UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DES MATÉRIAUX

Année 2003

N° attribué par la bibliothèque

Résolution de problèmes combinatoires modélisés par des contraintes quantifiées

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE NANTES

Discipline : INFORMATIQUE

présentée et soutenue publiquement par

Lucas BORDEAUX

le 19 septembre 2003

à l'UFR Sciences et Techniques, Université de Nantes

devant le jury ci-dessous

Président	:	M. Alain COLMERAUER	
Rapporteurs	:	M. Christian BESSIÈRE, chargé de recherche	LIRMM-CNRS
		M. François FAGES, directeur de recherche	INRIA Rocquencourt
Examinateurs	:	M. Frédéric BENHAMOU, professeur	Université de Nantes
		M. Alain COLMERAUER, professeur	Université de la Méditerranée, membre de l'Institut Universitaire de France
		M. Éric MONFROY, professeur	Université de Nantes
		M. Toby WALSH, SFI research professor	Cork Constraint Computation Center

Directeur de thèse : M. Frédéric BENHAMOU
Laboratoire : Institut de Recherche en Informatique de Nantes

N° ED 0366-108

RÉSOLUTION DE PROBLÈMES COMBINATOIRES MODÉLISÉS PAR DES CONTRAINTES QUANTIFIÉES

Resolution of combinatorial problems stated as quantified constraints

Lucas BORDEAUX

favet neptunus eunti

Université de Nantes

Lucas BORDEAUX

Résolution de problèmes combinatoires modélisés par des contraintes quantifiées

xiv+180 p.

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

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

Impression : my-thesis.tex – 23/9/2003 – 13:45

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

• ARTIFICIAL INTELLIGENCE

To my knowledge the only scientific area with an intrinsic conflict of interest: unlike medical researchers which are usually in good health, those AI guys badly need the stuff they are after.

— Jean-Yves GIRARD, Locus Solum [125].

Remerciements

En sortant, merci de laisser l'ambiance là où vous l'avez trouvée.

— Anonyme, proverbe écrit sur la porte des bars de Nantes et d'ailleurs.

C'est avec plaisir que j'adresse à MM Christian Bessière, Alain Colmerauer, François Fages et Toby Walsh mes plus sincères remerciements pour avoir évalué cette thèse et lui avoir apporté leurs questions, leurs commentaires et leurs critiques. Je suis honoré d'avoir bénéficié des conseils de scientifiques aussi compétents et reconnus. Qu'ils soient assurés de ma reconnaissance. Je remercie également mes encadrants, Frédéric Benhamou et Éric Monfroy, de m'avoir supporté (dans les acceptations francophone et anglophone du mot!) pendant toute cette période.

Je souhaite enfin exprimer toute ma gratitude envers Gérard Ferrand et les membres de l'équipe qu'il dirige à Orléans, dans laquelle j'ai effectué mon DEA et mon stage d'initiation à la recherche. J'ai eu le plaisir de recevoir les enseignements de Gérard depuis mes premiers pas en programmation en DEUG et je lui dois en grande partie mon intérêt pour l'informatique théorique et mon orientation vers la recherche ce qui, j'en suis conscient, est loin de constituer le plus grand service qu'il ait rendu aux sciences et à l'éducation.

Cette thèse a bénéficié de la relecture avisée de Martine Ceberio et de discussions avec (notamment) Arnaud Berny, Sebastian Brand, Evgueni Petrov et Stefan Ratschan. Les voyages que j'ai été amené à effectuer au cours de cette thèse ont été rendus plaisants par la compagnie de Martine, Frédéric Saubion, Franky Trichet, Marc Christie, et de nombreux collègues de la communauté *contraintes* venant de l'IRIN, de l'École des Mines et d'ailleurs. J'ai eu le plaisir de partager mon bureau avec Martine Ceberio, Mina Ouabiba, Arnaud Berny, Frédéric Goualard, Brice Pajot et Bruno Tixier.

De nombreuses autres personnes ont contribué à faire de ces années de thèse à Nantes une période agréable. Il serait trop long de tous les mentionner, mais on peut sans trop se tromper les désigner collectivement sous le nom de l'association de doctorants *Login*. Je souhaite aux thésards du futur LINA d'y trouver les conditions d'accueil que j'ai connues à l'IRIN, et de réussir à préserver l'ambiance qui y a régné entre doctorants pendant la période où j'ai eu la chance d'y séjourner.

Enfin, et conformément à la législation en vigueur, je tiens à préciser qu'aucun animal n'a été blessé lors de la réalisation de cette thèse, et que toute ressemblance avec des personnages existants serait fortuite. On pourra en revanche trouver ici ou là des ressemblances avec les contributions scientifiques d'autres auteurs, mais pas plus que dans les autres thèses. Le lecteur pourra d'ailleurs s'amuser au fil de la lecture à identifier les articles dont je me suis inspiré pour la rédaction de cette thèse. Les réponses sont indiquées à la fin du document.

Sommaire

Avant-propos

Ce document est basé sur les travaux de la fin de ma période de thèse. Les contributions qui y sont décrites ont fait l'objet de présentations partielles dans [35] et [33]. Certaines des recherches entamées au début de ma thèse ne sont pas mentionnées, notamment le contenu des articles [34] et [36]. La raison est que, partant d'un sujet consacré aux techniques de consistances fortes pour les contraintes continues, et après avoir proposé de travailler par ailleurs sur des techniques d'abstraction de contraintes, la dernière partie de ma thèse s'est finalement orientée vers un sujet qui me paraissait apporter des réponses potentielles à certaines des préoccupations récentes de la communauté : les contraintes quantifiées.

En accord avec mes encadrants, j'ai finalement décidé de consacrer mon mémoire à la partie de ces travaux pour laquelle mon intérêt était encore le plus vif, à savoir la dernière. Si ce choix limite quelque peu la quantité d'information exposée dans le document, il lui donne en revanche une ligne directrice plus claire, articulée autour d'un sujet principal unique. Enfin, il permet d'accorder d'avantage de temps et de place à un sujet bénéficiant d'un engouement récent, et dont la présentation préliminaire [33] était incomplète. J'espère donc que ce choix satisfera les (re)lecteurs de ce document. Les travaux évoqués précédemment et qui ne seront pas mentionnés dans la suite du document concernent les sujets suivants :

Consistances d'intervalles — Les techniques de propagation d'intervalles permettent la résolution efficace de contraintes non linéaires à domaines réels. Parmi ces techniques, les kB-consistances [179] forment une classe de consistances *fortes*, permettant d'obtenir un filtrage arbitrairement élevé si on augmente le paramètre k (ce qui, en contrepartie, augmente également le temps de propagation). La contribution présentée dans [34] porte sur la complexité de ce type de techniques. Nous y détaillons des améliorations techniques de l'algorithme naïf de filtrage par kB-consistance (certaines ayant par ailleurs déjà été suggérées dans les travaux d'Olivier LHOMME, inventeur de ce type de consistances). Notre contribution principale est d'introduire un algorithme dont la complexité de $O(md^2n)$ améliore la borne connue précédemment d'un facteur n (m représente le nombre de contraintes, n le nombre de variables, d la taille maximale des intervalles de la boîte). La consommation d'espace supplémentaire introduite par les structures de données de l'algorithme est acceptable.

Techniques d'abstraction et étude des propriétés des contraintes numériques — Après des travaux en début de thèse sur la propriété de *monotonie* des contraintes et des techniques de représentation abstraite des contraintes monotones, j'ai été amené à étudier les propriétés des contraintes numériques (notamment celles de monotonie et de convexité) et leur prise en compte dans les algorithmes de résolution de contraintes. De manière générale, les propriétés permettent de déterminer l'utilisation de résolveurs dédiés, ou peuvent être utilisées lors de la coopération entre différents résolveurs. L'article [36] propose un cadre dans lequel ces différentes propriétés sont vues comme des *abstractions* des contraintes numériques sous-jacentes. L'inférence de certaines propriétés est automatisée par un ensemble de règles de calcul simples (une notion de *typage* très similaire à cette notion d'abstraction a été développée indépendamment par [174] et appliquée aux contraintes discrètes).

PARTIE I

Introduction et contexte de la thèse

Cette partie présente les grandes lignes du contexte dans lequel la thèse s'inscrit, et sert de guide de lecture pour la suite du document.
Parmi les notions rencontrées au fil de la thèse, on trouvera des termes aussi divers que programmation par contraintes, complexité, combinatoire, théorie des graphes, calcul de plans d'action, vérification, logique de premier ordre, logique d'ordre supérieur, optimisation. *Un exposé complet de ces différentes notions étant hors de propos, et d'autres références écrites par des auteurs faisant autorité étant par ailleurs disponibles, nous avons choisi de présenter simplement un aperçu du contexte de la thèse et d'offrir une vision aussi globale que possible de notre problématique de recherche, des enjeux actuels qui la concernent et des connexions avec d'autres problématiques voisines. Cet exposé, présenté dans le chapitre 1, se veut introductif et accessible au public le plus large.*
Les prérequis plus directement liés aux techniques utilisées dans cette thèse seront développés dans la partie suivante. Ils concernent en particulier les algorithmes de déduction logique, de recherche de solutions et de propagation de contraintes.
Nous avons souhaité, enfin, donner un aperçu des contributions personnelles rapportées dans cette thèse, qui servira ainsi de guide de lecture du document (Chapitre 2).

CHAPITRE 1

Problèmes, langages, et méthodes de résolution

Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it.

La programmation par contraintes représente l'une des approches les plus proches du Saint Graal de la programmation : l'utilisateur exprime le problème, la machine le résout. [a]

[a] Mes traductions personnelles des propos cités dans cette thèse — et en particulier dans ce chapitre — pouvant être l'objet de contestations, je les accompagne systématiquement de l'extrait original.

— Eugene FREUDER, [111].

Ce chapitre présente le contexte de la thèse et donne un aperçu des problématiques mises en jeux, ainsi que des domaines d'application de notre travail. Peu technique, il se veut accessible au public le plus large.

Notre recherche s'inscrit dans une perspective d'utilisation de l'ordinateur pour la résolution de problèmes*. Les principales notions sont celles de* problème*, de* langage *pour exprimer un problème sous une forme exploitable par une machine, et de* méthode *permettant d'automatiser la résolution de certaines classes de problèmes. Nous présentons successivement ces trois notions.*

Un des obstacles majeurs à l'utilisation des techniques de résolution de problèmes est leur énorme coût en termes de temps de calcul. Nous donnons dans ce chapitre des précisions sur ce point essentiel.

1.1 Le contexte : résolution de problèmes assistée par ordinateur

Bien que la question *à quoi sert un ordinateur?* soit rarement posée de manière si explicite — et qu'elle trouve plus rarement encore de réponse pleinement satisfaisante —, on peut distinguer plusieurs domaines majeurs d'application de l'informatique. Les plus importants à l'heure actuelle en termes de retombées pratiques et économiques concernent certainement le stockage et l'exploitation de données et de documents (je regroupe sous cette catégorie aussi bien la gestion de bases de données que l'édition et la comptabilité) et la communication (commerce électronique, messagerie, réseaux). Un autre domaine d'application, moins connu du grand public mais devenu tout aussi indispensable à notre société, est l'utilisation de l'outil informatique pour l'*aide à la décision* et la *résolution de problèmes*.

De nombreuses situations issues de la vie quotidienne, de la logistique et de la prise de décision économique, ou de la recherche en automatisation du raisonnement, requièrent en effet de pouvoir trouver une *solution* satisfaisant un certain nombre de critères. On s'intéressera plus particulièrement aux problèmes pouvant être exprimés de manière *formelle*, par des moyens mathématiques ou logiques. Des exemples concrets sont par exemple l'affectation de voies de trafic aérien ou la configuration des composants d'un appareil électronique. Dans le premier cas, on peut par exemple souhaiter affecter pour chaque créneau horaire une zone de l'espace aérien à chacun des appareils d'une flotte en évitant d'autoriser deux appareils à circuler dans la même zone au même moment. Dans le second cas, le but est typiquement d'agencer un ensemble de composants électroniques dans un volume déterminé, tout en respectant une distance minimale entre la batterie et les éléments sensibles à la chaleur (entre autres contraintes). On pourra, pour chacun de ces problèmes, soit se contenter de trouver une solution satisfaisante (configuration respectant un certain volume, affectation de voies aériennes satisfaisante pour une plage horaire donnée), soit essayer de trouver une solution *optimale* par rapport à un certain critère de qualité (minimiser la place occupée par les composants, minimiser le temps d'attente des avions sur piste). Dans le premier cas, on parlera de *problème de satisfaction de contraintes*[1], et dans le second cas de problème d'*optimisation*, voire d'*optimisation sous contraintes*. Chacun des deux exemples mentionnés peut être exprimé grâce aux notions de distance, de dimension, de volume et de temps, et à certaines opérations mathématiques comme l'addition et la comparaison de nombres réels ou entiers. La possibilité d'exprimer un problème de manière formelle permet de recourir à des *méthodes de résolution* pour le résoudre. De telles méthodes pouvant requérir des calculs extrêmement laborieux, l'utilisation de l'ordinateur est devenue vitale pour un grand nombre de domaines dans lesquels il assiste la tâche de décision et de résolution de problèmes.

L'intérêt pour l'informatique de décision et de résolution de problèmes est donc motivé par un réel enjeu applicatif et industriel, mais également par les profonds problèmes académiques et théoriques qu'elle soulève. Du point de vue industriel, les technologies de résolution de problème sont désormais établies et reconnues, et il existe un réel marché pour ce type de logiciel, citons par exemple la gamme de produits des sociétés Ilog[2], Cosytec[3], Parc Technologies[4] et Prologia[5]. D'après [115], les revenus issus de la seule technologie *programmation par contraintes* s'élevaient à environ 100 millions de dollars pour l'année 1996 ; je ne dispose pas de chiffres plus récents mais les annonces récentes de contrats de plusieurs millions d'euros semblent indiquer que ces chiffres sont à l'heure actuelle largement dépassés. Pourtant, de sévères difficultés techniques restreignent les possibilités du logiciel existant. Parmi les limitations des technologies actuelles, certaines sont liées à la nature même de l'approche, dont une étape importante consiste à modéliser les problèmes à résoudre de façon formelle compréhensible par un ordinateur. Pour certains problèmes, plusieurs difficultés peuvent apparaître : certains paramètres intervenant dans la modélisation peuvent être incertains ou difficiles à estimer. Certains problèmes peuvent être insolubles : cette situation est bien connue, par exemple, des responsables de formations d'enseignement qui tentent de mettre en place un emploi du temps en tenant compte des *desiderata* des différents intervenants. Dans de tels cas, le but sera plutôt de trouver une solution moins "mauvaise" que les autres à défaut de solution parfaite[6]. Pour certains problèmes d'optimisation, la notion de comparaison entre solutions

[1] L'usage de ce terme est parfois restreint à un usage particulier, notamment dans la communauté *Intelligence Artificielle* ; je l'emploie ici dans son acceptation la plus générale.

[2] `http://www.ilog.com`

[3] `http://www.cosytec.com`

[4] `http://www.parc-technologies.com`

[5] `http://prologianet.univ-mrs.fr`

[6] De nombreuses références existent sur la résolution de ces problèmes *sur-contraints*, voir par exemple [30].

peut être problématique, car minimiser un certain paramètre (par exemple le coût de production) peut se faire au détriment d'autres paramètres (la qualité ou le respect des normes de sécurité), et il faudra donc trouver un compromis entre ces différents critères (on parlera alors d'optimisation *multi-critères*). Le langage utilisé pour la modélisation peut, enfin, ne pas être assez "expressif", dans la mesure où certaines constructions particulières peuvent être requises pour exprimer certains problèmes — on insistera par exemple dans cette thèse sur le rôle des *quantificateurs*, qui ne sont en général par autorisés dans les langages actuels de résolution de contraintes. Tous ces points sont des exemples de sujets de recherche récents, pour lesquels des solutions ont été proposées ou sont actuellement étudiées, et qui témoignent ainsi des progrès constants de ce domaine de recherche. Quoi qu'il en soit, le travail de formulation de problèmes demeure une tâche importante, qui produit en général un modèle simplifié et abstrait de la situation réelle à laquelle on souhaite apporter des solutions.

Passées les difficultés liées à la modélisation, reste une question fondamentale : celle de la résolution par des moyens informatiques du modèle mathématique produit. La plupart des problèmes réels actuellement traités par des techniques de résolution automatique sont modélisés par des *problèmes combinatoires*, c'est à dire des problèmes consistant à trouver une solution parmi un nombre fini de solutions candidates[7]. C'est le cas par exemple des problèmes de planification et d'emploi du temps, dans lesquels on dispose d'un certain nombre de créneaux horaires ou de tâches à effectuer, et pour lesquels le but est de trouver la bonne combinaison ou le bon enchaînement de tâches. C'est également le cas de problèmes fort différents, tels les problèmes de vérification de circuits, qui consistent (par exemple) à s'assurer qu'un circuit calcule la bonne valeur de sortie pour toutes les combinaisons d'entrées qui peuvent lui être appliquées. Les problèmes combinatoires sont, du point de vue de leur formulation mathématique, parfaitement triviaux : ordonnancer une chaîne de production, ce n'est rien d'autre que trouver une bonne permutation des tâches à ordonnancer. Une méthode de résolution naïve consiste à énumérer, au moyen d'un ordinateur, les différentes permutations possibles, et à calculer la meilleure. Cette méthode de calcul est facile à programmer et requiert un temps de calcul fini. Pourtant, *en pratique*, le traitement informatique des problèmes combinatoires peut s'avérer extrêmement difficile, en raison des énormes ressources calculatoires nécessaires. Un risque typique est le suivant : pour résoudre un problème dont la description tient en quelques lignes, il est parfois nécessaire d'explorer un ensemble de possibilités *exponentiellement* plus grand que la longueur de cette description. Or les fonctions exponentielles sont caractérisées par une croissance extrêmement rapide. En guise d'exemple ce paragraphe contient moins de 400 caractères et tient sur une demi-page. Si on considère maintenant une exponentielle de ce nombre (disons 2^{400}), on obtient un nombre proprement astronomique, supérieur au nombre de particules de l'univers connu[8]. En conséquence, il arrive que des problèmes mettant en jeu un petit nombre de variables et dont la formulation tient en quelques lignes requièrent l'exploration d'un *espace de recherche* gigantesque : même sur des machines pouvant effectuer un grand nombre de calculs par seconde, l'énumération exhaustive de toutes les possibilités générées par le problème peut requérir des mois, des années, ou des

[7] Il est intéressant de remarquer que de nombreuses notions mises en œuvre dans les problèmes de la vie réels sont essentiellement *continues* : le temps et l'espace en sont des exemples flagrants. Pourtant, ces problèmes sont en général modélisés par des approximations *finies* : on considère par exemple des créneaux horaires d'une certaine durée, ou on découpe un espace aérien en un nombre fini de zones. Ce constat n'est peut-être pas sans raison, et est sans doute lié à la nature même des ordinateurs, qui sont essentiellement des outils de manipulation *symbolique*, donc discrète. Même la manipulation des "nombres réels" en machine est en fait une approximation discrète et bornée. La distinction entre discret et continu est en fait un sujet de débats profonds à propos desquels on pourra consulter, par exemple [184, 262]

[8] Cette affirmation n'est évidemment pas de moi, mais de Leonid LEVIN [177], qui la tient sans aucun doute d'un astrophysicien bien informé! LEVIN va en fait plus loin puisqu'il résumera la situation par le slogan : *"Only math nerds would call* 2^{500} *finite" / "Il faudrait être un matheux borné pour considérer* 2^{500} *comme une nombre fini!"*

milliards d'années selon le problème.[9]

Une illustration des déceptions rencontrées lors des premières approches de l'utilisation des calculateurs pour la résolution de problèmes est donnée par le témoignage de Richard KARP [161] sur des expériences réalisées en 1959 (le même optimisme naïf est sensible dans de nombreux travaux de la même époque, citons ceux de Martin DAVIS sur l'arithmétique de Presburger [81], ou ceux d'un duo de chercheurs récompensés, comme KARP, par le *Turing Award*, Alan NEWELL et Herbert SIMON [208]) :

> Our group's mission was to create a computer program for the automatic synthesis of switching circuits. [...] The program we designed contained many elegant shortcuts and refinements, but its fundamental mechanism was simply to enumerate the possible circuits in order of increasing cost. The number of circuits that the program had to go through grew at a furious rate as the number of input variables increased, and as a consequence, we could never progress beyond the solution of toy problems. Today, our optimism in even trying an enumerative approach may seem utterly naive, but we are not the only ones to have fallen into this trap; much of the work in automated theorem proving over the past two decades has begun with an intial surge of excitement as toy problems were successfully solved, followed by disillusionment as the full seriousness of the combinatorial explosion phenomenon became apparent.
>
> *La mission de notre groupe était de créer un programme informatique pour la synthèse de circuits logiques. [...] Le programme réalisé contenait de nombreux raccourcis et raffinements, mais son mécanisme fondamental était simplement d'énumérer les circuits possibles par ordre de coût croissant. Le nombre de circuits que le programme devait parcourir grandissait le manière furieuse et, en conséquence, nous ne pouvions jamais progresser au-delà d'exemples jouets. De nos jours, notre enthousiasme pour le fait même d'essayer une approche énumérative peut sembler d'une naïveté outrageuse, mais nous ne sommes pas les seuls à être tombés dans ce piège ; la plupart des travaux en preuve automatique de théorèmes des vingt dernières années a débuté par une poussée d'excitation lors de la résolution de premiers problèmes jouets, suivie d'une désillusion lors de la prise de conscience de l'importance du phénomène d'explosion combinatoire.*

Les problèmes combinatoires ont donc cette caractéristique remarquable d'être d'un certain point de vue mathématiquement triviaux et pourtant extrêmement difficiles d'un point de vue calculatoire. Une partie importante de la recherche qui leur a été consacrée dans les dernières décennies a eu pour but d'améliorer les méthodes de résolution existantes afin de pouvoir traiter des problèmes de moins en moins triviaux et, d'autre part, de mieux comprendre les raisons de cette *complexité* calculatoire. La combinatoire joue en fait un rôle absolument central en informatique, au point de couvrir des axes de recherche entiers comme la théorie de la complexité, la théorie des graphes ou la logique calculatoire[10] et en particulier la théorie des modèles finis, pour mentionner seulement certains des plus directement reliés. Notre recherche s'inscrit principalement dans la première problématique mentionnée, celle d'amélioration des techniques de résolution. Nous reviendrons en fin de chapitre sur nos contributions personnelles.

