emptyset$ */* stocke les solutions abstraites */*
queue de domaines abstraits toExplore $\leftarrow \emptyset$ */* stocke les éléments abstraits à explorer */*
domaine abstrait $e \in E$ */* une structure de données pour le domaine abstrait */*

$e = \alpha(\hat{D})$ */* initialisation aux domaines initiaux */*
push e dans toExplore

tant que toExplore $\neq \emptyset$ **faire**
  $e \leftarrow$ **pop**(toExplore)
  $e \leftarrow E$-consistance(e)
  si $e \neq \emptyset$ **alors**
    si $\tau(e) \leq r$ **ou** $e \subseteq S$ **alors**
      sols $\leftarrow$ sols $\cup\, e$
    sinon
      choisir un opérateur de coupe $\oplus_E$
      push $\oplus_E(e)$ dans toExplore
    fin si
  fin si
fin tant que

Algorithme 3.1: Résolution avec des domaines abstraits.

L'algorithme 3.1 donne le pseudo-code d'une méthode de résolution basée sur les domaines abstraits. Cet algorithme ressemble beaucoup à celui du solveur continu (algorithme 2.2). En effet, si on remplace partout dans le pseudo-code "domaine abstrait" par "boîte", on obtient un solveur continu. De même pour le produit cartésien entier.

La proposition suivante donne des hypothèses sous lesquelles on peut affirmer que cet algorithme termine et est complet.

Proposition 3.3.1. *Si E est clos par intersection (H1), n'a pas de chaîne infinie décroissante (H2) et si $r \in \tau(E^f)$ (H3), alors l'algorithme de résolution 3.1 termine et est complet, dans le sens où aucune solution n'est perdue.*

Démonstration.

Terminaison.

Prouvons que cet algorithme termine. Supposons par l'absurde que l'arbre de recherche est infini. D'après la définition 3.2.3 de l'opérateur de coupe, la largeur de l'arbre est finie. Il existe donc une branche infinie. Le long de cette branche, les domaines abstraits sont strictement décroissants tant que e n'est pas le plus petit élément de E^f. Par hypothèse (H2), il existe un K tel que $\forall k \geq K, e_k = e_K$. Étudions les différents cas possibles :

— si $e_K = \emptyset$ alors l'algorithme termine car e_K ne contient pas de solution ;

— si $e_K \neq \emptyset$ et $(\tau(e_K) \leq r$ ou $e_K \subset S)$, alors l'algorithme termine car e_K est une solution ou ne contient que des solutions ;

— si $e_K \neq \emptyset$ et $\tau(e_K) > r$ alors e_K est coupé et $e_K \neq e_{K+1}$ ce qui contredit la définition de K.

On peut donc en conclure que la résolution proposée dans l'algorithme 3.1 termine. ◇

Correction.

L'opérateur de coupe (définition 3.2.3) et la consistance (définition 3.2.1) étant complets, l'algorithme 3.1 est complet. ◇

Sous les hypothèses (H1), (H2) et (H3), l'algorithme de résolution 3.1 termine et est complet.

□

Cette proposition définit un solveur générique pour n'importe quel domaine abstrait E sous certaines conditions. En effet, il faut que les hypothèses (H1) et (H2) soient vérifiées et que r soit bien choisi. Si r est trop grand, les éléments abstraits retournés par cette résolution seront grands et peuvent contenir un grand nombre de points qui ne soient pas des solutions. Au contraire, si r est trop petit, la résolution mettra beaucoup de temps et pire encore, dans le cas des produits cartésiens entiers en fonction de la fonction de précision τ, les solutions peuvent ne pas être trouvées. L'efficacité de ce solveur dépend bien entendu des algorithmes de consistance de E et de la représentation en machine du domaine abstrait choisi.

Les algorithmes de résolution habituels en programmation par contraintes peuvent être retrouvés comme montré ci-dessous.

Exemple 3.3.1 – Soit n fixé, l'ensemble $\mathbb{S}$ avec l'opérateur de coupe $\oplus_{\mathbb{N}^n,i}$ et la précision $\tau_{\mathbb{S}}(e) = \max(|X_i|)$ pour $i \in [\![1, n]\!]$ est un domaine abstrait vérifiant (H1) et (H2). Afin de modéliser le fait que la résolution termine quand toutes les variables sont instanciées, on peut prendre $r = 1$. Comme vu précédemment, la $\mathbb{S}$-consistance correspond à la consistance d'arc généralisée.

Exemple 3.3.2 – Soit n fixé, l'ensemble $\mathbb{IB}$ avec l'opérateur de coupe $\oplus_{\mathbb{N}^n,i}$ et la précision $\tau_{\mathbb{IB}}(e) = \max(b_i - a_i)$ pour $i \in [\![1, n]\!]$ est un domaine abstrait vérifiant (H1) et (H2). Afin de modéliser le fait que la résolution termine quand toutes les variables sont instanciées, on peut prendre $r = 1$. Comme vu précédemment, la $\mathbb{IB}$-consistance correspond à la consistance d'arc généralisée.

Exemple 3.3.3 – Soit n fixé, l'ensemble $\mathbb{B}$ avec l'opérateur de coupe $\oplus_{\mathbb{B},i}$ et la précision $\tau_{\mathbb{B}}(I) = \max(\overline{I_i} - \underline{I_i})$ pour $i \in [\![1, n]\!]$ est un domaine abstrait vérifiant (H1) et (H2). Afin de modéliser le fait que la résolution termine quand une certaine précision r est atteinte, on peut s'arrêter quand $\tau_{\mathbb{B}} \leq r$. La résolution sur $\mathbb{B}$ correspond à celle utilisée généralement dans les solveurs continus avec la Hull-consistance.

Ces trois exemples montrent comment retrouver les domaines abstraits de base, qui sont cartésiens et d'un seul type (produit cartésien d'entiers ou d'intervalles).

Il est maintenant possible de définir un solveur abstrait comme une alternance de propagations et de coupes, voir figure 3.1. Ce solveur retourne un tableau de domaines abstraits ne contenant que des solutions ($e \subseteq S$) ou dont la précision est plus petite que r. En faisant l'union de tous ces éléments, on obtient une approximation de l'ensemble des solutions.

3.4 Conclusion

Dans ce chapitre nous avons montré qu'il était possible de définir un solveur indépendamment de la représentation des domaines et donc du type des variables du problème à résoudre. Ce solveur inclut les résolutions habituelles de la programmation par contraintes, et ce quel que soit le type des variables (discrètes ou continues). En outre, les domaines ne sont plus limités à des représentations cartésiennes. Grâce à cette nouvelle définition, il nous est maintenant possible d'utiliser différentes représentations pour les domaines. Il en existe un grand nombre en interprétation abstraite telles que les polyèdres, les ellipsoïdes et les zonotopes, pour n'en nommer que quelques unes. Dans le prochain chapitre nous détaillerons une représentation non-cartésienne : les octogones.

CHAPITRE 4

Les octogones

L'algorithme de résolution générique présenté dans le chapitre précédent permet d'utiliser les domaines abstraits lors de la résolution d'un problème de satisfaction de contraintes. En nous inspirant du domaine abstrait des octogones introduit par Miné [Miné, 2006] pour l'interprétation abstraite, nous définissons le domaine abstrait des octogones en programmation par contraintes. Pour cela nous définissons dans un premier temps une forme représentable en machine, un opérateur de coupe et une fonction de précision pour les octogones, puis une correspondance de Galois entre les octogones et les boîtes à bornes flottantes.

In the generic solver presented in the previous chapter, abstract domains can be used to solve constraints satisfaction problems. In the same way as in Abstract Interpretation, we define an octagon abstract domain for Constraint Programming. This abstract domain already exists in Abstract Interpretation, it has been introduced by Miné [Miné, 2006]. We first define a representation that is computer representable, a splitting operator and a precision function. Then a Galois connection between the octagons and the boxes is given.

4.1 Définitions

En géométrie, un octogone est, dans $\mathbb{R}^2$, un polygone à huit côtés *. Dans le cadre de ces travaux, nous utilisons une définition plus générale qui fut introduite dans [Miné, 2006]. Par la suite le terme “octogone” est employé pour désigner les octogones tels que définis ci-dessous, quand on voudra parler des octogones tels que décrits en géométrie, le terme “octogone mathématique” sera utilisé.

Définition 4.1.1 (Contrainte octogonale)**.** Soient deux variables v_i et v_j. On appelle *contrainte octogonale* les contraintes de la forme $\pm v_i \pm v_j \leq c$ avec $c \in \mathbb{R}$ une constante.

Remarque 4.1.1 – Les contraintes d'intervalle ($v_i \geq a$, $v_i \leq b$) sont des cas particuliers des contraintes octogonales.

Remarque 4.1.2 – Deux contraintes octogonales d'une conjonction de contraintes sont dites redondantes si et seulement si elles possèdent la même partie gauche. Par exemple, les contraintes suivantes $v_1 - v_2 \leq c$ et $v_1 - v_2 \leq c'$ sont redondantes. Seule l'une de ces contraintes est effective, celle avec la plus petite constante.

Par exemple dans $\mathbb{R}^2$, les contraintes octogonales définissent des droites parallèles aux axes si $i = j$ et diagonales du plan si $i \neq j$. Ceci reste vrai dans $\mathbb{R}^n$, où les contraintes octogonales définissent des demi-espaces (coupés par des hyperplans). Considérons un cube dans $\mathbb{R}^3$, si on ajoute des contraintes octogonales ($\pm v_i \pm v_j \leq c$) celles-ci vont couper les arrêtes du cube, mais ne coupent pas les coins.

Définition 4.1.2 (Octogone)**.** Un octogone est un ensemble de points de $\mathbb{R}^n$ satisfaisant une conjonction de contraintes octogonales.

Remarque 4.1.3 – La forme géométrique définie ci-dessus inclut les octogones mais aussi d'autres polygones. Par exemple, dans $\mathbb{R}^2$, un octogone peut avoir moins de huit côtés. De manière générale, dans $\mathbb{R}^n$, un octogone a au plus $2n^2$ côtés, ce qui correspond au nombre maximum de contraintes octogonales non-redondantes pour n variables. De plus, les octogones satisfaisant une conjonction de contraintes octogonales sont forcément convexes.

Notons aussi qu'un octogone est un ensemble de points *réels* mais, comme pour les intervalles, on peut se restreindre au cas où les bornes sont des flottants ($c \in \mathbb{F}$). On a donc un octogone réel à bornes flottantes.

La première partie de la figure 4.2 est composée d'exemples d'octogones (4.2(a)). On voit qu'un octogone n'a pas forcément 8 côtés en 2 dimensions et que tous ses côtés sont soit parallèles aux axes, soit parallèles aux diagonales. La seconde (4.2(b)) quant à elle montre des polygones ne correspondant pas à la définition des octogones donnée précédemment (définition 4.1.2). Le premier exemple a bien tous ses côtés parallèles aux axes ou aux diagonales mais n'est pas convexe. Les deux autres exemples sont des octogones mathématiques mais leurs côtés n'étant pas tous parallèles aux axes ou aux diagonales, ils ne correspondent pas à la définition 4.1.2.

Cette définition des octogones offre des propriétés intéressantes, n'existant pas pour les octogones mathématiques, telles que la clôture par intersection. En effet, l'intersection de deux octogones quelconque est aussi un octogone.

*. http://mathworld.wolfram.com/Octagon.html

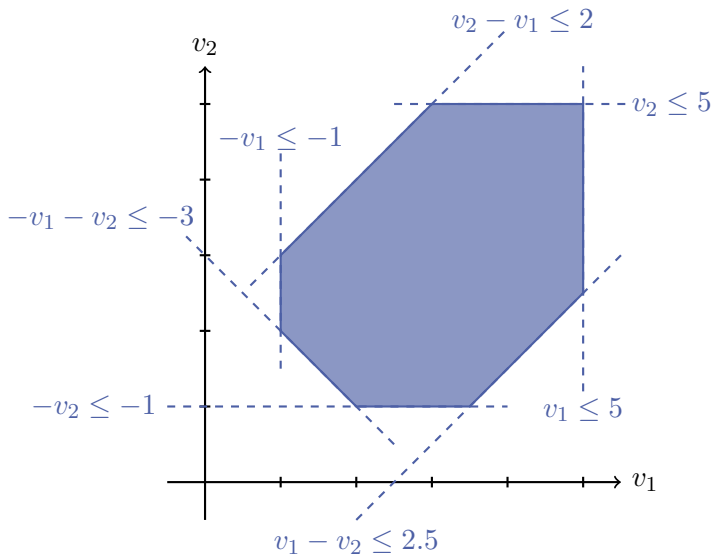
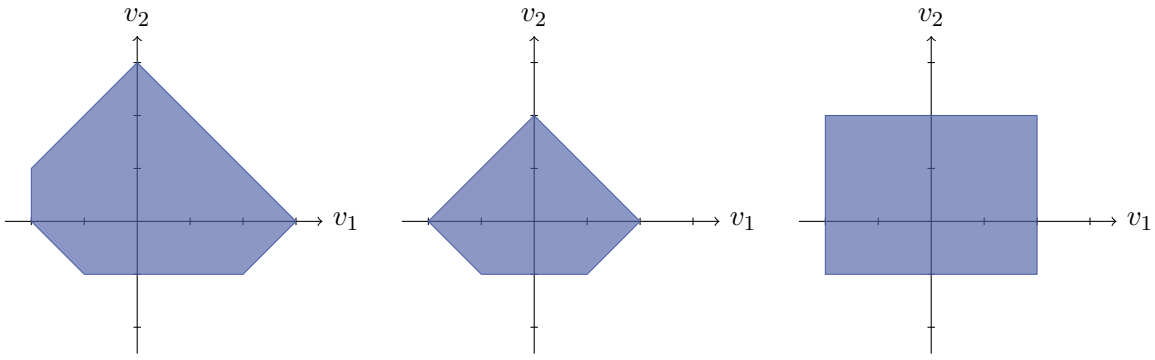
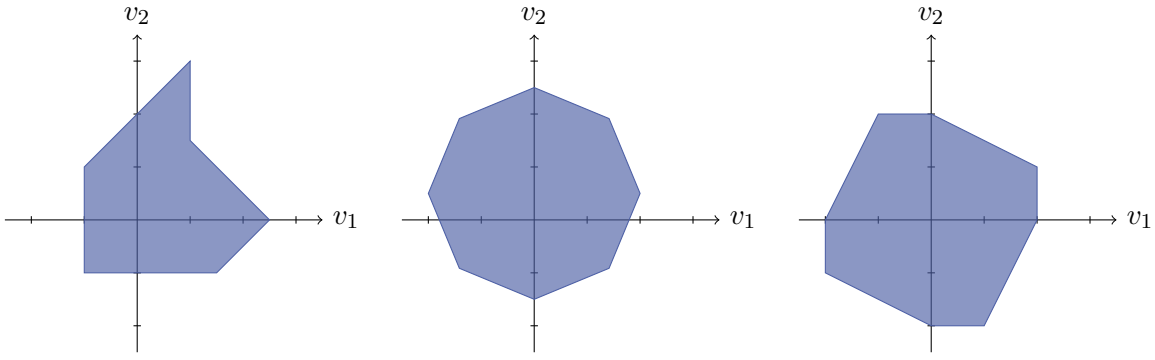


Figure 4.1 – Exemple d'un octogone dans $\mathbb{R}^2$.



(a) Exemples d'octogones



(b) Exemples de polygone ne correspondant pas à la définition 4.1.2

Figure 4.2 – Exemples d'octogones 4.2(a) et de polygones ne respectant pas la définition des octogones 4.2(b).

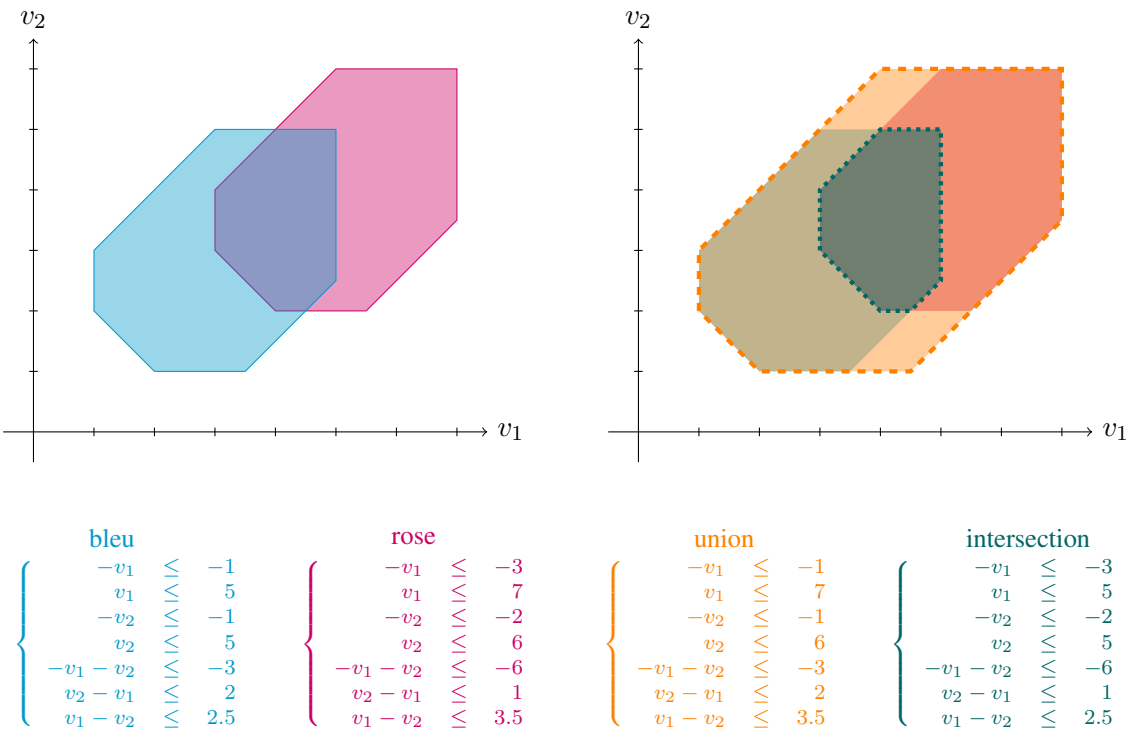


Figure 4.3 – Exemple de l'union et de l'intersection de deux octogones. L'union de l'octogone bleu et rose donne l'octogone orange (tirets) et l'intersection donne l'octogone vert (en pointillés).

Remarque 4.1.4 – Les octogones sont clos par intersection. Soit deux octogones quelconques $\mathbf{O} = \{\pm v_i \pm v_j \leq c\}$ et $\mathbf{O}' = \{\pm v_i \pm v_j \leq c'\}$ pour $i, j \in [\![1, n]\!]$. Leur intersection est aussi un octogone.

$$\mathbf{O} \cap \mathbf{O}' = \{\pm v_i \pm v_j \leq \min(c, c')\}$$

Les octogones ne sont pas clos par union, mais le plus petit octogone incluant l'union de deux octogones quelconques peut être facilement calculé.

Remarque 4.1.5 – Soient deux octogones quelconques $\mathbf{O} = \{\pm v_i \pm v_j \leq c\}$ et $\mathbf{O}' = \{\pm v_i \pm v_j \leq c'\}$ pour $i, j \in [\![1, n]\!]$. Le plus petit octogone incluant leur union est :

$$\mathbf{O} \cup \mathbf{O}' = \{\pm v_i \pm v_j \leq \max(c, c')\}$$

La figure 4.3 donne, géométriquement et à l'aide des conjonctions de contraintes, l'union et l'intersection de l'octogone rose et l'octogone bleu. L'octogone orange (tirets) correspond à l'octogone incluant l'union des octogones rose et bleu. L'octogone vert (pointillés) correspond quant à lui à l'intersection des octogones rose et bleu.

Dans la suite de ce chapitre, les octogones sont restreints aux octogones à bornes flottantes ($c \in \mathbb{F}$). De plus, nous considérons que les octogones sont définis sans redondance et ce sans perte de généralité.

4.2 Représentations

Une caractéristique nécessaire des domaines abstraits est que leurs éléments soient représentables en machine. Il n'existe pas d'unique façon de représenter un domaine abstrait, cependant certaines représentations sont mieux adaptées pour certains types de calculs. Nous détaillons ici deux représentations possibles pour les octogones. La première, la représentation matricielle, a été introduite en 1983 par Menasche et Berthomieu [Menasche et Berthomieu, 1983], elle est utilisée dans [Miné, 2006]. La seconde, la représentation par intersection de boîtes, fait partie des contributions de cette thèse et est plus adaptée pour certaines opérations en programmation par contraintes, telles que la consistance.

4.2.1 Représentation matricielle

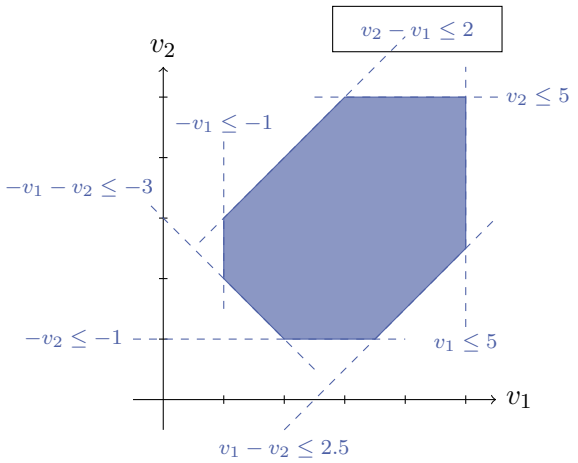
Les octogones peuvent être représentés à l'aide d'une *matrice à différences bornées* (DBM) comme décrit dans [Menasche et Berthomieu, 1983, Miné, 2006]. Cette représentation est basée sur une normalisation des contraintes octogonales comme suit.

Définition 4.2.1 (Contrainte de potentiel)**.** Soient w, w' deux variables. Une *contrainte de potentiel* est une contrainte de la forme $w - w' \leq c$ avec $c \in \mathbb{F}$ une constante.

En introduisant de nouvelles variables, il est possible de ré-écrire les contraintes octogonales sous forme de contraintes de potentiel. Soit $C \equiv (\pm v_i \pm v_j \leq c)$ une contrainte octogonale. De nouvelles variables w_{2i-1}, w_{2i} sont introduites telles que w_{2i-1} corresponde à la forme positive de v_i et w_{2i} à la forme négative de v_i, c'est-à-dire, $\forall i \in [\![1, n]\!], w_{2i-1} = v_i$ et $w_{2i} = -v_i$. On en déduit :

- pour $i = j$
  - si $C \equiv (v_i - v_i \leq c)$, alors
    - si $c \geq 0$, C est inutile et peut être supprimée,
    - sinon l'octogone correspondant est vide et on s'arrête. En effet, il n'existe pas de valeur de v_i telle que la contrainte $v_i - v_i < 0$ soit satisfaite,
  - si $C \equiv (v_i + v_i \leq c)$, alors C est équivalente à la contrainte de potentiel $(w_{2i-1} - w_{2i} \leq c)$,
  - si $C \equiv (-v_i - v_i \leq c)$, alors C est équivalente à la contrainte de potentiel $(w_{2i} - w_{2i-1} \leq c)$,
- pour $i \neq j$
  - si $C \equiv (v_i - v_j \leq c)$, alors C est équivalente aux contraintes de potentiel $(w_{2i-1} - w_{2j-1} \leq c)$ et $(w_{2j} - w_{2i} \leq c)$,
  - si $C \equiv (v_i + v_j \leq c)$, alors C est équivalente aux contraintes de potentiel $(w_{2i-1} - w_{2j} \leq c)$ et$(w_{2j-1} - w_{2i} \leq c)$,
  - si $C \equiv (-v_i - v_j \leq c)$, alors C est équivalente aux contraintes de potentiel $(w_{2i} - w_{2j-1} \leq c)$ et $(w_{2j} - w_{2i-1} \leq c)$,
  - si $C \equiv (-v_i + v_j \leq c)$, alors C est équivalente aux contraintes de potentiel $(w_{2i} - w_{2j} \leq c)$ et$(w_{2j-1} - w_{2i-1} \leq c)$.

Par la suite, les variables originales seront notées $(v_1 \dots v_n)$ et les nouvelles variables correspondantes $(w_1, w_2, \dots w_{2n})$ avec $w_{2i-1} = v_i$ et $w_{2i} = -v_i$. Miné montre dans [Miné, 2006] que



(a)

$$\begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{array} \begin{array}{c} \begin{array}{cccc} w_1 & w_2 & w_3 & w_4 \end{array} \\ \begin{pmatrix} 0 & -2 & \boxed{2} & -3 \\ 10 & 0 & +\infty & 2.5 \\ 2.5 & -3 & 0 & -2 \\ +\infty & \boxed{2} & 10 & 0 \end{pmatrix} \end{array}$$

(b)

Figure 4.4 – Représentations équivalentes d'un même octogone : avec les contraintes octogonales 4.4(a) et la matrice à différences bornées 4.4(b).

les contraintes de potentiel créées, en remplaçant les occurrences positives et négatives de la variable v_i par les w_i associés, représentent le même octogone que celui correspondant à la conjonction de contraintes octogonales. Les contraintes de potentiel pouvant être stockées dans une matrice à différences bornées et les contraintes octogonales pouvant être traduites en contraintes de potentiel, on en déduit qu'une matrice à différences bornées est une représentation possible pour les octogones.

Définition 4.2.2 (Matrice à différences bornées)**.** Soit $\mathbf{O}$ un octogone dans $\mathbb{F}^n$ et son ensemble de contraintes de potentiel comme décrit précédemment. La matrice à différences bornées est une matrice carrée de $2n \times 2n$ telle que l'élément à la ligne i, colonne j soit la constante c de la contrainte de potentiel $w_j - w_i \leq c$.

Un exemple est présenté Figure 4.4(b), l'octogone représenté graphiquement (4.4(a)) correspond à la matrice à différences bornées (4.4(b)). L'élément à la ligne 1 et colonne 3, et l'élément à la ligne 4 et colonne 2 correspondent à la même contrainte : $v_2 - v_1 \leq 2$.

À ce stade, différentes matrices à différences bornées peuvent représenter le même octogone. Par exemple Figure 4.4(b), l'élément ligne 2 et colonne 3 peut être remplacé, par exemple par 100, sans modifier l'octogone correspondant. Dans [Miné, 2006], un algorithme est défini afin de calculer de façon optimale les plus petites valeurs pour les cellules de la matrice à différences bornées. Cet algorithme est une adaptation de l'algorithme de plus court chemin de Floyd-Warshall [Floyd, 1962], modifié afin de prendre en compte la structure de la matrice à différences bornées. Il exploite le fait que w_{2i-1} et w_{2i} correspondent à la même variable.

