Relaxation de Contraintes pour les problèmes dynamiques

Narendra Jussien

N° Ordre : 1783

Thèse

présentée

devant l' Université de Rennes 1

pour obtenir

le grade de : **Docteur de l'Université de Rennes 1**

Mention : Informatique

Par

M. Narendra JUSSIEN

Équipe d'accueil : École des Mines de Nantes – Département Informatique
École doctorale : Sciences pour l'Ingénieur
Composante universitaire : IFSIC

Titre de la Thèse

Relaxation de contraintes pour les problèmes dynamiques

Soutenue le 24 octobre 1997 devant la commission d'Examen

Composition du Jury

Jean-Pierre	**Banâtre**	Professeur, Université de Rennes I	Président
François	**Fages**	Chargé de Recherche CNRS, LIENS	Rapporteur
Marie-Catherine	**Vilarem**	Professeur, Université de Montpellier II	Rapporteur
Pierre	**Cointe**	Professeur, École des Mines de Nantes	Examinateur
Marie-Odile	**Cordier**	Professeur, Université de Rennes I	Examinateur
Patrice	**Boizumault**	Professeur, École des Mines de Nantes	Directeur de Thèse

Remerciements

Les travaux présentés dans cette thèse ont été menés au sein de l'équipe «Contraintes» du département Informatique de l'École des Mines de Nantes. Que Monsieur Patrice Boizumault, responsable de l'équipe et directeur de cette thèse, soit ici remercié pour tout ce qu'il a fait pour moi depuis ce jour de 1992, je n'étais alors qu'en licence à l'IMA, où je lui ai annoncé que j'envisageais de poursuivre mes études supérieures jusqu'au doctorat.

Je tiens à remercier aussi tous les membres du département Informatique pour leur accueil chaleureux. Je tiens à remercier plus particulièrement Monsieur Philippe David pour son extrême disponibilité, son attention à la lecture et ses conseils toujours pertinents et Monsieur Olivier Lhomme dont l'abondance d'idées et de conseils ont permis de grandes avancées à mes travaux.

Je remercie le corps enseignant de l'Institut de Mathématiques Appliquées (en particulier Messieurs Éric Pinson et Yvon L'Hospitalier) pour m'avoir formé et soutenu.

Je tiens à remercier tous les membres du Jury pour avoir bien voulu y participer et, en particulier, Madame Marie-Catherine Vilarem et Monsieur François Fages qui ont accepté d'être rapporteurs de cette thèse.

Bien sûr, je remercie mes parents sans qui rien de tout cela ne serait arrivé.

Enfin, je veux remercier Mademoiselle Christelle Guéret qui était près de moi pendant cette thèse et qui, je l'espère, le restera encore pendant de nombreuses années.

Table des matières

Chapitre 1

Introduction

1.1 Contexte du travail

La programmation par contraintes, carrefour de diverses disciplines – l'intelligence artificielle, la recherche opérationnelle, la réécriture, le calcul formel, l'analyse numérique – a montré son intérêt dans de nombreux domaines d'application tels que les problèmes combinatoires, l'ordonnancement, l'analyse financière, la simulation et la synthèse de circuits intégrés, le diagnostic de pannes, l'aide à la décision ... La programmation par contraintes offre une réelle efficacité pour la modélisation et la résolution de systèmes complexes. De nombreux systèmes opérationnels sont commercialisés : PrologIII [Colmerauer, 1990] et maintenant PrologIV de la société PrologIA, CHIP [Dincbas *et al.*, 1988] de la société Cosytec ou encore Solver [Puget, 1994] de la société Ilog.

De nombreux problèmes réels interagissent avec leur environnement : le système de contraintes les définissant n'est donc pas figé. Par exemple, une interface graphique doit maintenir une certaine cohérence suite aux modifications provoquées par l'utilisateur. On peut aussi citer les problèmes d'emplois du temps (lorsqu'un professeur a un empêchement, un nouvel emploi du temps est reconstruit), les problèmes d'ordonnancement en ligne (lorsqu'une machine tombe en panne, il faut ré-ordonnancer), les problèmes de CAO ... De tels problèmes sont qualifiés de dynamiques[1].

Les évolutions d'un problème dynamique doivent être prises en compte très rapidement pour refléter l'état réel du système. Traiter un système dynamique de contraintes consiste donc à résoudre successivement une suite de problèmes (statiques) résultant des évolutions. Les systèmes commerciaux actuels de programmation par contraintes ne sont pas adaptés au traitement de ces problèmes dynamiques. En effet, toute modification conduit à la définition d'un nouveau problème qui est traité par réexécution (sans tenir compte de la résolution du problème précédent). Lorsque le problème à résoudre est particulièrement difficile, ou lorsque de nombreuses modifications interviennent, une telle réexécution devient très vite rédhibitoire.

Une façon élégante (introduite par Fages *et al.* [1995]) d'aborder la dynamicité des problèmes est de les considérer sous l'angle des systèmes réactifs [Harel et Pnueli, 1985]. De tels systèmes n'ont pas pour but de produire une solution à partir de données en entrée (systèmes transformationnels) mais plutôt de maintenir une relation avec l'environnement. Un système réactif, en relation permanente avec l'environnement, est capable de réagir à des stimuli externes (ici ajout/retrait de

1. La notion de problème dynamique a été introduite sur les CSP (problèmes de satisfaction de contraintes) par [Dechter et Dechter, 1988].

contraintes) et de maintenir une certaine propriété au cours de la résolution. Cette approche nous paraît parfaitement adaptée pour prendre en compte la dynamicité des systèmes de contraintes.

Enfin, la prise en compte de modifications successives du système de contraintes dans un problème dynamique peut conduire à un système de contraintes contradictoire. Il est alors nécessaire de relaxer manuellement ou automatiquement certaines contraintes.

1.2 Problématique

Dans le traitement des systèmes dynamiques de contraintes, le maître mot est l'incrémentalité. Il vaut éviter le phénomène de «*thrashing*»: le recalcul d'informations déjà découvertes[2]. Les travaux dans ce domaine traitent majoritairement des CSP (*Constraint Satisfaction Problems* – domaines finis[3]). Ces travaux se fondent principalement sur un mécanisme d'«*apprentissage*[4]» que cela soit par l'échec ([Schiex et Verfaillie, 1994; Verfaillie et Schiex, 1994; Sannella, 1993]), ou par enregistrement de calculs passés ([Bessière, 1991; Debruyne, 1995]).

Dans un problème dynamique, il existe un aspect important qui n'a jamais été pris en compte. Lorsque les modifications successives sur le système de contraintes d'un problème dynamique conduisent à une contradiction[5] (on parle alors de problème **sur-contraint**), les systèmes usuels ne peuvent que répondre qu'il n'existe pas de solution. Cette réponse n'est pas satisfaisante. Il peut arriver qu'il soit indispensable[6] de fournir une solution (ou de rétablir la propriété) quitte à ne pas respecter certaines contraintes. On parle alors de **relaxation de contraintes**.

Les méthodes proposées pour résoudre les problèmes sur-contraints ([Freuder, 1989; Wilson et Borning, 1993; Fowler, 1993; Jampel *et al.*, 1996]) traitent convenablement les systèmes statiques mais ne nous paraissent que peu adaptées à une utilisation dans un cadre dynamique. En effet, ces méthodes supposent que :

- d'une part, le système de contraintes est complètement connu avant la recherche de solution ;
- d'autre part, les préférences associées aux contraintes du problème sont toutes connues au moment de la résolution.

La première hypothèse n'est pas totalement incompatible avec un système dynamique. En effet, lorsqu'une contradiction est identifiée, le système de contraintes en cause est complètement connu. Le processus de recherche de solution relaxée ne sera donc pas perturbé par une modification du système de contraintes. Mais, si plusieurs contradictions se produisent successivement, le passé (informations recueillies au cours de la résolution des contradictions précédentes) ne peut être utilisé pour «*apprendre*». De plus le coût des résolutions de problèmes très proches indépendamment les unes des autres devient vite prohibitif.

2. Par exemple, ce recalcul partiel se produit lorsqu'on recommence une résolution en ne modifiant que quelques hypothèses (contrainte ajoutée ou retirée), ou lorsqu'on réalise un backtrack et que l'on revient dans une portion de l'espace de recherche déjà explorée auparavant.

3. Nous utiliserons indifféremment les termes domaines finis et CSP selon le contexte du discours. Si nous nous trouvons dans un cadre PLC nous parlerons plus de «*domaines finis*». Au contraire, si nous nous trouvons dans un cadre général, nous utiliserons plus «CSP».

4. Il ne s'agit pas ici d'apprentissage au sens IA du terme mais plutôt de la capacité à enregistrer des informations pour soit conserver une trace des calculs effectués, soit ne pas retomber sur des échecs passés.

5. Il n'existe pas de solution ou la propriété maintenue n'est pas vérifiée par le système de contraintes courant.

6. C'est le cas pour un ordonnancement de production ou encore pour les emplois du temps. Nous avons rencontré une telle situation [Boizumault *et al.*, 1994; Guéret *et al.*, 1996] et c'est d'ailleurs ce qui nous a mené à ces travaux.

La deuxième hypothèse est plus gênante. En effet, il paraît illusoire de demander à l'utilisateur de pouvoir *classer* toutes les contraintes d'un problème donné : d'une part, le nombre de ces contraintes peut être très grand et d'autre part, ce genre de *classement* étant nécessairement subjectif il peut dépendre fortement du contexte où il se fait.

À notre connaissance, il n'existe pas de système permettant de lever les deux hypothèses citées précédemment. Il n'est donc pas possible de traiter de manière efficace et incrémentale des problèmes devenus sur-contraints dans un environnement dynamique.

1.3 Notre thèse

Nous avons tout d'abord pris conscience de l'existence d'un besoin réel de systèmes capables de traiter les problèmes dynamiques. Puis nous avons constaté l'absence de systèmes capables de prendre en compte les différentes problématiques spécifiques à la résolution des problèmes dynamiques. Notre travail est une contributation au traitement des problèmes sur-contraints survenant dans un cadre dynamique. Pour cela, nous proposons un cadre général pour la relaxation de contraintes dans un environnement dynamique.

Nous proposons le cadre **relax**($\mathcal{D}$, $\mathcal{C}$, $\mathcal{P}$) permettant de maintenir une propriété $\mathcal{P}$ pour un ensemble de contraintes $\mathcal{C}$ sur le domaine $\mathcal{D}$ dans un environnement dynamique. Nous proposons ainsi un schéma algorithmique permettant de traiter de manière totalement incrémentale retrait et ajout de contraintes même lorsque le problème devient sur-contraint. Nous portons une attention toute particulière à ce dernier cas.

Notre approche permet de déterminer une configuration (partition des contraintes d'un problème entre contraintes actives et contraintes relaxées) vérifiant la propriété $\mathcal{P}$ et satisfaisant au mieux les préférences de l'utilisateur sur les contraintes du problème. Cette recherche est réalisée à l'aide d'une exploration en meilleur d'abord de l'espace des configurations.

Nous présentons trois instances de ce schéma général : une instance concernant les contraintes linéaires sur les rationnels avec maintien de la réalisabilité du système de contraintes et deux instances concernant les classes de problèmes résolus par des algorithmes de réduction (les domaines finis et les intervalles) avec maintien de cohérence partielle voire de solution (uniquement pour les domaines finis [7]). Pour les deux dernières instances, nous montrons comment un mécanisme d'« *apprentissage* » [8] peut être utilisé pour résoudre un problème sur-contraint et capturer la dimension dynamique du problème traité.

Nous validons notre proposition par des résultats expérimentaux, tant sur des problèmes aléatoires que sur des problèmes réels. Nous montrons de plus comment les idées développées dans cette thèse peuvent être utilisées pour d'autres problématiques comme, notamment, le traitement de la disjonction. Nous utilisons pour cela une nouvelle contrainte (la contrainte `one-of`) qui utilise pleinement les possibilités de la recherche en meilleur d'abord que nous proposons pour la relaxation de contraintes.

7. Le concept de solution en tant qu'instanciation de variables n'a pas de sens sur les intervalles.
8. Comme précédemment, il ne s'agit pas de l'apprentissage au sens IA du terme mais d'un mécanisme d'enregistrement. Ici, nous utilisons un système de maintien de déduction.

1.4 Contributions et organisation de la thèse

Dans cette thèse, nous montrons :

1. Comment un système de relaxation de contraintes peut être vu comme une recherche en meilleur d'abord dans l'espace des configurations. Nous proposons un cadre général pour la relaxation paramétré par le domaine $\mathcal{D}$ du calcul, un ensemble $\mathcal{C}$ de contraintes et une propriété $\mathcal{P}$ à vérifier.

2. Comment une instance directement déduite pour $\mathcal{D} = \mathbb{Q}$ et pour des contraintes linéaires peut être obtenue.

3. Comment une utilisation raisonnée d'un système de maintien de déductions (Deduction Maintenance System - DMS) permet de présenter une instance efficace et réaliste du schéma général pour des domaines de calcul résolus par réduction de domaines (domaines finis et intervalles). Nous proposons de plus différentes implémentations permettant de proposer un laboratoire d'études des CSP sur-contraints.

4. Comment un système de relaxation de contraintes peut être utilisé pour d'autres problématiques : nous proposons une nouvelle technique de traitement de la disjonction basée sur notre recherche en meilleur d'abord. Nous donnons ainsi des premiers résultats qui nous paraissent ouvrir de nombreuses perspectives.

Plus précisément, la suite de ce document est organisée de la façon suivante. Dans une **première partie** (chapitres 2 et 3), nous présentons le contexte général de notre étude. Nous introduisons chapitre 2 un certain nombre de définitions de base indispensables lorsque l'on parle de relaxation de contraintes. Nous présentons ensuite un état de l'art dans le domaine de la relaxation et du traitement (lié à la relaxation) des systèmes dynamiques de contraintes (chapitre 3).

Dans une **deuxième partie** (chapitres 4 à 9), nous présentons notre système de relaxation de contraintes depuis une vision abstraite jusqu'à l'implémentation. Nous proposons 4 niveaux d'abstraction. Le premier niveau (chapitre 4) présente le cadre général de notre méthode. Le deuxième niveau (chapitre 6) décrit l'utilisation d'un système de maintien de déduction lors d'une utilisation de notre approche avec un solveur procédant par réduction de domaine. Le troisième niveau (chapitres 5, 7 et 8) décrit les instances de ce cadre pour traiter, respectivement, les contraintes linéaires sur les rationnels, les contraintes sur les domaines finis et les systèmes de contraintes résolus par propagation d'intervalle. Enfin, le quatrième niveau (chapitre 9) traite de l'implémentation de notre approche sur les domaines finis.

La deuxième partie s'organise de la façon suivante : nous présentons dans le chapitre 4 le cadre **relax($\mathcal{D}$, $\mathcal{C}$,$\mathcal{P}$)** un cadre général pour la relaxation de contrainte. Ce cadre général est caractérisé par une méthode de recherche en meilleur d'abord paramétrée par trois fonctions :

- `explique-contradiction` qui permet, à partir d'une configuration ne vérifiant pas la propriété $\mathcal{P}$, de déterminer un sous-ensemble de contraintes dont la conjonction provoque à coup sûr une contradiction. C'est ce qu'on appelle une *explication de contradiction* ;

- `meilleure-configuration` qui permet, en tenant compte des préférences données par l'utilisateur, de déterminer la meilleure configuration[9] ne contenant aucune *explication de contradiction* connue. La détermination de cette

9. Cette meilleure configuration n'est pas nécessairement unique, mais par simplification nous nous référerons à ces configurations optimales de manière unique dans la suite de ce document.

meilleure configuration nécessite la donnée par l'utilisateur d'un comparateur de configurations ;

- `maj-configuration` permet de passer d'une configuration de départ donnée à une configuration d'arrivée (elle aussi donnée) en utilisant les informations recueillies jusqu'à présent pour réaliser les mises à jour nécessaires.

Toute instance de notre cadre général doit préciser ces trois fonctions afin de donner les modalités de réalisation des spécifications annoncées. Nous présentons une instance directement déduite de ce cadre pour le traitement des contraintes linéaires sur les rationnels dans le chapitre 5.

Nous nous attachons ensuite (chapitre 6) à une spécialisation de notre méthode pour les systèmes de contraintes résolus par réduction de domaines (les CSP et les CSP numériques). Nous utilisons pour cela un système de maintien de déduction (DMS) qui permet d'enregistrer le *savoir-faire* du solveur utilisé et ainsi de garder une trace des responsabilités quant aux retraits de valeurs dans les domaines des variables. Nous présentons une utilisation raisonnée du DMS permettant d'éviter une consommation prohibitive d'espace mémoire.

Dans les chapitres 7 et 8, nous décrivons deux instances de cette spécialisation. Ainsi, dans le chapitre 7, nous montrons comment notre système de maintien de déduction peut être utilisé pour mettre en œuvre notre cadre général de relaxation sur les CSP. L'énumération est alors traitée comme un cas particulier d'ajout/retrait de contraintes d'égalité entre variable et valeur. Cette vision «*dynamique*» de l'énumération permet un traitement uniforme de toutes les contraintes. De plus, nous détaillons dans ce chapitre les complexités tant spatiale que temporelle associées à notre approche. Nous montrons ainsi que l'on peut réaliser une implémentation d'autant plus efficace de ce système que le comparateur utilisé a de bonnes propriétés.

Dans le chapitre 8 nous montrons comment un système de maintien de déduction peut être simplifié pour proposer une instance allégée de notre approche dans le but de traiter les CSP numériques à l'aide de la B-consistance d'arc [Lhomme, 1994].

Enfin, nous présentons les principes généraux d'une mise en œuvre sur les CSP dans le chapitre 9. En particulier, nous décrivons l'architecture générale de toutes nos implémentations désignées par le terme générique DECORUM (Deduction-based Constraint Relaxation Management). Nous détaillons quelques aspects d'une implémentation réalisée en langage CLAIRE [10].

Dans une **troisième partie** (chapitres 10 et 11), nous présentons nos premiers résultats expérimentaux. Dans le chapitre 10, nous étudions le comportement de DECORUM sur des instances aléatoires de CSP. En particulier, nous montrons que l'on retrouve des résultats de transition de phase [Prosser, 1994b] même dans le cas de la recherche de solutions relaxées pour des problèmes sur-contraints et nous montrons que notre approche se comporte de manière favorable par rapport à l'algorithme MAC [Sabin et Freuder, 1994] sur des problèmes «*structurés*».

Nous montrons (chapitre 11) l'intérêt de notre méthode de recherche en meilleur d'abord pour le traitement des contraintes disjonctives. Nous proposons ainsi une nouvelle contrainte : la contrainte `one-of`. Cette contrainte utilise notre méthode de recherche et ses propriétés pour «*arbitrer*» des disjonctions. En particulier, nous nous sommes intéressés à la classe des problèmes d'ordonnancement appelés Open-Shop. Ces problèmes ont la particularité d'être purement disjonctifs. Par ailleurs, Christelle Guéret, en intégrant notre approche dans ses travaux sur l'Open-Shop a résolu pour la première fois un problème ouvert dans ce domaine [Guéret et Jussien, 1997].

10. Il existe actuellement trois implémentations du système : en MIT SCHEME, en CLAIRE et en C++.

Enfin, dans une **quatrième partie** (chapitre 12), nous dressons une conclusion et ouvrons des perspectives.

Schéma de Lecture

Ce document peut être lu de manière non linéaire. En effet, le cœur de cette thèse (chapitres 4 à 9) peut être abordé de différentes façons :

- Le lecteur intéressé uniquement par les domaines finis ou les CSP peut se contenter des chapitres 4, 6 et 7. Les implémentations étant détaillées chapitre 9.
- Le lecteur intéressé uniquement par les applications aux intervalles peut se contenter des chapitres 4, 6 et 8. Une implémentation sur les intervalles obéissant aux mêmes principes que pour un CSP, le chapitre 9 peut être considéré comme décrivant une implémentation dans ce cadre.
- Le lecteur intéressé uniquement par les contraintes linéaires sur les rationnels peut se contenter tout simplement des chapitres 4 et 5. Ce dernier chapitre montre comment en reprenant différents travaux de la littérature on obtient très facilement une instance de notre schéma général.

Première partie

Contexte de l'étude

Chapitre 2

Définitions

Sommaire

DANS CE CHAPITRE, nous présentons les notions de base utilisées tout au long de cette thèse. Nous définissons ainsi la notion de système de contraintes, puis nous introduisons les notions relatives à la dynamicité. Nous introduisons ensuite des notions de préférence sur les contraintes permettant de définir une solution pour un système de contraintes contradictoire. Enfin, nous fixons précisément le cadre de notre travail.

2.1 Problèmes de Satisfaction de Contraintes

2.1.1 Système de contraintes

On se donne un **domaine de calcul** $\mathcal{D}$. Il s'agit d'un ensemble quelconque (fini ou infini, dénombrable ou indénombrable) de valeurs.

> **Exemple 1 (Domaine de calcul) :**
> Si les valeurs que l'on considère sont discrètes, on peut alors prendre $\mathcal{D} = \mathbb{N}$.
> Si les valeurs considérées sont continues, on peut prendre $\mathcal{D} = \mathbb{R}$ ou encore $\mathcal{D} = \mathbb{Q}$. Les valeurs peuvent très bien être symboliques, ainsi on peut avoir $\mathcal{D} = \{\text{passable}, \text{assez bien}, \text{bien}, \text{très bien}\}$.

On se donne un ensemble $\mathcal{V} = \{v_1, \ldots, v_n\}$ de n **variables** prenant chacune leurs valeurs dans l'ensemble $\mathcal{D}$. À chaque variable v de $\mathcal{V}$ est associé un ensemble D_v appelé **domaine de** v qui contient l'ensemble fini ou infini des valeurs possibles pour la variable, *i.e.* $D_v \subset \mathcal{D}$. Le domaine d'une variable peut évoluer au cours de la résolution. On emploie alors les termes *domaine originel* et *domaine courant*. On note le domaine originel d'une variable D_v^{orig}.

Par souci de généralité nous allons étendre la définition usuelle de l'instanciation[1]. On appellera **instanciation** d'une variable le fait de lui attribuer un sous-ensemble de son domaine. En particulier, ce sous-ensemble peut être un singleton et on retrouve alors la notion habituelle d'instanciation.

On se donne un ensemble C de e **contraintes** portant sur des éléments de V. Une contrainte est une relation devant être vérifiée par l'instanciation des variables concernées.

> **Exemple 2 (Contrainte) :**
> Soient deux variables x et y. $x^2 = y + 1$ est alors une contrainte. Si x vaut 2 et y vaut 3, la contrainte est vérifiée.
> $x > y + 1$ est aussi une contrainte. Si $x > 2$ et $y < 0$ la contrainte est vérifiée. C'est un exemple d'instanciation par un ensemble de valeurs.
> Soient deux variables x et y, $c : \{(1,2),(2,4),(4,8)\}$ est aussi une contrainte. On dit qu'elle est exprimée en extension. Soit $(a,b) \in c$ (dans le cas d'une contrainte exprimée en extension, on identifie la vérification d'une contrainte et l'appartenance à la relation – l'ensemble – la définissant). Si x vaut a et y vaut b alors la contrainte est vérifiée.

On note $var(c)$ l'ensemble ordonné des variables concernées par la contrainte c. Le cardinal de $var(c)$ est appelé l'**arité** de la contrainte c.

Définition 1 (Système de Contraintes)
On appelle ***système de contraintes*** *la donnée d'un domaine de calcul $\mathcal{D}$, d'un ensemble $\mathcal{V}$ de variables, de l'ensemble D de leur domaine associé et d'un ensemble $\mathcal{C}$ de contraintes sur $\mathcal{V}$.*

1. Usuellement, il s'agit d'affecter une valeur unique à une variable. Ici, pour pouvoir capturer différents domaines comme, par exemple, les intervalles, nous étendons cette définition. Dans le reste du document, nous utiliserons donc cette définition de l'instanciation mais, la plupart du temps, l'acception usuelle sera respectée.

Définition 2 (Solution)
*Une **solution** pour un système de contraintes est la sélection d'un sous-ensemble du domaine (instanciation) de chaque variable de telle sorte que toutes les contraintes soient vérifiées.*

2.1.2 Satisfaisabilité d'un système de contraintes

On se donne une propriété $\mathcal{P}$ définie sur l'ensemble de contraintes et sur les domaines des variables concernées.

Définition 3 (Propriété)
Une propriété représente un degré de consistance devant être respecté par le système de contraintes : il s'agit d'une condition nécessaire pour la consistance du système de contraintes considéré. Ainsi, si celui-ci n'est pas consistant, la propriété $\mathcal{P}$ n'est pas vérifiée.

> **Exemple 3 (Propriété) :**
>
> Un CSP [Tsang, 1993] est un système de contraintes dont le domaine de calcul $\mathcal{D}$ est l'ensemble des entiers (ou tout ensemble dénombrable) et dont le domaine pour une variable est un ensemble fini de valeurs. Pour de tels problèmes, on peut ainsi considérer comme propriété, la k-consistance [Freuder, 1978], la consistance de pivot [David, 1994] ou bien encore la consistance globale. Sur les CSP numériques (intervalles), on pourra considérer la $2B$-consistance [Lhomme, 1993].

On définit alors la notion de satisfaisabilité d'un système de contraintes suivant une propriété $\mathcal{P}$.

Définition 4 ($\mathcal{P}$-satisfaisabilité d'un système de contraintes)
*Un système de contraintes est dit **$\mathcal{P}$-satisfaisable** si et seulement si la propriété $\mathcal{P}$ est vérifiée pour les contraintes du problème et les domaines courants des variables concernées.*

Rappelons quelques définitions de propriétés connues.

Définition 5 (k-consistance – [Freuder, 1978])
*Un CSP est **k-consistant** si et seulement si pour toute instanciation consistante de $k-1$ variables, il existe dans le domaine de toute variable non instanciée une valeur prolongeant cette instanciation en une instanciation consistante de k variables.*

Définition 6 (B-consistance d'arc)
*Soient $\mathcal{P}$ un CSP numérique, v une variable de $\mathcal{P}$ et $D_v = [a, b]$. D_v est **B-consistant d'arc** si et seulement si ces deux bornes a et b vérifient la 2-consistance. Un CSP est **B-consistant d'arc** si et seulement si tous ses domaines sont B-consistants d'arc.*

Nous utiliserons par la suite diverses propriétés. Nous proposons ici une notation pour les trois principales.

Définition 7 (Propriétés – Notations)
- *Il existe une instanciation consistante des variables avec le système courant de contraintes. Nous noterons cette propriété : $\mathcal{P}_{sol}$.*
- *Le filtrage par arc-consistance (2-consistance) ne vide aucun domaine. Nous noterons cette propriété : $\mathcal{P}_{ac}$.*
- *Le filtrage par consistance aux bornes ne vide aucun domaine. Nous noterons cette propriété : $\mathcal{P}_{bc}$.*

2.1.3 Instances du schéma général

Considérons quelques exemples d'instances usuelles du schéma général que nous venons d'introduire.

Exemple 4 (CSP) :
Pour un CSP, la propriété $\mathcal{P}$ à vérifier est bien souvent l'arc-consistance ($\mathcal{P}_{ac}$). Notons que dans le cas où toutes les variables du problème considéré sont instanciées, l'arc-consistance implique la consistance globale. Dans ce cas, la seule vérification de la propriété $\mathcal{P}_{ac}$ suffit à assurer l'existence d'une instanciation consistante (choix pour chaque variable d'une valeur de son domaine telle que toutes les contraintes soient vérifiées).

Exemple 5 (Problèmes linéaires) :
Lorsque le domaine de calcul est l'ensemble $\mathbb{Q}$ des rationnels et les contraintes sont linéaires, la propriété usuelle est la consistance globale ($\mathcal{P}_{sol}$). Les solveurs utilisés sont alors des algorithmes type simplexe et/ou Gauss (pivot) [Colmerauer, 1990].

Exemple 6 (CSP numériques) :
Dans le domaine des grandeurs physiques où on s'intéresse plus aux domaines de variation des variables plutôt qu'à leur valeur exacte, on utilise l'arithmétique et les techniques de réduction sur les intervalles. Le domaine de calcul est alors aussi $\mathbb{R}$ mais les contraintes définies sont le plus souvent non linéaires. La propriété $\mathcal{P}$ utilisée dans ce cas est souvent la *B-consistance d'arc* [Lhomme, 1993] (*cf.* définition 60 page 107) ou la *box-consistency* [Benhamou *et al.*, 1994].

2.2 Systèmes dynamiques de contraintes

2.2.1 Problèmes dynamiques

Comme nous l'avons rappelé en introduction, de nombreux problèmes réels se modélisent aisément sous forme de contraintes. Mais, bien souvent, ces problèmes sont amenés à évoluer au cours de la résolution. Par exemple, une interface graphique doit maintenir une certaine cohérence suite aux modifications provoquées par l'utilisateur [Freeman-Benson *et al.*, 1990]. On peut aussi citer les problèmes d'emplois du temps (un professeur a un empêchement, ...), les problèmes d'ordonnancement de chaînes de montage (une machine tombe en panne, ...), les problèmes de CAO, ...

Ces modifications doivent être prises en compte immédiatement pour refléter l'état réel du système. L'ensemble des contraintes peut ainsi évoluer très rapidement. En fait, on peut trouver différents types d'interventions : au cours de la résolution (on parle alors de réactivité car il s'agit plutôt de répondre à des stimuli externes[2] plutôt que de fournir une solution) et après la résolution (on parle alors plus d'incrémentalité).

Pour formaliser ces interventions externes sur le système de contraintes, nous introduisons la notion de configuration.

Définition 8 (Configuration)
Une ***configuration*** $\langle A, R\rangle$ *est une partition des contraintes du problème en deux sous-ensembles complémentaires: l'ensemble R des contraintes* «inactives» *(non prises en compte dans le système courant de contraintes) et l'ensemble A des contraintes actives (appartenant au système de contraintes).*

2. Ajout/retrait de contraintes.

À partir de cette notion de configuration, on peut exprimer simplement les opérations d'ajout et de retrait de contraintes.

Définition 9 (Ajout/retrait de contraintes)
Soit un problème dont la configuration courante est $\langle A, R \rangle$.

- *L'ajout d'une contrainte* c *consiste à passer de la configuration* $\langle A, R \rangle$ *à la configuration* $\langle A \cup \{c\}, R \setminus \{c\} \rangle$ *si* $c \in R$ *et à la configuration* $\langle A \cup \{c\}, R \rangle$ *sinon.*

- *Le retrait d'une contrainte* c *consiste à passer de la configuration* $\langle A, R \rangle$ *à la configuration* $\langle A \setminus \{c\}, R \cup \{c\} \rangle$.

On peut maintenant définir un problème dynamique.

Définition 10 (Problème dynamique)
Par analogie avec la définition d'un CSP *dynamique [Dechter et Dechter, 1988], nous définissons un problème dynamique comme étant une suite de problèmes différant l'un de l'autre par l'ajout ou le retrait d'une contrainte.*

C'est donc une séquence de configurations déduites les unes des autres par les opérations présentées dans la définition 9.

Pour «*résoudre*»[3] un problème dynamique, une simple réexécution paraît difficilement envisageable. En effet, la plupart des problèmes réels sont des problèmes difficiles et une réexécution peut être rédhibitoire[4].

C'est pourquoi il paraît plus raisonnable de mettre à profit l'information accumulée au cours de la résolution du problème précédent pour prendre en compte une modification du système de contraintes. On peut identifier deux avantages supplémentaires à cette approche :

- la solution que l'on obtient a alors plus de chances d'être proche de la solution originelle. En effet, *a priori* seules des modifications *utiles* seront réalisées sur la solution courante permettant de la modifier le moins possible ;

- les parties de la solution précédente qui n'ont pas lieu d'être modifiées ne sont pas recalculées inutilement. Le phénomène de «*thrashing*» est alors évité.

2.2.2 Systèmes réactifs

Nous considérons les systèmes servant à résoudre des problèmes dynamiques sous l'angle de la programmation réactive [Harel et Pnueli, 1985; Fages *et al.*, 1995]. Dans une telle optique, un problème n'est jamais résolu complètement puisque ses spécifications peuvent changer à tout moment. Il est ainsi naturel de chercher à maintenir une propriété donnée.

Notre notion de $\mathcal{P}$-satisfaisabilité pour un système de contraintes s'inscrit alors parfaitement dans ce cadre. En effet, on ne cherche alors plus une instanciation vérifiant les contraintes (solution au sens de la définition 2 page 17) mais plutôt à maintenir la propriété $\mathcal{P}$ quelles que soient les modifications apportées par l'environnement extérieur.

3. On entend ici par résoudre: le maintien de la propriété.

4. En termes de temps de calcul une telle option peut rapidement devenir prohibitive. D'autre part, la dimension dynamique du problème traité est complètement perdue dans ce cas.

2.2.3 Problèmes sur-contraints

Dans un système réactif, les contraintes arrivent indépendamment les unes des autres et peuvent être créées par des événements totalement indépendants. Bien souvent le système de contraintes devient **sur-contraint**.

Définition 11 (Problème sur-contraint)
Un problème est dit ***sur-contraint*** *pour une propriété $\mathcal{P}$ si le système de contraintes le définissant ne vérifie pas la propriété $\mathcal{P}$.*

Lorsqu'il est confronté à un problème sur-contraint, un solveur classique répond à l'utilisateur qu'il n'existe pas de solution. Cette réponse est vraiment très pauvre et souvent préjudiciable. Par exemple, si dans le cadre d'un ordonnancement dynamique, une nouvelle tâche est à prendre en compte et qu'aucune solution n'existe, il n'est pas acceptable que le système de résolution réponde simplement : il n'existe pas de solution. Il est plus intéressant d'être capable, par exemple, de dire quelle contrainte ne doit pas être prise en compte pour permettre l'existence d'une solution.

Lorsque certaines contraintes d'un problème ne sont pas prises en compte dans une solution proposée à l'utilisateur, on parle de **relaxation** de contrainte. Il s'agit d'un retrait de la contrainte considérée au sens de la définition 9. D'un point de vue opérationnel, tous les effets passés de la contrainte relaxée doivent être supprimés.

Dans la plupart des cas, on ne peut pas simplement relaxer la dernière contrainte introduite (celle qui a provoqué la contradiction). Dans une application réelle, une telle contrainte ajoutée au dernier moment est souvent un impératif que l'on vient de découvrir et qui ne peut être écarté. Dans d'autres applications, comme par exemple la CAO, l'ajout de contraintes peut correspondre à une *simulation* permettant de voir évoluer le résultat en fonction de nouvelles contraintes : une telle contrainte ne peut donc être simplement ignorée.

Il faut donc être capable de connaître l'avis de l'utilisateur sur les contraintes ajoutées. C'est pourquoi, on peut introduire des préférences sur les contraintes pour pouvoir faire des choix de relaxation pertinents du point de vue de l'utilisateur. C'est l'utilisateur qui définit ses préférences. Le système de résolution se chargera, lui, de déterminer les contraintes à relaxer pour rétablir la $\mathcal{P}$-satisfaisabilité.

2.3 Prise en compte de préférences

2.3.1 Préférence et hiérarchie

À chaque contrainte du problème est associée une **préférence** [Wilson et Borning, 1993] fournie par l'utilisateur et qui qualifie son souhait de la voir vérifiée. Ainsi, il y a des contraintes qui sont requises (par exemple, dans un problème d'emploi du temps, un professeur ne peut faire cours dans deux endroits au même moment) et il y a des contraintes plus ou moins importantes (par exemple, pas plus de 6 heures de cours dans une même journée).

Cette préférence peut s'exprimer de manière symbolique (la préférence est un élément d'un ensemble prédéfini de symboles) ou numérique (on peut alors parler de poids ou de force d'une contrainte). L'ensemble des valeurs pouvant être prises par les préférences est muni d'une structure d'ordre et éventuellement d'un opérateur d'agrégation. Plus formellement :

Définition 12 (Préférence)
Soit un ensemble P_{ref} ordonné. Soit une fonction $p : \mathcal{C} \to P_{ref}$. Pour une contrainte $c \in \mathcal{C}$, $p(c)$ est appelé la ***préférence*** *associée à c.*

On peut définir un opérateur $\bigoplus$ *permettant d'étendre* p *aux ensembles de contraintes de la façon suivante :*

$$\forall C \in \mathcal{C}, \quad p(C) = \bigoplus_{c \in C} p(c)$$

Le regroupement des contraintes par niveau de préférence définit ce que Wilson et Borning [1993] appellent une **hiérarchie** de contraintes.

2.3.2 Solution pour une hiérarchie

Une H-**solution**, pour un problème et une hiérarchie donnés, est une instanciation de chacune des variables apparaissant dans les contraintes de la hiérarchie. Une telle *solution* respecte plus ou moins les contraintes du problème. Notons que dans les travaux de Wilson et Borning [1993], ne sont considérées que les H-solutions vérifiant les contraintes requises[5].

Il existe différentes façons d'apprécier la vérification d'une contrainte. On peut prendre une approche simple (c'est celle que nous conserverons tout au long de ce travail) et dire qu'une contrainte est vérifiée ou qu'elle ne l'est pas. On peut aussi définir une notion de distance entre l'état courant des domaines des variables et l'idéalité que représente la vérification de la contrainte permettant ainsi de formaliser une notion de contrainte plus ou moins vérifiée.

2.3.3 Préférences et Interactivité

Il n'est pas toujours possible de connaître *a priori* toutes les préférences que l'utilisateur accorde aux contraintes du problème. En effet, il peut y avoir un très grand nombre de contraintes ou bien une préférence peut dépendre totalement du contexte de résolution. La préférence accordée à une contrainte peut ainsi être modifiée au cours du temps[6].

C'est pourquoi, un système interactif[7], ne faisant appel à l'utilisateur qu'en cas de nécessité, a son intérêt. Ainsi, il serait possible de demander à l'utilisateur de définir ses préférences sur de petits sous-ensembles de contraintes[8] ce qui paraît plus raisonnable que de toutes les demander avant de démarrer la résolution ou à chaque fois qu'une nouvelle contrainte est introduite.

2.4 Résolution d'un système de contraintes sur-contraint

La résolution d'un système de contraintes sur-contraint suppose la recherche d'une configuration vérifiant «*au mieux*» la propriété $\mathcal{P}$.

2.4.1 Configuration et propriété maintenue

À une H-solution correspond une configuration. Pour un système de contraintes, nous nous intéresserons en fait plus au concept de configuration qu'au concept de

5. Nous revenons sur ces travaux dans le chapitre 3.

6. On peut considérer le cas d'une contrainte donnée comme peu importante au début de la résolution pour se rendre compte en cours de résolution qu'elle doit être impérativement vérifiée.

7. Résolution réalisée en lien permanent avec l'utilisateur.

8. En particulier, des sous-ensembles provoquant une contradiction. Cela permet ainsi de circonscrire la demande d'information aux contraintes pertinentes.

H-solution de la même manière que nous nous intéressons plus à la vérification d'une propriété $\mathcal{P}$ donnée plutôt qu'àu calcul d'une solution au sens usuel du terme.

Définition 13 ($\mathcal{P}$-satisfaisabilité d'une configuration)
*Une configuration $\langle A, R \rangle$ est **$\mathcal{P}$-satisfaisable** si et seulement si la propriété $\mathcal{P}$ est vérifiée par l'ensemble A des contraintes actives de la configuration.*

> **Exemple 7 ($\mathcal{P}_{ac}$-satisfaisabilité) :**
> Pour un CSP donné, une configuration $\langle A, R \rangle$ sera $\mathcal{P}_{ac}$-satisfaisable si après filtrage par consistance d'arc des domaines des variables par les contraintes de A, aucun de ceux-ci n'est vide.

Une configuration qui n'est pas $\mathcal{P}$-satisfaisable est dite **$\mathcal{P}$-contradictoire**.

2.4.2 Comparateurs

En utilisant la hiérarchie de contraintes définie par les préférences de l'utilisateur, on peut vouloir comparer deux configurations pour préciser celle qui est préférée.

Nous étendons la notion de comparateur de solutions de Wilson et Borning [1993] à la notion de comparateur de configurations.

Définition 14 (Comparateur)
*Un **comparateur** C_{omp} est une relation d'ordre partiel sur les configurations. On notera $C_{omp}(C_1, C_2)$ si la configuration C_1 est préférée à la configuration C_2 selon le comparateur C_{omp}.*

On pourra faire correspondre un opérateur infixé à un comparateur donné pour faciliter la lecture des comparaisons.

Notons que si le comparateur n'est défini que sur l'une des deux parties de la configuration, une connaissance partielle des préférences suffit.

On a les propriétés suivantes :

- la configuration $\langle C, \varnothing \rangle$ pour un problème dont l'ensemble des contraintes est C est l'élément maximal pour tout comparateur C_{omp} *i.e.*

$$\forall C_{omp}, \forall \langle A, R \rangle, C_{omp}(\langle C, \varnothing \rangle, \langle A, R \rangle)$$

- l'élément minimal pour tout comparateur C_{omp} est bien évidemment la configuration $\langle \varnothing, C \rangle$ pour un problème dont l'ensemble des contraintes est C *i.e.*

$$\forall C_{omp}, \forall \langle A, R \rangle, C_{omp}(\langle A, R \rangle, \langle \varnothing, C \rangle)$$

Nous donnons trois exemples de comparateur : C_{LPB} et C_{GPB} définis par Wilson et Borning [1993] et C_{MM} un comparateur défini par Schiex [1992].

Pour ces exemples, nous considérerons que plus le poids est faible, plus la contrainte est importante[9]. Ainsi, les niveaux de la hiérarchie sont décroissants en importance. Pour un ensemble R de contraintes, on notera $R_{[\ell]}$ la restriction de R aux contraintes de niveau ℓ dans la hiérarchie. Soient $C_1 = \langle A_1, R_1 \rangle$ et $C_2 = \langle A_2, R_2 \rangle$ deux configurations.

9. Cette position sur le poids des contraintes est celle choisie par Borning *et al.*. Nous préférons utiliser une vision inverse qui rend bien la notion de poids. Ainsi, pour nous, plus le poids est faible, moins la contrainte est importante.

Exemple 8 (Comparateur LPB) :
C_{LPB} est un comparateur utilisé par Wilson et Borning [1993]. C'est le comparateur *locally-predicate-better*. Ce comparateur est défini de la façon suivante :

Définition 15 (Comparateur C_{LPB})

$$C_{\text{LPB}}(C_1, C_2) \equiv \left| \begin{array}{l} \exists k \geqslant 0 \text{ tel que} \\ \forall i < k, R_{1_{[i]}} = R_{2_{[i]}} \\ R_{1_{[k]}} \subsetneq R_{2_{[k]}} \end{array} \right.$$

Dans le cas de C_{LPB}, la comparaison se fait donc contrainte par contrainte.

Exemple 9 (Comparateur GPB) :
C_{GPB} est un comparateur utilisé par Wilson et Borning [1993]. C'est le comparateur *globally-predicate-better*. Ce comparateur est défini de la façon suivante :

Définition 16 (Comparateur C_{GPB})

$$C_{\text{GPB}}(C_1, C_2) \equiv \left| \begin{array}{l} \exists k \geqslant 0 \text{ tel que} \\ \forall i < k, |R_{1_{[i]}}| = |R_{2_{[i]}}| \\ |R_{1_{[k]}}| < |R_{2_{[k]}}| \end{array} \right.$$

La différence par rapport à C_{LPB} tient dans le fait que la comparaison entre les deux configurations ne se fait plus contrainte par contrainte mais niveau par niveau en considérant le nombre de contraintes relaxées par niveau.

Exemple 10 (Comparateur MM) :
Il s'agit d'un comparateur défini pour traiter les CSP possibilistes [Schiex, 1992] (MM pour *max-min*) et que nous utiliserons par la suite.

Définition 17 (Comparateur C_{MM})

$$C_{\text{MM}}(C_1, C_2) \equiv \left| \begin{array}{l} \exists k \geqslant 0, \text{ tel que} \\ \forall i \leqslant k, R_{1_{[i]}} = R_{2_{[i]}} = \varnothing \\ R_{1_{[k]}} = \varnothing \wedge R_{2_{[k]}} \neq \varnothing \end{array} \right.$$

Le critère de ce comparateur peut s'exprimer : la plus importante des contraintes relaxées dans C_1 est moins importante que la plus importante des contraintes relaxées dans C_2.

Proposition 1
C_{MM} est un comparateur.

▷ **Preuve :** Il s'agit bien d'une relation d'ordre partiel. ◁

Proposition 2

$$\forall C_1, C_2, C_{\text{MM}}(C_1, C_2) \Rightarrow C_{\text{LPB}}(C_1, C_2)$$
$$\forall C_1, C_2, C_{\text{MM}}(C_1, C_2) \Rightarrow C_{\text{GPB}}(C_1, C_2)$$

▷ **Preuve :** C'est une conséquence directe des définitions des trois comparateurs. ◁

2.4.3 Solution pour un problème sur-contraint

On peut maintenant définir une notion de solution pour un problème sur-contraint.

Définition 18 (C-solution)
Soit un comparateur C et une propriété $\mathcal{P}$ donnés. On appelle **$C_{\mathcal{P}}$-solution**[10] *d'un problème sur-contraint, une configuration $\mathcal{P}$-satisfaisable maximale pour le comparateur C.*

Dans la suite de cette thèse, nous nous intéressons donc à la recherche d'une $C_{\mathcal{P}}$-solution pour un problème donné.

2.5 Le cadre du travail : un système de relaxation de contraintes

Nous définissons un **système de relaxation de contraintes** comme un système capable de trouver une $C_{\mathcal{P}}$-solution à partir d'une configuration $\mathcal{P}$-contradictoire pour un problème donné.

Nous désirons lutter contre le «*thrashing*» en cas de relaxation de contraintes et nous nous proposons de ne pas supposer l'ensemble des préférences complètement défini lors de la mise en place des contraintes.

Définition 19 (Système de relaxation de contraintes)
Soit un ensemble $\mathcal{C}$ de contraintes. Soit une propriété $\mathcal{P}$ définie sur les contraintes de $\mathcal{C}$. Soit un comparateur C_{omp} donné.

*Soit C_1 une configuration $\mathcal{P}$-contradictoire. Un système de relaxation de contraintes (*SRC*) peut alors être défini comme un opérateur sur les configurations tel que :* SRC(C_1) *soit une configuration $\mathcal{P}$-satisfaisable, maximale au sens du comparateur C_{omp}* i.e. *une $C_{omp\,\mathcal{P}}$-solution.*

Dans la deuxième partie de ce mémoire, nous présentons un cadre général pour les systèmes de relaxation de contraintes dont nous étudierons trois instances : résolution de contraintes linéaires sur les rationnels, résolution de contraintes quelconques sur les domaines finis (CSP) et résolution de contraintes quelconques sur les réels par propagation d'intervalles («CSP *numériques*»).

10. *ou plus simplement C-solution lorsqu'il n'y a pas d'ambiguité sur $\mathcal{P}$.*

Chapitre 3

État de l'art

Sommaire

DANS CE CHAPITRE, nous présentons un état de l'art recouvrant diverses problématiques. Dans un premier temps, nous nous intéressons à la résolution des problèmes dynamiques. Ensuite, nous présentons une synthèse des travaux sur la relaxation de contraintes pour les problèmes statiques et nous concluons sur la relaxation de contraintes pour les systèmes dynamiques.

Nous supposerons le lecteur familiarisé avec les concepts généraux concernant la *Programmation Logique avec Contraintes* (PLC) et les *Constraint Satisfaction Problems* (CSP).

3.1 Systèmes de Maintien de Déduction

Le maître mot du traitement des problèmes dynamiques est l'*incrémentalité*. Le but est d'éviter autant que possible le phénomène de *thrashing*. Pour cela, la plupart des méthodes que nous allons discuter sont basées sur l'utilisation d'un mécanisme d'enregistrement : les systèmes de maintien de déduction (DMS).

La plupart des DMS sont basés sur les systèmes de maintien de vérité *Truth Maintenance Systems* (TMS). Ils ont été introduits par Doyle [1979]. Les TMS sont utilisés dans différentes situations [Forbus et de Kleer, 1993] :

- identifier les suppositions réalisées ou les faits introduits ayant conduit aux conclusions faites par un solveur de problèmes ;
- se sortir d'une impasse lors d'une recherche ;
- maintenir un cache des inférences réalisées pour éviter un phénomène de *thrashing* ;
- guider le backtrack pour éviter de partir dans des branches non productives (que l'on peut identifier grâce aux informations enregistrées) ;
- faire du raisonnement par défaut.

La *philosophie* TMS vise à partager le travail d'un solveur de problème en deux parties (*cf.* figure 3.1) : une partie inférentielle et le TMS à proprement parler qui prend en charge tout ce qui concerne la véracité des faits considérés et gère les hypothèses tenues.

Pendant la résolution, le moteur d'inférences et le TMS interagissent continuellement suivant un protocole bien défini. Chaque inférence réalisée est transmise au TMS en tant que justification. Ces justifications enregistrées par le TMS permettent alors de :

- générer des explications en les *traçant* ;
- *remonter* le système de justifications pour identifier la source d'une contradiction ou d'une inconsistance ;

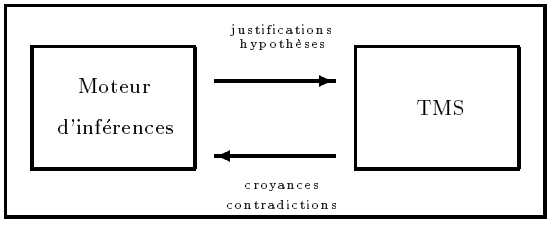

FIG. 3.1 – *Fonctionnement d'un* TMS

– contrôler la recherche de solution en tenant compte du passé.

Une chose difficile à réaliser avec les TMS est ce qu'on appelle le changement de contexte *i.e.* faire évoluer au moindre coût la véracité des déductions réalisées lorsqu'une ou plusieurs hypothèses sont remises en cause. C'est pourquoi ont été introduits les ATMS (*Assumption-based Truth Maintenance Systems*) [de Kleer, 1986a; de Kleer, 1986b; de Kleer, 1986c]. Dans un ATMS, on associe non seulement des justifications à des déductions mais on utilise aussi la notion d'*étiquettes* : l'ensemble des conjonctions d'hypothèses permettant de déduire une déduction donnée.

Le problème de ces approches est leur très grande consommation en espace puisque toutes les informations pouvant être tirées d'un problème donné sont conservées.

Citons les travaux de Roos [1994] qui proposent un mécanisme d'enregistrement d'informations dans le cadre d'une résolution et qui vont dans le sens de l'adaptation des techniques ATMS pour la résolution de CSP.

La plupart des systèmes que nous verrons dans la section suivante utilisent des versions simplifiées des TMS ou des ATMS. Ces simplifications permettent d'obtenir des systèmes d'enregistrement utilisables pour la résolution de problèmes conséquents. On parle alors plus de systèmes de maintien de déduction : ils permettent de conserver des *justifications* pour des informations déduites au cours de la résolution (échecs, retrait de valeur, ...).

3.2 «*Apprentissage*» pour les problèmes dynamiques

Pour produire un système efficace de résolution de problèmes dynamiques et en particulier pour être incrémental et éviter le *thrashing*, il faut être capable de déterminer des responsabilités pour les actions réalisées lors de la résolution (retrait de valeur dans le domaine d'une variable, suppression d'une portion de l'espace de recherche, ...). C'est ce que nous appelons ici «*apprentissage*».