[9] Une question essentielle concernant les temps de calcul exponentiels est de savoir si la *puissance de calcul* disponible est, elle-aussi, amenée à croître de manière exponentielle. C'est actuellement le cas, puisque la vitesse de calcul (en nombre d'instructions par seconde) des ordinateurs actuels continue de doubler tous les 18 mois suivant la fameuse *loi de* MOORE, formulée en 1965 [203] et qui s'est depuis invariablement vérifiée. Comme remarqué notamment par Samuel BUSS [45] :

> It is interesting to note that if the rate of increase could be sustained indefinitely, then exponential time algorithms would actually be feasible, since one would merely wait a polynomial amount of time for computer power to increase sufficiently to run the algorithm quickly.
>
> *Il est intéressant de noter que si cette croissance pouvait être maintenue indéfiniment, les algorithmes exponentiels seraient en fait réalisables puisqu'il suffirait d'attendre un nombre d'années polynomial pour que la puissance de calcul soit suffisamment élevée pour exécuter l'algorithme efficacement.*

Il est cependant conjecturé que certaines limites physiques empêcheront cette accélération des machines d'être infinie (MOORE prévoit pour sa part un ralentissement dès 2017).

[10] Encore une expression pour laquelle aucun terme français n'est, à ma connaissance, consacré : c'est bien de *computational logic* dont je parle ici.

Avant cela, nous nous proposons de présenter plus en détails la notion de logique formelle, et d'illustrer son emploi comme langage d'expression de problèmes.

1.2 Des langages formels pour exprimer des problèmes

Un énoncé mathématique ou un problème de la vie réelle peuvent souvent s'exprimer de manière plus ou moins *formelle*. Le langage parlé permet en général de décrire des problèmes de manière approximative, alors que l'utilisation de notations mathématiques permet d'exprimer les mêmes notions de manière plus rigoureuse. Les deux approches ont en général des buts différents et sont complémentaires : le langage naturel permet de communiquer et est plus facile à comprendre. Les notations formelles sont d'un abord ardu et il est difficile de percevoir leur sens sans avoir d'explication préalable, mais elles sont indispensables à tout effort de démonstration. L'avènement de l'ordinateur n'a fait qu'ajouter au caractère bureaucratique de la syntaxe formelle. Pour être traitées de manière automatique, les notations utilisées pour exprimer un programme ou un problème doivent être standardisées et obéir à des règles précises. En guise d'illustration de la différence entre formulation formelle et informelle, considérons un exemple tiré des mathématiques élémentaires. La notion de *fonction continue* (cf. Figure 1.1) peut s'exprimer ainsi en langage informel :

Une fonction est continue si on peut la dessiner "sans lever le crayon".

Si cette définition "parle" à l'intelligence humaine, elle ne se prête cependant guère à des déductions poussées. A y regarder de plus près, cette définition requière une notion de distance (on parlera d'*espace métrique*), car il s'agit en fait d'exprimer le fait que "le crayon ne s'écarte pas trop" quand on dessine la fonction. Précisons :

Une fonction f est continue en un point x si les points situés infiniment près de x ont des images par f infiniment proches de $f(x)$.

Une définition rigoureuse est par exemple la suivante (traduite de [168]) :

> Soit f une application d'un espace métrique X dans un autre espace métrique Y, tel que f associe un élément $y = f(x) \in Y$ à tout élément $x \in X$. f est dite *continue en* $x_0 \in X$ si, pour tout $\epsilon > 0$, il existe un $\delta > 0$ tel que
>
> $$\rho'(f(x), f(x_0)) < \epsilon$$
>
> dès lors que
>
> $$\rho(x, x_0) < \delta$$
>
> (ρ est la distance sur X et ρ' la distance sur Y). L'application f est dite *continue sur* X si elle est continue en tout point $x \in X$.

Cette définition mélange langage parlé et notation mathématique. A la fois rigoureuse et compréhensible, elle présente ainsi un compromis répondant aux buts d'un ouvrage destiné aux humains. En exprimant un énoncé de manière formelle, on a invariablement recours à des mots comme "implique", "pour tout" ou "il existe". Ces mots sont tellement utilisés dans les définitions mathématiques qu'on a défini des symboles spéciaux pour les représenter : $\forall$ signifie "pour tout", $\exists$ signifie "il existe", les symboles $\vee$ et $\wedge$ représentent, respectivement, "ou" et "et" ; $\rightarrow$ (également noté $\supset$ par certains auteurs) signifie "implique"[11]. On peut reformuler les énoncés précédents en utilisant ces symboles :

[11] Voir le récent livre de Martin DAVIS [82] pour un historique de la naissance de la logique de premier ordre, mise en parallèle avec l'histoire de la notion de calcul et d'ordinateur.

continue($f \in Y^X$, $x_0 \in X$) $\equiv$

$$\forall \epsilon > 0 \ \exists \delta < 0 \ \forall x \ \ \big(\ \rho(x, x_0) < \delta) \ \big) \rightarrow \big(\ \rho'(f(x), f(x_0)) < \epsilon \ \big)$$

continue($f \in Y^X$) $\equiv$

$$\forall x \in X \ \ continue(f, x)$$

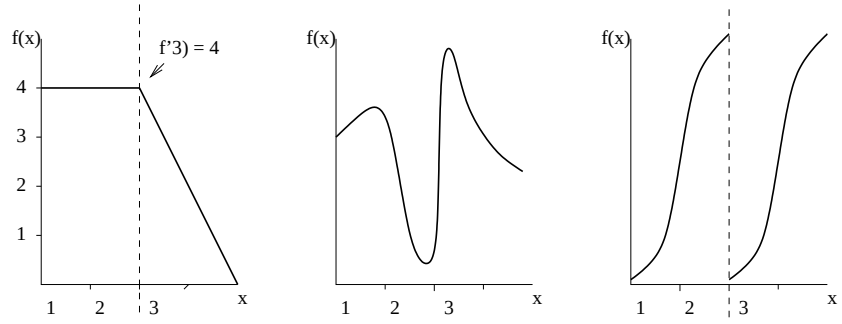


Figure 1.1 – Une fonction est définie par un résultat $f(x)$ unique associé à toute valeur fournie en entrée. La figure présente trois exemples de fonction : les deux premières sont continues. La troisième est quant à elle discontinue (droite) car elle change brusquement de valeur en 3 — il faut "lever le crayon".

L'intérêt de la formulation mathématique ne réside pas tant dans l'utilisation d'abréviations mathématiques[12] que dans les travaux formels que ces notations ont rendus possibles. La *syntaxe* des expressions mathématiques considérées peut en effet être définie de manière suffisamment précise pour qu'une machine puisse reconnaître les énoncés syntaxiquement corrects et construire une représentation interne de ces énoncés. De plus, une séparation claire entre la forme (syntaxe) et le sens des formules a permis de faire émerger deux points de vue complémentaires sur les formules logiques : la *théorie de la preuve* et la *théorie de la sémantique*. La théorie de la preuve vise à comprendre la façon dont les preuves sont construites de façon syntaxique, et elle permet dans une certaine mesure d'en automatiser la découverte. La théorie de la sémantique donne un sens formel aux formules logiques et aux opérations mécaniques de transformation de symboles, et permet ainsi de garantir la correction de ces transformations.

C'est donc tout un ensemble de qualités formelles, dû à la rigueur de la définition "bureaucratique" de ce langage qui confère à la logique les qualités requises pour un traitement informatique. Du point de vue de la machine, les déductions logiques se réduisent à des étapes de transformation purement syntaxique, et ce point de vue se trouve également être celui des théoriciens de la preuve.

1.2.1 Exprimer les problèmes combinatoires en logique

La logique se trouve être un excellent langage dans lequel exprimer les problèmes combinatoires mentionnés en début de chapitre. Pour s'en convaincre, analysons un exemple de problème combinatoire simple, celui du jeu de *morpion*.

Pour modéliser ce jeu, nous devons tout d'abord définir la notion de *plateau de jeu*. Un plateau est un ensemble de neuf cases pouvant soit recevoir un pion (croix ou rond), soit demeurer vides. On peut donc définir l'ensemble $\mathcal{P}$ des *positions de jeu* par $\mathcal{P} = \{1, 2, \ldots, 9\}$. Les valeurs possibles pour une

[12] En témoigne la répulsion de certains des mathématiciens les plus renommés aux débuts de la logique formelle. POINCARÉ (cité dans [82]) écrira qu'*il est difficile de voir en quoi le mot "si", lorsqu'il est écrit $\supset$, acquiert une quelconque vertu qu'il ne possédait déjà lorsqu'on l'écrivait "si".*

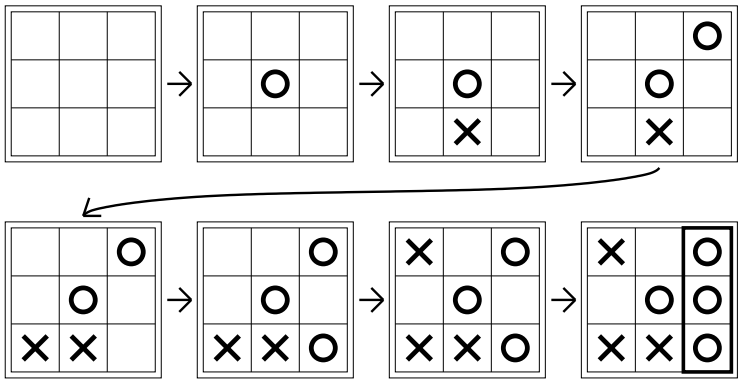


Figure 1.2 – Un exemple simple de problème combinatoire

case seront, par exemple, notées □ pour la case vide, O pour une case occupée par un rond, et X pour une case occupée par une croix.

1	2	3
4	5	6
7	8	9

Figure 1.3 – Numérotation des cases d'un plateau de jeu

Une *configuration de jeu* est représentée par la valeur prise par chacune des 9 cases à un instant donné. On notera C_i la valeur prise par la case de numéro $i \in 1..9$. C_i vaudra donc toujours soit □, soit O, soit X (en termes informatiques, C est un *tableau* indexé par des positions et à valeurs dans $\{\square, \mathsf{O}, \mathsf{X}\}$). La configuration de début de partie est par exemple définie par :

$$\text{config\_init}(C) \;\equiv\; \left(\; C_1 = \square \;\wedge \cdots \wedge\; C_9 = \square \; \right)$$

Les configurations gagnantes sont elles aussi faciles à exprimer : le joueur X (par exemple) a gagné la partie s'il est parvenu à aligner trois pions en forme de croix, c'est à dire si les cases 1, 4, et 7 portent des croix, ou les cases 3, 5, et 7, etc. (la combinaison soulignée dans la formule suivante correspond par exemple à la colonne grisée dans la figure 1.3) :

$$\mathsf{X}\text{\_gagne}(C) \equiv \left(\begin{array}{l} (C_1 = \mathsf{X} \wedge C_2 = \mathsf{X} \wedge C_3 = \mathsf{X}) \vee (C_1 = \mathsf{X} \wedge C_4 = \mathsf{X} \wedge C_7 = \mathsf{X}) \vee \\ (C_4 = \mathsf{X} \wedge C_5 = \mathsf{X} \wedge C_6 = \mathsf{X}) \vee \underline{(C_2 = \mathsf{X} \wedge C_5 = \mathsf{X} \wedge C_8 = \mathsf{X})} \vee \\ (C_7 = \mathsf{X} \wedge C_8 = \mathsf{X} \wedge C_9 = \mathsf{X}) \vee (C_3 = \mathsf{X} \wedge C_6 = \mathsf{X} \wedge C_9 = \mathsf{X}) \vee \\ (C_1 = \mathsf{X} \wedge C_5 = \mathsf{X} \wedge C_9 = \mathsf{X}) \vee (C_3 = \mathsf{X} \wedge C_5 = \mathsf{X} \wedge C_7 = \mathsf{X}) \end{array} \right)$$

Lorsqu'un joueur (disons X) pose une pièce sur la position p, supposée libre, la configuration change car la case p prend la valeur X. Notons C la configuration avant le jeu et C' la configuration après le coup, la relation entre les deux s'écrit :

$$\text{coup\_}\mathsf{X}(p, C, C') \equiv \quad \left(\; C_p = \square \wedge \quad \bigwedge_{i \in 1..9} \left(\begin{array}{l} i = p \;\rightarrow C'_i = \mathsf{X} \;\wedge \\ i \neq p \;\rightarrow C'_i = C_i \end{array} \right) \right)$$

La notation $\bigwedge_{i\in 1..9}$ représente la conjonction des 9 propositions obtenues en remplaçant i par $1, 2, \ldots 9$, successivement. Les implications ($\rightarrow$) signifient, respectivement, que "si la case considérée (C_i) est la case sur laquelle X a joué, alors cette case prend la valeur X", et "sinon, elle reste inchangée". Les règles du jeu sont entièrement modélisées. On peut maintenant essayer de formuler une question.

Une question fondamentale de l'étude mathématique des jeux[13] à deux joueurs consiste à déterminer s'il est possible au second joueur de gagner. Le fait de jouer *chacun son tour* induit en effet un déséquilibre entre les deux joueurs et il existe des jeux pour lesquels, *s'il jouait toujours parfaitement*, le premier joueur aurait la garantie d'être vainqueur. L'existence d'une stratégie gagnante doit prendre en compte tous les coups possibles de l'adversaire (O) : pour chacun de ces coups, il doit être possible de déterminer un coup tel que, quel que soit le reste la partie, le premier joueur conserve la certitude de gagner. La question "existe-t'il une stratégie gagnante en 5 coups" peut en fait s'exprimer de la manière suivante :

Il existe un coup pour X tel que
Pour tout coup de O,
Il existe un coup pour X tel que ... X gagne

c.-à-d., de manière totalement formelle[14] :

$$\mathsf{X\_gagne} \equiv \left(\begin{array}{l} \exists x_0 \in 1..9 \\ \quad \forall o_1 \in 1..9 \quad \exists x_1 \in 1..9 \\ \quad\quad \ldots \quad \forall o_4 \in 1..9 \quad \exists x_4 \in 1..9 \\ \quad\quad\quad \left(\begin{array}{lll} \text{config\_init}(C_0) & \wedge & \text{coup\_X}(x_1, C_0, C_1) \quad \wedge \\ \text{coup\_O}(o_1, C_1, C_2) & \wedge & \ldots \quad\quad\quad\quad\quad \wedge \\ \text{coup\_X}(x_4, C_8, C_9) & \wedge & \mathsf{X\_gagne}(C_9) \end{array} \right) \end{array} \right)$$

1.2.2 Quelle logique?

Même s'il s'agit d'un simple exemple jouet, l'exemple précédent illustre bien l'usage de la logique pour formuler des problèmes combinatoires. La logique dont nous avons eu besoin se limite en fait (en plus des variables introduites pour un problème particulier) à un jeu de symboles restreint, comportant principalement égalité, négation, implication, et quantificateurs. Il est cependant légitime de se demander si ce langage est le seul possible et en fait, un grand nombre de propositions ont été faites au cours du XX^e siècle pour définir d'autres "logiques" et d'autres langages formels. La position respective de ces différents formalismes est clarifiée par cette déclaration sans équivoque d'Alan ROBINSON [226] :

> First-order logic is all the logic we have and all the logic we need. It was certainly all the logic Gödel needed to present all of general set theory, defining in the process the "higher-order" notions of function, infinite cardinals and ordinals, and so on [...]. Within FOL we are completely free to postulate, by formulating suitably axiomatized first order theories, whatever more exotic constructions we may wish to contemplate in our ontology, or to limit ourselves to more pasimonious means of inference than the full classical repertoire. [...] Thus FOL

[13] Bien que les jeux étudiés dans cette thèse ne brillent pas par leur difficulté, il serait injuste de considérer l'étude mathématique des jeux comme un domaine futile ou purement académique. La *théorie des jeux* remonte à la première moitié du XX^e siècle (travaux de VON NEUMANN et MORGENSTERN [255], récompensés par un prix Nobel d'économie) et modélise de nombreuses situations mettant en présence un ensemble d'agents aux intérêts contradictoires. Elle s'applique à l'économie, la stratégie militaire, aux systèmes multiagents et à internet, et même à l'étude des comportements terroristes (*Libération*, 12/02/2003). Pour une introduction accessible, on pourra consulter [80].

[14] Une petite subtilité échappe toutefois à notre formalisation : la formule exprime l'existence d'une stratégie gagnante *en exactement 5 coups*, on cherche en général à gagner en *au plus* 5 coups, ce qui demande de modifier légèrement la définition des prédicats "coup".

> can be used to set up, as first-order theories, the many "other logics" such as modal logic, higher order logic, temporal logic, dynamic logic, concurrency logic, epistemic logic, nonmonotonic logic [...] linear logic, fuzzy logic, intuitionistic logic, causal logic, quantum logic; and so on and so on.
>
> *La logique de premier ordre est la seule logique dons nous disposons et la seule logique dont nous ayons besoin. C'était sans nul doute la seule logique dont Gödel a eu besoin pour présenter sa théorie générale des ensembles, introduisant au passage les notions d'"ordre supérieur" que sont les fonctions, les cardinaux et ordinaux infinis, etc. [...]. En logique du premier ordre, nous sommes totalement libres de postuler, par des théories de premier-ordre convenablement axiomatisées, les constructions les plus exotiques que l'on puisse souhaiter voir apparaître dans notre ontologie ou, à l'inverse, de nous limiter à des moyens d'inférence plus parcimonieux que ceux du répertoire classique pris dans son intégralité. [...] Par conséquent, la logique du premier ordre peut être utilisée pour construire, en tant que théories du premier ordre, les nombreuses "autres logiques" telles la logique modale, la logique d'ordre supérieur, la logique temporelle, la logique dynamique, la logique concurrente, la logique épistémique, la logique non monotone, [...] la logique linéaire, la logique floue, la logique intuitionniste, la logique causale, la logique quantique, etc.*

Plus loin dans son exposé, Robinson qualifie les autres logiques de *sucre syntaxique*, c'est à dire d'abréviations pour des constructions de logique premier ordre, et il est vrai que toutes les constructions des autres logiques s'expriment en logique classique, plus précisément dans la logique mentionnée par Robinson, celle *de premier ordre*. La raison en est simple : la logique de premier ordre est un langage TURING-*complet*, permettant d'exprimer toute fonction calculable[15], d'où son usage intensif en intelligence artificielle[16]. Deux remarques s'imposent à ce sujet. Tout d'abord, cette expressivité a ses effets pervers : la propriété de TURING-complétude rend la logique de premier ordre *indécidable* — aucune méthode systématique ne permet de déterminer si une formule de premier ordre est vraie. Deuxièmement, le même argument d'*universalité* est valable pour d'autres notations formelles, qui vont des langages de programmation classiques à des constructions aussi austères que les automates cellulaires. Il n'en reste pas moins que les notations logiques classiques sont d'un usage répandu en mathématiques et qu'il est souvent naturel de formuler un énoncé en logique classique si ce n'est en logique de premier ordre.

Bien qu'il faille en effet les considérer comme du *sucre syntaxique*, les autres formalismes logiques comme les logiques exotiques, notamment modales ou "d'ordre supérieur", ou les restrictions de la logique classique n'en présentent pas moins des avantages. Elles peuvent permettre d'exprimer les problèmes de certaines classes de manière plus naturelle, ou il peut être plus simple de déterminer si elles sont vraies. Nous nous rallierons aux propos de Robin MILNER [197] qui défend ainsi la coexistence de différentes notations formelles[17] :

> I make a disclaimer. I reject the idea that there can be a unique conceptual model or one preferred formalism [...]. We need many *levels of explanation*, many different languages, calculi, and theories for the different specialisms.
>
> *J'aimerais ici apporter une mise en garde : je rejette l'idée de l'existence d'un modèle conceptuel unique ou d'un formalisme privilégié [...]. Nous avons besoin d'une multitude de* niveaux d'explication*, de différents langages, calculs, et théories pour les spécificités les plus diverses.*

[15] Rappelons que la notion de fonction *calculable* a été formalisée par CHURCH et TURING quelques années après le premier résultat d'incalculabilité donné par le théorème de GOËDEL [131]. La définition de CHURCH et TURING [246, 79] traduit fidèlement la notion de fonction calculable par les machines actuelles et il est conjecturé qu'aucun processus physiquement réalisable (raisonnement humain inclus) ne peut effectuer de calculs significativement plus complexes (lecture "intelligence artificielle" de l'hypothèse de CHURCH-TURING ; voir par exemple [90] pour un plaidoyer récent en faveur de cette thèse).

[16] Pour être plus précis sur la position privilégiée de la logique de premier ordre, ajoutons qu'il est même communément admis par les spécialistes [64] que l'ensemble des preuves mathématiques connues peuvent être formalisées à partir d'une certaine *théorie* de premier ordre, c.-à-d. un certain ensemble infini d'axiomes dits de *Zermelo-Frænkel* avec axiome de choix.