L'algorithme 4.1 donne le pseudo-code de la version modifiée de l'algorithme de plus court chemin de Floyd-Warshall pour les octogones. L'instruction $i' \leftarrow (i \bmod 2 = 0)\,?\,i - 1 : i + 1$ signifie : si i est pair alors i' prend la valeur de $i - 1$, sinon i' prend la valeur de $i + 1$. Le début de cet algorithme correspond à celui de Floyd-Warshall (la partie calcul du plus court chemin). La suite a par contre été ajoutée et exploite le fait que w_{2i-1} et w_{2i} correspondent à la même variable v_i. De plus les éléments sur la diagonale descendante doivent être supérieurs ou égaux à zéro,

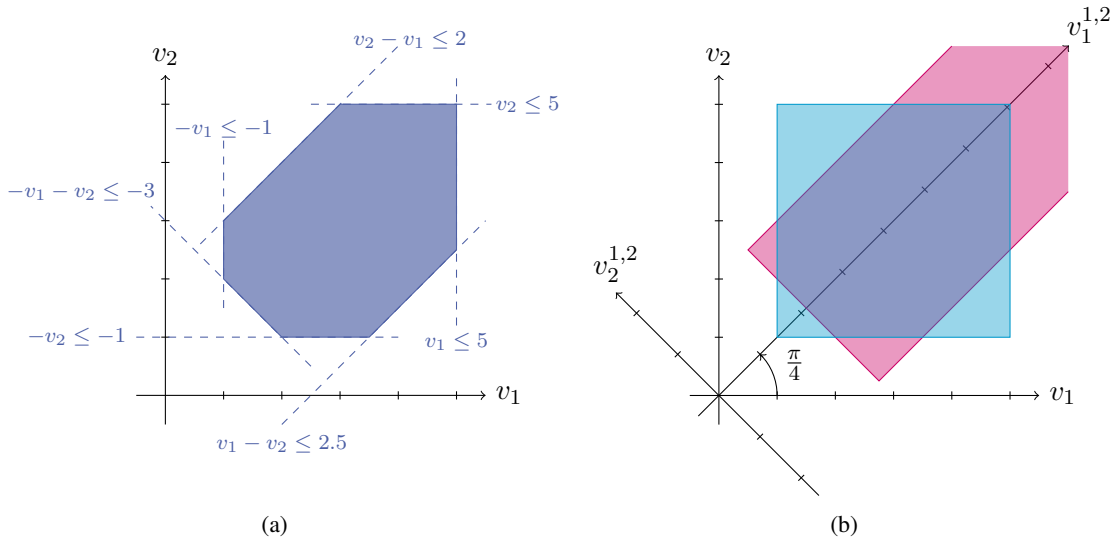
```
float dbm[2n][2n]                          /* Matrice à différences bornées de 2n × 2n */
int i, j, k, i', j'                        /* indices de parcours de la matrice à différences bornées */

pour k de 1 à n faire
  pour i de 1 à 2n faire
    pour j de 1 à 2n faire
      /* Calcul du plus court chemin */
      dbm[i][j] ← min( dbm[i][j], dbm[i][2k] + dbm[2k][j],
                       dbm[i][2k − 1] + dbm[2k − 1][j],
                       dbm[i][2k − 1] + dbm[2k − 1][2k] + dbm[2k][j],
                       dbm[i][2k] + dbm[2k][2k − 1] + dbm[2k − 1][j] )
    fin pour
  fin pour

  /* Ajout par rapport à la version originale */
  pour i de 1 à 2n faire
    pour j de 1 à 2n faire
      i' ← (i mod 2 = 0) ? i − 1 : i + 1
      j' ← (j mod 2 = 0) ? j − 1 : j + 1
      dbm[i][j] ← min(dbm[i][j], dbm[i][i'] + dbm[j'][j])
    fin pour
  fin pour
fin pour
pour i de 1 à 2n faire
  si dmb[i][i] < 0 alors
    retourner erreur
  sinon
    dbm[i][i] ← 0
  fin si
fin pour
```

Algorithme 4.1: Version modifiée de l'algorithme de plus court chemin de Floyd-Warshall pour les octogones.

sinon une erreur est retournée car l'octogone correspondant est vide. En effet, $\forall i, w_i - w_i \leq c$, c ne peut être inférieur strictement à 0. Une autre version de l'algorithme 4.1 est proposée dans [Bagnara *et al.*, 2009]. Ces deux versions ont la même complexité ($O(n^3)$).

L'exécution de cette version modifiée de Floyd-Warshall sur la matrice à différences bornées donnée figure 4.4(b) remplace les deux $+\infty$ par 10, ce qui correspond à ajouter la contrainte $v_1 + v_2 \leq 10$ à la conjonction de contraintes octogonales. Notons que l'exécution de l'algorithme de Floyd-Warshall remplace les $+\infty$ par 12.

Nous introduisons une nouvelle représentation pour les octogones basée sur celle des boîtes (définition 2.2.4). Cette représentation combinée avec une matrice à différences bornées sera utilisée pour définir, à partir d'un ensemble initial de contraintes continues, un système équivalent tenant compte des domaines octogonaux.



(a) (b)

Figure 4.5 – Représentations équivalentes d'un même octogone : avec les contraintes octogonales 4.5(a) et l'intersection de boîtes 4.5(b).

4.2.2 Représentation par intersection de boîtes

En deux dimensions, un octogone peut être représenté par l'intersection d'une boîte dans la base canonique $\mathbb{F}^2$ et d'une boîte dans une base obtenue par rotation d'angle $\pi/4$ de la base canonique. La figure 4.5 donne la représentation par intersection de boîtes (figure 4.5(b)) correspondant à l'octogone illustré figure 4.5(a). Dans cet exemple, l'octogone étant défini par 7 contraintes octogonales, l'une des boîtes de l'intersection se retrouve non bornée.

Afin de généraliser cette remarque à de plus grande dimensions, nous introduisons la notion de base tournée comme suit.

Définition 4.2.3 (Base tournée)**.** Soit $B = (u_1, \ldots, u_n)$ la base canonique de $\mathbb{F}^n$ et soit $\alpha = \pi/4$. La *(i,j)-base tournée* $B_\alpha^{i,j}$, pour $i, j \in [\![1, n]\!]$, correspond à la base canonique après rotation de α dans le plan défini par (u_i, u_j), les autres vecteurs restent inchangés :

$$B_\alpha^{i,j} = (u_1, \ldots, u_{i-1}, (\cos(\alpha)u_i + \sin(\alpha)u_j), \ldots u_{j-1}, (-\sin(\alpha)u_i + \cos(\alpha)u_j), \ldots u_n)$$

Par convention, pour tout $i \in [\![1, n]\!]$, $B_\alpha^{i,i}$ représente la base canonique. Par la suite, α étant toujours égal à $\pi/4$, il sera omis. Finalement, pour $i, j, k \in [\![1, n]\!]$, chaque variable v_k de domaine est D_k dans la base tournée $B^{i,j}$ sera notée $v_k^{i,j}$ et son domaine $D_k^{i,j}$.

Remarque 4.2.1 – Notons que l'idée des boîtes tournées n'est pas nouvelle, elle a déjà été proposée dans [Goldsztejn et Granvilliers, 2010] afin de calculer une meilleure approximation de l'ensemble des solutions. Cependant, la méthode proposée est dédiée aux systèmes d'équations.

La matrice à différences bornées peut aussi être interprétée comme une représentation de l'intersection de la boîte canonique et des $n(n-1)/2$ autres boîtes, chacune étant dans une base tournée.

Soit $\mathbf{O}$ un octogone dans $\mathbb{F}^n$ et M sa matrice à différences bornées, avec la même notation que précédemment (M est une matrice carrée $2n \times 2n$). Pour $i, j \in [\![1, n]\!]$, avec $i \neq j$, on note $\mathbf{B}_{\mathbf{O}}^{i,j}$ la boîte $I_1 \times \cdots \times I_i^{i,j} \times \cdots \times I_j^{i,j} \cdots \times I_n$, dans la base $B^{i,j}$, telle que $\forall k \in [\![1, n]\!]$

$$\begin{aligned} I_k &= [\quad \underline{-\tfrac{1}{2}M[2k-1, 2k]}\ ,\ \overline{\tfrac{1}{2}M[2k, 2k-1]}] \\ I_i^{i,j} &= [\quad \underline{-\tfrac{1}{\sqrt{2}}M[2j-1, 2i]}\ ,\ \overline{\tfrac{1}{\sqrt{2}}M[2j, 2i-1]}] \\ I_j^{i,j} &= [\underline{-\tfrac{1}{\sqrt{2}}M[2j-1, 2i-1]}\ ,\quad \overline{\tfrac{1}{\sqrt{2}}M[2j, 2i]}] \end{aligned}$$

Exemple 4.2.1 – On considère la DBM sur la figure 4.4(b), les boîtes sont $I_1 \times I_2 = [1, 5] \times [1, 5]$ et $I_1^{1,2} \times I_2^{1,2} = \left[\underline{3/\sqrt{2}}, +\infty\right] \times \left[\underline{-2.5/\sqrt{2}}, \overline{\sqrt{2}}\right]$, ce qui correspond aux boîtes de la figure 4.5(b).

Proposition 4.2.1. *Soient* $\mathbf{O}$ *un octogone dans* $\mathbb{F}^n$ *et* $\mathbf{B}_{\mathbf{O}}^{i,j}$ *les boîtes comme définies précédemment. On a* $\mathbf{O} = \bigcap_{i,j \in [\![1,n]\!]} \mathbf{B}_{\mathbf{O}}^{i,j}$.

Démonstration.

Soient $i, j \in [\![1, n]\!]$. On a $v_i^{i,j} = \frac{1}{\sqrt{2}(v_i + v_j)}$ et $v_j^{i,j} = \frac{1}{\sqrt{2}(v_j - v_i)}$ par définition 4.2.3. On en déduit $(v_1 \dots v_i^{i,j} \dots v_j^{i,j} \dots v_n) \in \mathbf{B}_{\mathbf{O}}^{i,j}$ si et seulement si les contraintes octogonales sur v_i et v_j et les contraintes unaires sur les autres coordonnées sont satisfaites dans la matrice à différences bornées. La boîte $\mathbf{B}_{\mathbf{O}}^{i,j}$ est donc l'ensemble des solutions pour les contraintes octogonales en question. Les points dans $\bigcap_{i,j \in [\![1,n]\!]} \mathbf{B}_{\mathbf{O}}^{i,j}$ sont exactement les points satisfaisants toutes les contraintes octogonales.

□

En résumé, il existe différentes représentations pour un octogone. Il peut être représenté à l'aide d'une matrice à différences bornées interprétée comme un ensemble de contraintes octogonales. Mais également comme l'intersection de plusieurs boîtes. De plus, le passage d'une représentation à l'autre est réalisé au prix d'une multiplication/division avec le bon mode d'arrondi.

Nous avons maintenant des formes représentables en machine pour les octogones. Afin de définir le domaine abstrait des octogones, il nous faut maintenant définir certains opérateurs tels que l'opérateur de coupe et la fonction de précision pour les octogones. Par la suite, nous supposons que les octogones sont clos par l'algorithme modifié de Floyd-Warshall.

4.3 Composants du domaine abstrait

Lors de la seconde phase de résolution (l'exploration), si l'élément abstrait courant n'est pas considéré comme une solution, un opérateur de coupe est utilisé afin de couper l'élément abstrait courant en plus petits éléments abstraits. Afin de résoudre à l'aide d'octogones, un opérateur de coupe est nécessaire de même qu'une fonction de précision permettant de calculer d'une certaine façon la taille de l'octogone. Pour définir ces opérateurs, nous utilisons la représentation par intersection de boîtes. Bien entendu ceux-ci peuvent aussi être définis de façon équivalente sur la matrice à différences bornées.

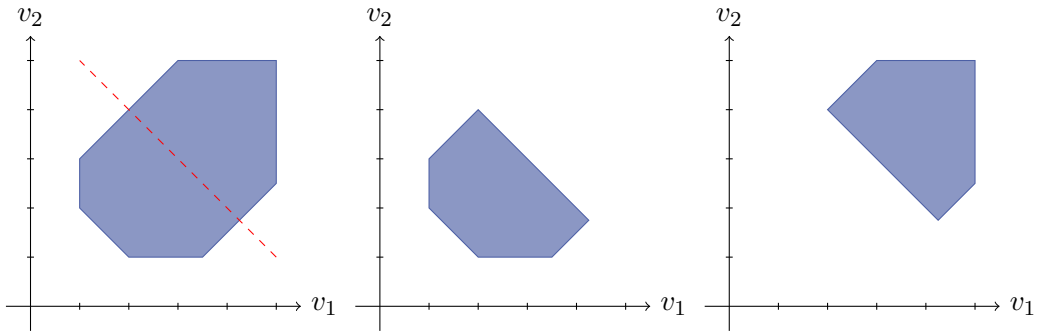


Figure 4.6 – Exemple de l'opérateur de coupe : l'octogone de gauche est coupé dans la base $B^{1,2}$.

4.3.1 Opérateur de coupe octogonal

L'opérateur de coupe octogonal défini ici étend l'opérateur de coupe usuel aux octogones. Les coupes peuvent être effectuées dans la base canonique, ce qui est équivalent aux coupes habituelles, ou dans une des bases tournées. Il est défini comme suit :

Définition 4.3.1 (Opérateur de coupe octogonal)**.** Soit $\mathbf{O}$ un octogone défini comme l'intersection de boîtes $I_1, \ldots, I_n, I_1^{1,2}, \ldots, I_1^{n-1,n}, \ldots, I_n^{n-1,n}$, tel que $I_k^{i,j} = [a, b]$ avec $a, b \in \mathbb{F}$. L'opérateur de coupe $\oplus_o(\mathbf{O})$ sur la variable $v_k^{i,j}$ calcule les deux sous-domaines octogonaux $I_1, \ldots, [a, h], \ldots, I_n^{n-1,n}$ et $I_1, \ldots, [h, b], \ldots, I_n^{n-1,n}$ avec $h = \frac{a+b}{2}$ arrondi au flottant.

Il est facilement vérifiable que l'union des deux sous-domaines octogonaux est égal à l'octogone de départ. En d'autres termes, cet opérateur de coupe ne perd pas de solutions et correspond à la définition 3.2.3. Notons aussi que cet opérateur a trois paramètres, i, j et k, et non plus un seul paramètre i comme pour l'opérateur de coupe sur les intervalles. Cependant, cette définition ne prend pas en compte les corrélations entre les variables des différentes bases. Afin de bénéficier des avantages de la représentation octogonale, la réduction d'un domaine est communiquée aux autres bases. La version modifiée de l'algorithme Floyd-Warshall est donc appelée après chaque coupe afin de mettre à jour les contraintes octogonales.

La figure 4.6 présente un exemple de l'opérateur de coupe octogonal. L'octogone de gauche est coupé le long de la ligne en pointillés rouges.

4.3.2 Précision octogonale

Dans la plupart des solveurs continus, la précision est définie comme le diamètre du plus grand domaine. Pour les octogones, cette définition entraîne une forte perte d'information car elle prend les domaines séparément et ne prend pas en compte les corrélations entre les variables. En d'autres termes, cette définition ne prend pas en compte le fait que les variables $v_1^{\{1,2\}}$ et $v_2^{\{1,2\}}$ dépendent des variables v_1 et v_2. Nous définissons donc une fonction de précision octogonale prenant plus en compte les corrélations entre les variables et correspondant aux diamètres sur tous les axes et diagonales.

Définition 4.3.2 (Précision octogonale)**.** Soit $\mathbf{O}$ un octogone et $I_1 \dots I_n, I_1^{1,2} \dots I_n^{n-1,n}$ les boîtes le représentant. La précision octogonale est

$$\tau_o(\mathbf{O}) = \min_{i,j \in [\![1,n]\!]} \left(\max_{k \in [\![1,n]\!]} \left(\overline{I_k^{i,j}} - \underline{I_k^{i,j}} \right) \right)$$

Pour une boîte, τ_o retourne la même précision que la fonction de précision usuelle pour les boîtes. Pour un octogone, on prend le minimum des précisions des boîtes de toutes les bases afin que la valeur retournée soit plus précise. De plus cette définition permet de retrouver la sémantique opérationnelle de la précision, comme montré dans la proposition suivante : dans un octogone de précision r sur-approximant l'ensemble des solutions S, tout point est à une distance au plus r de S.

Proposition 4.3.1. *Soit un problème de satisfaction de contraintes sur les variables $(v_1 \dots v_n)$, de domaines $(\hat{D}_1 \dots \hat{D}_n)$, avec les contraintes$(C_1 \dots C_p)$. Soit $\mathbf{O}$ un octogone sur-approximant l'ensemble des solutions S de ce problème et contenant au moins une solution. Soit $r = \tau_o(\mathbf{O})$. Soit $(x_1, \dots, x_n) \in \mathbb{F}^n$ un point de $I_1 \times \dots \times I_n, \forall i \in [\![1,n]\!], I_i \subseteq \hat{D}_i$. Alors $\forall i \in [\![1,n]\!]$, $\min_{s \in S} |v_i - s_i| \leq r$, où $s = (s_1 \dots s_n)$. Chaque coordonnée de tous points de $\mathbf{O}$ est à une distance projetée sur les axes au plus de r d'une solution.*

Démonstration.

Par définition 4.3.2, la précision r est le minimum d'un certaine quantité dans les bases tournées. Soit $B^{i,j}$ la base réalisant ce minimum. La boîte $\mathbf{B}_{\mathbf{O}}^{i,j} = I_1 \times \dots \times I_i^{i,j} \times \dots \times I_j^{i,j} \dots \times I_n$ est Hull-consistante et contient donc toutes les solutions S. Soit $s \in S$. Comme $r = max_k(\overline{I_k} - \underline{I_k}), \forall k \in [\![1,n]\!], |s_k - v_k| \leq \overline{I_k} - \underline{I_k} \leq r$. Tout point est donc bien à une distance au plus r d'une solution.

□

La figure 4.7 montre un exemple de précision octogonale. Considérons l'octogone obtenu par l'intersection de la boîte dans la base canonique (en bleu) et de celle dans la base tournée (en rose). La fonction de précision usuelle sur les boîtes retourne pour cet octogone la taille de I_1. La fonction de précision octogonale retourne, quant à elle, la taille de $I_1^{1,2}$, ce qui semble plus adéquat étant donné l'octogone considéré.

Afin de définir le domaine abstrait des octogones en programmation par contraintes, il ne reste plus qu'à prouver que l'ensemble des octogones forme un treillis complet et à définir une correspondance de Galois entre les octogones et les boîtes à bornes flottantes.

4.4 Domaines abstraits

Dans cette section, nous montrons dans un premier que l'ensemble des octogones forme un treillis complet pour l'inclusion.

Proposition 4.4.1. *L'ensemble des octogones forme un treillis complet $\mathbb{O}$.*

Démonstration.

Soient $\mathbf{O}$ et $\mathbf{O}$' deux octogones quelconques. L'ensemble $\{\mathbf{O}, \mathbf{O}'\}$ possède à la fois une *lub* $\mathbf{O} \cup \mathbf{O}'$ (remarque 4.1.4) et une *glb* $\mathbf{O} \cap \mathbf{O}'$ (remarque 4.1.5). On en déduit que tout ensemble fini possède une *lub* et une *glb* et donc que $\mathbb{O}$ forme un treillis complet pour l'inclusion.

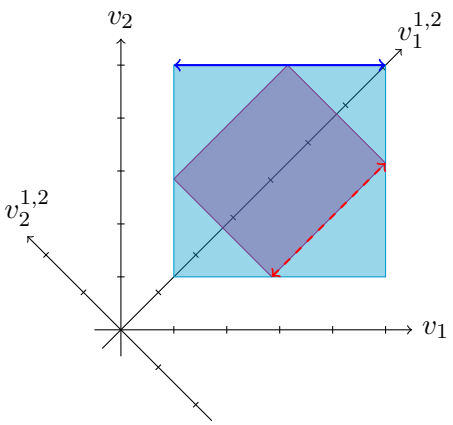


Figure 4.7 – Exemple de la précision octogonale : la précision usuelle sur les intervalles retourne la taille de I_1 (la flèche bleue) alors que la précision octogonale retourne la taille de $I_1^{1,2}$ (la flèche rouge en pointillés).

□

Montrons maintenant qu'il existe une correspondance de Galois entre les octogones et les boîtes à bornes flottantes.

Proposition 4.4.2. *Il existe une correspondance de Galois* $\mathbb{O} \underset{\alpha_o}{\overset{\gamma_o}{\leftrightarrows}} \mathbb{B}$.

Démonstration.

Soit
- $\alpha_o : \mathbb{O} \to \mathbb{B}, \mathbf{O} \mapsto \{\pm v_i \leq c, i \in [\![1, n]\!]\}$ avec $\mathbf{O} = \{\pm v_i \pm v_j \leq c\}$: seules les contraintes de la forme $\pm v_i \pm v_i \leq c$ sont conservées, ce qui correspond aux bornes de la boîte,
- $\gamma_o : \mathbb{B} \to \mathbb{O}, \mathbf{B} \mapsto \mathbf{B}$: est direct et exact car une boîte est un octogone.

La fonction d'abstraction α_o et la fonction de concrétisation γ_o forment une correspondance de Galois.

□

D'après les propositions 4.4.1 et 4.4.2, nous pouvons affirmer que les octogones peuvent être l'ensemble de base d'un domaine abstrait.

Définition 4.4.1 (Domaine abstrait des octogones)**.** Le domaine abstrait des octogones est défini par :
- le treillis complet $\mathbb{O}$;
- la correspondance de Galois $\mathbb{O} \underset{\alpha_o}{\overset{\gamma_o}{\leftrightarrows}} \mathbb{B}$;
- la représentation par intersection de boîtes ;
- l'opérateur de coupe octogonal $\oplus_o$;
- la fonction de précision τ_o.

Remarque 4.4.1 – Dans cette définition nous avons choisi la représentation par intersection de boîtes, mais elle peut être remplacée par la représentation matricielle étant donnée qu'elles sont équivalentes.

Cette définition des octogones repose sur la génération de toutes les bases tournées possibles, au nombre de $\frac{n(n+1)}{2}$. Cela transforme un CSP avec n variables en une représentation à n^2 variables. Parmi toutes ces bases certaines sont peut-être moins pertinentes que d'autres. Nous introduisons donc la notion d'octogone partiel.

Définition 4.4.2 (Octogone partiel)**.** Soit $\mathcal{J}, \mathcal{K} \in \mathcal{P}(\llbracket 1, n \rrbracket)$ des ensembles d'indices. Soit **O** l'ensemble de contraintes octogonales $\{\pm v_i \pm v_j \leq c \,|\, i \in \mathcal{J}, j \in \mathcal{K}\} \cup \{\pm v_i \pm v_i \leq c \,|\, i \in \llbracket 1, n \rrbracket\}$. L'ensemble des points de $\mathbb{R}^n$ satisfaisant toutes les contraintes de **O** est appelé octogone partiel.

Exemple 4.4.1 – Soit (v_1, v_2, v_3) un ensemble de variables. Soient $\mathcal{J} = \{1\}$ et $\mathcal{K} = \{2, 3\}$. L'octogone partiel associé correspond à l'ensemble des points satisfaisant l'ensemble des contraintes octogonales $\{\pm v_1 \pm v_2 \leq c, \pm v_1 \pm v_3 \leq c'\} \cup \{\pm v_i \pm v_i \leq c \,|\, i \in \llbracket 1, n \rrbracket\}$.

La définition des octogones partiels est très similaire à celle des octogones. Dans les deux cas la forme définie correspond à un ensemble de points satisfaisant un ensemble de contraintes octogonales. Cependant dans le cas des octogones partiels, seule une partie des contraintes octogonales non redondantes est exprimée. Notons que la définition des octogones est incluse dans cette définition. En effet, quand $\mathcal{J} = \mathcal{K} = \llbracket 1, n \rrbracket$ on retrouve la définition 4.1.2. Partant de ce constat, il est facilement démontrable que les propriétés sur les octogones restent valides pour les octogones partiels, et donc, que l'ensemble des octogones partiels est un ensemble de base adéquat pour un domaine abstrait.

Le choix des bases tournées à générer pour un octogone partiel dépend du problème à résoudre. Différentes heuristiques d'octagonalisation sont présentées section 5.3.2.

4.5 Conclusion

Dans ce chapitre nous avons donné deux formes équivalentes et représentables en machine pour les octogones. Nous avons montré que l'ensemble des octogones clos par l'algorithme modifié de Floyd-Warshall forme un treillis complet. De plus, nous avons défini l'opérateur de coupe et la précision octogonale ainsi que la correspondance de Galois entre les octogones et les boîtes à bornes flottantes. Grâce à tous ces éléments, nous avons pu définir les octogones comme un domaine abstrait en programmation par contraintes. Le domaine abstrait des octogones reposant sur la génération de toutes les bases possibles, nous avons défini l'ensemble des octogones partiels. Cet ensemble forme l'ensemble de base d'un domaine abstrait. Le chapitre suivant donne les détails supplémentaires quant à l'implémentation d'une méthode de résolution basée sur les octogones.

CHAPITRE 5

Résolution octogonale

Dans ce chapitre nous définissons une consistance basée sur les octogones ainsi qu'un schéma de propagation. Grâce à cette consistance octogonale et au domaine abstrait défini dans le chapitre précédent, nous obtenons une méthode de résolution basée sur les octogones. Un prototype de cette méthode a été implémenté dans un solveur continu. Les détails de cette implémentation sont donnés dans ce chapitre ainsi que des premiers résultats expérimentaux.

In this chapter we define a consistency based on octagons as well as a propagation schema. Thanks to this consistency and to the octagonal abstract domain defined in the previous chapter, we obtain a resolution method based on octagons. A prototype of this method has been implemented in a continuous solver. The details of this implementation are given in this chapter along with some preliminary results.

Considérons un problème de satisfaction de contraintes (CSP) sur les variables $(v_1 \dots v_n)$ dans $\mathbb{R}^n$. La première étape est de représenter ce problème sous forme octogonale. Nous détaillons ici comment construire un CSP octogonal à partir d'un CSP et montrons que ces deux systèmes sont équivalents.

5.1 CSP octogonal

Premièrement, le CSP est associé à un octogone en créant toutes les contraintes octogonales non redondantes possibles $\pm v_i \pm v_j \leq c$ pour $i, j \in [\![1, n]\!]$. Les constantes c représentent les bornes des boîtes dans les bases tournées, ou boîtes octogonales, et les bornes de la boîte dans la base canonique. Ces constantes sont modifiées dynamiquement tout au long du processus de résolution. Par défaut, elles sont initialisées à $+\infty$.

Les rotations définies dans le chapitre précédent (définition 4.2.3) introduisent de nouveaux axes, correspondant aux nouvelles variables $v_i^{i,j}$ et $v_j^{i,j}$ dans la (i, j)-base tournée. Ces variables sont redondantes avec les variables v_i et v_j de la base canonique, et si les variables v_i et v_j sont liées par une contrainte alors les nouvelles variables $v_i^{i,j}$ et $v_j^{i,j}$ le sont également. Les contraintes $C_1 \dots C_p$ du CSP doivent elles aussi être tournées. La suite de cette section explique comment tourner les contraintes.

Soit f une fonction sur les variables $(v_1 \dots v_n)$ dans la base canonique B, l'expression de f dans la (i, j)-base tournée est obtenue symboliquement en remplaçant les $i^{ème}$ et $j^{ème}$ coordonnées par leur expression dans $B_\alpha^{i,j}$. En d'autres termes, la variable v_i est remplacée par $\left(\cos(\alpha)v_i^{i,j} - \sin(\alpha)v_j^{i,j}\right)$ et la variable v_j par $\left(\sin(\alpha)v_i^{i,j} + \cos(\alpha)v_j^{i,j}\right)$ où $v_i^{i,j}$ et $v_j^{i,j}$ sont les coordonnées pour v_i et v_j dans $B_\alpha^{i,j}$. Les autres variables restent inchangées.

Définition 5.1.1 (Contrainte tournée)**.** Soit C une contrainte sur les variables $(v_1 \dots v_n)$. La (i, j)-contrainte tournée, notée $C^{i,j}$, est la contrainte obtenue en remplaçant chaque occurrence de v_i par $\left(\cos(\alpha)v_i^{i,j} - \sin(\alpha)v_j^{i,j}\right)$ et chaque occurrence de v_j par $\left(\sin(\alpha)v_i^{i,j} + \cos(\alpha)v_j^{i,j}\right)$.