3.2.1 Une première approche : les oracles

Van Hentenryck [1989] propose d'utiliser ce qu'il appelle un *oracle* pour réaliser un système incrémental de satisfaction de contraintes.

L'idée est de stocker lors de la résolution, le *chemin* emprunté pour faire la démonstration.

– Lorsqu'une contrainte est ajoutée, définissant ainsi un problème P_i déduit du problème précédent P_{i-1}, la méthode consiste à utiliser l'oracle calculé pour le problème P_{i-1} pour résoudre le problème P jusqu'à ce qu'une contradiction soit rencontrée. Ainsi, la résolution ne recommence pas au début évitant ainsi une partie du *thrashing*.

– Lorsqu'une contrainte est retirée, on utilise l'oracle ayant servi à résoudre le problème P_j (problème le plus récent ne contenant pas la contrainte retirée). On peut aussi utiliser un oracle *heuristique*, l'oracle du problème précédent mais des précautions doivent être prises car cet oracle peut laisser passer des solutions.

Cette approche ne propose pas une analyse fine lors de la réexécution. Il s'agit plus d'une démarche *heuristique*.

3.2.2 Maintien d'arc-consistance

Les algorithmes DNAC4 [Bessière, 1991] et DNAC6 [Debruyne, 1995] sont des extensions des algorithmes classiques d'arc-consistance AC4 [Mohr et Henderson, 1986] et AC6 [Bessière et Cordier, 1993; Bessière, 1994] qui permettent de gérer des systèmes dynamiques de contraintes.

Pour mener à bien un retrait efficace de contraintes, ces deux algorithmes enregistrent des informations au fur et à mesure de la résolution pour connaître les effets d'une contrainte. Ils utilisent la notion de *justification* : la contrainte qui à un moment donné est responsable du retrait d'une valeur dans le domaine d'une variable.

Ces informations sont alors utilisées pour déterminer quels sont les retraits de valeur qui dépendent directement ou indirectement (par propagation) de la contrainte retirée. Ensuite, de tels retraits étant susceptibles d'être réalisés (justifiés) d'une autre manière, il faut rétablir la cohérence.

Citons l'algorithme AC|DC proposé par Neveu et Berlandier [1994] qui n'utilise pas de système d'enregistrement et donc ne travaille que sur les contraintes du problème. Cette approche a été proposée pour réduire la complexité des autres algorithmes (DNAC4 et DNAC6). En fait, ce gain en espace se paye par une complexité en temps sensiblement augmentée.

Le tableau 3.1 donne les complexités spatiales et temporelles de ces trois algorithmes (n variables de domaine de taille maximum d sur lesquelles portent e contraintes). On trouve une comparaison expérimentale de ces trois approches dans [Debruyne, 1995].

	DNAC4	DNAC6	AC|DC
Complexité spatiale	$O(ed^2 + nd)$	$O(ed + nd)$	$O(e + nd)$
Complexité temporelle	$O(ed^2)$	$O(ed^2)$	$O(ed^3)$

TAB. 3.1 – *Complexités au pire pour des algorithmes d'arc-consistance dynamiques*

3.2.3 Diagnostic d'échec

Sans parler de relaxation de contraintes, il existe un certain nombre de travaux sur le diagnostic de système sur-contraints comme par exemple les techniques de backtrack intelligent [Sola, 1995; de Backer et Béringer, 1991b; Prosser, 1993] dans le cas général ou les travaux de Béringer et de Backer [1990] ou de Chinneck et Dravnieks [1991] sur les systèmes de contraintes linéaires sur les rationnels.

Dans un cadre général, Sola [1995] recommande de maintenir une structure permettant de définir la notion de *graphe de dépendance des liaisons* qui enregistre des informations liées à chaque unification réalisée dans un système de programmation logique. Ces informations permettent alors d'identifier avec précision des liaisons (unifications) responsables d'un échec et par là-même de réaliser un backtrack *intelligent*.

Dans d'autres cas [Holzbaur *et al.*, 1996], le solveur lui-même est capable de fournir les explications nécessaires à un backtrack intelligent. C'est le cas lorsqu'on utilise un algorithme de type simplexe pour résoudre des systèmes linéaires de contraintes sur les rationnels. Nous y reviendrons dans le chapitre 5.

3.2.4 PLC réactive sur les domaines finis

Fages *et al.* [1998; 1995] ont proposé un cadre général pour la PLC réactive sur les domaines finis. Ce cadre général est basé sur la transformation d'arbres de résolution CSLD permettant ainsi d'utiliser le plus possible les calculs réalisés. Les

opérateurs de transformation définis permettent ainsi de traiter de l'ajout/retrait de contraintes et de l'ajout/retrait d'atomes dans un programme PLC.

D'un point de vue opératoire, Sola [1995] présente une méthode de backtrack intelligent et de maintien de déduction dans un cadre PLC(FD). La méthode proposée ne permet pas de résoudre des problèmes sur-contraints. Elle permet par contre de prendre en compte efficacement des opérations d'ajout et de retrait de contraintes. D'ailleurs, si le système de contraintes obtenu est contradictoire[1], la méthode proposée permet de revenir à un système satisfaisable et à ne recommencer un calcul que sur des contraintes devant être remises en cause.

Sola associe à chaque variable, un ensemble de contraintes responsables des modifications sur le domaine de cette variable définissant ainsi un *graphe de dépendance entre contraintes*. Ces informations permettent de reconstituer l'historique des modifications du domaine. Ainsi, lorsqu'un domaine devient vide à cause d'une contrainte donnée, il est facile de déterminer à partir des valeurs retirées par cette contrainte quelles informations remettre en cause.

D'autre part, les informations recueillies par le système présenté dans [Sola, 1995] permettent de ne pas recommencer complètement la résolution lorsqu'une contrainte est remise en cause. En effet, il n'y a pas besoin de défaire ce qui est indépendant de la contrainte supprimée. Lors d'un retrait, les informations enregistrées (le graphe de dépendance entre contraintes) permettent de connaître les points de choix qui peuvent être conservés. En particulier, sur les domaines finis, ces informations permettent de déterminer des valeurs qui avaient été écartées et qui peuvent *revenir* dans le domaine des variables. À ce stade, Sola propose de rétablir l'arc-consistance en activant les contraintes sur les domaines ainsi *élargis*. Il signale que son implémentation ne tirerait pas profit de la connaissance précise des valeurs réintroduites comme savent le faire des algorithmes comme DNAC4 par exemple (*cf.* section 3.2.2).

D'une manière générale, les idées présentées dans [Sola, 1995] nous paraissent fondamentales pour la relaxation de contraintes : pouvoir identifier un ensemble de contraintes responsables d'un conflit et pouvoir ne pas recommencer la résolution au début lorsqu'une contrainte est retirée.

De notre point de vue, cette approche est améliorable selon trois axes :

- La limitation de la recherche de responsabilité à un sous-ensemble des valeurs retirées[2] (sur la variable provoquant la contradiction) ne nous paraît pas adaptée à la relaxation de contraintes. En effet, des retraits de valeurs beaucoup plus anciens peuvent être dûs à des contraintes peu importantes dont la relaxation permettrait alors de *libérer* le domaine de la variable considérée. Il est possible de ne pas se limiter aux dernières opérations sur un domaine pour rechercher des causes de conflit.

- Dans le cas de la relaxation, il nous semble qu'il est nécessaire, d'une part, d'enregistrer des informations plus précises (au niveau des valeurs retirées et non pas simplement au niveau des variables) et, d'autre part, d'utiliser ces informations pour réaliser une relaxation de manière plus efficace.

1. Dans un système PLC le système de contraintes n'est pas unique : il peut y avoir des points de choix sur la mise en place des contraintes. Ainsi, il est possible obtenir un système de contraintes contradictoire et d'en trouver un autre qui soit satisfaisable. Par contre, il n'y a pas de notion de préférence sur les contraintes: toutes les combinaisons de contraintes acceptables pour le programme sont équivalentes.

2. En effet, Sola [1995] limite la recherche de responsabilités sur le «*clash sous-domaine*» : sous-ensemble des valeurs de la variable dont le domaine est vide qui ne sont pas compatibles avec la contrainte ne pouvant plus être vérifiée. Le but étant de trouver une affectation satisfaisant toutes les contraintes, cette recherche sur un sous-ensemble suffit. Notons qu'en pratique, du fait du système d'explication employé, la recherche de responsabilité sera réalisée sur toutes les valeurs retirées dans le domaine de la variable concerncée.

– L'approche proposée ici ne prend pas en compte les préférences de l'utilisateur et se contente donc de trouver un ensemble de contraintes réalisables. À notre avis, il est possible de prendre en compte ces préférences dans un cadre dynamique tout en conservant les propriétés de cette approche.

3.3 Relaxation de contraintes sur les problèmes statiques

Comme le signale Jampel [1996], il existe plusieurs façons d'affaiblir (relaxer) un problème sur-contraint pour qu'il devienne soluble. Il est ainsi possible de modifier la *structure* même du problème en introduisant par exemple des préférences sur les contraintes. C'est l'approche choisie par des systèmes tels que HCLP (Hierarchical Constraint Logic Programming) [Borning *et al.*, 1989] (section 3.3.1).

Une autre possibilité est de modifier la *sémantique* des contraintes. Il s'agit alors d'ajouter des possibilités à une contrainte : on parle alors d'affaiblissement de la contrainte plutôt que de relaxation.

> **Exemple 11 (Affaiblissement d'une contrainte) :**
> Considérons les variables X et Y sur lesquelles porte la contrainte $X \geqslant Y$. Une façon d'affaiblir cette contrainte est d'ajouter un couple (x, y) tel que $y > x$ à la liste des couples autorisés pour la contrainte.

Ce genre d'affaiblissement est pris en compte dans le schéma PCSP (Partial Constraint Satisfaction Problems) [Freuder, 1989] (section 3.3.2).

Il existe d'autres approches que nous présentons brièvement sections 3.3.3 et 3.3.4.

3.3.1 Hierarchical Constraint Logic Programming

L'approche Hierarchical Constraint Logic Programming (HCLP) joue un rôle central dans la communauté PLC [Jampel, 1996].

Le cadre général

L'approche HCLP ([Borning *et al.*, 1989; Wilson et Borning, 1993]) est une extension du cadre de la PLC aux expressions de préférences sur les contraintes.

Nous avons déjà vu dans le chapitre précédent les principaux concepts de cette approche : *préférence*, *comparateur* et *H-solution*.

D'une manière plus formelle, Borning *et al.* définissent deux ensembles de solutions : l'ensemble S_0 des solutions pour les contraintes requises et l'ensemble S des solutions pour la hiérarchie définie à partir de S_0 et d'un comparateur donné *better*.

$$S = \{\theta \mid \theta \in S_0 \wedge \forall \sigma \in S_0, \neg \text{better}(\sigma, \theta)\}$$

Sémantique opérationnelle

Une clause HCLP est de la forme :

$$p(t) : -q_1(t), \ldots, q_m(t), l_1 c_1(t), \ldots, l_n c_n(t).$$

où t est une liste de termes, $p, q_1, \ldots, q_m$ sont des symbole de prédicat et $l_1 c_1(t)$, $\ldots$, $l_n c_n(t)$ sont des contraintes étiquetées par leur préférence. Un programme HCLP est une liste de telles clauses et une requête est un multi-ensemble d'atomes.

Les requêtes sont exécutées comme en PLC excepté que les contraintes non requises ne sont pas prises en compte et simplement accumulées[3]. Une fois la requête

3. Les contraintes ne servent alors qu'*a posteriori*, ce qui n'est pas totalement compatible avec la philosophie PLC.

complètement exécutée, la hiérarchie des contraintes non requises accumulées jusqu'à présent est utilisée pour raffiner les domaines des variables.

Mise en œuvre

L'approche HCLP($\mathcal{D}$, C) est paramétrée par le domaine de calcul $\mathcal{D}$ et par le comparateur C utilisé pour résoudre la hiérarchie *i.e.* renvoyer un élément de l'ensemble S des meilleures solutions.

Un interprète très simple pour HCLP($\mathbb{R}$, C_{LPB}) est proposé par Wilson et Borning [1993]. Cet interprète fonctionne en deux phases : dans un premier temps, seules les contraintes requises sont introduites dans le système et les contraintes de préférence sont accumulées. Dans un second temps, la hiérarchie de contraintes ainsi obtenue est résolue. Plus précisément, les contraintes sont introduites par niveau d'importance (de la plus importante à la moins importante) lorsque cela est possible jusqu'à ce qu'il n'y ait plus de contraintes ou que celles qui restent soient vérifiées (par les domaines courant des variables) ou contradictoires[4] avec les contraintes actives.

Cet interprète ne fonctionne donc que si le comparateur utilisé est le comparateur C_{LPB} et que les préférences associées aux contraintes du problème sont connues à l'avance. En effet, seul le comparateur C_{LPB} peut être ainsi optimisé incrémentalement par parcours de la hiérarchie et sans retour arrière. Notons que l'ajout d'une nouvelle contrainte (si elle provoque une contradiction) remet en cause la solution courante et demande une nouvelle résolution (pouvant reprendre complètement depuis le début[5]). De plus, on doit savoir après chaque introduction de contrainte si le système obtenu est soluble[6] et donc l'approche proposée n'est réalisable que pour des solveurs garantissant la consistance globale après chaque ajout de contraintes (c'est possible sur les réels ou les rationnels en prenant en compte des contraintes linéaires). Ainsi, de Backer et Béringer [1991a] proposent une implémentation efficace du cadre HCLP pour les rationnels.

Limitations

Les approches de type HCLP présentent donc trois types d'inconvénients :

- Elles demandent la connaissance complète du système de contraintes et des préférences associées à celles-ci avant la résolution.
- Elles ne sont pas adaptées au cas dynamique. Elles ne savent pas tirer profit du passé.
- La plupart des mises en œuvre proposées n'utilisent pas de manière active les contraintes de préférence. Celles-ci sont simplement utilisées en guise de test. Ceci ne nous paraît pas vraiment compatible avec la philosophie *contraintes*.

3.3.2 Partial Constraint Satisfaction Problems

Le cadre général

Freuder [1989] a proposé un cadre pour le traitements des CSP sur-contraints. Un *Partial Constraint Satisfaction Problem* (PCSP) est un triplet :

$$\langle P, (PS, <), (M, (N, S)) \rangle$$

4. Cette contradiction est constatée sans calcul. L'état courant des domaines des variables ne permet pas de vérifier la contrainte testée.

5. Considérons le cas de l'absence soudaine d'un professeur pour un problème d'emploi du temps, cette dernière contrainte est une contrainte requise puisqu'elle ne peut être relaxée. Une résolution utilisant les techniques d'HCLP demande donc de tout reprendre au début.

6. Dans le cas des domaines finis, il faudrait alors nécessairement énumérer. On résoudrait alors à chaque étape un problème difficile.

où :

- P est un CSP *i.e.* un triplet (V, D, C) où V est un ensemble de variables, D l'ensemble des domaines associés à ces variables et C un ensemble de contraintes,
- PS est un ensemble de problèmes (CSPs) et «$<$» un ordre partiel sur PS,
- M est une métrique sur PS,
- (N, S) sont des bornes **n**écessaires et **s**uffisantes sur la distance entre un problème $\mathcal{P}$-satisfaisable dans PS et le problème donné P.

Une solution pour un PCSP est un problème P' de PS possédant une instanciation satisfaisant toutes ses contraintes et tel que la distance de P à P' selon M est inférieure à N. Une solution sera dite suffisante si la distance est inférieure ou égale à S. Ces bornes sont utilisées lors d'une recherche arborescente. Une solution optimale est une solution pour laquelle la distance à P est minimale.

Recherche de solution

La relation d'ordre la plus utilisée est celle qui consiste à compter le nombre de contraintes vérifiées. Une telle instance de PCSP est appelée MaxCSP.

De nombreux résultats ont été publiés concernant MaxCSP [Wallace et Freuder, 1996; Wallace, 1996]. Les méthodes utilisées sont basées sur une recherche arborescente dans l'espace PS.

On constate dans ce dernier cadre l'absence du traitement des préférences. Sinon, comme pour HCLP on retrouve la nécessité de connaître à l'avance à la fois l'ensemble des contraintes du problème et les préférences associées à ces problèmes. De plus, là encore, les démarches retenues ne sont pas adaptées au cas dynamique : elles sont peu incrémentales.

3.3.3 Cadres généraux pour la relaxation

Dans le domaine des CSP, les approches proposées étendent toutes d'une certaine manière le cadre classique que ce soit en intégrant des préférences sur les contraintes, des coûts ou des probabilités. Chaque extension utilise des opérateurs mathématiques particuliers (+, max) pour agréger les violations de contraintes en vue de comparaisons ultérieures.

Deux cadres généraux permettant de généraliser toutes ces extensions ont été proposés : l'un basé sur les semi-anneaux [Bistarelli *et al.*, 1995] et l'autre basé sur un monoïde commutatif totalement ordonné [Schiex, 1992; Schiex et Verfaillie, 1995].

Dans le dernier cas, il s'agit du cadre dit des CSP valués (VCSP). Dans [Schiex et Verfaillie, 1995], les auteurs présentent des extensions aux VCSP des algorithmes classiques (backtrack, forward-checking, ...) de résolution de CSP.

Les liens entres les deux approches ont été présentés dans [Bistarelli *et al.*, 1996].

Ces approches nécessitent que l'ensemble des contraintes soient connues à l'avance, de même que les valuations associées. Elles ne sont pas non plus incrémentales.

3.3.4 Autres approches

Les travaux de Fages, Fowler et Sola [Fages *et al.*, 1994; Fowler, 1993; Fages, 1993] utilisent une recherche arborescente en intégrant les contraintes de préférences dans la fonction objectif cherchant ainsi à optimiser cette fonction.

On trouve dans [Jampel *et al.*, 1996] une récapitulation d'autres approches. Citons en particulier Bouquet et Jégou [1996] qui utilisent des OBDD (Ordered Binary Decision Diagrams) pondérés comme structure de donnée permettant de

coder l'aspect dynamique d'un problème mais la complexité spatiale obtenue est exponentielle.

3.4 Relaxation de contraintes sur les problèmes dynamiques

Dans cette section, nous nous intéressons aux travaux liés aux ajouts/retraits de contraintes dans les systèmes dynamiques de contraintes.

3.4.1 Instance «*incrémentale*» de HCLP : le système IHCS

Le système IHCS (Incremental Hierarchical Constraint Solver) proposé par Menezes [Menezes *et al.*, 1993; Menezes et Barahona, 1996] a pour but de fournir des solveurs de hiérarchie incrémentaux en vue d'une utilisation dans le cadre HCLP. En fait, il ne s'agit plus vraiment d'un système HCLP puisque seuls les concepts de base sont utilisés et que la vision opérationnelle proposée par Borning n'est pas respectée.

L'approche proposée par IHCS est une technique d'exploration d'un espace de configurations. Une configuration[7] est un triplet $\langle AS \bullet RS \bullet US \rangle$ de contraintes actives (AS), relaxées (RS) et à traiter (US). Les contraintes ne sont pas alors traitées forcément selon leur ordre d'arrivée dans le système. Le choix d'une contrainte à intégrer depuis l'ensemble US est laissé au libre choix de l'implanteur.

Afin, de limiter l'exploration de l'espace de recherche à une petite partie, Menezes *et al.* utilisent la notion de *dépendance entre contraintes*[8].

Définition 20 (Dépendance entre contraintes)
Une contrainte c_a dépend d'une contrainte c_b si et seulement si c_b a retiré une valeur dans le domaine d'une variable qui apparaît dans c_a.

Lorsqu'une contradiction est identifiée, c'est qu'au moins une contrainte n'est plus vérifiée. Ainsi, on peut limiter la recherche d'une configuration optimale aux configurations définies à partir des contraintes dont dépend la contrainte en échec.

Plus précisément, un ordre total est défini sur les configurations déterminées à partir du sous-ensemble des contraintes du problème dont dépend la contrainte en échec. Ensuite, la recherche est réalisée suivant cet ordre jusqu'à ce que soit trouvée une configuration satisfaisante.

Le changement de configuration est en fait réalisé par backtrack standard. Tout ce qui a été calculé entre l'instant d'introduction de la contrainte relaxée et l'instant de sa relaxation est complètement remis en cause. Il n'évite donc pas le phénomène de *thrashing* pendant le traitement d'un retrait de contrainte.

Cette approche, bien que nommée incrémentale, ne l'est pas totalement. Pour les retraits de contraintes, on ne retrouve pas l'incrémentalité que l'on a pour l'ajout. En effet, le retrait est réalisé par backtrack sans tenir compte de ce qui s'est passé entre l'introduction de la contrainte à relaxer et l'instant de sa relaxation.

De plus, on peut d'ores et déjà constater que la notion de dépendance entre contraintes pourrait être rendue plus précise permettant ainsi de circonscrire la recherche à un plus petit ensemble de configurations. On peut en effet préférer travailler sur les valeurs des variables permettant alors de préciser finement les liens entre les contraintes et les variables.

7. Notre notion de configuration n'a pas besoin de tenir compte des contraintes à introduire.
8. C'est le même système de dépendance que celui utilisé par [Fages *et al.*, 1995; Sola, 1995] (*cf.* section 3.2.4).

Néanmoins, signalons les travaux récents de Holzbaur *et al.* [1996] qui proposent une adaptation efficace (nous proposons quelque chose de très similaire chapitre 5) de la philosophie IHCS au cas des contraintes linéaires sur les rationnels.

3.4.2 Propagation locale

Si on se limite aux contraintes dites *fonctionnelles*[9], on peut appliquer des techniques de propagation locale [Trombettoni, 1997]. Les modifications sur les valeurs des variables sont alors propagées simplement en utilisant les *méthodes*.

L'utilisation de telles techniques permet alors de proposer des systèmes incrémentaux capables de résoudre des problèmes sur-contraints dans le cadre d'HCLP.

Ainsi, l'algorithme DeltaBlue [Freeman-Benson *et al.*, 1990] permet de traiter par propagation locale des systèmes de contraintes fonctionnelles simples (variable de sortie unique) dont le graphe de dépendance est sans circuit. Depuis, une extension a été proposée : l'algorithme SkyBlue [Sannella, 1993] qui permet de traiter les contraintes fonctionnelles à sorties multiples et dont le graphe de dépendance contient des circuits. Ces deux algorithmes sont conçus pour fonctionner avec le comparateur C_{LPB}.

Le système Houria [Bouzoubaa *et al.*, 1995] permet de calculer une solution respectant d'autres comparateurs. Par contre, le système de contraintes résolu doit être sans circuit.

L'utilisation de la propagation locale pour des contraintes non fonctionnelles a conduit au système Indigo [Borning *et al.*, 1996]. Ce système permet de traiter des contraintes d'inégalité ($x \leqslant y$) en utilisant une sorte de propagation d'intervalles. Malheureusement, les auteurs signalent qu'ils n'ont pu proposer une version incrémentale de cet algorithme. Ceci est dû au fait que jusqu'à maintenant il n'existe pas de proposition de système incrémental sur les intervalles dans la littérature.

Citons enfin les travaux théoriques présentés dans [Haselböck *et al.*, 1993] qui proposent un cadre formel pour les techniques de réparation de solution dans le cas d'une solution ne respectant pas l'ensemble des contraintes d'un problème.

3.4.3 «*Précurseurs*» de la relaxation de contraintes

Pour résoudre les CSP, une phase d'énumération est nécessaire. Une recherche utilisant le backtrack standard n'est pas efficace, c'est pourquoi différents algorithmes ont été proposés pour l'améliorer. Ces algorithmes utilisent un mécanisme d'«*apprentissage*[10]».

Nous présentons l'algorithme *Dynamic Backtracking* [Ginsberg, 1993] de recherche de solution dans un CSP. Pour introduire cet algorithme, nous utilisons, comme le font Verfaillie et Schiex [1995], deux autres algorithmes précurseurs (*Conflict Directed Backjumping* [Prosser, 1993] et *Nogood Recording* [Schiex et Verfaillie, 1994] version étendue de [Schiex et Verfaillie, 1993]). Dans leur article, Verfaillie et Schiex proposent des versions dynamiques de *Nogood Recording* et *Dynamic Backtracking*. Tout simplement, les informations enregistrées sont conservées d'une résolution à l'autre car ces algorithmes se prêtent bien à ce partage. Nous nous contenterons d'une présentation des versions statiques de ces algorithmes.

9. Définies par l'intermédiaire de *méthodes* permettant de calculer la valeur d'une variable connaissant les valeurs des autres variables. Il s'agit des contraintes *bijectives* au sens de [David, 1994]. On trouvera un état de l'art sur la propagation locale dans [Trombettoni, 1997].

10. Il ne s'agit pas là de l'apprentissage au sens IA du terme mais plutôt de la capacité d'un système de résolution à se *souvenir* de ce qu'il a fait pour soit ne plus le refaire, soit analyser cette trace pour décider la suite à donner à la résolution.

Définitions

Conflict Directed Backjumping [Prosser, 1993], *Nogood Recording* [Schiex et Verfaillie, 1994] et *Dynamic Backtracking* [Ginsberg, 1993] sont des extensions de l'algorithme standard dit de *backtrack*.

Un tel algorithme est tel que :

- dans tout état, l'ensemble des variables est divisé en variables affectées (affectation courante) et variables non affectées ;
- dans l'état de départ, aucune variable n'est affectée (affectation vide) ;
- une transition consiste à étendre l'affectation courante en choisissant une variable v non affectée et en lui affectant une valeur *val* choisie parmi les valeurs non encore explorées ;
- si l'affectation courante devient de ce fait incohérente, un retour est effectué sur le choix fait pour v ; si toutes les valeurs de v ont été explorées, un retour est effectué sur le choix de variable immédiatement précédent ;
- il y a arrêt si une affectation cohérente de l'ensemble des variables est produite (solution) ou s'il est constaté que l'affectation vide[11] ne peut pas être étendue en une affectation cohérente de l'ensemble des variables (non existence d'une solution).

De Conflict Directed Backjumping à Dynamic Backtracking

L'algorithme *Conflict Directed Backjumping* est une amélioration de l'algorithme *backtrack*. Cette amélioration porte sur le mécanisme de retour dans le cas où toutes les possibilités d'extension d'une affectation courante sur une variable v ont été explorées. Soit V' l'ensemble des variables affectées, ordonné selon l'ordre d'affectation ; soit v' la dernière variable de V' ; soit v'' la dernière variable de V' dont l'affectation est un élément de l'explication de l'échec constaté sur v. Une telle explication de l'échec est appelée un *nogood*.

Définition 21 (Nogood)
Un ***nogood*** *est une affectation partielle contradictoire. L'*arité *d'un nogood est le nombre de variables concernées par l'affectation partielle considérée.*

Alors que l'algorithme *backtrack* revient systématiquement sur v', l'algorithme *Conflict Directed Backjumping*, s'appuyant implicitement sur la notion de *nogood*, revient sur la dernière variable, selon l'ordre d'affectation, dont l'affectation est un élément de l'explication de l'échec constaté sur v : la variable v''. C'est un mécanisme de backtrack intelligent.

L'algorithme *Nogood Recording* peut être vu comme une «simple» amélioration de l'algorithme *Conflict Directed Backjumping*. Cette amélioration consiste à mémoriser les *nogoods* produits au cours de la recherche et à les réutiliser pour réduire l'espace de recherche restant.

Le nombre de *nogoods* produits au cours d'une recherche pouvant croître exponentiellement en fonction de la taille du problème, une limitation pratique, analogue à celle qui est utilisée pour le filtrage, consiste à limiter l'arité des *nogood* mémorisés[12].

L'algorithme *Dynamic Backtracking* [Ginsberg, 1993] propose un autre mécanisme de retour dans le cas où toutes les possibilités d'extension d'une affectation courante sur une variable ont été explorées.

11. Affectation cohérente du départ de la recherche.
12. Nous verrons section 7.3.3 que ce problème peut être résolu d'une autre manière.

Alors que l'algorithme *backtrack* revient systématiquement sur v' (nous reprenons les mêmes notations que précédemment), l'algorithme *Dynamic Backtracking* revient sur la variable v'' comme le fait *Conflict Directed Backjumping* mais **sans** désaffecter les variables suivant v'' dans V'.

La figure 3.2 représente les différences de traitement entre le backtrack standard, *Conflict Directed Backjumping*, *Nogood Recording* et *Dynamic Backtracking*. Sur cet exemple, concernant la recherche d'une instanciation pour huit variables, nous considérons le moment de la résolution où les trois premières variables étant instanciées, l'instanciation de la quatrième provoque une contradiction. Nous supposerons qu'un nogood peut alors être calculé concernant les variables V_1 et V_2 ce qui signifie que V_3 et V_4 n'ont rien à voir avec l'échec courant.

L'algorithme de *backtrack* standard ne peut tirer profit de cette information et revient donc sur l'instanciation de la variable V_4. Ainsi, toutes les possibilités pour V_3 et V_4 vont devoir être explorées avant de revenir au *point* intéressant V_2. Par contre, les algorithmes *Conflict Directed Backjumping* et *Nogood Recording* utilisent cette information et reviennent eux directement sur l'instanciation de V_2 en remettant en cause V_3 et V_4. La différence entre les deux algorithmes tient au fait que *Nogood Recording* va conserver le nogood identifié pour ne pas retomber sur une condition d'échec déjà connue. L'algorithme *Dynamic Backtracking*, quant à lui, utilise au maximum l'information extraite de l'échec courant : il remet en cause l'instanciation de V_2 mais sans modifier ni V_3 ni V_4 puisque ces deux variables n'ont rien à voir avec l'échec courant. Aucun calcul n'est donc nécessaire pour retrouver de *bonnes* instancations pour V_3 et V_4 contrairement aux trois autres algorithmes.

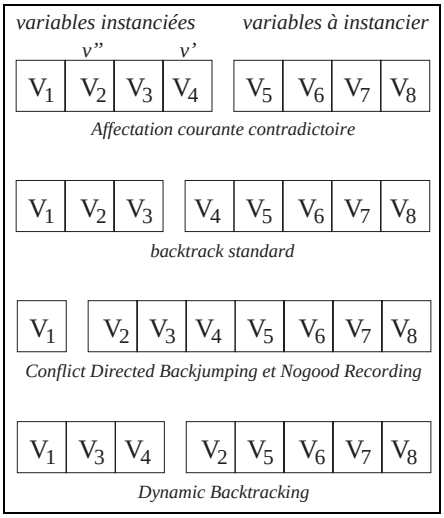

FIG. 3.2 – *Comportement des différentes techniques d'énumération*

L'espace de recherche résultant de ces transformations n'est plus un arbre mais un graphe. On peut aussi dire que l'arbre de recherche (originel) est exploré d'une manière non triviale : il s'agit d'une approche par sauts.

Pour réaliser cette recherche et assurer sa complétude, l'algorithme *Dynamic Backtracking* utilise la notion d'*eliminating explanation* qui est une variante de la notion de *nogood*.

Définition 22 (Eliminating explanation selon [Verfaillie et Schiex, 1995]) *Soit v une variable ; soit val une de ses valeurs ; soit A une affectation partielle n'impliquant pas v ; A est une* ***eliminating explanation*** *pour la valeur val de v si et seulement si $A \cup (v, val)$ est un* nogood.

Dynamic Backtracking et la relaxation de contraintes

L'algorithme *Dynamic Backtracking* fonctionne comme un véritable système de relaxation de contraintes sur des contraintes spécifiques : les contraintes d'affectation de valeur à une variable. En effet, considérons l'affectation d'une valeur à une variable comme un ajout de contrainte d'égalité et de même, la désaffectation comme un retrait (une relaxation) de cette même contrainte. Ainsi, lorsque l'affectation partielle est incohérente c'est que le système de contraintes courant est inconsistant ; l'algorithme détermine alors la *meilleure* (plus utile) contrainte (d'égalité) à relaxer (désaffectation) sans modifier l'état des autres contraintes (aucune modification des autres affectations). Ceci étant répété jusqu'à ce qu'une affectation partielle cohérente soit trouvée pour pouvoir continuer.

On peut récapituler les caractéristiques de ces trois algorithmes dans un tableau (*cf.* tableau 3.2) présentant leur comportement par rapport à l'identification d'un point de choix pertinent à remettre en cause, la capacité à éviter le *thrashing* et l'utilisation d'un DMS.

	CBJ	*NR*	*DBT*
Identification	oui	oui	oui
Pas de thrashing	non	oui (en partie)	oui
DMS	non	oui	oui

TAB. 3.2 – *Caractéristiques des algorithmes de recherche*

Comme on le voit sur le tableau 3.2, l'algorithme *Dynamic Backtracking* est l'algorithme qui répond le mieux aux critères d'un système de relaxation de contraintes. Il manque en fait à *Dynamic Backtracking* la gestion de la propagation des affectations à chaque instanciation à la manière de l'algorithme MAC (Maintaining Arc-Consistency – [Sabin et Freuder, 1994]).

3.5 Motivations

Comme nous l'avons fait à l'instant pour les algorithmes d'énumération, on peut récapituler certaines techniques utilisées pour résoudre les problèmes sur-contraints dans un tableau (*cf.* tableau 3.3). Nous n'avons conservé que la référence HCLP et celles capables de traiter les systèmes dynamiques de contraintes. Nous reportons dans ce tableau l'utilisation ou non d'une technique d'identification de contraintes responsables d'une contradiction, la capacité à éviter le *thrashing* et l'utilisation ou non d'un DMS.

	HCLP	IHCS	Sola *et al.*
Identification	non	oui (améliorable)	oui (améliorable)
Pas de thrashing	non	non	oui (en partie)
DMS	non	oui	oui

TAB. 3.3 – *Comparaisons de certaines techniques de relaxation*

Comme on le voit sur le tableau 3.3, aucun des systèmes proposés jusqu'à présent ne présente toutes les caractéristiques nécessaires à un système de relaxation de contraintes dans un cadre dynamique.

Nous pensons que l'identification de points de retour (contraintes à relaxer) peut être améliorée en considérant non plus le graphe de dépendance entre contraintes comme IHCS ou les travaux de Sola, mais plutôt une information précise sur la responsabilité précise des contraintes concernant le retrait des valeurs. Ces informations

permettraient d'ailleurs un traitement véritablement incrémental de la relaxation évitant le *thrashing*. En fait, on peut songer à une généralisation de l'algorithme *Dynamic Backtracking* en considérant non plus plusieurs valeurs possibles pour une variable mais deux états possibles pour une contrainte (active ou relaxée) et en évitant le *thrashing* de la même façon.

Par ailleurs, l'une des conclusions du projet ESPRIT CHIC (Constraint Handling in Industry and Commerce) [Chamard *et al.*, 1995] sur les problèmes sur-contraints montre l'importance de notre thème de recherche et confirme l'intérêt de la technologie pressentie (ATMS maîtrisés) :

> *With CLP, the user should have a way of reconsidering the constraints imposed to the solution. Unfortunately, there is no general method for explaining failures and detecting which constraints have to be relaxed. If technologies like dependency-based backtracking or ATMS were better mastered and worked at a reasonable cost, perhaps something might be done in terms of finding the latest decision causing a given failure. But it is apparently not the case (see the implementation by ECRC of an Intelligent Backtracking library in ECLiPSe, based on [CERT IBT 93], for a non-conclusive attempt to implement intelligent backtracking methods). Methods like that proposed e.g. in [De Backer and Beringer IJCAI 93: p.300] for the detection of minimal inconsistent sets are restricted to linear programming.*

Deuxième partie

Un cadre général pour la relaxation

Chapitre 4

Un cadre général : relax($\mathcal{D}$, $\mathcal{C}$, $\mathcal{P}$)

Sommaire

DANS CE CHAPITRE, nous définissons le cadre général relax($\mathcal{D}$, $\mathcal{C}$, $\mathcal{P}$) qui montre comment certaines contraintes de l'ensemble $\mathcal{C}$ portant sur le domaine $\mathcal{D}$ peuvent être relaxées si la propriété $\mathcal{P}$ n'est pas vérifiée. Nous montrons alors que la relaxation de contraintes peut être modélisée par une recherche de type meilleur d'abord sur l'espace des configurations. Cette recherche est paramétrée par trois fonctions (`explique-contradiction`, `meilleure-configuration`, `maj-configuration`) qui devront être spécialisées suivant le domaine de contraintes $\mathcal{D}$ et la nature des contraintes de $\mathcal{C}$. Un soin tout particulier sera porté aux changement de configurations afin d'éviter le phénomène de *thrashing*. Enfin, une classe de comparateurs particuliers permettant de maîtriser le nombre d'explications enregistrées sera étudiée. Ce chapitre présente et étend les concepts présentés dans [Jussien et Boizumault, 1997a; Jussien et Boizumault, 1997b].

4.1 L'espace de recherche

Nous cherchons la C-solution d'un problème donné dans l'espace des configurations. Cet espace de recherche peut se représenter à l'aide d'un arbre binaire dont les sommets (nœuds) sont les contraintes du problème et les arêtes l'état possible de la contrainte associée (active ou relaxée). Pour une contrainte c_a, on notera a l'état actif et a' l'état relaxé. Les sous-arbres correspondant à deux états différents d'une même contrainte sont identiques.

Un tel arbre est construit dynamiquement au fur et à mesure de l'introduction des contraintes[1]. Notons que sa taille est 2^e où e est le nombre de contraintes du problème.

> **Exemple 12 (Arbre de la recherche) :**
> On trouve figure 4.1 un exemple d'arbre binaire pour 3 contraintes c_a, c_b et c_c.

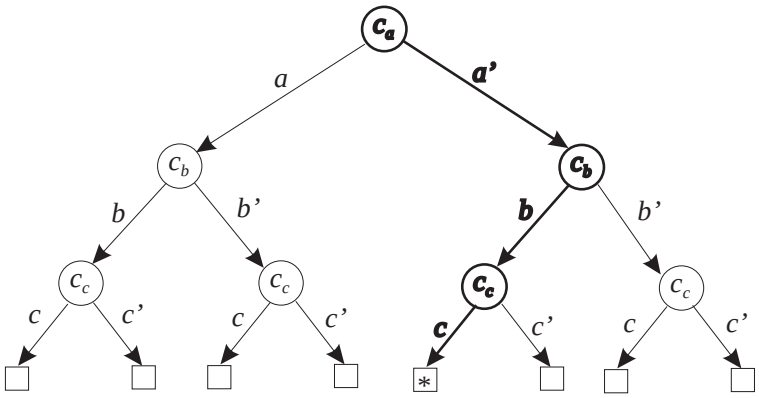

FIG. 4.1 – *Arbre binaire de recherche pour* $\{c_a, c_b, c_c\}$.

Un chemin de la racine à une feuille représente une et une seule *configuration*.

1. Le cas d'un retrait de contrainte est considéré comme une relaxation définitive et ne conduit donc pas à une modification de l'arbre de recherche: les sous-arbres correspondant à l'activation de la contrainte seront simplement écartés de la recherche.

Exemple 13 (Relation configuration - branche) :

La feuille marquée par $*$ sur la figure 4.1 (ou de manière équivalente, le chemin indiqué en gras de la racine vers cette feuille) correspond à la configuration $\langle\{c_b, c_c\}, \{c_a\}\rangle$.

Une configuration est représentée par le chemin qui lui est associé dans l'arbre de recherche.

Exemple 14 (Notation d'une configuration) :
Sur la figure 4.1, la configuration $\langle\{c_b, c_c\}, \{c_a\}\rangle$ peut être représentée de manière équivalente par le chemin $a'bc$.

La recherche d'une C-solution pour un problème donné correspond à la recherche de la meilleure feuille $\mathcal{P}$-satisfaisable, pour un comparateur C donné, sur l'arbre de recherche associé au problème traité.

4.2 Une approche de type meilleur d'abord

4.2.1 Un exemple

Nous utiliserons le problème de l'exemple 15 pour illustrer notre approche.

Exemple 15 (Problème traité) :
Soient 4 contraintes : c_a, c_b, c_c et c_d introduites dans cet ordre. L'arbre de recherche associé à ces quatre contraintes est dessiné figure 4.2.

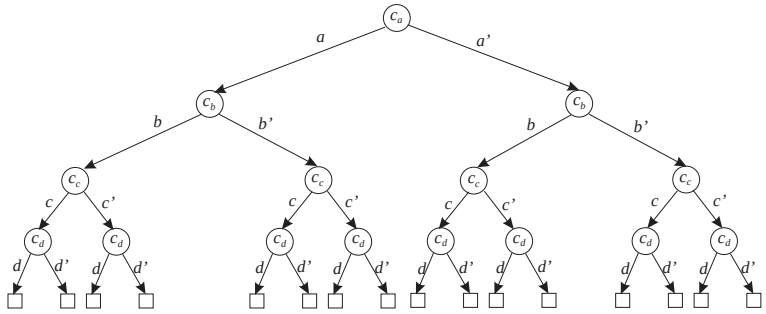

FIG. 4.2 – *Arbre de recherche pour* $\{c_a, c_b, c_c, c_d\}$.

Supposons que les préférences de l'utilisateur soient représentées par une valeur numérique dans ce problème. Nous conviendrons d'affecter les poids 1, 2, 4 et 8 aux contraintes c_a, c_b, c_c et c_d respectivement. Nous noterons $p(c)$ la préférence associée à la contrainte c.

Le comparateur utilisé sur ce problème est noté $\prec$ et est défini de la façon suivante :

Soient $C_1 = \langle A_1, R_1\rangle$ et $C_2 = \langle A_2, R_2\rangle$ deux configurations.

$$C_1 \prec C_2 \Longleftrightarrow \sum_{c \in R_1} p(c) < \sum_{c \in R_2} p(c) \tag{4.1}$$

Soit une propriété $\mathcal{P}$ quelconque, nous supposerons que $\mathcal{P}$ est vérifiée pour $\{c_a, c_b, c_c\}$ mais qu'elle ne l'est plus pour $\{c_a, c_b, c_c, c_d\}$. Nous sommes donc en présence d'un problème sur-contraint car le système de contraintes actives est $\mathcal{P}$-contradictoire. La configuration de départ est la configuration $\langle\{c_a, c_b, c_c, c_d\}, \varnothing\rangle$.

De plus, nous supposerons que les ensembles de contraintes $\{c_b, c_d\}$ et $\{c_c, c_d\}$ sont $\mathcal{P}$-contradictoires. Ces ensembles sont minimaux pour l'inclusion *i.e.* tout

ensemble de contraintes n'en contenant aucun des deux est $\mathcal{P}$-satisfaisable. Ces informations ne sont bien sûr pas disponibles au démarrage de la résolution.

Nous présentons dans cette section un algorithme de recherche de C-solution dans l'arbre de recherche associé à un problème donné. Cet algorithme conduit à une exploration en meilleur d'abord de l'arbre de recherche.

4.2.2 Notions de base

Nous introduisons les notions d'explication de contradiction et de configuration prometteuse avant de présenter notre algorithme de recherche.

Explication de contradiction

Notre méthode repose sur la notion fondamentale d'*explication de contradiction*[2].

Définition 23 (Explication de Contradiction)
Soit $\langle A, R\rangle$ une configuration. Soit $E_c \subset A$ un ensemble de contraintes.

E_c est une ***explication de contradiction*** *si et seulement si l'ensemble E_c ne vérifie pas la propriété $\mathcal{P}$.*

Ainsi, par définition de la notion de propriété (définition 3 page 17), E_c est une ***explication de contradiction*** *si E_c est un ensemble de contraintes inconsistant.*

> **Exemple 16 (Explication de contradiction) :**
> Soit un CSP pour lequel : $V = \{v_1, v_2, v_3\}$, $\forall i, D_{v_i} = \{1, 2, 3\}$ et $C = \{c_1, c_2, c_3, c_4\}$ avec $c_1 : v_1 > v_2$, $c_2 : v_3 > v_1$, $c_3 : v_1 = 2$ et $c_4 : v_3 < v_2$. La propriété $\mathcal{P}$ choisie est $\mathcal{P}_{ac}$: l'arc-consistance est vérifiée.
> La configuration $\langle C, \varnothing\rangle$ est $\mathcal{P}$-contradictoire. L'ensemble C est une explication de contradiction mais $\{c_1, c_2, c_4\}$ aussi. D'ailleurs, cette dernière est plus intéressante car plus précise : il ne sert à rien de relaxer uniquement la contrainte c_3, de toutes façons la contradiction restera.

Pour un problème donné, nous appelons $\mathcal{E}_c$ l'ensemble des *explications de contradiction* déterminées à moment donné de la résolution. Cet ensemble est important car il réunit les informations accumulées pendant la résolution. L'idée est d'essayer de ne pas retrouver une sous-configuration contradictoire déjà explorée.

Nous verrons plus tard que nous recherchons en fait les *explications de contradiction* les plus précises possibles. En effet, pour une configuration $\langle A, R\rangle$ $\mathcal{P}$-contradictoire, A est toujours une *explication de contradiction* mais bien sûr ceci ne nous apprend rien comme nous l'avons vu dans l'exemple 16. C'est pourquoi nous nous intéresserons plus aux *explications de contradiction* minimales pour l'inclusion (il s'agit des explications les plus précises possibles).

Configuration prometteuse

Une configuration *prometteuse*[3] est une configuration qui ne contient pas de sous-ensemble de contraintes $\mathcal{P}$-contradictoire.

---

2. Nous préférons ne pas employer le terme *nogood* [Verfaillie et Schiex, 1995] qui bien qu'ayant la même signification générale ne capture pas entièrement notre vision des choses.

3. Notre notion de configuration prometteuse est différente de celle du système IHCS [Menezes et Barahona, 1996]. En effet, une configuration prometteuse au sens IHCS est une configuration qui ne conduit pas à une contradiction *i.e.* qui vérifie la propriété $\mathcal{P}$. Ceci ne peut pas être vérifié sans calcul tandis que notre notion ne nécessite pas de calcul supplémentaire.

Définition 24 (Configuration prometteuse)
Une configuration $\langle A, R \rangle$ *est* **prometteuse** *relativement à* $\mathcal{E}_c$ *si l'ensemble* A *des contraintes actives ne contient pas de sous-ensemble qui soit une* explication de contradiction i.e.

$$\forall E \in \mathcal{E}_c, E \not\subset A \tag{4.2}$$

Exemple 17 (Configuration prometteuse) :
Pour l'exemple 16, $\langle \{c_1, c_2, c_3\}, \{c_4\} \rangle$ est une configuration prometteuse avec $\mathcal{E}_c = \{\{c_1, c_2, c_4\}\}$.

Proposition 3 (Configuration prometteuse et Recouvrement)
Une configuration $\langle A, R \rangle$ *est prometteuse si et seulement si :*

$$\forall E \in \mathcal{E}_c, R \cap E \neq \varnothing \tag{4.3}$$

▷ **Preuve :**

- Supposons que la condition de l'équation 4.3 soit vérifiée. Comme les ensembles A et R sont complémentaires, A ne peut contenir d'*explication de contradiction* déjà connue. $\langle A, R \rangle$ est donc une configuration prometteuse.
- Inversement, si $\langle A, R \rangle$ est une configuration prometteuse, chaque élément E de $\mathcal{E}_c$ contient un élément dans R car sinon A contiendrait une *explication de contradiction* déjà connue. La condition de l'équation 4.3 est donc vérifiée.

◁

En d'autres termes l'ensemble R «*recouvre*» l'ensemble $\mathcal{E}_c$ des *explications de contradiction*. Rappelons ce qu'est un recouvrement.

Définition 25 (Recouvrement – [Gondran et Minoux, 1990])
Un recouvrement des sommets d'un hypergraphe $H = (\mathcal{S}, \mathcal{E})$ *est une famille* $\mathcal{F} \subset \mathcal{E}$ *telle que :*

$$\bigcup_{E_j \in \mathcal{F}} E_j = \mathcal{S}$$

Exemple 18 (Configuration prometteuse et recouvrement) :
Considérons le problème de l'exemple 15. Considérons l'ensemble d'explications de contradiction suivante :

$$\mathcal{E}_c = \{\{c_a, c_b, c_d\}, \{c_a, c_c, c_d\}, \{c_b, c_d\}\}$$

Si les sommets du graphe associé sont les différentes explications de contradiction et que les arêtes sont étiquetées par les différentes contraintes et relient les explications de contradiction concernées celles-ci (voir figure 4.7 page 52), la configuration $\langle \{c_a, c_d\}, \{c_b, c_c\} \rangle$ est une configuration prometteuse puisque l'ensemble $\{c_b, c_c\}$ recouvre l'ensemble $\mathcal{E}_c$. $\langle \{c_a, c_b, c_c\}, \{c_d\} \rangle$ est une aussi une configuration prometteuse.

Ainsi, lorsqu'on connaît un recouvrement de l'ensemble d'*explications de contradiction* $\mathcal{E}_c$, on peut en déduire une configuration prometteuse associée. On peut noter qu'il est inutile de rechercher une C-solution parmi les configurations non prometteuses.

Proposition 4 (Caractérisation d'une C-solution)
Une C-solution est une configuration prometteuse.

▷ **Preuve :** Soit $C_f = \langle A, R \rangle$ une C-solution. Supposons que C_f ne soit pas une configuration prometteuse. Alors, il existe un élément E de $\mathcal{E}_c$ tel que $E \subset A$, ce qui implique que C_f est $\mathcal{P}$-contradictoire (l'ensemble des contraintes actives contient une *explication de contradiction* connue). Ceci est en contradiction avec le fait que C_f est une C-solution. ◁

Nous pouvons donc dans la suite nous contenter d'explorer l'espace des configurations prometteuses.

4.2.3 Algorithme de recherche de C-solution

Nous introduisons trois fonctions définies à partir des notions d'explication de contradiction et de configuration prometteuse qui vont nous permettre de proposer une méthode simple de recherche de C-solution.

- `explique-contradiction`($\langle A, R \rangle$) fournit, à partir d'une configuration $\mathcal{P}$-contradictoire $\langle A, R \rangle$, une ou plusieurs *explications de contradiction*. La définition de cette fonction dépend du domaine du calcul.

- `meilleure-configuration`($\langle A, R \rangle$,$\mathcal{E}_c$) calcule, à partir des *explications de contradiction* connues au moment de l'appel à cette fonction, la meilleure configuration prometteuse selon le comparateur donné par l'utilisateur. Cette fonction dépend donc du comparateur utilisé. Dans un souci de proximité des solutions proposées, on peut choisir de se baser sur la configuration précédente pour donner la meilleure configuration, c'est pourquoi la configuration courante $\langle A, R \rangle$ est un des paramètres de la fonction `meilleure-configuration`.

- `maj-configuration`(C_1,C_2) permet de transformer la configuration C_1 en la configuration C_2. Cette fonction modifie l'état courant du système de telle sorte qu'il soit *comme si* la configuration C_2 était la configuration de départ. En particulier, les domaines des variables sont modifiés afin de rétablir, si possible, la propriété $\mathcal{P}$. Si on ne le peut pas, les modifications s'arrêtent. Cette fonction dépend elle aussi du domaine de calcul.