[17] Je me suis permis d'extraire les propos de MILNER d'un contexte particulier, celui de *calcul concurrent*. Il me semble cependant ne pas trahir ses propos en les reportant ici.

De fait, une grande variété de langages ont été proposés pour modéliser et résoudre des problèmes, en particulier des problèmes combinatoires. Issue des travaux de COLMERAUER et KOWALSKI, la *programmation logique* est un paradigme de programmation déclarative d'usage général (Prolog est un langage TURING-complet), mais trouvant en particulier des applications à la résolution de problèmes [170]. En parallèle, et indépendamment de toute formulation logique, les *problèmes de satisfaction de contraintes* (CSP) ont émergé dans les années 1970 des travaux de MONTANARI et MACKWORTH [202, 186] (qui introduisent respectivement les notions de *consistance de chemin* et de *consistance d'arc*) puis FREUDER [112] (les travaux antérieurs de WALTZ [259] faisaient implicitement usage de techniques similaire dans le cadre particulier d'une application de reconnaissance de scènes). Les problèmes de satisfaction de contraintes consistent à exprimer des problèmes combinatoires en décrivant les contraintes que les solutions attendues doivent respecter.

La programmation logique fournit un langage permettant d'exprimer des problèmes ; les CSP sont quant à eux spécialisés sur les applications combinatoires, pour lesquels ils proposent des méthodes de résolution efficace. L'intérêt de confronter ces deux notions a été reconnu au cours des années 1980, pendant lesquelles la notion de *programmation par contraintes* a émergé des travaux de COLMERAUER [65], JAFFAR et LASSEZ [154] et, pour la partie combinatoire, des recherches menées à l'ECRC par l'équipe travaillant sur le système Chip [247] (voir également [155] pour un état des recherches de l'époque). De nombreux langages de programmation par contraintes basés sur Prolog ou sur d'autres notations ont depuis été proposés. Un exemple récent de langage de modélisation déclarative de problèmes est donné par OPL (cf. Fig. 1.4), un langage proche des langages de modélisation mathématique et qui, a de nombreux égards, renoue également avec une autre tradition de programmation déclarative, celle initiée dans les années 1970 avec le système Alice [172][18].

```
int n = ...;
range Domain 1 ..n;
var Domain queens [Domain];
solve
    forall (i, j in Domain) {
         queens[i]   <> queens[j];
         queens[i]+i <> queens[j]+j;
         queens[i]-i <> queens[j]-j;
    }
```

Figure 1.4 – Un exemple de "modèle" OPL d'un problème combinatoire jouet, celui des reines [250], consistant à placer n reines sur un échiquier de n cases sur n sans qu'aucune reine n'en menace une autre.

Notons enfin, pour conclure sur la variété des langages de modélisation des problèmes combinatoires, que les langages issus de la programmation par contraintes ne sont pas les seuls utilisés. Un tout autre exemple est donné par les problèmes de vérification de programme, dans lesquels les propriétés attendues sont exprimées dans certaines formes de logique modale appelées *logiques temporelles* [59, 101].

[18] Cet article, — une des fiertés de la recherche française en Intelligence Artificielle — semble être le premier à présenter un langage *déclaratif* de résolution de problèmes combinatoires. Les références antérieures généralement citées sur des réalisations similaires sont Sketchpad [242], un logiciel déclaratif mais limité, semble-t'il, aux applications graphiques et REF-ARF [108], un système de résolution de problèmes combinatoires permettant en fait d'exprimer des algorithmes non-déterministes et duquel la notion de déclarativité est donc absente.

1.3 Des méthodes pour raisonner sur les énoncés logiques

Le fait de pouvoir exprimer un problème dans un certain langage formel permet d'utiliser un ordinateur pour leur résolution. Une des conséquences est la suivante : résoudre un problème, de quelque nature qu'il soit, cela se réduit à un exercice de déduction logique. Bien entendu, tous les problèmes ne sont pas également faciles à résoudre ; certains sont triviaux, d'autres tout bonnement insolubles. En termes de logique, la difficulté d'un énoncé dépend de critères syntaxiques : selon la richesse du langage logique utilisé, il peut être possible d'exprimer des problèmes plus ou moins complexes. Déterminer si les formules de ce langage logique sont vraies sera par conséquent plus ou moins complexe.

La notion de complexité d'un calcul s'exprime en fonction des ressources nécessaires pour effectuer ce dernier. Les machines ont besoin de *temps* pour effectuer des opérations, et de *mémoire* pour stocker les résultats intermédiaires des calculs. La complexité des problèmes est donc évaluée en fonction des ressources requises par la résolution de ce problème en termes de temps et d'espace.

1.3.1 Une classification des problèmes combinatoires

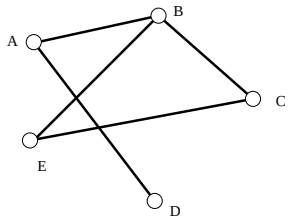
La classification des problèmes combinatoires en fonction de leur complexité remonte aux années 1960, notamment aux travaux de Juris HARTMANIS et Richard E. STEARNS [142][19]. Dit simplement, le résultat essentiel de cet article est que pour tout k, il existe un problème calculatoire non soluble en temps[20] $\mathcal{O}(n^k)$: il existe des problèmes requérant un temps de calcul polynomial de degré arbitrairement élevé. Ici, le temps de calcul est fonction de la longueur (n) de la description du problème dans un certain langage formel. Les problèmes calculatoires forment donc une hiérarchie infinie de *classes de complexité* : la classe des problèmes solubles en temps $\mathcal{O}(n^2)$, $\mathcal{O}(n^3)$, etc. ; ceux solubles en temps polynomial, c'est à dire en temps $\mathcal{O}(n^k)$ pour un certain k ; ceux solubles en temps exponentiel, et ainsi de suite. L'idée se généralise également aux ressources en espace, et on pourra parler de problèmes solubles en espace $\mathcal{O}(n^4)$, en espace polynomial, exponentiel, etc. Il est clair qu'un ordinateur consomme moins d'espace que de temps, puisque chaque accès mémoire nécessite une certaine unité de temps. La classe des problèmes solubles en espace polynomial contient par exemple l'ensemble des problèmes solubles en temps polynomial — on conjecture en fait que cette inclusion est stricte, et qu'elle contient infiniment plus de problèmes, bien que ce résultat demeure improuvé à l'heure actuelle.

La nature de la relation d'ordre entre problèmes mettra encore quelques années à être clairement explicitée. Le fait qu'un problème soit plus difficile qu'un autre signifie en fait que la résolution du plus difficile peut servir à résoudre le plus simple, au prix d'un *encodage*, c'est à dire d'une traduction du problème initial dans le langage du problème cible. Il est ainsi connu depuis les travaux de *programmation linéaire* du début du XX^e siècle [77, 56] que la *programmation linéaire en nombres entiers* permet d'exprimer un grand nombre de problèmes, et notamment la plupart des problèmes classiques sur les graphes. Un *problème de programmation linéaire* est un ensemble d'équations et "inéquations" linéaires (de forme $\sum_i a_i.x_i \leq 0$), et le but de la programmation linéaire en nombres entiers et de calculer des

[19] HARTMANIS et STEARNS recevront le *Turing Award* en 1993 *en reconnaissance de leur travail fondateur qui a établi les bases de la théorie de la complexité calculatoire*. L'article cité pose la notion de *classe de complexité* (en définissant les classes de type DTIME(n^k)) et prouve un des rares théorèmes de séparation de classes connus en théorie de la complexité — et probablement le plus important prouvé à ce jour (il implique, en particulier, que la classe *temps exponentiel* est strictement plus large que la classe *temps polynomial*). Il existe bien entendu des précurseurs de cet article, parmi lesquels les travaux de CHOMSKY, qui ont mis en évidence la notion de hiérarchie de langages formels, ou ceux de RABIN, COBHAM, ou EDMONDS, qui identifiaient dès les années 1960 la notion d'*effectivement calculable* à celle de *temps de calcul polynomial*.

[20] On rappelle que la notation $\mathcal{O}$ désigne la vitesse de *croissance* d'une fonction : $f \in \mathcal{O}(g)$ signifie que f est une fonction qui croît au plus aussi vite que g.

solutions entières (prises dans l'ensemble $\mathbb{N}$ des nombres *entiers*, c.-à-d. sans virgule) satisfaisant les égalités et inégalités formulées. Un exemple de problème de graphes[21] est donné par le problème de *coloration*, consistant à attribuer l'une des couleurs *rouge*, *vert* ou *bleu* à chaque sommet de façon à ce que deux sommets reliés par une arête soient toujours de couleur différente. Il est possible d'exprimer un problème de coloration de graphes comme un problème de satisfaction de formule logique propositionnele, ou comme un problème de programmation en nombre entiers. Cela signifie que ces deux problèmes sont "plus difficiles" que la coloration de graphes : le problème de coloration de graphes *se réduit* au problème de satisfaction de formule logique propositionnelle, qui se réduit lui-même au problème de programmation linéaire en nombres entiers. Il s'avère en fait que l'inverse est également vrai, et que les problèmes de programmation en nombre entiers et de satisfaisabilité peuvent être réduits à des problèmes de coloration de graphes. Ces trois problèmes en apparence fort différents sont donc, d'un certain point de vue, *équivalents*.



$\neg$ A_rouge $\vee$ $\neg$ B_rouge	A_rouge + B_rouge ≤ 1
$\neg$ A_vert $\vee$ $\neg$ B_vert	A_vert + B_vert ≤ 1
$\neg$ A_bleu $\vee$ $\neg$ B_bleu	A_bleu + B_bleu ≤ 1
...	...
A_rouge $\vee$ A_vert $\vee$ A_bleu	A_rouge + A_vert + A_bleu ≥ 1
...	...

Figure 1.5 – Le problème de coloration d'un graphe peut être exprimé sous forme d'un problème de satisfaction de formule logique, qui peut lui-même facilement être traduit en un problème de programmation linéaire en nombres entiers (à valeurs 0/1).

C'est avec le théorème de COOK [68][22] que le rôle du problème de satisfaisabilité (sous ses différents avatars) recevra un éclairage nouveau : ce théorème montre que tous les problèmes d'une certaine classe se réduisent en fait au problème de satisfaisabilité ; la classe en question est celle des problèmes de *recherche* de solution à un ensemble de contraintes pouvant être vérifiées en temps polynomial. Elle porte le nom de **NP**[23]. De nombreux travaux [160, 118] faisant suite au théorème de COOK montreront que ce phénomène de *complétude* est en fait extrêmement répandu : la plupart des problèmes de recherche pour lesquels aucun aucun algorithme polynomial n'est connu s'avèrent en fait **NP**-complets. De même, la plupart des autres classes de complexité sont représentées par un certain nombre de problèmes complets, qui sont en quelque sorte les représentants de l'ensemble des problèmes de la classe. PAPADIMITRIOU résume ainsi l'impact du théorème de COOK[24] :

[21] Un graphe [133] est un ensemble de points appelés *sommets* reliés par des traits appelés *arêtes*.

[22] Il est notoire que le théorème a été démontré indépendamment par Leonid LEVIN, qui le publiera en URSS en 1973, alors que peu de confrontation était possible entre les chercheurs soviétiques et occidentaux [178]. De manière surprenante, le problème **NP**-complet exhibé par LEVIN n'était pas SAT mais un problème de pavage. Il semblerait que, tout comme KARP [160], LEVIN ait pleinement entrevu l'étendue des résultats de **NP**-complétude, puisqu'il écrira (reporté dans [210]) :

> The method described here clearly provides a means for readily obtaining results of this type for the majority of sequential search problems.
>
> *La méthode décrite ici représente clairement un moyen d'obtenir des résultats similaires pour la majorité des problèmes de recherche séquentielle.*

[23] Pour (temps) *Non-déterministe Polynomial*.

[24] Les trois seuls problèmes importants dont j'ai jamais entendu parler à être dans **NP** et à n'être selon toute conjecture ni dans **P** ni **NP**-difficiles sont le problème d'*isomorphisme de graphes*, celui (parait-il) de détermination d'un équilibre de NASH en théorie des jeux, et le problème de factorisation de nombres premiers. A propos de ce dernier problème, dont le rôle est central en cryptographie puisqu'il rendrait possible le "cassage" de nombreux systèmes de cryptographie à clé publique dont

> Also crucial for the success of **NP**-completeness has been its surprizing ubiquity and effectiveness as a classification tool, and the scarcity of problems in **NP** that resist classification as either polynomial-time solvable or **NP**-complete.
>
> *Un point crucial dans le succès de la théorie de la **NP**-complétude a été sa surprenante ubiquité et sa pertinence comme outil de classification, et la rareté des problèmes résistant à une classification soit en tant que problèmes solubles en temps polynomial, soit en temps que problèmes **NP**-complets.*

La figure 1.6 donne un aperçu des classes de complexité "classiques", en temps et espace, avec non-déterminisme et complémentation (de nombreuses autres classes existent, par exemple celles basées sur des notions stochastiques). Pour chacune de ces classes, nous avons indiqué un ou plusieurs problèmes caractéristiques ; la plupart des exemples choisis sont en fait complets pour la classe considérée. **LOGSPACE** correspond à l'espace logarithmique ; **PH** dénote la hiérarchie polynomiale, qui fera

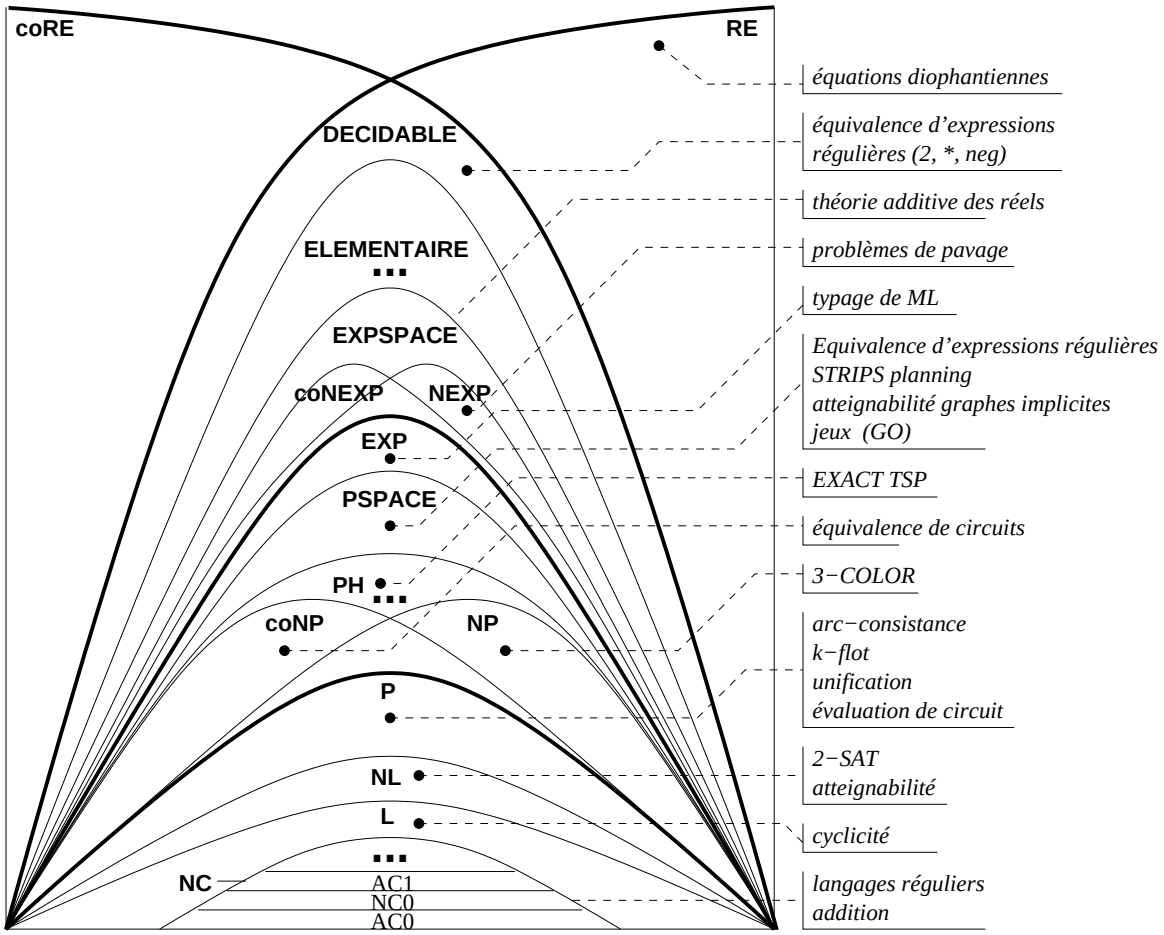


Figure 1.6 – Un aperçu des classes de complexité "classiques".

l'objet d'une brève présentation dans la suite de ce chapitre. **PSPACE** représente l'espace polynomial,

le fameux RSA [224], on remarquera que Peter SHOR a proposé un algorithme [235] polynomial pour résoudre ce problème... sur des machines quantiques. Quant au non moins fameux problème de déterminer si un nombre est premier, il a été résolu pendant le déroulement de cette thèse (mais pas par son auteur!) puisqu'un algorithme polynomial d'ores et déja classique a été proposé [2]. Remarquons enfin que, si peu de problèmes de complexité intermédiaire entre **P** et **NP** ont été identifiés, il en existe pourtant en fait une infinité de classes — sauf si **P**=**NP** (théorème de LADNER [171])!

E le temps exponentiel. Le préfixe **N** (e.g., **NLOGSPACE**) indique une classe *non-déterministe*, c.-à-d. une classe de problèmes de recherche dont la vérification des solutions candidates est un problème de la classe préfixée (e.g., **LOGSPACE**). Le préfixe **co** représente la complémentation, qui sera brièvement discutée par la suite. Les problèmes *élémentaires* sont ceux dont la complexité est bornée par une tour d'exponentielle de hauteur bornée, de forme "10 puissance 10 puissance 10 puissance ..." (certains problèmes sont, hélas, non-élémentaires). Enfin, **RE** représente l'ensemble des problèmes récursivement énumérables. Les problèmes complets pour cette classe sont donc seulement semi-décidables[25]

Les classes de complexité ne doivent pas être considérées comme une simple curiosité théorique, et la complexité d'un problème caractérise sa nature profonde. Le parallèle entre cette classification théorique et la pratique de la résolution de problèmes en Intelligence Artificielle est illustré par les propos suivants, extraits des notes d'un cours donné au MIT par David A. MCALLESTER [195] :

> For the past thirty years, Artificial Intelligence researchers have been studying heuristic search techniques. People writing AI programs have had strong intuitions about what it means for a program to search. Intuitively, they classified programs according to the kind of search performed. Certain programs, such as matrix multiplication routines did no search whatsoever. Other programs searched for solutions to a fixed finite set of constraints. Others searched for paths in graphs, or for strategies in game trees, or for proofs in formal inference systems. The intuitive classification of search programs seems to roughly correspond to complexity classes. The no-search procedures correspond to the complexity class **P**. Constraint satisfaction search procedures correspond to the complexity class **NP**, graph-search and game-search to the complexity class **PSPACE** and search in theorem proving to the class of recursively enumerable functions.
>
> *Au cours de ces trente dernières années, les chercheurs en Intelligence Artificielle ont étudié les techniques de recherche heuristique. Les personnes qui écrivaient ces programmes d'IA avaient des intuitions fortes sur ce que signifie pour un programme le fait de* chercher. *Intuitivement, ils ont classé les programmes en fonction du type de recherche effectuée. Certains programmes, comme la multiplication de matrices, n'effectuaient tout bonnement aucune recherche. D'autres cherchaient des solutions à un ensemble fini, fixé de contraintes. D'autres cherchaient des chemins dans des graphes*[26]*, des stratégies dans des arbres de jeux, ou encore des preuves dans des systemes formels d'inférence. La classification intuitive des algorithmes de recherche semble grossièrement correspondre aux classes de complexité. Les algorithmes n'effectuant pas de recherche correspondent à la classe **P**. Les procédures de satisfaction de contraintes correspondent à **NP**, la recherche dans des graphes et les jeux correspondent à **PSPACE** et la preuve de théorème aux fonctions récursivement énumérables.*

1.3.2 Quantification, expressivité, complexité

Les différentes classes de complexité correspondent à des problèmes aux caractéristiques bien distinctes en pratique, ce qui nous permet, suivant MCALLESTER, d'identifier certains types de problèmes

[25] La signification de la semi-décidabilité est parfaitement résumée par ces propos de Martin DAVIS [84] (à propos du problème de validité des théorèmes de logique du premier ordre, qui est un exemple de problème indécidable) :

> "The only algorithms which can be designed in this case will ultimately locate a proof for any formula of quantification theory which is valid but [...] will involve seeking forever in the case of a formula which is not valid".
>
> *Les seuls algorithmes pouvant être conçus dans ce cas parviendront finalement à localiser une preuve pour toute formule valide mais [...] chercheront éternellement dans le cas de formules non valides.*

[26] Remarquons que le terme *graph search* est employé par la communauté IA, et en particulier par MCALLESTER, pour désigner des *graphes d'états* ou *graphes implicites*, qui modélisent des systèmes de transition de taille exponentielle et ne peuvent donc être stockés. Nous en dirons plus sur ce type de graphes au cours du chapitre 8. Il est bien évident que les algorithmes de recherche dans des graphes représentés de manière explicite est un problème plus simple, de complexité polynomiale (**NLOGSPACE**-complet).