Exemple 5.1.1 – Soient v_1 et v_2 deux variables, et soit C la contrainte $2v_1 + v_2 \leq 3$. La $(1, 2)$-contrainte tournée est :

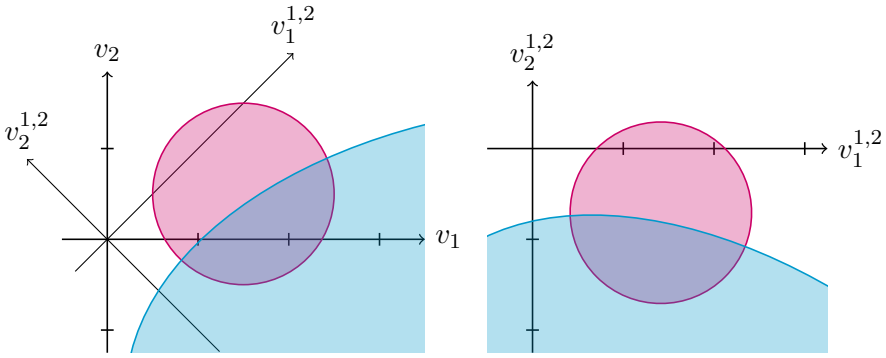
$$C^{1,2} \equiv 2\left(\cos\left(\frac{\pi}{4}\right)v_1^{1,2} - \sin\left(\frac{\pi}{4}\right)v_2^{1,2}\right) + \left(\sin\left(\frac{\pi}{4}\right)v_1^{1,2} + \cos\left(\frac{\pi}{4}\right)v_2^{1,2}\right) \leq 3$$

qui, étant donné que $\sin\left(\frac{\pi}{4}\right) = \cos\left(\frac{\pi}{4}\right) = \frac{1}{\sqrt{2}}$, peut être simplifiée :

$$C^{1,2} \equiv 3v_1^{1,2} - v_2^{1,2} \leq 3\sqrt{2}$$

La figure 5.1 compare graphiquement les contraintes initiales aux contraintes tournées. La figure 5.1(a) montre deux contraintes (rose et bleu) telles qu'elles sont données dans le CSP initial. L'équivalent de ces contraintes après une rotation de $\frac{\pi}{4}$ est illustré figure 5.1(b). Les contraintes tournées sont celles de la $(1, 2)$-base tournée $B^{1,2}$.

Soit un CSP continu sur les variables $(v_1 \dots v_n)$, de domaines continus $(\hat{D}_1 \dots \hat{D}_n)$ et de contraintes $(C_1 \dots C_p)$, nous définissons un CSP octogonal en ajoutant les variables tournées, les contraintes tournées et les domaines tournés stockés dans une matrice à différences bornées.



(a) Les contraintes dans la base canonique.

(b) Les contraintes dans $B^{1,2}$ la $(1,2)$-base tournée.

Figure 5.1 – Exemple de contraintes tournées : comparaison entre le CSP initial 5.1(a) et le CSP tourné 5.1(b).

Finalement, un CSP octogonal contient :
- les variables initiales $(v_1 \dots v_n)$;
- les variables tournées $(v_1^{1,2}, v_2^{1,2}, v_1^{1,3}, v_3^{1,3} \dots v_n^{n-1,n})$, où $v_i^{i,j}$ est la $i^{ème}$ variable dans la (i,j)-base tournée $B_\alpha^{i,j}$;
- les contraintes initiales $(C_1 \dots C_p)$;
- les contraintes tournées $(C_1^{1,2}, C_1^{1,3} \dots C_1^{n-1,n} \dots C_p^{n-1,n})$;
- les domaines initiaux $(\hat{D}_1 \dots \hat{D}_n)$;
- une matrice à différences bornées représentant les domaines tournés. Les éléments aux positions $(2i, 2i-1)$ et $(2i-1, 2i)$ pour $i \in [\![1, n]\!]$ sont initialisés aux bornes du domaine initial $\hat{D}_i$. Les autres éléments sont initialisés à $+\infty$.

Sous ces conditions, le CSP initial est équivalent au CSP octogonal restreint aux variables $v_1 \dots v_n$ comme montré dans la proposition suivante.

Proposition 5.1.1. *Soit un CSP sur les variables $(v_1 \dots v_n)$, de domaines $(\hat{D}_1 \dots \hat{D}_n)$ et de contraintes $(C_1 \dots C_p)$, et son CSP octogonal correspondant comme défini ci-dessus. L'ensemble des solutions du CSP d'origine S est égal à l'ensemble des solutions du CSP octogonal, restreint aux variables $(v_1 \dots v_n)$.*

Démonstration.

Solution octogonale ⇒ solution.

Soit $s \in \mathbb{R}^n$ une solution du CSP octogonal restreint aux variables $(v_1 \dots v_n)$. On a $C_1(s) \dots C_p(s)$ et $s \in \hat{D}_1 \times \dots \times \hat{D}_n$. Donc s est bien une solution du CSP d'origine. ◇

Solution ⇒ Solution octogonale.

Réciproquement, soit $s \in \mathbb{R}^n$ une solution du CSP d'origine. Les contraintes d'origine $(C_1 \dots C_p)$ sont vraies pour s. Montrons que s étendue aux variables tournées satisfait les contraintes tournées. Soit $i, j \in [\![1, n]\!]$, $i \neq j$ et $k \in [\![1, p]\!]$. Soit C_k une contrainte du CSP d'origine et $C_k^{i,j}$ la contrainte tournée correspondante. Par définition 5.1.1, $C_k^{i,j}(v_1 \dots v_{i-1}, \cos(\alpha)v_i^{i,j} - \sin(\alpha)v_j^{i,j}, v_{i+1} \dots \sin(\alpha)v_i^{i,j} + \cos(\alpha)v_j^{i,j} \dots v_n) \equiv$

$C_k(v_1 \dots v_n)$. Posons $s_i^{i,j} = \cos(\alpha)s_i + \sin(\alpha)s_j$ et $s_j^{i,j} = -\sin(\alpha)s_i + \cos(\alpha)s_j$ les images de s_i et s_j par rotation d'angle α. En inversant la rotation, $\cos(\alpha)s_i^{i,j} - \sin(\alpha)s_j^{i,j} = s_i$ et $\sin(\alpha)s_i^{i,j} + \cos(\alpha)s_j^{i,j} = s_j$, on a $C_k^{i,j}(s_1 \dots s_i^{i,j} \dots s_j^{i,j} \dots s_n) = C_k(s_1 \dots s_n)$ est vraie. Il reste à montrer que $(s_1 \dots s_i^{i,j} \dots s_j^{i,j} \dots s_n)$ appartient bien au domaine tourné, ce qui est vrai car la matrice à différences bornées est initialisée à $+\infty$. ◇

□

Pour un CSP sur n variables, cette représentation est beaucoup plus coûteuse avec un ordre de grandeur n^2. En effet, le CSP tourné a n^2 variables et domaines, et au plus $p\left(\frac{n(n-1)}{2} + 1\right)$ contraintes. Bien entendu, la plupart de ces objets sont redondants. Nous expliquons dans la prochaine section comment utiliser ces redondances afin d'accélérer le processus de résolution.

Lors de la définition des domaines abstraits en programmation par contraintes (section 3.2.3), nous avions précisé que les propagateurs devaient être définis de façon *ad hoc* étant donné qu'ils dépendent à la fois des contraintes et du domaine abstrait. Les prochaines sections montrent comment la consistance et le schéma de propagation peuvent être définis pour les octogones.

5.2 Consistance et propagation octogonale

Dans un premier temps nous généralisons la définition de la Hull-consistance (définition 2.2.9) aux domaines octogonaux et définissons des propagateurs pour les contraintes tournées. Puis, nous utiliserons la version modifiée de Floyd-Warshall (algorithme 4.1) afin de définir une boucle de propagation efficace à la fois pour les contraintes tournées et pour les contraintes octogonales.

5.2.1 Consistance octogonale

Nous généralisons la définition de la Hull-consistance sur les intervalles pour les contraintes continues aux octogones. Grâce à la représentation comme intersection de boîtes, nous montrons que tout propagateur pour la Hull-consistance sur les boîtes peut être étendu comme propagateur pour les octogones. Pour une relation n-aire sur $\mathbb{R}^n$ donnée, il existe un unique plus petit octogone (pour l'inclusion) contenant les solutions de cette relation, comme montré dans la proposition suivante.

Proposition 5.2.1. *Soit C une contrainte (resp. un système de contraintes $(C_1 \dots C_p)$), et S_C son ensemble de solutions (resp. S). Alors il existe un unique octogone $\mathbf{O}$ tel que : $S_C \subseteq \mathbf{O}$ (resp. $S \subseteq \mathbf{O}$), et pour tout octogone $\mathbf{O}'$, $S_C \subseteq \mathbf{O}'$ implique $\mathbf{O} \subseteq \mathbf{O}'$. $\mathbf{O}$ est l'unique plus petit octogone contenant les solution par rapport à l'inclusion. $\mathbf{O}$ est dit Oct-consistant pour la contrainte C (resp. pour le système de contraintes $(C_1 \dots C_p)$).*

Démonstration.

Les octogones sont clos par intersection (remarque 4.1.4), et l'ensemble des octogones à bornes flottantes forme un treillis fini et donc complet (proposition 4.4.1). La preuve est donc directement obtenue par la proposition 3.2.5.

□

Proposition 5.2.2. *Soit C une contrainte et $C^{i,j}$ la (i,j)-contrainte tournée pour $i,j \in [\![1,n]\!]$. Soit $\mathbf{B}^{i,j}$ la boîte Hull-consistante pour $C^{i,j}$, et $\mathbf{B}$ la boîte Hull-consistante pour C. L'octogone Oct-consistant pour C est obtenu par l'intersection de toutes les $\mathbf{B}^{i,j}$ et $\mathbf{B}$.*

Démonstration.

La représentation par intersection de boîtes étant une représentation exacte des octogones (proposition 4.2.1), la preuve vient directement de la proposition 5.2.1.

□

La consistance indique comment filtrer les valeurs inconsistantes pour une contrainte du CSP. Il faut maintenant choisir dans quel ordre propager les contraintes.

5.2.2 Schéma de propagation

La version modifiée de l'algorithme de plus court chemin de Floyd-Warshall (algorithme 4.1) propage efficacement et optimalement [Dechter *et al.*, 1989] les contraintes octogonales. Les contraintes initiales $(C_1 \dots C_p)$ et les contraintes tournées $(C_1^{1,2} \dots C_p^{n-1,n})$ sont quant à elles filtrées à l'aide de l'Oct-consistance. Il nous faut donc intégrer au sein d'un même schéma de propagation ces deux propagations. Les contraintes octogonales correspondant aux bornes des boîtes. Nous nous basons sur les informations recueillies lors de la propagation de celles-ci pour propager les contraintes initiales et tournées. Cela nous permet de bénéficier pleinement des propriétés relationnelles des octogones. Il est important de remarquer que tous les propagateurs définis précédemment sont monotones et complets, on peut donc les combiner dans n'importe quel ordre afin de calculer la consistance, comme montré proposition 3.2.5 ou dans [Benhamou, 1996].

L'idée principale de ce schéma de propagation est d'utiliser la version modifiée de l'algorithme de Floyd-Warshall et les propagateurs des contraintes initiales et tournées. Un pseudo-code est donné algorithme 5.1. Lors de la première itération dans la boucle de propagation, tous les propagateurs des contraintes initiales et tournées sont exécutés. Cette première phase permet de réduire les domaines en fonction des contraintes du problème. Les domaines étant stockés dans la matrice à différences bornées, les éléments de la matrice sont donc modifiés durant cette phase et un appel à la version modifiée de Floyd-Warshall est requise afin de calculer les bornes minimales. La seconde partie de la boucle de propagation correspond à l'algorithme 4.1 à ceci près que chaque fois qu'un élément est modifié, les propagateurs correspondants sont ajoutés à l'ensemble des propagateurs à exécuter. Les propagateurs contenus dans cet ensemble sont exécutés une fois la propagation des contraintes octogonales terminée. Ainsi si un élément est modifié plusieurs fois, il n'est propagé qu'une seule fois. On peut dire que ce schéma de propagation est guidé par les informations supplémentaires du domaine relationnel.

Un exemple d'une itération de cette propagation est donné figure 5.2 : les propagateurs des contraintes initiales ρ_{C_1}, ρ_{C_2} sont exécutés (figure 5.2(a)), suivi de ceux des contraintes tournées $\rho_{C_1^{1,2}}, \rho_{C_2^{1,2}}$ (figure 5.2(b)). Les deux boîtes ainsi obtenues sont rendues cohérentes l'une par rapport à l'autre, en exécutant la version modifiée de l'algorithme de Floyd-Warshall (figure 5.2(c)). Dans cet exemple, la boîte dans la base canonique est légèrement réduite, et donc les propagateurs ρ_{C_1}, ρ_{C_2} doivent être exécutés une nouvelle fois. L'application de ces propagateurs ne modifie pas la boîte dans la base canonique et l'octogone consistant correspond donc à l'intersection des deux boîtes (figure 5.2(d)).

```
float dbm[2n][2n]          /* Matrice à différences bornées contenant les domaines des variables */
ensemble de propagateurs propagSet   /* ensemble des propagateurs à exécuter */
int i, j, k, i', j'        /* indices de parcours de la matrice à différences bornées */
float m                    /* variable auxiliaire */

propagSet ← {ρ_{C_1}, ..., ρ_{C_p}, ρ_{C_1^{1,2}}, ..., ρ_{C_p^{n-1,n}}}
tant que propagSet ≠ ∅ faire
                           /* Propagation des contraintes initiales et tournées */
  exécuter tous les propagateurs de propagSet
  propagSet ← ∅
                           /* Propagation des contraintes octogonales */
  pour k de 1 à n faire
    pour i de 1 à 2n faire
      pour j de 1 à 2n faire
        m ← min( dbm[i][2k] + dbm[2k][j], dbm[i][2k − 1] + dbm[2k − 1][j],
                 dbm[i][2k − 1] + dbm[2k − 1][2k] + dbm[2k][j],
                 dbm[i][2k] + dbm[2k][2k − 1] + dbm[2k − 1][j] )
        si m < dbm[i][j] alors
          dbm[i][j] ← m                          /* mise à jour de la DBM */
          propagSet ← propagSet ∪{ρ_{C_1^{i,j}} ... ρ_{C_p^{i,j}}}  /* récupère les propagateurs à exécuter */
        fin si
      fin pour
    fin pour
    pour i de 1 à 2n faire
      pour j de 1 à 2n faire
        i' ← (i mod 2 = 0) ? i − 1 : i + 1
        j' ← (j mod 2 = 0) ? j − 1 : j + 1
        si (dbm[i][i'] + dbm[j'][j])) < dbm[i][j] alors
          dbm[i][j] ← dbm[i][i'] + dbm[j'][j]
          propagSet ← propagSet ∪{ρ_{C_1^{i,j}} ... ρ_{C_p^{i,j}}}
        fin si
      fin pour
    fin pour
  fin pour
  pour i de 1 à 2n faire
    si dbm[i][i] < 0 alors
      retourner erreur
    sinon
      dbm[i][i] ← 0
    fin si
  fin pour
fin tant que
```

Algorithme 5.1: Pseudo-code pour la boucle de propagation mélangeant la version modifiée de l'algorithme de Floyd-Warshall (propagation des contraintes octogonales) et les propagateurs originaux et tournés $\rho_{C_1} \dots \rho_{C_p}, \rho_{C_1^{1,2}} \dots \rho_{C_p^{n-1,n}}$, pour un CSP octogonal comme défini section 5.1.

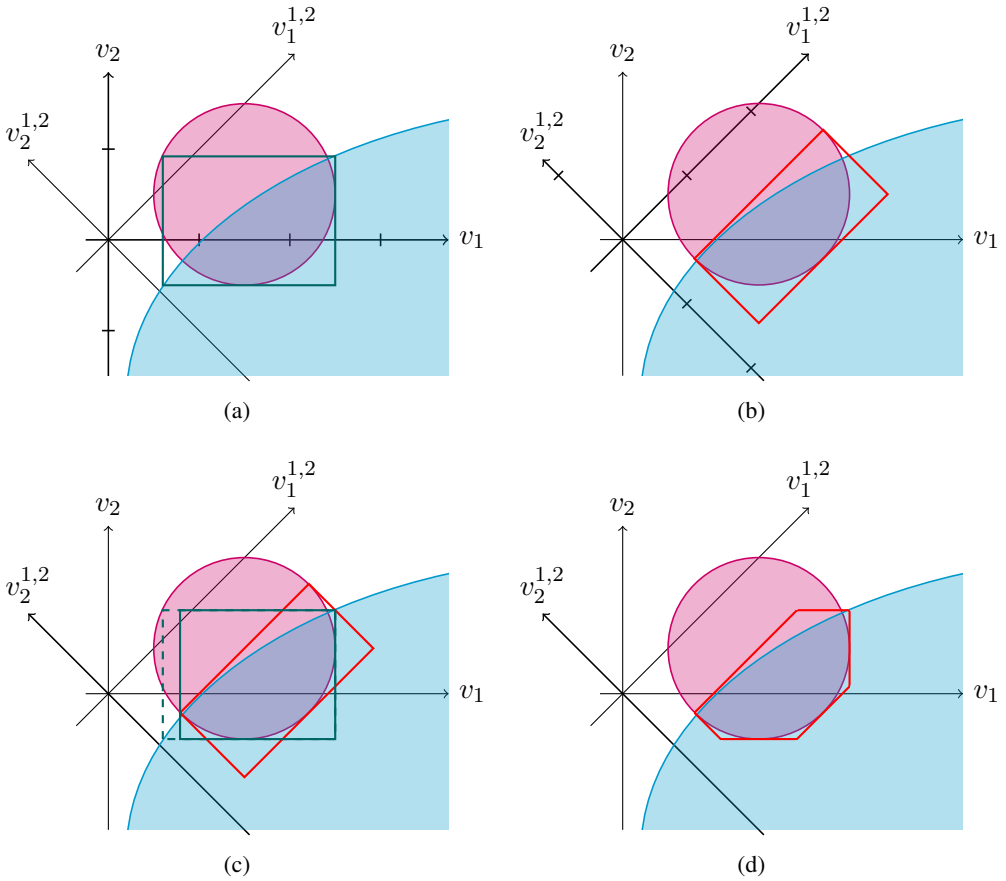


Figure 5.2 – Exemple d'une itération de la propagation pour la Oct-consistance : une consistance usuelle est appliquée dans chacune des bases (figures 5.2(a) et 5.2(b)) puis les boîtes sont rendues cohérentes à l'aide de la version modifiée de l'algorithme de Floyd-Warshall (figure 5.2(c)). L'octogone consistant est donné figure 5.2(d)

Montrons maintenant que la propagation donnée par l'algorithme 5.1 est correcte, c'est-à-dire, que cet algorithme retourne bien l'octogone consistant pour un système de contraintes.

Proposition 5.2.3 (Correction)**.** *Étant donné un CSP sur les variables $(v_1 \dots v_n)$, de domaines $(\hat{D}_1 \dots \hat{D}_n)$, et de contraintes $(C_1 \dots C_p)$. Supposons que pour toute contrainte C il existe un propagateur ρ_C tel que ρ_C atteigne la Hull-consistance, en d'autres termes $\rho_C(\hat{D}_1 \times \dots \times \hat{D}_n)$ retourne la boîte Hull-consistante pour C. Alors le schéma de propagation donné par l'algorithme 5.1 calcule l'octogone Oct-consistant pour $C_1 \dots C_p$.*

Démonstration.

Cette proposition découle de la proposition 5.2.2 et du schéma de propagation figure 5.1. Le schéma de propagation est défini de façon à terminer lorsque propagSet est vide, ce qui arrive quand la matrice à différences bornées n'est plus modifiée. En effet, les propagateurs ne sont ajoutés à propagSet que lorsqu'un des éléments de la matrice à différences bornées est modifié. Donc, lorsque ce schéma de propagation termine, les contraintes octogonales sont consistantes. De plus, les éléments de la matrice à différences bornées n'étant pas modifiés, on en déduit

que l'exécution des propagateurs n'a pas modifié les domaines des variables. Donc l'octogone final est stable par l'application de tous les $\rho_{C_k^{i,j}}$, pour tout $k \in [\![1, p]\!]$ et $i, j \in [\![1, n]\!]$. Étant donné que les propagateurs atteignent la consistance, les boîtes sont donc Hull-consistantes pour toutes les contraintes initiales et tournées. D'après la proposition 5.2.2, l'octogone en sortie de l'algorithme est donc bien Oct-consistant.

□

La version modifiée de l'algorithme de Floyd-Warshall a une complexité en temps de $O(n^3)$. À chaque exécution de cet algorithme, on ajoute dans le pire des cas tous les propagateurs à l'ensemble des propagateurs à exécuter, soit au pire $p\left(\frac{n(n-1)}{2} + 1\right)$ propagateurs. On a donc pour une itération du schéma de propagation donné par l'algorithme 5.1 une complexité en temps de $O(n^3 + pn^2)$. Au final, la propagation octogonale utilise les deux représentations pour les octogones. Cela permet d'avoir à la fois les avantages des propriétés relationnelles des contraintes octogonales (Floyd-Warshall) et des propagateurs de contraintes usuels sur les boîtes. Cela revient au même coût que celui de calculer l'octogone mais on s'attend à de meilleurs résultats en terme de précision.

Ayant désormais défini pour les octogones tous les composants nécessaires pour la résolution que sont la consistance octogonale, l'opérateur de coupe octogonal, la précision octogonale et le domaine abstrait des octogones, nous pouvons définir un solveur octogonal.

5.3 Solveur octogonal

Nous avons montré dans la section 4.4 que les octogones pouvaient former l'ensemble de base d'un domaine abstrait, et avec la consistance et le schéma de propagation comme décrits précédemment, nous obtenons un solveur octogonal. Le processus de résolution est le même que celui présenté dans l'algorithme 3.1 en utilisant les octogones. Les octogones étant clos par intersection (H1) et $\mathbb{O}$ n'ayant pas de chaine infinie décroissante (H2), cet algorithme termine et est correct. De plus, d'après l'algorithme 3.1, cet algorithme retourne un ensemble d'octogones dont l'union sur-approxime l'espace des solutions. Plus précisément, un octogone est considéré comme octogone solution si tous ses points sont des solutions, ou s'il sur-approxime un ensemble de solutions à une précision r.

Afin de guider l'exploration de l'espace de recherche, on peut définir une heuristique de choix de variable. Différentes heuristiques sont présentées dans la sous-section suivante.

5.3.1 Heuristiques de choix de variables

Une caractéristique importante d'un solveur est l'heuristique de choix de variable. Elle permet de choisir quelle variable couper. Pour les contraintes continues, on choisit généralement de couper en deux le plus grand des domaines des variables. En coupant en deux le plus grand domaine, cette heuristique permet d'arriver plus vite à la précision. Les trois heuristiques suivantes découlent de cette stratégie et diffèrent sur l'ensemble des variables qui peuvent être choisies. Soit V' l'ensemble de toutes les variables d'un CSP octogonal $V' = (v_1 \dots v_n, v_1^{1,2}, v_2^{1,2} \dots v_{n-1}^{n-1,n}, v_n^{n-1,n})$ que l'on re-numérote $V' = (v_1 \dots v_n, v_{n+1} \dots v_{n^2})$ pour des questions de simplicité.

Définition 5.3.1 (Heuristique LargestFirst (LF))**.** Soit $V' = (v_1 \dots v_n, v_{n+1} \dots v_{n^2})$ l'ensemble de toutes les variables du CSP octogonal. La variable choisie est la variable v_i réalisant le maximum de

$$\arg\max_{i \in [\![1,n^2]\!]} \left(\overline{D_i} - \underline{D_i}\right)$$

Comme pour l'heuristique de choix usuel en continu, la variable à couper est choisie parmi toutes les variables.

Définition 5.3.2 (Heuristique LargestCanFirst (LCF))**.** Soit $V' = (v_1 \dots v_n, v_{n+1} \dots v_{n^2})$ l'ensemble de toutes les variables du CSP octogonal. La variable choisie est la variable v_i qui réalise le maximum de

$$\arg\max_{i \in [\![1,n]\!]} \left(\overline{D_i} - \underline{D_i}\right)$$

Dans cette stratégie, le choix est restreint aux variables initiales, c'est-à-dire aux variables du CSP original.

Définition 5.3.3 (Heuristique LargestOctFirst (LOF))**.** Soit $V' = (v_1 \dots v_n, v_{n+1} \dots v_{n^2})$ l'ensemble de toutes les variables du CSP octogonal. La variable choisie est la variable v_i qui réalise le maximum de

$$\arg\max_{i \in [\![n+1,n^2]\!]} \left(\overline{D_i} - \underline{D_i}\right)$$

Dans cette stratégie, le choix est restreint aux variables tournées, c'est-à-dire aux variables générées par les rotations.

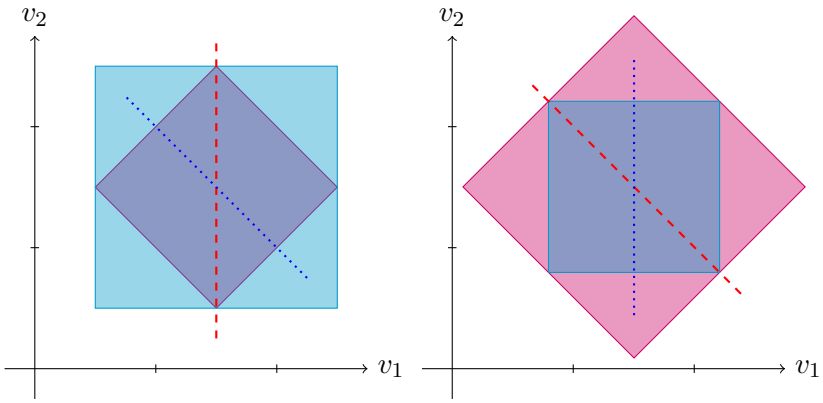
Les trois heuristiques de choix présentées précédemment ne prennent pas en compte les corrélations entres variables. De plus, la variable ayant le plus grand domaine peut être dans une base n'ayant pas grand intérêt pour le problème, ou il se peut que les domaines soient grands car les contraintes ne sont pas bien propagées dans cette base. Nous définissons donc une stratégie octogonale.

Définition 5.3.4 (Heuristique Oct-Split (OS))**.** Cette stratégie repose sur la même remarque que la définition 4.3.2 : la variable à couper est la variable $v_k^{i,j}$ réalisant le maximum de

$$\min_{i,j \in [\![1,n]\!]} \left(\max_{k \in [\![1,n]\!]} \left(\overline{D_k^{i,j}} - \underline{D_k^{i,j}} \right) \right)$$

La stratégie est la suivante : on choisit d'abord une base prometteuse, en d'autres termes, une base pour laquelle la boîte est étroite (choix de i, j). Puis, comme pour les stratégies usuelles, on prend la pire variable dans cette base (choix de k). Cette heuristique est conçue de façon à atteindre plus rapidement la précision octogonale (définition 4.3.2).

La figure 5.3 montre des exemples des différentes heuristiques de choix de variable. Dans ces deux exemples, on considère l'octogone obtenu par l'intersection de la boîte bleue avec la boîte rose. Dans le premier exemple, les stratégies LargestFirst et LargestCanFirst choisissent de couper le domaine D_1 en deux (le long de la ligne en tirets rouge). Quant aux stratégies LargestOctFirst et Oct-Split, elles choisissent de couper le domaine $D_1^{1,2}$ en deux (le long de la ligne bleue en pointillés). Dans le second exemple, les stratégies LargestFirst et LargestOctFirst choisissent de



(a) (b)

Figure 5.3 – Exemples des différentes heuristiques de choix de variable. Figure 5.3(a), les stratégies LF et LCF coupent le long de la ligne rouge en tirets alors que les stratégies LOF et OS coupent le long de la ligne bleue en pointillés. Figure 5.3(b), les stratégies LF et LOF coupent le long de la ligne rouge en tirets alors que les stratégies LCF et OS coupent le long de la ligne bleue en pointillés.

couper le domaine $D_1^{1,2}$ en deux (le long de la ligne en tirets rouge). Quant aux stratégies Largest-CanFirst et Oct-Split, elles choisissent de couper le domaine D_1 en deux (le long de la ligne bleue en pointillés).