L'algorithme de recherche est le suivant :

Algorithme 1 (Recherche en Meilleur d'Abord)

```
        % On part d'une configuration prometteuse C_f devenue
        % P-contradictoire à la suite d'une modification
(1)  début
(2)     tant que C_f est P-contradictoire faire
(3)        ajouter explique-contradiction(C_f) à E_c
(4)        maj-configuration(C_f,meilleure-configuration(C_f,E_c))
(5)     fintantque //C_f est une C-solution //
(6)  fin
```

Exemple 19 (Exemple de fonctionnement) :
Reprenons l'exemple 15 page 43 qui porte sur un ensemble de quatre contraintes $\{c_a, c_b, c_c, c_d\}$ dont les poids sont respectivement $1, 2, 4, 8$ et pour lequel, par hypothèse, $\{c_b, c_d\}$ et $\{c_c, c_d\}$ sont deux ensembles de contraintes $\mathcal{P}$-contradictoires.

- La première configuration testée $abcd$ est $\mathcal{P}$-contradictoire. Supposons que l'on ait : `explique-contradiction`$(abcd) = \{a, b, d\}$, la fonction `meilleure-configuration` fournit alors la configuration $a'bcd$ c'est la meilleure configuration prometteuse selon le comparateur défini pour ce problème.
- Cette configuration est $\mathcal{P}$-contradictoire car elle contient l'ensemble $\{c_b, c_d\}$ que nous avons supposé $\mathcal{P}$-contradictoire page 43. Supposons que `explique-contradiction`$(a'bcd) = \{b, d\}$. On a alors $\mathcal{E}_c = \{\{a, b, d\}, \{b, d\}\}$. La fonction `meilleure-configuration` renvoie alors la configuration $ab'cd$.
- Cette configuration est elle-aussi $\mathcal{P}$-contradictoire. Supposons que `explique-contradiction`$(ab'cd) = \{c, d\}$. La fonction `meilleure-configuration` renvoie alors la configuration $ab'c'd$.
- Cette configuration est $\mathcal{P}$-satisfaisable. C'est donc une C-solution.

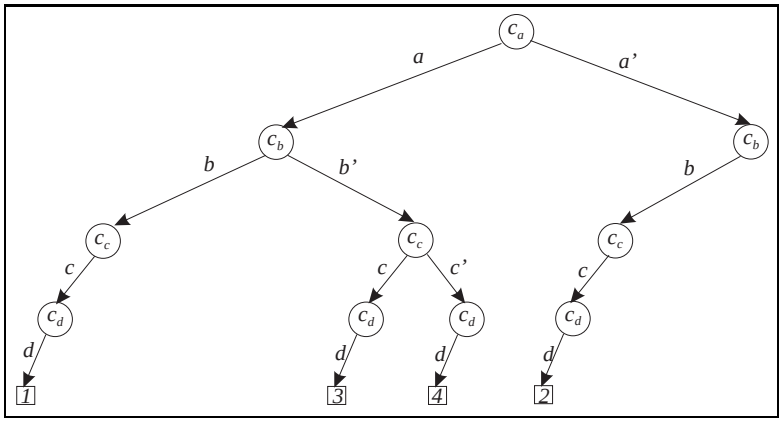

FIG. 4.3 – *Exploration réalisée en meilleur d'abord*

La figure 4.3 représente la portion de l'arbre de recherche qui a été effectivement explorée par cette méthode pour l'exemple 19. Les feuilles sont numérotées dans leur ordre d'exploration. Le tableau 4.1 permet de constater l'effort de calcul réalisé pour effectuer cette recherche.

	Configurations	Introductions	Changements d'État
Observations	4	4	4
Total	4 / 16	4	4

TAB. 4.1 – *Effort de calcul pour l'exploration en meilleur d'abord*

Cette méthode fonctionne sans retour arrière. On est en présence d'un algorithme d'exploration que l'on peut qualifier de type **meilleur d'abord** (*Best First*). Lorsqu'une contrainte est relaxée ou réintroduite, c'est la fonction `maj-configuration` qui se charge des modifications à apporter au système sans provoquer un retour

arrière.

Proposition 5 (Terminaison et Correction)
L'algorithme 1 termine et est correct.

▷ **Preuve :**

- L'algorithme termine parce que :
  1. Le nombre total de configurations existantes est fini : il y en a 2^e où e est le nombre de contraintes du problème courant.
  2. On ne peut passer deux fois par une même configuration. En effet, comme les configurations testées sont toutes prometteuses, elles ne contiennent pas d'explication de contradiction déjà connues, ainsi, seules de nouvelles *explication de contradiction* sont produites, permettant alors d'explorer des configurations non encore testées.
  3. Il existe au moins une configuration $\mathcal{P}$-satisfaisable : la configuration $\langle \varnothing, C_c \rangle$ où toutes les contraintes du problème sont relaxées.
- La correction de l'algorithme est une conséquence directe de la proposition 4 page 45.

◁

Cette méthode peut être utilisée dans le cadre HCLP pour déterminer une solution pour une hiérarchie donnée. Il s'agit alors d'une utilisation dans un cadre statique.

Pour illustrer l'intérêt de notre approche par rapport à des approches classiques, le tableau 4.2 récapitule le nombre de configurations testées, le nombre d'introduction de contraintes et le nombre de changements de statuts pour une contrainte réalisés pour trois techniques : notre recherche (algorithme 1), un backtrack standard et un backtrack *intelligent* utilisant les informations recueillies par la fonction `explique-contradiction` mais réalisant les relaxations par retour arrière. Notons que les deux dernières techniques nécessitent une étape de preuve d'optimalité que nous avons prise en compte dans le tableau proposé.

	Backtrack	Backtrack Intelligent	Meilleur d'Abord
Configurations	8	6	4
Introductions	14	12	4
Changements	12	8	4

TAB. 4.2 – *Comparaisons des techniques d'exploration*

Notons que l'algorithme d'énumération *Dynamic Backtracking* ([Ginsberg, 1993] et section 3.4.3) peut être considéré comme un cas particulier de notre algorithme pour un type précis de contraintes : les contraintes d'affectation de valeur à une variable apparaissant au cours de l'énumération. Nous y reviendrons section 7.3 où nous présentons un modèle d'intégration de l'énumération dans le cadre des domaines finis.

4.3 Sémantique opérationnelle de `maj-configuration`

Dans cette partie, nous présentons une sémantique opérationnelle du changement de configuration utilisé dans notre méthode d'exploration *i.e.* pour la fonction `maj-configuration`. Cette sémantique [Jussien et Boizumault, 1997c] permet de bien observer un des points clés de notre approche : la réutilisation des données collectées au cours de la résolution pour éviter le *thrashing*.

La figure 4.3, illustration graphique de notre approche, ne met pas l'accent sur la réutilisation. Nous donnons maintenant une sémantique opérationnelle du changement de configuration qui illustre les différentes propriétés de notre approche. Cette sémantique est basée sur la transformation [4] de l'arbre de recherche. En effet, comme dans [Fages *et al.*, 1995] l'arbre de recherche est modifié pour identifier les parties qui sont réutilisées lors d'un changement de configuration.

4.3.1 Mise à jour de la configuration courante

Le changement de configuration que nous préconisons comporte trois étapes :

- identification des portions de la branche courante (configuration courante) qui ne seront pas modifiées dans la prochaine configuration et qui sont indépendantes des contraintes relaxées ;
- relaxation des contraintes sélectionnées et propagation de ces relaxations [5] ;
- réintroduction des contraintes relaxées dans la configuration courante mais qui ne le seront plus dans la prochaine.

4.3.2 Modification de l'arbre de recherche

L'idée des transformations repose sur cette constatation : si les contraintes relaxées avaient été les dernières introduites, un backtrack simple aurait suffit. Comme l'ordre d'introduction des contraintes ne modifie en aucune façon l'ensemble des solutions du problème, l'arbre de recherche peut être transformé de telle sorte que toute contrainte dont le statut change (d'active à relaxée et vice-versa) soit placée à la fin de l'arbre (la partie du calcul qui est réexécutée).

Ces modifications permettent alors, d'une part, d'expliciter le comportement opérationnel du changement de configuration, et d'autre part, d'expliciter la partie réutilisée du calcul précédent.

Les opérateurs de transformation

Comme on travaille à chaque instant sur une seule configuration, nos opérateurs de transformation sont définis sur une branche de l'arbre (grâce à l'équivalent entre une branche et une configuration). Ceci nous permet de ne pas gérer explicitement l'arbre de recherche surtout, qu'à partir d'une branche donnée, l'arbre complet est complètement défini.

> **Exemple 20 (Branche et Arbre) :**
> La branche $abcd$ définit complètement l'arbre de la figure 4.2 page 43.

Soit $\bullet$ l'opération de concaténation sur les portions de branches : $ab \bullet cd = abcd$. Nous avons trois opérations de modification de base :

- **Ajout de Contrainte:** Opérateur $\oplus$
  L'ajout d'une contrainte c_c dans l'arbre de recherche courant dont la configuration courante est la branche A est défini par :

$$A \oplus c_c = A \bullet c \tag{4.4}$$

4. Ces opérations de transformations sont similaires à celles présentées dans [Fages *et al.*, 1995] mais elles sont simplifiées puisque dans notre cas nous n'avons pas à gérer les opérations sur les atomes car nous sommes sortis du cadre de la programmation logique.

5. C'est à partir de cette étape que les nouveaux calculs ont lieu. Par exemple, dans le cas des domaines finis, cette propagation consiste à réintroduire dans les domaines des variables les valeurs qui n'ont plus de raisons d'être supprimées.

- **Relaxation de Contrainte:** Opérateur $\ominus$
  La relaxation de la contrainte c_c dans la configuration courante est définie par :
  $$(x \bullet A) \ominus c_c = \left| \begin{array}{ll} A \bullet c' & \text{si } x = c \\ x \bullet (A \ominus c_c) & \text{sinon} \end{array} \right. \tag{4.5}$$
  Cet opérateur peut aussi s'écrire :
  $$A \bullet c \bullet B \ominus c_c = A \bullet B \bullet c' \tag{4.6}$$

- **Réintroduction de Contrainte:** Opérateur $\odot$
  La réintroduction de la contrainte c_c dans la configuration courante est définie par :
  $$(x \bullet A) \odot c_c = \left| \begin{array}{ll} A \bullet c & \text{si } x = c' \\ x \bullet (A \odot c_c) & \text{sinon} \end{array} \right. \tag{4.7}$$
  Cet opérateur peut aussi s'écrire :
  $$A \bullet c' \bullet B \odot c_c = A \bullet B \bullet c \tag{4.8}$$

Lorsqu'on veut réaliser un changement de configuration, la comparaison entre la configuration de départ (branche A) $\mathcal{P}$-contradictoire et la configuration d'arrivée (fournie par `meilleure-configuration`) permet d'identifier deux ensembles : un ensemble de contraintes à relaxer $(c_{k_1}, \ldots, c_{k_n})$ et un ensemble de contraintes à réintroduire $(c_{\ell_1}, \ldots, c_{\ell_m})$. La nouvelle configuration est alors définie par :

$$A \ominus c_{k_1} \cdots \ominus c_{k_n} \odot c_{\ell_1} \cdots \odot c_{\ell_m} \tag{4.9}$$

Dans l'équation 4.9, il est impératif que les relaxations soient réalisées (opérateur $\ominus$) avant les réintroductions (opérateur $\odot$) pour obtenir des propriétés pour ces transformations sur lesquelles nous reviendrons section 4.4.3 page 54.

Illustration

La résolution de l'exemple 19 page 46 conduit aux transformations successives suivantes.

- À partir de la première configuration $abcd$ $\mathcal{P}$-contradictoire, la fonction `meilleure-configuration` donne la configuration $a'bcd$. Il n'y a qu'une contrainte relaxée (c_a) dans cette configuration et l'ensemble des contraintes réintroduites est vide. La transformation est alors :
  $$abcd \ominus c_a = bcda' \tag{4.10}$$
  Cette nouvelle branche définit un nouvel arbre de recherche (*cf.* figure 4.4). Notons qu'une partie de ce nouvel arbre a déjà été explorée (en gras sur le dessin de droite).
  Sur la figure, on peut aussi voir quelle partie de l'arbre a pu être réutilisée sans modification.

- La configuration suivante est $ab'cd$. La transformation est :
  $$bcda' \ominus c_b \odot c_a = cdb'a \tag{4.11}$$
  L'arbre obtenu est représenté figure 4.5.

- $ab'c'd$ est la configuration suivante, c'est aussi la dernière. La transformation est :
  $$cdb'a \ominus c_c = db'ac' \tag{4.12}$$
  L'arbre obtenu est représenté figure 4.6.

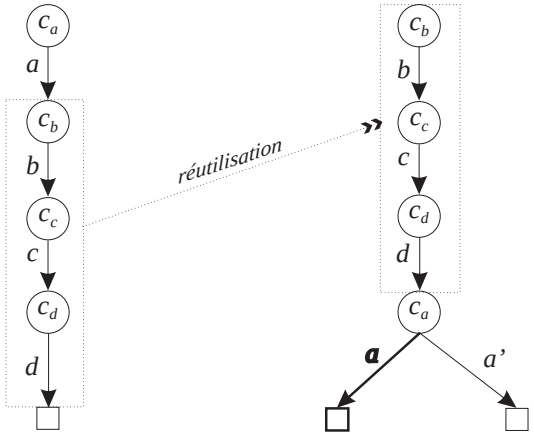

FIG. 4.4 – *Transformations de l'arbre de recherche – Transformation 4.10*

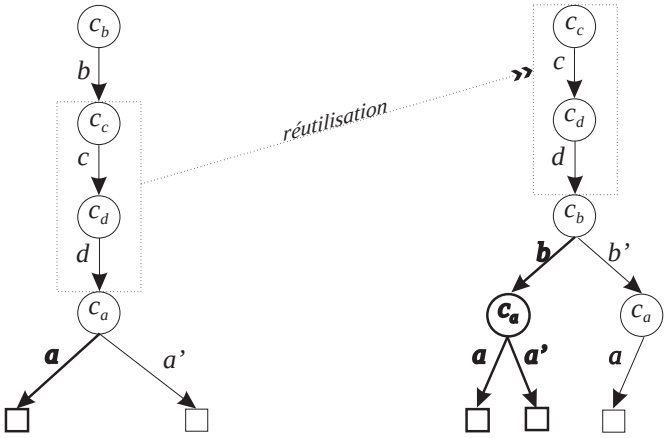

FIG. 4.5 – *Transformations de l'arbre de recherche – Transformation 4.11*

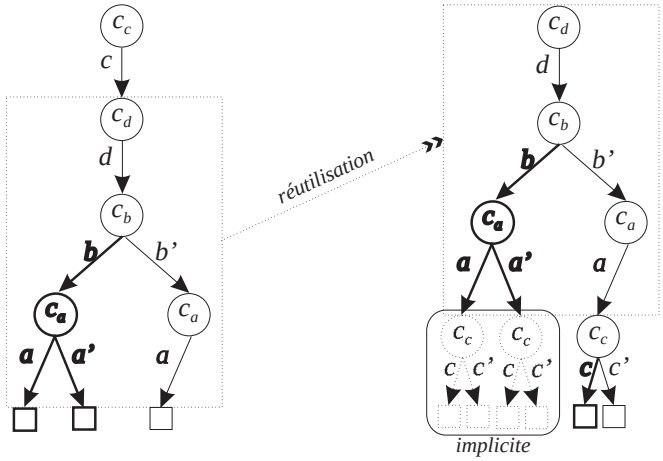

FIG. 4.6 – *Transformations de l'arbre de recherche – Transformation 4.12*

4.3.3 Conclusion

La sémantique opérationnelle présentée ici permet de bien identifier les différents apports de notre méthode :

- exploration originale de l'arbre de recherche (les points remis en cause ne sont pas forcément les plus récents) ;
- réutilisation d'une partie du calcul déjà effectué : tout ce qui est indépendant des points remis en cause (relaxations de contraintes) ;
- simplicité du changement de configuration apportée par la modification de l'arbre de recherche à l'aide des opérateurs définis.

4.4 Spécification de `meilleure-configuration`

La fonction `meilleure-configuration` dépend étroitement du comparateur utilisé. Cette fonction est chargée de déterminer la meilleure, selon le comparateur utilisé, des configurations prometteuses par rapport à un ensemble $\mathcal{E}_c$ d'explications de contradiction. Comme on l'a vu dans la section 4.2.2, cela revient à trouver un recouvrement (*cf.* définition 25) de ces explications, le meilleur possible au sens du comparateur utilisé.

4.4.1 Le problème de recouvrement

Précisons l'hypergraphe pour lequel doit être calculé un recouvrement dont nous avons parlé section 4.2.2.

Définition 26 (Hypergraphe $\mathcal{H}$)
On appelle hypergraphe $\mathcal{H}$ associé à un ensemble d'explications de contradiction $\mathcal{E}_c$, un hypergraphe dont les sommets sont les éléments de $\mathcal{E}_c$ et dont les hyperarêtes sont les contraintes intervenant dans les explications de contradiction $\mathcal{E}_c$. Ainsi une hyperarête associée à une contrainte c répertorie les explications de contradiction dans lesquelles c apparaît. Dans cet hypergraphe, les boucles sont autorisées.

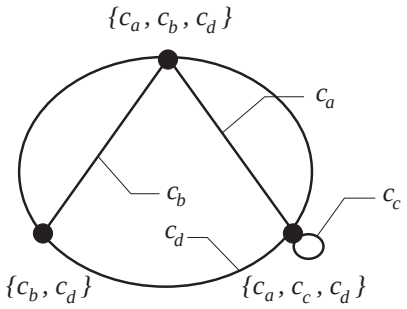

FIG. 4.7 – *Hypergraphe pour le recouvrement*

Exemple 21 (Exemple d'hypergraphe) :
La figure 4.7 représente un tel hypergraphe pour l'ensemble d'explications de contradiction $\{\{c_a, c_b, c_d\}, \{c_a, c_c, c_d\}, \{c_b, c_d\}\}$ (voir exemple 18 page 45). Sur cette figure, les sommets sont les explications de contradictions identifiées jusqu'à présent, et les hyperarêtes sont les contraintes. Ainsi, nous en trouvons deux d'arité 2 (c_a et c_b), une d'arité 1, (la boucle c_c) et une d'arité 3 (c_d).

Le problème général de recouvrement de coût minimum dans un hypergraphe est un problème NP-complet [Gondran et Minoux, 1990]. Le recouvrement que l'on recherche dans le cadre de notre approche est un recouvrement optimal au sens du comparateur. Pour des comparateurs précis, ce problème peut devenir facile. Nous y revenons dans les sections suivantes.

Pour induire le moins de modifications à prendre en compte pour la fonction `maj-configuration`, nous préconisons d'implémenter une version incrémentale de `meilleure-configuration` *i.e.* qui, partant d'une configuration prometteuse pour un ensemble d'explications de contradiction $\mathcal{E}_c$ en calcule une nouvelle lors de l'ajout d'une nouvelle explication de contradiction.

4.4.2 Utilisation d'un comparateur quelconque

Lorsqu'on utilise un comparateur quelconque, il est nécessaire de conserver toutes les explications de contradiction déterminées jusqu'à présent. Le nombre d'explications de contradiction que l'on peut déterminer est potentiellement exponentiel. L'utilisation d'un comparateur quelconque pourrait donc conduire à une occupation en espace prohibitive.

Par ailleurs, l'utilisation d'un comparateur quelconque conduit, comme nous l'avons vu, à la résolution d'un problème NP-complet. On peut envisager deux possibilités :

- Utiliser des heuristiques pour calculer le recouvrement optimal cherché. Le cadre général que nous avons défini ne permet alors plus de déterminer la meilleure configuration $\mathcal{P}$-satisfaisable mais une configuration d'autant «*meilleure*» que l'heuristique utilisée sera bonne. Par exemple, la meilleure heuristique connue pour un problème de recouvrement de coût minimal est présentée dans [Caprara *et al.*, 1996]. Il s'agit d'une méthode basée sur la relaxation lagrangienne[6].

- Utiliser un comparateur pour lequel la détermination d'un recouvrement optimal devient un problème facile.

Nous nous intéressons dans la section suivante à la deuxième possibilité et proposons une classe de comparateur garantissant en plus de la «*facilité*» de résolution du problème de recouvrement d'autres bonnes propriétés.

4.4.3 Les comparateurs *contradiction-local*

On définit une nouvelle classe de comparateurs : les comparateurs `contradiction-local`.

Définition 27 (Comparateur *contradiction-local*)
Soit C_{omp} un comparateur. Soit $\mathcal{E}_c$ un ensemble d'explication de contradiction et $\langle A, R\rangle$ une configuration prometteuse optimale pour C_{omp}. Soit E_n une nouvelle explication de contradiction ($E_n \subset A$).

C_{omp} est dit contradiction-local *si et seulement si toute configuration $\langle A', R'\rangle$ optimale pour C_{omp} recouvrant $\mathcal{E}_c \cup \{E_n\}$ est telle que :*

$$A' \cap R = \varnothing$$

6. La relaxation lagrangienne n'a rien à voir avec la relaxation de contraintes au sens où nous l'entendons. En effet, les méthodes basées sur la relaxation lagrangienne relaxent *temporairement* des contraintes en les prenant en compte dans la fonction objectif car elles permettent d'obtenir un problème plus simple. On peut se reporter à [Minoux, 1983] pour plus de détails.

Autrement dit, il n'y a jamais de retour de contrainte si on utilise un comparateur contradiction-local.

On peut donner une première propriété intéressante pour un comparateur *contradiction-local*.

Proposition 6 (Localité des relaxations)
Lorsque le comparateur utilisé est contradiction-local, *les seules nouvelles contraintes relaxées appartiennent à la nouvelle explication de contradiction.*

> ▷ **Preuve :** Soit C_{omp} un comparateur *contradiction-local*. Soit $\langle A, R \rangle$ une configuration optimale pour C_{omp} recouvrant un ensemble $\mathcal{E}_c$ d'explication de contradiction. Soit E_n une nouvelle explication de contradiction. Soit $\langle A', R' \rangle$ une configuration optimale pour C_{omp} recouvrant $\mathcal{E}_c \cup \{E_n\}$.
>
> Comme C_{omp} est *contradiction-local*, on a $A' \cap R = \varnothing$. Supposons que $\exists c, c \notin E_n, c \in R' \cap A$. Autrement dit, il existe une contrainte c nouvellement relaxée et n'appartenant pas à E_n. Alors, comme il n'y a aucun retour de contrainte ($A' \cap R = \varnothing$), la configuration $\langle A' \cup \{c\}, R' \setminus \{c\} \rangle$ recouvrirait aussi $\mathcal{E}_c \cup \{E_n\}$.
>
> On peut donc remplacer $\langle A', R' \rangle$ par $\langle A' \cup \{c\}, R' \setminus \{c\} \rangle$ car, intuitivement, cette dernière configuration ne peut être «moins bonne» que la première (moins de contraintes sont relaxées). Notons qu'elle ne peut pas non plus être «strictement meilleure» que la première car si tel était le cas, il y aurait une contradiction. Les deux configurations sont donc équivalentes.
>
> La propriété est donc montrée.
>
> ◁

Avec un comparateur *contradiction-local*, on peut donc garantir la pérennité des relaxations de contraintes (il ne sert à rien de réintroduire une contrainte relaxée auparavant) et assurer que la connaissance de la dernière explication de contradiction suffit à garantir le calcul d'une configuration prometteuse optimale. Cette dernière propriété est intéressante car elle permet de ne pas conserver l'ensemble des explications de contradiction.

On peut énoncer le résultat suivant sur les opérations de transformations de la section 4.3.

Théorème 1 (Propriétés des transformations)
Pour tout comparateur contradiction-local, *l'utilisation des transformations d'arbre permet d'assurer que si la configuration courante est $\mathcal{P}$-contradictoire, la prochaine configuration explorée sera à droite de la configuration courante dans l'arbre de recherche courant.*

> ▷ **Preuve :** Comme un comparateur *contradiction-local* garantit, par définition, que le retour d'une contrainte n'apporte rien au niveau du comparateur, toute configuration explorée sera de la forme :
>
> $$A \bullet c' \bullet B$$
>
> où A est composé uniquement de contraintes actives et B de contraintes relaxées. Ainsi, toute position à gauche de la configuration courante (passage d'un élément de B à un état actif) ne peut conduire qu'à une configuration $\mathcal{P}$-contradictoire si la configuration courante est $\mathcal{P}$-contradictoire. Sinon, il existe une configuration équivalente au regard du comparateur à droite de la configuration courante : il suffit de laisser cet élément relaxé car cela ne change rien au regard du comparateur. ◁

C'est en vertu de ce théorème qu'une partie de la figure 4.6 page 51 est marquée comme implicitement explorée.

Parmi les comparateurs que nous avons déjà présenté, il en est un qui est *contradiction-local* : C_{MM}. Pour le montrer, nous allons caractériser une solution optimale pour C_{MM}.

Rappelons qu'une configuration optimale pour C_{MM} est une configuration pour laquelle la plus importante des contraintes relaxées est la moins importante possible. On introduit une nouvelle notion : le minimum d'une explication.

Définition 28 (Minimum d'une explication)
On appelle ***minimum*** *d'une explication E et on note* $\min(E)$ *la contrainte la moins importante de l'explication E. Si celle-ci n'est pas unique, on sélectionne indifféremment une des candidates.*

On peut maintenant énoncer le théorème suivant qui permet de caractériser une configuration optimale pour C_{MM} et recouvrant un ensemble d'explications de contradiction.

Théorème 2 (Caractérisation d'une solution optimale pour C_{MM})
L'ensemble $Sol = \{c \in A \cup R \mid \exists E \in \mathcal{E}_c, c = \min(E)\}$ est un recouvrement optimal pour C_{MM} de l'ensemble $\mathcal{E}_c$.

▷ **Preuve :**

Sol est bien un recouvrement (chaque explication de contradiction contient un élément de *Sol*). Nous allons montrer par récurrence qu'il n'en existe pas de meilleur selon le comparateur C_{MM}.

- Si $\mathcal{E}_c = \varnothing$, $\langle C, \varnothing \rangle$ est bien la meilleure configuration au sens de C_{MM} (*cf.* section 2.4.2).
- Si $\mathcal{E}_c = \{E\}$, il est indispensable de relaxer au moins une contrainte, on ne peut trouver mieux (au sens de C_{MM} comme au sens de n'importe quel comparateur) que de relaxer $\min(E)$.
- Supposons que $\mathcal{E}_c = S$ et que le théorème soit vérifié pour S *i.e.* $R_S = \{c \in A \cup R \mid \exists E \in S, c = \min(E)\}$ est un recouvrement optimal au sens de C_{MM}. Soit E_n une nouvelle explication de contradiction. Montrons que $R_S \cup \{\min(E_n)\}$ est un recouvrement optimal pour $S \cup \{E_n\}$ selon C_{MM}. Soit k le niveau de la contrainte relaxée la plus importante dans R_S. Soit ℓ le niveau de $\min(E_n)$. Deux cas se présentent :
  1. ℓ est moins important ou d'importance égale à k. Le recouvrement obtenu est donc bien optimal pour $S \cup \{E_n\}$ car la relaxation de ℓ on ne peut pas espérer ne pas avoir à relaxer la contrainte de niveau k du fait de la relaxation de $\min(E_n)$ car R_S est un recouvrement optimal de S.
  2. ℓ est plus important que k. On ne peut pas faire mieux car pour ne pas relaxer la contrainte de niveau ℓ, comme il est impératif de recouvrir E_n, on serait obligé de relaxer une contrainte encore plus importante ou d'importance égale (ℓ est le niveau de $\min(E_n)$).

Le théorème est donc démontré par récurrence. ◁

Théorème 3 (C_{MM} est *contradiction-local*)
Le comparateur C_{MM} est contradiction-local.

▷ **Preuve :** Soit $\langle A, R\rangle$ une configuration optimale pour C_{MM} recouvrant $\mathcal{E}_c$. Soit E_n une nouvelle explication de contradiction. Soit $c = \min(E_n)$. Par le fait du théorème 2, $\langle A \setminus \{c\}, R \cup \{c\}\rangle$ est une configuration optimale pour C_{MM} recouvrant $\mathcal{E}_c \cup \{E_n\}$. On a bien $(A \setminus \{c\}) \cap R = \varnothing$. C_{MM} est donc bien *contradiction-local*. ◁

Les comparateurs *contradiction-local* sont intéressants parce qu'ils permettent de rendre facile le problème de recouvrement de la fonction `meilleure-configuration`. En effet, la détermination d'une solution optimale ne dépend que de la dernière explication de contradiction recontrée. Ils permettent à notre mécanisme de transformation d'arbre d'avoir de bonnes propriétés et ils assurent la pérennité des relaxations.

Un comparateur *contradiction-local* paraît tout à fait adapté à un fonctionnement réel d'un système de relaxation de contrainte. En effet, il s'accommode d'une connaissance parcellaire des préférences associées aux contraintes d'un problème. Ainsi, un comparateur *contradiction-local* n'a besoin de connaître à un moment donné que les préférences associées aux contraintes de la dernière explication de contradiction. Mieux encore, dans le cas de C_{MM} il lui suffit d'être capable d'identifier la contrainte la moins importante : il s'agit alors d'une connaissance relative des préférences.

La force de ces comparateurs (localité de la comparaison) est à double tranchant. En effet, cette localité ne permet pas d'avoir une vision globale de la solution obtenue et peut conduire à des configurations peu satisfaisantes pour l'utilisateur même si elle sont optimales pour le comparateur considéré. Ainsi, dans le cas de C_{MM}, on peut être amené à relaxer toutes les contraintes d'un niveau de la hiérarchie avant de se rendre compte qu'il suffisait de relaxer une seule contrainte plus importante. C'est le phénomène de «*noyade*».

4.5 Conclusion

Nous avons présenté dans ce chapitre une méthode générale de recherche en meilleur d'abord de C-solution. Cette recherche est basée sur trois fonctions : `explique-contradiction`, `meilleure-configuration` et `maj-configuration`.

Nous avons donné une sémantique opérationnelle de la dernière en terme de transformations de l'arbre de recherche. Nous avons spécifié le fonctionnement de la deuxième fonction et plus particulièrement fourni des résultats pour les comparateurs *contradiction-local*. Nous avons de plus exhibé les bonnes propriétés de cette classe de comparateurs.

Dans les chapitres suivants, nous allons montrer comment spécifier les fonctions `explique-contradiction` et `maj-configuration` de manière plus précise pour les domaines de contraintes dont la résolution utilise un algorithme de point fixe. Nous introduirons pour ces spécifications un nouvel outil : le système de maintien de déduction. Mais avant, nous présentons une instance directe de notre approche : relax($\mathbb{Q}$, Lin, $\mathcal{P}_{sol}$).

Chapitre 5

Systèmes linéaires sur les rationnels : relax($\mathbb{Q}$, Lin, $\mathcal{P}_{sol}$)

Sommaire

DANS CE CHAPITRE, nous présentons l'adaptation de notre méthode de recherche au cas des contraintes linéaires sur les rationnels. De tels systèmes de contraintes peuvent être résolus en utilisant des méthodes classiques de résolution de problèmes linéaires (simplexe, Gauss, ...). Nous présentons ici brièvement le lien entre l'algorithme du simplexe qui est une méthode d'optimisation en programmation linéaire et la réalisabilité d'un système de contraintes linéaires sur les rationnels. Ensuite, nous montrons comment les fonctions `explique-contradiction` et `maj-configuration` peuvent être spécialisées en rappelant brièvement les travaux de de Backer [1995] et de Jaakola [1990]. Ce court chapitre se présente donc comme un exemple illustratif de l'intérêt de notre approche et de la facilité avec laquelle on peut obtenir une instance pour traiter les contraintes linéaires sur les rationnels.

5.1 Présentation

5.1.1 Définitions

On se donne un système S de contraintes linéaires composé d'inégalités $A \cdot x \leqslant b$. A est une matrice rationnelle $m \times n$ de plein rang[1]. x est un n-vecteur de variables rationnelles quelconques et b un m-vecteur de constantes rationnelles. Tout système de contraintes peut se ramener à ce cas par l'introduction de nouvelles variables.

On s'intéresse à la solubilité d'un tel système.

Définition 29 (Solubilité)
Un système S de contrainte linéaires $A \cdot x \leqslant b$ est dit ***soluble*** *s'il existe un vecteur u de $\mathbb{Q}^n$ tel que $A \cdot u \leqslant b$.*

Dans notre cadre général, on a donc

- un domaine de calcul : $\mathbb{Q}$ l'ensemble des rationnels ;
- un ensemble de variables regroupées dans le vecteur x ;
- dont les domaines sont l'ensemble $\mathbb{Q}$ lui-même ;
- et un ensemble de contraintes représenté par S.

La propriété $\mathcal{P}$ devant être satisfaite par toute configuration est alors $\mathcal{P}_{sol}$ (*cf.* définition 7 page 17), on parle aussi de réalisabilité.

5.1.2 Réalisabilité d'un système linéaire

Pour un système S donné, on introduit m variables dites **variables d'écarts** $(x_{n+1}, \ldots, x_{n+m})$ toutes positives permettant d'écrire S sous la forme :

$$\left\{ \sum_{j=1}^{n} a_{ij} x_j + x_{n+i} = b_i \right\}_{i \in \{1..m\}} \tag{5.1}$$

On peut écrire alors cet ensemble de contraintes de la façon suivante, dite sous **forme standard**.

1. Le rang d'une matrice est défini comme étant la taille du plus grand sous-déterminant non nul extrait de cette matrice. Une matrice est de plein rang si son rang est le même que la plus petite de ses dimensions. On peut toujours se ramener à une matrice de plein rang en supprimant les lignes (colonnes) redondantes.

$$\left\{ x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \right\}_{i \in \{1..m\}} \tag{5.2}$$

D'après l'équation 5.2, si les b_i sont positifs, alors $x_{n+1} = b_i$ pour tout $i \in \{1..m\}$ et $x_j = 0$ pour tout $j \in \{1..n\}$ est un élément de S qui est donc non vide.

On appelle **base** toute sous-matrice carrée régulière[2] $m \times m$ extraite de A^*. A^* est composée de A à laquelle on accole la matrice identité $m \times m$ correspondant aux variables d'écart introduites dans la forme standard (équation 5.2).

En permutant les colonnes de A^*, on peut toujours mettre A^* sous la forme $A^* = (B, N)$ où N désigne les colonnes de A^* qui ne sont pas dans la base B. De même, on partitionne x en (x_B, x_N).

On appelle **solution de base** le cas où l'on considère $x_N = 0$ et ainsi on peut écrire[3] $x_B = B^{-1} \cdot b$. Si $B^{-1} \cdot b$ est positif alors S est non vide.

Pour étudier la solubilité d'un système de contraintes, dans le cas où au moins un b_i est strictement négatif, il faut trouver une base B permettant d'obtenir $B^{-1} \cdot b$ positif (pour les lignes correspondant à des variables d'écart) ou de montrer qu'il n'en existe pas.

Pour cela, on se propose d'écrire un programme linéaire déduit de S tel que la structure de la solution optimale permette de savoir si S est vide et dans le cas contraire, exhiber une solution réalisable de S.

Soit l'ensemble S' déduit de S ainsi :

$$\left\{ \sum_{j=1}^{n} a_{ij} x_j - x_0 \leqslant b_i \right\}_{i \in \{1..m\}} \tag{5.3}$$

x_0 est une nouvelle variable rationnelle. Soit le programme linéaire $\min_{\{x \in S'\}} x_0$. Soit z la valeur à l'optimum de ce programme linéaire.

Lemme 1

$$S \text{ non vide } \Longrightarrow z \leqslant 0$$

▷ **Preuve :** Soit $(x_1, \ldots, x_n) \in S$. En choisissant $x_0 = 0$, $(x_0, x_1, \ldots, x_n)$ est un élément de S'. z étant la valeur minimale de x_0, on a : $z \leqslant 0$. ◁

Lemme 2

$$S \text{ vide } \Longrightarrow z > 0$$

▷ **Preuve :** Supposons $z \leqslant 0$, prenons $(z, x_1, \ldots, x_n)$ un élément optimal de S'. D'après les systèmes de contraintes traités et comme $z \leqslant 0$, $(x_1, \ldots, x_n)$ est élément de S ce qui est contradictoire d'où le lemme. ◁

Proposition 7

$$S \text{ vide } \Longleftrightarrow z > 0$$

▷ **Preuve :** C'est une conséquence des deux lemmes précédents, c'est aussi un résultat classique de programmation linéaire. ◁

2. Régulière est équivalent à inversible. Une telle matrice existe puisque A^* est de plein rang.
3. B^{-1} existe puisque B est carrée régulière.

On a donc trouvé le programme linéaire que l'on cherchait. En utilisant la méthode du simplexe, si l'optimum obtenu est négatif ou nul, S n'est pas vide ; sinon le système S est contradictoire.

Nous avons montré jusqu'ici qu'il existait un programme linéaire permettant de décider de la réalisabilité d'un système de contraintes donné. Nous utiliserons par la suite un autre programme linéaire obtenu à partir du problème initial : le quasi-dual. Nous pourrons ainsi passer d'un problème à l'autre sans aucune difficulté.

5.1.3 Vérification de $\mathcal{P}_{sol}$

Écrivons le programme dual du programme linéaire défini sur S' (équation 5.3).

$$\begin{array}{l} \max \sum_{i=1}^{m} \lambda_i b_i \\ \qquad \begin{array}{rcl} \{\sum_{i=1}^{m} a_{ij}\lambda_i & = & 0\}_{j\in\{1..n\}} \\ \sum_{i=1}^{m}(-\lambda_i) & = & 1 \\ \forall i \in \{1..m\}\lambda_i & \leqslant & 0 \end{array} \end{array} \tag{5.4}$$

Ce qui peut aussi s'écrire :

$$\begin{array}{l} -\min \sum_{i=1}^{m} \lambda_i b_i \\ \qquad \begin{array}{rcl} \{\sum_{i=1}^{m} a_{ij}\lambda_i & = & 0\}_{j\in\{1..n\}} \\ \sum_{i=1}^{m} \lambda_i & = & 1 \\ \forall i \in \{1..m\}\lambda_i & \geqslant & 0 \end{array} \end{array} \tag{5.5}$$

On obtient au signe des λ_i près le quasi-dual de S :

Définition 30 (Quasi Dual)
*On définit le polyèdre Q (**quasi dual**) à partir de S : $\left\{\sum_{j=1}^{n} a_{ij}x_j \leqslant b_i\right\}_{j\in\{1..n\}}$ de la façon suivante :*

$$\begin{array}{rcl} \{\sum_{i=1}^{m} a_{ij}\lambda_i & = & 0\}_{j\in\{1..n\}} \\ \sum_{i=1}^{m} \lambda_i & = & 1 \\ \forall i \in \{1..m\}\lambda_i & \geqslant & 0 \end{array} \tag{5.6}$$

On a le théorème suivant (rappelé dans [de Backer, 1995] et dans [de Backer et Béringer, 1991b]) :

Théorème 4 ([Lassez, 1990])
Soit $M = \min \sum_{i=1}^{m} b_i\lambda_i$ avec $\lambda \in Q$. Alors :

- *Si $M \geqslant 0$, S n'est pas vide.*
- *Si $M < 0$, S est incohérent.*

▷ **Preuve :** Cela découle directement des résultats classiques de dualité [Minoux, 1983]. En effet, on a : $-\min \sum_{i=1}^{m} \lambda_i b_i = z = -M$. D'où le théorème. ◁

Ce nouveau programme linéaire est équivalent à celui de la section précédente, comme on vient de le voir mais, il va nous être plus utile dans le cadre d'une utilisation dans notre système de relaxation de contraintes.

5.2 Instance du schéma général de relaxation

Pour instancier notre schéma général de système de relaxation de contraintes, il suffit de spécifier les deux fonctions `explique-contradiction` et `maj-configuration`. En effet, rappelons que la fonction `meilleure-configuration` est indépendante du domaine de calcul.

5.2.1 Fournir une explication de contradiction

Lorsque S n'est pas cohérent, M (du théorème 4) est calculé en même temps qu'un sommet du quasi-dual Q sur lequel le minimum $M < 0$ est atteint, et de coordonnées $\lambda = (\lambda_1, \ldots, \lambda_n)$.

Théorème 5 (Explication de contradiction pour S)
Un sous-ensemble de S désigné par ses indices $I \subset \{1..m\}$ est une explication de contradiction[4] *si et seulement si il existe un sommet $\Lambda = \{\Lambda_1, \ldots, \Lambda_n\}$ du polyèdre Q tel que :*

$$\sum_{i=1}^{m} b_i \Lambda_i < 0 \text{ et } \Lambda_i \neq 0 \Leftrightarrow i \in I$$

▷ **Preuve :** Ce théorème présenté dans [Béringer et de Backer, 1990] est complètement démontré dans [de Backer, 1995]. ◁

Ainsi, pour une valeur optimale de M atteinte au sommet λ de Q, les contraintes correspondant aux coordonnées non nulles de λ forment une *explication de contradiction.*

On peut également identifier une explication de contradiction à partir du problème primal[5]. Si S est de la forme $\{A \cdot x \leqslant b\}$, la propriété $\mathcal{P}_{sol}$ est testée en résolvant le programme linéaire défini sur S' (*cf.* équation 5.3). Comme on l'a vu ce problème est le dual du quasi-dual au signe près. On a $\min x_0 = -\min \lambda \cdot b$.

En ajoutant les variables d'écart $s = (s_1, \ldots, s_m)$ le problème devient $\min x_0$ sur $A \cdot x + I \cdot s - 1 \cdot x_0 = b$. Dans ce cas, chaque λ_i du quasi dual correspond à un s_i du problème primal. Quand S est insoluble, x_0 atteint une valeur minimale positive. Le fait qu'un λ_i soit non nul correspond alors au fait que la variable d'écart s_i correspondante est hors-base (donc nulle). On obtient alors le résultat suivant.

Théorème 6 (Problème primal et explication de contradiction)
Si $\min x_0$ sur $A \cdot x - 1 \cdot x_0 \leqslant b$ est strictement positif, alors les contraintes dont les variables d'écart associées sont hors-base dans la solution optimale forment une explication de contradiction.

▷ **Preuve :** *cf.* [de Backer, 1995]. ◁

Nous venons donc de voir deux façons de spécifier la fonction `explique-contradiction`.

5.2.2 Effectuer la mise à jour de configuration

Comme l'outil de résolution utilisé est le simplexe, il faut, ici, être capable d'ajouter ou de retirer incrémentalement des contraintes. C'est ce que font les simplexes incrémentaux traditionnellement employés dans les langages de programmation logique avec contraintes.

Nous ne rentrerons pas ici dans les détails d'un simplexe incrémental. En fait, un simplexe incrémental représente les contraintes comme le fait un simplexe classique. En particulier, à une contrainte donnée correspond une unique variable d'écart dans le tableau simplexe correspondant.

On a déjà vu comment il était possible de savoir si un système de contraintes linéaires était réalisable. L'intégration d'une contrainte ne pose problème que si le tableau simplexe obtenu après l'intégration de la nouvelle contrainte n'est plus

4. *Ce théorème a été adapté à nos définitions, dans [de Backer, 1995], une explication de contradiction est appelé* incohérent.

5. Origine du dual.

réalisable (la propriété $\mathcal{P}_{sol}$ n'est pas vérifiée par la solution de base courante). Il faut essayer de retrouver une base réalisable : on se retrouve en fait dans le cas de départ.

Pour supprimer les effets d'une contrainte, on applique tout simplement la technique employée pour réaliser un backtrack dans un simplexe incrémental : si la variable d'écart associée à la contrainte à supprimer est «en base» il suffit de supprimer la ligne concernant la contrainte (la contrainte est ainsi tout simplement relaxée), sinon, on change de base pour la faire entrer en base (il faut bien sûr enlever la contrainte implicite de positivité associée à cette variable d'écart afin qu'il n'y ait plus de contradiction).

Pour plus de détails, on pourra se reporter à [Jaakola, 1990] qui explique complètement comment réaliser l'implémentation d'un simplexe incrémental dans le cas de CLP($\mathcal{R}$).

Notons enfin que les travaux de Holzbaur *et al.* [1996], spécifiant une version du système IHCS [Menezes *et al.*, 1993] pour les contraintes linéaires sur les rationnels, apparaissent comme une instance de notre approche au cas des contraintes linéaires. C'est d'ailleurs le seul cas où le système IHCS et notre approche ont des performances similaires[6].

5.3 Conclusion

Nous avons montré ici combien il était facile dans le cadre des contraintes linéaires sur les rationnels de proposer un système de relaxation de contraintes basé sur notre approche. De plus, on constate que cette intégration à notre approche se fait sans surcoût car toute l'information nécessaire est contenue dans le tableau simplexe maintenu au cours de la résolution.

6. En effet, notre mécanisme d'explication dans le cas des domaines finis est plus précis que celui d'IHCS (*cf.* chapitre 7).

Chapitre 6

Maintien de déduction pour la réduction de domaine

Sommaire

DANS CE CHAPITRE, nous nous intéressons à la spécialisation des fonctions `explique-contradiction` et `maj-configuration` lorsque le solveur utilisé travaille par réduction de domaines (domaines finis, intervalles, ...). Les domaines des variables considérées sont ici des ensembles finis de valeurs [1].

L'idée principale d'une telle spécialisation est d'enregistrer une «*trace* intelligente» des calculs effectués. Cette trace permet d'*expliquer* l'état de chaque domaine à chaque instant et de *défaire* les effets passés d'une contrainte en cas de relaxation. Nous appelons l'ensemble des informations ainsi enregistrées et le mécanisme de gestion associé un système de maintien de déduction.

Plus précisément, nous présentons dans ce chapitre un algorithme [2] général de réduction de domaine [Jussien et Boizumault, 1996a; Jussien et Boizumault, 1996b; Jussien et Boizumault, 1996c]. Nous présentons ensuite notre système de maintien de déduction [3] et précisons les informations enregistrées. Enfin, nous montrons comment ce système de maintien de déductions peut être utilisé pour spécialiser les fonctions `explique-contradiction` et `maj-configuration`.

6.1 Techniques de résolution par réduction de domaines

On cherche dans cette partie à généraliser les techniques de réduction employées pour les CSP [Tsang, 1993] et les CSP numériques [Lhomme, 1993]. Nous définissons ainsi un cadre général pour les problèmes définis par des contraintes et résolus par un algorithme de point fixe.

6.1.1 Présentation

Nous introduisons d'abord le cadre général d'un algorithme de point fixe pour résoudre des CSP. Nous utilisons pour cela un cadre déjà utilisé dans [Benhamou et Older, 1992]. Dans la suite nous appellerons indifféremment *relation* ou *contrainte* toute relation n-aire.

Nous présentons les notions de fonction d'approximation et de fonction de réduction. La première permet de donner un sur-ensemble d'un ensemble donné et la deuxième permet de retirer d'un ensemble donné, les valeurs ne vérifiant pas une relation. Ce sont les briques de base de l'algorithme de réduction que nous présenterons ensuite.

Approximation

Soit un ensemble D. On note $\mathcal{P}(D)$ l'ensemble des parties de D. Une fonction d'approximation est définie de la façon suivante :

Définition 31 (Fonction d'approximation)
Soit apx une fonction définie de $\mathcal{P}(D)$ dans $\mathcal{P}(D)$. La fonction apx est une ***approximation*** *sur D si et seulement si les 4 propriétés suivantes sont vérifiées :*

1. $apx(\varnothing) = \varnothing$

1. C'est le cas aussi pour les intervalles puisque les représentations machines utilisées sont basées sur les nombres flottants. Nous y revenons chapitre 8.

2. Les algorithmes que nous proposons dans ce chapitre sont adaptés à des propriétés comme $\mathcal{P}_{ac}$ (consistance d'arc) et $\mathcal{P}_{bc}$ (b-consistance d'arc). Ces algorithmes peuvent se généraliser en cas d'utilisation d'une autre propriété.

3. Le lecteur averti constatera que différentes notions présentées dans ce chapitre ont leur équivalent (sous un autre nom) dans le domaine des TMS [Doyle, 1979] ou des ATMS [de Kleer, 1986a]. Nous avons fait le choix de ne pas réutiliser ce vocabulaire. Nous avons préférer définir un vocabulaire plus proche de celui de l'algorithme *Dynamic Backtracking* [Ginsberg, 1993] pour mettre en évidence les liens existants.

2. $\forall \rho \in \mathcal{P}(D), \rho \subset apx(\rho)$

3. $\forall \rho, \rho' \in \mathcal{P}(D), \rho \subset \rho' \Rightarrow apx(\rho) \subset apx(\rho')$

4. $\forall \rho \in \mathcal{P}(D), apx(apx(\rho)) = apx(\rho)$

Une fonction d'approximation permet de donner une «*valeur approchée*» d'un ensemble donné.

On peut étendre une approximation sur un ensemble D à D^n de la façon suivante (reprise de [Benhamou et Older, 1992]) :

Définition 32 (Extension de l'approximation)
Soit apx une approximation sur D. Soit ρ une relation sur D^n. Alors, une fonction d'approximation étendue, notée apx_e, est définie de la façon suivante :

$$apx_e(\rho) = apx(\pi_1(\rho)) \times \ldots \times apx(\pi_n(\rho))$$

où $\pi_i(\rho)$, la i^{eme} projection de ρ, est définie de la façon suivante :

$$\pi_i(\rho) = \{x_i \in D \mid (\exists (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \in D^{n-1}), (x_1, \ldots, x_n) \in \rho\}$$

Une telle extension est bien une approximation [Benhamou et Older, 1992].

Par souci d'allégement des notations, nous noterons, dans la suite de ce chapitre, de la même façon une fonction d'approximation et son extension lorsque le contexte permettra de faire la différence.

On peut composer deux relations en utilisant la proposition suivante.

Proposition 8 (Composition)

$$apx(\rho \cup \rho') = apx(apx(\rho) \cup apx(\rho')) \tag{6.1}$$

$$apx(\rho \cap \rho') \subset apx(\rho) \cap apx(\rho') \tag{6.2}$$

Réduction

Une fois défini le concept de fonction d'approximation, on peut introduire la notion de fonction de réduction.

Soit une relation ρ sur un ensemble D. On définit une fonction de réduction de la façon suivante :

Définition 33 (Fonction de réduction)
*Une **fonction de réduction** rdx_ρ pour la relation ρ est une fonction de $\mathcal{P}(D^n)$ vers $\mathcal{P}(D^n)$ vérifiant les propriétés suivantes pour tout élément $u \in \mathcal{P}(D^n)$:*

1.	$rdx_\rho(u) \subset u$	Contraction
2.	$rdx_\rho(rdx_\rho(u)) = rdx_\rho(u)$	Idempotence
3.	$u \subset v \Longrightarrow rdx_\rho(u) \subset rdx_\rho(v)$	Monotonie
4.	$rdx_\rho(u) \cap \rho = u \cap \rho$	Correction

Proposition 9 (Réduction et Approximation)
Soit apx une fonction d'approximation. La fonction suivante est une fonction de réduction pour ρ :

$$rdx(u) = apx(u \cap \rho)$$

On la notera $\overline{\rho}_{apx}$.

▷ **Preuve :** *cf.* [Benhamou et Older, 1992]. ◁

Une fonction de réduction pour ρ permet de retirer d'un ensemble donné les valeurs non compatibles avec ρ.