à des classes de complexité données. Mais il existe également un parallèle entre classes de complexité et langages logiques. En effet, la richesse du langage logique utilisable pour exprimer une classe de problèmes est directement liée à la complexité des problèmes considérés. Plus le langage logique autorise de constructions, plus il permet d'exprimer de problèmes combinatoires mais, en contrepartie, plus il devient complexe de déterminer si les énoncés de ce langage sont vrais. On peut donc identifier certaines classes au langage logique utilisé pour décrire leurs problèmes. Ce constat est intéressant, car il apporte des éléments de réponse à la question posée précédemment : *quelle logique utiliser pour exprimer les problèmes combinatoires?*

Une caractérisation en termes de langage logique de **NP** a été donnée par le théorème de FAGIN [105]. Celui-ci montre que **NP** coïncide exactement avec l'ensemble des problèmes expressibles dans un certain langage logique appelé *logique existentielle de second ordre* caractérisé, comme son nom l'indique, par l'utilisation de quantificateurs essentiellement existentiels (des quantificateurs universels particuliers, dits de premier ordre, sont utilisables ; mais leur importance est limitée). Ce résultat est une forme particulière de complétude : le fait qu'un problème soit **NP**-complet signifie que, pour tout autre problème **NP**, il existe un algorithme d'encodage permettant de traduire toute instance du problème initial en instance du problème **NP**-complet. Le théorème de FAGIN stipule quant à lui que, pour tout problème **NP**, il existe une formule logique existentielle de second ordre telle que les instances satisfaisant le problème sont directement caractérisées comme les *modèles* de la formule, c.-à-d. les instances la rendant vraie[27]. Un exemple classique de formule existentielle de second ordre est la suivante, qui exprime le problème de coloration de graphes (existence de 3 couleurs telles que chaque sommet se voit attribuer une couleur unique et que deux sommets adjacents ne reçoivent jamais la même couleur) :

$$\exists \mathrm{R}\ \exists \mathrm{G}\ \exists \mathrm{B} \left(\begin{array}{ll} \forall x & \left(\begin{array}{c} \mathrm{R}(x) \vee \mathrm{B}(x) \vee \mathrm{G}(x), \\ \neg(\mathrm{R}(x) \wedge \mathrm{B}(x)), \\ \neg(\mathrm{B}(x) \wedge \mathrm{G}(x)), \\ \neg(\mathrm{G}(x) \wedge \mathrm{R}(x)) \end{array} \right), \\ \forall x\ \forall y & \left(\begin{array}{c} \neg(\mathrm{Arc}(x,y) \wedge \mathrm{R}(x) \wedge \mathrm{R}(y)), \\ \neg(\mathrm{Arc}(x,y) \wedge \mathrm{B}(x) \wedge \mathrm{B}(y)), \\ \neg(\mathrm{Arc}(x,y) \wedge \mathrm{G}(x) \wedge \mathrm{G}(y)) \end{array} \right) \end{array} \right)$$

On notera que le langage utilisé dans cette formule est relativement proche du langage de programmation par contraintes présenté précédemment (OPL [247]). La même remarque vaut pour Alice [172] ; les langages de ce type sont, d'un certain point de vue, des variantes de logique existentielle de second ordre.

Les problèmes **NP** sont donc essentiellement des problèmes exprimés dans une logique *existentielle*, ce qui n'est pas surprenant : résoudre un problème **NP**, c'est trouver une solution vérifiant certaines

[27] Marco CADOLI a démontré un résultat similaire pour le problème de programmation linéaire en nombres entiers (PLI) [46] : à tout problème **NP** correspond un *modèle* (au sens de la programmation linéaire) tel que l'ensemble des données validées par ce modèle est exactement l'ensemble des solutions du problème initial. En d'autres termes, et suivant la formulation de l'auteur, "la programmation linéaire en nombres entiers est une variante syntaxique de la logique existentielle de second ordre" (grâce au symbole de somme itérée Σ qui joue un rôle similaire à la quantification universelle). Ce résultat est intéressant car il justifie la pratique de la programmation linéaire en nombres entiers : il est bien connu qu'il est possible de modéliser "de nombreux problèmes" en fournissant des modèles PLI. Les langages de programmation mathématique fournissent d'ailleurs les outils de séparation entre modèles et données permettant de réutiliser les modèles. Une des conséquences de ce résultat concerne les langages plus proches de notre pratique, comme OPL : il est clair que les constructions fournies par ce langage lui permettent d'exprimer tous les problèmes **NP**. On peut d'ailleurs trouver surprenant que la question de l'expressivité des langages de programmation par contraintes n'aie pas fait l'objet de plus d'études.

contraintes, c.-à-d. déterminer qu'*il existe* une telle solution. Les problèmes de contraintes quantifiées universellement correspondent quant à eux à la classe **coNP**, celles des problèmes dont l'absence de solution peut être prouvée de manière concise. Un exemple de problème **coNP**-complet est par exemple de déterminer si deux circuits logiques sont équivalents c.-à-d. si, pour toute entrée, ils calculent la même sortie. En fait, la différence entre les problèmes quantifiés existentiellement ou universellement doit être relativisée : invalider un énoncé existentiel, c'est prouver un énoncé universel. Ce qui compte vraiment, c'est que dans les deux types de problèmes, un sens est (encore) plus difficile que l'autre : un problème existentiel est plus facile à vérifier (en exhibant une solution) qu'à invalider (pour toute solution potentielle), un problème universel est plus facile à invalider (en exhibant un contre-exemple) qu'à vérifier (pour toute valeur potentielle). Pour se convaincre qu'un sens est plus difficile que l'autre, remarquons qu'il est toujours possible par un tirage au sort (que certains algorithmes ne se privent pas d'utiliser) d'exhiber une coloration correcte pour un graphe donné, ou d'exhiber un contre-exemple montrant que deux circuits ne sont pas identiques. Impossible en revanche, même par un tirage heureux, de garantir qu'un graphe ne peut être coloré de manière satisfaisante, ou que deux circuits sont réellement équivalents[28]. De manière significative, les instances les plus difficiles de ce type de problèmes sont les problèmes universels (par exemple les instances insatisfaisables de problèmes existentiels)[29].

Bien entendu, une quantification plus générale est nécessaire pour exprimer certains problèmes. Un premier exemple vient des problèmes d'*optimisation* sous contraintes. Ceux-ci consistent, étant donné une fonction objectif f et un ensemble de contraintes, à trouver une valeur optimale (par exemple minimisant f) satisfaisant les contraintes. En toute généralité, le problème se formule comme suit :

$$\exists x \ \text{solution}(x) \wedge \left(\ \forall y \ (\text{solution}(y) \rightarrow y \geq x) \ \right)$$

Cette formule se réécrit sous forme *prénexe*, c.-à-d. avec tous les quantificateurs en tête de formule, sous la forme $\exists x \ \forall y \ (\text{solution}(x) \wedge (\text{solution}(y) \rightarrow y \geq x))$. On a donc, en général, besoin de quantificateurs universels pour exprimer les problèmes d'optimisation, et plus précisément d'une quantification de type "$\exists\forall$". Le fait d'avoir ainsi un changement de quantificateur ($\exists$ puis $\forall$) est appelé *alternance*. La complexité d'un problème augmente en fait avec le nombre d'alternances nécessaires pour exprimer ce problème. Le type d'alternance utilisé pour décrire un problème est donc une caractéristique importante,

[28] Il est conjecturé que **NP**$\neq$ **coNP**, et que les problèmes universels n'admettent donc en général pas de preuve de longueur polynomiale. Notez que cette conjecture est en fait plus forte que **P** $\neq$ **NP** (**P** étant quant à lui clos par complémentation).

[29] Certains instances insatisfaisables de taille très réduite demeurent hors de portée des meilleurs algorithmes de résolution. Pour SAT, certaines instances étudiées par TSEITSIN peuvent mettre en échec les meilleurs résolveurs avec seulement 60-70 variables — il est vrai que le temps de calcul des algorithmes basés sur la résolution propositionnelle (qui est à la base de la plupart des résolveurs SAT) est *prouvé* exponentiel pour ce type de formules. En programmation en nombres entiers, certains problèmes à 40 variables et 5 contraintes ("sacs à dos") [71] sont réputés insolubles par n'importe quelle méthode (je tiens ces chiffres d'un exposé de Mickael TRICK de 2001 ; peut-être certaines améliorations sont-elles survenues).

Ces instances difficiles illustrent bien le contraste entre les algorithmes polynomiaux et non polynomiaux, et en particulier entre les problèmes de classe **P** et les problèmes **NP**-complets. Considérons à titre de comparaison le problème de *programmation linéaire* classique, à solutions rationnelles (c.-à-d., en levant la restriction à des solutions entières). Ce problème appartient à **P** [165, 263, 158]. Le code commercial pour ce type de problèmes permet de résoudre couramment des instances de plusieurs *millions* de variables, alors que sur des problèmes SAT aléatoires générés avec des paramètres choisis pour maximiser leur difficulté (problèmes "au seuil" de la transition de phase), la taille de 700 variables représente une limite pour les meilleurs algorithmes, et les problèmes d'une telle taille demandent un temps de calcul de plusieurs jours (de plus grosses instances peuvent toutefois être résolues dans le cas de problèmes issus de la vie réelle, y compris des instances insatisfaisables). Les premières instances difficiles à résoudre à l'heure actuelle pour la programmation linéaire ont une taille dépassant la centaine de milliers de variables [31]. Ceci représente une réelle différence d'ordre de grandeur due à la restriction aux nombres entiers, qui rend le problème **NP**-complet, et illustre le fait que les questions de complexité de calcul ne sont pas une simple curiosité théorique. Merci à Laurent SIMON pour certaines de ces données, ainsi que pour ses réponses à certaines de mes questions sur l'état de l'art concernant les résolveurs SAT.

déterminant la complexité du problème. Une classe de complexité correspond aux problèmes pouvant s'exprimer grâce à des quantificateurs de type "$\exists\forall$" (cette classe s'appelle Σ_2^p), une autre correspond aux problèmes de type "$\forall\exists$" (cette classe s'appelle Π_2^p), et on peut continuer ainsi indéfiniment, en ajoutant des alternances. L'ensemble des classes obtenues, qui correspondent aux classes de problèmes s'exprimant avec un *nombre borné d'alternances de quantificateurs*, s'appelle la *Hiérarchie Polynomiale*, également notée **PH** [240]. Il existe des problèmes complets pour chacun des niveaux de la Hiérarchie Polynomiale, et il est conjecturé que chacune des classes qui la constituent correspond à un ensemble différent de problèmes.

Mais il existe également des problèmes qui, bien qu'expressibles en termes de contraintes quantifiées, requièrent un nombre *non borné* d'alternances de quantificateurs. C'est le cas, par exemple, de certains des jeux dérivés de celui présenté en Section 1.2.1. Pour de tels problèmes, le nombre de quantificateurs alternés correspond au nombre de tours de jeu ; il est clair que si on augmente la taille du problème (c.-à-d. la largeur du plateau de jeu), le nombre de quantificateurs est également amené à augmenter. Les problèmes de logique propositionnelle quantifiée sans limite sur le nombre d'alternances correspondent à une classe de complexité naturelle et déjà mentionnée : **PSPACE**, c.-à-d. la classe des problèmes solubles par des algorithmes polynomiaux en espace. Bien que la grande majorité des travaux en combinatoire aient été consacrés aux problèmes **NP**-complets ou à des problèmes d'optimisation, les problèmes **PSPACE** possèdent de nombreuses applications, qui seront développées dans la suite de cette thèse[30].

[30] La légitimité de l'étude de problèmes d'une telle complexité n'est pas incontestable, et les raisons de se restreindre à des problèmes **NP** sont illustrées, notamment, par ces propos de PAPADIMITRIOU [212] :

> Often, [solutions] are mathematical abstractions of actual, physical objects or real-life plans that will ultimately be constructed or implemented. Hence it is only natural that in most applications the [solutions] are not astronomically large, in terms of the input data. And specifications are usually simple, checkable in polynomial time. One should therefore expect that most problems arising in computational practice are in **NP**. And in fact, they are.
>
> *Bien souvent, [les solutions] sont les abstractions mathématiques d'objets concrets et physiques, ou de constructions de la vie réelle qui seront effectivement réalisées ou mises en œuvre. C'est donc tout naturellement que, dans plupart des applications, les [solutions] ne sont pas arbitrairement grandes en termes des données du problème. Et les spécifications sont généralement simples, et peuvent être vérifiées en temps polynomial. Il est donc légitime de s'attendre à ce que la plupart des problèmes survenant en pratique soient dans **NP**. Et dans les faits, ils le sont.*

Dans un cadre différent, Mark WALLACE [257] tiendra les propos suivants, qu'on peut rapprocher de l'argumentation de PAPADIMITRIOU :

> CSP have sufficient expressive power to formalize a wide class of search problems: arguably all the interesting ones.
>
> *Les Problèmes de Satisfaction de Contraintes ont un pouvoir d'expression suffisant pour formaliser un grand nombre de problèmes de recherche : on pourrait même soutenir que cet ensemble comporte en fait tous les problèmes réellement intéressants.*

Si ces propos sont pertinents, on remarquera cependant que certains problèmes de complexité supérieure à **NP** — en particulier des problèmes **PSPACE**-complets — ont fait l'objet de recherches depuis des années (calcul de plans d'actions en intelligence artificielle, *model-checking* de formules de logique temporelle, problèmes difficiles de planification ou de gestion de stocks), et que l'intérêt de cette classe de problèmes n'est plus à démontrer. Enfin, il ne faut pas oublier que la complexité considérée ici est celle *en pire cas*, et que le décalage avec les problèmes survenant dans les applications est parfois important. Un exemple particulièrement spectaculaire est donné par le problème de typage des langages fonctionnels de type ML : bien que le problème soit **E**-complet [159](!), des milliers de programmes sont compilés et typés chaque jour dans le monde, en général en temps linéaire!

CHAPITRE 2

Plan du document et aperçu des contributions

Les logiques quantifiées présentent différentes facettes ; cette thèse se propose d'étudier plusieurs aspects de la résolution de problèmes modélisés par des formules logiques quantifiées. Le cœur du manuscrit est consacré aux problèmes de contraintes quantifiées, c.-à-d. à la généralisation du problème de satisfaction de contraintes dans lequel un nombre arbitraire de variables peut être quantifié, sans restriction sur le nombre d'alternations. Ce problème ***PSPACE****-complet permet d'envisager la résolution d'applications pour lesquels le cadre CSP classique est en général insuffisant.*

La partie II, intitulée *logique et déduction*, récapitule les notions techniques utiles à la lecture du document ; il s'agit de notions de logique (en particulier de logique quantifiée) et de certains aspects du raisonnement propositionnel :

3. Le chapitre *logique et quantificateurs* rappelle brièvement certaines notions logiques élémentaires et présente les quelques techniques moins classiques liées à la gestion des quantificateurs, comme la mise sous forme prénexe, la décomposition de formules par introduction de variables existentielles, ou les techniques d'élimination de quantificateurs ;
4. La *déduction en logique propositionnelle* est alors abordée dans le chapitre 4 ; on y trouvera des rappels sur les techniques de résolution propositionnelle et de propagation de contraintes. Les principaux résultats de complétude et de correction utilisés dans la suite du document sont également formulés et démontrés.

La partie centrale (Partie III) comporte l'essentiel des contributions de cette thèse. Le thème en est la résolution des contraintes quantifiées, pour lesquelles une adaptation de la technique d'arc-consistance est proposée. La partie III est constituée des chapitres suivants :

5. Le problème étudié est présenté dans le chapitre 5, intitulé *Extension du cadre CSP aux contraintes quantifiées*. Y est introduite la notion de QCSP, qui généralise les CSP classiques. Les notions de solution et de vérité sont décrites de manière formelle grâce à l'emploi de l'algèbre relationnelle ; ceci nous permet, d'une part, de présenter certains des algorithmes existants permettant de résoudre des contraintes quantifiées et, d'autre part, d'exhiber un paramètre permettant d'évaluer la difficulté de résolution de certains QCSP. Enfin, l'étude du problème est motivée par un survol de leurs différents domaines d'application.
6. Le chapitre 6 donne une définition de la notion d'*arc-consistance pour les contraintes quantifiées*. Partant d'une définition de la notion de consistance, on y montre comment celle-ci peut être approximée de manière locale, ce qui nous permet de formuler une notion de *consistance locale* généralisant l'arc-consistance classique aux contraintes quantifiées. L'algorithme de propagation

y est décrit en termes de calcul de point-fixe d'un ensemble d'opérateurs ; la confluence et la complexité de la technique sont étudiées.

7. Le chapitre 7 présente des *améliorations de l'arc-consistance quantifée*. Plus précisément, on y montre comment définir des propagateurs optimisés, formulés en termes de règles, pour le domaine des booléens et pour les contraintes numériques. La propagation booléenne présentée est une généralisation aux contraintes quantifiées des règles classiques de propagation booléenne ; la résolution de contraintes numériques nous permet de définir un cadre de propagation d'intervalles pour les contraintes quantifiées.
8. La partie *résolution de problèmes modélisés par des contraintes quantifiées* présente différentes applications des QCSP. On y montre comment exprimer l'existence de chemins dans des graphes dont la relation d'adjacence est exprimée par des contraintes. De nombreux problèmes de vérification et de calcul de plans d'actions sont, pour l'essentiel, des problèmes de ce type. Un prototype et des résultats expérimentaux sont également brièvement décrits.

La thèse se conclut en ouvrant une problématique de recherche plus prospective. Le sujet d'étude en est la logique existentielle de second ordre, une logique quantifiée permettant d'exprimer les problèmes **NP** :

9. Intitulée *extraction de programmes à partir de spécifications logiques*, la partie 9 tente de déterminer si un problème décrit par un énoncé en logique existentielle du second ordre admet un algorithme de résolution polynomial en temps. Basés sur un algorithme d'exécution de ces spécifications inspiré de la programmation par contraintes, nous décrivons une application des techniques de compilation de connaissances à cette problématique.

Un récapitulatif des contributions principales de cette thèse, ainsi que des diverses contributions mineures éparpillées dans le document, conclut le manuscrit (Chapitre 10).

PARTIE II

Logique et déduction

La logique permet d'exprimer des problèmes, et les algorithmes de déduction manipulant des formules logiques permettent de résoudre ces problèmes. Le langage logique comporte un petit nombre d'éléments syntaxiques, essentiellement les variables, les prédicats, les connecteurs et les quantificateurs. Nous présentons les définitions et notations formelles utilisées dans la suite de ce document.
Il existe dans la littérature consacrée à la logique de nombreuses techniques de manipulation de formules permettant de transformer des énoncés en énoncés équivalents. Les transformations sont de différents types. Certaines permettent de normaliser *la forme des énoncés et de se ramener ainsi à des sous-ensembles du langage logique possédant de bonnes propriétés. D'autres permettent de* simplifier *un énoncé jusqu'à ce qu'il soit possible de déterminer si celui-ci est vrai ou faux.*
Les techniques consacrées aux formules quantifiées sont peu documentées dans la communauté programmation par contraintes*, je propose donc un aperçu des techniques utilisables dans la modélisation de problèmes combinatoires. On trouvera enfin dans cette partie un survol des algorithmes de déduction proposés pour un sous-ensemble important du langage logique : la* logique propositionnelle.

CHAPITRE 3

Logique et quantificateurs

Cette section présente les éléments de logique utiles aux problématiques développées dans la suite du document. Les formalismes et terminologies utilisées pouvant en général varier selon les auteurs, nous fixons ici les notations que nous utiliserons. Bien entendu, nous ne pouvons prétendre effectuer un survol complet des notions élémentaires de logique et renvoyons le lecteur aux références que sont, par exemple, [231, 167, 190].
Parmi les particularités de la logique dont nous avons besoin, la principale est que nous nous restreignons à des domaines de discours finis. Cette limitation est naturelle dans les applications combinatoires, dans lesquels les différents paramètres sont bornés. On parlera donc de logique à modèles finis. *La* théorie des modèles finis *est largement documentée et fait l'objet de développements récents en rapport avec des notions calculatoires [104, 153, 98]. Nous nous contentons d'en définir les notions rudimentaires utiles pour la suite du document.*
Enfin, notre présentation accorde une importance particulière aux techniques *de manipulation des formules logiques. Il existe en effet un certain nombre de méthodes de manipulation symbolique d'énoncés logiques visant, notamment, à mettre les formules considérées sous une* forme normale *particulière plus pratique, ou à déterminer si elles sont vraies ou fausses. Nous tentons ici de présenter un catalogue des techniques intéressantes dans le cadre de la résolution de problèmes combinatoires.*

Pour motiver l'approche de la logique présentée dans cette section, nous commençons par présenter un exemple introductif qui donnera une première idée des notions abordées et des transformations utilisées. Considérons la notion suivante[1] :

> Une fonction f (sur un domaine non précisé) est appelée *préfixe* d'une fonction h s'il existe une fonction g telle que $h = f \circ g$, c.-à-d. plus précisément si :
>
> $$\exists g \, \forall x \ \left(\ h(x) = f(g(x)) \ \right)$$
>
> (lire : "il existe f tel que, pour tout x, on a $h(x) = f(g(x))$")

Cette formule suggère plusieurs remarques. Tout d'abord, g et x sont quantifiés (on qualifie de *liées* les occurrences de variables placées sous la portée d'un quantificateur) alors que les éléments h et f ne le sont pas (variables *libres*). Les variables libres jouent en général le rôle de paramètres; en fait, la formule précédente exprime une relation entre les deux variables libres f et h, qui représentent des fonctions. On pourrait nommer cette relation *préfixe*(g, f).

[1] Cette notion ne présente d'intérêt que si les fonctions considérées sont soumises à certaines restrictions, mais ceci importe peu pour la compréhension de l'exemple.