Pour ces deux exemples, la coupe le long de la ligne bleue en pointillés semble plus pertinente. Dans le premier, couper le long de la ligne bleue en pointillés permet de réduire la taille des domaines dans la base tournée qui est pertinente pour cet octogone étant donné qu'il correspond à la boîte dans la base tournée. De même, dans le second exemple, réduire la taille des domaines dans la base canonique est plus pertinent car l'octogone correspond à la boîte dans la base canonique.

5.3.2 Heuristiques d'octogonalisation

Les heuristiques d'octogonalisation permettent de créer des octogones partiels. Pour les octogones toutes les $n(n+1)/2$ bases possibles sont générées, ce qui pour des problèmes de grande dimension (n grand) crée un octogone avec un grand nombre de faces. Comme dit précédemment, certaines des bases générées sont peut-être moins intéressantes par rapport au problème à résoudre. Nous définissons donc différentes stratégies afin de déterminer un sous-ensemble de bases à générer. Pour un problème donné, nous essayons de trouver, en regardant uniquement et de manière symbolique les contraintes, quelles sont les bases qui sont les plus susceptibles d'améliorer la résolution. En d'autres termes, nous essayons de déterminer dans quelles bases le problème est plus facile à résoudre, et ce en analysant uniquement les contraintes. Cette étape n'est effectuée qu'une seule fois au moment de la création de l'octogone ce qui diffère des travaux présentés dans [Goldsztejn et Granvilliers, 2010] où des changements de base sont effectués après chaque appel de l'opérateur de coupe dans la résolution.

Définition 5.3.5 (Heuristique ConstraintBased (CB))**.** Considérons un CSP sur les variables $(v_1 \dots v_n)$ à octogonaliser. La base $B_\alpha^{i,j}$ n'est générée que si les variables v_i et v_j apparaissent dans une même contrainte.

Si v_i et v_j apparaissent dans un contrainte C, alors C lie v_i avec v_j et nous dirons qu'il existe une relation entre v_i et v_j. L'idée derrière cette heuristique est de mettre l'accent sur une relation qui existe déjà entre deux variables. Dans le pire des cas, toutes les paires de variables apparaissent dans une contrainte et donc toutes les bases sont générées.

Définition 5.3.6 (Heuristique Random (R))**.** Considérons un CSP sur les variables $(v_1 \dots v_n)$ à octogonaliser. Parmi toutes les bases possibles, cette stratégie génère *une* base aléatoirement. Elle choisit aléatoirement i et j avec $i < j$ puis génère la base $B_\alpha^{i,j}$.

Nous avons fixé le nombre de bases générées à un, ceci est purement arbitraire et peut-être fixé aléatoirement afin d'avoir une heuristique totalement aléatoire.

Définition 5.3.7 (Heuristique StrongestLink (SL))**.** Considérons un CSP sur les variables $(v_1 \dots v_n)$ à octogonaliser. Cette heuristique génère une seule base, celle correspondant au couple de variables apparaissant dans le plus de contraintes. En d'autre termes, la base pour laquelle il existe un très fort lien entre les variables.

Exemple 5.3.1 – Soient les contraintes suivantes :

$$\begin{cases} v_1 + v_2 + v_1 \times v_2 & \leq & 3 \\ \cos(v_1) + v_3 & \leq & 10 \\ v_1 \times v_3 & \geq & 1 \end{cases}$$

La base $B_\alpha^{1,3}$ est générée car les variables v_1 et v_3 apparaissent ensemble dans 2 contraintes alors que la paire $\{v_1, v_2\}$ n'apparaît que dans une seule contrainte.

L'idée derrière l'heuristique StrongestLink est d'insister sur la plus forte relation entre deux variables. Nous voulons que les nouvelles variables soient dans le plus grand nombre de contraintes possibles afin que la résolution ait plus de chances de réduire leur domaine.

La dernière heuristique présentée repose sur la notion de schéma prometteur.

Définition 5.3.8 (Schéma prometteur)**.** Soit $(v_1 \dots v_n)$ un ensemble de variables. On appelle schéma prometteur les expressions correspondant aux patrons suivants : $\pm v_i \pm v_j$ ou $\pm v_i \times v_j$ pour $i, , j \in [\![1, n]\!]$.

Ces schémas sont dits prometteurs car ils peuvent être facilement simplifié dans la base tournée correspondante.

Exemple 5.3.2 – Soit v_1 et v_2 deux variables. Considérons l'expression $v_1 + v_2$, l'expression correspondante dans la $(1, 2)$-base tournée est :

$$\begin{aligned} v_1 + v_2 &= av_1^{1,2} - av_2^{1,2} + av_1^{1,2} + av_2^{1,2} \\ &= 2av_1^{1,2} \end{aligned}$$

où $a = \cos\left(\frac{\pi}{4}\right) = \sin\left(\frac{\pi}{4}\right)$. De même soit l'expression $v_1 \times v_2$. L'expression correspondante dans la $(1, 2)$-base tournée est :

$$\begin{aligned} v_1 \times v_2 &= \left(av_1^{1,2} - av_2^{1,2}\right) \times \left(av_1^{1,2} + av_2^{1,2}\right) \\ &= \left(av_1^{1,2}\right)^2 - \left(av_2^{1,2}\right)^2 \end{aligned}$$

Définition 5.3.9 (Heuristique Promising (P))**.** Considérons un CSP sur les variables $(v_1 \dots v_n)$ à octogonaliser. Cette heuristique génère la base $B_\alpha^{i,j}$ maximisant le nombre de schémas prometteurs dans les contraintes.

Exemple 5.3.3 – Soient les contraintes suivantes :

$$\begin{cases} v_1 + v_2 + v_1 \times v_2 & \leq & 3 \\ \cos(v_1) + v_3 & \leq & 10 \\ v_1 \times v_3 & \geq & 1 \end{cases}$$

La base $B_\alpha^{1,2}$ est générée car il existe deux schémas prometteurs avec les variables v_1 et v_2 ($v_1 + v_2$ et $v_1 \times v_2$) alors que pour la paire $\{v_1, v_3\}$ il n'y a qu'un seul schéma prometteur ($v_1 \times v_3$).

Cette heuristique a pour but de réduire le nombre de multiples occurrences des nouvelles variables dans les contraintes tournées. Nous savons que les implémentations de la Hull-consistance sont sensibles aux multiples occurrences des variables. Nous espérons donc avoir avec cette heuristique une consistance plus efficace ce qui devrait réduire le temps de résolution.

5.4 Résultats expérimentaux

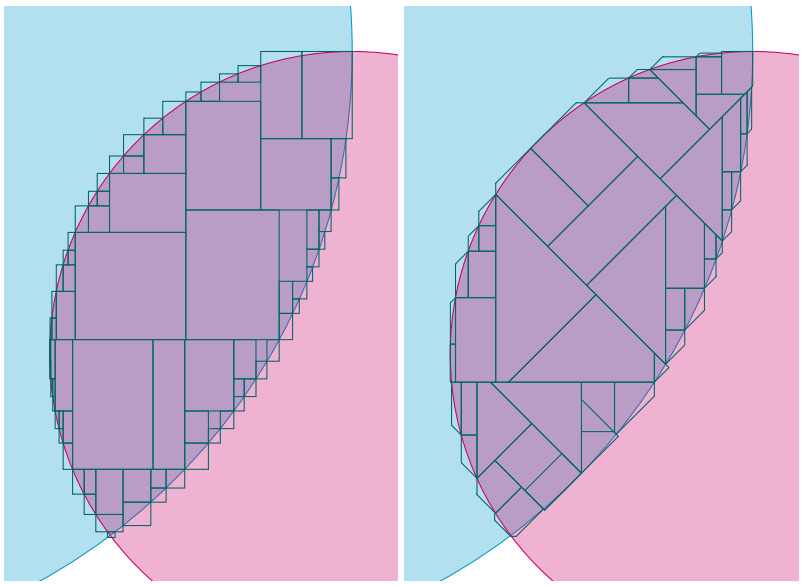
Cette section compare les résultats obtenus par le solveur octogonal à ceux obtenus avec un solveur continu par intervalles sur des benchmarks classiques.

5.4.1 Implémentation

Nous avons implémenté un prototype du solveur octogonal avec Ibex, une bibliothèque pour les contraintes continues [Chabert et Jaulin, 2009]. Nous avons utilisé l'implémentation d'Ibex de *HC4-Revise* [Benhamou *et al.*, 1999] pour propager les contraintes. Les octogones sont implémentés avec la représentation matricielle (DBM). Les variables et contraintes tournées additionnelles sont exprimées et prises en compte comme expliqué précédemment.

Un point important est à noter par rapport à la rotation des contraintes. L'algorithme HC4 est sensible aux multiples occurrences de variables, et la ré-écriture symbolique définie section 5.1 crée de multiples occurrences. Ainsi, la propagation de HC4 sur les contraintes tournées peut être très peu efficace si elle est effectuée directement sur les contraintes tournées. Il est nécessaire de réduire le nombres d'occurrences multiples dans les contraintes tournées. Pour ce faire, nous utilisons la fonction *Simplify* de Mathematica. Le temps de calcul indiqué ci-dessous ne comprend pas le temps de ce traitement, cependant, on observe qu'il est négligeable par rapport aux temps de résolution.

Le propagateur est une entrée de notre méthode : nous avons utilisé un propagateur classique (HC4). D'autres travaux comprendront l'ajout de propagateurs plus récents tel que [Araya *et al.*, 2010] qui prend en compte l'expression symbolique des contraintes afin d'améliorer leur propagation.



(a) (b)

Figure 5.4 – Comparaison des résultats obtenus avec la résolution continue usuelle 5.4(a) à ceux obtenus avec la résolution octogonale 5.4(b), pour un problème d'intersection de cercles dans la limite de 10ms.

5.4.2 Méthodologie

Nous avons essayé le prototype de solveur octogonal sur des problèmes du benchmark Coconut *. Ces problèmes ont été choisis par rapport au type des contraintes (inéquations, équations, ou les deux) et du nombre de variables.

Dans un premier temps nous avons comparé le temps nécessaire pour trouver la première solution ou toutes les solutions avec les intervalles et les octogones. Pour ces tests, nous avons fixé le paramètre de précision r à 0.01. Pour tous les autres tests, la précision r est fixée à 0.001. La deuxième expérience concerne l'étude des différentes heuristiques de choix de variables présentées section 5.3.1. La dernière série de tests compare les résultats obtenus par les octogones partiels générés par les différentes heuristiques d'octogonalisation (section 5.3.2) à ceux des octogones et des intervalles.

Toutes les expériences, ont été réalisées avec Ibex 1.18 sur un MacBook Pro Intel Core 2 Duo 2.53 GHz. Toutes ont été réalisées avec les mêmes configurations dans Ibex. En particulier, les mêmes propagateurs ont été utilisés de façon à comparer plus justement les résultats obtenus avec la résolution octogonale à ceux obtenus avec les intervalles. De plus, nous avons fixé une limite de temps à 3 heures dans tous les cas.

5.4.3 Résultats

La figure 5.4 compare les résultats obtenus par un solveur classique sur les intervalles (5.4(a)) à ceux obtenus avec la résolution octogonale (5.4(b)) dans une limite de temps de 10ms. On peut

∗. Ce benchmark peut être trouvé à l'adresse suivante http://www.mat.univie.ac.at/~neum/glopt/coconut/

voir qu'avec la résolution classique sur les intervalles, il y a un effet d'escalier dû aux boîtes aux bords de l'ensemble de solutions. Cet effet est moins présent avec la résolution octogonale, étant donné que les octogones permettent, par leur forme, d'être dans certains cas plus près de la courbe.

Les tableaux 5.1, 5.2, 5.3 comparent respectivement pour chaque problème sélectionné les résultats, en terme de temps CPU en secondes, le nombre de nœuds créés et le nombre de nœuds solutions, obtenus lors de la résolution avec les intervalles à ceux obtenus avec la résolution octogonale. Le tableau 5.4 compare pour chaque problème sélectionné, le temps nécessaire à sa résolution dépendant de l'heuristique de choix de variable choisie. Le tableau 5.5 compare pour chaque problème sélectionné, le temps en secondes nécessaire à sa résolution avec les intervalles à celui nécessité par les octogones et octogones partiels. Le tableau 5.6 compare pour chaque problème sélectionné le nombre de boîtes créées lors de la résolution intervalles au nombres d'octogones créés lors de la résolution octogonale. Enfin, le tableau 5.7 compare pour chaque problème sélectionné le nombre de boîtes solutions au nombre d'octogones solutions.

Dans tous les tableaux le symbole '-' correspond à un dépassement de la limite de temps (3 heures).

			Première solution		Toutes les solutions	
nom	# var	ctr type	$\mathbb{B}$	$\mathbb{O}$	$\mathbb{B}$	$\mathbb{O}$
h75	5	$\leq$	41.40	**0.03**	-	-
hs64	3	$\leq$	**0.01**	0.05	-	-
h84	5	$\leq$	5.47	**2.54**	-	**7238.74**
KinematicPair	2	$\leq$	**0.00**	**0.00**	53.09	**16.56**
pramanik	3	$=$	28.84	**0.16**	**193.14**	543.46
trigo1	10	$=$	18.93	**1.38**	**20.27**	28.84
brent-10	10	$=$	6.96	**0.54**	**17.72**	105.02
h74	4	$= \leq$	305.98	**13.70**	1304.23	**566.31**
fredtest	6	$= \leq$	3146.44	**19.33**	-	-

Table 5.1 – Résultats pour des problèmes du benchmark Coconut. Les trois premières colonnes donnent le nom, le nombre de variables et le type des contraintes du problème. Pour chaque problème, nous donnons le temps CPU en secondes pour trouver la première ou toutes les solutions avec les intervalles ($\mathbb{B}$) et les octogones ($\mathbb{O}$).

			Première solution		Toutes les solutions	
nom	# var	ctr type	$\mathbb{B}$	$\mathbb{O}$	$\mathbb{B}$	$\mathbb{O}$
h75	5	$\leq$	1 024 085	**149**	-	-
hs64	3	$\leq$	217	**67**	-	-
h84	5	$\leq$	87 061	**1 407**	-	**22 066 421**
KinematicPair	2	$\leq$	45	**23**	893 083	**79 125**
pramanik	3	$=$	321 497	**457**	2 112 801	**1 551 157**
trigo1	10	$=$	10 667	**397**	11 137	**5 643**
brent-10	10	$=$	115 949	**157**	238 777	**100 049**
h74	4	$= \leq$	8 069 309	**138 683**	20 061 357	**1 926 455**
fredtest	6	$= \leq$	29 206 815	**3 281**	-	-

Table 5.2 – Résultats pour des problèmes du benchmark Coconut. Les trois premières colonnes donnent le nom, le nombre de variables et le type des contraintes du problème. Pour chaque problème, nous donnons le nombre de nœuds créés pour trouver la première ou toutes les solutions avec les intervalles ($\mathbb{B}$) et les octogones ($\mathbb{O}$).

nom	# var	ctr type	$\mathbb{B}$	$\mathbb{O}$
h75	5	$\leq$	-	-
hs64	3	$\leq$	-	-
h84	5	$\leq$	-	**10 214 322**
KinematicPair	2	$\leq$	424 548	**39 555**
pramanik	3	$=$	**145 663**	210 371
trigo1	10	$=$	**12**	347
brent-10	10	$=$	854	**142**
h74	4	$= \leq$	700 669	**183 510**
fredtest	6	$= \leq$	-	-

Table 5.3 – Résultats pour des problèmes du benchmark Coconut. Les trois premières colonnes donnent le nom, le nombre de variables et le type des contraintes du problème. Pour chaque problème, nous donnons le nombre de nœuds solutions pour trouver toutes les solutions avec les intervalles ($\mathbb{B}$) et les octogones ($\mathbb{O}$).

5.4.4 Analyse

Nous analysons ici les résultats obtenus lors des différentes séries de tests.

Comparaison une solution vs. toutes les solutions

Le tableau 5.1 compare les résultats en terme de temps CPU nécessaire pour trouver la première ou toutes les solutions d'un problème en utilisant soit les intervalles, soit les octogones. Trouver la première solution est souvent plus rapide avec les octogones qu'avec les intervalles. En effet, les octogones étant plus précis, ils sont généralement plus près des solutions que les boîtes et, par conséquent, trouver la première solution est plus rapide. En revanche, trouver toutes les solutions peut s'avérer plus long avec les octogones qu'avec les boîtes. Un premier élément d'explication est que les contraintes des problèmes `brent-10`, `pramanik` et `trigo1` contiennent de nombreuses multiples occurrences de variables, ce qui augmente grandement le nombre de multiples occurrences des variables tournées, et ce malgré l'appel à la fonction *Simplify* de Mathematica.

nom	# var	ctr type	LF	LOF	LCF	OS
brent-10	10	$=$	-	-	362.77	**337.48**
o32	5	$\leq$	104.62	122.82	**40.60**	40.74
ipp	8	$=$	5 105.17	5 787.5	282.319	**279.36**
trigo1	10	$=$	-	-	256.71	**253.53**
KinematicPair	2	$\leq$	**59.72**	60.74	62.91	60.78
nbody5.1	6	$=$	105.93	121.33	27.33	**27.08**
pramanik	3	$=$	396.23	414.76	**240.93**	240.96
h74	4	$= \leq$	2896.58	4036.89	1553.67	**647.76**

Table 5.4 – Comparaison des différentes heuristiques de choix de variable sur de problèmes de benchmark Coconut. Les trois premières colonnes donnent le nom du problème, le nombre de variables et le type des contraintes. Pour chaque problème, le temps CPU en secondes nécessaire à la résolution en utilisant l'heuristique LargestFirst (LF), LargestOctFirst (LOF), LargestCanFirst (LCF) et Oct-Split (OS) est donné.

nom	# var	ctr type	$\mathbb{B}$	$\mathbb{O}$	CB	R	SL	P
brent-10	10	$=$	**21.58**	330.73	89.78	92.59	105.91	109.74
o32	5	$\leq$	27.25	40.74	40.74	**17.63**	20.68	21.23
ipp	8	$=$	38.83	279.36	279.36	30.14	**29.07**	41.60
trigo1	10	$=$	40.23	253.53	253.53	38.60	**37.03**	**37.03**
KinematicPair	2	$\leq$	**59.04**	60.78	60.78	60.78	60.78	60.78
nbody5.1	6	$=$	95.99	27.08	**22.13**	443.07	50.87	439.69
bellido	9	$=$	**111.12**	-	-	362.69	361.56	318.09
pramanik	3	$=$	281.80	240.96	240.96	141.94	137.42	**131.30**
caprasse	4	$=$	9175.36	-	-	**1085.21**	1131.33	2353.58
h74	4	$= \leq$	-	647.76	647.76	-	**0.15**	**0.15**

Table 5.5 – Résultats pour des problèmes du benchmark Coconut. Les trois premières colonnes donnent le nom, le nombre de variables et le type des contraintes du problème. Pour chaque problème, nous donnons le temps CPU en secondes pour le résoudre avec les intervalles ($\mathbb{B}$), les octogones ($\mathbb{O}$) et les différents octogones partiels, Constraint Based (CB), Random (R), Strongest Link (SL) et Promising (P).

nom	# var	ctr type	$\mathbb{B}$	$\mathbb{O}$	CB	R	SL	P
brent-10	10	$=$	211 885	**5 467**	5 841	175 771	205 485	21 495
o32	5	$\leq$	161 549	**25 319**	**25 319**	52 071	62 911	72 565
ipp	8	$=$	237 445	**21 963**	**21 963**	51 379	50 417	75 285
trigo1	10	$=$	13 621	**4 425**	**4 425**	6 393	5 943	5 943
KinematicPair	2	$\leq$	847 643	**373 449**	**373 449**	**373 449**	**373 449**	**373 449**
nbody5.1	6	$=$	598 521	5 435	**5 429**	578 289	137 047	542 263
bellido	9	$=$	774 333	-	-	577 367	573 481	**496 729**
pramanik	3	$=$	1 992 743	**243 951**	**243 951**	346 633	315 861	319 037
caprasse	4	$=$	150 519 891	-	-	**4 445 655**	4 472 839	9 933 597
h74	4	$= \leq$	-	418 867	418 867	-	**625**	**625**

Table 5.6 – Résultats pour des problèmes du benchmark Coconut. Les trois premières colonnes donnent le nom, le nombre de variables et le type des contraintes du problème. Pour chaque problème, nous donnons le nombre de boîtes créées lors de la résolution avec les intervalles ($\mathbb{B}$), ainsi que le nombre d'octogones créés lors de la résolution avec les octogones ($\mathbb{O}$) et les différents octogones partiels, Constraint Based (CB), Random (R), Strongest Link (SL) et Promising (P).

nom	# var	ctr type	$\mathbb{B}$	$\mathbb{O}$	CB	R	SL	P
brent-10	10	$=$	825	**149**	153	636	1 643	1 765
o32	5	$\leq$	74 264	**12 523**	**12 523**	35 868	31 216	25 833
ipp	8	$=$	2 301	**2243**	**2243**	4 329	6 922	6 194
trigo1	10	$=$	40	**32**	**32**	168	140	140
KinematicPair	2	$\leq$	346 590	**186 717**	**186 717**	**186 717**	**186 717**	**186 717**
nbody5.1	6	$=$	1 012	1 003	**996**	1 794	50 526	2 230
bellido	9	$=$	**7 372**	-	-	34 756	32 482	37 508
pramanik	3	$=$	149 011	**54 659**	**54 659**	72 052	69 621	71 239
caprasse	4	$=$	1 544	-	-	164	148	**110**
h74	4	$= \leq$	-	209 406	209 406	-	**293**	**293**

Table 5.7 – Résultats pour des problèmes du benchmark Coconut. Les trois premières colonnes donnent le nom, le nombre de variables et le type des contraintes du problème. Pour chaque problème, nous donnons le nombre de boîtes solutions retournées par la résolution avec les intervalles ($\mathbb{B}$), ainsi que le nombre d'octogones solutions retournés par la résolution avec les octogones ($\mathbb{O}$) et les différents octogones partiels, Constraint Based (CB), Random (R), Strongest Link (SL) et Promising (P).

En regardant maintenant le nombre de nœuds créés lors de la résolution (tableau 5.2), le nombre d'octogones créés est souvent bien inférieur au nombre de boîtes créées tout au long de la résolution. Cela vient du fait que les octogones étant plus précis, on passe plus de temps lors de la consistance et on a moins besoin de couper.

Quant au dernier tableau de résultats de cette comparaison, tableau 5.3, il compare le nombre de boîtes solutions au nombre d'octogones solutions. On voit que sur deux problèmes `pramanik` et `trigo1`, le nombre d'octogones solutions est plus grand que le nombre de boîtes solutions. Pour ces deux problèmes les octogones ont du mal à contracter efficacement les éléments aux bords de l'ensemble de solutions.

Comparaison des heuristiques de choix de variables

Le tableau 5.4 compare les résultats, en terme de temps CPU, obtenus par les différentes heuristiques de choix de variables. On peut voir que couper le plus grand domaine (LF) n'est pas la bonne stratégie. En effet, si les domaines ne sont pas bien propagés, cela signifie peut-être que la base dans laquelle ces domaines se trouvent n'a pas grand intérêt pour le problème à résoudre. Restreindre le choix aux variables des bases tournées (LOF) n'améliore pas les résultats et même les détériore. Cela montre qu'il existe des bases tournées dans lesquelles les domaines sont très grands et qu'elles n'apportent pas d'informations pertinentes. De plus, cela signifie qu'il est parfois nécessaire de couper dans la base canonique.

Quand le choix des variables est restreint aux variables de la base canonique (LCF), les résultats sont bien meilleurs et la limite de temps n'est dépassée pour aucun des problèmes sélectionnés. Cela peut s'expliquer par le fait que toutes les bases tournées dépendent de la base canonique, et donc chaque changement dans cette base est répercuté sur plusieurs autres bases et peut éventuellement améliorer l'approximation.

Finalement, la stratégie Oct-Split (OS) est le plus souvent la meilleure. En coupant dans une base de grand intérêt, cette heuristique permet d'explorer efficacement l'espace de recherche. De plus, elle est la plus adaptée à la définition de la fonction de précision, qui fait partie du critère de

terminaison. Pour ces raisons, nous avons utilisé cette heuristique pour la résolution octogonale dans les tests suivants.

Comparaison des heuristiques d'octogonalisation

En regardant les temps de résolution, tableau 5.5, les octogones et les octogones partiels obtiennent de très mauvais résultats sur les problèmes `brent-10` et `bellido`. Dans ces deux problèmes, les bases tournées contiennent très peu de contraintes tournées. Ce faible nombre de contraintes tournées n'aide pas à la réduction des variables tournées et donc à la réduction des variables initiales. En effet, les variables tournées étant liées aux variables initiales, si les unes sont réduites, les autres le seront aussi. Inversement si les variables tournées ne sont pas réduites, aucune information ne sera donnée aux variables initiales. En d'autres termes, le gain d'information apporté par ces contraintes et variables tournées ne compense pas le temps passé à les propager.

En revanche, pour les problèmes `h74`, `caprasse` et `nbody5.1` la résolution octogonale ou l'une utilisant des octogones partiels, améliore grandement les résultats. Nous nous attendions à améliorer le temps sur ces problèmes étant donnée la forme des contraintes. Le problème `h74` contient des contraintes de la forme $v_i - v_j \leq c$, pour un seul i et j donné, ce qui correspond à des contraintes octogonales et donc dans la (i, j)-base tournée ces contraintes sont très bien propagées et les solutions sont plus rapidement trouvées. Notons que, pour ce problème, lorsque toutes les bases sont générées, le temps de calcul est bien plus grand. Cela montre bien que la résolution passe beaucoup de temps à essayer de réduire les variables des autres bases sans que cela n'apporte d'informations. Le constat est le même pour le problème `caprasse`. Au contraire, dans le problème `nbody5.1`, toutes les variables peuvent être regroupées en paires suivant les schémas prometteurs (définition 5.3.8). Ainsi, générer toutes les bases ($\mathbb{O}$) ou juste un sous-ensemble correspondant aux couples de variables (CB) améliore grandement les résultats. De plus, on constate que l'une de ces paires est plus importante que les autres, car les résultats obtenus par l'heuristique StrongestLink (SL) ne sont pas trop détériorés. Nous pouvons même affirmer que cette paire ne correspond pas au couple ayant le plus de schémas prometteurs, étant donné que l'heuristique Promising (P) obtient de très mauvais résultats, mais plutôt à celle maximisant le nombre de contraintes tournées.

Pour le problème `KinematicPair`, le temps obtenu par la résolution octogonale est équivalent à celui obtenu avec les intervalles. Pour les autres problèmes, le temps obtenu par l'une des résolutions partiellement octogonales améliore légèrement le temps obtenu avec les intervalles.

Regardons toujours le temps de résolution mais comparons cette fois-ci les résultats obtenus pour chacune des heuristiques octogonalisation dans leur ensemble et non plus au cas par cas pour chacun des problèmes. Globalement, on peut voir que, contrairement à ce qui était attendu, c'est l'heuristique d'octogonalisation Strongest Link (SL) qui donne les meilleurs résultats. En effet, nous nous attendions à de meilleurs résultats avec l'heuristique Promising (P). On peut donc en déduire que plus le nombre de contraintes tournées est grand, meilleure est la propagation des variables tournées. Cela corrobore l'hypothèse donnée pour expliquer les mauvais résultats sur les problèmes `brent-10` et `bellido`.