6.1.2 Un algorithme de réduction

Nous proposons maintenant un algorithme utilisant les fonctions de réduction associées aux relations. Ces relations sont définies sur des variables de l'ensemble $\{v_1, \ldots, v_n\}$ prenant leurs valeurs chacune dans D. On note $var(\rho)$ les variables concernées par la relation ρ. Nous ne nous préoccupons pas ici d'arc-consistance ni d'aucune autre propriété pour l'instant. L'algorithme que nous proposons ici est en fait une généralisation de l'algorithme AC5 [Van Hentenryck *et al.*, 1992] qui était déjà un cadre général pour le maintien de l'arc-consistance. Nous l'avons simplement repris en remplaçant les références à l'arc-consistance (fonctions ArcCons et LocalArcCons) par des références à une propriété quelconque[4].

Considérons l'algorithme suivant :

Algorithme 2 (Algorithme basé sur la réduction)
Soit A un ensemble de relations et Q une file. $D_1, \ldots, D_n$ sont les domaines des variables.

```
point-fixe(in A; inout {D_1,...,D_n})
(1)  début
(2)     initialiser(Q)
(3)     pour tout ρ ∈ A faire
(4)        Δ ← prop(D_1,...,D_n,ρ)
(5)        pour i de 1 à n faire
(6)           si Δ_i ≠ ∅ alors
(7)              enqueue(Q,(i,Δ_i,ρ))
(8)              D_i ← D_i \ Δ_i
(9)           finsi
(10)       finpour
(11)    finpour
(12)    tant que Q ≠ ∅ faire
(13)       (i,S,ρ) ← dequeue(Q)
(14)       pour tout ρ' ∈ {θ ∈ A \ ρ | v_i ∈ var(θ)} faire
(15)          Δ ← localprop(D_1,...,D_n,ρ',i,S)
(16)          pour j de 1 à n sauf i faire
(17)             si Δ_j ≠ ∅ alors
(18)                enqueue(Q,(j,Δ_j,ρ'))
(19)                D_j ← D_j \ Δ_j
(20)             finsi
(21)          finpour
(22)       finpour
(23)    fintantque
(24) fin
```

avec les primitives usuelles sur les files `initialiser`, `dequeue` *et* `enqueue`.
Les fonctions `prop` *et* `localprop` *sont définies de la façon suivante :*

- `prop`$(D_1, \ldots, D_n, \rho)$ *est l'effet provoqué par la relation ρ sur les domaines D^n* i.e. *les ensembles de valeurs retirées des domaines par l'opération de réduction. On a :*

$$\text{prop}(D_1, \ldots, D_n, \rho) = D_1 \times \ldots \times D_n \ominus \overline{\rho}_{apx}(D_1 \times \ldots \times D_n)$$

pour une fonction d'approximation apx donnée[5]*. Notons que les*

4. L'algorithme que l'on obtient alors est parfaitement adapté à toute technique de réduction basée sur l'exploration des «arcs» (contraintes) d'un système de contraintes comme pour l'arc-consistance ou la B-consistance d'arc. Lorsqu'on veut maintenir une propriété plus forte, on peut redéfinir l'algorithme à utiliser de la même manière qu'on le fait ici pour AC5.

5. *L'opérateur $\ominus$ correspond ici à un différence ensembliste si les produits cartésiens sont représentés par l'ensemble des tuples présents.*

modifications sont limitées aux variables concernées par la fonction `prop`.

- `localprop`$(D_1, \ldots, D_n, \rho, i, S)$ *est l'effet provoqué par* ρ *sur les domaines* D^n *suite au retrait de* S *de* D_i. *On a* $S \cap D_i = \varnothing$. *On définit deux ensembles* Δ' *et* Δ'' *qui donnent un certain degré de liberté sur la réponse de* `localprop`. Δ' *correspond aux réductions réellement dues au retrait de* S, *alors que* Δ'' *correspond à une plus forte réduction ne tenant pas compte de* S. *Plus précisément, on pose :*

$$\Delta' = \overline{\rho}_{apx}(D_1, \times \ldots \times D_i \cup S \times \ldots D_n) \ominus \overline{\rho}_{apx}(D_1 \times \ldots \times D_n)$$

et

$$\Delta'' = D_1 \times \ldots \times \ldots \times D_n \ominus \overline{\rho}_{apx}(D_1 \times \ldots \times D_n)$$

On a :

$$\Delta' \subseteq \text{localprop}(D_1, \ldots, D_n, \rho, i, S) \subseteq \Delta''$$

Notons qu'ici aussi, les modifications sont en fait limitées aux variables concernées par la fonction `localprop`.

Proposition 10 (Propriétés de l'algorithme 2)

1. *L'algorithme termine.*
2. *L'algorithme atteint un point fixe pour les relations considérées :*

$$\forall \rho \in A, \forall u \in \mathcal{P}(D^n), \overline{\rho}_{apx}(u) = u$$

où u *est la sortie de* `point-fixe` i.e. $D_1 \times \ldots \times D_n$.

▷ **Preuve :**

1. Les domaines des variables sont des ensembles finis. Comme une fonction de réduction est contractante, l'algorithme termine.
2. - Dans le cas où `localprop` renvoie l'ensemble Δ'' cette propriété est évidente car :

$$\Delta'' = \varnothing \Longleftrightarrow \overline{\rho}_{apx}(D_1 \times \ldots \times D_n) = D_1 \times \ldots \times D_n$$

   Ce qui prouve aussi que la propriété est aussi vérifiée pour `prop` puisque cette fonction renvoie en fait l'ensemble Δ''.
   - Dans le cas où `localprop` renvoie l'ensemble Δ' et si on note $\rho\downarrow_\ell$ la projection de ρ sur sa composante ℓ. On a :

$$\forall \ell, \Delta'\downarrow_\ell = D_\ell \ominus (\overline{\rho}_{apx}(D_1 \times \ldots \times D_n)\downarrow_\ell \cap \overline{\rho}_{apx}(D_1 \times \ldots \times S \times \ldots \times D_n)\downarrow_\ell)$$

$$\begin{aligned}
\Delta' = \varnothing \iff & \forall \ell \neq i \quad (\overline{\rho}_{apx}(D_1 \times \ldots \times D_n)\downarrow_\ell \cap \\
& \qquad \overline{\rho}_{apx}(D_1 \times \ldots \times S \times \ldots \times D_n)\downarrow_\ell) = D_\ell \\
\Longrightarrow & \quad D_1 \times \ldots \times D_n \subset \overline{\rho}_{apx}(D_1 \times \ldots \times D_n) \\
\Longrightarrow & \quad \overline{\rho}_{apx}(D_1 \times \ldots \times D_n) = D_1 \times \ldots \times D_n
\end{aligned}$$

◁

À partir de l'algorithme précédent, on obtient facilement une version incrémentale permettant d'ajouter dynamiquement une relation.

Algorithme 3 (Ajout de relation)
Soit A l'ensemble des contraintes actives d'un problème donné. Soit ρ une nouvelle relation et $D_1, \ldots, D_n$ les domaines des variables considérées.

```
point-fixe-ajout(in ρ; in A; inout {D_1,...,D_n})
(1)  début
(2)     initialiser(Q)
(3)     Δ ← prop(D_1,...,D_n,ρ)
(4)     pour i de 1 à n faire
(5)        si Δ_i ≠ ∅ alors
(6)            enqueue(Q,(i,Δ_i,ρ))
(7)            D_i ← D_i \ Δ_i
(8)        finsi
(9)     finpour
(10)    tant que Q ≠ ∅ faire
(11)       (i,S,ρ) ← dequeue(Q)
(12)       pour tout ρ' ∈ {θ ∈ A \ ρ | v_i ∈ var(θ)} faire
(13)           Δ ← localprop(D_1,...,D_n,ρ',i,S)
(14)           pour j de 1 à n sauf i faire
(15)               si Δ_j ≠ ∅ alors
(16)                   enqueue(Q,(j,Δ_j,ρ'))
(17)                   D_j ← D_j \ Δ_j
(18)               finsi
(19)           finpour
(20)       finpour
(21)    fintantque
(22) fin
```

Théorème 7 (Filtrage et Point fixe)
Si pour une propriété $\mathcal{P}$ donnée, on peut trouver une approximation apx telle que le point fixe pour la réduction corresponde à la vérification de la propriété $\mathcal{P}$ alors l'algorithme 2 (algorithme `point-fixe`*) permet de faire un filtrage pour $\mathcal{P}$.*

▷ **Preuve :** C'est exactement le comportement d'un algorithme de point fixe. ◁

6.2 Maintien de déduction pour la réduction

Maintenant que nous avons donné un algorithme de réduction permettant de restreindre les domaines des variables d'un problème de telle sorte qu'une propriété donnée soit vérifiée (à l'aide du théorème 7), nous pouvons définir notre système de maintien de déduction.

Le système de maintien de déduction va nous permettre, d'une part, de spécifier les fonctions paramètres de notre recherche en meilleur d'abord (`explique-contradiction` et `maj-configuration`) dans le cas des propriétés proposant un filtrage par réduction et, d'autre part, de généraliser les algorithmes de maintien dynamique d'arc-consistance.

On se propose d'enregistrer (de maintenir) un certain nombre d'informations au cours de la résolution. On peut ainsi facilement définir et spécifier la fonction `explique-contradiction`. Ces informations seront utilisées pour une première spécification de `maj-configuration`.

Dans la suite, nous appellerons *domaine de la variable i* l'élément D_i renvoyé par l'algorithme `point-fixe` (algorithme 2).

6.2.1 Concepts de base

Définition 34 (Déduction)
On appelle ***déduction*** *le retrait d'une valeur dans le domaine d'une variable.*

On notera $\delta_{v \neq a}$ le retrait de la valeur a du domaine de v. On utilisera aussi une fonction $\delta(v, a)$ définie sur le domaine originel de v (D_v^{orig}) et renvoyant la déduction associée au retrait de a du domaine de v, si celui-ci a été réalisé. Cette fonction sera utilisée dans la section 6.2.4 pour étendre la notion de déduction.

Nous notons Δ l'ensemble des déductions réalisées lors de la résolution d'un problème donné. La taille de Δ est bornée par $n \times d$ où d est la taille maximum d'un domaine et n le nombre de variables.

Le retrait de valeur est l'opération de base réalisée par un algorithme de réduction de domaine tel que l'algorithme 2 (aux lignes 8 et 19). C'est pourquoi la notion de déduction est la notion de base de notre DMS.

En fait, nous ne cherchons pas à enregistrer les déductions mais plutôt la raison de leur réalisation. C'est pourquoi, nous introduisons la notion d'explication pour une déduction reposant sur les contraintes *responsables* du retrait et sur d'éventuels retraits précédents causes du nouveau retrait considéré.

Définition 35 (Explication)
Une ***explication*** *pour une déduction δ est un couple (E_C, E_D) où E_C et E_D sont deux ensembles définis de la manière suivante :*

- *$E_C \subset \mathcal{C}$, c'est un sous-ensemble des contraintes du problème ;*
- *$E_D \subset \Delta$, c'est un ensemble de déductions ;*
- *soit D^{in} tel que*

$$\forall i, D_i^{in} = D_i^{\text{orig}} \setminus \{a \mid \delta_{v_i \neq a} \in E_D\}$$

Soit $D^{out} =$ `point-fixe(`E_C`,` D^{in}`)`. On a alors, si $\delta = \delta_{v_k \neq a}$, $a \notin D_k^{out}$.

En d'autres termes, l'application de l'algorithme de réduction sur les contraintes de E_C, les domaines ne contenant aucune valeur concernée par les déductions de E_D, conduit à la réalisation de la déduction δ i.e., *à la fin de l'algorithme, la valeur concernée par δ n'apparaît plus dans le domaine de la variable concernée.*

> **Exemple 22 (Explication de déduction) :**
> Soient v_1 et v_2 deux variables de domaine $\{1, 2, 3\}$. Soit $c_1 : v_1 > 2$ et $c_2 : v_1 = v_2$.
> L'utilisation d'un algorithme de réduction établissant l'arc-consistance conduit, entres autres, au retrait de la valeur 2 de v_1. Une explication pour ce retrait est alors $(c_1, \varnothing)$.
> De même, l'arc-consistance demande que la valeur 2 soit retirée du domaine de v_2, l'explication est ici : $(c_2, \delta(v_1, 2))$.

Il peut y avoir plusieurs explications pour une même déduction δ, lorsque c'est le cas, on les regroupe dans un ensemble noté Ex_δ.

L'ensemble des explications enregistrées pour tous les éléments de l'ensemble Δ permet de définir la notion de système d'explication.

Définition 36 (Système d'explication)
On appelle ***système d'explication*** *la donnée d'un ensemble de déductions et de leurs explications associées.*

Du fait de la modification possible du statut (active, relaxée) de chaque contrainte au cours de la résolution, il faut introduire la notion de validité pour les déductions et les explications.

Définition 37 (Validité)

- *Une déduction δ est* ***valide*** *dans une configuration $\langle A, R \rangle$ si et seulement si Ex_δ contient au moins une explication valide,*
- *Une explication (C, D) est* ***valide*** *dans une configuration $\langle A, R \rangle$ si et seulement si :*
  - $C \subset A$
  - $\forall \delta \in D \quad \delta$ valide. *(Si $D = \varnothing$ la propriété précédente suffit à prouver la validité)*

Exemple 23 (Explication valide) :
Ainsi, dans l'exemple 22, l'explication $(c_2, \delta(v_1, 2))$ est valide dans la configuration $\langle \{c_1, c_2\}, \varnothing \rangle$ mais ne l'est plus pour la configuration $\langle \{c_1\}, \{c_2\} \rangle$.

Comme nous utilisons un algorithme de réduction, une fois qu'une valeur est retirée d'un domaine, elle ne peut bien sûr plus l'être ultérieurement sauf si elle a été réintroduite à la suite d'une relaxation de contrainte. C'est pourquoi on peut énoncer la proposition suivante.

Proposition 11 (Une seule explication valide)
Lorsqu'aucune contrainte n'est réintroduite (passage du statut relaxé au statut actif), il y a au plus une explication valide par déduction.

▷ **Preuve :** En effet, on ne peut ajouter une explication nouvelle que s'il n'y en a pas ou si l'ensemble de valeurs concerné est inclus dans le domaine considéré (la déduction était invalide jusqu'à présent). Et comme il n'y a pas eu de retour de contrainte, les explications invalides le sont restées. ◁

Bien sûr, ces définitions de validité ne sont valables que si le système d'explications ne produit pas de «*boucle*» : on dit alors qu'il est bien fondé. Pour définir cette notion, nous devons introduire la notion de graphe de dépendance des déductions qui permet de représenter les liens qui existent entre les déductions, permettant alors de déterminer leur validité.

Définition 38 (Graphe de dépendance des déductions)
On peut associer un ***graphe de dépendance de déduction*** (X, U) *à un système d'explication. L'ensemble des sommets est l'ensemble des déductions du problème ($X = \Delta$). Un arc a entre δ_1 et δ_2 appartient à U si et seulement si :*

$$\exists (C, D) \in Ex_{\delta_1}, \delta_2 \in D$$

Un tel graphe permet de donner une représentation graphique des dépendances entre les déductions (par leurs explications associées).

Exemple 24 (Graphe de dépendance de déductions) :
Considérons le système d'explications suivant.

Soient v_1 et v_2 deux variables de domaines $D_1 = D_2 = \{1, 2, 3\}$. Nous supposerons l'existence des contraintes c_1, c_2 et c_3. Nous avons répertorié dans le tableau 6.1 les déductions et leur explication associée (dans l'ordre d'apparition).

Le graphe de dépendance des déductions associé à ce système est représenté figure 6.1.

Déduction	Explication
$\delta(v_1, 1)$	$(\{c_1\}, \varnothing)$
$\delta(v_2, 1)$	$(\{c_3\}, \{\delta(v_1, 1)\})$
$\delta(v_2, 2)$	$(\{c_2\}, \{\delta(v_1, 1)\})$
$\delta(v_1, 3)$	$(\{c_3\}, \{\delta(v_2, 2)\})$
$\delta(v_2, 2)$	$(\{c_1\}, \{\delta(v_2, 1)\})$

TAB. 6.1 – *Déductions et explication associée dans l'ordre d'apparition*

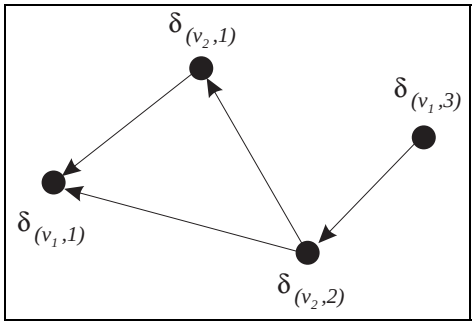

FIG. 6.1 – *Exemple de graphe de dépendance de déduction*

Définition 39 (Système d'explications bien-fondé)
Un système d'explication est dit ***bien fondé*** *si et seulement si son graphe de dépendance de déduction associé est sans circuit.*

Si le système d'explications utilisé est bien fondé, la définition récurrente de la validité ne pose pas de problème. Dans le cas contraire, on ne peut plus décider de la validité de certaines déductions.

> **Exemple 25 (Système d'explication mal fondé) :**
> Le système d'explication de l'exemple 24 est bien fondé comme on peut le voir sur la figure 6.1. Par contre, si on y ajoute l'explication $(\{c_3\}, \{\delta(v_1, 3)\})$ pour la déduction $\delta(v_2, 1)$, le système d'explication n'est plus bien fondé. Le graphe de dépendance associé est représenté figure 6.2. On ne peut plus déterminer la validité des déductions $\delta(v_1, 3)$, $\delta(v_2, 1)$ et $\delta(v_2, 2)$.

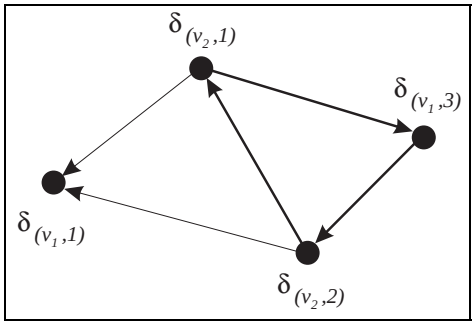

FIG. 6.2 – *Exemple de système d'explications mal fondé*

À partir d'un système d'explication, on peut définir la notion de système de

maintien de déduction.

Définition 40 (Système de Maintien de Déduction – DMS)
On appelle système de maintien de déduction, la donnée d'un système d'explication et des procédures de modifications et d'accès sur les données de ce système : ajout d'explication, test de validité, ...

6.2.2 Conditions fondamentales de bon fonctionnement

Nous nous proposons ici d'introduire des conditions de fonctionnement pour notre système qui nous permettent d'assurer le caractère bien fondé du système d'explication utilisé. Elles sont au nombre de deux.

Définition 41 (Condition de fonctionnement C1)
Lorsqu'une explication est ajoutée au système d'explications celle-ci doit être valide.

Définition 42 (Condition de fonctionnement C2)
Si pour une déduction donnée, une nouvelle explication est déterminée, celle-ci ne sera prise en compte que si :

- *la déduction n'a pas encore d'explication ;*
- *ou bien l'explication courante n'est pas valide et la nouvelle explication remplace alors l'ancienne.*

Lemme 3 (Unicité de l'explication)
*À tout instant, sous la condition de fonctionnement **C2**, il y a au plus une explication associée à une déduction.*

▷ **Preuve :** Par définition de **C2**. ◁

Proposition 12 (Condition de validité)
*Sous la condition de fonctionnement **C2**, si une déduction δ est invalide, tous les éléments de la fermeture transitive des prédécesseurs de δ sont invalides.*

▷ **Preuve :** C'est une conséquence de la définition de la validité (définition 37 page 70) et du fait qu'il n'y a qu'une seule explication associée à une déduction sous la condition de fonctionnement **C2** (lemme 3). ◁

Ces quelques résultats nous permettent d'énoncer le théorème fondamental suivant :

Théorème 8 (Système d'explication bien fondé)
*Tout système d'explication vérifiant les conditions **C1** et **C2** est bien fondé.*

▷ **Preuve :** Soit un système d'explication bien fondé. Soit $E = (C, D)$ une nouvelle explication pour une déduction δ donnée. Supposons que E sous les conditions de fonctionnement **C1** et **C2** rende le système d'explication mal fondé.

La prise en compte de la nouvelle explication introduit alors un circuit dans le graphe de dépendance des déductions. Soit δ' un élément de D tel que l'arc (δ, δ') soit un élément du circuit.

Comme la condition **C1** est vérifiée, (C, D) est une explication valide, donc δ' est valide. De plus, comme (δ, δ') est un élément du circuit, δ' appartient à la fermeture transitive des prédécesseurs de δ. Or, la condition **C2** est vérifiée, donc :

- soit il n'existe pas d'explication associée à δ. Alors δ n'est pas valide, ainsi δ' non plus (proposition 12) ;

– soit l'explication déjà associée à δ est invalide, δ' ne l'est donc pas non plus (proposition 12).

On a donc une contradiction, la nouvelle explication ne peut donc pas rendre mal fondé le système d'explication de départ. ◁

Ces conditions de fonctionnement seront reprises plus tard. Elles nous permettront alors de constater leur adéquation avec un comparateur *contradiction-local* et leur intérêt pour les résultats que l'on peut alors obtenir sur la complexité de notre approche (*cf.* section 7.4). De plus, elles nous seront aussi très utiles pour proposer des simplifications dans l'enregistrement proposé.

6.2.3 Simplification

On définit la notion d'environnement qui permet, à un instant donné de la résolution, de capturer synthétiquement toute l'information contenue dans une explication. Nous supposerons que le système d'explication considéré est bien fondé.

Définition 43 (Environnement)
On appelle ***environnement*** *d'une déduction δ, déterminé à partir d'un élément $(C, D) \in Ex_\delta$, un sous-ensemble des contraintes du problème noté $E_{(C,D)}$ tel que :*

$$E_{(C,D)} = \begin{cases} C & \text{si } D = \varnothing \\ C \cup \bigcup_{\delta' \in D, (C',D') \in Ex'_\delta} E_{(C',D')} & \text{sinon} \end{cases}$$

Un environnement pour une déduction donnée est donc calculé à partir d'une explication possible pour cette déduction. L'opération d'union de la définition est donc réalisée sur la sélection d'une explication pour chaque élément de D.

Exemple 26 (Environnement) :
Considérons le système d'explications donné dans l'exemple 24. L'ensemble $\{c_1, c_2, c_3\}$ est un environnement pour la déduction $\delta(v_1, 3)$.

On peut noter que si E est un environnement associé à une déduction δ, $(E, \varnothing)$ est une explication pour δ. L'ensemble des environnements existants pour une déduction δ sont regroupés dans son **étiquette** qui est notée E_δ.

On peut maintenant voir d'une autre façon la validité d'une déduction.

Définition 44 (Validité – nouvelle version)

- *Un environnement E (un ensemble de contraintes) est* ***valide*** *dans une configuration $\langle A, R \rangle$ si et seulement si $E \subset A$.*
- *Une explication est* ***valide*** *dans une configuration $\langle A, R \rangle$ si et seulement si un environnement qui lui est associé est valide dans cette même configuration.*
- *Une déduction δ est* ***valide*** *dans une configuration $\langle A, R \rangle$ si et seulement si il existe un environnement valide pour la configuration $\langle A, R \rangle$ dans son étiquette.*

Théorème 9 (Explications statiques et Système bien fondé)
Soit un système d'explication bien fondé. Soit une explication (C, D). Soit un environnement E de (C, D). Le système d'explication défini en remplaçant (C, D) par $(E, \varnothing)$ est bien fondé.

▷ **Preuve :** L'ajout d'une telle explication ne créé aucun arc dans le graphe de dépendance des déductions, le système d'explication est donc nécessairement bien fondé. ◁

On peut qualifier le système ainsi obtenu de statique. En effet, une modification sur l'ensemble des explications associées à une déduction δ ne peut plus être répercutée sur les autres déductions dépendant de δ. Nous proposons le paramètre `statique`.

Définition 45 (Paramètre `statique`)
Lorsque le paramètre `statique` *est* `on` *les transformations proposées dans le théorème 9 sont effectuées systématiquement.*

Lorsque le paramètre `statique` *est* `off`*, ces transformations ne sont pas effectuées. Dans ce cas, les conditions de fonctionnement* ***C1*** *et* ***C2*** *doivent être vérifiées pour garantir la pertinence de l'approche (caractère bien fondé du système d'explication).*

6.2.4 Extensions de la notion de déduction

Gestion de retraits groupés

Lorsque les conditions de fonctionnement garantissent l'unicité de l'explication du retrait d'une valeur d'une variable (condition **C2** – *cf.* lemme 3), on peut étendre la notion de déduction au retrait de plusieurs valeurs avec la même explication.

Définition 46 (Déduction étendue)
On appelle ***déduction étendue*** *le retrait d'une* ou plusieurs *valeurs dans le domaine d'une variable. On utilisera une fonction $\delta^e(v, a)$ renvoyant la déduction étendue associée à v concernant la suppression de la valeur a.*

$\delta^e(v, a)$ est définie sur le domaine originel D_v^{orig} de x et renvoie, si elle existe, *la*[6] déduction étendue dont le retrait concerne a.

Nous avons aussi besoin de l'explication du retrait d'un ensemble de valeurs.

Définition 47 (Explication d'ensemble)
On appelle ***explication*** *du retrait* ***de l'ensemble*** *S de la variable v la réunion des explications associées aux déductions de chacun des éléments de l'ensemble $S \cap D_v^{\text{orig}}$.*

Rappelons que comme la condition de fonctionnement **C2** est vérifiée, il existe une unique explication de retrait de valeur pour toute valeur du problème.

Cette extension permet alors de *compacter* les informations collectées lors du traitement d'une contrainte (plus particulièrement le retrait de valeurs consécutives). On verra de plus leur intérêt particulier dans le chapitre 8 pour une instance de notre approche sur les problèmes numériques (utilisation des *intervalles*).

Explication de retrait de contrainte

On peut étendre la notion de déduction dans une autre direction. On peut associer une explication à chaque retrait de contrainte. Le retrait de contrainte devient donc une déduction lui aussi.

Définition 48 (Déduction et Relaxation)
On appelle déduction de relaxation le retrait d'une contrainte de l'ensemble des contraintes actives dans le cadre de la fonction `meilleure-configuration`*.*
On note $\delta(c)$ la déduction de relaxation associée à la contrainte c.

6. Cette déduction est bien unique car on ne retire qu'une fois une valeur (la condition **C2** est vérifiée). On ne peut donc avoir le cas où une valeur a été retirée de deux façons différentes.

On peut donner une explication pour cette relaxation.

Définition 49 (Explication de relaxation)
Soit c une contrainte. Soit $S = \{E \in \mathcal{E}_c \mid c \in E\}$. L'ensemble $\{E \setminus \{c\} \mid E \in S\}$ constitue un ensemble d'explications pour le retrait de la contrainte c.

Notons que cette explication ne correspond pas aux critères de la définition 35 (*cf.* page 69) car l'ensemble des contraintes d'une explication ne suffit pas, ici, à assurer la relaxation de la contrainte associée. Par contre, lorsque le comparateur utilisé est *contradiction-local* les critères de la définition sont vérifiés. Par exemple, lorsqu'on utilise le comparateur C_{MM}, on peut donner une explication simple pour cette relaxation.

Proposition 13 (Explication de relaxation pour C_{MM})
Soit c une contrainte. Soit $E \in \mathcal{E}_c$ tel que $c = \min(E)$.
$(E \setminus \{c\}, \varnothing)$ est une explication pour $\delta(c)$ lorsque le comparateur C_{MM} est utilisé.

> ▷ **Preuve :** En effet, lorsque le comparateur C_{MM} est utilisé, c doit être relaxée puisqu'elle est le minimum d'une explication de contradiction. Ainsi, si $E \setminus \{c\} \subset A$ pour une configuration $\langle A, R \rangle$, alors $c \in R$ car E est une explication de contradiction. La proposition est donc vraie. Cette explication de relaxation est unique. ◁

Nous utiliserons l'explication de relaxation dans les chapitres 7 et 8 car elle permet de gérer l'énumération et les contraintes dites *disjonctives*.

6.3 Intégration du Maintien de déduction dans la recherche

On peut maintenant utiliser le système de maintien de déduction pour définir un système de relaxation de contraintes.

6.3.1 Fournir des explications

Pour intégrer un système de maintien de déduction permettant de mettre en place les fonctionnalités d'un système réactif de traitement des CSP, il faut modifier le fonctionnement normal de l'algorithme 2 de réduction (présenté page 66) de telle sorte que :

- les opérations sur les domaines (retrait de valeur, accès au domaine courant) ne sont réalisables que par l'intermédiaire du système de maintien de déduction. Celui-ci connaît l'état courant du domaine d'une variable en examinant (et maintenant) la validité des déductions associées à cette variable ;
- l'algorithme utilisé est modifié pour fournir des *explications* pour chaque retrait de valeur demandé.

Ainsi, les explications pour les retraits de valeurs des lignes 8 et 19 de l'algorithme 2 sont simplement définies de la façon suivante.

Proposition 14 (Explication de retrait)
Le couple $(\{\rho\}, \{\delta(v_j, a) \mid v_j \in vars(\rho) \setminus v, a \in D_j^{\text{orig}}, a \notin D_j\})$ est une explication *du retrait de Δ_i du domaine de la variable v effectué lignes 8 et 19 de l'algorithme 2 et aussi lignes 7 et 17 de la version incrémentale de l'algorithme 2 : l'algorithme 3.*

▷ **Preuve :** La connaissance de ces informations suffit pour les fonctions `prop` et `localprop` telles que définies jusqu'à présent. ◁

Lorsqu'on connaît la sémantique des contraintes traitées, on peut, à la manière d'AC5, *spécialiser* le contenu des explications. En particulier, pour les retraits propagés (ligne 19 de l'algorithme 2) des explications plus précises peuvent être alors calculées.

En intégrant ces fournitures d'explication, l'algorithme de réduction devient alors (nous avons repris la version incrémentale de l'algorithme – algorithme 3 page 68) :

Algorithme 4 (Ajout de relation et DMS)

```
point-fixe-ajout(in ρ; inout {D_1,...,D_n})
(1)  début
         % Initialisations
(2)      initialiser(Q)
(3)      Δ ← prop(D_1,...,D_n,ρ)
(4)      pour i de 1 à n faire
(5)         si Δ_i ≠ ∅ alors
(6)             enqueue(Q,(i,Δ_i,ρ))
(7)             dms-retrait(i,Δ_i,explication de retrait)
(8)         finsi
(9)      finpour
         % Propagations
(10)     tant que Q ≠ ∅ faire
(11)        (i,S,ρ) ← dequeue(Q)
(12)        pour tout ρ' ∈ {θ ∈ A \ ρ | v_i ∈ var(θ)} faire
(13)            Δ ← localprop(D_1,...,D_n,ρ',i,S)
(14)            pour j de 1 à n sauf i faire
(15)                si Δ_j ≠ ∅ alors
(16)                    enqueue(Q,(j,Δ_j,ρ'))
(17)                    dms-retrait(j,Δ_j,explication de retrait)
(18)                finsi
(19)            finpour
(20)        finpour
(21)     fintantque
(22) fin
```

La fonction `dms-retrait` enregistre la nouvelle explication apportée pour expliquer le retrait des valeurs spécifiées pour le domaine considéré selon la proposition 14L'implémentation de `dms-retrait` dépend de l'état courant du paramètre `statique`.

Algorithme 5 (Ajout d'explication dans le DMS)
On peut définir la fonction `dms-retrait` *de la façon suivante :*

```
dms-retrait(i,Δ_i,explication)
(1)  début
(2)      retirer les éléments de Δ_i de D_i
(3)      pour tout a ∈ Δ_i faire
(4)         si statique alors
(5)             explication ← (environnement d'explication,∅)
(6)         finsi
(7)         si explication est valide alors
(8)             remplacer l'explication courante de δ(i,a) par explication
(9)         finsi
(10)        si non statique alors
(11)            propager la nouvelle information si nécessaire
```

```
(12)       finsi
(13)    finpour
(14) fin
```

Proposition 15 (Conditions de fonctionnement)
Les conditions de fonctionnement ***C1*** *et* ***C2*** *sont vérifiées.*

▷ **Preuve :** L'algorithme 5 en est l'expression directe. ◁

On peut donc utiliser indifféremment les deux valeurs du paramètre `statique`.

Théorème 10 (Système bien fondé)
Le système d'explications utilisé jusqu'à présent est bien fondé.

▷ **Preuve :** Comme les conditions **C1** et **C2** sont vérifiées, par application du théorème 8, le système d'explication utilisé est bien fondé. ◁

6.3.2 Spécification de `explique-contradiction`

On peut en fait étendre le concept de déduction à la notion de $\mathcal{P}$-contradiction. Selon les définitions d'une explication de contradiction (définition 23 page 44) et d'une explication (définition 35 page 69), on peut affirmer qu'une *explication de contradiction* est bien une *explication* de la déduction de la contradiction.

Une contradiction est identifiée sur les CSP (classiques ou numériques) lorsque le domaine d'une variable x devient vide *i.e.* les déductions associées à toutes les valeurs possibles de la variable (les éléments de D_x) ont toutes une explication valide.

On peut déterminer une *explication de contradiction* de la façon suivante.

Lemme 4 (Fournir une explication de contradiction)
Soit v la variable cause de la contradiction (dont le domaine est devenu vide). $(\varnothing, \{\delta_{v \neq a} \mid a \in D_v\})$ est une explication pour la contradiction et un environnement associé est une explication de contradiction.

▷ **Preuve :** Un domaine vide provoque bien une $\mathcal{P}$-contradiction. ◁

La fonction `explique-contradiction` fournit l'environnement mentionné dans le lemme 4. Si la condition de fonctionnement **C2** est vérifiée, par application du lemme 3, cet environnement est unique.

6.3.3 Spécification de `maj-configuration`

La fonction `maj-configuration` qui se charge d'effectuer les mises à jour liées au changement de configuration doit faire en sorte de réaliser ce changement de manière incrémentale. Le but est en fait de rétablir la vérification de la propriété $\mathcal{P}$ après un retrait de contraintes comme si l'ajout ainsi *remis en cause* n'avait pas eu lieu. Il s'agit en fait du «*pendant*» de l'algorithme 4 qui cherche un point fixe en cas d'ajout de contrainte. Ici, nous cherchons un point fixe en cas de retrait de contrainte. Nous calquerons notre présentation sur celle habituellement proposée pour présenter DNAC4 [Bessière, 1991; Debruyne, 1995].

La fonction `maj-configuration` que nous proposons est une mise en œuvre pour un algorithme de réduction de la sémantique opérationnelle présentée section 4.3. Plus précisément, nous définissons exactement la partie réutilisation de la sémantique proposée.

Le changement de configuration est réalisé en deux temps :

- dans un premier temps, le changement de statut des contraintes à relaxer est propagé. Il s'agit d'effacer les effets passés de la contrainte ;
- dans un second temps, le changement de statut des contraintes à réintroduire est propagé. Il s'agit alors de rétablir la propriété $\mathcal{P}$ dans le problème traité.

Nous présentons d'abord l'algorithme lié à la suppression (relaxation) d'une relation. Ensuite, nous présenterons un algorithme lié à la réintroduction d'une relation.

Bien qu'un algorithme d'ajout de relation ait déjà été donné (l'algorithme 4), ici, comme il s'agit d'une réintroduction, et que la relation a déjà été traitée par cet algorithme, nous donnerons un algorithme légèrement différent puisqu'il peut arriver que des informations puissent être réutilisées.

Relaxation de contraintes

Nous utilisons ici un algorithme de type DNAC4 (*cf.* section 3.2.2 page 28). Nous utilisons pour cela la notion de dépendance entre une déduction et une contrainte (relation).

Définition 50 (Déductions dépendant d'une relation)
On note $\alpha(\rho)$ les déductions dépendant d'une contrainte ρ.

$$\alpha(\rho) = \{\delta \in \Delta \mid \forall E \in E_\delta, \rho \in E\} \tag{6.3}$$

Chacun des éléments de $\alpha(\rho)$ devient invalide lorsque ρ est relaxée. Pour rétablir la propriété $\mathcal{P}$, il faut alors tenter de trouver une autre explication pour chacune de ces déductions[7]. Le fonctionnement est alors similaire à celui de l'algorithme DNAC4. La différence essentielle vient du fait qu'aucune *propagation* des retours de valeurs n'est à envisager. En effet, l'ensemble $\alpha(\rho)$ réunit tous les retraits devant être reconsidérés suite à la relaxation de la relation ρ.

On obtient alors l'algorithme de relaxation suivant :

Algorithme 6 (Retrait de relation)

```
point-fixe-retrait(in ρ; inout {D_1, ..., D_n})
(1)  début
(2)     initialiser Q
(3)     pour tout δ_{v_i ∉ S} ∈ α(ρ) faire
(4)        autre-explication(v_i, S, D_1 × ... × D_n, Q)
(5)     finpour
(6)     propage-retraits(Q)
(7)  fin
```

où

- `autre-explication` *ajoute le retrait (v_i, a, ρ) dans Q pour tout élément a de S pour lequel il existe*[8] *une explication valide pour un retrait du domaine de v_i. Sinon, les valeurs S sont réintroduites dans le domaine de v_i.*
- `propage-retraits` *réalise les opérations présentées entre les lignes 10 et 21 de l'algorithme* `point-fixe-ajout` *(algorithme 4 page 76).*

7. Il ne s'agit pas d'une explication existante mais bien de propager les contraintes concernées pour voir si cette déduction peut être retrouvée.

8. *Pour cela, les contraintes portant sur v_i sont testées. Si on utilise un algorithme de type* AC4*, les contraintes sont passées en revue pour voir si un compteur associé à la variable concernée est nul (comme dans* DNAC4*). Si on utilise un algorithme de type* AC5*, la sémantique des contraintes peut être utilisées pour déterminer rapidement (en fait, dans le même ordre de complexité que l'algorithme considéré) si une valeur peut être retirée d'une autre façon ou non.*

Les algorithmes 4 et 6 permettent de généraliser l'algorithme DNAC4 pour toute méthode de réduction utilisée sur les domaines des variables d'un problème dynamique.

Réintroduction de contrainte

La problématique est ici différente. Il faut bien sûr relancer le solveur pour rétablir la propriété $\mathcal{P}$. Mais auparavant une part du travail peut être facilitée par le fait que des explications (et donc des déductions) invalides peuvent redevenir valides et ainsi une partie du calcul pourrait être évitée ; cela peut être le cas lorsque `statique` est `off` (définition 45 section 6.2.3 page 73).

Pour cela, on étend la relation de dépendance entre une déduction et une contrainte.

Définition 51 (Dépendance déduction–contrainte étendue)
Soit $\langle A, R\rangle$ la configuration avant réintroduction. On note $\beta(\rho)$ l'ensemble des déductions invalides susceptibles de redevenir valides si la relation ρ est réactivée.

$$\beta(\rho) = \{\delta \in \Delta_{\text{invalides}} \mid \exists E \in E_\delta, E \cap R = \{\rho\}\} \tag{6.4}$$

où $\Delta_{\text{invalides}}$ représente la restriction de Δ aux déductions invalides.

Les éléments de $\beta(\rho)$ redeviennent valides lorsque ρ est réintroduite. Les retraits de valeurs associés sont donc rétablis et on peut alors relancer l'algorithme utilisé pour établir incrémentalement la propriété $\mathcal{P}$ sur un ajout de relation (algorithme 4). On obtient l'algorithme suivant :

Algorithme 7 (Réintroduction de relation)

```
point-fixe-retour(in ρ; inout D_1,...,D_n)
(1)  début
(2)     pour tout δ_{v_i ∉ S} ∈ β(ρ) faire
(3)        D_i ← apx(D_i ∪ S)
(4)     finpour
(5)     point-fixe-ajout(ρ, D_1,...,D_n)
(6)  fin
```

Changement de configuration

On peut maintenant spécifier complètement la fonction `maj-configuration`.

Algorithme 8 (`maj-configuration`)

```
maj-configuration(in (A,R); in (A',R'); inout D_1,...,D_n)
(1)  début
(2)     pour tout ρ ∈ (R' \ R) faire
(3)        considérer ρ comme inactive
(4)     finpour
(5)     pour tout ρ ∈ (R' \ R) faire
(6)        point-fixe-retrait(ρ, D_1,...,D_n)
(7)     finpour
(8)     pour tout ρ ∈ (A' \ A) faire
(9)        point-fixe-retour(ρ, D_1,...,D_n)
(10)    finpour
(11) fin
```

6.4 Discussion

Nous souhaitons aborder ici un point qui n'a pas encore été évoqué : pourquoi affiner le système d'explication jusqu'au niveau des retraits de valeurs?

Nous avons pris parti d'affiner le plus possible notre système d'explication. C'est pourquoi, la déduction de base est le retrait de valeur dans le domaine d'une variable et non pas simplement la modification d'un domaine comme dans d'autres approches [Menezes *et al.*, 1993; Sola, 1995].

Cette précision se justifie par le fait que, surtout lorsque la sémantique des contraintes est utilisée pour fournir des explications, un système d'explication précis permettra de ne pas noyer la responsabilité de manière diffuse à de volumineux ensembles de contraintes. Le champ de responsabilité d'une contrainte est donc clairement identifié.

Nous avons besoin de cette précision dans deux situations :

- le retrait d'une contrainte (opération courante dans un système de relaxation de contraintes) doit pouvoir être réalisé à moindre coût *i.e.* en évitant le plus possible le *thrashing*, c'est pourquoi une extension de DNAC4 est utilisée permettant de fournir un système précis dans ces modifications suite au retrait d'une contrainte ;

- la recherche de la meilleure configuration prometteuse gagne elle aussi à être précise car on ne sait pas si la configuration vers laquelle on se dirige va être $\mathcal{P}$-satisfaisable (contrairement à l'approche proposée par [Menezes *et al.*, 1993]).

6.5 Conclusion

Nous avons montré dans ce chapitre comment utiliser un système de maintien de déductions pour la spécialisation des fonctions `explique-contradiction` et `maj-configuration` en vue de proposer un système de relaxation de contraintes pour les problèmes résolus avec des techniques de réduction de domaine.

Les deux chapitres suivants détaillent ces propositions dans le cas du traitement des domaines finis (chapitre 7) et dans le cas des intervalles (chapitre 8).

Chapitre 7

Instance sur les CSP : relax(FD,$\mathcal{C}$,$\mathcal{P}_{ac}$)

Sommaire

DANS CE CHAPITRE, nous présentons une instance complète de notre approche pour traiter les CSP. Pour cela, nous présentons tout d'abord une instance de l'algorithme de réduction présenté dans le chapitre précédent spécialisée pour traiter les domaines finis : nous rendons plus performantes les fonctions génériques `prop` et `local-prop` en les adaptant à différents types de contraintes.

Ensuite, comme la propriété vérifiée par les configurations déterminées par notre approche est une consistance locale (l'arc-consistance), il est indispensable de passer par une phase d'énumération qui est susceptible de modifier notre méthode de recherche. Mais, nous montrons comment l'énumération est intégrée de façon totalement transparente dans notre cadre général de relaxation. Pour cela, nous introduisons un nouveau type de contrainte : une contrainte de disjonction appelée contrainte `one-of` qui nous permet de considérer l'énumération comme une séquence d'ajouts et de retraits de contraintes d'égalité.

Avant de dérouler un exemple complet, nous présentons différents résultats de complexité montrant que notre proposition (soumise à certaines conditions de fonctionnement) peut être implémentée sur les domaines finis de manière tout à fait raisonnable sans tomber dans les travers habituels des méthodes basées sur l'enregistrement d'informations.

7.1 Le cadre général des CSP

Nous rappelons quelques définitions sur les problèmes de satisfaction de contraintes (CSP).

Définition 52 (CSP)
Formellement, un ***problème de satisfaction de contraintes*** *(CSP) est défini par un triplet (V, D, C) où :*

- *V est un ensemble de n variables ;*
- *D est l'ensemble des domaines associés aux variables de V. Un domaine est un ensemble fini discret de valeurs possibles pour la variable concernée. Pour une variable x, on note D^x le domaine qui lui est associé. On note d la taille maximum des domaines des variables du problème ;*
- *C est un ensemble de e contraintes portant sur les variables de V.*

Nous nous limiterons ici aux CSP binaires *i.e.* les contraintes du problème font entrer en jeu au plus 2 variables. On peut d'ailleurs montrer que tout CSP peut se ramener à un CSP binaire. Nous noterons c_{xy} une contrainte entre les variables x et y.

7.2 Spécialisation de l'algorithme de réduction

7.2.1 Fonction d'approximation pour un CSP

De manière habituelle, nous choisissons comme propriété $\mathcal{P}$ à vérifier l'arc-consistance ($\mathcal{P}_{ac}$: 2-consistance).

Définition 53 (k-consistance – *cf.* définition 5)
Un CSP est ***k-consistant*** *si et seulement si pour toute instanciation consistante de $k-1$ variables, il existe dans le domaine de toute variable non instanciée une valeur prolongeant cette instanciation en une instanciation consistante de k variables.*

Considérons la fonction identité sur $\mathcal{P}(D)$. Cette fonction est clairement une approximation au sens où nous l'avons définie dans le chapitre précédent (définition 31 page 64). Nous noterons cp son extension sur $\mathcal{P}(D^n)$ (*cf.* définition 32 page 65).

On peut montrer le résultat suivant [Benhamou, 1995].

Théorème 11 (cp et arc-consistance)
Soit D un ensemble. Soit $u \subset D^2$ et ρ une relation binaire sur D. ρ est arc-consistant pour u si et seulement si :

$$\overline{\rho}_{cp}(u) = u$$

Nous sommes exactement dans les conditions d'application du théorème 7 page 68 qui nous permet de dire que l'algorithme 2 fournit un algorithme de filtrage pour l'arc-consistance puisque lorsqu'on a un point fixe, la propriété associée est vérifiée. On peut donc utiliser les résultats du chapitre précédent pour proposer un système de relaxation de contrainte sur les CSP.

7.2.2 Différents algorithmes d'arc-consistance

L'algorithme de réduction proposé dans le chapitre précédent (algorithme 2 page 66) étant une généralisation de l'algorithme AC5 de Van Hentenryck *et al.* [1992], on peut le modifier simplement pour qu'il se comporte comme les algorithmes d'arc-consistance AC3, AC4 ou AC5.

- Ainsi, en spécialisant les fonctions `prop` et `localprop` en fonction des contraintes traitées comme présenté dans la section suivante, on obtient l'algorithme AC5.

- En modifiant la ligne 14 de cet algorithme (*cf.* page 66) pour passer de

  $$\rho' \in \{\theta \in A \setminus \rho \mid v_i \in var(\theta)\}$$

  à

  $$\rho' \in \{\theta \in A \mid v_i \in var(\theta)\}$$

  et en ne spécialisant pas les fonctions `prop` et `localprop`, on obtient l'algorithme AC4.

- Enfin, en remplaçant l'appel à `localprop` par un appel à `prop` ligne 15 pour ce même algorithme et en ne spécialisant toujours pas la fonction `prop`, on obtient le fonctionnement de l'algorithme AC3.

7.2.3 Spécialisation des fonctions paramètres

En pratique, les fonctions `prop` et `localprop` sont spécialisées suivant le type des contraintes traitées. Cette spécialisation conduit à une définition plus précise des explications. Nous utilisons ici les spécialisations utilisées pour l'algorithme d'arc-consistance AC5 [Van Hentenryck *et al.*, 1992].

Les algorithmes 4 (ajout de relation) et 6 (retrait de relation) sont modifiés pour intégrer l'appel à `dms-retrait` (algorithme 5) à l'intérieur des fonctions `prop` et `local-prop` puisque les explications fournies sont spécialisées pour ces fonctions en tenant compte du type de contrainte traitée. Nous donnons ici les explications que l'on peut définir pour des contraintes monotones, bijectives et anti-bijectives.

Contraintes monotones

Définition 54 (Contrainte Monotone)
Une contrainte c est dite ***monotone*** *[Van Hentenryck* et al.*, 1992] sur un domaine de calcul $\mathcal{D}$ si et seulement s'il existe un ordre total $\leqslant$ sur $\mathcal{D}$ tel que :*

$$\forall (a,b) \in \mathcal{D}^2 \quad c_{xy}(a,b) \Rightarrow \forall (a',b') \in \mathcal{D}^2, a' \leqslant a, b \leqslant b', c_{xy}(a',b')$$

Pour la fonction `prop`, on pose[1] : $v_M = \max\{a \mid a \in D_x\}$ et $w_M = \max\{b \mid b \in D_y\}$. On pose aussi, $f_M(w) = \max\{a \mid c_{xy}(a,w)\}$.

On pose de même : $v_m = \min\{a \mid a \in D_y\}$ et $w_m = \min\{b \mid b \in D_x\}$. On pose aussi, $f_m(v) = \min\{b \mid c_{xy}(v,b)\}$.

Proposition 16 (Explication de retrait – contrainte monotone)
$(\{c\}, \{\delta_{y \neq b} \mid b \in D_y^{\text{orig}}, b > w_M\})$ est une explication pour le retrait des valeurs de l'intervalle $]f_M(w_M), v_M]$ du domaine de la variable x dans le cadre de la fonction `prop` *pour la contrainte monotone c.*

De même, $(\{c\}, \{\delta_{x \neq a} \mid a \in D_x^{\text{orig}}, a < w_m\})$ est une explication pour le retrait des valeurs de l'intervalle $[w_m, f_m(v_m)[$ du domaine de la variable y dans le cadre de la fonction `prop`*.*

> ▷ **Preuve :** Montrons-le pour les retraits sur x. Pour ceux sur y le raisonnement est le même. Toutes les valeurs supérieures à $f_M(w_M)$ doivent être supprimées du domaine D_x si la contrainte c est active. $f_M(w_M)$ ne dépend que de c et du maximum w_M du domaine courant de y. Cette dernière valeur ne dépend que du fait que les valeurs supérieures à w_M dans D_y ont été supprimées. Ainsi, on a bien une explication. ◁

Exemple 27 (Exemple de contrainte monotone) :
Soient x et y deux variables de domaines (respectivement) $\{2,3,4,5\}$ et $\{1,2,3,4\}$. Supposons que l'on ait une contrainte $y \neq 4$ (4 est donc retirée du domaine de y).
Soit $c : x \leqslant y$ une nouvelle contrainte. c est une contrainte monotone. Dans le cadre d'un appel à `prop`(D_x, D_y, c), on est amené à :

- supprimer les valeurs 4 et 5 de x avec l'explication $(\{c\}, \{\delta(y,4)\})$. Dans le cas de 5, il s'agit bien d'une explication même si on pouvait faire mieux avec $(\{c\}, \varnothing)$;
- supprimer la valeur 1 de y avec l'explication $(\{c\}, \varnothing)$. En effet, aucune valeur strictement inférieure à 2 ne fait partie du domaine de x.

La figure 7.1 illustre ces calculs pour les retraits sur la variable x.

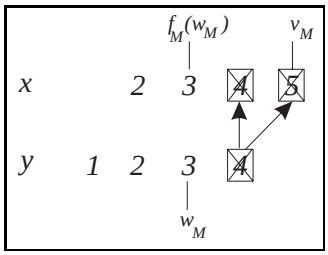

FIG. 7.1 – *Exemple d'explication pour une contrainte monotone : $x \leqslant y$*

La fonction `localprop` appelle la fonction `prop`.