Deuxièmement, les éléments quantifiés sont de natures différentes (dans la terminologie des langages de programmation, on dira que leur *type* est différent) : f représente une *fonction* dont ont cherche à prouver l'existence, alors que x est une variable du domaine de discours. On appellera quantificateurs de *premier ordre* les quantificateurs portant sur des variables, alors que les quantificateurs portant sur des fonctions et des relations seront dits de *second ordre*.

Du point de vue de la théorie des ensembles, il est bien connu que les notions d'ensemble et de fonction sont redondantes : une relation peut être vue comme une fonction dans $\{0,1\}$, et une fonction est une relation binaire associant à chaque argument possible un résultat unique. Pour des raisons essentiellement techniques, on préférera employer la vision relationnelle et éviter tout symbole de fonction. On remplacera donc chaque fonction f par une relation F binaire[2], définie comme suit : $F(x,y)$ est vraie si et seulement si on a $f(x) = y$. La conversion entre une forme fonctionnelle et une forme relationnelle se fait alors par simple introduction d'une variable supplémentaire, quantifiée existentiellement. Nommant par exemple z le résultat de $h(x)$, la formule se récrit[3] :

$$\exists g\, \forall x\, \exists z \left(\; h(x) = z \;\; \wedge \;\; f(g(x)) = z \; \right)$$

En décomposant de plus le terme $f(g(x))$, on obtient :

$$\exists g\, \forall x\, \exists z \left(\begin{array}{l} h(x) = z \;\wedge \\ \exists y \;\left(\; g(x) = y \;\wedge\; f(y) = z \; \right) \end{array} \right)$$

c.-à-d.

$$\exists G\, \forall x\, \exists z \left(\begin{array}{l} H(x,z) \;\wedge \\ \exists y \;\left(\; G(x,y) \;\wedge\; F(y,z) \; \right) \end{array} \right)$$

On remarque dans cette formule la présence d'un quantificateur existentiel imbriqué dans une conjonction. Dans certains cas, on préfère que les quantificateurs soient tous explicitement écrits en tête de formule, certains algorithmes imposant cette restriction. Dans notre cas, la transformation est simple :

$$\exists G\, \forall x\, \exists y\, \exists z \;\left(\; H(x,z) \;\wedge\; G(x,y) \;\wedge\; F(y,z) \; \right) \tag{3.1}$$

Une telle formule est appelée *formule prénexe*. On s'est permis d'inverser l'ordre des quantificateurs pour y et z car au sein d'un bloc de quantificateurs de même type (ici existentiels) l'ordre est indifférent.

Le chapitre est découpé en deux parties : nous commençons par décrire formellement la logique utilisée, puis nous présentons un bref catalogue de techniques de transformation de formules incluant la *décomposition par introduction de variables existentielles* et la *mise sous forme prénexe* suggérées précédemment.

[2] On peut considérer que les fonctions prenant plus d'un argument en entrée prennent en fait un seul argument de type "n-uplet" mais, bien évidemment, on peut également généraliser la transformation suggérée, et remplacer chaque fonction de n arguments par une relation entre $n+1$ arguments.

[3] Cette traduction est en fait imparfaite et sert de simple illustration des concepts. De manière plus rigoureuse, une relation est *fonctionnelle* si elle associe une image unique à certains éléments de son domaine (pas nécessairement à tous). L'existence d'une fonction f s'écrit en fait sous la forme: $\exists F \;\; \forall x \; \exists! y \; F(x,y)$ — ou encore $\exists F \;\; \forall x \; (\exists y \; F(x,y) \wedge (\forall y' \; (F(x,y') \rightarrow y' = y)))$, ce qui suppose donc que le vocabulaire contienne un prédicat d'égalité.

3.1 Les logiques à modèles finis

3.1.1 Syntaxe

Nous commençons par préciser la *syntaxe* des formules logiques. La syntaxe donne, de manière inductive, les règles de construction des formules.

3.1.1.1 Logique du premier ordre

La *logique du premier ordre* est le langage comportant les *connecteurs* logiques habituels ("ou", "implique", etc.) et des *quantificateurs* portant sur les variables. Ceux-ci peuvent être aussi bien *universels* ($\forall$) qu'*existentiels* ($\exists$). Plus précisément :

Définition 1. (Logique du premier ordre)

- *Un* vocabulaire fonctionnel *est un ensemble fini de* symboles de fonctions $f_1, \dots f_t$*. Chacun de ces symboles possède une* arité *(nombre d'arguments) fixée. Les fonctions d'arité* 0 *sont appelées* constantes.
- *Un* vocabulaire relationnel *est un ensemble fini de* symboles de prédicat $P_1, \dots P_m$*, où chaque symbole* P *possède également une* arité*. On utilisera en général les majuscules pour désigner les symboles de prédicat. Au besoin, on notera l'arité en exposant du symbole (par exemple* R^1*, etc.). Les prédicats d'arité* 0 *sont appelés* variables propositionnelles*.*
- *Soit de plus* $\mathcal{X} = \{x_1, \dots, x_n\}$ *un ensemble fini de* noms de variables*. Un* terme *est soit une variable, soit une expression de forme* $f(t_1, \dots, t_k)$*, où* f *est un symbole de fonction d'arité* k *et chacun des* t *est un terme.*
- *Une* proposition atomique *(ou plus simplement* proposition*) est une expression de forme* $P(t_1, \dots, t_k)$*, où* P *est un symbole de prédicat d'arité* k *et chacun des* t *est un terme.*
- *Une* formule logique de premier ordre *est soit une proposition, soit la* conjonction *(*$a \wedge b$*) de deux formules de premier ordre (*a *et* b*), soit la* négation *(*$\neg a$*) d'une formule de premier ordre (*a*), soit une* formule quantifiée universellement *de forme* $(\forall x\ \Phi)$ *où* Φ *est une formule de premier ordre et* x *est une variable.*

Les symboles de disjonction ($\vee$), d'implication ($\rightarrow$), et de quantification existentielle ($\exists$) (d'autres symboles comme le *ou exclusif* $\otimes$ pourraient éventuellement être ajoutés à la liste) sont vus comme des abréviations définies par les équivalences suivantes :

$$\begin{array}{rcl} a \vee b & \stackrel{\text{def}}{=} & \neg(\neg a \wedge \neg b) \\ a \rightarrow b & \stackrel{\text{def}}{=} & \neg a \vee b \\ \exists x\ \Phi & \stackrel{\text{def}}{=} & \neg(\forall x\ \neg\Phi) \end{array}$$

Exemple 1. *Considérons un vocabulaire comportant le symbole relationnel* $<$*, noté de manière infixe (on écrira* $x < y$ *et non* $< (x, y)$*). La formule suivante (conjonction des "axiomes des ordres linéaires") est une formule de premier ordre :*

$$\begin{array}{lll} \big(\ \forall x & \neg(x < x)\ \big) & \wedge \\ \big(\ \forall x\, \forall y\, \forall z & (x < y \wedge y < z) \rightarrow x < z\ \big) & \wedge \\ \big(\ \forall x\, \forall y & x < y \vee x = y \vee x > y\ \big) & \end{array}$$

Il est bien connu que les termes et formules peuvent être vus de diverses manières (chaînes de caractères, arbres, graphes acycliques directs, applications, etc.), on consultera par exemple la partie I de [166] pour plus de détails sur ce sujet et sur les autres définitions précédentes. Parmi les questions "bureaucratiques" soulevées par la syntaxe logique utilisée se pose celle du parenthésage. On considérera que la négation est le symbole de priorité la plus forte, alors que les quantificateurs possèdent la priorité la plus basse. Pour le reste, on prendra soin dans la suite de ce document d'être le plus explicite possible sur la façon de lire ces formules. En guise de raccourci, on remplacera à l'occasion le symbole de conjonction par une simple virgule. De plus, on autorisera les abréviations suivantes :

$$\bigvee_{i \in 1..k} C_i \quad \stackrel{\text{def}}{=} \quad (C_1 \vee \cdots \vee C_k)$$

$$\bigwedge_{i \in 1..k} C_i \quad \stackrel{\text{def}}{=} \quad (C_1 \wedge \cdots \wedge C_k)$$

(chaque C est une formule logique) Ces symboles de conjonction et de disjonction "itérés" sont des raccourcis analogues au symbole Σ pour représenter des sommes.

3.1.1.2 Restrictions syntaxiques de la logique du premier ordre

Plusieurs types remarquables de formules de premier ordre sont obtenus en ajoutant des restrictions syntaxiques. On distingue les suivants :

- Les formules construites sans utiliser les symboles de quantificateur seront simplement appelées *formules non quantifiées*. Une formule utilisant uniquement des quantificateurs existentiels (resp. universels) est appelée *existentielle* (resp. universelle).
- Les symboles de prédicat ne possédant pas d'argument (arité 0) jouant le rôle de *variables logiques* (ou variables propositionnelles), les formules comportant exclusivement des prédicats d'arité 0 seront nommées *formules propositionnelles*.
- Une formule de premier ordre utilisant la conjonction comme seul connecteur est appelée *formule conjonctive*.
- Une formule n'utilisant pas de symboles de fonctions est appelée *formule relationnelle*.
- Notre définition de la logique du premier ordre autorise la présence de quantificateurs à des positions arbitraires, or il est parfois utile de supposer que tous ceux-ci se trouvent en tête de la formule. Une formule est dite *prénexe* si elle est de forme $Q_1 x_1, \ldots, Q_m x_m\ \Phi$, où Φ est une formule non quantifiée et chacun des Q représente un quantificateur choisi dans $\{\exists, \forall\}$.

3.1.1.3 Logique du second ordre

Nous avons vu que la logique du premier ordre est caractérisée par le type d'éléments pouvant être quantifiés, en l'occurrence les variables. Les logiques autorisant de quantifier d'autres éléments sont appelées *logiques d'ordre supérieur* [173]. La plus utile (du moins pour nos besoins) est la logique du second ordre :

Définition 2. (Logique du second ordre)

- *Une* formule logique de second ordre *est une expression de forme* $Q_1 P_1, \ldots, Q_m P_m\ \Phi$ *(*m *étant un naturel éventuellement nul), où chaque* Q *est un symbole de quantificateur parmi* $\{\exists, \forall\}$*, chaque* P *est un symbole de prédicat et* Φ *est une formule du premier ordre.*

- *Une formule de second ordre dans laquelle les seuls quantificateurs de second ordre utilisés sont existentiels est appelée formule de* logique existentielle de second ordre.

Remarquez que nous imposons d'utiliser les quantificateurs de second ordre "en tête" de la formule (forme prénexe de second ordre). Il y a en pratique assez peu d'intérêt à généraliser cette définition aux formules non prénexes.

3.1.2 Sémantique et notion de modèles finis

La vérité d'un énoncé logique dépend du sens que l'on donne aux symboles qu'il utilise, et la séparation entre syntaxe et sens est un élément important de la logique moderne. Le *domaine de discours* sur lequel les variables portent est en particulier important. Considérons par exemple l'énoncé logique suivant :

$$\forall x \; \exists m \; (2.m = x)$$

Il faudrait un anticonformisme exagéré pour donner au symbole 2 ou au symbole de multiplication des significations exotiques[4]. Il n'en subsiste pas moins un doute sur la vérité de l'énoncé : si l'on s'intéresse à des nombres *réels* ($\mathbb{R}$) cet énoncé est vrai. Mais si c'est de nombres *entiers* que l'on parle, alors il est faux puisqu'il existe des entiers impairs. Le sens d'une formule dépend donc du domaine de discours et de l'interprétation des symboles qui la constituent.

3.1.2.1 Interprétations et modèles

Ayant donné une définition syntaxique des énoncés logiques, il nous reste à en préciser le sens grâce à la notion d'*interprétation* :

Définition 3. (Structure/Interprétation)

- *Une* structure *est définie par :*
  - *Un* domaine d'interprétation $\mathbb{D}$ *dans lequel les variables et constantes sont supposées prendre leur valeur;*
  - *Une interprétation de chaque symbole de fonction f d'arité k par une fonction φ de signature $\mathbb{D}^k \rightarrow \mathbb{D}$ et, en particulier, de chaque symbole de constante c du langage par une valeur $v \in \mathbb{D}$;*
  - *Une interprétation de chaque symbole de prédicat P par une relation R sur le domaine $\mathbb{D}$, c'est à dire un sous-ensemble de $\mathbb{D}^i$, où i est l'arité de P.*

  Une structure est donc définie par le tuple $\langle \mathbb{D}, f_1, \ldots f_t, R_1, \ldots, R_m \rangle$ décrivant l'interprétation attribuée aux symboles de fonction et de relation qui la composent.

Bien entendu, toutes les interprétations ne sont pas nécessairement "correctes". Par exemple, l'interprétation sur les nombres entiers de la formule $\forall x \; \exists m \; (2.m = x)$ est fausse, alors que son interprétation sur les réels est vraie. On dira que cette deuxième interprétation est un *modèle* de la formule :

[4] Et encore, une telle pratique est courante chez les mathématiciens, qui sont des gens plutôt ouverts sur le sens des symboles. Par exemple, définissons un *anneau* comme un ensemble muni de deux opérations, notées $+$ et ., et d'une constante particulière notée 0. Le seul prérequis sur ces composants est qu'ils respectent certaines propriétés (axiomes) d'associativité, distributivité, etc. Selon le cas, il est possible de donner différentes significations à ces opérations. On peut notamment considérer que $+$, . et 0 représentent, respectivement, l'addition, la multiplication et le zéro sur les réels. On peut également considérer le domaine de calcul $\{vrai, faux\}$ et interpréter l'opération $+$ par le "ou" logique et l'opération . par le "et" logique, ou interpréter $+$ et . comme les opérations $\min$ et $\max$ sur l'intervalle $[0, 1]$. Il est même possible dans certains cas, sur le domaine de calcul $]-\infty, 0]$, d'interpréter l'opération $+$ comme $\max$ et l'opération . comme une addition!

Définition 4. (Modèle)

- *On utilisera la notation $I \models F$ pour exprimer le fait que la formule F est vraie dans l'interprétation I. On dira alors que I est un* modèle *de la formule F.*
- *Deux formules Φ et Ψ sont dites* équivalentes, *ce que l'on note $\Phi \equiv \Psi$, si tout modèle de l'une est également modèle de l'autre.*

Cette "définition" n'en est pas vraiment une puisqu'on a en fait principalement introduit une notation. On a en effet contourné les difficultés en utilisant la notion de "vérité" dans son acceptation la plus intuitive. Les livres de logique sont en général plus verbeux sur cette notion de vérité en précisant par exemple que la formule $\exists x\ \Phi$ est vraie si, en remplaçant toutes les occurrences de x dans Φ par une certaine valeur du domaine $\mathbb{D}$, on obtient une formule vraie (de même pour les autres constructions du langage logique). Une définition entièrement formelle de cette notion de vérité dans le cadre restreint des formules propositionnelles sera donnée dans la section 5.

3.1.2.2 Problèmes de satisfaisabilité et de vérification de modèles

Les énoncés logiques sont l'objet de différents problèmes calculatoires, visant à déterminer s'ils sont vrais ou faux. En fonction des paramètres fournis, les deux questions principales dont nous tenterons d'automatiser la résolution sont les suivantes :

Définition 5. (Problèmes de satisfaisabilité, vérification de modèle[5])

- *Vérification de modèle ($I \models ?F$)*
  Données *Formule F, interprétation I sur un domaine $\mathbb{D}$*
  Problème *Déterminer si I est un modèle de F*
- *Satisfaisabilité ($\exists I \models F$)*
  Données *Formule F, domaine d'interprétation $\mathbb{D}$*
  Problème *Déterminer s'il existe un modèle de F ayant pour domaine $\mathbb{D}$.*

Dans le problème de vérification de modèle, le modèle est fixé alors que dans le problème de satisfaisabilité il faut exhiber un modèle satisfaisant. La satisfaisabilité est manifestement plus difficile (voir [138]), mais la richesse du langage logique considéré (premier/second ordre, avec ou sans restrictions) est également à prendre en compte.

3.1.2.3 Hypothèses sur les formules et les domaines

Nous imposons les limitations suivantes, valables dans tout le reste du document sauf mention explicite :

Proviso 1. *On supposera que le langage logique utilisé comporte toujours le symbole de prédicat d'*égalité $(=)$. *Ce symbole est toujours interprété comme la relation binaire d'égalité sur $\mathbb{D}$, c.-à-d.comme la relation $\{(x, x) \mid x \in \mathbb{D}\}$.*

Proviso 2. *On se restreindra dans cette thèse aux problèmes dont le domaine $\mathbb{D}$ est un ensemble* fini, *dont l'ensemble des valeurs est donné explicitement.*

[5] Le terme anglais *model-checking* est en général employé dans le contexte particulier développé notamment par CLARKE, EMERSON, BRYANT et MAC MILLAN (cf. [102, 59]), qui s'applique à la vérification de programmes et représente en fait une application particulière du problème de vérification de modèles logiques.

Dans le cas du problème de vérification de modèles, on supposera que les fonctions et relations du domaine peuvent être décrites de manière explicite (liste des tuples des relations), ce qui impose un domaine fini. Cette restriction est naturelle dans le cas des problèmes combinatoires, pour lesquels les différents paramètres sont bornés. Remarquez de plus que le problème de satisfaisabilité est indécidable si on ne restreint pas le domaine de calcul à un ensemble fini donné.

3.1.3 SAT et CSP

Un sous-ensemble du langage logique particulièrement étudié est le fragment propositionnel non quantifié. On appelle souvent les formules de ce type des formules (ou problèmes) SAT car on s'intéresse en général à leur satisfaisabilité. Une telle formule est donc soit une variable propositionnelle, soit la négation, la conjonction, la disjonction, ... d'une formule SAT. Il est d'usage d'utiliser une forme particulière pour ces formules, dite *Forme Normale Conjonctive* (CNF). Une formule CNF est une conjonction de disjonctions de (négations de) variables. Plus précisément :

Définition 6. (Problème de satisfaisabilité propositionnelle en Forme Normale Conjonctive)

- *Un* littéral *est soit une variable propositionnelle, soit la négation d'une variable propositionnelle.*
- *Une* clause *est une disjonction de littéraux.*
- *Une formule CNF est une conjonction de clauses.*

Un exemple de formule CNF est $(x \vee \neg y \vee z) \wedge (\neg x \vee z)$. SAT est le problème **NP**-complet "canonique" [68], et l'intérêt pour sa résolution vient du fait qu'il permet d'exprimer les autres problèmes **NP**-complets, fréquents dans les applications combinatoires. Cependant, dans de nombreuses applications il est peu naturel de se restreindre à des domaines booléens caractéristiques de SAT. On a alors recours aux *Problèmes de Satisfaction de Contraintes*, qui sont des formules conjonctives existentielles. Plus précisément :

Définition 7. (Problème de Satisfaction de Contraintes)

Un problème de satisfaction de contraintes *(CSP) est un couple* $\langle \Phi, I \rangle$[6] *où*
- Φ *est une formule conjonctive existentielle ;*
- I *est une interprétation des symboles de relation sur le domaine* $\mathbb{D}$.

Le problème de satisfaction de contraintes *est le problème de vérification de modèle (*$I \models ?\Phi$*) sur ce type de formules.*

Une formulation légèrement différente de ces deux problèmes sera donnée dans les sections qui leur seront consacrées dans le chapitre suivant. Les deux problèmes possèdent une expressivité comparable.

3.2 Techniques de transformation de formules

Nous présentons ici un catalogue de techniques de base de transformation d'énoncés logiques. Celles-ci sont connues et suggérées dans de nombreux ouvrages de logique, mais il nous semble cependant qu'un état de l'art des techniques utilisables pour la résolution de problèmes combinatoires fait souvent défaut. Notez que toutes les techniques que nous rappelons ici sont des techniques simples, pouvant être mises en œuvre de façon efficace (les complexités obtenues sont toujours polynomiales).

[6] La formulation traditionnelle impose d'encoder également le domaine et de formuler un CSP comme un triplet $\langle \Phi, \mathbb{D}, I \rangle$, mais le domaine est ici encodé dans la structure.