Dans le tableau 5.6, nous pouvons voir que le nombre d'octogones créés durant la résolution est toujours plus petit que le nombre de boîtes créées. De plus, dans la majeure partie des problèmes, plus les octogones utilisés sont composés d'un grand nombre de bases, plus le nombre d'octogones

créés lors de la résolution est faible. On en déduit que plus la représentation est précise, plus on passera de temps pour la propagation mais moins de coupes seront nécessaires.

Le constat est le même pour le tableau 5.6, où le nombre d'octogones solutions est généralement plus petit quand les octogones sont composés d'un grand nombre de bases.

5.5 Conclusion

Dans ce chapitre nous avons implémenté la méthode de résolution unifiée présentée dans le chapitre 3 à l'aide du domaine abstrait des octogones (chapitre 4). Nous avons fourni un algorithme possible pour les différents composants de la résolution octogonale que sont la consistance et le schéma de propagation. De plus nous donnons différentes heuristiques pour le choix du domaine à couper, et de création d'octogones partiels. Les détails d'une implémentation de ce solveur basé sur Ibex sont donnés ainsi que des résultats préliminaires sur un benchmark classique pour les problèmes continus. Ces résultats sont encourageants et montrent que l'élaboration d'un opérateur de coupe et de stratégie d'exploration octogonale sont nécessaires afin de prendre en compte les relations des octogones et ainsi obtenir une meilleure résolution octogonale. Plusieurs perspectives sont possibles telles que l'utilisation de récents propagateurs moins sensibles aux multiples occurrences de variables comme Mohc [Araya *et al.*, 2010] ; la définition d'une heuristique d'octogonalisation qui offrirait un meilleur compromis entre StrongestLink (SL) et ConstraintBased (CB) ; ou encore la définition d'autres domaines abstraits en programmation par contraintes tels que les polyèdres [Cousot et Halbwachs, 1978] ou les intervalles polyédriques [Chen *et al.*, 2009].

CHAPITRE 6

Un solveur abstrait : AbSolute

Dans les chapitres précédents nous avons redéfini certaines notions d'interprétation abstraite en programmation par contraintes, ce qui nous a permis d'avoir un schéma de résolution uniforme indépendant du domaine abstrait choisi. Ces premiers travaux permettent en théorie l'élaboration d'une méthode de résolution pour les problèmes mixtes. Il reste un obstacle pratique : l'absence, dans les solveurs de programmation par contraintes, de représentations pouvant mêler à la fois des variables entières et réelles. Une solution repose sur le travail inverse : utiliser les domaines abstraits d'interprétation abstraite qui ne dépendent pas du type des variables et ajouter les techniques de résolution par contraintes au cadre de l'interprétation abstraite. Dans ce chapitre, nous présentons la programmation par contraintes en utilisant uniquement les notions d'interprétation abstraite. Puis nous présentons un solver abstrait, AbSolute, développé au-dessus d'une bibliothèque implémentant les domaines abstraits issus de l'interprétation abstraite.

In the previous chapters, we have defined some notions of Abstract Interpretation in Constraint Programming, allowing us to define a unified resolution scheme that does not depend on the chosen abstract domain. This scheme has been implemented giving us an octagonal solver. This early work makes it possible, in theory, to develop a mixed solver. However, there is a practical obstacle : the lack of representation for both integer and real variables, in solvers in Constraint Programming. A solution is to do the opposite : use Abstract Interpretation abstract domains that do not depend on the variables type and add the Constraint Programming resolution process in Abstract Interpretation. In this chapter, we present Constraint Programming using only notions from Abstract Interpretation. We then present an abstract solver, AbSolute, implemented on top of an abstract domain library.

En utilisant les liens mis en évidence dans la sous-section 2.3.1, nous exprimons la résolution par contraintes en interprétation abstraite, la résolution d'un CSP est vue ici comme une sémantique concrète. Nous définissons donc des domaines concrets et abstraits, des opérateurs abstraits pour les coupes et la consistance, et un schéma itératif calculant une approximation de sa sémantique concrète. Ces définitions nous permettent d'obtenir une méthode de résolution abstraite, que nous avons implémentée et testée sur différents problèmes. Tout comme les définitions énoncées précédemment, les détails de cette implémentation ainsi qu'une analyse des résultats obtenus sont donnés dans ce chapitre.

6.1 Une méthode de résolution abstraite

Comme pour l'analyse d'un programme en interprétation abstraite, nous définissons la sémantique concrète de la résolution d'un problème de satisfaction de contraintes. Puis, nous montrons que les représentations utilisées en programmation par contraintes correspondent à des domaines abstraits munis d'une fonction de précision. De plus, nous montrons que les opérateurs de clôture inférieurs en interprétation abstraite sont similaires aux propagateurs en programmation par contraintes, et de ce fait les itérations locales sont équivalentes à la boucle de propagation. Les opérateurs de coupe n'existant pas en interprétation abstraite, nous les définissons, et obtenons ainsi une méthode de résolution abstraite.

6.1.1 Résolution concrète comme analyse de la sémantique concrète

Précédemment, section 2.2.1, nous avons vu que résoudre un problème de satisfaction de contraintes revient à calculer l'ensemble des solutions d'une conjonction de contraintes. Étant données des valeurs pour les variables, les contraintes répondent vrai ou faux dans le cas de variables discrètes et vrai, faux ou peut-être dans le cas des variables continues. En partant de ce constat, la résolution d'un problème de satisfaction de contraintes peut être vue comme l'analyse d'une conjonction de tests, les contraintes. La sémantique concrète correspond donc à l'ensemble des solutions du problème. On considérera comme domaine concret $\mathcal{D}^\flat$ les sous-ensembles de l'espace de recherche initial $\hat{D} = \hat{D}_1 \times \cdots \times \hat{D}_n$ (définition 2.2.1), c'est-à-dire, $(\mathcal{P}(\hat{D}), \subseteq, \emptyset, \cup)$. De même, en programmation par contraintes, chaque contrainte C_i est associée à un propagateur, ce qui correspond en interprétation abstraite à un opérateur de clôture inférieur (définition 2.1.8) $\rho_i^\flat : \mathcal{P}(\hat{D}) \to \mathcal{P}(\hat{D})$, tel que $\rho_i^\flat(X)$ conserve uniquement les points de X satisfaisant la contrainte C_i. Finalement, les solutions concrètes du problème sont simplement $S = \rho^\flat(\hat{D})$, où $\rho^\flat = \rho_1^\flat \circ \cdots \circ \rho_p^\flat$.

De plus, en formalisant la résolution d'un problème de satisfaction de contraintes en termes d'itérations locales, les solutions peuvent être exprimées sous la forme d'un point fixe $\mathrm{gfp}_{\hat{D}}\, \rho^\flat$, c'est-à-dire, le plus grand point fixe de la composition des propagateurs,$\rho^\flat$, plus petit que l'espace de recherche initial, $\hat{D}$. Résoudre un problème revient à calculer un point fixe. Exprimer l'ensemble des solutions comme un plus grand point fixe a déjà été fait dans [Schulte et Stuckey, 2005].

L'espace des solutions étant identifié comme le domaine concret, les représentations existantes pour les domaines en programmation par contraintes peuvent être, quant à elles, vues comme des domaines abstraits.

6.1.2 Domaines abstraits existants en programmation par contraintes

Les solveurs ne manipulent pas des points dans $\hat{D}$ mais plutôt des collections de points d'une certaine forme, telles que des boîtes, appelées domaines en programmation par contraintes. Nous redéfinissons ici les représentations des domaines en programmation par contraintes comme l'ensemble de base $\mathcal{D}^\sharp$ d'un domaine abstrait $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \sqcup^\sharp)$ en interprétation abstraite.

Définition 6.1.1 (Produit cartésien d'entiers)**.** Soient $v_1 \dots v_n$ des variables de domaines discrets finis $\hat{D}_1 \dots \hat{D}_n$. Un produit cartésien d'ensembles d'entiers est appelé produit cartésien d'entiers et est exprimé dans $\mathcal{D}^\flat$ par :

$$\mathcal{S}^\sharp = \left\{ \prod_i X_i \,|\, \forall i, X_i \subseteq \hat{D}_i \right\}$$

Cette définition correspond à la définition 2.2.2.

Définition 6.1.2 (Boîte entière)**.** Soient $v_1 \dots v_n$ des variables de domaines discrets finis $\hat{D}_1 \dots \hat{D}_n$. Un produit cartésien d'intervalles entiers est appelé boîte entière et est exprimé dans $\mathcal{D}^\flat$ par :

$$\mathcal{I}^\sharp = \left\{ \prod_i [\![a_i, b_i]\!] \,|\, \forall i,\ [\![a_i, b_i]\!] \subseteq \hat{D}_i,\ a_i \leq b_i \right\} \cup \emptyset$$

Cette définition correspond à la définition 2.2.3.

Définition 6.1.3 (Boîte)**.** Soient $v_1 \dots v_n$ des variables de domaines continus $\hat{D}_1 \dots \hat{D}_n$. Un produit cartésien d'intervalles est appelé boîte et est exprimé dans $\mathcal{D}^\flat$ par :

$$\mathcal{B}^\sharp = \left\{ \prod_i I_i \,|\, I_i \in \mathbb{I}, I_i \subseteq \hat{D}_i \right\} \cup \emptyset$$

Cette définition correspond à la définition 2.2.4.

6.1.3 Opérateurs des domaines abstraits

En programmation par contraintes, à chaque représentation est associée une consistance. De la même façon, nous associons à chaque domaine abstrait une consistance et, en plus des opérateurs standards de l'interprétation abstraite, nous définissons une *fonction de précision* croissante $\tau : \mathcal{D}^\sharp \to \mathbb{R}^+$, utilisée comme le critère d'arrêt de l'opérateur de coupe (définition 6.1.7).

Exemple 6.1.1 – La consistance d'arc généralisée (définition 2.2.7) correspond au domaine abstrait des produits cartésiens d'entiers $\mathcal{S}^\sharp$ (définition 6.1.1), ordonné par l'inclusion élément à élément sur les ensembles. Elle est liée au domaine concret $\mathcal{D}^\flat$ par une correspondance de Galois standard :

$$\begin{aligned}
&\mathcal{D}^\flat \underset{\alpha_a}{\overset{\gamma_a}{\leftrightarrows}} \mathcal{S}^\sharp \\
&\gamma_a(S_1, \dots, S_n) = S_1 \times \dots \times S_n \\
&\alpha_a(X^\flat) = \lambda i.\{v \,|\, \exists (x_1, \dots, x_n) \in X^\flat,\ x_i = v\}
\end{aligned}$$

La fonction de précision τ_a utilise la taille du plus grand ensemble d'entiers :

$$\tau_a(S_1, \dots, S_n) = \max_i(|S_i|)$$

Ainsi, si l'élément considéré est une solution, toutes les variables sont instanciées, c'est-à-dire que tous les ensembles sont des singletons et τ_a vaut 1.

Exemple 6.1.2 – La consistance de bornes (définition 2.2.8) correspond au domaine des boîtes entières $\mathcal{I}^\sharp$ (définition 6.1.2), ordonné par l'inclusion. Nous avons une correspondance de Galois :

$$\begin{aligned}
&\mathcal{D}^\flat \underset{\alpha_b}{\overset{\gamma_b}{\leftrightarrows}} \mathcal{I}^\sharp\\
&\gamma_b([\![a_1,b_1]\!],\ldots,[\![a_n,b_n]\!]) = [\![a_1,b_1]\!]\times\cdots\times[\![a_n,b_n]\!]\\
&\alpha_b(X^\flat) = \lambda i.[\![\ \min\{v\in\mathbb{Z}\,|\,\exists(x_1,\ldots,x_n)\in X^\flat,\ x_i = v\},\\
&\qquad\qquad \max\{v\in\mathbb{Z}\,|\,\exists(x_1,\ldots,x_n)\in X^\flat,\ x_i = v\}]\!]
\end{aligned}$$

Nous utilisons comme fonction de précision la taille de la plus grande dimension sur les axes plus un afin que, comme pour les produits cartésiens d'entiers, si l'élément est une solution, la fonction de précision vaut 1 :

$$\tau_b([\![a_1,b_1]\!],\ldots,[\![a_n,b_n]\!]) = \max_i(b_i - a_i) + 1$$

Exemple 6.1.3 – La consistance d'enveloppe (définition 2.2.9) correspond au domaine des boîtes à bornes flottantes $\mathcal{B}^\sharp$ (définition 6.1.3). Nous utilisons la correspondance de Galois suivante :

$$\begin{aligned}
&\mathcal{D}^\flat \underset{\alpha_h}{\overset{\gamma_h}{\leftrightarrows}} \mathcal{B}^\sharp\\
&\gamma_h([a_1,b_1],\ldots,[a_n,b_n]) = [a_1,b_1]\times\cdots\times[a_n,b_n]\\
&\alpha_h(X^\flat) = \lambda i.[\ \max\{v\in\mathbb{F}\,|\,\forall(x_1,\ldots,x_n)\in X^\flat,\ x_i \geq v\},\\
&\qquad\qquad \min\{v\in\mathbb{F}\,|\,\forall(x_1,\ldots,x_n)\in X^\flat,\ x_i \leq v\}]
\end{aligned}$$

La fonction de précision correspond à la taille de la plus grande dimension sur les axes :

$$\tau_h([a_1,b_1],\ldots,[a_n,b_n]) = \max_i(b_i - a_i)$$

Notons qu'à chacun des choix correspond un domaine abstrait non-relationnel classique en interprétation abstraite. De plus, il correspond à un produit cartésien homogène d'un seul et même ensemble de base (type de représentation pour une variable). Cependant, ceci n'est pas obligatoire, de nouveaux solveurs peuvent être conçus allant plus loin que ceux traditionnels en programmation par contraintes en modifiant davantage les domaines abstraits. Une première idée est d'appliquer différentes consistances à différentes variables, ce qui permettrait en particulier de mélanger des variables de domaines discrets avec des variables de domaines continus. Cependant, les domaines correspondent toujours à des produits cartésiens. Une seconde idée est de paramètrer le solveur avec d'autres domaines abstraits existant en interprétation abstraite, en particulier les domaines abstraits relationnels ou mixtes. Nous choisissons la seconde idée, ainsi les représentations ne sont plus restreintes aux produits cartésiens et peuvent représenter différents types de variables. Cette idée est illustrée ci-dessous.

Exemple 6.1.4 – Le domaine abstrait des octogones $\mathcal{O}^\sharp$ [Miné, 2006] assigne une borne (flottante) supérieure à chaque expression binaire unitaire $\pm v_i \pm v_j$ sur les variables $v_1,\ldots,v_n$.

$$\mathcal{O}^\sharp = \{\alpha v_i + \beta v_j \,|\, i,j\in[\![1,n]\!],\ \alpha,\beta\in\{-1,1\}\,\} \to \mathbb{F}$$

Il admet une correspondance de Galois avec $\mathcal{D}^\flat$. Nous rappelons cette correspondance et la fonction de précision définie précédemment (définition 4.3.2). $\forall X^\sharp \in \mathcal{O}^\sharp, \forall i,j,\in [\![1,n]\!], \forall\alpha,\beta\in$

$\{-1, 1\}$, on note $X^\sharp(\alpha v_i + \beta v_j)$ la borne flottante supérieure de l'expression binaire $\alpha v_i + \beta v_j$.

$$\begin{array}{l}\mathcal{D}^\flat \underset{\alpha_o}{\overset{\gamma_o}{\leftrightarrows}} \mathcal{O}^\sharp \\ \gamma_o(X^\sharp) = \{(x_1, \ldots, x_n) \,|\, \forall i, j, \alpha, \beta,\ \alpha x_i + \beta x_j \leq X^\sharp(\alpha v_i + \beta v_j)\} \\ \alpha_o(X^\flat) = \lambda(\alpha v_i + \beta v_j).\, \max\{\alpha x_i + \beta x_j \,|\, (x_1, \ldots, x_n) \in X^\flat\} \\ \tau_o(X^\sharp) = \min(\ \max\limits_{i,j,\beta}(X^\sharp(v_i + \beta v_j) + X^\sharp(-v_i - \beta v_j)), \\ \qquad\qquad \max\limits_{i}(X^\sharp(v_i + v_i) + X^\sharp(-v_i - v_i))/2 \quad)\end{array}$$

Exemple 6.1.5 – Le domaine abstraits des polyèdres $\mathcal{P}^\sharp$ [Cousot et Halbwachs, 1978] abstrait des ensembles convexes délimités par des polyèdres clos. Des implémentations modernes [Jeannet et Miné, 2009] utilisent généralement une approche de double description et maintiennent deux représentations pour chaque polyèdre : un ensemble de contraintes linéaires et un ensemble de générateurs. Un générateur est soit un sommet du polyèdre, soit un rayon. Un rayon correspond à un vecteur suivant lequel, en partant d'un point du polyèdre, tout point fait partie du polyèdre. Cependant, les polyèdres que nous utilisons n'ont pas de rayons étant donné qu'ils sont bornés. Il n'existe pas de fonction d'abstraction α pour les polyèdres, et donc il n'existe pas de correspondance de Galois. Les opérateurs sont généralement plus faciles à calculer dans une représentation que dans l'autre. En particulier, nous définissons la fonction de précision comme la distance euclidienne maximale entre deux sommets. Soit $X^\sharp \in \mathcal{P}^\sharp$,

$$\tau_p(X^\sharp) = \max_{g_i, g_j \in X^\sharp} ||g_i - g_j||$$

Exemple 6.1.6 – Le domaine abstrait des boîtes mixtes $\mathcal{M}^\sharp$, ordonné par l'inclusion. Soit $v_1 \ldots v_n$, l'ensemble des variables et soit $(v_1, \ldots, v_m), m \in [\![1, n]\!]$ l'ensemble des variables entières et $(v_{m+1}, \ldots, v_n)$ l'ensemble des variables réelles. Le domaine abstrait des boîtes mixtes assigne aux variables entières un intervalle entier et aux variables réelles un intervalle réel à bornes flottantes.

$$\mathcal{M}^\sharp = \left\{ \prod_{i=1}^{m} [\![a_i, b_i]\!] \,|\, \forall i,\ [\![a_i, b_i]\!] \subseteq \hat{D}_i,\ a_i \leq b_i \right\} \times \left\{ \prod_{i=m+1}^{n} I_i \,|\, I_i \in \mathbb{I}, I_i \subseteq \hat{D}_i \right\} \cup \emptyset$$

Ce domaine abstrait admet une correspondance de Galois avec le domaine concret $\mathcal{D}^\flat$:

$$\begin{array}{l}\mathcal{D}^\flat \underset{\alpha_m}{\overset{\gamma_m}{\leftrightarrows}} \mathcal{M}^\sharp \\ \gamma_m([\![a_1, b_1]\!], \ldots, [\![a_m, b_m]\!], [a_{m+1}, b_{m+1}], \ldots, [a_n, b_n]) \\ \qquad = [\![a_1, b_1]\!] \times \cdots \times [\![a_m, b_m]\!] \times [a_{m+1}, b_{m+1}] \times \cdots \times [a_n, b_n] \\ \alpha_m(X^\flat) = \lambda i. \begin{cases} [\![\ \min\{v \in \mathbb{Z} \,|\, \exists (x_1, \ldots, x_n) \in X^\flat,\ x_i = v\}, \\ \quad \max\{v \in \mathbb{Z} \,|\, \exists (x_1, \ldots, x_n) \in X^\flat,\ x_i = v\}\ ]\!] & i \in [\![1, m]\!] \\ [\ \max\{v \in \mathbb{F} \,|\, \forall (x_1, \ldots, x_n) \in X^\flat,\ x_i \geq v\}, \\ \quad \min\{v \in \mathbb{F} \,|\, \forall (x_1, \ldots, x_n) \in X^\flat,\ x_i \leq v\}\ ] & i \in [\![m+1, n]\!] \end{cases}\end{array}$$

Il utilise la fonction de précision de la plus grande dimension sur les axes :

$$\tau_m([\![a_1, b_1]\!], \ldots, [\![a_m, b_m]\!], [a_{m+1}, b_{m+1}], \ldots, [a_n, b_n]) = \max_i (b_i - a_i)$$

6.1.4 Contraintes et consistance

À partir de maintenant, nous supposons qu'un domaine abstrait $\mathcal{D}^\sharp$ sous-jacent au solveur est fixé.

Pour une sémantique concrète de contraintes $\rho^\flat = \rho_1^\flat \circ \cdots \circ \rho_p^\flat$ donnée, et si $\mathcal{D}^\sharp$ admet une correspondance de Galois $\mathcal{D}^\flat \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$, alors la sémantique du propagateur parfait réalisant la consistance pour toutes les contraintes est simplement : $\alpha \circ \rho^\flat \circ \gamma$. Les solveurs réalisent ceci de façon algorithmique. Ils appliquent le propagateur de chaque contrainte jusqu'à atteindre un point fixe ou, quand ce processus est estimé trop coûteux, s'arrêtent avant que le point fixe ne soit atteint.

Remarque 6.1.1 – En observant que chaque propagateur correspond à une fonction abstraite de transfert de test $\rho_i^\sharp$ dans $\mathcal{D}^\sharp$, nous retrouvons les itérations locales de Granger pour analyser des conjonctions de tests [Granger, 1992]. Un opérateur de rétrécissement naïf est utilisé ici : arrêter après un certain nombre d'itérations.

De plus, chaque $\rho_i^\sharp$ peut être implémenté en interne par les itérations locales [Granger, 1992], une technique qui est utilisée à la fois en interprétation abstraite et en programmation par contraintes. Il existe une autre ressemblance frappante entre l'algorithme utilisé lors de l'analyse dans les domaines non-relationnels utilisant des itérations d'aller-retour sur les arbres syntaxiques d'expressions [Miné, 2004, §2.4.4], et l'algorithme HC4-Revise [Benhamou *et al.*, 1999] développé indépendamment en programmation par contraintes.

On peut donc voir la consistance comme une abstraction de la sémantique concrète des contraintes.

6.1.5 Complétion disjonctive et coupe

Afin d'approximer les solutions à une précision arbitraire, les solveurs utilisent un pavage fini d'éléments abstraits de $\mathcal{D}^\sharp$. En interprétation abstraite, un pavage fini correspond à une complétion disjonctive [Cousot et Cousot, 1992a].

Définition 6.1.4 (Complétion disjonctive)**.** Soit $\mathcal{D}^\sharp$ un ensemble. Une complétion disjonctive $\mathcal{E}^\sharp = \mathcal{P}_{\text{finite}}(\mathcal{D}^\sharp)$ est un sous-ensemble de $\mathcal{D}^\sharp$ dont les éléments ne sont pas comparables. C'est-à-dire :

$$\mathcal{E}^\sharp = \{X^\sharp \subseteq \mathcal{D}^\sharp \mid \forall B^\sharp, C^\sharp \in X^\sharp, B^\sharp \not\sqsubseteq^\sharp C^\sharp\}$$

Exemple 6.1.7 – Tous les éléments d'un même niveau dans un treillis forment une complétion disjonctive car ils ne sont pas comparables les uns aux autres. Considérons le treillis des boîtes entières $\mathcal{I}^\sharp$ muni de l'inclusion, l'ensemble

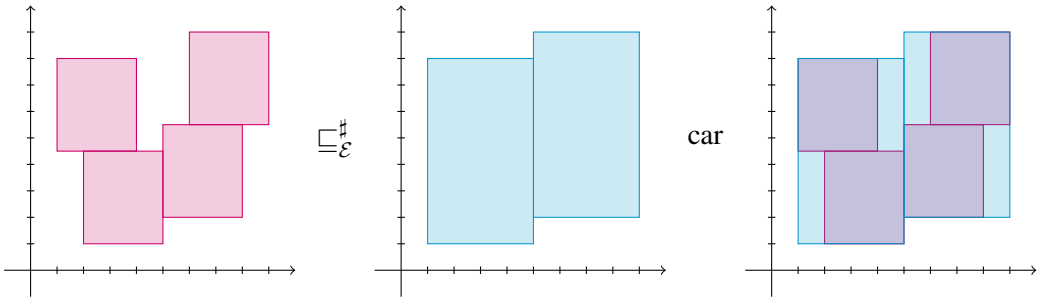
$$\{[\![1,2]\!] \times [\![1,2]\!], [\![1,2]\!] \times [\![3,7]\!]\}$$

est une complétion disjonctive, car la boîte entière $[\![1,2]\!] \times [\![1,2]\!]$ n'est pas comparable à la boîte entière $[\![1,2]\!] \times [\![3,7]\!]$ par l'inclusion. En effet, la première boîte n'est pas incluse dans la seconde et inversement.

Nous considérons maintenant le domaine abstrait $\mathcal{E}^\sharp = \mathcal{P}_{\text{finite}}(\mathcal{D}^\sharp)$, et l'équipons d'un ordre de Smyth $\sqsubseteq^\sharp_\mathcal{E}$, tel que $\forall X^\sharp, Y^\sharp \in \mathcal{E}^\sharp$ deux complétions disjonctives, $X^\sharp$ est incluse dans $Y^\sharp$ si et seulement si chacun des éléments de $X^\sharp$ est inclus dans l'un des éléments de $Y^\sharp$. C'est un ordre classique pour les complétions disjonctives et il est défini ainsi :

$$X^\sharp \sqsubseteq^\sharp_\mathcal{E} Y^\sharp \Longleftrightarrow \forall B^\sharp \in X^\sharp,\ \exists C^\sharp \in Y^\sharp,\ B^\sharp \sqsubseteq^\sharp C^\sharp$$

Exemple 6.1.8 – Considérons le treillis des boîtes $\mathcal{B}^\sharp$ muni de l'inclusion. Soit $X^\sharp = \{[2,5] \times [1,4.5], [1,4] \times [4.5,8], [5,8] \times [2,5.5], [6,9] \times [5.5,9]\}$ et $Y^\sharp = \{[1,5] \times [1,8], [5,9] \times [2,9]\}$ deux complétions disjonctives. On a $X^\sharp \sqsubseteq^\sharp_\mathcal{E} Y^\sharp$, car chacune des boîtes de $X^\sharp$ est incluse dans une des boîtes de $Y^\sharp$. Graphiquement on a la figure suivante, les boîtes de gauche (en rose) sont incluses dans une des boîtes de droite (en bleu).



L'opérateur de coupe $\oplus$, coupe un élément de $\mathcal{D}^\sharp$ en deux ou plusieurs éléments. Il transforme donc un élément de $\mathcal{D}^\sharp$ en un élément de $\mathcal{E}^\sharp$, et permet donc de créer de nouvelles complétions disjonctives. L'opérateur de coupe n'existant pas en interprétation abstraite, nous le redéfinissons.

Définition 6.1.5 (Opérateur de coupe)**.** Un *opérateur de coupe* est un opérateur $\oplus : \mathcal{D}^\sharp \to \mathcal{E}^\sharp$ tel que $\forall e \in \mathcal{D}^\sharp$,

1. $|\oplus(e)|$ est fini,
2. $\forall e_i \in \oplus(e),\ e_i \sqsubseteq^\sharp e$, et
3. $\gamma(e) = \bigcup \{\gamma(e_i) \,|\, e_i \in \oplus(e)\}$.

Cette définition est la version pour les domaines abstraits de la définition 3.2.3.

Chaque élément de $\oplus(e)$ étant inclus dans e (condition 2), on a $\oplus(e) \sqsubseteq^\sharp_\mathcal{E} \{e\}$. De plus, la condition 3 montre que $\oplus$ est une abstraction de l'identité. Et donc, $\oplus$ peut être utilisé à n'importe quel moment de la résolution sans en altérer la correction. L'instanciation des entiers et la bissection des boîtes peuvent bien entendu être retrouvées.