1. Les relation max et min sont relatives à l'ordre total $\leqslant$ considéré.

Contraintes bijectives

Définition 55 (Contrainte bijective)
Une contrainte c sur x et y est dite ***bijective***[2] *sur $\mathcal{D}$ si et seulement si pour tout $v \in D_x$ (resp. $w \in D_y$) il existe au plus un $w \in D_y$ (resp. $v \in D_x$) tel que $c_{xy}(v, w)$ est vrai.*

On note $f_{xy}(v)$ la valeur w telle que $c_{xy}(v, w)$ et $f_{yx}(w)$ la valeur v telle que $c_{xy}(v, w)$.

Proposition 17 (Explication de retrait – contrainte bijective)
$(\{c\}, \{\delta_{y \neq f_{xy}(v)}\})$ est une explication du retrait de toute valeur v telle que $f_{xy}(v) \notin D_y$ du domaine de la variable x dans le cadre de la fonction `prop`.

De même, $(\{c\}, \{\delta_{x \neq f_{yx}(w)}\})$ est une explication du retrait de toute valeur w telle que $f_{yx}(w) \notin D_x$ du domaine de la variable y dans le cadre de la fonction `prop`.

▷ **Preuve :** Comme $f_{xy}(v)$ est la seule valeur permettant de conserver v dans x si la contrainte c est active, il s'agit bien d'une explication. Ces deux seules informations suffisent à montrer le retrait de v du domaine de x. Le raisonnement est le même pour les retraits sur y. ◁

La proposition suivante découle directement de ce dernier résultat.

Proposition 18 (Minimalité)
L'explication fournie dans la proposition 17 est minimale pour l'inclusion.

Proposition 19 (Explication de retrait propagé – contrainte bijective)
$(\{c\}, \{\delta_{y \neq w}\})$ est une explication du retrait de $f_{yx}(w)$ du domaine courant de la variable x dans le cadre de la fonction `localprop`.

$(\{c\}, \{\delta_{x \neq v}\})$ est une explication du retrait de $f_{xy}(v)$ du domaine courant de la variable y dans le cadre de la fonction `localprop`.

▷ **Preuve :** Conséquence directe de la proposition 17. ◁

Proposition 20 (Minimalité)
L'explication fournie dans la proposition 19 est minimale pour l'inclusion.

Exemple 28 (Contrainte bijective) :
Soient x et y deux variables de domaines (respectivement) $\{1, 2, 3, 4\}$ et $\{1, 2, 3\}$. Soit $c : x = y$ une contrainte entre x et y. c est une contrainte bijective. Dans le cadre d'un appel à `prop`(D_x, D_y, c), on est amené à retirer la valeur 4 du domaine de x avec l'explication $(\{c\}, \varnothing)$ car elle n'appartient pas au domaine originel de y.
Supposons qu'une contrainte donnée vienne à retirer la valeur 1 du domaine de y. Ce retrait conduit au retrait de 1 du domaine de x avec l'explication $(\{c\}, \{\delta(y, 1)\})$.
La figure 7.2 illustre ces deux situations.

2. functional constraint *dans [Van Hentenryck* et al., *1992].*

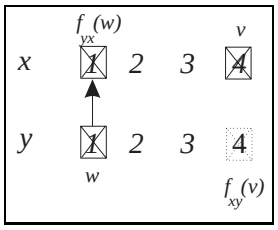

FIG. 7.2 – *Exemple d'explication pour une contrainte bijective : $x = y$*

Contraintes anti-bijectives

Définition 56
Une contrainte c sur x et y est dite ***anti-bijective***[3] *sur un domaine de calcul $\mathcal{D}$ si et seulement si $\neg c$ est une contrainte bijective.*

On note $f_{xy}(v)$ la valeur w telle que $\neg c_{xy}(v, w)$ est vrai et $f_{yx}(w)$ la valeur v telle que $\neg c_{xy}(v, w)$ est vrai.

Pour les contraintes anti-bijectives, les fonctions `prop` et `localprop` sont définies de la même manière.

Proposition 21 (Explication de retrait – contrainte anti-bijective)
Si D_y est réduit au singleton $\{w\}$, $(\{c\}, \{\delta_{y \neq a} \mid a \in D_y^{\text{orig}} \setminus w\})$ est une explication du retrait de $f_{yx}(w)$ de D_x dans le cadre de la procédure `prop`.

De même, si D_x est réduit au singleton $\{v\}$, $(\{c\}, \{\delta_{x \neq a} \mid a \in D_x^{\text{orig}} \setminus v\})$ est une explication du retrait de $f_{xy}(v)$ de D_y dans le cadre de la procédure `prop`.

▷ **Preuve :** C'est une conséquence directe de la proposition 17. ◁

La proposition suivante découle directement de ce dernier résultat.

Proposition 22 (Minimalité)
L'explication fournie dans la proposition 21 est minimale pour l'inclusion.

Exemple 29 (Contrainte anti-bijective) :
Soit x et y de domaine $\{1, 2, 3\}$. Supposons que les valeurs 1 et 3 aient été retirées de x. Soit $c : x \neq y$ une nouvelle contrainte. c est une contrainte anti-bijective. Un appel à `prop` provoque le retrait de la valeur 2 de y avec l'explication $(\{c\}, \{\delta(x, 1), \delta(x, 3)\})$.
La figure 7.3 illustre cette situation.

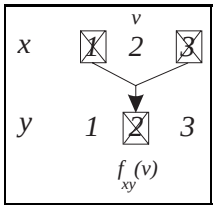

FIG. 7.3 – *Exemple d'explication pour une contrainte anti-bijective : $x \neq y$*

3. anti-functional constraint *dans [Van Hentenryck* et al., *1992].*

7.3 Énumération sur les CSP : la contrainte one-of

La phase d'énumération est une phase essentielle lorsque l'on traite des CSP. En effet, dans la plupart des cas, les consistances employées étant locales, seule une phase d'énumération permet de construire une instanciation consistante des variables.

Nous allons montrer comment on peut considérer l'énumération comme une séquence d'ajout et de retraits de contraintes d'égalité permettant ainsi de gérer l'énumération de manière «*transparente*» dans notre système. Cette gestion passe par l'intermédiaire d'une nouvelle contrainte permettant de modéliser une disjonction : la contrainte `one-of`.

7.3.1 Intégration de la phase d'énumération

Classiquement, l'énumération se réalise par backtrack : une variable est instanciée par une valeur dans son domaine courant, cette instanciation est propagée jusqu'à ce qu'une contradiction soit rencontrée (auquel cas l'instanciation est remise en cause) ou bien une autre variable est instanciée ...

En fait, lorsqu'on réalise une énumération sur un CSP jusqu'à présent arc-consistant, lorsqu'on rencontre une contradiction, on ne sait pas si le problème est sur-contraint ou si c'est simplement l'affectation courante qui n'est pas valide et qui doit être remise en cause.

Une façon élégante de contourner cette difficulté est de simplement considérer :

- l'affectation d'une valeur à une variable comme l'ajout d'une contrainte d'égalité ;
- le retour sur une instanciation comme le retrait de la contrainte d'égalité associée et l'ajout d'une autre contrainte d'égalité.

Ainsi, une contradiction rencontrée lors de l'énumération peut être traitée dans le cadre de notre système de relaxation de contrainte. En effet, si l'explication de contradiction ne contient aucune contrainte introduite par le processus d'énumération alors le problème est sur-contraint et doit être relaxé. Sinon, une des contraintes d'égalité introduite par l'énumération est supprimée et une autre est ajoutée.

7.3.2 Modifications pour le traitement de l'énumération

L'idée maîtresse est de modéliser l'appartenance d'une variable à un domaine comme la disjonction exclusive de contraintes d'égalités variable/valeur. C'est la contrainte `one-of` [4].

Définition 57 (Contrainte `one-of`)
Une contrainte dite `one-of` *sur un ensemble C_o de contraintes est vérifiée pour une configuration $\langle A, R \rangle$ si et seulement si une seule des contraintes de C_o appartient à A.*

L'importance associée à une contrainte sur laquelle porte une contrainte `one-of` doit être neutre vis-à-vis du comparateur utilisé. De plus, une contrainte `one-of`

4. La contrainte `one-of` n'est pas limitée aux contraintes d'égalité. Nous avons choisi de la présenter ici ainsi pour illustrer l'aspect dynamique de l'énumération. En effet, nous verrons dans le chapitre 11 qu'elle peut être utilisée pour traiter les contraintes disjonctives en général.

doit toujours être vérifiée (elle n'est pas relaxable) lorsqu'elle est active.

Proposition 23 (Association domaine-`one-of`)
Une spécification de domaine peut être réalisée par la déclaration d'une contrainte `one-of`*. Ainsi, au domaine* D_x^{orig} *d'une variable* x *correspond une contrainte* `one-of` *sur l'ensemble :* $C_o = \{c_{x=a} \mid a \in D_x^{\text{orig}}\}$.

Il faut s'assurer qu'une contrainte `one-of` est toujours vérifiée. En particulier, si le processus de relaxation conduit au retrait d'une contrainte permettant de vérifier le `one-of`, il faut aussitôt activer une autre contrainte de ce même `one-of`.

Notons que la fonction `meilleure-configuration` doit être modifiée de telle sorte que si plus d'une contrainte provenant d'un même `one-of` sont actives dans la configuration réponse, un certain nombre d'entre elles soient relaxées afin qu'il n'en reste plus qu'une active. Remarquons que cette modification dans la réponse fournie n'entraîne pas de modifications vis à vis du comparateur puisque de telles contraintes sont neutres vis à vis de ce dernier.

Lorsqu'on relaxe une contrainte d'énumération ρ (*i.e.* lors d'un appel à `maj-configuration`$(\langle A, R\rangle, \langle A', R'\rangle)$, $\rho \in R' \setminus R$), il est nécessaire de s'assurer qu'une autre contrainte ρ' du `one-of` concerné est dans A'.

Deux cas se présentent :

- A' contient déjà une contrainte du `one-of`, il n'y a rien à faire ;
- A' ne contient pas de telle contrainte, il faut alors en trouver une dans le `one-of` qui n'appartienne pas à R'. Une telle contrainte, si elle existe, n'a jamais été introduite et doit donc être ajoutée à A'.

Lorsque la contrainte ρ' n'existe pas, cela signifie que toutes les contraintes du `one-of` considéré sont relaxées et ne peuvent être réintroduites. Il y a alors une contradiction, non pas parce qu'un domaine est devenu vide mais parce qu'un `one-of` ne peut être vérifié. On a donc un nouveau cas de contradiction[5].

Pour pouvoir définir une notion d'explication de contradiction pour traiter ce nouveau cas, on utilise l'extension de la notion de déduction au retrait de contrainte (*cf.* section 6.2.4 page 74). On associe donc un ensemble d'explication à chaque retrait de contrainte liée à l'énumération : il s'agit des explications de contradiction recouvertes par la contrainte concernée (en retirant bien sûr cette même contrainte)[6].

ρ' est donc choisie parmi les contraintes dont la déduction de retrait n'existe pas ou est invalide. On constate ici que quel que soit le comparateur utilisé, on ne peut pas considérer une relaxation de contrainte d'énumération comme définitive. Si ρ' n'existe pas, on a le résultat suivant :

5. En fait, on peut ne pas créer cette nouvelle condition de contradiction. En effet, on peut considérer le retrait d'une contrainte d'un `one-of` comme la suppression de la valeur associée à la contrainte de la variable concernée. Ainsi, il ne reste qu'un seul cas de contradiction : le domaine d'une variable devient vide. Nous avons préféré conserver le nouveau cas de contradiction, car, comme nous y reviendrons chapitre 11, la contrainte `one-of` peut être utilisée pour modéliser une disjonction plus générale entre contraintes quelconques et donc ne plus offrir cette similitude entre retrait d'une contrainte et retrait d'une valeur.

6. Ces explications ne contenant qu'une partie *contrainte* n'ont pas besoin de vérifier les condition de fonctionnement **C1** et **C2**, le système d'explications obtenu sera toujours bien fondé. On peut noter que cet ensemble peut être assimilé aux ensembles *Alt* (*Alternative Backtrack points*) des méthodes de backtrack intelligents [Sola, 1995]. Pour ces méthodes, comme pour la nôtre, ces ensembles assurent la complétude de la recherche effectuée. Notons que losqu'on utilise C_{MM}, comme on l'a vu proposition 13 page 75, on se contente d'une seule explication de retrait de contrainte : l'explication de contradiction dont la contrainte relaxée est le minimum.

Proposition 24 (Explication de contradiction)
Lorsqu'une contrainte `one-of` *portant sur un ensemble C_o de contraintes ne peut plus être vérifiée, une explication de contradiction est :*

$$\{\varnothing, \{\delta(\rho) \mid \rho \in C_o\}\}$$

Une contrainte `one-of` *ne pouvant être relaxée, elle n'apparaît pas dans l'explication.*

Comme dans le cas d'un environnement pour un retrait de valeur (définition 43 page 73) il peut exister plusieurs explications pour un retrait de contrainte. Une explication de contradiction dans le cadre d'une contrainte `one-of` est alors définie à partir d'une sélection d'une explication pour chaque contrainte du `one-of`.

Proposition 25 (Explication de contradiction)
L'explication de contradiction proposée précédemment peut s'écrire aussi :

$$\{\bigcup_{c \in C_o} Ex_{\delta(c)}, \varnothing\}$$

L'algorithme de mise à jour de configuration (algorithme 8 page 79) doit être modifié pour tenir compte des spécificités d'une contrainte `one-of`. Lorsqu'on gère les contraintes `one-of`, la relaxation d'une contrainte permettant de vérifier une contrainte `one-of` pouvait conduire à d'autres relaxations ou d'autres ajouts de contraintes. Ainsi, la configuration d'arrivée fournie jusqu'à présent à la fonction `maj-configuration` ne garantit plus que la configuration courante effective à la fin de l'exécution soit bien la configuration attendue.

C'est pourquoi dans le nouvel algorithme de mise à jour que nous proposons, la configuration d'arrivée est calculée *en interne*. Le nom de la fonction peut donc être changé en `quitte-configuration` ce qui reflète mieux son fonctionnement.

Algorithme 9 (`maj-configuration` pour l'énumération)

```
quitte-configuration(in (A,R); out (A',R'); inout D_1,...,D_n)
(1)  début
(2)     (A',R') ← meilleure-configuration((A,R),E_c)
(3)     pour tout ρ ∈ (R' \ R) faire
(4)        considérer ρ comme inactive
(5)        si ρ est une contrainte d'énumération alors
(6)           trouver ρ' contrainte introduisible du même one-of que ρ
(7)           si ρ' n'existe pas alors
(8)              calculer une explication pour la contradiction
(9)              l'intégrer à l'ensemble E_c
(10)             retour ligne 2
(11)          sinon
(12)             ajouter ρ' à A'
(13)          finsi
(14)       finsi
(15)    finpour
(16)    pour tout ρ ∈ (R' \ R) faire
(17)       point-fixe-retrait(ρ,D_1,...,D_n)
(18)    finpour
(19)    pour tout ρ ∈ (A' \ A) faire
(20)       point-fixe-retour(ρ,D_1,...,D_n)
(21)    finpour
(22) fin
```

7.3.3 Propriétés de la solution proposée

Lorsqu'on prend en compte des contraintes de type `one-of`, l'arbre de la recherche de C-solution n'est plus un arbre binaire. En effet, on n'a plus le choix entre l'activation ou la relaxation d'une contrainte mais entre plusieurs membres d'une alternative.

> **Exemple 30 (Contrainte one-of) :**
> Considérons un CSP concernant au moins une variable v de domaine initial $\{1, 2, 3\}$. Considérons l'introduction d'une contrainte c_a puis d'une contrainte c_b et enfin d'une contrainte `one-of` sur l'ensemble : $C_o = \{c_{v=1}, c_{v=2}, c_{v=3}\}$. L'arbre de recherche sur lequel s'applique la méthode présentée dans le chapitre 4 est présenté figure 7.4.

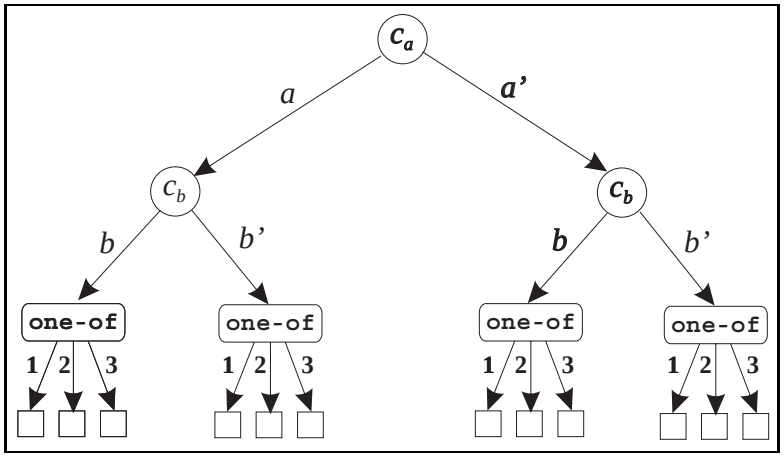

FIG. 7.4 – *Arbre de recherche pour* $\{c_a, c_b, \text{one} - \text{of}(C_o)\}$.

Le théorème suivant s'applique dans le cas général.

Théorème 12 (Complétude)
La méthode de gestion de l'énumération est complète et correcte.

> ▷ **Preuve :** Comme les contraintes membres d'un `one-of` sont neutres vis à vis du comparateur utilisé, la méthode de recherche employée permet de ne pas laisser passer de solution. Si une contrainte membre d'un `one-of` peut être réintroduite, elle le sera. La méthode est donc complète.
>
> Les explications de contradiction étant toutes conservées, la méthode est aussi correcte. En particulier, il ne peut y avoir phénomène de bouclage. ◁

Théorème 13 (Complétude et non conservation)
Si

- *les relaxations de contraintes* normales *(*i.e. *non membres d'un* `one-of`*) sont définitives (c'est le cas avec un comparateur* contradiction-local*) et qu'on se contente alors d'une seule explication de retrait par contrainte d'énumération ;*

- *en cas d'égalité pour le critère utilisé pour déterminer une contrainte à relaxer pour une explication de contradiction on choisit la contrainte la plus récente,*

alors la méthode reste complète et correcte.

▷ **Preuve :** Le seul problème posé par rapport au théorème précédent est qu'un certain nombre d'informations ne sont pas conservées. Ainsi, il pourrait y avoir *bouclage*. Nous allons donc montrer que cela ne peut être le cas sous les hypothèses du théorème 13. Nous allons faire une démonstration par l'absurde en montrant que si le bouclage est possible alors le problème contient un nombre infini de contraintes.

1. Supposons que l'on ait un phénomène de bouclage lors du traitement de l'énumération. Il n'y a donc pas d'ajout de contrainte «*normale*» au système.
   Soit $C_p = \langle A_p, R_p \rangle$ une configuration déjà explorée. Soit $C = \langle A, R \rangle$ la configuration courante postérieure à C_p telle que $A = A_p$.
2. Nécessairement, les explications de contradiction identifiées entre C_p et la configuration courante contiennent toutes au moins une contrainte d'énumération (liée à un `one-of`). En effet, si ce n'est pas le cas, au moins une contrainte «*normale*» a été relaxée et comme toute relaxation est dans ce cas définitive, on ne peut avoir $A_p = A$.
3. Soit E une explication de contradiction pour C_p. On a $E \subset A_p$. Soit ρ la contrainte d'énumération sélectionnée dans E pour relaxation. Comme $A_p \subset A$, ρ doit *revenir* dans l'ensemble des contraintes actives. Il faut donc que $E \setminus \rho$ devienne invalide. Il existe donc une contrainte $c_1 \in E \setminus \rho$ relaxée postérieurement. Comme $c_1 \in A_p$ et $c_1 \in A$, c_1 doit aussi *revenir* postérieurement. c_1 est donc aussi une contrainte d'énumération. De plus, ρ est plus récente que c_1 car sinon ρ n'aurait pas été relaxée dans E.
4. Soit E_1 l'explication de contradiction dans laquelle c_1 a été sélectionnée pour relaxation. Comme c_1 doit *revenir*, E_1 contient une contrainte c_2 qui doit être relaxée plus tard. c_2 ne peut être une contrainte «*normale*». En effet, on aurait alors $c_2 \notin A$ et $c_2 \in A_p$ ce qui est contradictoire avec l'hypothèse ($A_p = A$). c_2 est donc une contrainte d'énumération. Deux cas se présentent alors :
   - Soit $c_2 \in A_p$. Donc, si c_2 doit partir, c_2 doit aussi revenir. Il existe donc c_3 dans E_2, l'explication de contradiction dans laquelle c_2 a été sélectionnée. On se retrouve alors dans la même situation que pour c_1. (point 4)
   - Soit $c_2 \notin A_p$. C'est possible à condition que c_2 ait été introduite après C_p et que c_2 soit repartie postérieurement mais avant C. Si c_2 a pu être introduite, cela signifie qu'il existe une contrainte c_2' appartenant à A_p du même `one-of` que c_2. c_2' doit donc partir puis revenir. On est donc dans le même cas que pour c_2 dans le point précédent. On se retrouve donc là aussi dans le même cas que pour c_1. (point 4)
5. On peut donc créer ainsi un ensemble infini de contraintes d'énumération toutes plus anciennes les unes que les autres. Il y a donc une contradiction puisque l'ensemble des contraintes du problème est fini. On ne peut donc boucler.

◁

On peut remarquer que cette façon de faire (ne conserver que peu d'informations) peut conduire bien sûr à explorer des configurations que l'on saurait implicitement [7] conduire à une $\mathcal{P}$-contradiction si on conservait toutes les informations. L'avantage indéniable de cette approche est que l'occupation en espace peut alors être fortement améliorée. C'est donc un choix à faire. Nous nommerons dans notre système ce paramètre `enregistrement`. Lorsqu'il est activé (`on`), on enregistre toutes les

7. En effet, ces configurations n'ont pas déjà été explorées.

informations (toutes les explications de contradiction et de retrait) et lorsqu'il est `off`, on enregistre le minimum d'informations.

On peut aussi noter que si les relaxations de contraintes «*normales*» sont interdites *i.e.* on cherche une solution respectant impérativement toutes les contraintes, l'utilisation de la contrainte `one-of` pour réaliser l'énumération ne dépend plus du comparateur utilisé. On peut même aller plus loin :

Proposition 26 (Dynamic Backtracking et notre approche)
La gestion de l'énumération proposée revient en fait à une extension de l'algorithme Dynamic Backtracking *[Ginsberg, 1993].*

> ▷ **Preuve :** Nous ne donnons pas ici une preuve formelle. Néanmoins, rappelons que l'algorithme *Dynamic Backtracking* (*cf.* section 3.4.3 page 35) adopte une technique d'exploration similaire à la nôtre. La différence tient aux faits que :
>
> - nous avons une étape de propagation après chaque affectation de valeur. Cette propagation a des effets qui doivent être défaits en cas de remise en cause ;
> - nous gérons les affectations de valeurs comme des contraintes quelconques.
>
> Notre approche est donc plus générale que celle de *Dynamic Backtracking.* ◁

En ce qui concerne l'algorithme *Nogood Recording* que nous avons évoqué section 3.4.3 page 35, on peut voir ici que la limitation proposée pour l'arité des *nogoods* enregistrés peut être ici levée et remplacée par la limitation que nous nous fixons sur les explications de retrait de contraintes.

7.4 Complexité

Nous détaillons ici les résultats de complexité associés à la présentation théorique réalisée jusqu'ici. Nous conviendrons de ne pas préciser les valeurs exactes des complexités lorsqu'elles sont exponentielles.

7.4.1 Données de base

Nous noterons n le nombre de variables dans le problème, e le nombre courant de contraintes (sans compter les contraintes dues à l'énumération) et d la taille maximum d'un domaine pour une variable.

Le nombre effectif de contraintes du problème est au plus : $c = e + n$. En effet, pour chaque variable on ajoute une contrainte d'égalité (associée à la valeur courante de la variable).

Le nombre maximum de déductions valides dans le système est : $a = n \times d$ cas qui se produit lorsque toutes les valeurs des domaines des variables ont été retirées une à une[8].

7.4.2 Complexité spatiale

L'occupation en mémoire de notre approche est due au stockage des explications (de déduction, de contradiction, de retrait de contrainte).

Il faut donc déterminer le nombre maximum d'explications associées à une déduction donnée et la taille de celles-ci.

8. Nous considérons ici la condition de fonctionnement `C2` qui garantit l'unicité d'explication pour un retrait de valeur (*cf.* lemme 3) : les retraits de valeurs consécutives ne peuvent donc se chevaucher.

Taille des explications

Quel que soit l'état du paramètre `statique`[9], une explication est constituée dans le pire cas (que ce soit pour une explication de contradiction ou de retrait) d'au plus c contraintes *i.e.* l'occupation en espace d'une explication est en $O(n+e)$.

Nombre d'explications

Dans le cas général, il peut y avoir un nombre exponentiel d'explications de déduction sauf si on est dans l'état de fonctionnement **C2** où il y a unicité d'explication pour une déduction (*cf.* lemme 3 page 72).

Le nombre d'explications de contradiction et donc d'explications de retrait est lui aussi exponentiel dans le pire cas (il est quand même borné[10] par $2^{c+(d-1)\times n}$). Par contre, avec un comparateur contradiction-local (comme C_{MM}), il y a unicité pour les explications de retrait de contrainte et pour les explications de contradiction conservées. Plus exactement, comme les relaxations de contraintes *normales* sont définitives, il est inutile de conserver une explication pour ces retraits ; seuls les retraits de contraintes provenant d'un `one-of` doivent être expliqués.

Ainsi, en tenant compte de l'énumération et sous la condition de fonctionnement **C2**, le nombre de d'explications stockées est en $c = O(n \times d)$.

Taille de la file de propagation

La file de propagation ne concerne que des retraits de valeur, il y en a donc au plus a dans la file puisque l'on se trouve dans un algorithme de réduction. Bien sûr, dès que la configuration courante est modifiée, il faut vider la file de propagation des retraits qui ne sont plus valides.

Complexité spatiale

- Sous la condition de fonctionnement **C2**, en utilisant un comparateur *contradiction-local* et avec le paramètre `enregistrement off`, la complexité spatiale de notre approche est en $O(c \times a) = O(n \times d \times (n+e))$.

- Sinon, la complexité spatiale est potentiellement exponentielle.

7.4.3 Complexité temporelle

La complexité temporelle globale de l'approche reste exponentielle (on résout un problème NP-Complet). Nous donnons la complexité temporelle des algorithmes principaux de notre système en termes d'opérations de base.

Nous considérons comme opérations de base :

- l'association d'une explication à une déduction ;

- la consultation d'une étiquette (l'environnement associé à l'explication) pour une déduction ; on suppose alors que cette étiquette est bien en accès direct. Cette structure doit être maintenue malgré d'éventuelles modifications – principalement remplacement d'une explication lorsque le paramètre `statique` est `off`.

9. définition 45 section 6.2.3 page 73.
10. C'est l'ensemble de toutes les explications que l'on peut déterminer: c'est le nombre de sous-ensemble de contraintes.

Ajout de Contrainte

La complexité de l'ajout de contrainte (algorithme 4 page 76) est dépendante de l'implémentation et de l'utilisation faite des fonctions `prop` et `localprop` de l'algorithme 3.

Comme nous l'avons vu, on peut modifier cet algorithme (on l'a montré pour l'algorithme 2 dans la section 7.2.2) pour obtenir un fonctionnement identique aux algorithmes AC3, AC4 ou AC5.

La complexité d'un tel algorithme est exprimée en nombre de tests de consistance sur une contrainte. Notons s cette complexité.

Lorsqu'un test de consistance échoue, les valeurs correspondantes sont retirées du domaine de la variable concernée : il s'agit d'un appel à la fonction `dms-retrait` (algorithme 5). Soit r la complexité de cette opération de retrait.

La complexité a_c de l'algorithme d'ajout de contrainte (algorithme 3) est donc en $O(s \times r)$. On suppose ici que l'ajout de contrainte ne conduit pas à une contradiction. Nous verrons plus loin la complexité du traitement d'une contradiction.

Complexité de l'algorithme 5 Lorsque `statique` est `on`, l'algorithme 5 (algorithme `dms-retrait`), est en $O(d \times d)$. Une boucle ayant au plus d pas et un calcul d'environnement à chaque pas (au plus d opérations).

Lorsque `statique` est `off`. Il y a une étape de propagation qui peut nous conduire à parcourir le graphe de dépendance des déductions (au plus a opérations). La complexité est alors en $O(d \times a) = O(n \times d^2)$.

En fait, lorsqu'on ajoute une contrainte, le nombre total des suppressions de valeurs pour une même variable est au plus d, compter d pas à chaque suppression conduit à une sur-évaluation de la complexité dans le pire cas réelle. On divisera donc les complexités précédentes par d pour obtenir une complexité marginale unitaire.

Complexité de l'ajout de contrainte Le tableau 7.4.3 récapitule les complexités temporelles obtenues pour l'ajout de contrainte en fonction de diverses valeurs de paramètres.

Algorithme simulé	Complexité classique	`statique on`	`statique off`
	s	$s \times r$	$s \times r$
AC4	$e \times d^2$	$e \times d^3$	$e \times n \times d^3$
AC5	$e \times d$	$e \times d^2$	$e \times n \times d^2$

TAB. 7.1 – *Complexité temporelle de l'ajout de contrainte*

Notons que la complexité temporelle donnée lors de l'utilisation de l'algorithme AC5 est vraiment la complexité au pire : lorsqu'il n'y a que des contraintes monotones. Lorsqu'il n'y en a pas, la complexité du retrait (lorsque `statique` est `on`) est en $O(1)$ car les déductions utilisées n'ont qu'une partie *déduction* réduite à un singleton.

Traitement d'une contradiction

Nous ne traiterons ici que des appels aux fonctions `explique-contradiction` et `meilleure-configuration`. Nous considérons la fonction `maj-configuration` comme une opération indépendante du traitement d'une contradiction.

La fonction `explique-contradiction` est chargée de fournir une explication de contradiction. Comme nous l'avons vu dans la section 6.3.2, il s'agit ici de déterminer la réunion des explications associées à chacune des valeurs du domaine de la variable provoquant la contradiction. Il y en a au plus d. La réunion se fait donc en $O(d)$

si on considère les ensembles représentés par des tableaux de bits (les opérations élémentaires étant réalisables en $O(1)$).

La fonction `meilleure-configuration` dépend principalement du comparateur utilisé. Dans le cas de l'utilisation d'un comparateur *contradiction-local* (comme C_{MM}), il suffit de déterminer une contrainte parmi celles de la dernière explication de contradiction calculée : cette opération se réalise en $O(c)$ (c taille d'une explication de contradiction) dans le cas de C_{MM} (détermination de la contrainte la moins importante). Sinon, c'est un problème difficile, il n'existe donc pas *a priori* d'algorithme polynômial.

La complexité du traitement de la contradiction sans compter le changement de configuration est donc en $O(d+c) = O(n+e+d)$.

Changement de configuration

- **Retrait de contrainte** Nous reprenons la définition de la fonction `point-fixe-retrait` selon l'algorithme 6 page 78.

  La fonction `retrait-possible` peut être implémentée en $O(a)$, le nombre de contraintes. Il faut en effet tester toutes les contraintes portant sur la variable traitée.

  La fonction `propage-retrait` équivaut en fait à un appel à la fonction d'ajout de contrainte. La complexité de `propage-retrait` est donc la complexité de l'algorithme d'ajout de contrainte : $O(s \times r)$.

- **Retour de contrainte** Un tel retour (*cf.* algorithme 7 page 79) est réalisé en deux phases :

  - une première phase de rétablissement des déductions redevenues valides[11]. Il y en a au plus a ;
  - une deuxième phase correspondant à un ajout de contrainte. Sa complexité est donc en $O(s \times r)$.

7.4.4 Récapitulation

Le tableau 7.2 récapitule les résultats sur la complexité lorsqu'on utilise le comparateur C_{MM}. Rappelons que lorsque le paramètre `enregistrement` est à `off`, seule la dernière explication de contradiction est conservée. Rappelons de plus que lorsque la condition de fonctionnement **C2** est vérifiée, le paramètre `statique` est `on`.

7.5 Discussion

Nous donnons ici quelques paramètres réglables pour le fonctionnement de notre système dans le cas des CSP. Nous commençons par traiter le cas où les conditions **C1** et **C2** ne sont pas vérifiées.

Vouloir pouvoir ajouter des explications de retrait non valides au moment où elles sont recherchées (*i.e.* ne pas respecter la condition **C1**) nous paraît totalement déraisonnable car cela nous conduirait à rechercher toutes les explications possibles à la manière d'un ATMS ce qui conduirait immanquablement à une explosion combinatoire de l'approche proposée.

11. En effet, même dans le cas de la condition de fonctionnement `C2` qui supprime de fait certaines explications invalides, il peut en subsister car celles-ci ne sont pas supprimées dès qu'elles le deviennent mais simplement remplacées s'il y a lieu. Dans une implémentation pratique, il n'est pas utile de tenir compte de cette étape (car beaucoup d'informations ont potentiellement disparu) et considérer tout simplement le retour d'une contrainte comme une première apparition.

Complexité Spatiale	
`enregistrement off`	`enregistrement on`
$O(n \times d \times (n+e))$	exponentielle.

Complexité Temporelle			
Opération	Algorithme simulé	`statique on`	`statique off`
ajout de contrainte	AC4	$O(e \times d^3)$	$O(end^3)$
retrait de contrainte retour de contrainte	AC5	$O(e \times d^2)$	$O(end^2)$
Traitement contradiction		$O(n+e+d)$	

TAB. 7.2 – *Complexité de l'approche – Récapitulation*

Par contre, on peut vouloir ne pas respecter la condition **C2** *i.e.* mettre non seulement le paramètre `enregistrement` (présenté section 7.3.3) à `on` en ce qui concerne les explications de retrqaits et de contradiction mais aussi autoriser l'enregistrement de plusieurs explications pour un même retrait. Ainsi, toutes les informations connues sont conservées pour prévoir le retour d'une contrainte. Cette façon de faire suppose alors que :

- les valeurs sont retirées une à une des domaines des variables[12] ;
- le paramètre `statique` (présenté section 6.2.3) doit rester à `on` car sinon le bien-fondé du système d'explications ne serait plus garanti.

La non vérification de la condition **C2** peut entraîner très rapidement une explosion dans l'occupation en espace pour le stockage des explications. Il faut donc réfléchir à des mécanismes de simplification de l'information enregistrée. Il faut supprimer[13] l'information qui a le moins de chance de nous servir.

7.6 Exemple complet : la conférence

Nous présentons ici un exemple que nous traitons entièrement. Il s'agit d'une adaptation du problème de la conférence proposé par Cousin [1991].

7.6.1 Énoncé et modélisation du problème

Considérons la scène suivante, événement quotidien dans un laboratoire de recherche dynamique.

> Michel, Pierre et Alain se rencontrent inopinément dans un couloir. Michel interpelle les deux autres et leur rappelle leur promesse de se rencontrer pour se présenter leurs travaux. Pierre et Alain doivent présenter leurs travaux à Michel qui lui même doit leur présenter ses travaux. Il y a donc 4 exposés : de Pierre à Michel, d'Alain à Michel, de Michel à Pierre et enfin de Michel à Alain.
>
> Ils conviennent de bloquer 4 demi-journées pour faire ces exposés. Un exposé prend une demi-journée.
>
> Dès le début, Michel signale qu'il lui paraît important d'*avoir vu ce qu'ont à dire Pierre et Alain avant* de faire ses présentations. Ensuite,

12. En effet, on ne peut plus supprimer un ensemble de valeurs en une seule fois, car ceci suppose que **C2** soit vérifée (*cf.* section 6.2.4).

13. Dans le pire, elle sera recalculée. On n'évitera alors pas complètement le phénomène de *thrashing*.

> Michel intervient pour signaler qu'il *aimerait autant ne pas venir la quatrième demi-journée*. Enfin, Michel signale qu'il *ne veut pas* présenter ses travaux à Pierre et Alain en même temps.

Ce problème met donc en jeu 4 variables : P_m, A_m, M_p et M_a chacune représentant un exposé devant être réalisé. Les préférences (que l'on peut identifier en italiques dans le texte) sont données de manière qualitative. Nous donnerons dans la présentation des contraintes un poids[14] à ces contraintes. Le comparateur utilisé sera C_{MM}.

Le domaine initial de ces 4 variables est le même : $\{1, 2, 3, 4\}$, chaque valeur représentant une demi-journée.

Les contraintes du problème sont alors les suivantes (dans leur ordre d'apparition) :

- **Contraintes d'intégrité**
  Une personne ne peut être à la fois auditeur et orateur. Ce qui peut s'exprimer par : $M_a \neq A_m$, $M_a \neq P_m$, $M_p \neq A_m$ et $M_p \neq P_m$.

  Une personne ne peut assister à deux exposés à la fois, ce qui s'exprime par : $c_1 : A_m \neq P_m$. C'est une contrainte requise de poids 100.

- **Contraintes d'antériorité**
  Le fait que Michel souhaite entendre les présentations de Pierre et Alain avant de faire sa présentation peut s'écrire[15] ainsi : $c_2 : M_a > A_m$, $c_3 : M_a > P_m$, $c_4 : M_p > A_m$ et $c_5 : M_p > P_m$. Ces contraintes sont importantes, elles ont toutes un poids 3.

- **Contraintes de la quatrième demi-journée**
  Le fait que Michel ne veut pas venir lors de la 4^{eme} demi-journée peut s'écrire : $c_6 : M_a \neq 4$, $c_7 : M_p \neq 4$, $c_8 : P_m \neq 4$ et $c_9 : A_m \neq 4$. Ces contraintes sont accessoires, elles ont un poids 1.

- **Contraintes de non-simultanéité**
  Le fait que Michel ne veut pas présenter ses travaux à Pierre et Alain en même temps, s'écrit : $c_{10} : M_a \neq M_p$. Cette contrainte est relativement importante, elle a un poids 2.

Nous allons donc introduire ces contraintes dans leur ordre d'apparition puis déclencher une énumération (à l'aide de la contrainte `one-of`) si les variables ne sont pas instanciées.

7.6.2 Résolution

Le tableau 7.3 rappelle pour chaque variable quelles sont les contraintes concernées.

Ajout des contraintes

- L'ajout de la contrainte c_1 ($A_m \neq P_m$) ne provoque aucune modification dans le système de maintien de déduction. En effet, une telle contrainte antibijective n'a d'effet que si l'une des variables concernées a son domaine courant réduit à un singleton.

14. Rappelons que plus ce poids est fort, plus la contrainte est importante.

15. Ces contraintes «*impliquent*» les premières contraintes d'intégrité. C'est pourquoi nous ne tiendrons pas compte de ces dernières lors de la résolution. Cela nous permettra d'alléger la présentation de l'exemple. C'est aussi pour cela qu'elles ne sont pas identifiées par un numéro.

Variables	Contraintes
P_m	c_1, c_3, c_5, c_8
A_m	c_1, c_2, c_4, c_9
M_p	c_4, c_5, c_7, c_{10}
M_a	c_2, c_3, c_6, c_{10}

TAB. 7.3 – *Associations Variables / Contraintes*

- L'ajout de la contrainte c_2 $(M_a > A_m)$ conduit, d'après la proposition 16 sur le traitement des contraintes monotones (*cf.* page 84), aux retraits[16] suivants.

$$(M_a, 1, (\{c_2\}, \varnothing), \{c_2\}) \tag{7.1}$$

$$(A_m, 4, (\{c_2\}, \varnothing), \{c_2\}) \tag{7.2}$$

  Le retrait 7.2 n'a pas besoin d'être propagé sur la contrainte c_1 car le domaine de A_m n'est pas un singleton.

- L'ajout de la contrainte c_3 $(M_a > P_m)$ conduit au retrait suivant :

$$(P_m, 4, (\{c_3\}, \varnothing), \{c_3\}) \tag{7.3}$$

  La contrainte c_1 n'est pas réveillée par le retrait survenu dans le domaine de P_m.

- L'ajout de la contrainte c_4 $(M_p > A_m)$ conduit au retrait suivant :

$$(M_p, 1, (\{c_4\}, \varnothing), \{c_4\}) \tag{7.4}$$

- L'ajout de la contrainte c_5 $(M_p > P_m)$ ne conduit à aucune modification dans les domaines des variables.

- L'ajout de la contrainte c_6 $(M_a \neq 4)$ provoque dans un premier temps le retrait suivant :

$$(M_a, 4, (\{c_6\}, \varnothing), \{c_6\}) \tag{7.5}$$

  Dans un second temps, les contraintes c_2 et c_3 sont réveillées. Le réveil de la contrainte c_2 conduit au retrait suivant :

$$(A_m, 3, (\{c_2\}, \{\delta(M_a, 4)\}), \{c_2, c_6\}) \tag{7.6}$$

  Le réveil de la contrainte c_3 quant à lui conduit au retrait suivant :

$$(P_m, 3, (\{c_3\}, \{\delta(M_a, 4)\}), \{c_3, c_6\}) \tag{7.7}$$

  Le réveil des contraintes c_1, c_4 et c_5 ne provoque aucune autre modification.

- L'ajout de la contrainte c_7 $(M_p \neq 4)$ conduit au retrait suivant :

$$(M_p, 4, (\{c_7\}, \varnothing), \{c_7\}) \tag{7.8}$$

  Le réveil des contraintes ne conduit à aucune autre modification.

- L'ajout des contraintes c_8 $(P_m \neq 4)$, c_9 $(A_m \neq 4)$et c_{10} $(M_a \neq M_p)$ ne conduit à aucune modification.

Variable	Valeur	Explication	Présente ?	Variable	Valeur	Explication	Présente ?
P_m	1	$\varnothing$	**oui**	A_m	1	$\varnothing$	**oui**
P_m	2	$\varnothing$	**oui**	A_m	2	$\varnothing$	**oui**
P_m	3	$\{c_3, c_6\}$	non	A_m	3	$\{c_2, c_6\}$	non
P_m	4	$\{c_3\}$	non	A_m	4	$\{c_2\}$	non
M_p	1	$\{c_4\}$	non	M_a	1	$\{c_2\}$	non
M_p	2	$\varnothing$	**oui**	M_a	2	$\varnothing$	**oui**
M_p	3	$\varnothing$	**oui**	M_a	3	$\varnothing$	**oui**
M_p	4	$\{c_7\}$	non	M_a	4	$\{c_6\}$	non

TAB. 7.4 – *État des domaines avant énumération*

Le tableau 7.4 représente l'état courant des domaines et récapitule les explications associées à chacune des suppressions dans les domaines des variables concernées.

La figure 7.5 représente les dépendances entre retraits.

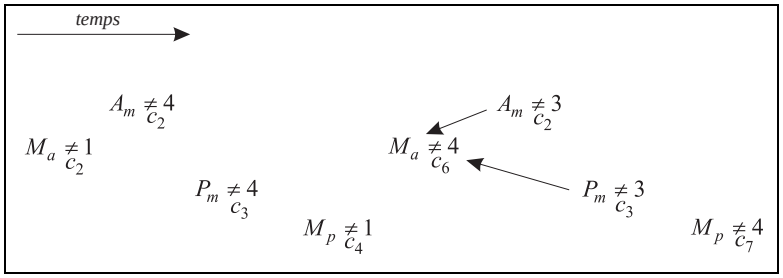

FIG. 7.5 – *Dépendance entre retraits avant énumération*

Les variables de notre problème n'étant pas complètement instanciées (domaines non réduits à un singleton), nous passons à la phase d'énumération.

Énumération

Nous choisissons de débuter l'énumération sur la variable A_m. Les contraintes $c_1^{am} : A_m = 1$, $c_2^{am} : A_m = 2$, $c_3^{am} : A_m = 3$ et $c_4^{am} : A_m = 4$ sont créées et regroupées dans une contrainte `one-of` c^{am}.

- La contrainte c_1^{am} est donc ajoutée. Cette contrainte provoque le retrait suivant :

$$(A_m, 2, (\{c_1^{am}\}, \varnothing), \{c_1^{am}\}) \tag{7.9}$$

  Les contraintes c_1, c_2 et c_4 doivent être réveillées.

- Le réveil de la contrainte c_1 provoque le retrait :

$$(P_m, 1, (\{c_1\}, \{\delta(A_m, 2), \delta(A_m, 3), \delta(A_m, 4)\}), \{c_1^{am}, c_1, c_2, c_6\}) \tag{7.10}$$

  Les contraintes à réveiller sont maintenant les contraintes c_2, c_4, c_3 et c_5.

- Le réveil des contraintes c_2 et c_4 ne provoque aucune modification. Celui de la contrainte c_3 provoque le retrait suivant :

$$(M_a, 2, (\{c_3\}, \{\delta(P_m, 1)\}), \{c_1^{am}, c_1, c_2, c_3, c_6\}) \tag{7.11}$$

16. Un retrait est un quadruplet (a, b, c, d) précisant le retrait de la valeur b du domaine de la variable a avec l'explication c dont l'environnement associé est d.

Les contraintes à réveiller sont maintenant les contraintes c_5, c_2 et c_{10}.

- Le réveil de la contrainte c_5 conduit au retrait suivant :

$$(M_p, 2, (\{c_5\}, \{\delta(P_m, 1)\}), \{c_1^{am}, c_1, c_2, c_5, c_6\}) \tag{7.12}$$

Les contraintes à réveiller sont maintenant les contraintes c_2, c_{10} et c_4.

- Le réveil de la contrainte c_2 n'introduit aucune modification. Par contre, le réveil de la contrainte c_{10} provoque le retrait :

$$(M_p, 3, (\{c_{10}\}, \{\delta(M_a, 1), \delta(M_a, 2), \delta(M_a, 4)\}), \{c_1^{am}, c_1, c_2, c_3, c_6, c_{10}\}) \tag{7.13}$$

Le domaine de la variable M_p est vide. Il y a donc une contradiction. Le tableau 7.5 représente la situation des domaines lors de cette contradiction.

Variable	Valeur	Explication	Présente?
P_m	1	$\{c_1^{am}, c_1, c_2, c_6\}$	non
P_m	2	$\varnothing$	**oui**
P_m	3	$\{c_3, c_6\}$	non
P_m	4	$\{c_3\}$	non
A_m	1	$\varnothing$	**oui**
A_m	2	$\{c_1^{am}\}$	non
A_m	3	$\{c_2, c_6\}$	non
A_m	4	$\{c_2\}$	non
M_p	1	$\{c_4\}$	*non*
M_p	2	$\{c_1^{am}, c_1, c_2, c_5, c_6\}$	*non*
M_p	3	$\{c_1^{am}, c_1, c_2, c_3, c_6, c_{10}\}$	*non*
M_p	4	$\{c_7\}$	*non*
M_a	1	$\{c_2\}$	non
M_a	2	$\{c_1^{am}, c_1, c2, c3, c_6, c_{10}\}$	non
M_a	3	$\varnothing$	**oui**
M_a	4	$\{c_6\}$	non

TAB. 7.5 – *État des domaines à la première contradiction*

Traitement de la première contradiction

- La fonction `explique-contradiction` fournit à partir des informations de l'ensemble $\{\delta(M_p, i) \mid i \in [1, 4]\}$ l'explication de contradiction suivante :

$$\{c_1^{am}, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\} \tag{7.14}$$

- La fonction `meilleure-configuration` choisit alors de relaxer la contrainte c_1^{am} puisqu'il s'agit de la moins importante[17] de l'explication 7.14. Son explication de retrait est alors $\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}$.

- La fonction `maj-configuration` rétablit la cohérence comme si la contrainte c_1^{am} n'était jamais apparue. On se retrouve donc dans la situation avant énumération.

17. Rappelons qu'une contrainte d'énumération est neutre vis à vis du comparateur quel qu'il soit.

- On peut identifier le retrait d'une valeur du domaine d'une variable et le retrait de la contrainte d'énumération qui lui est associée. Ainsi, le retrait de la contrainte c_1^{am} conduit au retrait suivant :

$$(A_m, 1, (\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}, \varnothing), \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}) \quad (7.15)$$

  Il faut propager les contraintes c_1, c_2 et c_4.

- Le réveil de la contrainte c_1 provoque le retrait :

$$(P_m, 2, (\{c_1\}, \{\delta(A_m, 1), \delta(A_m, 3), \delta(A_m, 4)\}), \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}) \quad (7.16)$$

  Il faut alors propager les contraintes c_2, c_4, c_3 et c_5.

- Le réveil de la contrainte c_2 provoque le retrait :

$$(M_a, 2, (\{c_2\}, \{\delta(A_m, 1)\}), \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}) \quad (7.17)$$

  Il faut propager les contraintes c_4, c_3, c_5 et c_{10}.

- Le réveil de la contrainte c_4 conduit au retrait :

$$(M_p, 2, (\{c_4\}, \{\delta(A_m, 1)\}), \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}) \quad (7.18)$$

  Les contraintes à propager sont : c_3, c_5 et c_{10}.

- Le réveil des contraintes c_3 et c_5 ne modifie rien de plus. Le réveil de la contrainte c_{10} provoque le retrait :

$$(M_p, 3, (\{c_{10}\}, \{\delta(M_a, 1), \delta(M_a, 2), \delta(M_a, 4)\}), \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}) \quad (7.19)$$

  Le domaine de la variable M_p est vide. Il y a donc une contradiction. Le tableau 7.6 récapitule l'état des domaines à la deuxième contradiction.

Variable	Valeur	Explication	Présente?
P_m	1	$\{c_1^{am}, c_1, c_2, c_6\}$	**oui**
P_m	2	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}$	non
P_m	3	$\{c_3, c_6\}$	non
P_m	4	$\{c_3\}$	non
A_m	1	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}$	non
A_m	2	$\{c_1^{am}\}$	**oui**
A_m	3	$\{c_2, c_6\}$	non
A_m	4	$\{c_2\}$	non
M_p	1	$\{c_4\}$	*non*
M_p	2	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}$	*non*
M_p	3	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}$	*non*
M_p	4	$\{c_7\}$	*non*
M_a	1	$\{c_2\}$	non
M_a	2	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}$	non
M_a	3	$\varnothing$	**oui**
M_a	4	$\{c_6\}$	non

TAB. 7.6 – *État des domaines à la deuxième contradiction*

Traitement de la deuxième contradiction

- La fonction `explique-contradiction` fournit à partir des informations de l'ensemble $\{\delta(M_p, i) \mid i \in [1,4]\}$ l'explication de contradiction suivante :

$$\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\} \tag{7.20}$$

- La fonction `meilleure-configuration` choisit alors de relaxer la contrainte c_7 la plus récente des moins importantes de l'explication de contradiction 7.20. Son explication de retrait est alors $\{c_1, c_2, c_3, c_4, c_5, c_6, c_{10}\}$.

- La fonction `maj-configuration` rétablit la cohérence comme si la contrainte c_7 n'avait jamais été introduite. On se retrouve alors dans la situation dépeinte dans le tableau 7.7.