3.2.1 Renommage des variables

Il est possible de *renommer* les variables d'une formule quantifiée en utilisant l'équivalence suivante :

$$Qx\ \Phi \quad \equiv \quad Qy\ \left(\ \Phi[x \leftarrow y]\ \right)$$

Où y est une variable fraîche n'apparaissant pas dans Φ, et $\Phi[x \leftarrow y]$ désigne la formule obtenue en remplaçant toutes les occurrences de x par y dans Φ, et Q désigne un quantificateur choisi parmi $\{\forall, \exists\}$. L'utilisation de cette technique et la remarque précédente nous amènent à formuler le *proviso* suivant :

Proviso 3. *Dans les formules quantifiées considérées dans cette thèse, toutes les variables seront supposées liées, et on supposera que tous les quantificateurs portent sur des variables différentes.*

3.2.2 Mise sous forme prénexe d'une formule de premier ordre

Il est possible de mettre une formule de premier ordre sous forme prénexe grâce aux équivalences suivantes (notez que les choses sont un peu simplifiées grâce au *proviso* 3) :

$$\begin{array}{rclcrcl} \neg(\forall x\ A) & \equiv & \exists x\ (\neg A) & \qquad & \neg(\exists x\ A) & \equiv & \forall x\ (\neg A) \\ (\forall x\ A) \wedge B & \equiv & \forall x\ (A \wedge B) & & (\exists x\ A) \wedge B & \equiv & \exists x\ (A \wedge B) \end{array}$$

Ces équivalences, utilisées comme des règles de réécriture (orientées de la droite vers la gauche) permettent en effet de "remonter" les quantificateurs en tête des formules. La seule subtilité lors de l'extraction des quantificateurs concerne les implications — il faut prendre en compte la négation implicite de la partie gauche de la négation :

$$\begin{array}{rclcrcl} A \rightarrow (\forall x\ B) & \equiv & \forall x\ (A \rightarrow B) & \qquad & A \rightarrow (\exists x\ B) & \equiv & \exists x\ (A \rightarrow B) \\ (\forall x\ A) \rightarrow B & \equiv & \exists x\ (A \rightarrow B) & & (\exists x\ A) \rightarrow B & \equiv & \forall x\ (A \rightarrow B) \end{array}$$

Exemple 3.1. *Ayant un symbole de prédicat Φ binaire, considérons la formule :*

$$\forall x\ \left(\ (\forall y\ \Phi(x,y)) \rightarrow \exists y\ \Phi(x,y)\ \right)$$

Pour éviter tout conflit de noms, il est possible de renommer l'une des variables (disons le y quantifié existentiellement) :

$$\forall x\ \left(\ (\forall y\ \Phi(x,y)) \rightarrow \exists z\ \Phi(x,z)\ \right)$$

On respecte ainsi le proviso *3. Il est dès lors possible d'extraire successivement les deux quantificateurs apparaissant en parties gauche et droite de l'implication :*

$$\forall x\ \exists y\ \left(\ \Phi(x,y) \rightarrow \exists z\ \Phi(x,z)\ \right)$$

$$\forall x\ \exists y\ \exists z\ \left(\ \Phi(x,y) \rightarrow \Phi(x,z)\ \right)$$

Notez le changement de quantification en partie gauche de l'implication. Ce changement est en fait dû à la négation implicite : on peut l'expliciter en récrivant la formule initiale sous la forme

$$\forall x\ \left(\ \neg(\forall y\ \Phi(x,y))\ \vee\ \exists z\ \Phi(x,z)\ \right) \quad \equiv \quad \forall x\ \left(\ (\exists y\ \neg\Phi(x,y))\ \vee\ \exists z\ \Phi(x,z)\ \right)$$

3.2.2.1 *Négation d'une formule quantifiée en forme prénexe*

La négation $\neg\Phi$ d'une formule Φ en forme prénexe peut très facilement s'exprimer sous forme prénexe. L'algorithme obtenu en appliquant les réécritures correspondant aux équivalences suivantes est linéaire :

$$\text{négation}(\forall x\ A) \quad \equiv \quad \exists x\ (\text{négation} A) \qquad\qquad \text{négation}(\exists x\ A) \quad \equiv \quad \forall x\ (\text{négation} A)$$

3.2.3 Décomposition par introduction de variables existentielles

La technique de décomposition par introduction de variables existentielles est implicite dans de nombreux contextes, mais elle est en fait rarement explicitée. Elle est basée sur la simple équivalence suivante : ayant un symbole de prédicat P et un symbole de fonction f d'arité a, la formule

$$P\left(\ x_1, \ldots, f(_{y_1,\ldots,y_a}), \ldots, x_b\ \right)$$

est équivalente à la conjonction suivante :

$$\exists z\ \left(\ z = f(_{y_1,\ldots,y_a})\ \ \wedge\ \ P(x_1, \ldots, z, \ldots, x_b)\ \right)$$

Où z est une variable fraîche, n'apparaissant pas dans le contexte de la formule. Cette technique est notamment utilisée dans les cas suivants.

3.2.3.1 SAT *et Formes Normales Conjonctives*

La mise sous forme CNF d'une formule SAT se fait par introduction de variables (la quantification existentielle peut ne pas apparaître puisque le problème est un problème de satisfaisabilité, implicitement existentiel). En fait, l'élimination est possible grâce à une petite astuce, rarement explicitée, consistant à considérer les connecteurs comme des symboles de fonction. On peut en effet associer à chaque connecteur $\wedge$, $\neg$ (on se restreint par exemple à ce jeu de connecteurs complet), une *fonction* booléenne définie comme suit (on adopte volontairement des symboles différents de ceux des connecteurs pour marquer la nuance) :

$$(X\ \overline{\wedge}\ Y = vrai)\ \ ssi\ \ X \wedge Y \qquad\qquad (\overline{\neg} X = vrai)\ \ ssi\ \ \neg X$$

Cette technique visant à faire apparaître la valeur résultat d'une relation fonctionnelle est appelé *réification* [20, 145] dans la littérature sur les contraintes. Une formule SAT peut ainsi être traduite à l'aide de ces symboles fonctionnels ; par exemple $(\neg x) \wedge y$ peut s'écrire sous la forme $((\overline{\neg} x)\ \overline{\wedge}\ y) = vrai$. Il est possible de décomposer une formule ainsi exprimée en introduisant une variable existentielle pour chacun des résultats des symboles de fonction, mais on obtient une conjonction d'*équations* entre formules booléennes. Une difficulté technique mineure vient du fait qu'il faut simuler l'égalité par des clauses. On utilisera les équivalences suivantes :

$$(\overline{\neg} X = Y) \quad \equiv \quad \bigwedge \begin{pmatrix} \neg X \vee \neg Y \\ X \vee Y \end{pmatrix} \qquad\qquad (X\ \overline{\wedge}\ Y = Z) \quad \equiv \quad \bigwedge \begin{pmatrix} \neg Z \vee X \\ \neg Z \vee Y \\ \neg X \vee \neg Y \vee Z \end{pmatrix}$$

Remarquez qu'il est en fait possible de se restreindre à des clauses portant sur trois variables (on appelle 3-SAT cette restriction du problème SAT; la réduction polynomiale présentée en fait à la base de la preuve de **NP**-complétude de 3-SAT). Cependant, si les résolveurs booléens utilisent en général une

forme CNF, la plupart acceptent des clauses de longueurs arbitraires (plus de détails sur les algorithmes de résolution de SAT et en particulier de CNF seront donnés dans le prochain chapitre).

Le fait de pouvoir considérer les connecteurs à la fois comme des symboles de prédicat et comme des symboles de fonction introduit une légère subtilité car il est possible de décomposer une formule de plusieurs manières. L'exemple suivant en donne une illustration :

Exemple 2. *Considérons la formule* $(\neg A \wedge B) \rightarrow B$*, c.-à-d.*

$$\neg(\neg A \wedge B) \vee B$$

Nous souhaitons renommer le sous-terme $\neg A$ *en* X*. Deux façons d'y parvenir sont les suivantes. Remplaçant* $\neg A \wedge B$ *par* $\exists X\ (X = \overline{\neg} A) \wedge (X \wedge B)$*, on obtient :*

$$\neg \left(\ \exists X\ (X = \overline{\neg} A) \wedge (X \wedge B) \ \right) \vee B$$

c.-à-d.

$$\neg \left(\ \exists X\ (X \vee A) \wedge (\neg X \vee \neg A) \wedge (X \wedge B) \ \right) \vee B$$

L'introduction d'une existentielle dans une sous-formule comme précédemment peut être intéressante dans certains cas, mais il faudrait une mise sous forme prénexe pour obtenir un CNF. La mise sous forme CNF se fait plus directement en introduisant directement les quantificateurs en tête; on obtient successivement :

$$\exists X \ \left(\ (X = \overline{\neg} A) \ \wedge \ (\neg(X \wedge B) \vee B) \ \right)$$

$$\exists X\ \exists Y \ \left(\ (X = \overline{\neg} A) \ \wedge \ (Y = X \ \overline{\wedge}\ B) \ \wedge \ (\neg Y \vee B) \ \right)$$

Soit finalement, la CNF suivante :

$$\exists X\ \exists Y \ \bigwedge \left(\begin{array}{c} (X \vee A) \wedge (\neg X \ \vee \ \neg A) \\ (\neg Y \vee X) \wedge (\neg Y \vee B) \wedge (\neg X \vee \neg B \vee Y) \\ (\neg Y \vee B) \end{array} \right)$$

La technique précédente se généralise aux *Formules Booléennes Quantifiées (QBF)*, c.-à-d. aux instances du problème de satisfaisabilité de formules propositionnelles autorisant une quantification arbitraire. Les résolveurs de formules de ce type apparus récemment (voir une présentation de ces résolveurs dans le chapitre 5) prennent en entrée des formules prénexes mises sous forme normale conjonctive.

3.2.3.2 Décomposition de CSP en contraintes primitives

La même technique de *décomposition par introduction de variables existentielles* est utilisée en programmation par contraintes pour compiler des contraintes numériques complexes en *contraintes primitives*. Cette approche est en général utilisée pour la partie numérique des résolveurs en domaines finis [247, 62]. Les expressions numériques utilisées dans ce type de langage sont construites à partir d'un petit ensemble d'opérateurs comportant addition, multiplication, égalité et opérateurs de comparaison, opérateurs booléens, etc. Il est possible de remplacer les expressions construites grâce à ces fonctions par une conjonction utilisant une version *relationnelle* de chacune de ces opérateurs de base. Le langage cible du résolveur peut ainsi être restreint aux contraintes $x.y = z$, $x + y = z$, $x \leq y$, etc., portant sur des variables du domaine de calcul et aux contraintes $x \vee y$, $\neg x$, etc., portant sur des variables logiques. Là encore, la même technique demeure applicable si on autorise une quantification arbitraire du problème.

Exemple 3. *La formule :*

$$\forall x\ \forall y\ \exists mod\ \exists k\ (x + y - 9.k = mod)$$

peut être décomposée en :

$$\forall x\ \forall y\ \exists mod\ \exists k\ \exists s\ \exists k'\ \bigwedge \begin{pmatrix} x+y & = & s \\ 9.k & = & k' \\ s-k' & = & mod \end{pmatrix}$$

Cette technique justifie la présentation classique des CSP comme des formules *conjonctives* : les disjonctions sont considérées comme des contraintes comme les autres connectant des variables booléennes (celles-ci peuvent notamment être issues notamment être issues de contraintes *réifiées*). Remarquez que la décomposition d'un problème s'effectue toujours en temps linéaire, et que la taille du problème généré est multipliée par une facteur constant. En revanche, la taille des domaines générés pour les variables existentielles introduites peuvent être significativement plus larges que celles des variables originales, car elles doivent prendre en compte les opérations dont elles représentent le résultat. Par exemple, supposons dans l'exemple précédent que le domaine des deux variables x et y soit $[0, 9]$; leur somme peut prendre une valeur entre 0 et 18, et la variable s introduite pour représenter leur somme doit donc avoir cet intervalle pour domaine.

Il est possible d'utiliser des versions *réifiées* des opérateurs de comparaison ($\leq$, $=$, etc.) et des opérateurs booléens pour permettre d'exprimer des disjonctions ou négations de contraintes arbitraires : une contrainte réifiée[7] est une contrainte primitive dans laquelle un argument booléen a été ajouté pour représenter la valeur de vérité de la contrainte. Cette valeur peut dès lors être utilisée dans une quelconque combinaison de contraintes booléennes.

Exemple 4. *La contrainte de* distance *entre deux variables x et y à valeurs entières, c.-à-d. $z = |x - y|$, peut être exprimée comme suit :*

$$(x \geq y \wedge x - y = z) \vee (x < y \wedge y - x = z)$$

Une alternative consisterait à utiliser la conjonction d'implications $((x \geq y) \rightarrow (x - y = z)) \wedge ((x < y) \rightarrow (y - x = z))$, qui nécessite également des manipulations booléennes de contraintes. Pour traduire la disjonction (ou les implications), il convient d'introduire des variables représentant la valeur de vérité de chacune des contraintes. on utilise donc les versions réifiées des contraintes de comparaison, de différence, et de conjonction :

$$(G \vee D) \wedge \bigwedge \begin{pmatrix} (x \geq y) & \leftrightarrow & A, & C & \leftrightarrow & (x < y) \\ (x - y = z) & \leftrightarrow & B, & D & \leftrightarrow & (y - x = z) \\ (A \wedge B) & \leftrightarrow & G, & D & \leftrightarrow & (C \wedge D) \end{pmatrix}$$

Insistons, pour conclure, sur le fait que l'introduction de quantificateurs dans un problème de contraintes quantifiées prénexe peut parfois se faire indifféremment à plusieurs positions différentes. Nous choisirons en général d'introduire les quantificateurs existentiels *en fin* du préfixe de quantificateurs car ce moyen est systématique — il convient toutefois de garder à l'esprit que cette solution peut ne pas être optimale pour certaines applications :

[7] Le terme de *contrainte réifiée*, désormais relativement standard, semble avoir été proposé dans [145]; la technique de réification a cependant été utilisée dès le début des années 1990, notamment dans les premières versions de [20].

Exemple 5. *La formule suivante* [8] *:*

$$\forall x\ \forall y\ (x^2 + y \leq 0)$$

est équivalente à

$$\forall x\ \forall y\ \exists z\ (z + y \leq 0\ \wedge\ z = x^2)$$

Et également à

$$\forall x\ \exists z\ \forall y\ (z + y \leq 0\ \wedge\ z = x^2)$$

(en revanche, pas à $\exists z\ \forall x\ \forall y\ (z + y \leq 0\ \wedge\ z = x^2)$*)*

3.2.4 Techniques d'élimination de quantificateurs

Nous regroupons sous cette dénomination plusieurs techniques permettant de supprimer certains types de quantificateurs. Pour ce qui concerne le problème de satisfaisabilité, les quantificateurs existentiels peuvent être éliminés par la technique de *skolémisation*. Pour le problème de vérification de modèle, il est possible d'éliminer les variables d'une instance par développement des quantificateurs.

3.2.4.1 Suppression des quantificateurs existentiels de premier ordre par skolémisation

La *skolémisation* consiste à expliciter les dépendances entre les variables d'une séquence de quantificateurs de premier ordre en remplaçant ces variables par des *fonctions*. Par exemple, il est clair que dans la formule :

$$\forall x\ \exists y\ (\Phi(x, y))$$

La valeur de y dépend en fait de celle de x. La variable y peut donc être remplacée par une fonction $y(x)$[9] ; on obtient une formule quantifiée universellement :

$$\forall x\ \Phi(x, y(x))$$

Plus généralement, une variable existentielle précédée de n quantificateurs universels peut être remplacée par une fonction de ces n variables. On obtient ainsi une formule universelle avec symboles de fonction dont la *satisfaisabilité* est équivalente à celle de la formule non skolémisée. En fait, les fonctions introduites sont implicitement quantifiées existentiellement, ce qu'il serait possible de rendre explicite par une quantification de second ordre[10] :

$$\exists Y\ \ \forall x\ \Phi(x, Y(x))$$

[8] Je reprends cet exemple de notes laissées par le professeur Hoon HONG, que je remercie pour le temps qu'il ma consacré lors de son passage à l'IRIN.

[9] Noter que le lange logique est alors basé sur un vocabulaire différent, autorisant notamment des symboles de fonctions.

[10] Il n'est pourtant pas tout à fait exact en général de dire que la skolémisation permet de transformer toute formule de premier ordre en une formule existentielle de second ordre dans laquelle les quantificateurs de premier ordre sont tous universels. Il semble qu'on obtienne dans le cas général des formules de forme suivante [100] :

$$\exists R_1, \ldots R_a\ \ \forall x_1, \ldots x_b\ \ \exists y_1, \ldots, y_c\ \ \Phi$$

où Φ est non quantifiée. La forme purement universelle peut toutefois être obtenue dans le cas de structures munies d'une *relation d'ordre*, ce que nous supposerons en général. D'après ce que je comprends du problème, il vient du fait que l'utilisation de symboles relationnels n'est pas totalement équivalente à l'introduction de symboles de fonctions dans ce cas précis (encore faut-il exprimer l'existence d'une image par cette fonction). Comme signalé par un relecteur anonyme de la conférence JFPLC, "l'existence d'un élément individuel ne peut en général être remplacé par l'existence d'un *ensemble*, car celui-ci peut être vide, et un quantificateur de premier ordre est nécessaire pour exprimer sa non-vacuité".

La skolémisation apporte un éclairage intéressant, peu exploré, sur les contraintes quantifiées : elle met en valeur le fait que les *variables* utilisées dans le problème sont en fait des fonctions[11].

3.2.4.2 Élimination de quantificateurs par développement

Le fait de fixer le domaine d'interprétation de formules logiques sur un ensemble *fini* donné confère un sens particulier aux quantificateurs de premier ordre : ceux-ci servent en quelque sorte de *boucles*, qu'il est possible de dérouler pour obtenir de simples conjonctions ou disjonctions.

$$\begin{array}{rcl} \forall x\ \Phi & \rightsquigarrow & \bigwedge_{v\in\mathbb{D}} \Phi[x \leftarrow v] \\ \exists x\ \Phi & \rightsquigarrow & \bigvee_{v\in\mathbb{D}} \Phi[x \leftarrow v] \end{array}$$

Il est ainsi possible, pour le problème de vérification de modèle, d'éliminer[12] un à un les quantificateurs. Dans le cas d'une formule de premier ordre, on obtient ainsi un circuit de taille $|\mathbb{D}|^n$, où n est le nombre de variables (donc de quantificateurs) du problème, qu'il suffit d'évaluer pour déterminer la valeur de vérité de la formule.

L'idée se généralise de façon particulièrement intéressante à la logique existentielle du second ordre. Une relation d'arité k sur un domaine $\mathbb{D}$ est définie par sa valeur de vérité pour chacune des valeurs des k variables sur lesquelles elle porte. Déterminer l'existence d'une telle relation revient donc à calculer la valeur de vérité de cette relation sur chacune des $|\mathbb{D}|^k$ valeurs possibles pour ces k variables. Une fois le domaine fixé, on peut donc remplacer chaque relation par un ensemble de $|\mathbb{D}|^k$ variables booléennes.

L'ensemble de variables associées à une relation R peut être vu comme un tableau de booléens à k dimensions indexé par les valeurs du domaine $\mathbb{D}$, d'où l'utilisation d'une notation avec crochets. Un quantificateur existentiel de second ordre peut donc, lorsque le domaine de calcul est fixé, être remplacé par un ensemble de quantificateurs existentiels portant sur des variables logiques. On peut alors obtenir une formule SAT par simple déroulement des quantificateurs de premier ordre :

Exemple 3.2. *Considérons la formule suivante :*

$$\begin{array}{l} \exists R^1\ \exists B^1 \\ \quad \forall x \qquad \left(\begin{array}{l} R(x) \vee B(x), \\ \neg(R(x) \wedge B(x)) \end{array} \right), \\ \\ \quad \forall x\ \forall y \quad \left(\begin{array}{l} \neg(\mathit{Arc}(x,y) \wedge R(x) \wedge R(y)), \\ \neg(\mathit{Arc}(x,y) \wedge B(x) \wedge B(y)) \end{array} \right) \end{array}$$

Une interprétation de cette formule est donnée par un domaine de calcul V et par la définition (qu'on représente par un tableau booléen) du prédicat Arc $\subseteq V \times V$). Pour le problème de vérification de modèle, les quantificateurs peuvent être éliminés, et on obtient la formule propositionnelle suivante :

$$\bigwedge_{x,y\in V} \left(\begin{array}{l} (R[x] \vee B[x]), (\neg R[x] \vee \neg B[x]), \\ (\neg \mathit{Arc}[x,y] \vee \neg R[x] \vee \neg R[y]), \\ (\neg \mathit{Arc}[x,y] \vee \neg B[x] \vee \neg B[y]) \end{array} \right)$$

[11] Une perspective intéressante, bien qu'assez prospective, concerne l'emploi d'un cadre de réduction de *fonctions* pour la résolution de problèmes de contraintes quantifiées; des rapports avec les travaux sur CLP(F) semblent par exemple envisageables [146].

[12] Par *élimination de quantificateurs* on désigne le plus souvent les techniques de preuve sur la théorie additive/multiplicative des réels (polynômes arbitrairement quantifiés). La décidabilité de ce problème de vérification de modèles a été montrée par TARSKI [244], dont la méthode de décomposition a été notamment améliorée par COLLINS (méthode de *décomposition algébrique cylindrique* [8]). Il va cependant de soi que le terme s'applique également au cas plus simple que représentent les domaines finis.

dont la satisfaisabilité détermine si l'interprétation est un modèle ou non.

Un tel "mode d'exécution" des spécifications existentielles de second ordre présente une analogie assez flagrante avec le paradigme de programmation logique par contraintes, dont le mode d'exécution consiste à construire de manière incrémentale un *store* de contraintes dont un résolveur spécialisé détermine la satisfaisabilité. Cette approche sera d'ailleurs développée dans le chapitre 9 de ce document.

CHAPITRE 4

Déduction en logique propositionnelle

Le but de cette section est de présenter les méthodes de déduction logique utilisées dans la suite de cette thèse. Nous parlerons notamment de résolution*, de* résolution unitaire *(ou* propagation booléenne*), de* recherche-élagage*, et d'*arc-consistance*. Nous présentons également les notions et résultats utiles à l'analyse théorique de ce type d'algorithmes, notamment les notions de* complétude*,* correction*,* confluence*, et certains résultats de* complexité*.*
Il est clair qu'un survol exhaustif de la littérature sur ce type de domaines (dont certains aspects sont étudiés depuis plus de quarante ans) est hors de portée d'un chapitre limité à quelques pages. Nous tentons donc de donner une présentation concise et unifiée autour d'un petit nombre de concepts centraux, les principaux étant celui de résolution *et celui d'*itération chaotique*.*

4.1 Les problèmes de satisfaction de contraintes

Le fragment propositionnel non quantifié occupe une place centrale en déduction automatique, et en particulier le problème de *satisfaisabilité* de ce type de formules, également appelé SAT. SAT est un cas particulier de problème de satisfaction de contraintes (CSP). Nous présentons successivement ces deux problèmes, et discutons les différents types d'algorithmes de résolution qui leur sont consacrés.