Exemple 6.1.9 – L'instanciation d'une variable v_i de domaine discret $X^\sharp = (S_1, \ldots, S_n) \in \mathcal{S}^\sharp$ est un opérateur de coupe :

$$\oplus_a(X^\sharp) = \{(S_1, \ldots, S_{i-1}, x, S_{i+1}, \ldots, S_n) \,|\, x \in S_i\}$$

Exemple 6.1.10 – L'instanciation d'une variable v_i de domaine discret $X^\sharp = (X_1, \ldots, X_n) \in \mathcal{I}^\sharp$ est un opérateur de coupe :

$$\oplus_b(X^\sharp) = \{(X_1 \times \cdots \times X_{i-1} \times x \times X_{i+1} \times \cdots \times X_n) \,|\, x \in X_i\}$$

Exemple 6.1.11 – Couper une boîte en deux le long d'une variable v_i de domaine continu $X^\sharp = (I_1, \ldots, I_n) \in \mathcal{B}^\sharp$ est un opérateur de coupe :

$$\oplus_h(X^\sharp) = \{(I_1 \times \cdots \times I_{i-1} \times [a,h] \times I_{i+1} \times \cdots \times I_n),\ (I_1 \times \cdots \times I_{i-1} \times [h,b] \times I_{i+1} \times \cdots \times I_n)\}$$

où $I_i = [a,b]$ et $h = \overline{(a+b)/2}$ ou $h = \underline{(a+b)/2}$.

D'autres opérateurs de coupe peuvent être définis, et ce aussi pour les domaines abstraits non-relationnels ou mixtes.

Exemple 6.1.12 – Soit une expression $\alpha v_i + \beta v_j, i, j \in [\![1,n]\!], \alpha, \beta \in \{-1,1\}$, nous définissons l'opérateur de coupe pour les octogones $X^\sharp \in \mathcal{O}^\sharp$ le long de cette expression comme :

$$\oplus_o(X^\sharp) = \{X^\sharp[(\alpha v_i + \beta v_j) \mapsto h],\ X^\sharp[(-\alpha v_i - \beta v_j) \mapsto -h]\}$$

où $h = (X^\sharp(\alpha v_i + \beta v_j) - X^\sharp(-\alpha v_i - \beta v_j))/2$, arrondi dans $\mathbb{F}$ dans la direction adéquate.

Cet opérateur de coupe correspond à celui présenté définition 4.3.1. En effet, couper le long d'une expression binaire $\alpha v_i + \beta v_j$ est équivalent à couper le domaine de la variable $v_i^{i,j}$ ou $v_j^{i,j}$ quand $i \neq j$, ou de la variable v_i ou v_j quand $i = j$.

Exemple 6.1.13 – Considérons un polyèdre $X^\sharp \in \mathcal{P}^\sharp$ représenté par un ensemble de contraintes linéaires, et une expression linéaire $\sum_i \beta_i v_i$. Nous définissons l'opérateur de coupe pour les polyèdres le long de cette expression linéaire comme :

$$\oplus_p(X^\sharp) = \left\{ X^\sharp \cup \left\{ \sum_i \beta_i v_i \leq h \right\}, X^\sharp \cup \left\{ \sum_i \beta_i v_i \geq h \right\} \right\}$$

avec $h = \left(\min_{\gamma(X^\sharp)} \sum_i \beta_i v_i + \max_{\gamma(X^\sharp)} \sum_i \beta_i v_i \right) /2$ pouvant être calculé par un algorithme du Simplexe [Dantzig et Thapa, 1997].

Exemple 6.1.14 – Considérons une boîte mixte $X^\sharp = (X_1, \ldots, X_m, I_{m+1}, \ldots, I_n) \in \mathcal{M}^\sharp$. Nous définissons l'opérateur de coupe tel que soit une des variables discrètes est instanciée, soit le domaine d'une des variables continues est coupé :

$$\oplus_m(X^\sharp) = \begin{cases} \{(X_1 \times \cdots \times X_{i-1} \times x \times X_{i+1} \times \cdots \times X_m \times I_{m+1} \times \cdots \times I_n) \\ \quad | \, x \in X_i\} & i \in [\![1,m]\!] \\ \{(X_1 \times \ldots X_m \times I_{m+1} \times \cdots \times I_{i-1} \times [a,h] \times I_{i+1} \times \cdots \times I_n), \\ \quad (X_1 \times \ldots X_m \times I_{m+1} \times \cdots \times I_{i-1} \times [h,b] \times I_{i+1} \times \cdots \times I_n)\ \} & i \in [\![m+1,n]\!] \end{cases}$$

où $\forall i \in [\![m+1,n]\!], I_i = [a,b]$ et $h = \overline{(a+b)/2}$ ou $h = \underline{(a+b)/2}$.

Ces opérateurs de coupe sont paramétrés par le choix d'une direction de coupe (le long d'une variable ou d'une expression). Pour un domaine non-relationnel, nous pouvons utiliser les deux stratégies classiques en programmation par contraintes : définir un ordre et couper les variables tour à tour suivant l'ordre choisi, ou couper le long de la variable ayant le plus grand ou le plus petit domaine ; c'est-à-dire $|S_i|$ pour une variable représentée par un ensemble ou $b - a$ pour une variable représentée par l'intervalle $[a,b]$ ou $[\![a,b]\!]$. On peut naturellement étendre ces stratégies aux octogones en remplaçant l'ensemble des variables par l'ensemble (fini) des expressions binaires. Pour les polyèdres, on peut couper le segment reliant les sommets les plus éloignés afin de réduire la précision τ_p. Cependant, même pour les domaines relationnels, nous pouvons utiliser un opérateur de coupe plus rapide et plus simple tel que couper le long de la variable ayant le plus grand domaine.

Définition 6.1.6 (Opérateur de sélection)**.** Un *opérateur de sélection* est un opérateur $\pi : \mathcal{E}^\sharp \to \mathcal{D}^\sharp$ tel que $\forall X^\sharp \in \mathcal{E}^\sharp$, pour un $r \in \mathbb{R}^{>0}$ fixe,

1. $\pi(X^\sharp) \in X^\sharp$, et
2. $(\tau \circ \pi)(X^\sharp) > r$.

L'opérateur de sélection choisit un élément de la complétion disjonctive $X^\sharp$, plus grand qu'une certaine valeur r.

Afin d'assurer la terminaison du solveur, nous imposons que toute suite de réductions, coupes et sélections retourne à partir d'un moment un élément assez petit pour τ :

Définition 6.1.7 (Compatibilité de τ et $\oplus$)**.** Les deux opérateurs $\tau : \mathcal{D}^\sharp \to \mathcal{R}^+$ et $\oplus : \mathcal{D}^\sharp \to \mathcal{E}^\sharp$ sont dits compatibles si et seulement si, pour tout opérateur de réduction $\rho^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ (*c-à-d,* $\forall X^\sharp \in \mathcal{D}^\sharp, \rho^\sharp(X^\sharp) \sqsubseteq^\sharp X^\sharp$) et pour toute famille d'opérateurs de sélection $\pi_i : \mathcal{E}^\sharp \to \mathcal{D}^\sharp$ (*c-à-d,* $\forall Y^\sharp \in \mathcal{E}^\sharp, \pi_i(Y^\sharp) \in Y^\sharp$) :

$$\forall e \in \mathcal{D}^\sharp,\ \forall r \in \mathbb{R}^{>0}, \exists K \text{ tel que } \forall j \geq K,\ (\tau \circ \pi_j \circ \oplus \circ \rho^\sharp \circ \cdots \circ \pi_1 \circ \oplus \circ \rho^\sharp)(e) \leq r$$

Tous les opérateurs $\oplus$ et $\rho^\sharp$ étant contractants, l'opérateur $\pi \circ \oplus \circ \rho^\sharp$ est aussi contractant. Et donc pour tout r strictement positif, la définition précédente est vérifiée.

On vérifie aisément que chacun des opérateurs de coupe présentés précédemment, $\oplus_a$, $\oplus_b$, $\oplus_h$, $\oplus_o$, $\oplus_p$, et $\oplus_m$, est compatible respectivement avec la fonction de précision, τ_a, τ_b, τ_h, τ_o, τ_p, et τ_m, proposée respectivement pour le domaine abstrait $\mathcal{S}^\sharp$, $\mathcal{I}^\sharp$, $\mathcal{B}^\sharp$, $\mathcal{O}^\sharp$, $\mathcal{P}^\sharp$ et $\mathcal{M}^\sharp$.

Remarque 6.1.2 – La procédure d'exploration peut être représentée par un arbre de recherche (définit section 2.2.4). Avec cette représentation, l'ensemble des nœuds à une profondeur donnée correspond à une disjonction sur-approximant l'ensemble des solutions. De plus, une série d'opérateurs de réduction (ρ), de sélection (π) et de coupe ($\oplus$) correspond à une branche. La définition 6.1.7 stipule que chaque branche de l'arbre de recherche est finie.

6.1.6 Résolution abstraite

L'algorithme de résolution abstraite est donné algorithme 6.1. Il maintient dans toExplore et sols deux disjonctions dans $\mathcal{E}^\sharp$, et itère les étapes suivantes :

1. choisir un élément abstrait e de toExplore (**pop**),
2. appliquer la consistance ($\rho^\sharp$),
3. et soit supprimer e, l'ajouter à la liste de solutions sols soit le couper ($\oplus$).

La résolution commence avec le plus grand élément $\top^\sharp$ de $\mathcal{D}^\sharp$ représentant $\gamma(\top^\sharp) = \hat{D}$.

Cet algorithme correspond à l'algorithme 3.1 dans lequel on remplace le domaine abstrait E défini en programmation par contraintes par le domaine abstrait $\mathcal{D}^\sharp$.

La terminaison est assurée par la proposition suivante :

Proposition 6.1.1. *Si τ et $\oplus$ sont compatibles, l'algorithme 6.1 termine.*

liste de domaines abstraits sols $\leftarrow \emptyset$ */* stocke les solutions abstraites */*
queue de domaines abstraits toExplore $\leftarrow \emptyset$ */* stocke les éléments abstraits à explorer */*
domaine abstrait $e \in \mathcal{D}^\sharp$

push $\top^\sharp$ dans toExplore */* initialisation avec l'espace de recherche abstrait : $\gamma(\top^\sharp) = \hat{D}$ */*

tant que toExplore $\neq \emptyset$ **faire**
  $e \leftarrow$ **pop**(toExplore)
  $e \leftarrow \rho^\sharp(e)$
  si $e \neq \emptyset$ **alors**
    si $\tau(e) \leq r$ **ou** isSol(e) **alors** */* isSol(e) retourne **vrai** si e ne contient que des solutions */*
      sols $\leftarrow$ sols $\cup\, e$
    sinon
      push $\oplus(e)$ dans toExplore
    fin si
  fin si
fin tant que

Algorithme 6.1: Solveur abstrait générique

Démonstration.

Montrons que l'arbre de recherche est fini. Supposons par l'absurde que l'arbre de recherche est infini. Sa largeur étant finie par définition 6.1.5, il devrait exister une branche infinie (lemme de König), ce qui contredit la définition 6.1.7.

L'arbre de recherche étant fini, l'algorithme 6.1 termine.

□

Proposition 6.1.2. *L'algorithme 6.1 est correct.*

Démonstration.

À chaque itération, $\bigcup\{\gamma(x) \mid x \in \text{toExplore} \cup \text{sols}\}$ est une sur-approximation de l'ensemble de solutions, car la consistance $\rho^\sharp$ est une abstraction du ρ de la sémantique concrète pour les contraintes et l'opérateur de coupe $\oplus$ est une abstraction de l'identité. Notons que les éléments abstraits dans sols sont consistants et soit contiennent uniquement des solutions soit sont plus petits que r. L'algorithme termine quand toExplore est vide, et donc sols sur-approxime l'ensemble de solutions avec des éléments consistants contenant uniquement des solutions ou plus petits que r. Afin de calculer l'exact ensemble des solutions dans le cas discret, il suffit de choisir $r = 1$.

Donc l'algorithme 6.1 calcule une sur-approximation ou une approximation exacte dans le cas discret de l'ensemble des solutions.

□

La résolution présentée algorithme 6.1 utilise une structure de données de file et coupe le plus ancien élément abstrait en premier. Des stratégies de sélection plus élaborées, telles que couper l'élément le plus grand pour τ, peuvent être définies. Comme les deux propositions précédentes

ne dépendent pas du choix des opérateurs, quelle que soit la stratégie choisie, l'algorithme reste correct et termine.

Comme les itérations locales en interprétation abstraite, notre solveur effectue des itérations abstraites décroissantes. Plus précisément, $\mathrm{toExplore} \cup \mathrm{sols}$ est décroissant pour $\sqsubseteq^\sharp_{\mathcal{E}}$ dans le domaine de complétion disjonctive $\mathcal{E}^\sharp$ à chaque itération de la boucle. En effet, $\rho^\sharp$ est contractant dans $\mathcal{D}^\sharp$, $\rho^\sharp(e) \sqsubseteq^\sharp e$, de plus chacun des éléments retournés par l'opérateur de coupe est inclus dans l'élément de départ, $\oplus(e) \sqsubseteq^\sharp_{\mathcal{E}} \{e\}$. Cependant, notre solveur diffère de l'interprétation abstraite classique sur deux points. Premièrement, il n'existe pas d'opérateurs de coupe en interprétation abstraite, les nouveaux éléments d'une complétion disjonctive étant généralement ajoutés uniquement au niveau des intersections dans un flot de contrôle. Lors de l'analyse d'une conditionnelle par exemple, deux éléments sont créés, l'un vérifiant la condition, l'autre ne la vérifiant pas. Le premier est utilisé pour l'analyse des instructions du bloc *alors*, le second pour l'analyse du bloc *sinon*. Cependant, une fois sorti de la conditionnelle, l'union abstraite $\sqcup^\sharp$ des deux éléments est effectuée, et l'analyse continue avec un seul élément. Deuxièmement, la stratégie de résolution itérative est plus élaborée qu'en interprétation abstraite. L'utilisation de l'opérateur de rétrécissement est remplacée par une structure de données permettant de maintenir une liste ordonnée des éléments abstraits et une stratégie de coupe permettant un processus de raffinement et assure sa terminaison. En programmation par contraintes, il existe de très nombreux algorithmes calculant efficacement la consistance d'arc généralisée. Par exemple, dans l'algorithme AC-5 [van Hentenryck *et al.*, 1992] chaque fois que le domaine d'une variable change, la variable décide quelles contraintes doivent être propagées. La conception d'algorithmes efficaces pour la propagation est un challenge scientifique en programmation par contraintes [Schulte et Tack, 2001].

6.2 Le solveur AbSolute

Nous avons implémenté un prototype de solveur abstrait, d'après les idées ci-dessus, afin de démontrer la faisabilité de notre approche. Les principales caractéristiques et des résultats préliminaires sont présentés ici.

6.2.1 Implémentation

Notre prototype de solveur, appelé AbSolute, est implémenté au-dessus de Apron, une bibliothèque OCaml de domaines abstraits numériques pour l'analyse statique [Jeannet et Miné, 2009]. Différents domaines abstraits sont implémentés dans Apron, tels que les intervalles, les octogones et les polyèdres. De plus, Apron offre une API uniforme permettant de cacher les algorithmes internes des domaines abstraits. Ainsi, une fonction peut être implémentée de façon générique, sans tenir compte du domaine abstrait utilisé. Par exemple, la consistance peut être implémentée de manière générale pour tout domaine abstrait, et n'a pas besoin d'être implémentée de façon spécifique pour chacun des domaines abstraits.

Une autre caractéristique importante d'Apron est que les domaines abstraits sont définis pour les variables entières et réelles. En d'autres termes, un domaine abstrait peut être défini pour un ensemble de variables pouvant contenir uniquement des variables entières ou réelles, mais aussi pour un ensemble de variables contenant à la fois des variables entières et réelles.

Finalement, Apron fournit un langage de contraintes arithmétiques suffisant pour exprimer la plupart des CSP numériques : égalités et inégalités d'expressions numériques, comprenant les opérateurs $+$, $-$, $\times$, $/$, $\sqrt{}$, puissance, modulo et arrondi vers des entiers.

Pour toutes ces raisons, nous avons choisi Apron et tirons avantages de ses domaines abstraits, de son API uniforme, et de sa gestion des variables entières et réelles et des contraintes non linéaires.

6.2.1.1 Modélisation d'un problème

En programmation par contraintes, un problème est formalisé sous la forme d'un CSP. Afin de représenter un CSP dans AbSolute, nous stockons les variables dans un environnement composé de deux tableaux, l'un pour les variables entières l'autre pour les variables réelles. Les contraintes pouvant être exprimées dans Apron, elles sont stockées dans un tableau. En revanche, les domaines des variables n'étant pas une notion existant en interprétation abstraite, ils sont traduits en contraintes linéaires et stockés dans un tableau de contraintes.

Exemple 6.2.1 – Considérons le CSP sur les variables entières v_1, v_2 de domaine $D_1 = D_2 = [\![1, 5]\!]$ et la variable réelle v_3 de domaine $D_3 = [-10, 10]$ avec les contraintes
$C_1 : 6v_1 + 4v_2 - 3v_3 = 0$
$C_2 : v_1 \times v_2 \geq 3.5$

Il se traduira dans AbSolute de la façon suivante :

```
let v1 = Var.of_string "v1";;
let v2 = Var.of_string "v2";;
let v3 = Var.of_string "v3";;
let csp =
  let vars = Environment.make [|v1; v2|] [|v3|] in
  let doms = Parser.lincons1_of_lstring vars ["v1 ≥ 1";"v1 ≤ 5";"v2 ≥ 1";
  "v2 ≤ 5";"v3 ≥ −10";"v3 ≤ 10"] in
  let cons = Parser.tcons1_of_lstring vars ["6 ∗ v1 − 4 ∗ v2 − 3 ∗ v3 = 0"; "v1 ∗
  v2 ≥ 3.5"] in
(vars, doms, cons);;
```

Les trois premières lignes correspondent à la création des variables. Elles sont créées avec la fonction `Var.of_string` qui prend en paramètre la chaîne de caractères associée à la variable. Puis le CSP est créé, l'ensemble des variables (`vars`) est créé avec la fonction `Environment.make` dont le premier argument est l'ensemble des variables entières (ici $\{v_1, v_2\}$) et le second, l'ensemble des variables réelles ($\{v_3\}$). Puis les domaines sont créés à partir de la fonction `Parser.lincons1_of_lstring` qui transforme le tableau de chaînes de caractères en un tableau de contraintes linéaires sur les variables `vars`. Finalement, la conjonction des contraintes est créée avec la fonction `Parser.tcons1_of_lstring` qui transforme le tableau de chaînes de caractères en un tableau de contraintes (pouvant être non-linéaires) sur les variables `vars`.

6.2.1.2 Abstraction

Un domaine abstrait $\mathcal{D}^\sharp$ est choisi et initialisé à l'aide des domaines du CSP. Il est créé grâce à un "manager". Chacun des domaines abstraits possède son propre manager. Ensuite, les fonctions de transfert des contraintes linéaires correspondant aux domaines sont appelées afin que le domaine abstrait corresponde à l'espace de recherche initial.

Exemple 6.2.2 – Continuons l'exemple 6.2.1. Afin de créer un domaine abstrait à partir des domaines, on écrira dans AbSolute :

```
let abs = Abstract1.of_lincons_array man vars doms;;
```

où `man` correspond au type de domaine abstrait que l'on veut créer, par exemple on utilisera `Oct.manager_alloc()` pour les octogones. La fonction `Abstract1.of_lincons_array` permet de créer directement un domaine abstrait pour les variables `vars` à partir d'un tableau de contraintes linéaires `doms`.

Finalement, $\mathcal{D}^\sharp$ est représenté par un domaine abstrait de l'interprétation abstraite (tel que les octogones) et une conjonction de contraintes correspondant aux domaines en programmation par contraintes ($v_1 \geq 1, v_1 \leq 5$...).

6.2.1.3 Consistance

Les fonctions de transfert de tests fournissent naturellement des propagateurs pour les contraintes. En interne, chaque domaine abstrait implémente ses propres algorithmes afin de gérer les tests, comprenant des méthodes sophistiquées permettant de traiter les contraintes non-linéaires. Par exemple, dans le domaine abstrait des intervalles, l'algorithme HC4-Revise est implémenté afin de propager au mieux les contraintes non-linéaires. Pour tous les domaines abstraits, si aucune méthode n'arrive à propager les contraintes non-linéaires, celles-ci sont linéarisées en utilisant l'algorithme décrit dans [Miné, 2004]. Cet algorithme remplace dans chacun des termes non-linéaires un sous-ensemble des variables par l'intervalle des valeurs possibles pour ces variables, on obtient ainsi une contrainte *quasi-linéaire*, c'est-à-dire une contrainte dont certains des coefficients sont des intervalles. Puis tous les intervalles $[a, b]$ sont remplacés par leur valeur médiane $(a + b/2)$, et l'intervalle $[(a - b)/2, (b - a)/2]$ est ajouté aux constantes.

Exemple 6.2.3 – Toujours en considérant le CSP de l'exemple 6.2.1, la contrainte C_2 correspond aux contraintes quasi-linéaires suivantes :

$C_{2.1} : [\![1, 5]\!]v_2 \geq 3.5$
$C_{2.2} : [\![1, 5]\!]v_1 \geq 3.5$

La contrainte $C_{2.1}$ correspond à la contrainte C_2 dans laquelle la variable v_1 a été remplacée par son domaine $[\![1, 5]\!]$. De même, la contrainte $C_{2.2}$ correspond à la contrainte C_2 dans laquelle la variable v_2 a été remplacée par son domaine $[\![1, 5]\!]$. Finalement, les intervalles sont remplacés par leur valeur médiane et l'intervalle $[\![-2, 2]\!]$ est ajouté aux constantes. On obtient les contraintes linéaires suivantes :

$C'_{2.1} : 3v_2 + [\![-2, 2]\!] \geq 3.5$
$C'_{2.2} : 3v_1 + [\![-2, 2]\!] \geq 3.5$

D'autres algorithmes de linéarisation pourraient être considérés, tels que celui proposé dans [Borradaile et van Hentenryck, 2005] qui, dans la dernière étape, remplace les intervalles par soit la borne supérieure, soit la borne inférieure en fonction de la contrainte considérée et de l'intervalle de départ.

Afin de simuler la boucle de propagation, notre solveur effectue des itérations locales jusqu'à ce qu'un point fixe ou un nombre maximum d'itérations soit atteint. Ce nombre maximal est fixé à trois afin d'éviter les convergences lentes et ainsi assurer une résolution rapide.

Exemple 6.2.4 – Considérons le CSP sur les variables réelles v_1 et v_2 de domaine $D_1 = D_2 = [-4, 4]$ avec les contraintes

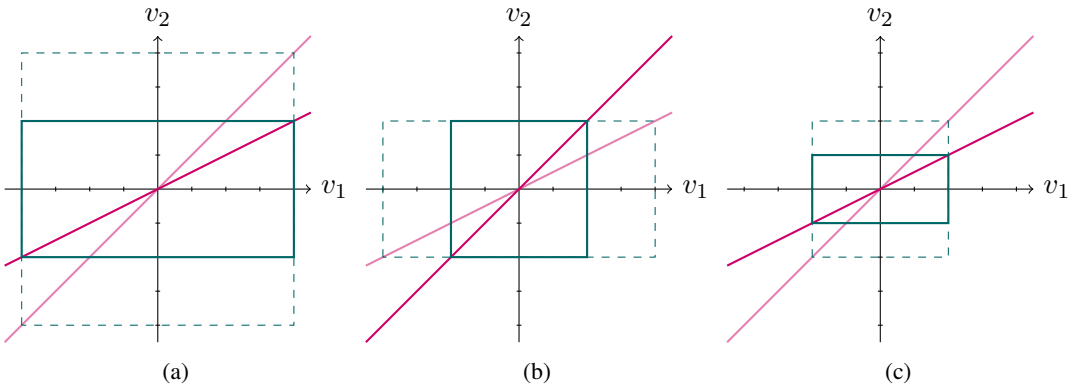


Figure 6.1 – Exemple des trois premières itérations d'un calcul de consistance avec une convergence lente.

$C_1 : v_1 = v_2$
$C_2 : v_1 = \frac{1}{2} v_2$

Ce problème admet une unique solution $v_1 = v_2 = 0$, et la consistance a une convergence très lente. En effet, en appliquant les propagateurs itérativement, le point fixe n'est jamais atteint. L'application du propagateur de la contrainte C_2 réduit le domaine de v_2 à $[-2, 2]$, puis le propagateur de C_1 réduit le domaine de v_1 à $[-2, 2]$, puis le propagateur de C_2 réduit le domaine de v_2 à $[-1, 1]$ et ainsi de suite. Les propagateurs réduisent le domaine d'une des variables de moitié à chaque appel. Les trois premières itérations sont illustrées figure 6.1.

Remarque 6.2.1 – Afin d'éviter les convergences lentes, AbSolute n'effectue, par défaut, que 3 itérations de la consistance. Dans les solveurs de programmation par contraintes, tels qu'Ibex, la consistance est arrêtée si à l'itération précédente les domaines ont été réduits de moins de 10%.

Remarque 6.2.2 – Dans les solveurs de programmation par contraintes, seules les contraintes contenant au moins une variable dont le domaine a été modifié à l'itération précédente sont propagées. Cependant, pour des raisons de simplicité, notre solveur propage toutes les contraintes à chaque itération.

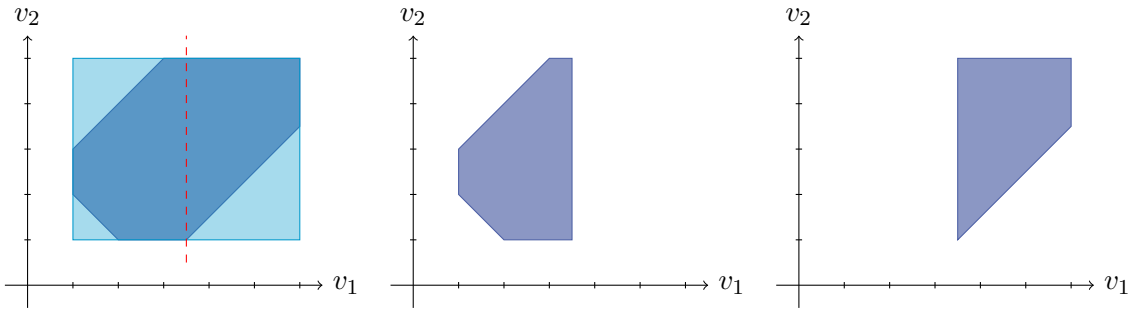
Exemple 6.2.5 – Continuons l'exemple 6.2.2. Un fois le domaine abstrait créé, la consistance peut être appelée de la façon suivante dans AbSolute :

```
let abs = consistency man abs cons max_iter;;
```

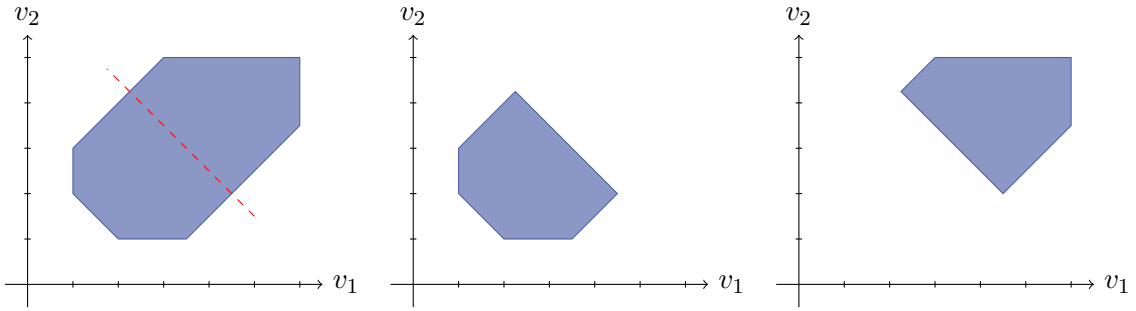
où `max_iter` est le nombre d'itérations maximal pouvant être effectuées et dont la valeur par défaut est 3. Le type du domaine abstrait `man` est nécessaire afin qu'Apron puisse exécuter les bonnes fonctions de transfert.