Variable	Domaine Courant
P_m	$\{1,2\}$
A_m	$\{1,2\}$
M_p	$\{2,3,4\}$
M_a	$\{2,3\}$

TAB. 7.7 – *Situation après relaxation de c_7*

- Depuis la première contradiction la contrainte `one-of` c^{am} n'a toujours pas été réveillée. Son réveil provoque le retour de la contrainte c_1^{am} car son explication de retrait n'est plus valide.

- On peut ainsi rétablir la suppression de 1 depuis le domaine de P_m puisque l'explication associée redevient valide. On peut aussi rétablir le retrait de 2 depuis le domaine de A_m pour la même raison [18].

- La propagation du retour de la contrainte c_1^{am} provoque les retraits suivants :

$$(M_a, 2, (\{c_3\}, \{\delta(P_m, 1)\}), \{c_1^{am}, c_1, c_2, c_3, c_6\}) \tag{7.21}$$

$$(M_p, 2, (\{c_5\}, \{\delta(P_m, 1)\}), \{c_1^{am}, c_1, c_2, c_5, c_6\}) \tag{7.22}$$

$$(M_p, 3, (\{c_{10}\}, \{\delta(M_a, 1), \delta(M_a, 2), \delta(M_a, 4)\}), \{c_1^{am}, c_1, c_2, c_3, c_6, c_{10}\}) \tag{7.23}$$

- Le tableau 7.8 récapitule la situation finale pour les domaines des variables. Toutes les variables sont donc instanciées et bien sûr le CSP obtenu est arc-consistant. On a donc montré plusieurs choses.

  - le CSP originel est sur-contraint, il n'existait pas d'instanciation vérifiant toutes les contraintes. On l'a montré lorsqu'on a découvert l'explication de contradiction 7.20 ;
  - relaxer une unique contrainte (en l'occurrence c_7) parmi les moins importantes du problème permet de trouver une solution ;
  - la configuration proposée est la meilleure suivant le comparateur C_{MM}.

18. Dans l'implémentation que nous utilisons présentée dans le chapitre 9, nous ne recherchons pas ces explications non valides qui redeviennent valides, nous laissons le solver s'en charger.

Variable	Valeur	Explication	Présente?
P_m	1	$\{c_1^{am}, c_1, c_2, c_6\}$	non
P_m	2	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}$	**oui**
P_m	3	$\{c_3, c_6\}$	non
P_m	4	$\{c_3\}$	non
A_m	1	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_{10}\}$	**oui**
A_m	2	$\{c_1^{am}\}$	non
A_m	3	$\{c_2, c_6\}$	non
A_m	4	$\{c_2\}$	non
M_p	1	$\{c_4\}$	non
M_p	2	$\{c_1^{am}, c_1, c_2, c_5, c_6\}$	non
M_p	3	$\{c_1^{am}, c_1, c_2, c_3, c_6, c_{10}\}$	non
M_p	4	$\{c_7\}$	**oui**
M_a	1	$\{c_2\}$	non
M_a	2	$\{c_1^{am}, c_1, c2\_c3, c_6, c_{10}\}$	non
M_a	3	$\varnothing$	**oui**
M_a	4	$\{c_6\}$	non

TAB. 7.8 – *État des domaines à la fin de la résolution*

7.7 Conclusion

Nous avons présenté dans ce chapitre une spécialisation complète de notre approche pour traiter les CSP. Nous avons ainsi pu constater qu'une utilisation raisonnable (ce qu'attestent les complexités données) et raisonnée (les conditions de fonctionnement proposées) d'un système de maintien de déductions était possible. Nous en montrerons une justification expérimentale dans les chapitres 10 et 11.

Chapitre 8

Instance sur les intervalles : relax(Interval,$\mathcal{C}$,$\mathcal{P}_{bc}$)

Sommaire

DANS CE CHAPITRE, nous montrons comment notre modèle général peut s'appliquer aux CSP numériques munis de la consistance aux bornes : relax(Interval, $\mathcal{C}$, $\mathcal{P}_{bc}$).

Nous présentons tout d'abord brièvement le cadre général des CSP numériques et nous montrons comment spécialiser l'algorithme de réduction proposé chapitre 6.

Comme dans le chapitre précédent, nous montrons ensuite comment le système de maintien de déductions peut être adapté pour ce cas particulier et évoquons le cas des contraintes disjonctives (provoquant des *trous* dans les intervalles).

Ce chapitre est, à notre connaissance, la première véritable proposition de système de relaxation de contraintes sur les intervalles et par la même une version dynamique des algorithmes de réduction d'intervalle. Nous présentons ici les grandes lignes d'un tel système et nous ouvrons la voie à une implémentation.

8.1 Le cadre général des CSP numériques

Cette présentation est basée sur les travaux d'Olivier Lhomme [1994].

8.1.1 Présentation

Les problèmes provenant du monde physique font fréquemment intervenir des contraintes numériques non linéaires, des données imprécises ou des paramètres mal définis. Les valeurs numériques traitées sont donc souvent des intervalles de réels.

8.1.2 Réels et domaines finis

L'implémentation des nombres réels en machine repose généralement sur une représentation en virgule flottante munie de mécanismes d'arrondis qui garantissent la correction des opérations.

Olivier Lhomme propose la notion de *valeur numérique abstraite* (*vna*) qui permet de rester indépendant de la représentation des nombres manipulés et de l'implémentation en machine des fonctions mathématiques.

Définition 58 (Valeur numérique abstraite – [Lhomme, 1994])
Soit $\mathcal{F} = \{f_0, f_1, \ldots, f_n\}$ un ensemble fini ordonné permettant de représenter les nombres réels. Une ***valeur numérique abstraite*** *est un élément de l'ensemble $\mathcal{A}$:*

$$\mathcal{A} = \{]-\infty, f_0[, f_0,]f_0, f_1[, f_1,]f_1, f_2[, f_2, \ldots, f_n,]f_n, +\infty[\}$$

Une valeur numérique abstraite est alors un objet représentant un ensemble E de valeurs :

- $E = \{f\}$ avec $f \in \mathcal{F}$.
- $E = \{x \mid x \in]f, f^+[\}$ avec f et f^+ deux éléments de $\mathcal{F}$ consécutifs.
- $E = \{x \mid x \in]-\infty, f_0[\}$.
- $E = \{x \mid x \in]f_n, +\infty[\}$.

Une *vna* correspondant à un ensemble infini de réels est considérée comme une *valeur unique*. Ainsi, les *vna* dénotées par $]-\infty, f_0[\}$ et $]f_n, +\infty[\}$ représentent sous un même nom, non seulement les infinis mathématiques mais aussi, respectivement, toutes les valeurs plus petites que f_0 ou plus grandes que f_n.

On se donne une fonction d'approximation $\alpha_{\mathcal{A}}$ qui à tout réel associe une *vna*. On peut alors définir une extension des relations et fonctions réelles sur les *vna* de manière tout à fait intuitive (*cf.* [Lhomme et Rueher, 1996] ou [Lhomme, 1994]).

Ainsi, on peut considérer le domaine d'une variable réelle comme un ensemble fini tout en préservant la continuité du domaine. Un CSP numérique peut alors être vu comme un CSP «*classique*». Les mécanismes mis en œuvre dans le chapitre précédent peuvent alors être utilisés pour traiter les CSP numériques. Il faut tout de même noter que les domaines des variables concernées peuvent être de grande taille [1]. Il paraît donc illusoire de les traiter de la même façon que les CSP. C'est pourquoi nous proposons de spécialiser l'algorithme de réduction et le système de maintien de déduction pour les CSP numériques.

Pour les subtilités propres aux problèmes numériques, comme la précision, les problèmes de cycle, ... on se reportera aux travaux d'Olivier Lhomme [Lhomme, 1993; Lhomme, 1994; Lhomme *et al.*, 1996; Lhomme et Rueher, 1996].

8.2 Spécialisation de l'algorithme de réduction

Nous donnons ici la définition d'une consistance partielle pour les intervalles : la consistance aux bornes. Nous définissons ensuite une fonction d'approximation (au sens de la définition 31 page 64) pour les intervalles qui nous permettra d'appliquer le théorème 7 liant notre algorithme de réduction (algorithme 2 page 66) et la propriété que l'on désire maintenir pour un problème donné.

8.2.1 Consistance aux bornes

Les domaines originaux des variables pour un CSP numérique sont la plupart du temps constitués de *vna* consécutives. Ils peuvent donc être représentés par un intervalle :

Définition 59 (Intervalle – [Lhomme, 1994])
Soient v_1 et v_2 deux vna, un ***intervalle*** $[v_1, v_2]$ *représente l'ensemble des vna comprises entre v_1 et v_2 :*

$$[v_1, v_2] = \{\alpha_{\mathcal{A}}(x) \mid x \in \mathbb{R}, v_1 \leqslant \alpha_{\mathcal{A}}(x) \leqslant v_2\}$$

$\mathcal{I}(\mathcal{A})$ représente l'ensemble des intervalles définis sur $\mathcal{A}$.

Cette notion d'intervalle est intéressante car, dans le domaine des grandeurs physiques, il est souvent plus intéressant de connaître les domaines de variation des variables d'un problème plutôt qu'une instanciation permettant de vérifier toutes les contraintes du problème.

> **Exemple 31 (Intérêt des domaines de variations) :**
> Nous reprenons un exemple présenté dans [Lhomme et Rueher, 1996]. Soient trois variables réelles liées par une contrainte : $U = R \times I$. Si I varie entre 1 et 2 (domaine : intervalle $[1, 2]$) et que R varie entre 10 et 11 (domaine : intervalle $[10, 11]$), il est plus intéressant de connaître le domaine de variation de U (*i.e.* l'intervalle $[10, 22]$) qu'une solution (par exemple, $R = 10.53$, $I = 1$ et $U = 10.53$).

Comme le signale [Lhomme, 1994], on peut *limiter* l'arc-consistance aux bornes des domaines traités. C'est ce qu'on appelle la **B-consistance d'arc**.

1. Pour donner un ordre de grandeur, le nombre de flottants entre 0.0 et 1000.0 est approximativement de 4.6×10^{18} en double précision. Il peut donc y avoir 9.2×10^{18} *vna*.

Définition 60 (B-consistance d'arc)
Soient $\mathcal{P}$ un CSP*, v une variable de $\mathcal{P}$ et $D_v = [a, b]$. D_v est* ***B-consistant d'arc*** *si et seulement si :*

$$\begin{array}{rcl} \forall c, var(c) & = & \{v, v_1, \ldots, v_k\} \\ & & \exists a_1 \ldots a_k \in D_{v_1} \times \ldots \times D_{v_k} \mid c \text{ vérifiée pour } \{a, a_1, \ldots, a_k\} \\ & & \exists b_1 \ldots b_k \in D_{v_1} \times \ldots \times D_{v_k} \mid c \text{ vérifiée pour } \{b, b_1, \ldots, b_k\} \end{array}$$

Un CSP *est* ***B-consistant d'arc*** *si et seulement si tous ses domaines sont B-consistants d'arc.*

8.2.2 Fonction d'approximation pour un CSP numérique

Considérons la fonction *hull* sur $\mathcal{P}(D)$. Cette fonction calcule la couverture convexe d'un élément de $\mathcal{P}(D)$ *i.e.* le plus petit intervalle fermé de *vna* contenant l'élément considéré.

Proposition 27 (*hull* est une approximation)
La fonction hull est une fonction d'approximation.

> ▷ **Preuve :** Reprenons la définition 31. Par définition de la couverture convexe, on a :
>
> 1. $hull(\varnothing) = \varnothing$
> 2. $\forall \rho \in \mathcal{P}(D), \rho \subset hull(\rho)$
> 3. $\forall \rho, \rho' \in \mathcal{P}(D), \rho \subset \rho' \Rightarrow hull(\rho) \subset hull(\rho')$
> 4. $\forall \rho \in \mathcal{P}(D), hull(hull(\rho)) = hull(\rho)$
>
> *hull* est donc bien une fonction d'approximation. ◁

On note aussi *hull* l'extension de *hull* à $\mathcal{P}(D^n)$ telle que précisée par la définition 32 page 65.

On peut montrer le résultat suivant [Lhomme, 1994].

Théorème 14 (*hull* et B-consistance d'arc)
Soit D un ensemble. Soit $u \subset D^n$ et ρ une relation n-aire sur D. ρ est B-consistant d'arc pour u si et seulement si :

$$\overline{\rho}_{hull}(u) = u$$

Nous sommes donc exactement dans les conditions d'applications du théorème 7 liant le point fixe de l'algorithme de réduction et la propriété à vérifier. L'algorithme 2 pour lequel la fonction d'approximation utilisée est *hull* fournit donc un algorithme de filtrage par B-consistance d'arc. On peut donc utiliser les résultats du chapitre 6 pour proposer un système de relaxation de contraintes sur les CSP numériques.

8.3 Spécialisation du système de maintien de déduction

L'utilisation des intervalles dans le cadre des CSP numériques conduit à diverses adaptations par rapport aux résultats présentés dans le chapitre 6. Ainsi, il est possible de simplifier le mécanisme d'enregistrement, ce qui conduit d'ailleurs à certaines modifications dans les traitements proposés. De plus, les intervalles présentent des caractéristiques spécifiques devant être prises en compte comme, par exemple, les contraintes dite «*disjonctives*» et l'«*énumération*» sur les intervalles.

8.3.1 Simplification du système d'enregistrement

Un système de résolution de CSP numériques utilisant la B-consistance d'arc comme mécanisme de filtrage ne tient compte que des valeurs extrêmes des domaines des variables concernées (la B-consistance d'arc ne s'intéresse qu'aux bornes des domaines des variables).

C'est pourquoi nous proposons de ne conserver que ces bornes pour spécifier le domaine d'une variable d'un CSP numérique. Ainsi, pour une variable x dont le domaine originel est l'intervalle $[a, b]$, nous noterons $D_x^{\text{orig}} = \{a, b\}$. a sera noté $m_o(x)$ et b, $M_o(x)$.

Ainsi, le domaine courant D_x d'une telle variable sera lui-aussi un couple de valeurs $\{m(x), M(x)\}$représentant les valeurs extrèmes de l'intervalle approximation du domaine effectif de la variable x.

8.3.2 Modifications induites

Cette simplification induit un certain nombre d'autres modifications pouvant être réalisées pour le système de maintien de déduction.

- Tout d'abord, l'algorithme de réduction ne s'intéresse qu'aux explications permettant d'obtenir les valeurs extrèmes du domaine courant d'une variable. On peut donc stocker l'explication permettant de déduire que d'une part le maximum du domaine d'une variable x est $M(x)$ et que le minimum est $m(x)$. Ce sont les seules explications intéressantes. On l'a, en effet, vu pour l'algorithme de réduction, mais, pour la fonction `explique-contradiction` cela suffit aussi (il y a contradiction lorsque $\exists x, m(x) > M(x)$). Cette information peut se calculer de manière incrémentale.

- On peut stocker les retraits successifs à gauche (sur le minimum) et à droite (sur le maximum) dans des piles. En effet, supposons qu'une relaxation de contrainte provoque l'invalidation d'un retrait de valeur concernant un des extrema d'une variable. Comme le domaine d'une variable est représenté par ces extrema, les retraits de valeurs subséquents sont, quoi qu'il arrive, remis en cause. Il faudra donc tout dépiler. Ainsi, on peut convenir d'expliquer en partie une modification d'extremum par l'explication de l'extremum courant.

Exemple 32 (Gestion simplifiée sur les intervalles) :
Soit une variable x dont le domaine originel est l'intervalle $[Min, Max]$. Supposons qu'après plusieurs modifications des bornes, on arrive à un retrait (noté r_1 sur la figure 8.1 – en blanc apparaissent les valeurs faisant partie du domaine) dont l'explication contient la contrainte c et plus tard à un autre retrait (r_2 sur la figure) ne dépendant pas de c. Supposons enfin que l'on doive relaxer la contrainte c.
Le retrait de la contrainte c conduit à l'invalidation de la déduction r_1 et donc au retour des valeurs correspondantes. Lorsqu'on réalise ce retour de valeurs, un «*trou*» apparaît alors dans le domaine de la variable (*cf.* figure 8.1). Notre représentation des intervalles interdit une telle situation, il faut donc aussi invalider le retrait r_2.
Pour ne pas avoir à gérer la détermination des retraits à invalider mais ne dépendant pas de la contrainte relaxée (comme r_2), nous proposons d'ajouter à l'explication d'un retrait l'explication du retrait précédent réalisé sur la borne. Ainsi, dans notre exemple, si l'explication de r_1 est ajoutée à celle de r_2, la relaxation de c invalide le retrait r_2 ce qui nous donne le comportement désiré.

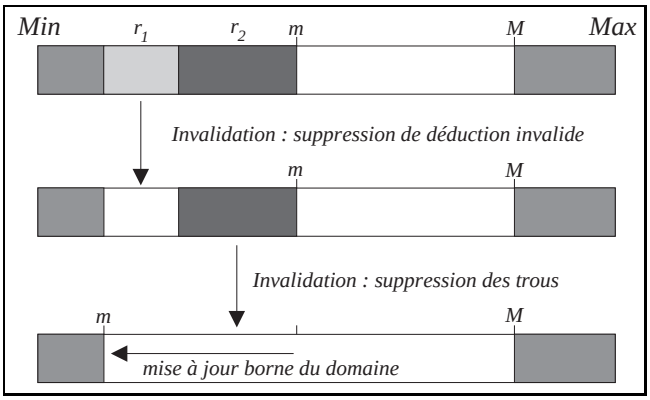

FIG. 8.1 – *Gestion des invalidations de déduction dans un intervalle*

8.3.3 Le cas des contraintes *disjonctives*

Définition 61 (Contrainte Disjonctive – [Lhomme, 1994])
Une contrainte est ***disjonctive*** *sur une partie de l'espace si la projection de l'ensemble des solutions sur une variable au moins n'est pas un intervalle.*

> **Exemple 33 (Contrainte disjonctive) :**
> La contrainte $x^2 = y$ n'est pas disjonctive sur $D_x = [-2, 2]$ et $D_y = [0, 4]$ car les projections sur x et y de l'ensemble des solutions sont des intervalles.
> Par contre, cette même contrainte est disjonctive sur $D_x = [-2, 2]$ et $D_y = [1, 4]$ car la projection sur x de l'ensemble des solutions est $[-2, -1] \cup [1, 2]$.

On se rend compte que lorsqu'il y a des contraintes disjonctives dans le système de contrainte des *trous* peuvent apparaître dans les domaines. Il y a trois façons de *traiter* ces *trous*.

1. Utiliser toute l'information numérique disponible mais en ne considérant que des intervalles. Ainsi, un problème comportant une contrainte disjonctive sera transformé en une disjonction de sous-problèmes pour lesquels la contrainte est non-disjonctive. On décompose alors les domaines de telle sorte que la contrainte soit non disjonctive pour chacun des éléments de cette décomposition.

   On peut donc poser des points de choix pour chacun des éléments de cette décomposition. Cette approche introduit une très grande combinatoire dans le problème.

2. On peut aussi utiliser toute l'information numérique disponible et traiter les unions d'intervalles disjoints en tant que telles.

   Malheureusement, plusieurs problèmes peuvent se présenter : d'une part, une fonction appliquée à une telle union peut donner une union d'intervalles quelconques qu'il faut alors *simplifier* pour obtenir de nouveau une union d'intervalles disjoints, et d'autre part, ces unions d'intervalles peuvent conduire à une combinatoire élevée puisqu'il faut examiner la situation pour chacun des intervalles.

3. On peut enfin appliquer le principe de la B-consistance : on se contente de travailler sur les valeurs extrèmes. Il n'y a donc pas de points de choix ni de décomposition : seulement, une partie de l'information est perdue. Mais, comme les contraintes sont conservées, lorsque des informations plus précises

seront demandées, celles-ci pourront alors se réveiller et *nettoyer* convenablement les domaines des variables concernées.

Dans INTERLOG [Lhomme, 1994] l'approche utilisée par défaut est la troisième approche. La première est proposée à la demande et est mise en œuvre à l'aide de points de choix PROLOG. Quant à la deuxième, elle n'est pas envisagée.

L'utilisation de notre cadre général pour la relaxation de contraintes spécialisé pour la réduction de domaine envisage implicitement la troisième approche. Pour modéliser la première approche, on peut utiliser une contrainte `one-of` sur le découpage en intervalles disjoints. En effet, si un choix conduit à une contradiction, une part non négligeable du travail réalisé entre temps pourra être conservée, c'est un des avantages de notre approche qui évite autant que faire se peut le *thrashing*.

8.3.4 Énumération pour les intervalles

Des techniques d'énumération sur les intervalles sont habituellement envisagées pour préciser les réponses apportées à un problème donné. L'approche retenue en général passe par une exploration dichotomique du domaine des variables considérées jusqu'à ce qu'une précision fixée au préalable soit atteinte pour le résultat.

Nous proposons de modéliser cette recherche dichotomique par une contrainte de type `one-of`. Il ne s'agit pas véritablement d'une contrainte `one-of` au sens présenté jusqu'à présent.

En effet, lors d'une recherche dichotomique sur le domaine d'une variable x de domaine D_x, on commence par découper ce domaine en deux : les sous-domaines complémentaires D_x^1 et D_x^2. Ce découpage peut être modélisé par une contrainte `one-of` usuelle. Par contre, lorsqu'ultérieurement on redécoupe le domaine courant (par exemple D_x^1) en deux autres sous-domaines $D_x^{1,1}$ et $D_x^{1,2}$ le fait que la variable x est dans D_x^1 est toujours vrai et donc ces nouvelles contraintes (les nouveaux sous-domaines) ne peuvent provoquer le retrait de la contrainte concernant D_x^1. On ne peut donc gérer ce découpage au même niveau, il faut donc créer un nouveau `one-of`.

De plus, il faut non seulement conserver les deux niveaux de découpage mais aussi conserver leurs liens de dépendance. En effet, on peut imaginer qu'une contradiction qui dépend du premier découpage puisse être découverte après le deuxième découpage. Nous proposons donc de donner un nouveau type de contrainte `one-of` : une contrainte `one-of-dependant` relaxable. Ce `one-of-dependant` défini pour un niveau de découpage $k+1$ dépend d'une contrainte élément d'un `one-of-dependant` défini au niveau k. Ainsi, lorsqu'une contrainte de niveau k est relaxée, toutes[2] les contraintes de niveau $k+1$ dépendant de celle-ci doivent aussi être relaxées. Un `one-of-dependant` au niveau 0 est alors un `one-of` classique.

Pour une contrainte `one-of-dependant`, lorsqu'une contradiction est levée (plus de possibilité d'activer une contrainte du `one-of`), le traitement habituel d'une contradiction est déclenché. En effet, si la responsabilité de cette contradiction incombe à la contrainte de niveau supérieur ayant généré ce `one-of-dependant`, l'explication de contradiction en tiendra nécessairement compte. Ainsi, grâce à la simple particularité de la relaxation systématique des contraintes induites par un découpage dichotomique, on peut utiliser le mécanisme habituel de traitement des contradictions.

2. En fait, il y en a au plus une active à tout moment.

8.4 Complexité

La complexité de l'approche instanciée sur les intervalles est la même que l'approche sur les CSP, car la réorganisation du système de maintien de déduction n'influe pas sur son occupation mémoire, ni sur sa complexité temporelle. À ceci près que les opérations de fournitures d'explication sont facilitées par la limitation des consultations aux bornes des domaines des variables considérées. Ainsi, la gestion des ajouts et retraits de contraintes avec un DMS ne modifient pas la complexité originelle du solveur utilisé.

8.5 Conclusion

L'utilisation des résultats présentés dans les chapitres précédents nous a permis de proposer dans ce chapitre un cadre dynamique sur les intervalles qui, à notre connaissance, n'existait pas encore. Ainsi, on peut penser qu'il est maintenant possible de donner une version incrémentale efficace des algorithmes utilisés dans le système *Indigo* [Borning *et al.*, 1996] (*cf.* section 3.4.2 page 34). De plus, les propositions que nous faisons pour la gestion de l'énumération, dont l'implémentation est en cours, devraient apporter des résultats intéressants.

Chapitre 9

Implémentation pour les CSP

Sommaire

DANS CE CHAPITRE, nous présentons les grandes lignes d'une l'implémentation du schéma relax(FD,$\mathcal{C}$,$\mathcal{P}_{ac}$) tel que nous l'avons présenté au chapitre 7.

Nous nous situons dans les conditions[1] d'utilisation **C1** (ajout d'explication valide) et **C2** (déduction associée unique) nous permettant d'obtenir une complexité raisonnable pour l'approche (*cf.* section 7.4 page 92).

Nous regroupons toutes les implémentations de notre système de relaxation de contraintes sous le nom générique de système DECORUM (Deduction-based Constraint Relaxation Management) [Jussien et Boizumault, 1996c]. À ce jour, il existe quatre implémentations de notre système, trois maquettes (en MIT SCHEME, en CLAIRE – langage développé à l'École Normale Supérieure [DMI, 1996] – et en STK – un SCHEME *graphique* [Gallesio, 1996]) et un prototype (en C++, porté en collaboration avec David Fraszko pendant son stage de DEA). Ces implémentations sont adaptées aussi bien au cadre relax(FD,$\mathcal{C}$,$\mathcal{P}_{ac}$) qu'au cadre relax(Interval, $\mathcal{C}$, $\mathcal{P}_{bc}$) comme on l'a vu dans le chapitre précédent.

9.1 Architecture générale

Nous présentons ici l'architecture générale d'un solveur interactif utilisant une méthode de recherche de C-solution lors de l'obtention d'un système de contraintes $\mathcal{P}$-contradictoire.

9.1.1 Schéma de fonctionnement

La figure 9.1 présente le schéma général d'organisation d'un tel solveur. Les diverses implémentations de notre méthode de recherche utilisent :

- trois modules principaux (représentés par des rectangles aux coins arrondis) ;
- deux files de propagations (représentées par des rectangles) ;
- trois espaces de stockage (représentés par des ovales).

La figure 9.1 représente les flux de données entre les différentes entités du système.

Les trois espaces de stockage concernent :

- les **contraintes** du problème, permettant ainsi de connaître leur état courant (active ou relaxée). Cet espace de stockage contient aussi les explications de relaxation des contraintes concernées ;
- les **explications** pour les réductions déterminées par le solveur et maintenues par le DMS ;
- les **explications de contradiction** déterminées par le module de recherche de C-solution par la fonction `explique-contradiction` et utilisées par la fonction `meilleure-configuration`. Rappelons que sous les conditions d'utilisation **C1** et **C2**, seule la dernière explication de contradiction nécessite un stockage.

Les deux files de propagation sont :

- **La file des contraintes**
  Cette file contient les contraintes du problème. Il s'agit de contraintes en cours de prise en compte. Elles ont été intégrées au système et vont être propagées. Pour cela, on utilise l'algorithme d'ajout dynamique de contrainte (algorithme 4 du chapitre 6 décrit page 76). Le réveil d'une contrainte demandé par le module de recherche provoque son ajout dans à cette file.

1. Ces conditions sont présentées dans le chapitre 6 section 6.2.2 page 72.

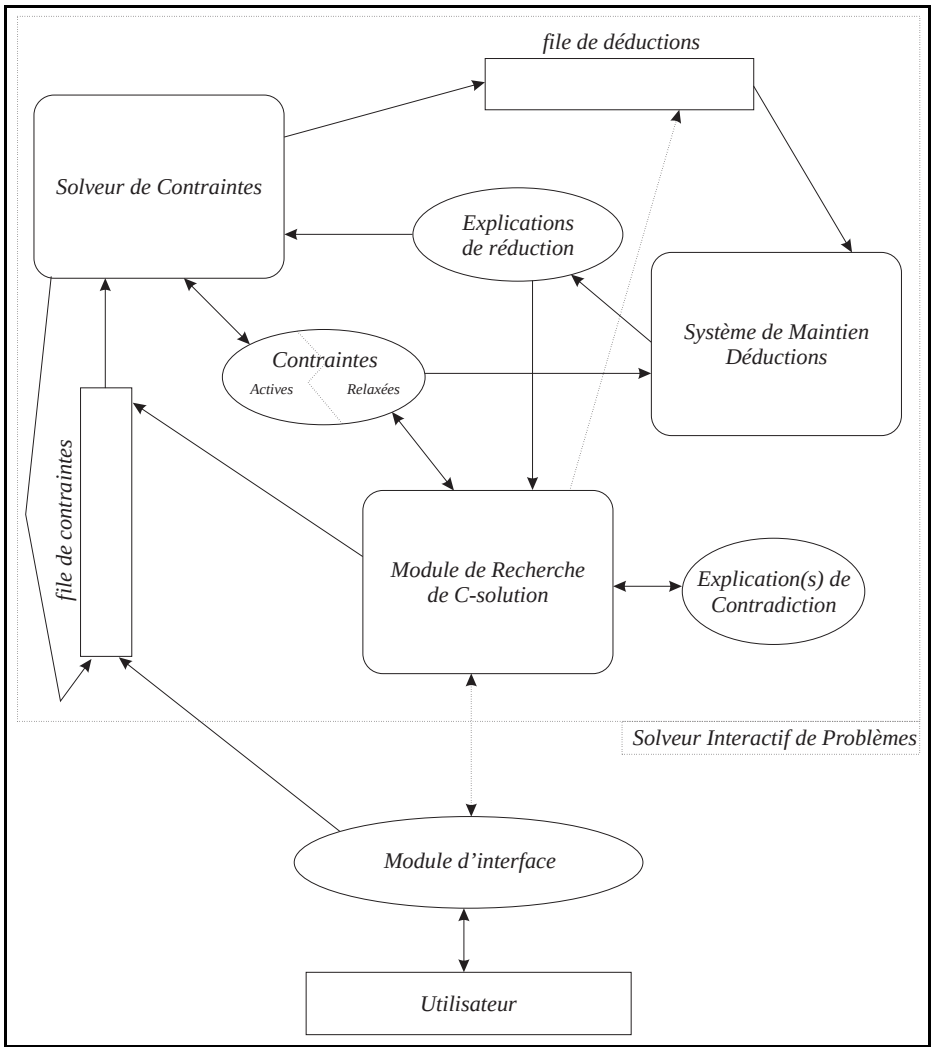

FIG. 9.1 – *Schéma de fonctionnement d'un solveur interactif de problèmes*

- **La file des déductions**
  Cette deuxième file contient les réductions à effectuer sur les domaines des variables. Les réductions à prendre en compte sont calculées par l'algorithme de résolution et *envoyées* dans cette file. Elles seront traitées par le système de Maintien de Déductions (algorithme 5 du chapitre 6 décrit page 76).

Notre système possède trois modules principaux :

- **Module de recherche de C-solution**
  Ce module, lors de l'identification d'une contradiction par le DMS, est chargé de :

  - Calculer une nouvelle explication de contradiction (fonction `explique--contradiction`) stockée dans le module. Il faut pour cela demander, pour la variable dont le domaine est vide, l'état courant des explications au DMS qui consulte l'espace de stockage des explications.
  - Calculer la prochaine configuration prometteuse à explorer à l'aide de la fonction `meilleure-configuration`. On utilise pour cela les explications de contradiction. Il serait ici possible de faire appel à l'utilisateur pour arbitrer sur un ensemble de contraintes (lien en pointillés entre le module de Recherche et un module d'interface sur la figure 9.1).
  - Effectuer le changement de configuration (utilisation de la fonction `maj--configuration`). Il s'agit d'invalider les déductions du DMS touchées par la relaxation d'une contrainte et d'intégrer de manière préemptive (en tête) des nouvelles contraintes dans le cas du traitement d'une contrainte dans la file des contraintes ou de provoquer le réveil d'autres contraintes dans le cas d'un retour de contrainte. On modifiera l'état courant des contraintes. Ce module se charge aussi de nettoyer la file des actions d'éventuelles déductions en attente devenues invalides (lien en pointillés entre le module de Recherche et la file de déductions sur la figure 9.1).

- **Solveur de Contraintes**
  Ce module, ayant accès à l'intégralité du système de contraintes, est chargé de mettre en œuvre l'algorithme de réduction (algorithme 4). Ce module doit consulter l'espace de stockage des explications de réduction pour pouvoir connaître l'état courant des domaines pour réaliser le calcul de point fixe. Ce module a accès à la file des contraintes en attente pour mettre en œuvre leur traitement.

- **Système de maintien de Déductions**
  Le DMS est chargé de maintenir l'espace de stockage des explications. Il doit de plus prendre en compte les modifications d'état des contraintes. Enfin, il est chargé de gérer les nouvelles réductions stockées dans la file des déductions.

On voit apparaître sur la figure 9.1, l'utilisateur du système qui peut, à tout moment, par l'intermédiaire d'un module d'interface :

- ajouter une contrainte à la file des contraintes ;
- retirer une contrainte ;
- intervenir dans le choix de la meilleure configuration suivante lorsque ce même module ne peut se décider seul. Nous considérons ici que le comparateur implicite utilisé par l'utilisateur est *contradiction-local*.

9.1.2 Composants

L'architecture présentée dans la section précédente suppose l'existence de composants de base ayant un comportement minimal que nous présentons ici.

La figure 9.2 reprend les informations fournies dans la suite. On pourra s'y reporter pour avoir une vue d'ensemble des «*objets*» du système. Une flèche représente sur cette figure un lien de dépendance [2] alors qu'une simple arête représente un lien d'utilisation.

Tout d'abord nous avons les composants normaux d'un système de résolution de contraintes :

- **Variables**
  Les variables sont les constituants basiques d'un système de résolution de contraintes. Une variable doit avoir accès :
  - aux contraintes la concernant ;
  - à la contrainte `one-of` modélisant l'énumération de cette variable ;
  - à son domaine.
- **Contraintes**
  Les contraintes sont bien évidemment très importantes. Elles doivent savoir :
  - se propager à l'initialisation (*i.e.* fonction `prop`) ;
  - se propager en cas de modification (*i.e.* fonction `localprop`).

  Une contrainte doit avoir accès à un certain nombre d'informations :
  - la préférence qui lui est associée (ou un indicateur de demande à l'utilisateur) ;
  - l'explication qui lui est associée en cas de relaxation ;
  - son `one-of` d'origine s'il existe ;
  - la liste des retraits de valeurs dont elle assure la validité.

  On distingue deux types de contraintes : les méta-contraintes qui, comme leur nom l'indique, sont des contraintes sur les contraintes (c'est le cas de la contrainte `one-of`) qui ne sont pas relaxables [3] et les contraintes *basiques* qui elles sont véritablement les contraintes du problème et sont relaxables.
- **Domaines**
  Les domaines représentent le troisième élément constitutif fondamental d'un CSP. C'est le point de liaison entre CSP et système de maintien de déductions. Un domaine doit connaître :
  - l'ensemble initial de ses valeurs ;
  - l'ensemble courant de ses valeurs ;
  - la liste des suppressions de valeurs.

On peut maintenant présenter les éléments permettant de prendre en compte le système de maintien de déduction :

- **Retraits de valeurs**
  C'est une information concernant une variable et un ensemble de valeurs dans le domaine de cette variable. À une suppression est associée une unique explication fournie par le solveur. Une suppression doit être capable de calculer sa validité.

2. Le concept de départ dépend du concept d'arrivée.
3. Sauf dans le cas des contraintes `one-of-dependant` pour les CSP numériques.

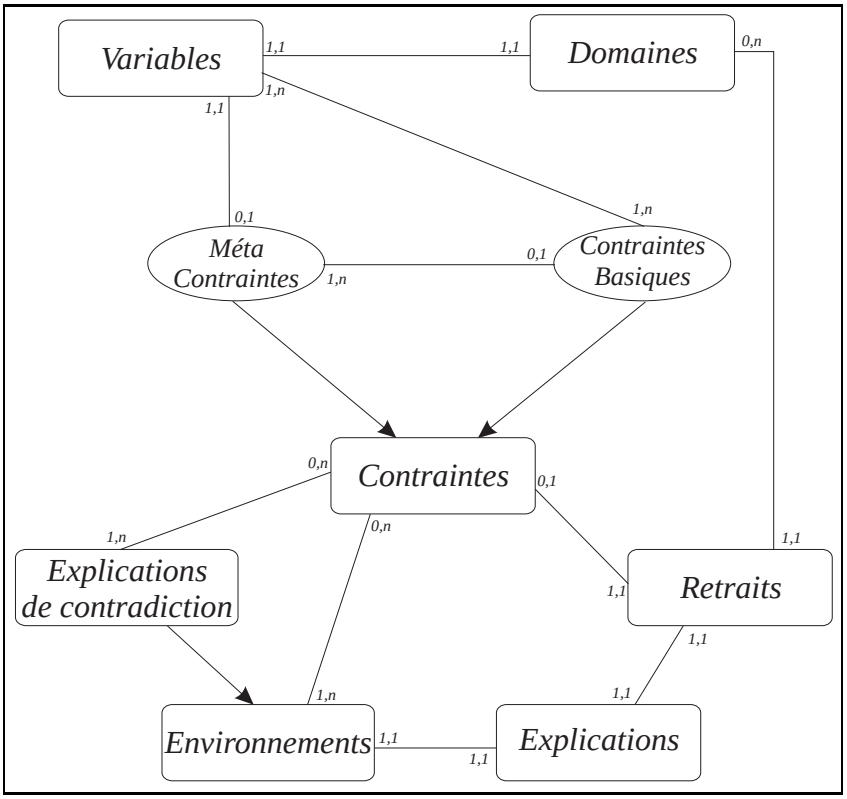

FIG. 9.2 – *Liens entre les concepts nécessaires à notre approche*

- **Explications**
  Une explication est composée de trois informations: la partie contraintes, la partie déductions (*i.e.* d'autres retraits de valeur) et l'environnement associé. Une explication doit être capable de calculer sa validité.

- **Environnements**
  Un environnement est un ensemble de contraintes. Il peut être valide ou invalide. Par un simple test d'inclusion, un environnement est capable de tester sa validité.

- **Explication de contradiction**
  Une explication de contradiction est en fait un environnement (*cf.* définition 23 page 44).

9.2 Implémentation en CLAIRE

Nous présentons ici brièvement une implémentation réalisée avec le langage CLAIRE [DMI, 1996]. Ce langage créé par Yves Caseau à l'École Normale Supérieure a été utilisé pour traiter de nombreux problèmes combinatoires avec des techniques de type CSP (problèmes d'ordonnancement, problèmes de placement, ... *cf.* [Caseau et Laburthe, 1996]).

9.2.1 Le langage et ses spécificités

CLAIRE (*Combining Logical Assertions, Inheritance, Relations and Entities*) est un langage fonctionnel à objets utilisant un mécanisme de règles de propagation. Il a été conçu pour permettre d'exprimer des algorithmes complexes en peu de lignes et d'une manière lisible et élégante.

Un des avantages principaux de CLAIRE est la possibilité de définir des règles de propagation qui sont caractérisées par une entité dont toute modification, si elle satisfait certaines conditions, provoque l'exécution du corps de la règle. Cela permet de déséquentialiser les traitements. C'est le principal apport de CLAIRE que nous avons utilisé.

CLAIRE fournit un mécanisme d'allocation dynamique de mémoire et utilise un *ramasse-miettes*. Le compilateur CLAIRE génère du code C++ mais il existe aussi un interprète.

L'implémentation en CLAIRE obéit bien sûr aux grand principes évoqués précédemment dans ce chapitre. Les spécificités du langage ont été utilisées principalement pour faciliter l'expression des traitements.

9.2.2 Les variables

Une variable d'un problème est un objet CLAIRE décrit figure 9.3.

Une variable est en effet une *entité* portant un `nom`, pour laquelle on connaît le `min`, le `max`, le `min` initial, le `max` initial, la taille et l'ensemble définissant le domaine de la variable. On reconnaît là le modèle d'implémentation proposé par Diaz [1995] pour `clp(fd)`.

De plus, on a accès aux retraits concernant cette variable (`Ddel`) et la réunion d'explications permettant d'expliquer le minimum courant (`Dwhy-min`), le maximum courant (`Dwhy-max`) et l'état courant du domaine (`Dwhy`).

D'autres part, pour la propagation, on a accès à la liste des contraintes devant être réveillées en cas de modification du minimum (`Dcte-min`), du maximum (`Dcte-max`) et pour toute modification (`Dcte`).

```
[ Dvars <: thing( Dnom:string, Dmin:integer, Dmax:integer,
                  Dmin-init:integer, Dmax-init:integer,
                  Dsize: integer, Dset: set[integer],

                  Ddel: list[Retraits],
                  Dwhy-min:set[Contraintes],
                  Dwhy-max:set[Contraintes],
                  Dwhy:set[Contraintes],

                  Dcte-min:set[Contraintes],
                  Dcte-max:set[Contraintes],
                  Dcte:set[Contraintes],

                  Dcte-enum:Contraintes_Enumeration )]
```

FIG. 9.3 – *Code* CLAIRE *définissant une variable*

Enfin, on a accès à la contrainte `one-of` modélisant l'énumération sur cette variable (`Dcte-enum`).

9.2.3 Les contraintes

La version de base de l'implémentation en CLAIRE permet de traiter les contraintes unaires, binaires et ternaires bijectives, anti-bijectives et monotones. Le traitement de l'énumération est bien entendu pris en compte.

Contraintes de bases

À titre d'exemple, la figure 9.4 présente le code CLAIRE se chargeant de la propagation d'une contrainte de type $X > c$. Lorsqu'on propage une telle contrainte, il faut retirer du domaine de la variable X (`Ct.X`) les valeurs entre la borne inférieure courante de X (`Ct.X.Dmin`) et c (`Ct.C`). L'explication est alors un singleton : `set(Ct)`.

```
[ propage_contrainte(Ct:Contraintes_X>c): void ->
        if (Ct.Cetat = CT-IN)    ; La contrainte est active
           remove_values(Ct.X,Ct.X.Dmin, Ct.C,set(Ct)) ]
```

FIG. 9.4 – *Code* CLAIRE *propageant une contrainte* $X > c$.

Extensions

Nous avons ajouté le traitement d'autres contraintes :

- Ainsi, le traitement d'une contrainte de type `all-different` en utilisant les travaux de [Régin, 1994] repris dans sa thèse [Régin, 1995] demande environ 250 lignes de code CLAIRE.
- Le traitement des contraintes disjonctives dans les problèmes d'ordonnancement en s'inspirant des travaux de [Carlier et Pinson, 1990; Carlier et Pinson, 1994] sur le problème de Job-Shop demande lui environ 100 lignes de code CLAIRE.

On voit ainsi la puissance d'expression du langage CLAIRE. Pour l'illustrer nous donnons le code CLAIRE (figure 9.5) permettant de vérifier une des conditions[4] des ajustements proposés par [Carlier et Pinson, 1994] :

$$\min_{i\ inJ} r_i + \sum_{i \in J} p_i + \min_{i \in J} q_i + p_c > \mathrm{UB}$$

```
[ condition(c:Taches, J:set[Taches], UB:integer): boolean ->
      min({ri(t) | t in J}) +
      (let s := 0 in (for t in J s :+ pi(t), s)) +
      min({qi(t) | t in J}) +
      pi(c)
             > UB ]
```

FIG. 9.5 – *Code* CLAIRE *vérifiant une condition ensembliste*

On voit donc que l'intégration de nouvelles contraintes à notre implémentation en CLAIRE peut être réalisée très simplement et très efficacement.

9.2.4 Adaptation de l'approche

La propagation des contraintes et le traitement d'une contradiction se font grâce aux règles de propagation permettant de rendre totalement indépendants les différents modules de notre architecture générale.

En effet, dès qu'une modification est réalisée sur le domaine d'une variable, le déclenchement de la propagation des contraintes concernées se fait par une règle de production CLAIRE.

En CLAIRE, ce fonctionnement peut être garanti en utilisant le code de la figure 9.6. L'instruction `event` permet de signaler au système CLAIRE que toute modification sur les bornes (`Dmin` et `Dmax`) d'une variable doit être signalée. Ensuite, la règle `propagation_modif_min` est chargée de propager la nouvelle valeur `NMin` de la borne inférieure du domaine de la variable `V` sur toute les contraintes devant être propagées pour une modification de cette borne (ensemble `V.Dcte-min`).

```
event(Dmin,Dmax)
[ propagation_modif_min(V:Dvars) ::
     exists(NMin, V.Dmin = NMin)
     => for c in V.Dcte-min propage_contrainte_min(c,V,NMin) ]
[ propagation_modif_max(V:Dvars) ::
     exists(NMax, V.Dmax = NMax)
     => for c in V.Dcte-max propage_contrainte_max(c,V,NMax) ]
```

FIG. 9.6 – *Code* CLAIRE *assurant la propagation des contraintes*

De même, lors d'une telle modification de domaine, dès que celui-ci devient vide, les conditions de déclenchement du module de recherche de C-solution sont réunies et ainsi la recherche de C-solution est provoquée. La figure 9.7 présente le code CLAIRE correspondant à cette règle.

4. Nous ne donnons pas plus d'explications, le but est simplement de montrer l'avantage du langage.

```
event(Dsize)
[ domaine_vide(V:Dvars) ::
     (V.Dsize = 0)
     =>  mode(15), handle_contradiction(V) ]
```

FIG. 9.7 – *Code* CLAIRE *identifiant une contradiction*

L'instruction `event` demande au système CLAIRE de signaler toute modification sur la taille du domaine d'une variable. La règle `domaine_vide` se déclenche si le domaine de la variable `V` est devenu vide (`Dsize = 0`) et déclenche alors le traitement d'une contradiction. L'instruction `mode` permet de donner une priorité plus élevée à cette règle. Ainsi, si plusieurs règles peuvent être déclenchées, le système choisira celle de plus grande priorité en premier.

La figure 9.8 présente le code CLAIRE correspondant au traitement d'une contradiction. Cette fonction récupère l'explication de la contradiction (la réunion des explications des retraits sur la variable concernée : `V.Dwhy`), puis choisit une contrainte à relaxer et la relaxe en donnant comme explication de retrait : l'explication de contradiction (sans – opérateur `but` – la contrainte concernée).

```
[ handle_contradiction(V:Dvars): void ->
   let Explication := V.Dwhy in
       ( if valide?(Explication)
            let ToRelax := choose_relax_cte(Explication) in
                retrait(ToRelax,Explication but ToRelax)) ]
```

FIG. 9.8 – *Code* CLAIRE *de traitement d'une contradiction*

L'implémentation réalisée représente environ 800 lignes de code CLAIRE. Ce langage s'est donc avéré être un outil très adapté au prototypage de solveurs.

9.3 Conclusion

Nous avons présenté dans ce chapitre les grandes lignes d'une implémentation de notre approche sur les CSP. Pour des raisons d'efficacité et pour pouvoir contrôler plus finement tous les mécanismes mis en jeu, nous avons préféré réaliser un prototype de DECORUM en C++ plutôt que d'utiliser le compilateur de CLAIRE. Ce portage a été réalisé avec l'aide d'un stagiaire de DEA [Fraszko, 1997].

La version C++ de DECORUM apparaît comme un ensemble de composants logiciels pouvant être utilisés pour programmer un système interactif utilisant la résolution de contraintes. [Fraszko, 1997] décrit en détail l'implémentation réalisée.

Troisième partie

Expérimentations et Premières Applications

Chapitre 10

Expérimentations

Sommaire

DANS CE CHAPITRE, nous étudions de manière expérimentale le comportement du système DECORUM. Pour cela, nous l'avons soumis à différents ensembles de tests.

Nous reportons ici les résultats obtenus dans les quatre situations suivantes :

- **Mesure de la pertinence de l'approche.** Nous avons cherché à comparer notre approche (recherche d'une C-solution dans le cas d'un problème surcontraint) sur des problèmes pour lesquelles l'énumération n'intervient que lorsque toutes les contraintes du problème sont intégrées. Ce procédé nous permet de nous comparer aux résultats obtenus avec des systèmes usuels (qui s'arrêtent s'il n'y a pas de solution). En particulier, nous montrons que DECORUM respecte bien la transition de phase (section 10.2).

- **Problèmes dynamiques.** Il n'existe pas de problèmes de référence pour les CSP dynamiques. Nous avons donc opté pour une étude de DECORUM en fonction du moment choisi pour l'énumération (depuis «*avant que les contraintes ne soient introduites*» – approche maintien de *solution*, jusqu'à «*après intégration de toutes les contraintes*» – approche maintien d'*arc-consistance*). Ceci permet de faire en quelque sorte varier la propriété maintenue au cours de la résolution. Nous montrons ici que la différence n'est pas significative (section 10.3).

- **Problèmes de grande taille.** Nous montrons que DECORUM peut être utilisé sur des problèmes de grande taille (section 10.4).

- **Comparaison avec l'algorithme MAC.** Nous comparons DECORUM en version «*énumération*» (arrêt dès que le problème est identifié comme étant surcontraint) avec l'algorithme MAC (Maintaining Arc-Consistency – [Sabin et Freuder, 1994]) considéré comme un bon algorithme de résolution de CSP (section 10.5). Nous étudions ainsi la pertinence de l'utilisation d'un DMS dans le cadre de notre approche.

10.1 Génération de CSP aléatoires

Les CSP aléatoires sont habituellement générés en fonction de 4 paramètres : le nombre n de variables, la taille uniforme d des domaines de ces variables, la densité[1] p_1 du problème et la dureté[2] p_2 des contraintes du problèmes. Nous utilisons le générateur *vraiment* aléatoire proposé par Bessière, Dechter, Frost et Régin (*cf.* `http://www.ics.uci.edu/~dfrost/csp/generator.html` et [Bessière et Régin, 1996]).

L'approche habituelle adoptée sur ces instances aléatoires est de trouver une solution vérifiant toutes les contraintes ou de prouver qu'il n'en existe pas de la manière la moins coûteuse possible. Ce *coût* est souvent mesuré en nombre de tests de consistance réalisés sur des paires de variables [Prosser, 1994a]. Les résultats obtenus sur l'étude de CSP binaires aléatoires [Smith, 1994; Smith et Grant, 1995; Prosser, 1994b] montrent qu'en maintenant n, d et p_1 constants, il existe une petite région de valeurs pour p_2[3] où le coût moyen de résolution connaît un pic d'activité, pendant qu'au même moment le nombre de problèmes solubles[4] devient proche de zéro. C'est ce qu'on appelle la transition de phase [Prosser, 1994a].

1. $p_1 = \frac{e}{n(n-1)/2}$ où e est le nombre de contraintes du problème.

2. $p_2 = \frac{ct}{d^2}$ où ct représente le nombre de couples autorisés dans une contrainte binaire exprimée en extension.

3. Plus exactement, autour de $p_2 = 1 - d^{-2/(n-1)p_1}$. C'est ce qu'on appelle la valeur critique de p_2.

4. Problèmes pour lesquels il existe une affectation vérifiant toutes les contraintes.

10.2 Conservation de la transition de phase

Nous montrons ici qu'en utilisant le système DECORUM (qui relaxe des contraintes en cas de problème sur-contraint) la transition de phase est conservée au même endroit alors que l'on continue le calcul après l'identification d'un système de contraintes contradictoire. De plus, nous montrons que l'utilisation de DECORUM n'entraîne pratiquement pas de surcoût lorsqu'au lieu de signaler qu'un problème *très* sur-contraint[5] n'a pas de solution, une solution *relaxée* est calculée.