4.1.1 Satisfaisabilité propositionnelle

Une formule propositionnelle non quantifiée est soit une variable propositionnelle, soit la négation, la conjonction, la disjonction, ... d'une formule propositionnelle non quantifiée. Il est cependant d'usage d'utiliser une forme particulière pour ces formules, dite *Forme Normale Conjonctive* (CNF). Des techniques simples d'*introduction de variables*, présentées dans le chapitre précédent, permettent de mettre une formule SAT arbitraire sous une forme CNF. Rappelons qu'une formule CNF est une conjonction de disjonctions de (négations de) variables. Plus précisément :

Définition 8. (Problème de satisfaisabilité propositionnelle en Forme Normale Conjonctive)

- *Un* littéral *est soit une variable propositionnelle, soit la négation d'une variable propositionnelle ;*
- *Une* clause *est une disjonction de littéraux ;*
- *Une formule CNF est une conjonction de clauses.*

Les opérateurs de conjonction et disjonction étant commutatifs, on s'autorisera parfois, selon le point de vue le plus adapté, à considérer les clauses comme des *ensembles* de littéraux et les formules CNF comme des ensembles de clauses, ce qui permet de parler de l'*ajout* d'un littéral à une clause, ou de

l'*union* de deux CNF. Un exemple de formule CNF est $(x \vee \neg y \vee z) \wedge (\neg x \vee z)$. SAT est le problème **NP**-complet "canonique" [68], et l'intérêt pour sa résolution vient du fait qu'il permet d'exprimer les autres problèmes de type **NP**, fréquents dans les applications combinatoires.

4.1.2 Problèmes de satisfaction de contraintes

SAT est un exemple de *problème de satisfaction de contraintes* : résoudre à tel problème consiste à exhiber une valuation attribuant à chaque variable une valeur booléenne rendant la formule vraie. De nombreux problèmes combinatoires peuvent en fait être formulés sous la forme de problèmes de la satisfaction de contraintes. Un autre exemple fameux est celui de *coloration de graphes* introduit dans le chapitre 1, qui consiste à attribuer à chaque sommet d'un graphe une couleur choisie parmi un ensemble d'un certain cardinal, avec pour contrainte qu'aucune arête ne relie deux sommets de même couleur.

Les problèmes de satisfaction de contraintes peuvent être modélisés sous forme SAT, mais de nombreux problèmes sont plus naturellement modélisés par des contraintes sur des domaines finis de taille arbitraire (éventuellement non booléens). Les contraintes sont alors définies, de manière plus générale, comme des *relations* sur le domaine de calcul considéré. La présentation des problèmes de satisfaction de contraintes requiert donc l'introduction d'un minimum d'éléments formels concernant les notions relationnelles.

4.1.2.1 Tuples & relations

On dispose d'un ensemble $X = \{x_1, \ldots, x_n\}$ de *noms de variables*, possédant chacune un *domaine* fini, noté D_x, qui représente l'ensemble des valeurs que x est autorisé à prendre. On notera $\mathbb{D}$ l'union des D_x.

Soit $\chi \subseteq X$ un sous-ensemble des variables. Un χ*-tuple* (t) est une application de signature $\chi \to \mathbb{D}$, associant à chaque variable x une valeur (notée t_x) prise dans le domaine de x. Étant donné un sous-ensemble de variables $\chi' \subseteq \chi$, on définit la *restriction* de t à χ' (notée $t|_{\chi'}$) comme le χ'-tuple associant les mêmes valeurs que t aux variables de χ' et qui n'est pas défini sur les autres variables. Une χ*-relation* est définie comme un ensemble de χ-tuples. Un X-tuple t *satisfait* une χ-relation c si $t|_\chi \in c$.

4.1.2.2 Problèmes de satisfaction de contraintes

Un *problème de satisfaction de contraintes* (CSP) [186] est formulé comme un ensemble de relations portant sur un ensemble de variables pouvant prendre leurs valeurs dans un certain domaine de calcul, plus précisément :

Définition 9. (Problème de satisfaction de contraintes (CSP))

Un CSP est un triplet $\langle \mathcal{X}, D, C \rangle$ où :

- *$\mathcal{X}$ est l'ensemble de variables du problème ;*
- *D associe à chaque variable $x \in \mathcal{X}$ son domaine $D_x \subseteq \mathbb{D}$;*
- *$C = \{C_1, \ldots, C_m\}$ est un ensemble de contraintes, chacune étant une χ-relation sur un certain $\chi \in X$.*

Une *solution* d'un CSP est un $\mathcal{X}$-tuple satisfaisant toutes les contraintes de $\mathcal{C}$, c.-à-d. un tuple t donnant une valeur à chacune des variables et qui, pour toute χ-contrainte c de $\mathcal{C}$, vérifie $t|_\chi \in c$. Enfin, la complexité des méthodes de déduction s'exprime en fonction des paramètres suivants :

- $n = |\mathcal{X}|$ (le nombre de variables) ;
- $m = |\mathcal{C}|$ (le nombre de contraintes) ;
- $d = |\mathbb{D}|$ (la taille du domaine de calcul).

4.1.3 Méthodes de résolution

La résolution des problèmes de satisfaction de contraintes sous leurs diverses formes (problèmes de coloration, SAT et CSP notamment) a fait l'objet de recherches actives depuis plusieurs dizaines d'années, et nous ne pouvons rendre compte dans ce chapitre de la richesse et de la variété des techniques proposées au fil du temps. S'il nous est par exemple impossible de présenter les méthodes récentes basées sur des notions aussi diverses que les *backbones* [97, 269] la prise en compte d'information sur les transitions de phase ou les algorithmes récents issus de la physique théorique [206, 132, 40], (voir [144] pour un exposé grand public de ces techniques) on peut remarquer que les résolveurs les plus performants actuellement restent cependant basés sur un petit nombre de techniques fondamentales éprouvées, remontant parfois à des travaux des années 1960 [84]. On peut diviser grossièrement les méthodes proposées en deux grandes classes [1] :

Les méthodes incomplètes visent à trouver des solutions par des moyens heuristiques, sans exploration exhaustive de l'espace de recherche. Le but est ainsi de générer le plus rapidement possible une solution au problème considéré. La contrepartie est que ces méthodes ne permettent pas, en général, de détecter l'absence de solution d'un CSP[2].

Un grand nombre de méthodes entrent dans cette catégorie, comme les algorithmes génétiques ou à base de populations, la recherche Tabou ou le *recuit simulé*. Un représentant simple et caractéristique des méthodes incomplètes est par exemple GSAT [233], un algorithme qui consiste à générer des valeurs de manière aléatoire, à essayer de les corriger si elles ne sont pas satisfaisantes, et à s'arrêter dès qu'une solution est ainsi rencontrée (voir également le successeur WalkSAT [232]).

Parmi les *surveys* consacrés aux méta-heuristiques de résolution de contraintes, on pourra consulter [140], et [196] pour le cas particulier des méthodes évolutionnistes.

Les méthodes complètes consistent à parcourir ou à représenter de la manière la plus intelligente possible l'espace de recherche dans son intégralité. Là encore, le catalogue des techniques applicables est vaste, mais nous nous consacrerons principalement à un type de méthodes, celles basées sur l'*exploration* de l'espace de recherche[3].

L'exploration exhaustive étant trop coûteuse, elle est en général utilisée conjointement à des techniques de *filtrage*, permettant de supprimer certains pans de l'espace par des déductions de coût

[1] On note que la séparation entre les méthodes complètes et incomplètes est analogue à la séparation existant, dans le cas des problèmes d'optimisation, entre optimisation *locale* et *globale*.

[2] Certaines méthodes, y compris parmi les méthodes stochastiques, possèdent cependant certaines formes de propriétés de complétude, voir par exemple la notion de *complétude probabiliste asymptotique* (PAC) [149].

[3] Parmi les autres techniques, mentionnons rapidement les *diagrammes de décision binaire* [42, 43], dont une brève présentation sera donnée dans le chapitre 5, Appliqué avec succès à certains problèmes de vérification, ce type de méthode est cependant moins utilisé dans le cadre de problèmes de contraintes, d'où notre intérêt pour les techniques d'énumération.

Les techniques issues de la programmation mathématique (et en particulier la programmation en nombre entiers) ont elles aussi souvent recours à une exploration de l'espace de recherche, dans laquelle un algorithme de *relaxation* du problème (consistant typiquement à autoriser des solutions non entières et à utiliser des algorithmes de programmation linéaire, plus efficaces), est utilisé pour réduire l'espace de recherche. S'ajoute à la liste une large gamme de techniques : génération de colonnes ou de plans de coupe (permettant d'éliminer les solutions non entières), utilisation de la *programmation dynamique* [15] basée sur un stockage intensif des résultats calculés (ce qui permet d'éviter certains recalculs), etc.

polynomial ; on obtient donc des méthodes dites de *recherche-élagage*[4]. La plus utilisée pour les problèmes de satisfaction de contraintes est probablement la recherche avec maintien de l'arc-consistance (MAC)[5] [228], dont le principe reprend pour l'essentiel celui de l'algorithme de DAVIS, LOGEMANN et LOVELAND [83], ou "DLL"[6].

De nombreuses techniques peuvent être utilisées pour réduire le coût de ces méthodes. Certaines méthodes d'*apprentissage* permettent notamment à l'algorithme d'analyser les conflits survenus en cours de recherche afin de déterminer un meilleur point de backtrack [216, 123] ou d'accumuler des informations utiles au reste de la recherche (voir, e.g., [191] pour une application de ces techniques à SAT). Enfin, d'autres techniques applicables à des classes de problèmes plus spécifiques visent à casser les *symétries* du problème considéré [121], ou à prendre en compte de manière optimisée des contraintes spécifiques à un type d'applications (contraintes dites *globales* [14, 219]).

Méthodes complètes et incomplètes servent en général à des buts différents et complémentaires ; il est en général admis que les méthodes incomplètes permettent de résoudre plus rapidement les instances satisfaisables, mais cette affirmation se trouve parfois contredite, notamment sur SAT, par la dernière génération de résolveurs complets [205], considérés comme les plus performants à l'heure actuelle (voir [267] pour un *survey* des techniques utilisées de ces résolveurs SAT, basés sur l'algorithme DLL). Ajoutons de plus que les méthodes incomplètes attendent parfois des paramètres (nombre d'itération, taille des populations) qui doivent être correctement déterminés pour obtenir des performances optimales sur une application donnée. Notons, enfin, que des algorithmes hybrides permettant de cumuler les bénéfices de la recherche locale et d'heuristiques tout en conservant les propriétés de complétude ont évidemment été proposés (cf. par exemple la Recherche à Divergence Limitée[7] [143], qui permet de rendre complet un algorithme de recherche basé sur une heuristique, ou encore UNITSAT, une technique récente particulièrement prometteuse mêlant recherche locale et propagation unitaire [147]).

Dans cette thèse et en particulier dans la suite de ce chapitre, nous nous concentrons sur les méthodes complètes de satisfaction de contraintes, qui offrent donc des garanties de trouver des solutions et permettent de prouver l'insatisfaisabilité. Cette caractéristique est en effet indispensable dans le cas de contraintes quantifiées, qui formeront l'essentiel de notre propos et seront discutées dans la partie III[8]. De plus, nous nous concentrerons plus particulièrement sur les techniques de type *recherche-élagage* ; celles-ci forment en général la partie principale des résolveurs complets (sur CSP comme sur SAT), les autres techniques étant en général des améliorations spécifiques à un type d'application particulier.

[4] Branch & Prune.

[5] *Maintaining Arc-Consistency during the search*.

[6] Certains auteurs font référence à cette procédure sous le nom de *procédure de* DAVIS *et* PUTNAM (DP), ce qui est inexact. Rapellons que celle-ci ("première version" de 1960) consiste en fait à éliminer tour à tour chacune des variables du problème et relève essentiellement de la *résolution*, présentée dans la prochaine section. Certaines heuristiques proposées dans l'article préfiguraient cependant déjà assez clairement la notion de *propagation unitaire*. C'est à cause de la forte complexité en mémoire de cette première technique que la deuxième version (DLL), qui correspond directement à un algorithme de *recherche-élagage* a été proposée. Cette petite précision historique est rétablie, par exemple, dans [69]!

Rappelons par ailleurs que les deux algorithmes ont été proposés initialement pour la preuve de théorèmes de premier ordre, et que leur utilisation dans le contexte SAT, auquel on les assimile souvent, est donc un cas très particulier (voir [13] pour une réconsidération récente de "DPLL" pour la preuve de théorèmes).

[7] *Limited Discrepancy Search*.

[8] Nous devons quelque peu nuancer cette affirmation *a priori* imparable : des travaux récents ont montré que les techniques de recherche locale sont en fait applicables aux contraintes quantifiées (en l'occurrence booléennes) [120]!

4.2 Résolution

L'algorithme de *résolution*[9] est un bon point de départ pour présenter les algorithmes de recherche-élagage. Il se trouve en effet qu'à la fois les algorithmes de recherche, ceux d'élagage, et ceux d'apprentissage peuvent être vus comme des variantes de la résolution [199]. La résolution est formulée sur le problème SAT ; nous verrons dans les sections suivantes qu'il est possible de généraliser son emploi aux CSP discrets. Considérons donc une formule propositionnelle en forme CNF.

La notion de base des systèmes de résolution de contraintes est celle de *déduction* : partant d'un ensemble de contraintes $C_1 \wedge \cdots \wedge C_m$ (qui peuvent en particulier être des clauses dans le cas de SAT), on cherche à calculer de nouvelles contraintes qui sont des conséquences de la conjonction initiale. La résolution considère un cas particulier de conséquences, appelées *implicats* :

Définition 10. (Implicat)

Un implicat d'une formule CNF $C_1 \wedge \cdots \wedge C_m$ *est une clause* c *telle que* $C_1 \wedge \cdots \wedge C_m \rightarrow c$ *soit une tautologie. Il est clair que si* $(c_1 \vee \cdots \vee c_p)$ *est un implicat, toute clause formée d'un sur-ensemble des littéraux* $c_1 \ldots c_p$ *est également un implicat. Un* implicat premier *(ou* minimal*) est un implicat* $(c_1 \vee \cdots \vee c_p)$ *tel qu'aucune clause formée d'un sous ensemble de ses littéraux ne soit un implicat.*

Un implicat est une déduction d'un type particulier puisqu'elle est donc de forme *clausale* (disjonction). La notion duale existe, elle porte le nom d'*implicant* (conjonction de littéraux), mais nous n'aurons pas à la considérer dans la suite de la thèse. La résolution est une règle de déduction permettant de calculer certains implicats appelés *résolvants* :

Définition 11. (Résolvant)

Soit Φ *une formule CNF contenant deux clauses* $A \vee x$ *et* $\neg x \vee B$*. La clause* $A \vee B$ *est un implicat de* Φ*, appelé* résolvant.

Par exemple, ayant $(y \vee \neg x \vee \neg z) \wedge (\neg u \vee x)$, on déduira le résolvant $(y \vee \neg z \vee \neg u)$. Il est clair qu'un résolvant est bien un implicat, c.-à-d. que la déduction ainsi effectuée est *correcte* du point de vue logique. La résolution est une règle de déduction *complète* dans le sens où si une formule est inconsistante, il est possible de le détecter par génération successive de résolvants aboutissant sur la clause vide. En fait, la complétude de la résolution peut s'exprimer sous une forme légèrement plus forte, dont nous aurons besoin par la suite : la résolution permet en fait de générer l'ensemble des implicats premiers d'une CNF — et donc en particulier la clause vide si la formule est fausse. Je détaille ici une formulation (semble-t'il) non standard du théorème de complétude et en donne une preuve basée sur la présentation d'Armin HAKEN [137] de la preuve de complétude réfutationnelle classique.

4.2.1 Complétude de la résolution

Rappelons qu'un tuple associe à chaque variable une valeur (ici booléenne). On dit qu'un tuple est *rejeté* par une clause s'il ne possède aucune des valeurs imposées par cette clause. Par exemple, si

[9] Dans cette thèse, on parlera uniquement de résolution *propositionnelle*, on s'intéresse donc à un cas particulier de la résolution en logique du premier ordre, outil important en démonstration automatique [11] dont une certaine forme constitue également le mécanisme opérationnel de Prolog [169].

l'ensemble des variables est $\{x_1, x_2, x_3, x_4\}$, la clause $(\neg x_1 \vee x_4)$ rejette les 4 tuples suivants :

$$\langle x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 0\rangle$$
$$\langle x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0\rangle$$
$$\langle x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0\rangle$$
$$\langle x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0\rangle$$

Soit n le nombre de variables, une clause à k littéraux rejette 2^{n-k} tuples ; la clause vide (inconsistante) rejette par exemple l'ensemble des 2^n tuples de l'espace de recherche.

Proposition 1. *Partant d'un ensemble de clauses $\mathcal{C}$ portant sur les variables $x_1, \ldots x_n$, on peut obtenir par résolution un ensemble de clauses $\mathcal{C}'$ représentant la projection des solutions de $\mathcal{C}$ sur $x_2, \ldots, x_n$. En d'autres termes on a :*

Un tuple $\langle x_2 = v_2, \ldots, x_n = v_n\rangle$ est une solution de $\mathcal{C}'$ si et seulement si il existe une valeur v_1 telle que $\langle x_1 = v_1, \ldots, x_n = v_n\rangle$ est une solution de $\mathcal{C}$.

(notez que l'ordre de numérotation des variables n'importe pas et qu'il est en fait possible d'éliminer ainsi n'importe quelle variable parmi $x_1, \ldots, x_n$).

Démonstration. Définissons $\mathcal{C}'$ comme l'union

- des clauses de $\mathcal{C}$ ne portant pas sur x_1 et
- des résolvants de chaque paire formée d'une clause de $\mathcal{C}$ contenant x_1 et d'une clause contenant $\neg x_1$.

Dans le cas où un tuple t n'est pas rejeté par une des clauses de $\mathcal{C}$, il est clair que t n'est pas non plus rejeté par $\mathcal{C}'$: par correction de la règle de résolution, les clauses inférées ne rejettent que des tuples déjà rejetés par les clauses dont elles sont issues.

À l'inverse, considérons maintenant un tuple $t = \langle x_2 = v_2, \ldots, x_n = v_n\rangle$ pour lequel à la fois $t_1 = \langle x_1 = 0, x_2 = v_2, \ldots, x_n = v_n\rangle$ et $t_2 = \langle x_1 = 1, x_2 = v_2, \ldots, x_n = v_n\rangle$ sont rejetés par $\mathcal{C}$. Il existe donc deux clauses C_1 et C_2 (éventuellement identiques) de $\mathcal{C}$ rejetant respectivement t_1 et t_2. Raisonnons par cas :

- Si l'une des clauses C_1 ou C_2 ne porte pas sur x_1, alors cette clause (qui rejette à la fois les deux tuples t_1 et t_2) est conservée dans $\mathcal{C}'$ (qui rejette donc bien t) ;
- Sinon, soit l_i le littéral exprimant le rejet de la valeur $x_i = v_i$ ($i \in 2..n$). En d'autres termes, l_i est défini comme le littéral x_i si $v_i = 0$ et comme le littéral $\neg x_i$ si $v_i = 1$.
  - C_1 est alors de forme $x_1 \vee R_1$ et C_2 de forme $\neg x_1 \vee R_2$, où R_1 et R_2 sont tous deux des sous-ensembles des l_i.
  - Le résolvant obtenu $(R_1 \vee R_2)$ est donc un sous-ensemble des l_i ; il est clair que cette clause de $\mathcal{C}'$ rejette le tuple t.

□

Corollaire 1. *La résolution est complète du point de vue réfutationnel.*

Démonstration. Si une formule est inconsistante (tous les tuples sont rejetés), il est possible d'éliminer chacune de ses variables par projections successives jusqu'à obtenir la clause vide. □

Corollaire 2. *Tout implicat premier d'une formule propositionnelle en forme normale conjonctive peut être déduit par résolution.*

Démonstration. Soit $c = (l_k \vee \cdots \vee l_n)$ un implicat premier d'une conjonction de clauses $\mathcal{C}$, où chaque littéral l_i porte sur la variable x_i (là encore le choix de numéroter les variables concernées de k à n est une simple facilité de notation qui n'entraîne aucune perte de généralité). Définissons $\mathcal{C}'$ comme la projection de $\mathcal{C}$ sur $x_k, \ldots, x_n$ ($\mathcal{C}'$ est donc obtenue par une séquence de résolutions qui élimine successivement les variables x_1 à x_{k-1}).

L'ensemble des tuples rejetés par $\mathcal{C}'$ est le singleton $\{\langle x_k = v_k, \ldots, x_n = v_n\rangle\}$ avec $v_i = 0$ si l_i est de forme x_i, et $v_i = 1$ si l_i est de forme $\neg x_i$. En effet :

- ce tuple est clairement rejeté par l'implicat ;
- supposons qu'un tuple attribuant une valeur différente à l'une des variables x_i $(i \in k..n)$ soit rejeté. Alors la clause obtenue en supprimant l_i de c serait un implicat, ce qui viole l'hypothèse de minimalité de c.

$\mathcal{C}'$ est donc constitué uniquement de la clause c (et éventuellement de clauses subsumées par c). En effet, la seule manière d'exprimer le rejet d'un tuple unique par une conjonction de clauses est d'utiliser une clause unique formée de littéraux exprimant la négation de chacune des valeurs du tuple rejeté. □

Le résultat de complétude permet de définir un premier algorithme permettant de déterminer la satisfaisabilité d'une formule SAT par utilisation de la règle de résolution. Celui-ci consiste à générer l'ensemble des résolvants de la formule ; si la formule est inconsistante, on produira la clause vide. Sinon, l'algorithme termine en temps fini (exponentiel) puisque l'on peut générer au plus 3^n clauses sur un ensemble de n variables (chaque variable pouvant être soit absente d'une clause, soit présente positivement, soit présente négativement) ; tester si une clause a déja été générée peut s'effectuer, par exemple, de manière naïve par un parcours des clauses existantes, on obtient une complexité de $\mathcal{O}((3^n)^2)$. Notez que l'algorithme est également exponentiel en mémoire, puisqu'on conserve l'ensemble des clauses générées.