6.2.1.4 Opérateur de coupe

Notre solveur permet de couper le long d'une variable. L'opérateur de coupe utilisé coupe le plus grand domaine en deux, et ce quel que soit le domaine abstrait choisi (relationnel ou non). Lors de l'appel à l'opérateur de coupe, celui-ci récupère la boîte englobante de l'élément à couper, calcule la plus grande dimension sur les axes et la coupe en deux.



(a) Opérateur de coupe dans AbSolute



(b) Opérateur de coupe octogonal

Figure 6.2 – Comparaison entre l'opérateur de coupe naïf développé dans AbSolute appliqué à un octogone, et l'opérateur de coupe octogonal tel que donné définition 4.3.1.

La figure 6.2(a), illustre un exemple de cet opérateur de coupe appliqué à un octogone. La boîte englobante est d'abord calculée, puis le domaine de la variable v_1 est coupé le long de la ligne rouge en pointillés. À titre de comparaison, la figure 6.2(b), illustre la coupe effectuée par l'opérateur de coupe octogonal tel que donné définition 4.3.1.

Exemple 6.2.6 – Continuons l'exemple 6.2.5. Un fois le domaine abstrait consistant calculé, on peut le couper avec l'opérateur de coupe, ce qui se traduit dans AbSolute par :

```
let list_abs = split man abs vars;;
```

où l'opérateur de coupe `split` retourne une liste de domaines abstraits dont l'union abstraite est équivalente au domaine abstrait de départ `abs`.

Comparée à celles de la plupart des solveurs en programmation par contraintes, la stratégie de coupe est très basique, et ne dépend ni du domaine abstrait ni du type des variables (figure 6.2(a)). Des stratégies plus intelligentes de la littérature de programmation par contraintes sont à envisager dans des travaux futurs, telles que celles définies section 2.2.5.

6.2.1.5 Cas particulier des polyèdres

Dans le cas particulier du domaine abstrait des polyèdres, nous nous sommes rendus compte, en pratique, qu'il fallait limiter la taille des polyèdres. En effet, lors de la résolution avec les polyèdres, celle-ci peut être très lente, car la consistance n'est pas toujours contractante. Elle va parfois ajouter des expressions linéaires qui ne sont pas toujours très pertinentes. Prenons par

exemple le polyèdre pour les variables v_1 et v_2 composé des expressions linéaires suivantes :

$$\{v_1 \geq 1, v_1 \leq 5, v_2 \geq 1, v_2 \leq 5\}$$

La consistance d'un ensemble de contraintes appliquée à ce polyèdre peut vouloir ajouter l'expression linéaire $y - 5x \leq 0$ au polyèdre. Cette expression linéaire est vraie pour tous points du polyèdre, elle est donc considérée comme non pertinente.

Finalement, dans le cas du domaine abstrait des polyèdres, seul l'opérateur de coupe est contractant. Nous avons donc décidé de limiter la taille des polyèdres en limitant leurs nombre de faces. Plutôt qu'imposer un nombre fixe de faces, nous avons choisi de limiter les valeurs pour les coefficients des expressions linéaires du polyèdre. Ainsi nous n'avons pas à vérifier à chaque coupe que les polyèdres ont bien le bon nombre de faces.

Nous avons arbitrairement mis à 20 la valeur maximale pour les coefficients des expressions linéaires des polyèdres. Cependant nous n'avons pas eu le temps de faire des expérimentations exhaustives afin de déterminer la meilleure valeur possible pour cette limite, si elle existe. De plus, sur les tests que nous avons faits, nous pouvons voir que cette valeur n'est pas assez limitante.

6.2.1.6 Résolution

Nous avons implémenté deux versions du processus de résolution. La première version correspond à la méthode de résolution donnée dans l'algorithme 6.1. La seconde s'arrête dés que la première solution est trouvée. Dans ces deux versions, une file est utilisée afin de maintenir l'ensemble des éléments abstraits à explorer. Aucun opérateur de sélection n'est implémenté pour le moment et les éléments abstraits sont visités selon leur ordre d'ajout dans la file. Ces deux méthodes de résolution sont paramétrées par le domaine abstrait utilisé et la valeur r. Plus important encore, elles ne dépendent pas du problème à résoudre et ne dépendent donc pas du type des variables. Ainsi, elles peuvent être utilisées afin de résoudre des problèmes contenant à la fois des variables entières et réelles.

6.2.2 Résultats expérimentaux

Nous avons testé AbSolute sur deux classes de problèmes : la première est constituée de problèmes continus, le but étant de comparer l'efficacité d'AbSolute à celle d'un solveur classique en programmation par contraintes. La deuxième est composée de problèmes mixtes, que les solveurs classiques ne peuvent pas traiter.

6.2.2.1 Résolution continue

Nous avons utilisé des problèmes du benchmark COCONUT * contenant uniquement des variables continues et comparé les résultats obtenus avec AbSolute à ceux obtenus avec Ibex †, un solveur continu basé sur les intervalles. De plus, nous avons comparé AbSolute à notre extension d'Ibex pour les octogones (présentée section 5.4.1), ce qui nous permet de comparer le choix du domaine abstrait (intervalles vs. octogones) indépendamment du choix du solveur (solveur classique de PPC ou solveur basé sur l'IA). Les tableaux 6.1 et 6.2 donnent les temps de calculs en secondes pour trouver toutes les solutions ou uniquement la première pour chacun des problèmes.

∗. Disponible à l'adresse http://www.mat.univie.ac.at/~neum/glopt/coconut/.
†. Disponible à l'adresse http://www.emn.fr/z-info/ibex/.

			Intervals		Octagons	
name	# vars	ctr type	Ibex	AbSolute	Ibex	AbSolute
b	4	=	**0.02**	0.10	0.26	**0.14**
nbody5.1	6	=	**95.99**	1538.25	**27.08**	$\geq$ 1h
ipp	8	=	**38.83**	39.24	**279.36**	817.86
brent-10	10	=	**21.58**	263.86	**330.73**	$\geq$ 1h
KinematicPair	2	$\leq$	59.04	**23.14**	60.78	**31.11**
biggsc4	4	$\leq$	800.91	**414.94**	1772.52	**688.56**
o32	5	$\leq$	27.36	**22.66**	40.74	**33.17**

Table 6.1 – Comparaison du temps CPU, en secondes, pour trouver toutes les solutions, avec l'implémentation en Ibex et AbSolute.

			Intervals		Octagons	
name	# vars	ctr type	Ibex	AbSolute	Ibex	AbSolute
b	4	=	**0.009**	0.018	0.053	**0.048**
nbody5.1	6	=	**32.85**	708.47	**0.027**	$\geq$ 1h
ipp	8	=	**0.66**	9.64	19.28	**1.46**
brent-10	10	=	7.96	**4.57**	**0.617**	$\geq$ 1h
KinematicPair	2	$\leq$	**0.013**	0.018	0.016	**0.011**
biggsc4	4	$\leq$	**0.011**	0.022	0.096	**0.029**
o32	5	$\leq$	**0.045**	0.156	**0.021**	0.263

Table 6.2 – Comparaison du temps CPU, en secondes, pour trouver la première solution, avec l'implémentation en Ibex et AbSolute.

En moyenne, AbSolute est compétitif comparé à l'approche de programmation par contraintes. Plus précisément, il est globalement plus lent sur les problèmes contenant des égalités et plus rapide sur les problèmes contenant des inégalités. Ces différences de performances ne semblent pas être dues au type des contraintes mais plutôt au ratio : nombre de contraintes dans lesquelles une variable apparait sur le nombre total de contraintes. Comme précisé précédemment, à chaque itération, toutes les contraintes sont propagées, même celles dont aucune des variables n'a été modifiée. Ceci augmente le temps de calcul à chaque itération et donc augmente le temps de calcul total. Par exemple, le problème `brent-10` possède dix variables, dix contraintes et chaque variable apparaît dans au plus trois contraintes. Même si une seule variable a été modifiée, nous propageons toutes les dix contraintes au lieu de trois au maximum. Ceci explique probablement les mauvaises performances d'AbSolute sur les problèmes `brent-10` et `nbody5.1`.

De plus, dans AbSolute, la boucle de propagation est arrêtée après trois itérations, alors que dans l'approche de programmation par contraintes on la poursuit jusqu'au point fixe. De ce fait, la consistance dans AbSolute peut être moins précise que celle utilisée dans Ibex, ce qui réduit le temps passé lors de la propagation mais peut augmenter la phase d'exploration.

Ces expériences montrent que notre prototype, bien que n'ayant que très peu des stratégies de programmation par contraintes, se comporte raisonnablement bien sur un benchmark classique. De futurs travaux comprendront une analyse plus poussée des performances et bien sûr des améliorations d'AbSolute sur ses faiblesses identifiées (stratégie de coupe, boucle de propagation). La boucle de propagation est un point important dans les méthodes de résolution PPC-complètes car elle est appelée en tout point de l'arbre de recherche, et son efficacité influe sur l'efficacité de la méthode de résolution.

name	# vars		ctr type	$\mathcal{M}^\sharp$	$\mathcal{O}^\sharp$	$\mathcal{P}^\sharp$
	int	real				
gear4	4	2	$=$	**0.017**	0.048	0.415
st_miqp5	2	5	$\leq$	**2.636**	3.636	$\geq$ 1h
ex1263	72	20	$= \leq$	**473.933**	$\geq$ 1h	$\geq$ 1h
antennes_4_3	6	2	$\leq$	**520.766**	1562.335	$\geq$ 1h

Table 6.3 – Comparison du temps CPU, en secondes, obtenu par AbSolute en utilisant différents domaines abstraits disponibles dans Apron.

name	# vars		ctr type	$\mathcal{M}^\sharp$	$\mathcal{O}^\sharp$	$\mathcal{P}^\sharp$
	int	real				
gear4	4	2	$=$	**0.016**	0.036	0.296
st_miqp5	2	5	$\leq$	**0.672**	1.152	$\geq$ 1h
ex1263	72	20	$= \leq$	**8.747**	$\geq$ 1h	$\geq$ 1h
antennes_4_3	6	2	$\leq$	**3.297**	22.545	$\geq$ 1h

Table 6.4 – Première solution.

6.2.2.2 Résolution mixte discret-continu

Comme les solveurs en programmation par contraintes traitent rarement des problèmes mixtes, il n'existe pas de benchmark standard. Nous avons donc rassemblé des problèmes de MinLP-Lib [‡], une bibliothèque de problèmes mixtes d'optimisation issus de la communauté de Recherche Opérationnelle. Ces problèmes ne sont pas des problèmes de satisfaction de contraintes mais des problèmes d'optimisation, c'est-à-dire des problèmes ayant une fonction à minimiser. Nous les avons donc traduits en problèmes de satisfaction en utilisant la même méthode que celle présentée dans [Berger et Granvilliers, 2009]. Nous avons remplacé chaque critère d'optimisation $\min f(x)$ par une contrainte $|f(x) - \text{best\_known\_value}| \leq \epsilon$. Nous comparons les résultats obtenus par AbSolute au schéma de résolution proposé dans [Berger et Granvilliers, 2009], utilisant le même ϵ et les mêmes problèmes, et trouvons des résultats similaires en termes de temps d'exécution (nous ne fournissons pas une comparaison détaillée, ce serait apporter de l'information dénuée de sens en raison des différences de machines).

Plus intéressant, nous observons qu'AbSolute peut résoudre des problèmes mixtes en un temps raisonnable et se comporte mieux avec les intervalles qu'avec les domaines relationnels. Une raison possible est que les propagations et heuristiques actuelles ne sont pas en mesure d'utiliser pleinement l'information relationnelle disponible avec les octogones ou les polyèdres. Dans la section 5.3.1, nous avions suggéré qu'un opérateur de coupe soigneusement conçu est la clef pour obtenir une résolution octogonale efficace. De futurs travaux incorporeront les idées développées pour la résolution octogonale en programmation par contraintes dans notre solveur. Cependant, AbSolute est en mesure de faire face naturellement à des problèmes mixtes de programmation par contraintes en un temps raisonnable, ouvrant la voie à de nouvelles applications de la programmation par contraintes telles que le problème de restauration et réparation du réseau électrique après une catastrophe naturelle [Simon *et al.*, 2012], où l'on cherche à établir un plan d'action ainsi que le trajet à effectuer par chaque équipe de réparation afin de rétablir l'électricité le plus rapidement possible après une catastrophe naturelle. Une autre application est celle des problèmes géométriques, une extension du problème présenté dans [Beldiceanu *et al.*, 2007], dans ces problèmes,

‡. Disponible à l'adresse `http://www.gamsworld.org/minlp/minlplib.htm`.

on cherche à placer des objets de formes variées (polygones, variétés algébriques, *etc.*) de façon à ce qu'aucun objet ne recouvre un autre tout en minimisant la surface ou le volume occupé.

6.3 Conclusion

Dans ce chapitre nous avons exploré certains liens entre l'interprétation abstraite et la programmation par contraintes, et les avons utilisés pour concevoir un schéma de résolution de problèmes de satisfaction de contraintes entièrement basé sur les domaines abstraits. Les résultats obtenus avec notre prototype sont encourageants et ouvrent la voie au développement de solveurs PPC–IA permettant de traiter naturellement les problèmes mixtes.

Dans de futurs travaux, nous souhaitons améliorer notre solveur en adaptant et intégrant les méthodes avancées de la littérature en programmation par contraintes. Les améliorations incluent des opérateurs de coupe pour les domaines abstraits, des propagations spécialisées, telles que la consistance octogonale ou des contraintes globales, et plus important encore, l'amélioration de la boucle de propagation.

Nous avons conçu notre solveur sur des abstractions de façon modulaire, ainsi les nouvelles méthodes et les méthodes existantes peuvent être combinées, comme ce qui est fait pour le produit réduit en interprétation abstraite. Lors de l'analyse d'un programme, plusieurs domaines abstraits sont généralement utilisés. Dans ce cas, le produit réduit est utilisé afin de communiquer aux domaines abstraits les informations recueillies au cours de l'analyse par les autres domaines abstraits. Finalement, chaque problème sera résolu dans le domaine abstrait qui lui sied le mieux comme ce qui est fait en interprétation abstraite. Ainsi les problèmes linéaires seront résolus avec les polyèdres et les polynômes du second degré avec des ellipsoïdes.

Un autre développement intéressant est d'utiliser certaines méthodes de la programmation par contraintes dans un analyseur statique d'interprétation abstraite, telles que l'utilisation d'opérateur de coupe dans les complétion disjonctive, ainsi que la capacité de la programmation par contraintes d'affiner un élément abstrait pour réaliser la complétude.

CHAPITRE 7

Conclusion et Perspectives

7.1 Conclusion

Dans cette thèse, nous avons d'abord étudié et comparé certaines méthodes de programmation par contraintes et d'interprétation abstraite. Nous nous sommes plus particulièrement intéressés aux outils permettant de sur-approximer des domaines impossibles ou difficiles à calculer exactement (domaine concret en interprétation abstraite, ensemble des solutions en programmation par contraintes). Nous nous sommes appuyés sur cette étude pour abstraire la notion de domaine utilisée en programmation par contraintes, au sens où la représentation choisie pour les domaines devient un paramètre du solveur. Cela revient à intégrer les domaines abstraits de l'interprétation abstraite au cadre de la programmation par contraintes, et permet d'utiliser de nouvelles représentations de domaines, notamment des représentations relationnelles, qui capturent certaines relations entre variables (par exemple, le domaine abstrait des polyèdres intègre des relations linéaires). Par ailleurs, les domaines abstraits nous ont permis de définir un cadre de résolution unifié indépendant du type des variables, intégrant uniformément les méthodes de résolution discrètes et continues de programmation par contraintes.

Dans un deuxième temps, nous avons défini et implémenté le domaine abstrait faiblement relationnel des octogones en programmation par contraintes pour des variables continues. Nous avons adapté l'opérateur de clôture inférieure existant en interprétation abstraite sur les octogones afin de concevoir une consistance octogonale optimale. Nous avons ensuite utilisé les relations entre variables exprimées par les octogones pour guider la recherche. Les tests réalisés sur un benchmark classique montrent que ces heuristiques sont efficaces. Cette nouvelle méthode de résolution montre l'intérêt de développer de nouvelles représentations des domaines en programmation par contraintes, en particulier pour intégrer des domaines relationnels plus expressifs que le produit cartésien généralement utilisé en programmation par contraintes. Cependant, pour ce faire, il faut, pour chaque nouveau domaine, redéfinir des consistances et opérateurs de coupes adéquats, ce qui constitue un obstacle.

Or, des opérateurs proches de la consistance ont déjà été définis en interprétation abstraite pour de nombreux domaines abstraits. Nous avons donc réalisé le travail inverse du précédent et, au lieu d'intégrer des notions d'interprétation abstraite en programmation par contraintes, nous avons exprimé la programmation par contraintes dans le cadre de l'interprétation abstraite, en considérant l'ensemble des solutions cherchées comme un domaine concret. Nous avons ainsi redéfini la résolution de contraintes avec les opérateurs d'interprétation abstraite. Ceci nous permet d'utiliser directement les domaines abstraits existants. Nous avons implémenté cette méthode au-

dessus d'Apron, une bibliothèque de domaines abstraits. Le prototype développé, AbSolute, bien que n'intégrant pas encore les stratégies d'exploration de la programmation par contraintes, a donné de premiers résultats prometteurs.

7.2 Perspectives

Cette thèse ouvre plusieurs perspectives de recherche. Premièrement, à court terme, AbSolute doit être développé pour intégrer des techniques de recherche plus fines, qu'il s'agisse des heuristiques de choix de variables/valeur, des opérateurs de coupe ou du paramétrage de la boucle de propagation (nombre et ordre des itérations locales). Pour les domaines relationnels (octogones, polyèdres) en particulier, le travail sur les octogones semble montrer que la clef de l'efficacité est de s'appuyer sur les relations entre variables pour guider la recherche. Pour les domaines mixtes, on peut améliorer la méthode actuelle en définissant et implémentant des produits réduits entre les domaines réels et entiers. Par ailleurs, AbSolute doit aussi être testé sur des applications réelles, notamment pour des problèmes mixtes tels que le problème de restauration et réparation du réseau électrique après une catastrophe naturelle [Simon *et al.*, 2012]. Les résultats obtenus sur ces applications permettront de travailler à des heuristiques dédiées.

À moyen terme, il faudrait étudier les nombreux domaines abstraits existant en interprétation abstraite, tels que les intervalles polyédriques, les polyèdres, les zonotopes ou les ellipsoïdes, dans le cadre de la programmation par contraintes. Cela suppose de définir les consistances et opérateurs de coupe associés. En particulier, pour les polyèdres, il faudrait trouver une approximation efficace des contraintes non-linéaires, ce qui nécessite une étude approfondie des algorithmes de linéarisation ou de quasi-linéarisation existants.

À plus long terme, nous prévoyons enfin d'exploiter le lien entre les sous-approximations en programmation par contraintes et le point fixe calculé en interprétation abstraite. En interprétation abstraite, l'efficacité des analyseurs repose en grande partie sur le calcul de l'opérateur d'élargissement (widening). Nous pouvons exploiter les travaux importants sur le widening pour les adapter à une méthode de résolution d'approximation intérieure en programmation par contraintes.

Bibliographie

[Abdallah *et al.*, 1996] ABDALLAH, C., DORATO, P., LISKA, R., STEINBERG, S. et YANG, W. (1996). Applications of quantifier elimination theory to control theory. *In Proceedings of the 4th IEEE Mediterranean Symposium on New Directions in Control and Automation*.

[Alberganti, 2005] ALBERGANTI, M. (2005). L'avion qui "bat des ailes" a fédéré de nombreux chercheurs. *Le Monde*, 18741:18.

[Alpuente *et al.*, 1993] ALPUENTE, M., FALASCHI, M., RAMIS, M. J. et VIDAL, G. (1993). Narrowing approximations as an optimization for equational logic programs. *In Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming (PLILP '93)*, Lecture Notes in Computer Science, pages 391–409. Springer-Verlag.

[analyse le code de Curiosity,] analyse le code de CURIOSITY, C. http://www.coverity.com/html/press/nasa-jpl-relies-on-coverity-to-ensure-the-seamless-touchdown-and-operation-of-the-curiosity-mars-rover.html.

[analyse le code des installations nucléaire,] analyse le code des installations NUCLÉAIRE, P. http://www.mathworks.fr/company/user_stories/Institute-for-Radiological-Protection-and-Nuclear-SafetyVerifies-Nuclear-Safety-Software-with-Polyspace-Productsfor-CC++.html.

[analyse le code du LHC,] analyse le code du LHC, C. http://www.coverity.com/html/press/cern-chooses-coverity-to-ensure-accuracy-of-large-hadron-collider-software.html.

[Analyser,] ANALYSER, P. http://www.mathworks.fr/products/polyspace.

[Ansótegui *et al.*, 2009] ANSÓTEGUI, C., SELLMANN, M. et TIERNEY, K. (2009). A gender-based genetic algorithm for the automatic configuration of algorithms. *In Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP'09)*, volume 5732 de *Lecture Notes in Computer Science*, pages 142–157. Springer-Verlag.

[Apt, 1999] APT, K. R. (1999). The essence of constraint propagation. *Theoretical Computer Science*, 221.

[Apt, 2003] APT, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.

[Apt et Wallace, 2007] APT, K. R. et WALLACE, M. (2007). *Constraint logic programming using Eclipse*. Cambridge University Press.

[Araya *et al.*, 2012] ARAYA, I., NEVEU, B. et TROMBETTONI, G. (2012). An interval extension based on occurrence grouping. *Computing*, 94:173–188.

[Araya *et al.*, 2010] ARAYA, I., TROMBETTONI, G. et NEVEU, B. (2010). Exploiting monotonicity in interval constraint propagation. *In Proceedings of the 24th AAAI Conference on Artificial Intelligence, AAAI 2010*.

[Arbelaez *et al.*, 2009] ARBELAEZ, A., HAMADI, Y. et SEBAG, M. (2009). Online heuristic selection in constraint programming. *In Proceedings of the 4th International Symposium on Combinatorial Search (SoCS 2009)*.

[Bagnara *et al.*, 2005a] BAGNARA, R., HILL, P. M., MAZZI, E. et ZAFFANELLA, E. (2005a). Widening operators for weakly-relational numeric abstractions. *In Proceedings of the 12th International Static Analysis Symposium (SAS'05)*, volume 3672 de *Lecture Notes in Computer Science*, pages 3–18. Springer.

[Bagnara *et al.*, 2005b] BAGNARA, R., HILL, P. M., RICCI, E. et ZAFFANELLA, E. (2005b). Precise widening operators for convex polyhedra. *Science of Computer Programming - Special issue : Static analysis symposium (SAS 2003)*, 58(1-2):28–56.

[Bagnara *et al.*, 2006] BAGNARA, R., HILL, P. M. et ZAFFANELLA, E. (2006). Widening operators for powerset domains. *International Journal on Software Tools for Technology Transfer*, 8(4):449–466.

[Bagnara *et al.*, 2009] BAGNARA, R., HILL, P. M. et ZAFFANELLA, E. (2009). Weakly-relational shapes for numeric abstractions : improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279–323.

[Beldiceanu *et al.*, 2007] BELDICEANU, N., CARLSSON, M., PODER, E., SADEK, R. et TRUCHET, C. (2007). A generic geometrical constraint kernel in space and time for handling polymorphic *k*-dimensional objects. *In Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 de *Lecture Notes in Computer Science*, pages 180–194. Springer.

[Benhamou, 1995] BENHAMOU, F. (1995). Interval constraint logic programming. *In Constraint Programming : Basics and Trends*, volume 910 de *Lecture Notes in Computer Science*, pages 1–21. Springer Berlin / Heidelberg.

[Benhamou, 1996] BENHAMOU, F. (1996). Heterogeneous constraint solvings. *In Proceedings of the 5th International Conference on Algebraic and Logic Programming*, pages 62–76.

[Benhamou *et al.*, 1997] BENHAMOU, F., GOUALARD, F. et GRANVILLIERS, L. (1997). Programming with the DecLIC Language. *In Proceedings of the 2nd International Workshop on Interval Constraints*.

[Benhamou *et al.*, 1999] BENHAMOU, F., GOUALARD, F., GRANVILLIERS, L. et PUGET, J.-F. (1999). Revisiting hull and box consistency. *In Proceedings of the 16th International Conference on Logic Programming*, pages 230–244.

[Benhamou *et al.*, 2004] BENHAMOU, F., GOUALARD, F., LANGUENOU, É. et CHRISTIE, M. (2004). Interval constraint solving for camera control and motion planning. *ACM Transactions on Computational Logic*, 5(4):732–767.

[Benhamou *et al.*, 1994] BENHAMOU, F., MCALLESTER, D. A. et van HENTENRYCK, P. (1994). Clp(intervals) revisited. *In Proceedings of the 1994 International Symposium on Logic programming (ILPS '94)*, pages 124–138. MIT Press.

[Benhamou et Older, 1997] BENHAMOU, F. et OLDER, W. J. (1997). Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1):1–24.

[Berger et Granvilliers, 2009] BERGER, N. et GRANVILLIERS, L. (2009). Some interval approximation techniques for minlp. *In Proceedings of the The 8th Symposium on Abstraction, Reformulation and Approximation (SARA'09)*.

[Bertrane *et al.*, 2010] BERTRANE, J., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A. et RIVAL, X. (2010). Static analysis and verification of aerospace software by abstract interpretation. *In AIAA Infotech@Aerospace 2010*, Atlanta, Georgia. American Institute of Aeronautics and Astronautics.

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

[Bessière, 2006] BESSIÈRE, C. (2006). Constraint propagation. *In* ROSSI, F., van BEEK, P. et WALSH, T., éditeurs : *Handbook of Constraint Programming*, chapitre 3. Elsevier.

[Bessiere *et al.*, 2011] BESSIERE, C., CARDON, S., DEBRUYNE, R. et LECOUTRE, C. (2011). Efficient algorithms for singleton arc consistency. *Constraints*, 16(1):25–53.

[Bessière *et al.*, 1999] BESSIÈRE, C., FREUDER, E. C. et RÉGIN, J.-C. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148.

[Bessière *et al.*, 2004] BESSIÈRE, C., HEBRARD, E., HNICH, B. et WALSH, T. (2004). The complexity of global constraints. *In Proceedings of the 19th Conference on Artificial Intelligence (AAAI'04)*, pages 112–117.

[Bessière et Régin, 1996] BESSIÈRE, C. et RÉGIN, J.-C. (1996). Mac and combined heuristics : Two reasons to forsake fc (and cbj ?) on hard problems. *In Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, volume 1118 de *Lecture Notes in Computer Science*. Springer.

[Bessière et Régin, 2001] BESSIÈRE, C. et RÉGIN, J.-C. (2001). Refining the basic constraint propagation algorithm. *In Proceedings of the 17th International Joint Conference on Artificial intelligence (IJCAI'01)*, pages 309–315. Morgan Kaufmann.

[Bessière et van Hentenryck, 2003] BESSIÈRE, C. et van HENTENRYCK, P. (2003). To be or not to be ... a global constraint. *In Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 de *Lecture Notes in Computer Science*, pages 789–794. Springer.

[Borradaile et van Hentenryck, 2005] BORRADAILE, G. et van HENTENRYCK, P. (2005). Safe and tight linear estimators for global optimization. *Mathematical Programming*, 102:495 – 517.

[Bourdoncle, 1992] BOURDONCLE, F. (1992). Abstract interpreting by dynamic partitioning. *Journal of Functional Programming*, 2:407–435.