10.2.1 Protocole expérimental

Nous avons généré 100 instances de problèmes $\langle n, d, p_1, p_2 \rangle$. Nous avons utilisé le comparateur C_{MM} et considéré que toutes les contraintes[6] étaient d'importance équivalente.

Pour chaque série de problèmes, nous avons recueilli les informations suivantes :

- le nombre moyen d'affectations de valeur à une variable testées et remises en cause : on peut parler en quelque sorte du nombre de *backtracks*[7] ;
- le nombre moyen de tests de consistance réalisés[8] ;
- l'espace moyen occupé pour stocker les informations du DMS. L'unité choisie ici est la contrainte. En effet, les seules informations enregistrées sont des explications, qui ne contiennent que des contraintes ;
- le temps `cpu` moyen passé pour la résolution.

Ces informations sont recueillies à deux instants précis : lorsque la première contrainte du problème[9] est relaxée *i.e.* lorsqu'on vient de montrer que le problème était $\mathcal{P}$-contradictoire et à la fin de la résolution *i.e.* lorsqu'une configuration $\mathcal{P}$-satisfaisable a été déterminée.

10.2.2 Premiers résultats

Les premiers résultats présentés concernent des problèmes $\langle 10, 10, 0.5, p_2 \rangle$ avec p_2 variant de 0 à 1 par pas de 0.01. Pour ces problèmes, la valeur critique de p_2 est 0.63. Dans nos tests, les premiers problèmes $\mathcal{P}$-contradictoires apparaissent pour $p_2 = 0.57$.

Les figures 10.1 à 10.4 représentent l'évolution, en fonction de la valeur de p_2, du nombre moyen, respectivement, de tests de consistance, espace occupé, backtracks, temps `cpu` en ms, nécessaires pour résoudre les problèmes étudiés. La courbe en gras représente les résultats pour obtenir une C_{MM}-solution et l'autre les résultats obtenus pour trouver une solution ou montrer que le problème est sur-contraint.

En ce qui concerne le nombre de tests de consistance (*cf.* Fig. 10.1) et le nombre de backtracks nécessaires (*cf.* Fig. 10.3), on constate très nettement un pic d'activité

5. Un problème très sur-contraint est un problème dont il est facile (rapide) de prouver l'insatisfaisabilité.

6. Nous ne tenons pas compte des contraintes d'énumération qui conservent leur poids négligeable par rapport aux contraintes vis à vis du comparateur utilisé.

7. La notion de *backtrack* utilisée ici est celle de la communauté programmation logique *i.e.* nous comptons *un* backtrack pour chaque désinstanciation ou changement de valeur pour une variable.

8. L'algorithme d'arc-consistance utilisé est l'algorithme AC4. On appelle test de consistance, le test d'appartenance d'un couple de valeur à la liste des couples possibles définissant une contrainte. Dans le cas d'utilisation d'AC4, nous comptons aussi chaque incrémentation et décrémentation des compteurs utilisés dans l'algorithme car ce sont les opérations de base.

9. On ne tient donc pas compte des contraintes liées à l'énumération.

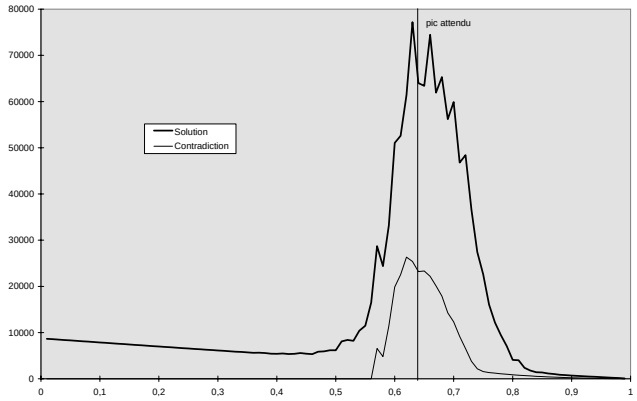

FIG. 10.1 – *Évolution du nombre de tests de consistance pour des problèmes* $\langle 10, 10, 0.5, p_2 \rangle$.

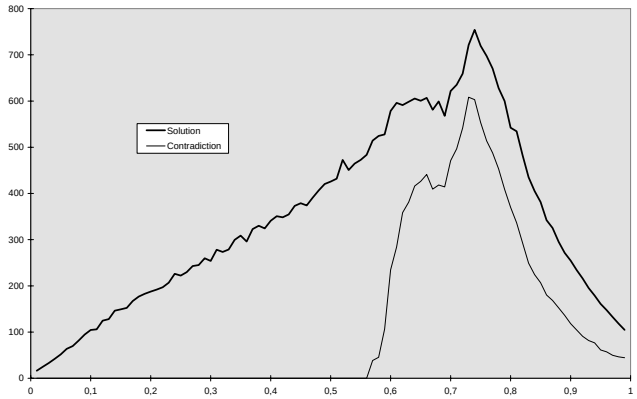

FIG. 10.2 – *Évolution de l'espace occupé pour des problèmes* $\langle 10, 10, 0.5, p_2 \rangle$.

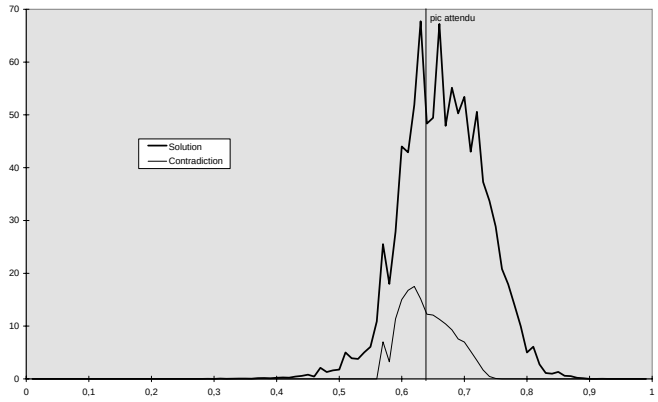

FIG. 10.3 – *Évolution du nombre de* backtracks *nécessaires pour des problèmes* $\langle 10, 10, 0.5, p_2 \rangle$.

autour de la valeur attendue aussi bien pour les valeurs correspondant à l'identification de la première *vraie* contradiction que pour l'obtention d'une solution. Il est intéressant de constater la persistance du phénomène de transition de phase malgré le fait que l'on calcule une configuration $\mathcal{P}$-satisfaisable au lieu de se contenter de signaler la non existence de solution. Ces résultats montrent que DECORUM présente un comportement comparables aux systèmes usuels sur les CSP aléatoires.

Notons que même au niveau de la consommation `cpu`[10], on retrouve la transition de phase (*cf.* Fig. 10.4). Apparemment, le système DECORUM ne présente donc pas un «overhead» démesuré lorsqu'il s'agit de relaxer des contraintes.

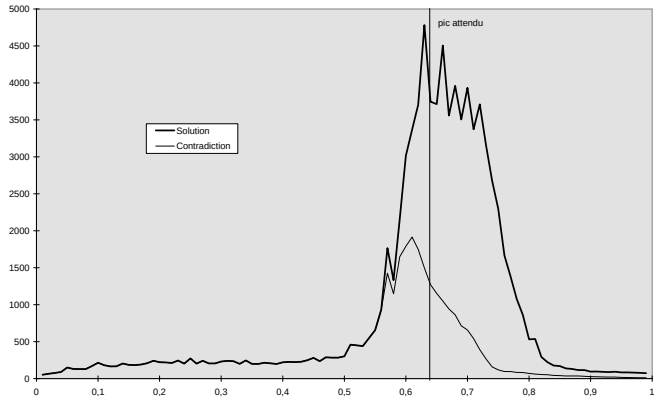

FIG. 10.4 – *Évolution du temps* `cpu` *consommé pour des problèmes* $\langle 10, 10, 0.5, p_2 \rangle$.

Enfin, notre utilisation raisonnée de l'enregistrement d'information dans le DMS conduit à un comportement spatial parfaitement maîtrisé de DECORUM. Ainsi, sur

10. Mesurée en ms sur une Sun Sparc 20.

la figure 10.2, l'espace occupé obéit aussi au phénomène de transition de phase.

Les résultats présentés ici nous ont permis de confirmer l'idée intuitive qu'en utilisant un système de relaxation de contraintes :

- les problèmes peu contraints (faibles valeurs de p_2) restent aisément résolus ;
- les problèmes très sur-contraints (grandes valeurs de p_2) sont eux aussi aisément résolus ;
- les problèmes *tangents* (valeurs de p_2 autour de la valeur critique) sont quant à eux aussi difficiles à résoudre qu'avec une approche classique.

Il faut noter que la facilité de résolution des problèmes très contraints est en grande partie due à la réutilisation que permet notre approche. Ainsi, une méthode demandant une réexécution complète en cas de relaxation (comme le fait le système IHCS– *cf.* section 3.4.1) n'obtiendrait vraisemblablement pas de tels résultats. En effet, pour trouver une solution l'approche utilisée aussi bien par DECORUM que par IHCS conduit à relaxer à chaque étape une unique contrainte pour voir si cela suffit à trouver une solution. Si la méthode employée ne profite pas du passé pour effectuer les calculs, la résolution de problèmes très sur-contraints (pour lesquels une bonne partie des contraintes doit être supprimée) conduit très vite à une *explosion* du temps de calcul et de l'espace nécessaire.

Il est rassurant de constater que DECORUM respecte le phénomène de transition de phase. Ainsi, notre système de relaxation de contraintes présente le même comportement global qu'un système classique alors qu'il offre en plus la relaxation de contraintes.

10.3 Étude sur des problèmes dynamiques

Il n'existe pas de problèmes de référence pour les CSP dynamiques. Nous proposons donc d'étudier le comportement de DECORUM dans différents degrés de dynamicité. Pour cela, nous mesurons l'influence (en espace occupé et en nombre de tests de consistance) de la *position* de l'énumération dans le système de contraintes : depuis l'énumération avant tout ajout de contrainte (en quelque sorte maintien de *solution*) jusqu'à l'énumération après tous les ajouts de contraintes (en quelque sorte maintien d'*arc-consistance*).

L'approche que nous avons utilisée jusqu'à présent dans nos expérimentations ne déclenche l'énumération que lorsque toutes les contraintes du problèmes ont été introduites, il s'agit donc d'un maintien d'arc-consistance. Dans une approche réactive, le but est plus de maintenir une solution. Ainsi, dans cette approche, une instanciation cohérente est maintenue au fur et à mesure de l'ajout de nouvelles contraintes. Une telle approche peut être beaucoup plus coûteuse. En effet, lorsqu'une contrainte est ajoutée, il y a peu de chances que la solution courante la vérifie, il faut alors modifier (par relaxation) certaines valeurs attribuées aux variables ; tandis que l'ajout d'une contrainte dans le cadre du maintien d'arc-consistance ne conduit qu'à des réductions supplémentaires dans les domaines des variables.

Nous introduisons donc un nouveau paramètre noté p_3 qui représente, pour un problème donné, la proportion de contraintes introduites avant l'énumération. Nous présentons ici des résultats sur des problèmes $\langle 10, 10, 0.5, p_2 \rangle$ avec la dureté p_2 qui varie entre 0.01 et 0.99 par pas de 0.01 et en considérant le paramètre p_3 prenant ses valeurs dans l'ensemble $\{0, 0.25, 0.5, 0.75, 1)\}$. $p_3 = 1$ correspond donc à un comportement classique de maintien d'arc-consistance et $p_3 = 0$ correspond à une approche maintien de solution. Ce paramètre permet en quelque sorte de faire varier la propriété maintenue par le système.

Pour présenter les résultats nous avons choisi une représentation graphique qu'il faut interpréter de la manière suivante :

- pour chaque valeur de p_2, nous avons mesuré la quantité observée (tests de consistance, espace, backtracks, temps) pour des instances de problèmes $\langle 10, 10, 0.5, p_2 \rangle$ et pour chaque valeur de p_3 ;
- nous avons alors reporté les valeurs maximum et minimum de la quantité observée en faisant varier p_3, c'est ce que représente la bande gris foncé sur chaque schéma. De plus, dans la partie inférieure du graphique, nous avons indiqué la valeur de p_3 responsable du minimum (courbe en pointillés) et responsable du maximum (courbe pleine). L'échelle de cette partie du graphique se trouve à droite du schéma ;
- enfin, pour comparaison, nous avons reporté les valeurs correspondant à $p_3 = 1$ qui nous donne les résultats de la section précédente. Ces valeurs nous donnent la courbe en gras représentée dans le schéma supérieur.

Ainsi, sur ce modèle, la figure 10.5 représente les résultats obtenus pour le nombre de tests de consistance.

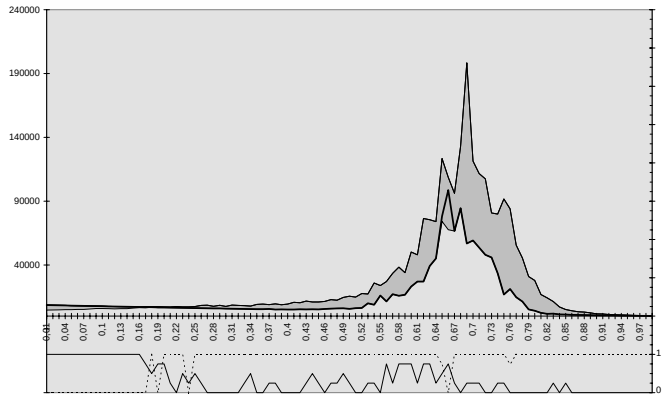

FIG. 10.5 – *Résultats pour le nombre de tests de consistance pour des problèmes* $\langle 10, 10, 0.5, p_2 \rangle$ *en tenant compte de* p_3.

On constate que pour les problèmes très peu contraints ($p_2 < 0.4$) ou pour les problèmes fortement sur-contraints ($p_2 > 0.85$), la valeur de p_3 n'influe pratiquement pas sur le nombre de tests de consistance. Pour les problèmes situés autour de la transition de phase attendue ($p_2 = 0.63$), il est intéressant de constater que la responsabilité pour les valeurs maximum observées est partagée entre les différentes valeurs de $p_3 < 1$ alors que le nombre minimum est pratiquement toujours observé pour $p_3 = 1$.

Pour les autres problèmes, c'est l'approche *maintien de solution* ($p_3 = 0$) qui est la plus coûteuse. Une observation très importante peut être faite : l'amplitude de variation des résultats est faible. On retrouve ce constat sur les figures 10.6 à 10.8 mais l'amplitude est plus forte dans les deux derniers cas.

En ce qui concerne l'occupation en espace (*cf.* Fig. 10.6), on peut tout d'abord noter la faiblesse des valeurs observées qui confirme encore l'intérêt de notre approche raisonnée du maintien de déduction. Sinon, comme pour l'espace occupé,

l'approche maintien d'arc-consistance est la plus coûteuse pour les problèmes très peu contraints mais est la plus économique au fur et à mesure que les problèmes deviennent très sur-contraints. D'autre part, on observe encore le flou autour de la transition de phase et l'étroitesse de la bande de variation des valeurs observées.

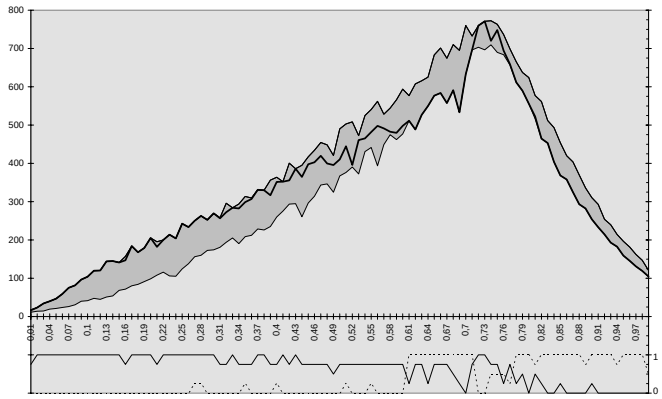

FIG. 10.6 – *Résultats pour l'espace occupé pour des problèmes* $\langle 10, 10, 0.5, p_2 \rangle$ *en tenant compte de* p_3*.*

La figure 10.7 présente les résultats obtenus pour le nombre de contraintes relaxées[11]. Il est intéressant de constater ici que l'approche *maintien d'arc-consistance* est l'approche la plus «*radicale*» (plus de contraintes relaxées) alors que le *maintien de solution* est la plus «*économique*» (moins de contraintes relaxées). Ceci est assez normal, car lorsqu'une contrainte est ajoutée dans le cadre du maintien de solution, comme on cherche à obtenir une solution, les contraintes relaxées le seront pour en obtenir une. En revanche, lors d'un maintien d'arc-consistance les relaxations effectuées ne sont peut être pas suffisantes pour trouver effectivement une solution, le système n'ayant pas assez d'information, il ne relaxe alors pas forcément tout de suite les «*bonnes*» contraintes[12].

Pour les systèmes de contraintes très peu contraints ou très sur-contraints (valeurs extrêmes pour p_2), toutes les approches se valent.

Enfin, lorsque l'on considère le nombre de backtracks effectués (*cf.* Fig. 10.8), on constate que le fait de maintenir l'arc-consistance et donc de n'énumérer que sur des systèmes de contraintes arc-consistants conduit, sans surprise, à un nombre de backtracks moins élevé et ce, quelle que soit la valeur de p_3.

Par contre, la méthode la plus chère n'est pas forcément celle maintenant une solution ($p_3 = 0$). En effet, même si c'est le cas pour la plupart des problèmes, autour de la valeur attendue pour la transition de phase, on retrouve un certain flou. Cette observation mériterait une étude plus poussée. Notons qu'ici la bande de variation est un peu plus large que pour les observations précédentes. En fait, ce résultat est assez normal puisqu'un maintien de solution demande de nombreuses modifications sur la valeur d'une variable car l'information apportée par les contraintes ajoutées ne peut évidemment pas être prise en compte dès le début de la résolution.

On retrouve, sans surprise, pour le temps `cpu` (figure 10.9) les observations

11. De tels problèmes contiennent 23 contraintes. Notons qu'elles ne sont jamais toutes relaxées.
12. Notons tout de même que comme nous utilisons le comparateur C_{MM}, les solutions sont toutes équivalents au regard du comparateur.

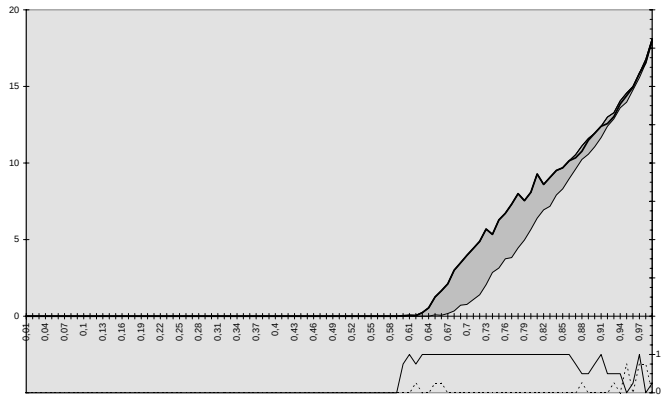

FIG. 10.7 – *Résultats pour le nombre de contraintes relaxées pour des problèmes* $\langle 10, 10, 0.5, p_2 \rangle$ *en tenant compte de* p_3.

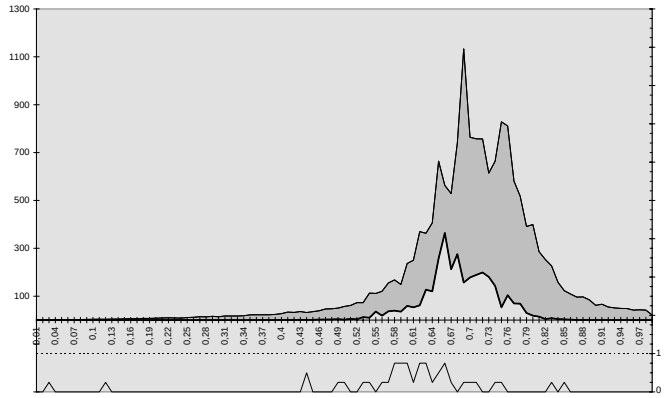

FIG. 10.8 – *Résultats pour le nombre de backtracks pour des problèmes* $\langle 10, 10, 0.5, p_2 \rangle$ *en tenant compte de* p_3.

réalisées pour le nombre de backtracks, car celui-ci est une bonne mesure de l'effort en temps à consentir pour résoudre un problème.

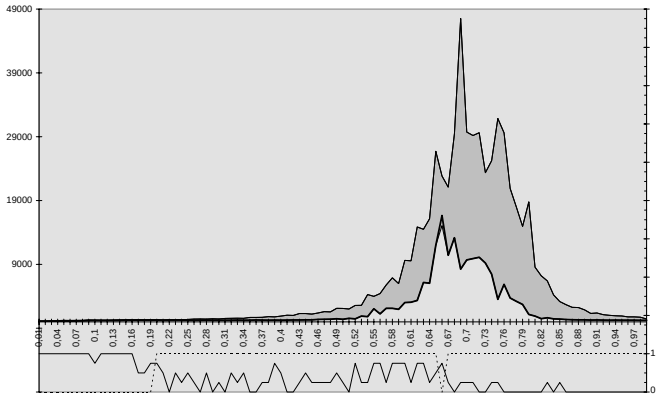

FIG. 10.9 – *Résultats pour le temps* **cpu** *(en ms) pour des problèmes* $\langle 10, 10, 0.5, p_2 \rangle$ *en tenant compte de* p_3*.*

Pour conclure, on peut synthétiser ces résultats en signalant que :

- maintien de solution comme maintien d'arc-consistance sont bien traités par le système DECORUM;
- pour la résolution de problèmes très peu contraints ou au contraire très contraints, les deux approches sont similaires et ne conduisent pas à des différences significatives ;
- en ce qui concerne les problèmes plus *tangents* il est souvent plus coûteux de maintenir une solution, mais la différence avec le maintien d'arc-consistance n'est pas exagérée.

Les performances du système DECORUM ne dépendant que peu de la valeur de p_3, il apparaît comme un outil fiable et stable pour une approche réactive.

10.4 Résolution de problèmes de grande taille

Nous avons porté notre attention sur des problèmes dont les paramètres sont grands pour voir comment se comporte DECORUM. Nous ne proposons pas ici une étude exhaustive et parfaitement significative[13]) de tels problèmes mais plutôt quelques résultats ponctuels pour certains problèmes.

Ainsi, le tableau 10.1 présente quelques résultats pour des instances de problèmes $\langle 50, 25, 0.2, p_2 \rangle$ qui sont déjà assez grands pour montrer l'utilisation pratique de DECORUM sur de grandes instances.

Voici quelques exemples de problèmes et les résultats obtenus :

- Un $\langle 700, 2, 0.05, 0.5 \rangle$ qui comporte donc 12500 contraintes est résolu en 27s[14]

13. En effet, le nombre d'instances pour une même combinaison est inférieur à 35 – nombre d'éléments dans une population à partir duquel on peut faire des hypothèses de normalité qui permettent de réaliser des tests statistiques.

14. Les temps donnés sont pour une résolution effectuée sur une station Sun Sparc Ultra 1 avec 64 Mo de mémoire.

p_2	0.16	0.32	0.96
Tests de consistance	501625	595189	14990
Espace occupé	12270	22236	4820
Relaxations	0	0	191
Backtracks	0	46	1

TAB. 10.1 – *Résultats sur quelques* $\langle 50, 25, 0.2, p_2 \rangle$.

lorsque $p_3 = 1$. Il s'agit d'un problème de satisfaction booléenne puisque les domaines sont de taille 2. Les résultats complets sont répertoriés dans le tableau 10.2.

p_3	Temps	Tests de consistance	Espace	Backtracks	Relaxations
1	27s	211320	18024	0	5967
0	31s	220424	17368	1192	5911

TAB. 10.2 – *Résultats pour un* $\langle 700, 2, 0.05, 0.5 \rangle$

– Un $\langle 120, 5, 0.32, 0.6 \rangle$ comportant donc 2300 contraintes est résolu en 250s pour $p_3 = 1$. Les résultats complets concernant cette résolution sont présentés dans le tableau 10.3. Pour ce problème, la détection de l'incohérence est réalisée avant l'énumération.

  On observe une grande différence sur le nombre de contraintes relaxées pour les deux valeurs de p_3. En fait, énumérer avant l'intégration des contraintes permet de recueillir plus d'informations en cas de contradiction car toutes les variables ont une valeur, comme on l'a vu dans la section précédente.

p_3	Temps	Tests de consistance	Espace	Backtracks	Relaxations
1	250s	963503	54144	5193	1738
0	51s	172070	49581	1240	1397

TAB. 10.3 – *Résultats pour un* $\langle 120, 5, 0.32, 0.6 \rangle$

– La résolution d'un problème $\langle 50, 10, 0.49, 0.75 \rangle$ comportant donc 600 contraintes est résolu en 260s pour $p_3 = 1$. Les résultats complets concernant cette résolution sont présentés dans le tableau 10.4. Pour ce problème, la contradiction est identifiée avant l'énumération.

  D'une manière générale, on peut réaliser les mêmes remarques que pour les problèmes précédents.

10.5 Comparaison avec l'algorithme MAC

L'algorithme MAC est un algorithme de résolution de CSP [Sabin et Freuder, 1994] qui a la particularité de propager les contraintes du problème après chaque nouvelle instanciation[15]. Pour pouvoir réaliser une comparaison avec DECORUM, nous l'avons implémenté avec les mêmes structures[16]. Il ne s'agit donc pas forcément de la meilleure implémentation possible mais elle permet de donner un sens à nos comparaisons.

15. Ce principe de propagation systématique après chaque instanciation est celui choisi par tous les systèmes de PLC.

16. Nous avons aussi utilisé la même base pour la propagation: l'algorithme AC4 [Mohr et Henderson, 1986].

p_3	Temps	Tests de consistance	Espace	Backtracks	Relaxations
1	260s	927587	31229	1980	517
0	588s	2555450	38579	25904	362

TAB. 10.4 – *Résultats pour un* $\langle 50, 10, 0.49, 0.75 \rangle$

De plus, DECORUM a été modifié pour arrêter la résolution dès que le système de contraintes initial est montré comme étant contradictoire (découvert d'une contradiction ne pouvant être invalidée par la relaxation d'une contrainte liée à l'énumération). Ainsi, DECORUM se comporte comme une intégration de la propagation à l'algorithme *Dynamic Backtracking* (*cf.* section 3.4.3 page 34) comme MAC est une intégration de la propagation au *backtrack* standard.

Nous présentons les résultats moyens obtenus pour des séries de 35 problèmes aléatoires $\langle 15, 10, 0.47, p_2 \rangle$ pour lesquels p_2 varie de 0.4 à 0.65. Ces problèmes mettent en jeu 50 contraintes. La dureté (p_2) critique a pour valeur 0.49.

10.5.1 Problèmes aléatoires et DECORUM

Pour chaque série de problèmes, nous avons recuilli le nombre de tests de consistance réalisés, le nombre de backtracks[17] réalisés et le temps `cpu`[18] nécessaire à la réalisation en ms.

Les figures 10.10 à 10.12 représentent graphiquement les résultats obtenus.

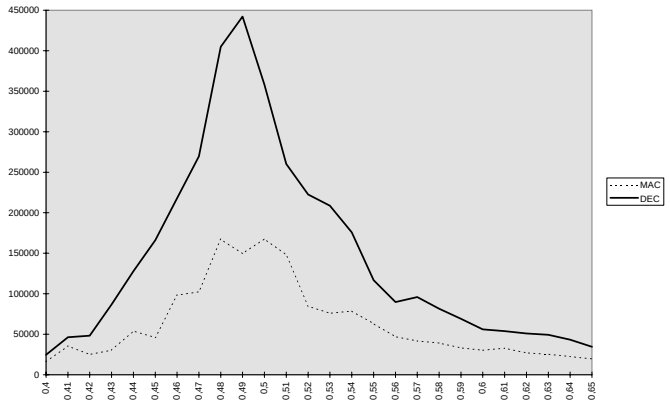

FIG. 10.10 – *Moyennes du nombre de tests de consistance réalisés pour des problèmes* $\langle 15, 10, 0.5, p_2 \rangle$ *pour* DECORUM *et* MAC.

On constate que DECORUM présente de moins bons résultats que MAC. Mais, le *surcoût* observé reste limité : rappelons que la complexité dans le pire cas de l'ajout de contrainte est modifié par un facteur d (10) alors que le surcoût est bien inférieur : ici de l'ordre de 4 dans le pire cas (autour de la transition de phase).

Le surcoût observé sur le nombre de *backtracks* peut surprendre pour une méthode censée améliorer les recherches par backtrack. Ceci s'explique par le fait que dans un CSP aléatoire peu ou aucune information pertinente ne peut être obtenue pour identifier de bons points de backtracks lorsqu'il existe une solution. En effet,

17. Au sens PLC.
18. Les tests ont été réalisés sur une Sun Sparc Ultra 1 avec 64 Mo de mémoire.

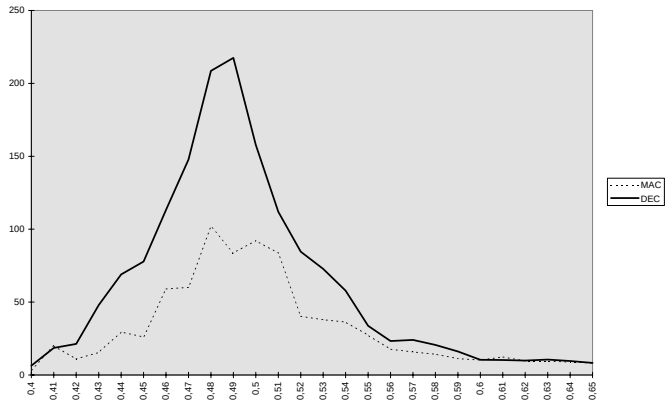

FIG. 10.11 – *Moyennes du nombre de backtracks pour des problèmes* $\langle 15, 10, 0.5, p_2 \rangle$ *pour* DECORUM *et* MAC.

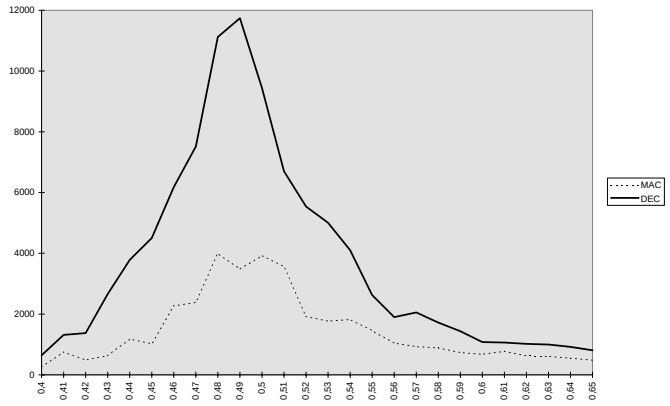

FIG. 10.12 – *Moyennes du temps* `cpu` *en ms nécessaire pour résoudre des problèmes* $\langle 15, 10, 0.5, p_2 \rangle$ *pour* DECORUM *et* MAC.

de tels problèmes n'ont pas de structure. DECORUM n'est donc pas utilisé au maximum de ses possibilités. Ceci explique pourquoi le surcoût naturel de l'utilisation de DECORUM n'est pas compensé par une recherche améliorée.

10.5.2 Problèmes structurés et DECORUM

Pour introduire une structure dans les problèmes étudiés, nous avons choisi de procéder de la façon suivante. L'idée exploitée est de résoudre des problèmes décomposables en plusieurs sous-problèmes complètement indépendants. Ainsi, lorsque nous générons un problème aléatoire, nous le dupliquons autant de fois que nécessaire.

Ensuite, l'énumération commence par les premières variables pour chacun des problèmes imbriqués, puis les deuxièmes variables...

Notons que l'utilisation d'une heuristique pour ordonner l'énumération, qu'elle soit statique et basée sur la structure du problème résolu (taille des domaines, nombre de contraintes incidentes, ...), ou dynamique et basée sur la taille des domaines ou la structure du graphe de contraintes, conduirait quasiment au même ordre puisque toutes les n-ème variables de chaque problème sont parfaitement équivalentes pour ces critères. En ce qui concerne une éventuelle heuristique dynamique, nous considérons les problèmes ne provoquant presque pas de réduction dans les domaines après les instanciations d'une bonne partie des variables.

Ce processus de résolution pour les problèmes étudiés conduit dans le cas de MAC non pas à une croissance linéaire des statistiques observées mais à une croissance exponentielle. En effet, leurs variables étant imbriquées, les problèmes ne sont pas résolus successivement mais simultanément ainsi un retour sur une variable d'un problème défait en partie ce qui a été fait pour les autres problèmes. Notons qu'un mécanisme utilisant le *Backtrack Intelligent* souffre du même défaut même si c'est dans une moindre mesure. Par contre, dans le cas de DECORUM, ce découplage interne des problèmes sera pris en compte par le mécanisme utilisé pour la relaxation de contrainte qui évite le *thrashing*.

Les figures 10.13, 10.14 et 10.15 représentent graphiquement les résultats obtenus pour les mêmes séries de problèmes que dans la section précédente mais avec 2 problèmes imbriqués. Les problèmes étudiés sont donc des $\langle 30, 10, 0.23, p_2 \rangle$.

FIG. 10.13 – *Moyennes du nombre de tests de consistance réalisés pour des séries de 2 problèmes* $\langle 15, 10, 0.5, p_2 \rangle$ *imbriqués pour* DECORUM *et* MAC.

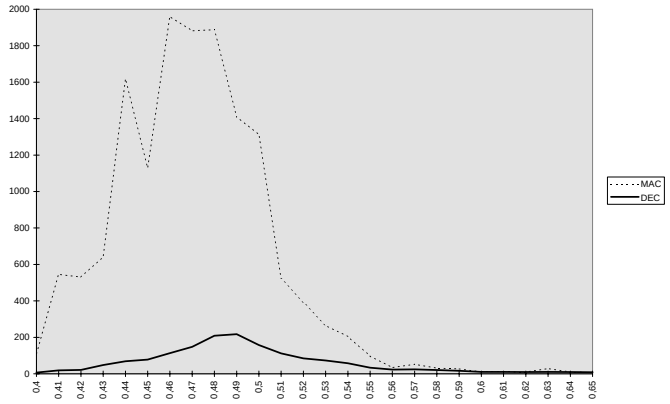

FIG. 10.14 – *Moyennes du nombre de backtracks pour des séries de 2 problèmes* $\langle 15, 10, 0.5, p_2 \rangle$ *imbriqués pour* DECORUM *et* MAC.

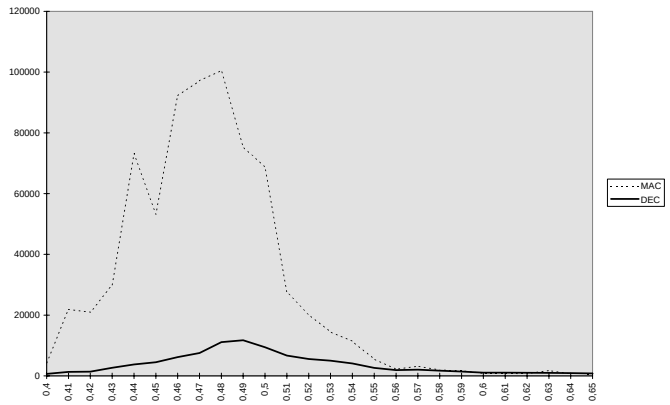

FIG. 10.15 – *Moyennes du temps* **cpu** *en ms nécessaire pour résoudre des séries de 2 problèmes* $\langle 15, 10, 0.5, p_2 \rangle$ *imbriqués pour* DECORUM *et* MAC.

Les résultats montrent très nettement l'avantage de DECORUM sur MAC. En fait, DECORUM devrait en toute logique multiplier par 2 dans le pire cas les résultats obtenus en terme de nombre de backtracks pour les instances de la section précédente alors que, MAC peut voir ses performances être portées au moins au carré. Le tableau 10.5 présente les facteurs multiplicatifs minimum, maximum et moyen obtenus pour les deux algorithmes entre les deux séries. De même, le tableau 10.6 présente les facteurs exponentiateurs minimum, maximum et moyen obtenus pour les deux algorithmes.

Ces tableaux montrent parfaitement l'excellent comportement de DECORUM par rapport à MAC. En particulier, on voit que MAC peut aller jusqu'à porter les résultats de la première série à la puissance 2.60. Autour de la transition de phase, le coefficient multiplicateur varie entre 1.4 et 2 pour DECORUM alors qu'il varie entre

	DECORUM	MAC
Tests de Consistance	1.07 – 2	1.03 – 42.05
Moyenne	1.42	11.95
Backtracks	1–2	0.85 – 54.7
Moyenne	1.37	15.82
Temps `cpu`	0.96 – 1.87	1 – 62.12
Moyenne	1.30	17.35

TAB. 10.5 – *Coefficients multiplicatifs minimum, maximum et moyen entre la série originelle et la série avec deux problèmes imbriqués.*

	DECORUM	MAC
Tests de Consistance	1 – 1.06	1 – 1.34
Moyenne	1.02	1.16
Backtracks	1 – 1.37	0.93 – 2.60
Moyenne	1.07	1.60
Temps `cpu`	0.99 – 1.09	1 – 1.60
Moyenne	1.02	1.29

TAB. 10.6 – *Coefficients exponentiateurs minimum, maximum et moyen entre la série originelle et la série avec deux problèmes imbriqués.*

18 et 31 pour MAC. Le coefficient exponentiateur, quant à lui varie entre 1.04 et 1.05 pour DECORUM et entre 1.6 et 1.84 pour MAC. C'est la partie la plus intéressante à étudier. On voit que sur ces problèmes difficiles les résultats de MAC évoluent beaucoup plus vite que pour DECORUM.

On voit ainsi que lorsque de l'information peut être utilisée, le système DECORUM compense largement le surcoût lié au DMS par une exploration améliorée. L'information utilisée ici concerne les liens entre les variables du problème, nous verrons dans le chapitre suivant que lorsque l'information peut être obtenue sur les contraintes le gain est du même ordre.

10.6 Conclusion

Les résultats que nous avons présentés ici nous permettent de constater que :

- le système DECORUM respecte la transition de phase sur les CSP aléatoires. C'est donc un système aussi fiable qu'un système usuel, on peut identifier des classes de problèmes particulièrement ardus à résoudre ;
- le système DECORUM montre sa stabilité quel que soit le degré de dynamicité utilisé ;
- le système DECORUM montre qu'il peut être utilisé pour des problèmes de grande taille ;
- enfin, le système DECORUM montre sa supériorité sur des algorithmes classiques dès lors que le problème à résoudre est un tant soit peu structuré, ce qui est le cas d'un très grand nombre de problèmes réels. Il montre aussi que l'utilisation d'un DMS n'induit pas de dérives lors de la résolution de CSP.

Chapitre 11

Traitement de la disjonction et contrainte `one-of`

Sommaire

La contrainte **one-of** (section 7.3.3), traduit l'appartenance de la valeur d'une variable à un domaine comme une disjonction (exclusive) de contraintes d'égalité entre la variable et les valeurs de son domaine. Ainsi, x de domaine $\{1, 2, 3\}$ se traduit par **one-of**$(x = 1, x = 2, x = 3)$. Dans le chapitre 7, la contrainte **one-of** nous a permis de modéliser l'énumération comme une succession d'ajouts/retraits de contraintes d'égalité.

Dans ce chapitre, nous montrons comment la contrainte **one-of** peut être efficacement utilisée pour traiter la disjonction exclusive de contraintes quelconques.

L'idée maîtresse est la même que pour l'énumération. Les contraintes **one-of** tentent chacune d'installer leur première contrainte. Lorsqu'un échec se produit, un point de retour est identifié et la contrainte **one-of** incriminée installe sa contrainte suivante (le travail indépendant entre les deux n'est pas défait).

Dans ce chapitre, nous avons choisi comme domaine d'application privilégié les problèmes d'ordonnancement purement disjonctifs appelés Open-Shops.

11.1 Traitements de la disjonction

Les problèmes définis par une disjonction de contraintes introduisent une forte combinatoire [de Backer, 1995]. Ces disjonctions de contraintes se rencontrent par exemple dans les problèmes d'allocation de ressources[1] où une ressource ne peut être partagée à un instant donné.

Différentes approches pour la résolution de tels problèmes sont envisageables :

- considérer une disjonction comme un simple point de choix et laisser le backtrack standard faire son office. Une telle approche construit un nombre exponentiel de systèmes de contraintes à résoudre. De plus, on est souvent amené à explorer toutes ces possibilités pour déterminer la meilleure ;

- reporter le point de choix jusqu'à ce qu'une décision puisse être prise. Ce mécanisme peut être mis en œuvre, par exemple, en utilisant le mécanisme de démon de CHIP [Dincbas *et al.*, 1988] ou l'opérateur de Cardinalité de Van Hentenryck et Deville [1991] qui permet de borner le nombre de contraintes devant être vérifiées dans un ensemble donné. Une telle approche n'a d'intérêt que si les autres contraintes du problème permettent de prendre assez vite des décisions et est donc peu intéressante lorsqu'il y a beaucoup de contraintes disjonctives ;

- utiliser des cas particuliers de contraintes globales (comme la contrainte *cumulative* de CHIP [Aggoun et Beldiceanu, 1993]) lorsque la disjonction s'y prête bien. On pourra se reporter à [Varnier *et al.*, 1993; Boizumault *et al.*, 1995; Boizumault *et al.*, 1996] pour des exemples d'utilisation. Il faut noter que l'utilisation de la contrainte cumulative est subordonnée au fait que la disjonction traitée provient d'une utilisation dans un cadre « *allocation de ressource*[2] » ;

- enfin, on peut utiliser la *disjonction constructive* [Jourdan et Sola, 1993] qui extrait immédiatement l'information commune aux membres de l'alternative. Bien sûr, lorsque les contraintes prises en compte sont mutuellement exclusives, la disjonction constructive n'apporte que très peu d'information.

En conclusion, les techniques utilisées ont principalement le défaut d'utiliser de manière passive les contraintes faisant partie d'une disjonction. Ce n'est pas le cas

1. *cf.* le *Hoist Scheduling Problem* [Varnier *et al.*, 1993], ou plus simplement les problèmes classiques d'ateliers : Job-Shop, Open-Shop et Flow-Shop.

2. On peut très bien imaginer une disjonction, par exemple, entre les contraintes $x = 2$ et $y < z + 3$ qui ne peut pas s'exprimer à l'aide de la contrainte cumulative.

lorsqu'on utilise des contraintes globales mais celles-ci ne sont pas adaptées à toutes les situations.

Dans le cadre des contraintes linéaires sur les rationnels, différents résultats sont présentés dans [de Backer, 1995]. Il en ressort plusieurs points intéressants :

- on peut se rendre compte du non intérêt de membres d'une alternative. Les techniques alors employées sont plutôt adaptées aux contraintes linéaires sur les rationnels car elles font appel à une interprétation géométrique qu'il est difficile de retrouver dans le cadre général qui nous intéresse ;
- utiliser des techniques basées sur le backtrack intelligent permet de rendre intéressante l'utilisation d'un point de choix standard pour modéliser la disjonction car alors les contraintes sont utilisées de manière active et l'*intelligence* du backtrack permettra de remettre assez vite en cause un mauvais choix ;
- il est intéressant non seulement d'utiliser des explications sur un échec pouvant identifier un point de choix pertinent à remettre en cause, mais aussi de conserver ces explications pour éviter de retomber dans une sous-configuration déjà explorée et que l'on sait conduire à une contradiction.

On reconnaît des idées à la base de notre proposition pour un système de relaxation de contraintes. C'est pourquoi, nous proposons d'utiliser la contrainte `one-of` pour modéliser les disjonctions.

11.2 Notre proposition

L'utilisation de la méta-contrainte `one-of` dans le cadre de notre système de relaxation de contrainte permet :

- d'étendre le backtrack intelligent à un mécanisme qui n'est plus du backtrack mais une technique basée sur la réutilisation : on identifie un *bon* point de choix à remettre en cause et on ne défait pas le travail indépendant fait depuis l'intégration de ce choix ;
- de ne pas retomber dans une sous-configuration contradictoire déjà rencontrée. Cette propriété est assurée par notre méthode de recherche et de conservation des explications de contradiction (même dans le cadre restrictif des conditions de fonctionnement **C1** et **C2**).

Cette proposition améliore une approche basée sur le *backtrack intelligent* ([de Backer, 1995]) de deux manières complètement distinctes :

- d'une part, c'est le plus visible, les explications conduisant à la remise en cause d'une possibilité pour l'alternative, sont utilisées pour ne pas défaire les calculs réalisés entre temps et complètement indépendants de ce choix malheureux ;
- d'autre part, contrairement par exemple à l'approche proposée dans [de Backer, 1995], lorsque tous les membres d'une alternative ont été rejetés, on ne passe pas au nœud père mais on utilise plutôt les explications de retrait de contraintes pour identifier un point de remise en cause *intelligent*. Notons que, Thierry Sola [1995] utilise aussi cette amélioration.

11.3 Étude sur un problème purement disjonctif

Pour tester l'intérêt des contraintes `one-of` pour modéliser une disjonction, nous avons choisi de nous intéresser à une famille de problèmes d'ordonnancement purement disjonctif : l'Open-Shop.

11.3.1 Le problème d'Open-Shop

Le problème d'Open-Shop fait partie des problèmes d'ateliers[3]. On considère un ensemble M de m machines $\{M_1, M_2, \ldots, M_m\}$ et un ensemble N de n jobs $\{J_1, J_2, \ldots, J_n\}$ à exécuter sur ces machines. Chaque job J_j est composé de m tâches $O_{1j}, O_{2j}, ..., O_{mj}$. Chaque tâche O_{ij} a une durée entière p_{ij} et doit être exécutée sur la machine i. Les contraintes du problème sont les suivantes :

- une machine ne peut exécuter qu'une seule tâche à la fois ;
- plusieurs tâches d'un même job ne peuvent être exécutées en même temps.

Il s'agit de trouver un ordonnancement (donner une date de démarrage à chaque tâche) se terminant le plus tôt possible.

Contrairement au Job-Shop ou au Flow-Shop, le séquencement[4] des tâches pour chaque job n'est pas imposé. Ce problème ne contient donc pas de contrainte de précédence.

Le problème d'Open-Shop est uniquement spécifié par des contraintes de ressource. Ces contraintes peuvent être définies simplement par des disjonctions entre toutes paires de tâches (I, J) sur un même job ou sur une même machine : `one-of(`I avant J, J avant I`)`.

Il s'agit donc d'un cadre idéal pour tester notre proposition pour le traitement des disjonctions. En effet, comme il n'y a aucune autre contrainte, les techniques basées sur la mise en attente des disjonctions sont peu efficaces. Il serait possible d'utiliser une contrainte globale comme la contrainte *cumulative* de CHIP, mais nous cherchons à montrer l'intérêt de la contrainte `one-of` d'une manière générale.

Tout d'abord, nous pouvons remarquer que le problème d'Open-Shop est totalement symétrique : les rôles des machines ou des jobs sont interchangeables. En pratique, on considérera la vision permettant d'avoir le plus petit nombre de machines. Dans la suite, nous considérons des problème d'Open-Shop pour lesquels $n \geqslant m$.

Gonzales et Sahni [1976] ont montré que l'optimisation d'un ordonnancement de type Open-Shop est un problème NP-difficile au sens fort[5] dès que $m > 2$. De plus, les problèmes carrés ($n = m$) sont considérés comme plus difficiles que les problèmes rectangulaires ($n > m$) [Guéret et Prins, 1998].

Ce problème est plus difficile que le problème de Job-Shop ou de Flow-Shop car il n'offre que peu de prise : il n'y a pas de contraintes de précédence auxquelles se raccrocher. De plus, jusqu'à très récemment, il n'existait qu'une borne inférieure : la borne inférieure classique (la charge[6] maximum pour les jobs et pour les machines). De bonnes heuristiques ont été récemment proposées pour l'Open-Shop [Guéret et Prins, 1998]. Il ressort des expérimentations qui y sont présentées que :

- plus le problème est grand (dès $m = 9$), plus les heuristiques donnent de très bons résultats car il y a très peu de problèmes difficiles[7] ;
- les problèmes à grand *gap* (différence entre l'optimum et la borne inférieure classique) sont les plus difficiles. De plus, les problèmes à grand *gap* sont

3. Comme le Flow-Shop et le Job-Shop.

4. Pour le Job-Shop on donne un ordre de passage sur les machines pour chaque job. Pour le Flow-Shop cet ordre est le même pour tous les jobs.

5. Un problème d'optimisation est NP-difficile si le problème d'existence qui lui est associé (existe-t-il un ordonnancement de durée inférieure ou égal à K) est NP-complet. Un problème d'optimisation est NP-difficile au sens fort si le problème reste NP-difficile pour un codage unaire des données.

6. La charge est la somme des durées des opérations concernées – opérations du job ou opérations de la machine.

7. Un problème est difficile lorsque le nombre de solutions optimales est faible et lorsque les solutions non optimales les plus proches en valeur sont très éloignées (toujours en valeur).

souvent ceux dont les charges sur les machines et sur les jobs sont toutes égales. Ceci confirmait la conjecture de [Brucker *et al.*, 1994].

Nous avons donc choisi de réaliser notre étude sur des petits problèmes carrés ($n = m = 3$ et $n = m = 4$) à grand *gap* dont les machines et les jobs ont une même charge : 1000. Nous cherchons simplement à résoudre un problème d'existence : existe-t-il un ordonnancement terminant avant une valeur donnée, en l'occurrence l'optimum? Ces problèmes (même d'existence) sont considérés comme parmi les plus difficiles.

11.3.2 Premiers résultats

Nous modélisons donc les problèmes à traiter à l'aide d'une contrainte `one-of` pour chaque couple de tâches en disjonction. Il y en a en tout $n^2(n-1)$ pour un problème carré de taille $n \times n$.

Les contraintes en jeu ne travaillant que sur les bornes des domaines (à la manière des intervalles), une solution réalisable est obtenue sans énumération en affectant le minimum de son domaine à chaque variable (cela correspond en fait à la notion de date au plus tôt).

Méthodes comparées

Nous avons cherché à comparer DECORUM avec un backtrack intelligent et un backtrack standard. En effet, dans un problème d'Open-Shop toutes les variables sont liées entre elles. Il est donc intéressant, d'une part, de voir si un backtrack intelligent apporte quelque chose et, d'autre part, voir si la lutte contre le *thrashing* menée par DECORUM apporte quelque chose de plus.

Nous avons donc trois méthodes : une basée sur DECORUM (`decorum`), une basée sur le backtrack intelligent (`bckint`) et enfin une basée sur le backtrack standard (`bckstd`).

Il est bien évident que les problèmes d'Open-Shop ne sont pas traités aussi simplement. Nous avons donc choisi d'intégrer quelques améliorations qui, malgré tout, restent limitées.

Pour cela, on utilise les résultats suivants (r_i est la date au plus tôt de la tâche i, p_i son temps d'opération et q_i sa durée de latence [8]).

Ainsi, pour une tâche i, la quantité $r_i + p_i + q_i$ représente une borne inférieure de la durée d'un ordonnancement contenant la tâche i. Pour deux tâches i et j, la quantité $r_i + p_i + p_j + q_j$ est une borne inférieure d'un ordonnancement pour lequel la tâche i est placée avant la tâche j.