[Boussemart *et al.*, 2004] BOUSSEMART, F., HEMERY, F., LECOUTRE, C. et SAIS, L. (2004). Boosting systematic search by weighting constraints. *In Proceedings of the 16th Eureopean Conference on Artificial Intelligence, (ECAI'2004)*, pages 146–150. IOS Press.

[Brélaz, 1979] BRÉLAZ, D. (1979). New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256.

[Chabert et Jaulin, 2009] CHABERT, G. et JAULIN, L. (2009). Contractor programming. *Artificial Intelligence*, 173:1079–1100.

[Chabert *et al.*, 2009] CHABERT, G., JAULIN, L. et LORCA, X. (2009). A constraint on the number of distinct vectors with application to localization. *In Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP'09)*, pages 196–210, Berlin, Heidelberg. Springer-Verlag.

[checker CBMC,] checker CBMC, M. http://www.cprover.org/cbmc/.

[Chen *et al.*, 2009] CHEN, L., MINÉ, A., WANG, J. et COUSOT, P. (2009). Interval polyhedra : An abstract domain to infer interval linear relationships. *In Proceedings of the 16th International Static Analysis Symposium (SAS'09)*, pages 309–325.

[Chi *et al.*, 2008] CHI, K., JIANG, X., HORIGUCHI, S. et GUO, M. (2008). Topology design of network-coding-based multicast networks. *IEEE Transactions on Parallel and Distributed Systems*, 19(5):627–640.

[Choi *et al.*, 2006] CHOI, C. W., HARVEY, W., LEE, J. H.-M. et STUCKEY, P. J. (2006). Finite domain bounds consistency revisited. *In Proceedings of the 19th Australian joint conference on Artificial Intelligence : advances in Artificial Intelligence (AI'06)*, volume 4304 de *Lecture Notes in Computer Science*, pages 49–58. Springer-Verlag.

[Christie *et al.*, 2006] CHRISTIE, M., NORMAND, J.-M. et TRUCHET, C. (2006). Calcul d'approximations intérieures pour la résolution de max-ncsp. *In Proceedings des Deuxièmes Journées Francophones de Programmation par Contraintes (JFPC06)*.

[Clarisó et Cortadella, 2004] CLARISÓ, R. et CORTADELLA, J. (2004). The octahedron abstract domain. *In Proceedings of the 11th International Static Analysis Symposium (SAS'04)*, pages 312–327.

[Clarke *et al.*, 2003] CLARKE, E., GRUMBERG, O., JHA, S., LU, Y. et VEITH, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50:752–794.

[Collavizza *et al.*, 1999] COLLAVIZZA, H., DELOBEL, F. et RUEHER, M. (1999). Extending consistent domains of numeric csp. *In Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 406–413.

[Collavizza et Rueher, 2007] COLLAVIZZA, H. et RUEHER, M. (2007). Exploring different constraint-based modelings for program verification. *In Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 de *Lecture Notes in Computer Science*, pages 49–63. Springer.

[Colmerauer, 1994] COLMERAUER, A. (1994). Spécifications de prolog iv. Rapport technique, Faculté des Sciences de Luminy, 163, Avenue de Luminy - 13288 Marseille cedex 9 - France.

[Cortesi, 2008] CORTESI, A. (2008). Widening operators for abstract interpretation. *In* CERONE, A. et GRUNER, S., éditeurs : *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods*, pages 31–40.

[Cortesi et Zanioli, 2011] CORTESI, A. et ZANIOLI, M. (2011). Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42.

[Cousot et Cousot, 1976] COUSOT, P. et COUSOT, R. (1976). Static determination of dynamic properties of programs. *In Proceedings of the 2nd International Symposium on Programming*, pages 106–130.

[Cousot et Cousot, 1977a] COUSOT, P. et COUSOT, R. (1977a). Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California. ACM Press, New York, NY.

[Cousot et Cousot, 1977b] COUSOT, P. et COUSOT, R. (1977b). Static determination of dynamic properties of generalized type unions. *In Proceedings of an ACM conference on Language design for reliable software*, pages 77–94, New York, NY, USA. ACM.

[Cousot et Cousot, 1979] COUSOT, P. et COUSOT, R. (1979). Systematic design of program analysis frameworks. *In Proceedings of the 6th ACM SIGACT-SIGPLAN symposium of Principles of Programming Languages*, pages 269–282.

[Cousot et Cousot, 1992a] COUSOT, P. et COUSOT, R. (1992a). Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547.

[Cousot et Cousot, 1992b] COUSOT, P. et COUSOT, R. (1992b). Comparing the galois connection and widening/narrowing approaches to abstract interpretation. *In* BRUYNOOGHE, M. et WIRSING, M., éditeurs : *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 de *Lecture Notes in Computer Science*, pages 269–295. Springer.

[Cousot *et al.*, 2007] COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D. et RIVAL, X. (2007). Combination of abstractions in the ASTRÉE static analyzer. *In Proceedings of the 11th Asian computing science conference on Advances in computer science : secure software and related issues (ASIAN'06)*, volume 4435 de *Lecture Notes in Computer Science*, pages 272–300. Springer-Verlag.

[Cousot *et al.*, 2011] COUSOT, P., COUSOT, R. et MAUBORGNE, L. (2011). The reduced product of abstract domains and the combination of decision procedures. *In Proceedings of the 14th International Conference on Fondations of Software Science and Computation Structures (FoSSaCS 2011)*, volume 6604 de *Lecture Notes in Computer Science*, pages 456–472. Springer-Verlag.

[Cousot et Halbwachs, 1978] COUSOT, P. et HALBWACHS, N. (1978). Automatic discovery of linear restraints among variables of a program. *In Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96.

[Coverity,] COVERITY. http://www.coverity.com/.

[Dantzig et Thapa, 1997] DANTZIG, G. B. et THAPA, M. N. (1997). *Linear programming 1 : introduction*. Springer-Verlag.

[de la Commission d'enquête Ariane 501,] de la Commission d'enquête ARIANE 501, R. http://www.astrosurf.com/luxorion/astronautique-accident-ariane-v501.htm.

[Dechter, 1990] DECHTER, R. (1990). Enhancement schemes for constraint processing : backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312.

[Dechter, 2003] DECHTER, R. (2003). *Constraint processing*. Elsevier Morgan Kaufmann.

[Dechter *et al.*, 1989] DECHTER, R., MEIRI, I. et PEARL, J. (1989). Temporal constraint networks. *In Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*.

[Dechter et Pearl, 1987] DECHTER, R. et PEARL, J. (1987). Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38.

[Defense,] DEFENSE, G. R. P. M. http://www.fas.org/spp/starwars/gao/im92026.htm.

[des contraintes globales,] des contraintes GLOBALES, C. http://www.emn.fr/z-info/sdemasse/gccat/.

[Diaz *et al.*, 2012] DIAZ, D., ABREU, S. et CODOGNET, P. (2012). On the implementation of gnu prolog. *In Theory and Practice of Logic Programming*, volume 12, pages 253–282. Cambridge University Press.

[D'Silva, 2006] D'SILVA, V. (2006). *Widening for automata*. Thèse de doctorat, Universität Zürich.

[D'Silva *et al.*, 2012] D'SILVA, V., HALLER, L. et KROENING, D. (2012). Satisfiability solvers are static analysers. *In Proceedings of the 19th International Static Analysis Symposium (SAS'12)*, volume 7460 de *Lecture Notes in Computer Science*. Springer.

[Fages et Lorca, 2011] FAGES, J.-G. et LORCA, X. (2011). Revisiting the tree constraint. *In Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 de *Lecture Notes in Computer Science*, pages 271–285. Springer-Verlag.

[Feret, 2004] FERET, J. (2004). Static analysis of digital filters. *In* SPRINGER, éditeur : *European Symposium on Programming (ESOP'04)*, volume 2986, pages 33–48.

[Floyd, 1962] FLOYD, R. (1962). Algorithm 97 : Shortest path. *Communications of the ACM*, 5(6).

[Freuder, 1978] FREUDER, E. C. (1978). Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966.

[Freuder, 1982] FREUDER, E. C. (1982). A sufficient condition for backtrack-free search. *Journal of the ACM (JACM)*, 29(1):24–32.

[Freuder, 1997] FREUDER, E. C. (1997). In pursuit of the holy grail. *Constraints*, 2(1):57–61.

[Frost et Dechter, 1995] FROST, D. et DECHTER, R. (1995). Look-ahead value ordering for constraint satisfaction problems. *In Proceedings of the 14th International Joint Conference on Artificial intelligence (IJCAI'95)*, pages 572–578. Morgan Kaufmann Publishers Inc.

[Geelen, 1992] GEELEN, P. A. (1992). Dual viewpoint heuristics for binary constraint satisfaction problems. *In Proceedings of the 10th European conference on Artificial intelligence (ECAI'92)*, pages 31–35. John Wiley & Sons, Inc.

[Gent *et al.*, 2006] GENT, I. P., JEFFERSON, C. et MIGUEL, I. (2006). Minion : A fast, scalable, constraint solver. *In Proceedings of 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 98–102. IOS Press.

[Gent *et al.*, 1996] GENT, I. P., MACINTYRE, E., PROSSER, P., SMITH, B. M. et WALSH, T. (1996). An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. *In Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming*, volume 1118 de *Lecture Notes in Computer Science*, pages 179–193. Springer.

[Gent *et al.*, 2008] GENT, I. P., MIGUEL, I. et NIGHTINGALE, P. (2008). Generalised arc consistency for the alldifferent constraint : An empirical survey. *Artificial Intelligence*, 172(18):1973–2000.

[Ginsberg *et al.*, 1990] GINSBERG, M. L., FRANK, M., HALPIN, M. P. et TORRANCE, M. C. (1990). Search lessons learned from crossword puzzles. *In Proceedings of the 8th National conference on Artificial Intelligence (AAAI'90)*, pages 210–215. AAAI Press.

[Goldberg, 1991] GOLDBERG, D. (1991). What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48.

[Goldsztejn et Granvilliers, 2010] GOLDSZTEJN, A. et GRANVILLIERS, L. (2010). A new framework for sharp and efficient resolution of ncsp with manifolds of solutions. *Constraints*, 15(2):190–212.

[Granger, 1992] GRANGER, P. (1992). Improving the results of static analyses of programs by local decreasing iterations. *In Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*.

[Granvilliers et Benhamou, 2006] GRANVILLIERS, L. et BENHAMOU, F. (2006). Realpaver : An interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32(1):138–156.

[Grimes et Hebrard, 2011] GRIMES, D. et HEBRARD, E. (2011). Models and strategies for variants of the job shop scheduling problem. *In Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 de *Lecture Notes in Computer Science*, pages 356–372. Springer-Verlag.

[Halbwachs et Henry, 2012] HALBWACHS, N. et HENRY, J. (2012). When the decreasing sequence fails. *In Proceedings of the 19th International Static Analysis Symposium (SAS'12)*, volume 7460 de *Lecture Notes in Computer Science*. Springer.

[Hansen, 1992] HANSEN, E. (1992). *Global optimization using interval analysis*. Marcel Dekker.

[Haralick et Elliott, 1979] HARALICK, R. M. et ELLIOTT, G. L. (1979). Increasing tree search efficiency for constraint satisfaction problems. *In Proceedings of the 6th International Joint Conference on Artificial intelligence (IJCAI'79)*, pages 356–364. Morgan Kaufmann Publishers Inc.

[Havelund *et al.*,] HAVELUND, K., GROCE, A., SMITH, M. et BARRINGER, H. Monitoring the execution of space craft flight software. http://compass.informatik.rwth-aachen.de/ws-slides/havelund.pdf.

[Hermenier *et al.*, 2011] HERMENIER, F., DEMASSEY, S. et LORCA, X. (2011). Bin repacking scheduling in virtualized datacenters. *In Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 de *Lecture Notes in Computer Science*, pages 27–41. Springer-Verlag.

[Hervieu *et al.*, 2011] HERVIEU, A., BAUDRY, B. et GOTLIEB, A. (2011). Pacogen : Automatic generation of pairwise test configurations from feature models. *In Proceedings of the 22nd International Symposium on Software Reliability Engineering*, pages 120–129.

[Hickey et Ju, 1997] HICKEY, T. et JU, Q. (1997). Efficient implementation of interval arithmetic narrowing using ieee arithmetic. Rapport technique, IEEE Arithmetic, Brandeis University CS Dept.

[Hoeve, 2004] HOEVE, W. (2004). A hyper-arc consistency algorithm for the soft alldifferent constraint. *In Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 de *Lecture Notes in Computer Science*, pages 679–689. Springer.

[Jeannet et Miné, 2009] JEANNET, B. et MINÉ, A. (2009). Apron : A library of numerical abstract domains for static analysis. *In Proceedings of the 21th International Conference Computer*

Aided Verification (CAV 2009), volume 5643 de *Lecture Notes in Computer Science*, pages 661–667. Springer.

[Kask *et al.*, 2004] KASK, K., DECHTER, R. et GOGATE, V. (2004). Counting-based look-ahead schemes for constraint satisfaction. *In Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 de *Lecture Notes in Computer Science*, pages 317–331. Springer.

[Katriel et Thiel, 2003] KATRIEL, I. et THIEL, S. (2003). Fast bound consistency for the global cardinality constraint. *In Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 de *Lecture Notes in Computer Science*, pages 437–451. Springer Berlin / Heidelberg.

[Kearfott, 1996] KEARFOTT, R. B. (1996). *Rigorous global search : continuous problems*. Kluwer.

[King, 1969] KING, J. C. (1969). *A Program Verifier*. Thèse de doctorat, Carnegie Mellon University, Pittsburgh, USA.

[Kroening et Strichman, 2008] KROENING, D. et STRICHMAN, O. (2008). *Decision procedures*. Springer.

[Lacan *et al.*, 1998] LACAN, P., MONFORT, J. N., RIBAL, L. V. Q., DEUTSCH, A. et GONTHIER, G. (1998). Ariane 5 - the software reliability verification process. *In Proceedings of the conference on Data Systems in Aerospace*.

[Lazaar *et al.*, 2012] LAZAAR, N., GOTLIEB, A. et LEBBAH, Y. (2012). A cp framework for testing cp. *Constraints*, 17(2):123–147.

[Lecoutre *et al.*, 2003] LECOUTRE, C., BOUSSEMART, F. et HEMERY, F. (2003). Exploiting multidirectionality in coarse-grained arc consistency algorithms. *In Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 de *Lecture Notes in Computer Science*, pages 480–494. Springer.

[López-Ortiz *et al.*, 2003] LÓPEZ-ORTIZ, A., QUIMPER, C.-G., TROMP, J. et van BEEK, P. (2003). A fast and simple algorithm for bounds consistency of the all different constraint. *In Proceedings of the 18th International Joint Conference on Artificial intelligence (IJCAI'03)*, pages 245–250. Morgan Kaufmann Publishers Inc.

[Mackworth, 1977a] MACKWORTH, A. K. (1977a). Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118.

[Mackworth, 1977b] MACKWORTH, A. K. (1977b). On reading sketch maps. *In Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 598–606.

[Mackworth et Freuder, 1982] MACKWORTH, A. K. et FREUDER, E. C. (1982). The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Rapport technique, University of British Columbia.

[Mehlhorn et Thiel, 2000] MEHLHORN, K. et THIEL, S. (2000). Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. *In Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP '00)*, volume 1894 de *Lecture Notes in Computer Science*, pages 306–319. Springer.

[Menasche et Berthomieu, 1983] MENASCHE, M. et BERTHOMIEU, B. (1983). Time petri nets for analyzing and verifying time dependent communication protocols. *In Protocol Specification, Testing, and Verification*.

[Miné, 2004] MINÉ, A. (2004). *Domaines numériques abstraits faiblement relationnels*. Thèse de doctorat, École Normale Supérieure.

[Miné, 2006] MINÉ, A. (2006). The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100.

[Miné, 2012] MINÉ, A. (2012). Abstract domains for bit-level machine integer and floating-point operations. *In Proceedings of The 4th International Workshop on Invariant Generation (WING'12)*, EpiC, page 16. EasyChair.

[Mohr et Henderson, 1986] MOHR, R. et HENDERSON, T. C. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233.

[Mohr et Masini, 1988] MOHR, R. et MASINI, G. (1988). Good old discrete relaxation. *In Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656.

[Monniaux, 2009] MONNIAUX, D. (2009). A minimalistic look at widening operators. *Higher Order and Symbolic Computation*, 22(2):145–154.

[Montanari, 1974] MONTANARI, U. (1974). Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132.

[Moore, 1966] MOORE, R. E. (1966). *Interval Analysis*. Prentice-Hall, Englewood Cliffs N. J.

[Moreno-Navarro *et al.*, 1993] MORENO-NAVARRO, J. J., KUCHEN, H., no CARBALLO, J. M., WINKLER, S. et HANS, W. (1993). Efficient lazy narrowing using demandedness analysis. *In Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming (PLILP '93)*, Lecture Notes in Computer Science, pages 167–183. Springer-Verlag.

[Pachet et Roy, 2001] PACHET, F. et ROY, P. (2001). Musical harmonization with constraints : A survey. *Constraints*, 6(1):7–19.

[Pelleau *et al.*, a] PELLEAU, M., MINÉ, A., TRUCHET, C. et BENHAMOU, F. A constraint solver based on abstract domains. (soumis).

[Pelleau *et al.*, b] PELLEAU, M., TRUCHET, C. et BENHAMOU, F. The octagon abstract domain for continuous constraints. *Constraints*. (soumis).

[Pelleau *et al.*, 2011] PELLEAU, M., TRUCHET, C. et BENHAMOU, F. (2011). Octagonal domains for continuous constraints. *In Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 de *Lecture Notes in Computer Science*, pages 706–720. Springer-Verlag.

[Pelleau *et al.*, 2009] PELLEAU, M., van HENTENRYCK, P. et TRUCHET, C. (2009). Sonet network design problems. *In Proceedings of the 6th International Workshop on Local Search Techniques in Constraint Satisfaction*, pages 81–95.

[Perriquet et Barahona, 2009] PERRIQUET, O. et BARAHONA, P. (2009). Constraint-based strategy for pairwise rna secondary structure prediction. *In Proceedings of the 14th Portuguese Conference on Artificial Intelligence : Progress in Artificial Intelligence (EPIA '09)*, volume 5816 de *Lecture Notes in Computer Science*, pages 86–97. Springer-Verlag.

[Petit *et al.*, 2011] PETIT, T., RÉGIN, J.-C. et BELDICEANU, N. (2011). A Θ(n) bound-consistency algorithm for the increasing sum constraint. *In Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 de *Lecture Notes in Computer Science*, pages 721–728. Springer-Verlag.

[Ponsini *et al.*, 2011] PONSINI, O., MICHEL, C. et RUEHER, M. (2011). Refining abstract interpretation-based approximations with constraint solvers. *In Proceedings of the 4th International Workshop on Numerical Software Verification*.

[Puget, 1998] PUGET, J.-F. (1998). A fast algorithm for the bound consistency of alldiff constraints. *In Proceedings of the 15th National/10th Conference on Artificial Intelligence/Innovative applications of artificial intelligence (AAAI '98/IAAI '98)*, pages 359–366. American Association for Artificial Intelligence.

[Quimper *et al.*, 2003] QUIMPER, C.-G., van BEEK, P., LÓPEZ-ORTIZ, A., GOLYNSKI, A. et SADJAD, S. (2003). An efficient bounds consistency algorithm for the global cardinality constraint. *In Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 de *Lecture Notes in Computer Science*, pages 600–614. Springer Berlin / Heidelberg.

[Ramamoorthy *et al.*, 2011] RAMAMOORTHY, V., SILAGHI, M. C., MATSUI, T., HIRAYAMA, K. et YOKOO, M. (2011). The design of cryptographic s-boxes using csps. *In Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 de *Lecture Notes in Computer Science*, pages 54–68. Springer-Verlag.

[Ratzet, 1994] RATZET, D. (1994). Box-splitting strategies for the interval gauss-seidel step in a global optimization method. *Computing*, 53:337–354.

[Rival et Mauborgne, 2007] RIVAL, X. et MAUBORGNE, L. (2007). The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5).

[Robinson, 1999] ROBINSON, S. (1999). Beyond 2000 : Further troubles lurk in the future of computing. *The New York Times*.

[Rodriguez-Carbonell et Kapur, 2004] RODRIGUEZ-CARBONELL, E. et KAPUR, D. (2004). An abstract interpretation approach for automatic generation of polynomial invariants. *In Proceedings of the 11th International Static Analysis Symposium (SAS'04)*, volume 3148 de *Lecture Notes in Computer Science*, pages 280–295. Springer.

[Rossi *et al.*, 2006] ROSSI, F., van BEEK, P. et WALSH, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier.

[Schulte, 2002] SCHULTE, C. (2002). *Programming Constraint Services*, volume 2302 de *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

[Schulte et Stuckey, 2005] SCHULTE, C. et STUCKEY, P. J. (2005). When do bounds and domain propagation lead to the same search space ? *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):388–425.

[Schulte et Tack, 2001] SCHULTE, C. et TACK, G. (2001). Implementing efficient propagation control. *In Proceedings of the 3rd workshop on Techniques for Implementing Constraint Programming Systems*.

[Sellmann, 2002] SELLMANN, M. (2002). An arc-consistency algorithm for the minimum weight all different constraint. *In Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 de *Lecture Notes in Computer Science*, pages 744–749. Springer-Verlag.

[Simon et Chen, 2010] SIMON, A. et CHEN, L. (2010). Simple and precise widenings for *h*-polyhedra. *In* UEDA, K., éditeur : *Proceedings of the 8th Asian Symposium on Programming*

Languages and Systems (APLAS 2010), volume 6461 de *Lecture Notes in Computer Science*, pages 139–155. Springer.

[Simon et King, 2006] SIMON, A. et KING, A. (2006). Widening polyhedra with landmarks. *In* KOBAYASHI, N., éditeur : *Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS 2006)*, volume 4279 de *Lecture Notes in Computer Science*, pages 166–182. Springer.

[Simon *et al.*, 2012] SIMON, B., COFFRIN, C. et van HENTENRYCK, P. (2012). Randomized adaptive vehicle decomposition for large-scale power restoration. *In Proceedings of the 9th international conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'12)*, volume 7298 de *Lecture Notes in Computer Science*, pages 379–394. Springer-Verlag.

[Souyris et Delmas, 2007] SOUYRIS, J. et DELMAS, D. (2007). Experimental assessment of astrée on safety-critical avionics software. *In Proceedings of the 26th International Conference on Computer Safety, Reliability, and Security*, pages 479–490.

[Stølevik *et al.*, 2011] STØLEVIK, M., NORDLANDER, T. E., RIISE, A. et FRØYSETH, H. (2011). A hybrid approach for solving real-world nurse rostering problems. *In Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 de *Lecture Notes in Computer Science*, pages 85–99. Springer-Verlag.

[Team, 2010] TEAM, C. (2010). Choco : an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes.

[Truchet et Assayag, 2011] TRUCHET, C. et ASSAYAG, G., éditeurs (2011). *Constraint Programming in Music*. ISTE.

[Truchet *et al.*, 2010] TRUCHET, C., PELLEAU, M. et BENHAMOU, F. (2010). Abstract domains for constraint programming, with the example of octagons. *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 72–79.

[van Beek, 2006] van BEEK, P. (2006). Backtracking search algorithms. *In* ROSSI, F., van BEEK, P. et WALSH, T., éditeurs : *Handbook of Constraint Programming*, chapitre 4. Elsevier.

[van Hentenryck *et al.*, 1992] van HENTENRYCK, P., DEVILLE, Y. et TENG, C.-M. (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57.

[van Hentenryck et Michel, 2005] van HENTENRYCK, P. et MICHEL, L. (2005). *Constraint-Based Local Search*. MIT Press.

[van Hentenryck *et al.*, 1997] van HENTENRYCK, P., MICHEL, L. et DEVILLE, Y. (1997). *Numerica : a Modeling Language for Global Optimization*. MIT Press.

[van Hentenryck *et al.*, 1995] van HENTENRYCK, P., SARASWAT, V. A. et DEVILLE, Y. (1995). Design, implementation, and evaluation of the constraint language cc(fd). *In Selected Papers from Constraint Programming : Basics and Trends*, pages 293–316. Springer-Verlag.

[van Hentenryck *et al.*, 2008] van HENTENRYCK, P., YIP, J., GERVET, C. et DOOMS, G. (2008). Bound consistency for binary length-lex set constraints. *In Proceedings of the 23rd National Conference on Artificial intelligence (AAAI'08)*, pages 375–380. AAAI Press.

[Wolinski *et al.*, 2004] WOLINSKI, C., KUCHCINSKI, K. et GOKHALE, M. (2004). A constraints programming approach to communication scheduling on sopc architectures. *In Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA '04)*, pages 252–252. ACM.

[Zhang et Yap, 2000] ZHANG, Y. et YAP, R. H. C. (2000). Arc consistency on n-ary monotonic and linear constraints. *In Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP '00)*, volume 1894 de *Lecture Notes in Computer Science*, pages 470–483. Springer-Verlag.

Liste des figures

Liste des définitions

Liste des exemples

Domaines abstraits en programmation par contraintes

Marie PELLEAU

Résumé

La programmation par contraintes permet de formaliser et résoudre des problèmes fortement combinatoires, dont le temps de calcul évolue en pratique exponentiellement. Les méthodes développées aujourd'hui résolvent efficacement de nombreux problèmes industriels de grande taille dans des solveurs génériques. Cependant, les solveurs restent dédiés à un seul type de variables : réelles ou entières, et résoudre des problèmes mixtes discrets-continus suppose des transformations *ad hoc*. Dans un autre domaine, l'interprétation abstraite permet de prouver des propriétés sur des programmes, en étudiant une abstraction de leur sémantique concrète, constituée des traces des variables au cours d'une exécution. Plusieurs représentations de ces abstractions, appelées domaines abstraits, ont été proposées. Traitées de façon générique dans les analyseurs, elles peuvent mélanger les types entiers, réels et booléens, ou encore représenter des relations entre variables. Dans cette thèse, nous définissons des domaines abstraits pour la programmation par contraintes, afin de construire une méthode de résolution traitant indifféremment les entiers et les réels. Cette généralisation permet d'étudier des domaines relationnels, comme les octogones déjà utilisés en interprétation abstraite. En exploitant l'information spécifique aux octogones pour guider la recherche de solutions, nous obtenons de bonnes performances sur les problèmes continus. Dans un deuxième temps, nous définissons notre méthode générique avec des outils d'interprétation abstraite, pour intégrer les domaines abstraits existants. Notre prototype, AbSolute, peut ainsi résoudre des problèmes mixtes et utiliser les domaines relationnels implémentés.

Mots-clés : Programmation par contraintes, Interprétation abstraite, Domaine abstrait, Octogone, Continu-discret.

Abstract

Constraint Programming aims at solving hard combinatorial problems, with a computation time increasing in practice exponentially. The methods are today efficient enough to solve large industrial problems, in a generic framework. However, solvers are dedicated to a single variable type: integer or real. Solving mixed problems relies on *ad hoc* transformations. In another field, Abstract Interpretation offers tools to prove program properties, by studying an abstraction of their concrete semantics, that is, the set of possible values of the variables during an execution. Various representations for these abstractions have been proposed. They are called abstract domains. Abstract domains can mix any type of variables, and even represent relations between the variables. In this PhD dissertation, we define abstract domains for Constraint Programming, so as to build a generic solving method, dealing with both integer and real variables. We can also study the octagons abstract domain, already defined in Abstract Interpretation. Guiding the search by the octagonal relations, we obtain good results on a continuous benchmark. In a second part, we define our solving method using Abstract Interpretation techniques, in order to include existing abstract domains. Our solver, AbSolute, is able to solve mixed problems and use relational domains.

Keywords: Constraint Programming, Abstract Interpretation, Abstract Domain, Octagon, Continuous-Discrete.