Si les deux membres de l'alternative (i avant j et j avant i) sont disponibles alors :

- si $r_i + p_i \leqslant r_j$ alors nécessairement i est avant j. On peut même signaler (en ajoutant une explication de retrait de contrainte) que j *avant* i n'aura pas à être testée (l'explication est donnée par l'explication des retraits des valeurs de $i \leqslant r_i$ et celles de $j \leqslant r_j$) ;
- si $r_j + p_j \leqslant r_i$ alors nécessairement j est avant i, c'est la proposition duale de la précédente ;

8. Lorsqu'on considère le graphe conjonctif (prenant en compte les précédences entre tâches) du problème (au démarrage il n'y a que les précédences relatives à la fin de l'ordonnancement) pour lequel la source (tâche fictive origine de toutes les autres) est notée $\circ$ et la fin de l'ordonnancement est notée $\star$, si on note $l(i,j)$ un plus long chemin entre i et j dans le graphe conjonctif alors: $r_i = l(\circ, i)$ et $q_i = l(i, \star) - p_i$. Notons que $l(i,\star)$ peut être calculé à partir de f_i la date de démarrage au plus tard de i en utilisant la formule $l(i,\star) = f_{fin} - f_i$. Enfin, les dates au plus tôt et au plus tard sont directement accessibles sous la forme des bornes des domaines des variables concernées.

- si $r_i + p_i + p_j + q_j > UB$ [9] alors nécessairement j est avant i. C'est un résultat de Carlier et Pinson [1989] applicable à l'Open-Shop. On peut là aussi spécifier que i avant j n'aura pas à être testée en ajoutant l'explication de retrait de contrainte déterminée à partir des valeurs de $i \leqslant r_i$ et $j \leqslant r_j$;

- si $r_j + p_j + p_i + q_i > UB$ alors nécessairement i est avant j. C'est la proposition duale de la précédente.

La prise en compte ou non de ces améliorations nous donnent donc en tout 6 méthodes : `decorum` et `decorum*`, `bckint` et `bckint*` et `bckstd` et `bckstd*`. Une méthode décorée d'une `*` représentant la méthode avec les améliorations.

Il faut signaler que nous utilisons cette façon de résoudre le problème uniquement pour présenter l'intérêt de la recherche en meilleur d'abord et de son utilisation pour les contraintes `one-of`. Il est bien évidemment qu'on aurait tout intérêt à utiliser le fait que des ensembles de tâches sont en disjonction et non pas uniquement deux par deux, en utilisant, par exemple, des résultats sur le Job-Shop [Carlier et Pinson, 1994].

Résultats expérimentaux

Nous avons relevé le nombre de contraintes introduites puis rejetées [10] (membres d'une disjonction) pour chaque méthode jusqu'à obtention de la solution pour des Open-Shop carrés de taille 3×3.[11] Notons que le nombre maximum de *backtracks* est, pour ces problèmes, théoriquement de 2^{18}.

Les tableaux 11.1 et 11.2 synthétisent les résultats obtenus.

3×3	1	2	3	4	5	6
1	–	90	100	55	83	100
2	10	–	99	38	60	98
3	0	1	–	0	2	61
4	45	62	100	–	100	100
5	17	40	98	0	–	100
6	0	2	39	0	0	–

TAB. 11.1 – *Nombre de problèmes dont la recherche est améliorée par la méthode en ligne par rapport à la méthode en colonne. Les méthodes* `decorum*` *(1),* `bckint*` *(2),* `bckstd*` *(3),* `decorum` *(4),* `bckint` *(5) et* `bckstd` *(6) sont considérées.*

Ces tableaux nous permet de constater que l'effort consenti pour résoudre ces problèmes en utilisant notre proposition (que ce soit avec la méthode `decorum` ou `decorum*`) est bien moindre que pour une méthode basée sur le *backtrack intelligent*. Ceci est tout à fait compréhensible puisque l'information qui a permis d'identifier le point de *backtrack* est, dans notre proposition, utilisée au maximum, permettant ainsi de ne modifier que ce qui est nécessaire.

À partir des résultats présentés dans le tableau 11.2, des tests de comparaisons de moyennes [12] montrent que les résultats obtenus avec les améliorations sont significativement [13] meilleurs que les résultats sans améliorations excepté pour les

9. UB est une borne supérieure de l'ordonnancement (on peut prendre la borne supérieure de la variable correspondant à la fin de l'ordonnancement).

10. Nous appelons ce nombre, le nombre de *backtracks*. C'est le nombre de disjonctions arbitrées dans le mauvais sens.

11. Rappelons que même et surtout les petits problèmes sont difficiles.

12. Attention, on est dans le cas de *séries appariées*, il faut donc comparer la moyenne des différences à zéro et non pas les deux moyennes directement.

13. Les tests sont réalisés à 5% ce qui signifie que l'on a 5 chances sur 100 de se tromper lorsque l'on conclut que les différences sont significatives.

3×3	1	2	3	4	5	6
Moyenne	55.96	114.37	3134.86	64.7	136.69	9280.4
Variance	1369.18	5142.19	$9.27668e+07$	1792.03	8020.15	$2.76654e+08$
Minimum	14	34	74	15	33	153
Maximum	188	486	88794	230	421	66322

TAB. 11.2 – *Statistiques élémentaires sur* 100 *Open-Shop carrés pour les méthodes* `decorum*` *(1),* `bckint*` *(2),* `bckstd*` *(3),* `decorum` *(4),* `bckint` *(5) et* `bckstd` *(6).*

méthodes basées sur DECORUM pour lesquelles les résultats ne sont pas significativement différents. De plus, toutes les différences sont statistiquement significatives pour tout couple de méthodes. Quelles interprétation pouvons-nous déduire des résultats présentés?

- Notre proposition (avec ou sans améliorations: méthodes `decorum*` et `decorum`) améliore significativement les résultats des autres méthodes. On gagne ainsi un facteur 2 par rapport au *backtrack intelligent* et un facteur 100 par rapport au *backtrack standard.*

  On fait souvent le reproche aux techniques basées sur le *backtrack intelligent* de demander trop de temps pour simplement remonter plus haut dans l'arbre de recherche alors que souvent, un backtrack simple irait plus vite. Dans le cadre de notre approche un tel reproche semble plus difficile à faire: en effet, non seulement les enregistrements d'informations réalisés permettent d'identifier un bon point de *retour* mais aussi ils permettent de ne pas tout reprendre: ils évitent le phénomène de *thrashing* [14].

- L'utilisation des améliorations, intéressante pour les méthodes basées sur le backtrack, n'apporte rien dans le cadre de notre proposition.

  Ceci est dû au fait que faire un mauvais choix lors d'une alternative n'est pas trop pénalisant pour nos méthodes puisque celles-ci permettent de corriger un tel choix sans tout remettre en cause.

- Notre proposition présente une variabilité des résultats beaucoup plus faible que les autres approches. Ceci peut s'expliquer par le fait que notre proposition est beaucoup plus indépendante de la position de la solution dans l'arbre de recherche que les autres méthodes: en fait, quelle que soit la position de la solution dans l'arbre, une méthode basée sur le backtrack doit explorer (explicitement ou implicitement) toute la partie de l'arbre située à gauche de la solution. Dans notre approche (par sauts) ce n'est plus vrai et c'est aussi un point intéressant car les résultats restent stables quel que soit le problème traité.

Nous avons cherché à savoir si ces résultats étaient dûs aux problèmes étudiés. C'est pourquoi nous avons réalisé des tests sur des problèmes plus grands: des Open-Shop 4×4 choisis de la même manière que les 3×3.

Les tableaux 11.3 et 11.4 récapitulent les résultats obtenus.

On constate au vu de ces résultats, que:

- notre proposition est bien meilleure que le *backtrack intelligent*;

14. Nous ne proposons pas de comparaison en terme de temps de calcul car les simulations effectuées pour les backtrack standard et intelligent utilisent DECORUM ce qui pénalise les retours arrière. À titre d'information, en utilisant les simulations, le temps moyen de résolution d'un 3×3 est de 300 ms avec `decorum*`, 800 ms avec `bckint*` et 2 min avec `bckstd*` sur une station Sun Sparc Ultra 1 disposant de 64 Mo de mémoire.

4×4	1	2	3	4
Moyenne	8673.95	49095.6	8515.94	37852.9
Variance	$6.09692e+07$	$1.50614e+09$	$8.55807e+07$	$1.07438e+09$
Minimum	166	841	185	1673
Maximum	44027	198949	50807	163048

TAB. 11.3 – *Statistiques élémentaires pour les méthodes* `decorum*` *(1),* `bckint*` *(2),* `decorum` *(3) et* `bckint` *(4) sur* 100 *Open-Shop* 4×4 *difficiles.*

4×4	1	2	3	4
1	–	100	38	87
2	0	–	15	28
3	62	85	–	100
4	13	72	0	–

TAB. 11.4 – *Nombre de problèmes dont la recherche est améliorée par la méthode en ligne par rapport à la méthode en colonne.*

- l'utilisation des améliorations apporte peu pour notre méthode. Par contre, étonnamment, elles ont un effet négatif sur la méthode basée sur le *backtrack intelligent* ;

- le gain par rapport au *backtrack intelligent* n'est plus d'un facteur 2 mais d'un facteur 5 à 6.[15]

Les résultats obtenus sur les 3×3 sont donc confirmés.

11.3.3 Résolution de problèmes plus faciles

Nous nous sommes intéressés à la stabilité des résultats lorsqu'on fait varier la difficulté du problème. Pour cela, nous avons cherché une solution non plus inférieure ou égale à l'optimum mais plutôt inférieure ou égale à l'optimum multiplié par une constante (strictement supérieure à 1).

Ainsi, nous avons calculé le nombre de *backtracks* pour chacune des méthodes en utilisant les coefficients de facilitation[16] 1.1, 1.2, 1.3 et 1.7. Le tableau 11.5 récapitule les statistiques élémentaires sur le nombre de *backtracks* pour chacune des 6 méthodes et pour chacun des coefficients utilisés pour résoudre les 100 Open-Shop 3×3 qui nous servent de référence. De même, les tableaux 11.6 récapitulent le nombre de problèmes dont la résolution est améliorée comme dans le tableau 11.1.

On peut faire les observations[17] suivantes :

- même si le problème étudié devient facile, la méthode `decorum*` reste la plus performante que ce soit en terme de nombre de backtracks ou en terme de variabilité des résultats ;

15. En ce qui concerne le temps de calcul, avec les réserves que nous avons déjà présentées et uniquement à titre d'information, une résolution avec `decorum*` demande 1 min 20 tandis qu'un résolution avec `bckint*` demande 14 min sur une station Sun Sparc Ultra 1 disposant de 64 Mo de mémoire.

16. Il est intéressant de signaler ici qu'il a été démontré [Williamson *et al.*, 1992] qu'il n'existait aucune heuristique polynômiale permettant d'assurer le calcul d'une solution à moins de 5/4 de l'optimum. De plus, aucune heuristique fournissant à tout coup une solution à moins de 2 fois l'optimum n'a encore été trouvée.

17. Dans la suite, lorsqu'il est dit que deux méthodes donnent des résultats différents (ou le contraire), cette affirmation n'est pas simplement faite sur la base des résultats présentés mais elle est confirmée par un test statistique montrant la significativité de la différence.

Coeff. 1.1	1	2	3	4	5	6
Moyenne	34.72	64.77	1561.08	49.24	93.76	3079.12
Variance	615.262	1723.62	$1.84921e+07$	1422.06	4543.04	$3.75931e+07$
Minimum	5	6	8	8	15	27
Maximum	107	194	32260	232	338	45747
Coeff. 1.2	1	2	3	4	5	6
Moyenne	29.34	45.73	682.68	45.99	75.18	1569.23
Variance	435.364	1434.42	$1.46616e+06$	844.53	2478.27	$4.49006e+06$
Minimum	0	0	0	7	7	7
Maximum	101	169	8211	128	227	16409
Coeff. 1.3	1	2	3	4	5	6
Moyenne	22.53	26.96	357.92	41.36	59.17	1351.52
Variance	323.369	988.658	621326	1093.07	3457.2	$6.4496e+06$
Minimum	0	0	0	3	3	3
Maximum	116	203	6035	137	273	12882
Coeff. 1.7	1	2	3	4	5	6
Moyenne	10.52	18.89	82.11	10.95	15.53	66.93
Variance	339.73	1153.96	45703.4	171.147	371.529	16455.6
Minimum	0	0	0	0	0	0
Maximum	168	305	1737	56	81	601

TAB. 11.5 – *Statistiques élémentaires pour les* 6 *méthodes sur les* 100 *Open-Shop carrés* 3×3 *facilités par les coefficients* 1.1, 1.2, 1.3 *et* 1.7 *(de haut en bas).*

1.1	1	2	3	4	5	6
1	–	82	98	66	86	100
2	18	–	100	39	65	98
3	2	0	–	8	16	70
4	34	61	92	–	100	100
5	14	35	84	0	–	100
6	0	2	30	0	0	–

1.2	1	2	3	4	5	6
1	–	68	92	72	85	96
2	32	–	100	59	68	95
3	8	0	–	19	26	65
4	28	41	81	–	99	100
5	15	32	74	1	–	100
6	4	5	35	0	0	–

1.3	1	2	3	4	5	6
1	–	54	92	73	78	92
2	46	–	100	66	73	91
3	8	0	–	29	36	63
4	27	34	71	–	95	100
5	22	27	64	5	–	100
6	8	9	37	0	0	–

1.7	1	2	3	4	5	6
1	–	97	97	46	53	60
2	3	–	100	34	40	51
3	3	0	–	29	34	46
4	54	66	71	–	100	100
5	47	60	66	0	–	100
6	40	49	54	0	0	–

TAB. 11.6 – *Nombre de problèmes dont la recherche est améliorée par la méthode en ligne par rapport à la méthode en colonne pour les problèmes facilités par le coefficient* 1.1, 1.2, 1.3 *et* 1.7 *(de gauche à droite puis de haut en bas).*

- on constate que le gain apporté par notre proposition s'amenuise pour les problèmes que nous qualifions de tangents : ni trop faciles, ni trop difficiles. Par contre, pour les problèmes très faciles, le gain est encore significatif. On peut noter ici l'influence très forte de la position d'une solution dans l'arbre de recherche sur la recherche elle-même, sauf lorsqu'on utilise notre proposition ;
- les améliorations proposées pour le choix d'un membre de l'alternative pour une disjonction présentent un intérêt qui s'amenuise au fur et à mesure que le problème devient plus facile pour n'apporter aucune différence significative quel que soit la méthode pour le coefficient 1.7.

11.4 Conclusions

Les résultats que nous avons présentés ici permettent de tirer les premières conclusions suivantes :

- Notre approche semble une alternative intéressante au méthodes classiques basées sur le backtrack standard ou sur le backtrack intelligent car elle diminue de manière significative le nombre de backtracks nécessaires pour trouver une solution.

  Ces résultats peuvent ainsi *réhabiliter* les systèmes de maintien de déduction qui ne servent qu'à l'identification d'un point de retour lorsqu'ils sont utilisés dans les méthodes de *backtrack intelligent* alors qu'ils peuvent être utilisés aussi pour éviter le *thrashing*.

- Notre approche est beaucoup moins dépendante de la position de la première solution dans l'arbre de recherche que les autres approches qui balaient cet arbre de la gauche vers la droite. Cette relative indépendance est montrée par les variances des résultats nettement plus faibles que pour les approches basées sur le backtrack.

  Ceci montre un des intérêts majeurs de l'approche par saut qui tout en restant complète permet de s'affranchir en partie de la structure utilisée *a priori* pour représenter l'arbre de recherche.

- Notre approche comparée à des systèmes commerciaux devient vite compétitive lorsque la combinatoire est grande et que les problèmes considérés sont composés de sous-problèmes peu interconnectés. Nous avons mis en évidence ce résultat en appliquant la méthode utilisée sur l'algorithme MAC dans le chapitre précédent : la résolution en parallèle de plusieurs instances d'un même problème. Ici, nous avons utilisé le problème du pont [Van Hentenryck, 1989] dont nous avons modélisé les disjonctions à l'aide d'un point de choix avec les systèmes CHIP et Ilog Solver (une contrainte `one-of` dans DECORUM). Les résultats obtenus montrent que dès trois problèmes imbriqués DECORUM est meilleur que Solver et CHIP même lorsque les problèmes ne sont pas complètement déconnectés.

L'utilisation de la contrainte `one-of` avec le système DECORUM présente donc de nombreuses pistes de travail futur sur l'amélioration de recherche arborescente et sur l'exploitation de l'indépendance face à la position de la solution dans l'arbre de recherche.

Quatrième partie

Conclusion

Chapitre 12

Conclusion et perspectives

Dans ce mémoire, nous sommes intéressés à la définition d'un système de relaxation de contraintes permettant de maintenir une propriété donnée dans un environnement dynamique. Nous avons mené ces travaux depuis une présentation abstraite d'un tel système jusqu'à son implémentation.

Plus précisément, nous avons présenté un schéma algorithmique général abstrait de la recherche d'une solution à un problème sur-contraint basée sur l'exploration en meilleur d'abord d'un espace de configurations.

Nous avons présenté trois instances de ce schéma algorithmique : une instance directe pour traiter les systèmes de contraintes linéaires sur les rationnels, une instance traitant les systèmes de contraintes sur les domaines finis (CSP) et une instance traitant les systèmes de contraintes résolus par propagation d'intervalles.

Pour les deux dernières qui fonctionnent à l'aide d'un algorithme de point fixe par réduction de domaine, nous avons proposé l'utilisation d'un système de maintien de déduction dont la maîtrise raisonnée nous a permis de donner une implémentation de ces instances ayant une *bonne* complexité : le système DECORUM. Il s'agit en quelque sorte d'une réponse à une des conclusions du projet ESPRIT CHIC rappelée à la fin du chapitre 3.

Nous avons aussi réalisé un certain nombre d'expérimentations sur des problèmes aléatoires permettant de valider notre approche. Nous avons ainsi montré que l'utilisation de DECORUM pour résoudre des problèmes quelconques (en particulier sur-contraints) permettait de retrouver les résultats classiques sur la transition de phase. D'autre part, nous avons montré la stabilité de notre approche en faisant varier la propriété maintenue (entre le maintien de solution et le maintien d'arc-consistance). Des expérimentations conduites sur des instances aléatoires de grandes tailles ont permis de confirmer l'utilisabilité pratique de DECORUM. Enfin, nous avons montré que DECORUM, par son utilisation d'informations retirées du problème traité, permettait d'obtenir de meilleurs résultats que l'algorithme MAC sur certains types de problèmes.

Enfin, nous avons proposé la contrainte `one-of` permettant de modéliser une disjonction. Cette contrainte tire profit du mécanisme d'exploration de DECORUM et permet, d'une part, d'utiliser activement les contraintes provenant d'une disjonction avant leur arbitrage définitif, et d'autre part, d'intéger simplement à DECORUM le mécanisme de l'énumération dans un CSP. Nous avons validé l'intérêt de la contrainte `one-of` sur des problèmes d'ordonnancement : les Open-Shop.

Nous pensons que ces travaux ouvrent la voie vers de nombreuses applications et extensions possibles. Nous proposons ici quelques pistes prometteuses suivant quatres axes :

– les diverses extensions de notre approche ;

- les applications potentielles du système DECORUM ;
- les applications potentielles de la contrainte `one-of` ;
- l'interaction avec l'utilisateur.

12.1 Extensions de notre approche

À court terme, il nous paraît intéressant d'envisager l'implémentation de notre approche dans :

- le cadre relax($\mathbb{Q}$, Lin, $\mathcal{P}_{sol}$) qui traite des contraintes linéaires sur les rationnels. Nous avons présenté chapitre 5 les résultats fondamentaux nécessaires à une telle instanciation ;
- le cadre relax(Interval,$\mathcal{C}$,$\mathcal{P}_{bc}$) qui traite des contraintes numériques sur les intervalles de flottants. Ici, le travail est plus avancé puisque cette instance, comme nous l'avons montré chapitres 6 et 8, profite pleinement du système de maintien de déduction utilisé dans le cadre relax(FD,$\mathcal{C}$,$\mathcal{P}_{ac}$) présenté chapitre 7 et base du système DECORUM.

Ces implémentations ouvriraient alors la voie, à plus long terme, vers une coopération non seulement entre les différents solveurs comme le font par exemple Béringer et de Backer [1995] mais aussi entre les différents mécanismes d'explications. On pourrait alors fournir un véritable système générique de relaxation de contraintes.

D'autre part, pour résoudre des problèmes réels, il nous semble indispensable d'intégrer au solveur de contraintes utilisé une plus grande déclarativité au sein des contraintes. Ainsi, le concept de contrainte globale [Beldiceanu et Contejean, 1994] nous semble parfaitement illustrer cette possibilité. En effet, une contrainte globale permet d'exprimer et surtout d'utiliser une connaissance de haut niveau sur les variables d'un problème: comme, par exemple, la contrainte `cumulative` qui permet d'exprimer des notions complexes d'ordonnancement avec ressources [Aggoun et Beldiceanu, 1993].

L'intégration de nouvelles contraintes dans DECORUM se fait en deux étapes : la définition d'un mécanisme de filtrage adapté à cette contrainte puis la fourniture d'explications précises (à défaut d'être minimales) pour ce filtrage afin de l'intégrer à notre système de maintien de déduction.

La tendance actuelle dans le domaine des contraintes globales est à l'utilisation de résultats obtenus en Recherche Opérationnelle pour des problèmes spécifiques. Ainsi, l'utilisation des résultats de Carlier et Pinson [1994] sur le Job-Shop a conduit à la proposition de mécanismes puissants de traitement des contraintes disjonctives [Caseau et Laburthe, 1996]. Le but est alors d'utiliser ces résultats pour fournir des explications intéressantes pour notre système.

Nous avons commencé à intégrer à DECORUM diverses contraintes dites globales : la contrainte `all-different` telle que proposée par [Régin, 1994] et sa généralisation `global cardinality constraint` [Régin, 1996]. De plus, nous avons aussi intégré les travaux de Carlier et Pinson [Carlier et Pinson, 1994] pour définir une contrainte de disjonction sur plusieurs variables.

12.2 Applications de DECORUM

DECORUM est un système générique de résolution de problèmes dynamiques. Ses domaines privilégiés d'application nous semblent être la planification et l'or-

donancement dynamiques ou de manière plus générale tout problème d'allocation dynamique de ressources.

Une première application [Fraszko, 1997] a été réalisée pour un problème de planification dynamique dans le domaine du militaire.

Le domaine des réseaux et des télécommunications est lui aussi riche en problèmes d'allocation dynamique de ressources. Une thèse en collaboration avec le CNET Lannion débutera en octobre 1997. Ce travail étudiera les extensions de DECORUM pour la résolution de problèmes de réservation et d'allocation de ressources dans les réseaux.

Enfin un troisième axe d'application de DECORUM qu'il nous paraîtrait intéressant d'explorer est la classe des problèmes d'ordonnancement dynamiques que l'on trouve en particulier dans le domaine du temps-réel.

12.3 Applications de la contrainte `one-of`

Les idées développées dans ce mémoire ne sont pas spécifiques à la seule relaxation de contraintes. En effet, comme nous l'avons vu, notre approche permet de généraliser l'algorithme *Dynamic Backtracking* [Ginsberg, 1993] qui est une méthode d'énumération améliorant le backtrack standard et le backtrack intelligent. L'idée fondamentale est, en cas de contradiction, non seulement de remettre en cause un point de choix plus pertinent que le plus récent, mais aussi de ne pas défaire, à cause de cette remise en cause, toute la partie du calcul réalisé qui est indépendante du choix reconsidéré.

Cette idée très générale dont nous avons montré la réalisabilité dans le chapitre 11 sur le traitement de la disjonction, peut être utilisée, par exemple, pour :

- Améliorer les recherches arborescentes usuelles (par exemple, en Recherche Opérationnelle) pour ne plus utiliser simplement un backtrack standard. Il s'agit en quelque sorte de la démarche inverse de la proposition de la section précédente : il s'agit ici d'adapter notre approche à une utilisation dans un cadre classique.

  Ainsi, Christelle Guéret, en intégrant une partie de notre approche dans ses travaux sur l'Open-Shop a résolu pour la première fois un problème ouvert dans ce domaine [Guéret et Jussien, 1997].

- Améliorer le traitement de la disjonction. Nous l'avons abordé dans le chapitre 10. Nous avons, en particulier, montré l'intérêt de l'utilisation du concept de contrainte `one-of` pour gérer la disjonction. Cette voie est à explorer, en particulier pour étudier le comportement de la contrainte `one-of` sur d'autres types de problèmes ...

12.4 Interactivité

Nous pensons aussi qu'il existe de nombreuses voies d'amélioration et d'utilisation de nos idées dans le cadre d'une résolution de problèmes vraiment interactive.

On peut ainsi citer :

- L'utilisation de notre système de maintien de déduction dans pour faire du *debugging* d'application par contraintes. Ainsi, lorsqu'un système de contraintes ne présente pas de solution, notre système est capable de présenter un sous-ensemble de ces contraintes qui *explique* l'absence d'une solution. On peut alors restreindre la recherche d'une éventuelle *erreur* de modélisation à ce

sous-ensemble. Dans cette optique, nous avons présenté le système DECORUM lors d'un séminaire de l'Association Française de Programmation en Logique (AFPL) en Janvier 1997 sur les environnements de programmation par contraintes.

- La définition de mécanisme de *traduction* des explications pour qu'elles soient compréhensibles par l'utilisateur final. En effet, bien souvent, une contrainte exprimée au niveau de l'utilisateur par une expression simple, est représentée dans le système de contraintes par un ensemble (*agrégat* potentiellement conséquent) de contraintes de bases. Dans ce cas, les explications fournies sont exprimées en fonction de ces contraintes de bases et ne sont pas très explicites pour l'utilisateur.

Références bibliographiques

[Aggoun et Beldiceanu, 1993] cité pp. 142, 154
Abderrahmane Aggoun et Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.

[Beldiceanu et Contejean, 1994] cité page 154
Nicolas Beldiceanu et E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.

[Benhamou *et al.*, 1994] cité page 18
Frédéric Benhamou, David McAllester, et Pascal Van Hentenryck. Clp(intervals) revisited. Rapport Technique CS-94-18, Department of Computer Science, Brown University, Avril 1994.

[Benhamou et Older, 1992] cité pp. 64, 65
Frédéric Benhamou et William Older. Applying interval arithmetic to integer and boolean constraints. Rapport technique, Bell Northern Research, Septembre 1992.

[Benhamou, 1995] cité page 83
Frédéric Benhamou. Interval Constraint Logic Programming. In Andreas Podelski, éditeur, *Constraint Programming: Basics and Trends*, LNCS 910. Springer, 1995. (Châtillon-sur-Seine Spring School, France, May 1994).

[Béringer et de Backer, 1990] cité pp. 28, 61
Henri Béringer et Bruno de Backer. Piecewise linear constraints under assumptions a new approach to model based diagnosis. In *ITESM : Third international symposium on artificial intelligence*, Monterrey, Mexico, 1990.

[Béringer et de Backer, 1995] cité page 154
Henri Béringer et Bruno de Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. In North Holland, éditeur, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*, 1995.

[Bessière et Cordier, 1993] cité page 27
Christian Bessière et Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *AAAI-93: Proceedings 11th National Conference on Artificial Intelligence*, Washington, DC, Juillet 1993. American Association for Artificial Intelligence.

[Bessière et Régin, 1996] cité page 126
Christian Bessière et Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problem. In *CP'96*, Cambridge, MA, 1996.

[Bessière, 1991] cité pp. 8, 27, 77
Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.

[Bessière, 1994] cité page 27
Christian Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[Bistarelli *et al.*, 1995] cité page 32
Stefano Bistarelli, Ugo Montanari, et Francesca Rossi. Constraint solving over semirings. In Chris Mellish, éditeur, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, Août 1995.

[Bistarelli *et al.*, 1996] cité page 32
Stefano Bistarelli, Hélène Fargier, Ugo Montanari, Francesca Rossi, Thomas Schiex, et Gérard Verfaillie. Semiring-based csps and valued csps: basic properties and comparison. In Michael Jampel, Eugene Freuder, et Michael Maher, éditeurs, *Over-Constrained Systems*, numéro 1106 in Lecture Notes in Computer Science, pages 111–150. Springer Verlag, 1996.

[Boizumault *et al.*, 1994] cité page 8
Patrice Boizumault, Christelle Guéret, et Narendra Jussien. Efficient labeling and constraint relaxation for solving time tabling problems. In Pierre Lim et Jean Jourdan, éditeurs, *Proceedings of the 1994 ILPS post-conference workshop on Constraint Languages/Systems and their use in Problem Modeling : Volume 1 (Applications and Modelling)*, Technical Report ECRC-94-38, ECRC, Munich, Germany, Novembre 1994.

[Boizumault *et al.*, 1995] cité page 142
Patrice Boizumault, Yann Delon, Christelle Guéret, et Laurent Péridy. Résolution de problèmes en programmation logique avec contraintes. *Revue d'Intelligence Artificielle*, 9(3):383–406, 1995.

[Boizumault *et al.*, 1996] cité page 142
Patrice Boizumault, Yann Delon, et Laurent Péridy. Constraint logic programming for examination timetabling. *Special Issue of the Journal of Logic Programming: Applications of Logic Programming*, 26(2):217–233, 1996.

[Borning *et al.*, 1989] cité page 30
Alan Borning, Michael Maher, Amy Martindale, et Molly Wilson. Constraint hierarchies and logic programming. In Giorgio Levi et Maurizio Martelli, éditeurs, *ICLP'89: Proceedings 6th International Conference on Logic Programming*, pages 149–164, Lisbon, Portugal, Juin 1989. MIT Press.

[Borning *et al.*, 1996] cité pp. 34, 112
Alan Borning, Richard Anderson, et Bjorn Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *ACM Symposium on User Interface Software and Technology*, pages 129–136, 1996.

[Bouquet et Jégou, 1996] cité page 32
Fabrice Bouquet et Philippe Jégou. Solving over-constrained CSP using weighted OBDD. In Michael Jampel, Eugene Freuder, et Michael Maher, éditeurs, *Over-Constrained Systems*, numéro 1106 in Lecture Notes in Computer Science, pages 151–170. Springer Verlag, 1996.

[Bouzoubaa *et al.*, 1995] cité page 34
Mouhssine Bouzoubaa, Bertrand Neveu, et Gier Hasle. Houria: A solver for equational constraints in a hierarchical system. In Michael Jampel, Eugene Freuder,

et Michael Maher, éditeurs, *OCS'95: Workshop on Over-Constrained Systems at CP'95*, Cassis, Marseilles, 18 Septembre 1995.

[Brucker *et al.*, 1994] cité page 144
P. Brucker, J. Hurink, B. Jurisch, et B. Westmann. A branch and bound algorithm for the open-shop problem. Rapport technique, Osnabrueck University, 12 1994.

[Caprara *et al.*, 1996] cité page 53
Alberto Caprara, Matteo Fischetti, et Paolo Toth. A heuristic algorithm for the set covering problem. In *Integer Programming and Combinatorial Optimization*, Vancouver, June 1996.

[Carlier et Pinson, 1989] cité page 145
Jacques Carlier et Éric Pinson. An algorithm for solving the job shop problem. *Management Science*, 35(2):164–176, 1989.

[Carlier et Pinson, 1990] cité page 120
Jacques Carlier et Éric Pinson. A practical use of jackson's preemptive schedule for solving the job shop problem. *Annals of Operations Research*, 26:269–287, 1990.

[Carlier et Pinson, 1994] cité pp. 120, 146, 154
Jacques Carlier et Éric Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[Caseau et Laburthe, 1996] cité pp. 119, 154
Yves Caseau et François Laburthe. Cumulative scheduling with task intervals. In *Joint International Conference and Symposium on Logic Programming*. MIT Press, 1996.

[Chamard *et al.*, 1995] cité page 38
André Chamard, Annie Fischler, Benoît Guinaudeau, et André Guillaud. CHIC lessons on CLP methodology. Rapport Technique ESPRIT EP5291 / D2.1.2.3, CHIC Project / Dassault Aviation / Bull, Janvier 1995. (Constraint Handling in Industry and Commerce).

[Chinneck et Dravnieks, 1991] cité page 28
John W. Chinneck et Erik W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal on Computing*, 3(2):157–168, Spring 1991.

[Colmerauer, 1990] cité pp. 7, 18
Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, Juillet 1990.

[Cousin, 1991] cité page 96
Xavier Cousin. Meilleures solutions en programmation logique. In *Proceedings Avignon'91*, 1991.

[David, 1994] cité pp. 17, 34
Philippe David. *Prise en compte de la sémantique dans les problèmes de satisfaction de contraintes: étude des contraintes fonctionnelles.* Thèse de doctorat, Université de Montpellier II – Sciences et Techniques du Languedoc, Février 1994.

[de Backer et Béringer, 1991a] cité page 31
Bruno de Backer et Henri Béringer. An efficient solver for HCLP($\mathcal{R}$). Rapport technique, IBM France Scientific Center, 1991. (ICLP'91, Pre-Conference Workshop on CLP).

[de Backer et Béringer, 1991b] cité pp. 28, 60
Bruno de Backer et Henri Béringer. Intelligent backtracking for CLP languages: An application to CLP($\mathcal{R}$). In Vijay Saraswat et Kazunori Ueda, éditeurs, *ILPS'91: Proceedings International Logic Programming Symposium*, pages 405–419, San Diego, CA, Octobre 1991. MIT Press.

[de Backer, 1995] cité pp. 58, 60, 61, 142, 143
Bruno de Backer. *Méthodes de résolution de disjonctions de contraintes linéaires. Application à la Programmation Logique avec Contraintes.* Thèse de doctorat, Université d'Orléans, France, 17 Mars 1995.

[de Kleer, 1986a] cité pp. 26, 64
Johan de Kleer. An assumption-based tms. *Artificial Intelligence*, 28:127–162, 1986.

[de Kleer, 1986b] cité page 26
Johan de Kleer. Extending the ATMS. *Artificial Intelligence*, 28:163–196, 1986.

[de Kleer, 1986c] cité page 26
Johan de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28:197–224, 1986.

[Debruyne, 1995] cité pp. 8, 27, 28, 77
Romuald Debruyne. Les algorithmes d'arc-consistance dans les CSP dynamiques. *Revue d'Intelligence Artificielle*, 9(3), 1995.

[Dechter et Dechter, 1988] cité pp. 7, 19
R. Dechter et A. Dechter. Belief maintenance in dynamic constraint networks. In *AAAI'88*, pages 37–42, St Paul, MN, 1988.

[Diaz, 1995] cité page 119
Daniel Diaz. *Étude de la compilation des langages logiques de programmation par contraintes sur les domaines finis: le système* `clp(FD)`. Thèse de doctorat, Université d'Orléans, France, Janvier 1995.

[Dincbas *et al.*, 1988] cité pp. 7, 142
Mehmet Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, et F. Berthier. The Constraint Logic Programming Language CHIP. In *FGCS-88: Proceedings International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, Décembre 1988. ICOT.

[DMI, 1996] cité pp. 114, 119
DMI-ENS – Département Mathématiques et Informatique – École Normale Supérieure, Paris, France. *Introduction to the CLAIRE programming language*, 1996.

[Doyle, 1979] cité pp. 26, 64
J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Fages *et al.*, 1994] cité page 32
François Fages, Julian Fowler, et Thierry Sola. Handling preferences in constraint logic programming with relational optimization. In *PLILP'94*, Madrid, Septembre 1994.

[Fages *et al.*, 1995] cité pp. 7, 19, 28, 33, 48
François Fages, Julian Fowler, et Thierry Sola. A reactive constraint logic programming scheme. In *International Conference of Logic Programming, ICLP'95*, Tokyo, 1995.

[Fages *et al.*, 1998] cité page 28
François Fages, Julian Fowler, et Thierry Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, Septembre 1998.

[Fages, 1993] cité page 32
François Fages. On the semantics of optimization predicates in CLP languages. In *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Newport, Rhode Island, Avril 1993.

[Forbus et de Kleer, 1993] cité page 26
Kenneth D. Forbus et Johan de Kleer. *Building Problem Solvers*. MIT Press, Cambridge, 1993.

[Fowler, 1993] cité pp. 8, 32
Julian Fowler. Preferred constraints as optimization. In *Proceedings Journées Françaises de Programmation en Logique*, 1993.

[Fraszko, 1997] cité pp. 122, 155
David Fraszko. DECORUM. Mémoire de DEA, École des Mines de Nantes, 1997.

[Freeman-Benson *et al.*, 1990] cité pp. 18, 34
Bjorn Freeman-Benson, John Maloney, et Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, Janvier 1990.

[Freuder, 1978] cité page 17
Eugene Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21:958–966, Novembre 1978.

[Freuder, 1989] cité pp. 8, 30, 31
Eugene Freuder. Partial constraint satisfaction. In *IJCAI-89: Proceedings 11th International Joint Conference on Artificial Intelligence*, pages 278–283, Detroit, 1989.

[Gallesio, 1996] cité page 114
Érick Gallesio. Un langage objet pour la construction d'interfaces graphiques. In *Journée plénière*, Orléans, France, Novembre 1996. GDR Programmation du CNRS.

[Ginsberg, 1993] cité pp. 34, 35, 48, 64, 92, 155
Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[Gondran et Minoux, 1990] cité pp. 45, 53
Michel Gondran et Michel Minoux. *Graphes et Algorithmes*. Direction des Études et Recherche d'Électricité de France, Éditions Eyrolles, 1990.

[Gonzales et Sahni, 1976] cité page 144
Teofilo Gonzales et Sartaj Sahni. Open shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery*, 23(4):665–679, 10 1976.

[Guéret *et al.*, 1996] cité page 8
Christelle Guéret, Narendra Jussien, Patrice Boizumault, et Christian Prins. Building university timetables using Constraint Logic Programming. In Edmund Burke et Peter Ross, éditeurs, *The Practice and Theory of Automated Timetabling*, volume 1153 of *Lecture Notes in Computer Science*, pages 130–145. Springer-Verlag, 1996.

[Guéret et Jussien, 1997] cité pp. 11, 155
Christelle Guéret et Narendra Jussien. Résolution de l'open-shop avec une recherche arborescente améliorée. Rapport Technique 97-x-AUTO, École des Mines de Nantes, 1997.

[Guéret et Prins, 1998] cité page 144
Christelle Guéret et Christian Prins. Classical and new heuristics for the open-shop problem. *European Journal of Operations Research*, 1998. À paraître.

[Harel et Pnueli, 1985] cité pp. 7, 19
D. Harel et A. Pnueli. On the development of reactive systems. In K. R. Apt, éditeur, *Logics and Models of Concurrent Systems*, volume F13 of *NATO ASI Series*. Springer Verlag, Janvier 1985.

[Haselböck *et al.*, 1993] cité page 34
Alois Haselböck, Thomas Havelka, et Markus Stumptner. Revising inconsistent variable assignments in constraint satisfaction problems. In Manfred Meyer, éditeur, *Constraint Processing: Proceedings of the International Workshop at CSAM'93, St. Petersburg, July 1993*, Research Report RR-93-39, pages 113–122, DFKI Kaiserslautern, Août 1993.

[Holzbaur *et al.*, 1996] cité pp. 28, 33, 62
Christian Holzbaur, Francisco Menezes, et Pedro Barahona. Defeasibility in clp(q) throught generalized slack variables. In Eugene C. Freuder, éditeur, *Principles and Practice of Constraint Programming – CP 96*, numéro 1118 in Lecture Notes in Computer Science, pages 209–223, Cambridge, MA, USA, Août 1996.

[Jaakola, 1990] cité pp. 58, 62
Juhani Jaakola. Modifying the simplex algorithm to a constraint solver. In *PLILP'90*, pages 89–105, 1990.

[Jampel *et al.*, 1996] cité pp. 8, 32
Michael Jampel, Eugene Freuder, et Michael Maher, éditeurs. *Over-Constrained Systems*, numéro 1106 in Lecture Notes in Computer Science. Springer Verlag, 1996.

[Jampel, 1996] cité page 30
Michael Jampel. A brief overview of over-constrained systems. In Michael Jampel, Eugene Freuder, et Michael Maher, éditeurs, *Over-Constrained Systems*, numéro 1106 in Lecture Notes in Computer Science, pages 1–22. Springer-Verlag, 1996.

[Jourdan et Sola, 1993] cité page 142
Jean Jourdan et Thierry Sola. The versatility of handling disjunctions as constraints. In *PLILP'93*, 1993.

[Jussien et Boizumault, 1996a] cité page 64
Narendra Jussien et Patrice Boizumault. CSP dynamiques et Relaxation de Contraintes. In TEKNEA, éditeur, *CNPC'96: deuxième Conférence Nationale sur la résolution pratique de Problèmes NP-Complets*, pages 135–149, Dijon, France, Mars 1996. CRID – Université de Bourgogne, Dijon.

[Jussien et Boizumault, 1996b] cité page 64
Narendra Jussien et Patrice Boizumault. Implementing constraint relaxation over finite domains using atms. In Michael Jampel, Eugene Freuder, et Michael Maher, éditeurs, *Over-Constrained Systems*, numéro 1106 in Lecture Notes in Computer Science, pages 265–280. Springer-Verlag, 1996.

[Jussien et Boizumault, 1996c] cité pp. 64, 114
Narendra Jussien et Patrice Boizumault. Maintien de déduction pour la relaxation de contraintes. In *Journées Francophones de Progammation en Logique et avec Contraintes*, Clermont-Ferrand, Juin 1996.

[Jussien et Boizumault, 1997a] cité page 42
Narendra Jussien et Patrice Boizumault. A best first approach for solving over-constrained dynamic problems. In *IJCAI'97*, Nagoya, Japan, August 1997. (poster – RR 97-6-INFO – École des Mines de Nantes).

[Jussien et Boizumault, 1997b] cité page 42
Narendra Jussien et Patrice Boizumault. Best-first search for property maintenance in reactive constraints systems. In *International Logic Programming Symposium*, Port Jefferson, N.Y., oct 1997. MIT Press.

[Jussien et Boizumault, 1997c] cité page 48
Narendra Jussien et Patrice Boizumault. Stratégies en meilleur d'abord pour la relaxation de contraintes. In *Journées Francophones de Progammation en Logique et avec Contraintes*, Orléans, Mai 1997.

[Lassez, 1990] cité page 60
Jean-Louis Lassez. Parametric queries, linear constraints and variable elimination. In *DISCO'90: Proceedings Conference on Design and Implementation of Symbolic Computation Systems*, Capri (???), 1990.

[Lhomme *et al.*, 1996] cité page 107
Olivier Lhomme, Arnaud Gotlieb, Michel Rueher, et Patrick Taillibert. Boosting the interval narrowing algorithm. In *JICSLP'96*, Bonn, Germany, 2–6 Septembre 1996.

[Lhomme et Rueher, 1996] cité pp. 106, 107
Olivier Lhomme et Michel Rueher. Application des techniques csp au raisonnement sur les intervalles. *Revue d'Intelligence Artificielle*, 1996.

[Lhomme, 1993] cité pp. 17, 18, 64, 107
Olivier Lhomme. Consistency techniques for numeric CSPs. In *IJCAI'93*, pages 232–238, Chambery, France, August 1993.

[Lhomme, 1994] cité pp. 11, 106, 107, 108, 109, 111
Olivier Lhomme. *Contribution à la résolution de contraintes sur les réels par propagation d'intervalles.* Thèse de doctorat, Université de Nice – Sophia Antipolis, 1994.

[Menezes *et al.*, 1993] cité pp. 33, 62, 79, 80
Francisco Menezes, Pedro Barahona, et Philippe Codognet. An incremental hierarchical constraint solver. In Paris Kanellakis, Jean-Louis Lassez, et Vijay Saraswat, éditeurs, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993.

[Menezes et Barahona, 1996] cité pp. 33, 44
Francisco Menezes et Pedro Barahona. Defeasible constraint solving. In Michael Jampel, Eugene Freuder, et Michael Maher, éditeurs, *Over-Constrained Systems*, numéro 1106 in Lecture Notes in Computer Science, pages 151–170. Springer Verlag, 1996.

[Minoux, 1983] cité pp. 53, 60
Michel Minoux. *Programmation Mathématique, Théorie et Algorithmes.* Dunod, 1983.

[Mohr et Henderson, 1986] cité pp. 27, 135
R. Mohr et T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Neveu et Berlandier, 1994] cité page 28
Bertrand Neveu et Pierre Berlandier. Arc-consistency for dynamic constraint satisfaction problems: an RMS free approach. In *Proc. ECAI-94, Workshop on Constraint satisfaction issues raised by practical applications*, Amsterdam, The Netherlands, 1994.

[Prosser, 1993] cité pp. 28, 34
Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, Août 1993. (Also available as Technical Report AISL-46-91, Stratchclyde, 1991).

[Prosser, 1994a] cité page 126
Patrick Prosser. Binary constraint satisfaction problems: Some are harder than others. In *ECAI'94*, pages 95–99, 1994.

[Prosser, 1994b] cité pp. 11, 126
Patrick Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. Rapport Technique AISL-49-94, Department of Computer Science, University of Strathclyde, Glasgow, Scotland, 1994.

[Puget, 1994] cité page 7
Jean-Francois Puget. A c++ implementation of clp. In *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore, 1994.

[Régin, 1994] cité pp. 120, 154
Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI 94, Twelth National Conference on Artificial Intelligence*, pages 362–367, Seatle, Washington, 1994.

[Régin, 1995] cité page 120
Jean-Charles Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.* Thèse de doctorat, Université de Montpellier II, 21 Décembre 1995.

[Régin, 1996] cité page 154
Jean-Charles Régin. Generalized arc-consistency for global cardinality constraint. In *AAAI'96*, 1996.

[Roos, 1994] cité page 27
Nico Roos. Improving backtracking to solve an area allocation problem. In Thomas Schiex et Christian Bessiére, éditeurs, *Proceedings ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications*, Amsterdam, Août 1994.

[Sabin et Freuder, 1994] cité pp. 11, 37, 126, 135
Daniel Sabin et Eugene Freuder. Contradicting conventional wisdom in constraint satisfaction. In Alan Borning, éditeur, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer, Mai 1994. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).

[Sannella, 1993] cité pp. 8, 34
Michael Sannella. The SkyBlue constraint solver and its applications. In Paris Kanellakis, Jean-Louis Lassez, et Vijay Saraswat, éditeurs, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993.

[Schiex et Verfaillie, 1993] cité page 34
Thomas Schiex et Gérard Verfaillie. Nogood recording for static and dynamic CSP. In 5^{th} *IEEE International Conference on Tools with Artificial Intelligence*, pages 48–55, Boston, MA., 1993.

[Schiex et Verfaillie, 1994] cité pp. 8, 34
Thomas Schiex et Gérard Verfaillie. Nogood Recording fot Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[Schiex et Verfaillie, 1995] cité page 32
Thomas Schiex et Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In Chris Mellish, éditeur, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, Août 1995.

[Schiex, 1992] cité pp. 22, 23, 32
Thomas Schiex. Possibilistic constraint satisfaction problems or "How to handle soft constraints?". In *8th International Conference on Uncertainty in Artificial Intelligence*, Stanford, Juillet 1992.

[Smith et Grant, 1995] cité page 126
Barbara Smith et Stuart Grant. Sparse constraint graphs and exceptionally hard problems. In Chris Mellish, éditeur, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, Août 1995.

[Smith, 1994] cité page 126
Barbara Smith. The phase transition in constraint satisfaction problems: A closer look at the mushy region. In *Proceedings ECAI'94*, 1994. (Also Leeds University Research Report 93.41).

[Sola, 1995] cité pp. 28, 29, 33, 79, 88, 143
Thierry Sola. *Modèles d'exécution de la Programmation Logique basés sur le graphe de dépendances des liaisons, Backtracking Intelligent, Maintien des déductions et Contraintes.* Thèse de doctorat, Université Paris XI, Décembre 1995.

[Trombettoni, 1997] cité page 34
Gilles Trombettoni. *Algorithmes de maintien de solution par propagation local pour les systèmes de contraintes.* Thèse de doctorat, Université de Nice – Sophia Antipolis, 1997.

[Tsang, 1993] cité pp. 17, 64
Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[Van Hentenryck, 1989] cité page 27
Pascal Van Hentenryck. Incremental constraint satisfaction in logic programming. In *Proceedings* 6^{th} *International Conference on Logic Programming*, 1989.

[Van Hentenryck *et al.*, 1992] cité pp. 65, 83, 84, 85
Pascal Van Hentenryck, Yves Deville, et Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, Octobre 1992.

[Van Hentenryck et Deville, 1991] cité page 142
Pascal Van Hentenryck et Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In Koichi Furukawa, éditeur, *ICLP'91 Proceedings 8th International Conference on Logic Programming*, pages 745–759. MIT Press, 1991.

[Van Hentenryck, 1989] cité page 150
Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. MIT Press, Cambridge, MA, 1989.

[Varnier *et al.*, 1993] cité page 142
C. Varnier, P. Baptiste, et B. Legeard. Le traitement des contraintes disjonctives dans un problème d'ordonnancement : le hoist scheduling problem. In *Proceedings Journées Françaises de Programmation en Logique*, 1993.

[Verfaillie et Schiex, 1994] cité page 8
Gérard Verfaillie et Thomas Schiex. Solution reuse in dynamic constraint satisfaction problems. In *AAAI'94*, pages 307–312, Seattle, WA, 1994.

[Verfaillie et Schiex, 1995] cité pp. 34, 36, 44
Gérard Verfaillie et Thomas Schiex. Maintien de solution dans les problèmes dynamiques de satisfaction de contraintes : bilan de quelques approches. *Revue d'Intelligence Artificielle*, 9(3), 1995.

[Wallace et Freuder, 1996] cité page 32
Richard Wallace et Eugene C. Freuder. Heuristics methods for over-constrained constraint satisfaction problems. In Michael Jampel, Eugene Freuder, et Michael Maher, éditeurs, *Over-Constrained Systems*, numéro 1106 in Lecture Notes in Computer Science, pages 207–216. Springer Verlag, 1996.

[Wallace, 1996] cité page 32
Richard Wallace. Cascaded arc consistency preprocessing as a strategy for maximal constraint satisfaction. In *Over-Constrained Systems*, numéro 1106 in Lecture Notes in Computer Science, pages 217–228. Springer Verlag, 1996.

[Williamson *et al.*, 1992] cité page 148
D. Williamson, L. Hall, J. Hoogeveen, C. Hurkens, J. Lenstra, et D. Shmoys. Short shop schedules. Rapport technique, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1992.

[Wilson et Borning, 1993] cité pp. 8, 20, 21, 22, 23, 30, 31
Molly Wilson et Alan Borning. Hierarchical constraint logic programming. *Journal of Logic Programming*, 16(3):277–318, Juillet 1993.

Liste des définitions

Liste des théorèmes

Liste des algorithmes

Liste des exemples

Table des figures

Liste des tableaux

Index

Dans cet index, une référence en **gras** indique une *définition* du terme référencé et une référence en *italique* indique une *utilisation* du terme référencé.

– K –

– M –

– O –

– P –

– R –

– S –

– T –