Algorithme *Calcul_resolvents*

tant que Φ contient deux clauses résolvables $A \vee x$ et $\neg x \vee B$ non marquées **faire**
  $\Phi \leftarrow \Phi \vee (A \vee B)$
  Marquer le couple $(A \vee x, \neg x \vee B)$
fin tant que

Fin Algorithme

4.2.2 Recherche et élagage

La résolution est un système de preuve, c.-à-d. une règle de calcul qui spécifie un algorithme *non-déterministe* : il est ainsi possible d'appliquer la règle de calcul suivant différentes stratégies. De nombreuses formes de raisonnement propositionnel peuvent être vues comme des cas particuliers (variantes déterministes) de la résolution.

4.2.2.1 Chaînage avant et propagation unitaire

Le *chaînage avant* est une technique de déduction opérant sur le cas particulier de clauses écrites sous la forme d'implications (également appelées *règles*[10] ou *clauses de Horn*) de forme :

$$a_1 \wedge \cdots \wedge a_p \ \rightarrow c$$

où chacun des a est une *prémisse* de la règle et c est la *conclusion*. Le chaînage avant consiste à inférer la conclusion de chacune des règles dont toutes les prémisses sont vérifiées, par exemple à inférer z si x et y sont vrais et s'il existe une règle $(x \wedge y \rightarrow z)$. En fait, le chaînage avant représente un cas très particulier de déduction, qui ne tient pas compte du fait que $x \wedge y \rightarrow z$ est équivalent à $\neg x \vee \neg y \vee z$. Par exemple, si z est faux et y est vrai, il est correct de déduire que x est faux, or ce raisonnement ne relève pas du chaînage avant puisque le littéral déduit n'est pas à droite de l'implication. Il est donc possible de formuler un type de raisonnement propositionnel plus général, non restreint aux clauses de *Horn*, nommé *propagation unitaire*. La règle de déduction ainsi formulée se trouve être un cas particulier de résolution :

Définition 12. (propagation unitaire)
La propagation (ou résolution) unitaire *correspond au cas particulier de résolution dans lequel l'une des deux clauses utilisées est réduite à un littéral. En d'autres termes, on infère un résolvant A si la formule CNF considérée contient soit deux clauses x et $\neg x \vee A$, soit deux clauses $\neg x$ et $x \vee A$.*

Notez que le fait d'inférer un littéral x ou $\neg x$ peut être vu alternativement comme le fait d'attribuer la valeur *vrai* ou *faux* (respectivement) à la variable considérée (x). La propagation unitaire est donc essentiellement un mécanisme de *réduction* du domaine des valeurs possibles des variables (déduire un littéral x (resp. $\neg x$) signifie en effet que "x ne peut valoir 0 (resp. 1)").

La propagation unitaire n'est pas une règle de déduction complète : elle ne permet pas, par exemple, de déduire que la formule $(\neg x \vee \neg y) \wedge (x \vee y) \wedge (\neg y \vee \neg z) \wedge (y \vee z) \wedge (\neg z \vee \neg x) \wedge (z \vee x)$, qui impose à trois valeurs booléennes x, y et z de prendre des valeurs deux à deux différentes, est inconsistante. En revanche, il est possible de mettre en œuvre la propagation unitaire de manière très efficace par l'algorithme suivant :

Algorithme *propagation_unitaire*

$q \leftarrow \{$littéraux initialement vrais$\}$
tant que $q \neq \emptyset$ **faire**
  % *choix arbitraire d'un littéral l :* %
  $q \leftarrow q - \{l\}$
  pour tout c : clause, $t.q.\ \neg l \in c$ **faire**
    $c \leftarrow c - \{l\}$
    si (Nb_littéraux$[c] = 1$) **alors**
      Soit $c = \{$unique_littéral$\}$
      $q \leftarrow q \cup \{$unique_littéral$\}$
    fin si
  fin pour
fin tant que

Fin Algorithme

[10] De telles règles seront utilisées dans le chapitre 7, elles fournissent en effet un outil pratique permettant d'exprimer certains systèmes de déduction.

Grâce à l'utilisation pour chaque clause d'un compteur représentant le nombre de littéraux de cette clause (test de singleton en $\mathcal{O}(1)$), à une représentation des littéraux de chaque clause par double chaînage (suppression d'un littéral en temps $\mathcal{O}(1)$) et au calcul préalable de l'ensemble des clauses dans laquelle chaque littéral apparaît ; il est possible pour chaque littéral inféré de supprimer toutes les occurrences de négations de ce littéral en temps $\mathcal{O}(1)$ pour chaque suppression. L'algorithme obtenu est linéaire par rapport à la longueur de la formule (somme du nombre de variables de chacune des clauses) [265]. La propagation unitaire est de plus complète pour les cas particuliers Horn-SAT/anti-Horn-SAT (clauses ayant au plus un seul littéral positif/négatif) et 2-SAT (clauses contenant au plus deux littéraux), et elle fournit un algorithme optimal de résolution de ce type de formules[11].

4.2.2.2 Recherche-élagage

La propagation unitaire n'est pas un mécanisme complet, mais elle permet, au prix d'un coût calculatoire faible, d'obtenir certaines déductions permettant de simplifier le problème considéré. Il est donc possible, dans un algorithme complet de recherche de solution, d'utiliser une propagation de contraintes à chaque branche de l'arbre pour limiter le nombre de branches réellement parcourues ; on obtient un algorithme de *recherche-élagage*, dont une version typique est donnée par l'algorithme de DAVIS, LOGEMANN et LOVELAND, ou DLL :

Algorithme *DLL* ($C_1 \wedge \cdots \wedge C_m$**)**

Propagation_unitaire ($C_1 \wedge \cdots \wedge C_m$)
si (inconsistance détectée) **alors**
  retourner *faux*
fin si

si (il existe une variable x non instanciée) **alors**
  retourner DLL($C_1 \ldots C_m \wedge x$) $\vee$ DLL($C_1 \ldots C_m \wedge \neg x$)
sinon
  retourner *vrai* (1)
fin si

Fin Algorithme

Notez qu'il est correct de retourner *vrai* (cf. ligne (1) dans l'algorithme) lorsque toutes les variables sont instanciées car la propagation unitaire aurait détecté une inconsistance si l'évaluation retournait *faux*. Il se trouve par ailleurs, comme remarqué dans [199], que la partie *recherche* de l'algorithme de recherche-élagage est elle-même un cas particulier de l'algorithme général de résolution.

4.3 Propagation pour les CSP

Les techniques de satisfaction de contraintes peuvent être vues comme des adaptations et des généralisations des techniques SAT. Il est clair, par exemple, qu'il est possible de généraliser l'algorithme

[11] Il y a donc asse peu d'intérêt, dans le cas propositionnel, à se restreindre à la technique de chaînage avant : celle-ci est plus restrictive, et l'algorithme optimal de chaînage avant revient de toute façon à l'algorithme de propagation décrit précédemment.

DLL à des domaines non booléens, à condition de définir une technique de propagation sur les domaines arbitraires. La première de ces techniques de propagation est l'*arc-consistance* (ou AC), et l'algorithme de recherche-élagage obtenu est appelé *Maintaining Arc-Consistency*, ou MAC [228]. Nous présentons ici seulement la partie propagation, c.-à-d. l'arc-consistance.

4.3.1 Arc-consistance

L'arc-consistance est souvent étudiée dans le cas particulier des contraintes *binaires*, auquel nous nous restreignons momentanément.

4.3.1.1 Contraintes binaires

Une contrainte binaire portant sur les variables x et y est un ensemble de tuples t de forme $\langle t_x, t_y\rangle \in \mathbb{D}^2$. Un exemple de contrainte binaire est la suivante (relation $x < y$ sur le domaine $1..4$) :

$$C_{x,y} \;=\; \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 2,4\rangle, \langle 3,4\rangle\}$$

De même que pour les contraintes booléennes, le but des techniques de propagation sur les domaines finis est de déduire que certaines variables ne sont pas autorisées à prendre certaines valeurs. Si, par exemple, il est établi que y ne peut pas prendre la valeur 4, on peut déduire que x ne prendra pas la valeur 3, car $t = \langle 3,4\rangle$ est le seul tuple de la contrainte pour lequel $t_x = 3$, or ce tuple impose à y de prendre la valeur 4. On dit que $y = 4$ est un *support* de $x = 3$. L'ensemble des supports d'une valeur v_x pour une contrainte $C_{x,y}$ est l'ensemble des valeurs v_y telles que $\langle t_x = v_x, t_y = v_y\rangle \in C_{x,y}$. Il est clair que, si tous les supports d'une valeur v_x sont éliminés, alors on peut déduire $x \neq v$.

Un tel raisonnement est une forme de propagation unitaire : soit $\{a_1, \ldots a_s\}$ l'ensemble des valeurs de y supportant $x = b$; la clause :

$$\neg(y \neq a_1) \vee \cdots \vee \neg(y \neq a_s) \vee (x \neq b)$$

exprime le fait que, si y n'est autorisé à prendre sa valeur dans aucun des supports de $x = b$, alors on peut supprimer b du domaine de x. L'arc-consistance est donc un raisonnement propositionnel dont les variables représentent le fait qu'une certaine valeur est exclue du domaine d'une certaine variable.

En effectuant une conversion explicite d'un problème CSP binaire en SAT grâce à l'encodage ainsi suggéré[12], on obtient une formule SAT de taille $\mathcal{O}(md^2)$, où m désigne le nombre de contraintes et d la taille du domaine. La propagation unitaire étant linéaire, on en déduit que l'arc-consistance peut être calculée en $\mathcal{O}(md^2)$. Cette complexité est réputée optimale[13]. Bien entendu, de nombreux algorithmes ont

[12] Cet algorithme optimal d'arc-consistance a été proposé par KASIF [162], qui a également utilisé la réduction mutuelle Horn-SAT/CSP pour établir la **P**-complétude du problème d'arc-consistance. L'utilisation de résolveurs SAT comme moteurs d'arc-consistance a par ailleurs été reconsidérée récemment dans [119, 22], qui présente des expérimentations avec le résolveur Chaff [205].

[13] Je n'ai jamais lu de preuve *formelle* de cette borne inférieure de complexité. Un argument est en tout cas que $\mathcal{O}(md^2)$ est le nombre de cases de la représentation du problème par matrices d'adjacence, et on a peine à concevoir un algorithme qui, dans le pire cas, n'aurait pas besoin de visiter au moins une fois chacune de ces cases. Plus généralement, il semble que certains des arguments d'optimalité des techniques de consistance soient à prendre avec des pincettes. Il est par exemple affirmé dans [70] que l'algorithme proposé, de complexité exponentielle en k, *possède une complexité pire cas optimale en temps et en espace quand k est fixé à une constante*. Cette optimalité n'est vraie que sous certaines hypothèses puisque personne, me semble-t'il, n'a jamais démontré de borne inférieure super-linéaire de complexité pire cas pour un problème **NP**. Par ailleurs, certains résultats récents suggèrent que les complexités pire cas de certains algorithmes peuvent parfois être améliorées de manière assez subtile, voir par exemple la récente borne 1.481^n pour 3-SAT (et $(2 - \frac{2}{(k+1)})^n$ pour k-SAT en général) [76], atteinte par certains algorithmes de recherche locale.

été proposés pour obtenir cette borne optimale tout en évitant la construction des structures de données de taille $\mathcal{O}(md^2)$. Une telle consommation d'espace, qui est la même que celle d'AC4 [200], peut être pénalisante dans le cas où le problème n'est pas lui-même exprimé en extension et que sa formulation est de taille inférieure à $\mathcal{O}(md^2)$. Parmi les algorithmes de référence de filtrage par arc-consistance, citons la série AC5 - AC7 [249, 24, 25], ainsi que les deux variantes récentes de l'algorithme original AC3 [264, 27] de complexité optimale. Nous reviendrons sur AC3 en Section 4.3.2.

4.3.1.2 Généralisation

L'arc-consistance n'est évidemment pas la seule forme de propagation imaginable, et des techniques similaires peuvent par ailleurs être proposées pour des contraintes de dimension arbitraire (c.-à-d., non nécessairement binaires). Parmi les références sur les contraintes non binaires, citons les travaux sur l'arc-consistance et le *forward-checking* "généralisés" [26, 23], ainsi que le cadre élégant et général des *consistances relationnelles* [87]. Pour ce qui concerne l'arc-consistance, les généralisations non binaires proposées par ces différents auteurs sont pour l'essentiel identiques. L'idée de lever la restriction à des contraintes binaires n'est en fait pas récente et des références antérieures peuvent être mentionnées, notamment [185, 201].

Définition 13. (hyper-arc-consistance)
Une valeur $v \in D_x$ d'une variable x est (hyper-)arc-consistante par rapport à une χ-contrainte c s'il existe un χ-tuple t, appelé support, *vérifiant :*

$$t_x = v \qquad t \text{ prend } v \text{ pour valeur sur } x$$
$$\forall y \in \chi \;\; t_y \in D_y \qquad \text{les valeurs attribuées aux autres variables sont autorisées}$$

La règle de réduction par arc-consistance du domaine D_x par rapport à une contrainte c consiste à éliminer de D_x les valeurs non arc-consistantes par rapport à c.

Le rôle d'une application de la règle d'arc-consistance est d'éliminer de $\mathbb{D}^n$ certains points ne satisfaisant pas la contrainte considérée (et qui, *a fortiori*, ne seront donc pas solutions du CSP). Suivant le point de vue développé dans [6], on peut donc voir chaque règle ainsi formée comme un *opérateur de réduction*, de signature $P(\mathbb{D}^n) \rightarrow P(\mathbb{D}^n)$ vérifiant les propriétés suivantes :

correction : aucun tuple satisfaisant la contrainte considérée n'est supprimé par application de l'opérateur ;

contractance : l'ensemble de points est réduit (non strictement : dans le pire cas il peut rester inchangé) par application de la règle de l'opérateur.

D'autres opérateurs de réduction, correspondant notamment à d'autres méthodes de propagation ou à d'autres techniques plus générales encore, peuvent ainsi être modélisés par de tels opérateurs. Nous présenterons par exemple également dans cette thèse (chapitre 7) certaines techniques de propagation d'*intervalles*, qui rentrent également dans le cadre présenté. L'application de l'ensemble d'opérateurs associé à un CSP consiste à appliquer l'ensemble des opérateurs jusqu'à ce que plus aucun d'entre eux ne permette de réduire les domaines considérés. Les domaines ainsi réduits correspondent donc à un certain *point fixe* de l'ensemble d'opérateurs. Nous montrerons en section 4.3.2 que ce point est unique, et donc indépendant de l'ordre d'application des opérateurs.

Notons pour conclure sur les contraintes binaires et n-aires que la tradition de se restreindre au cas binaire est en général justifiée par la possibilité de conversion binaire des formulations non binaires ; cette conversion est cependant en général irréaliste. Les différents encodages binaires et leurs avantages respectifs en termes de force de filtrage et de complexité sont étudiées de manière très complète dans [10].

4.3.2 Confluence des algorithmes de propagation

Certains des algorithmes présentés dans cette thèse, et en particulier les algorithmes de propagation, sont décrits de manière non-déterministe : ils consistent à appliquer en boucle un ensemble de règles de calcul, jusqu'à ce que plus aucune de celles-ci ne puisse être appliquée. On qualifie de *confluents*[14] les algorithmes non-déterministes dont le résultat est caractérisé de manière indépendante de l'ordre d'application des règles de calcul. Issues notamment de travaux de P. COUSOT sur l'analyse statique de programmes et l'*interprétation abstraite* [74, 75] et de travaux antérieurs de TARSKI [245], les *itérations chaotiques* de K. APT [6] fournissent un cadre pratique pour prouver la confluence de nombreux algorithmes non-déterministes, et notamment de diverses variantes de propagation de contraintes. L'un des intérêts de ce cadre est qu'il est générique : il suffit de démontrer des propriétés simples de monotonie et d'expansivité pour que les conditions d'application de la preuve soient remplies.

Dans cette section, nous présentons les résultats de confluence utiles pour la suite de la thèse. Nous les formulons de manière générale et aussi indépendante que possible du cadre considéré précédemment : nous nous placerons sur un domaine de calcul arbitraire que nous noterons $\mathbb{D}$ (sans ambiguité pour cette section), et utiliserons une relation d'ordre noté $\sqsubseteq$. Les résultats s'appliquent aux opérateurs de réduction considérés précédemment, et en particulier à l'arc-consistance, auquel cas le domaine considéré est celui des parties de l'espace de recherche, et la relation d'ordre est l'inclusion $\supseteq$. Notre présentation de la preuve de confluence est moins générale que celle donnée par APT puisque restreinte au cas particulier qui nous intéresse, celui des domaines *finis* ; elle est plus proche de celle donnée dans [153].

4.3.2.1 Théorèmes de confluence

Soit $\mathbb{D}$ un ensemble fini muni d'une relation d'ordre partiel $\sqsubseteq$ (réflexive, transitive, anti-symétrique). On suppose de plus que tout sous-ensemble (fini) E d'éléments de $\mathbb{D}$ possède *borne supérieure*, notée $\bigsqcup E$ (on dit que $(\mathbb{D}, \sqsubseteq)$ est un *sup-demi-treillis*).

Définition 14. (expansivité, monotonie d'un opérateur)
Une fonction $f : \mathbb{D} \to \mathbb{D}$ est:

expansive	*si*	$\forall v \in \mathbb{D}$	$f(v) \sqsubseteq v$
monotone	*si*	$\forall v \in \mathbb{D}\ \forall v' \in \mathbb{D}$	$v \sqsubseteq v' \ \rightarrow\ f(v) \sqsubseteq f(v')$

Soit Φ une fonction monotone expansive sur un domaine $\mathbb{D}$ de cardinal d *fini*. On appelle *point fixe* de Φ un élément $x \in \mathbb{D}$ tel que $\Phi(x) = x$.

Proposition 2. ***(KNASTER-TARSKI)***

Soit une valeur $v \in \mathbb{D}$, dite valeur initiale. *Pour tout $t \geq d$, on a $\Phi^t(v) = \Phi^d(v)$; de plus, l'élément $\Phi^d(v)$ ainsi caractérisé est le plus petit point fixe supérieur à v de la fonction Φ.*

Démonstration. Considérons la séquence :

$$v \sqsubseteq \Phi(v) \sqsubseteq \Phi^2(v) \sqsubseteq \cdots \sqsubseteq \Phi^d(v)$$

Les inclusions sont dues au fait que Φ est expansive. Il est clair que cette séquence se stabilise : supposons au contraire que pour chaque $t \in 1..d$ on ait $\Phi^{t-1}(v) \neq \Phi^t(v)$; on a alors exhibé une séquence de $d+1$

[14] Le terme *fonctionnel* est également correct, puisque la relation entre entrée et sortie de l'algorithme se trouve être une fonction (image unique).

éléments différents de $\mathbb{D}$, qui est de cardinal d. De plus, ayant pour un certain t l'égalité $\Phi^t(v) = \Phi^{t+1}(v)$, il est trivial de voir que la suite restera stable au rang $t+2$ puisque $\Phi^{t+2}(v) = \Phi(\Phi^{t+1}(v)) = \Phi(\Phi^t(v)) = \Phi^{t+1}(v) = \Phi^t(v)$.

$\Phi^d(v)$ est donc un point fixe de Φ, et il est également supérieur à v. Soit $v' \sqsupseteq v$ un autre point fixe de Φ ; pour montrer que $\Phi^d(v) \sqsubseteq v'$, on va en fait montrer plus généralement que $\Phi^t(v) \sqsubseteq v'$, pour tout t :

- On a bien $\Phi^0(v) = v \sqsubseteq v'$
- Supposant que $\Phi^t(v) \sqsubseteq v'$, on conserve bien l'inégalité au rang $t+1$. Par monotonie, on a en effet :

$$\Phi^{t+1}(v) = \Phi(\Phi^t(v)) \sqsubseteq \Phi(v') = v'$$

□

Le théorème de KNASTER-TARSKI donne un résultat d'existence de point fixe pour une fonction unique, mais il se généralise au cas de l'itération chaotique d'un nombre fini de fonctions. Supposons maintenant qu'on dispose d'un ensemble fini $F = \{f_1, \ldots, f_k\}$ de fonctions expansives monotones de signature $\mathbb{D} \to \mathbb{D}$.

Définition 15. ***(Itération chaotique)***

- *On appelle* plan d'itération *une séquence infinie de numéros d'opérateurs choisis dans* $1 \ldots k$*, vérifiant une propriété d'*équité *: chaque numéro d'opérateur apparaît un nombre de fois infini dans la séquence.*
- *L'*itération chaotique *partant d'une valeur* $v \in \mathbb{D}$ *et basée sur la séquence d'itération* $i_1, i_2, \ldots$ *est la séquence infinie de valeurs* $d_0, d_1, \ldots$ *définie comme suit :*

$$\begin{aligned} v_0 &= v \\ v_{t+1} &= f_{i_{t+1}}(v_t) \end{aligned}$$

Le t-ème élément de la suite est donc obtenu en appliquant à l'élément précédent le t-ème opérateur du plan d'itération. Une itération chaotique se *stabilise* si sa valeur ne change plus à partir d'un certain temps, c.-à-d. s'il existe un certain t à partir duquel l'itération vérifie :

$$\forall t' > t,\ v_{t'} = v_t$$

On appelle v_t la *limite* de l'itération. Il est clair que la limite d'une itération est un point fixe de chacun des f. En effet, la condition d'équité signifie que pour chaque opérateur $f \in F$, il existe un rang $t' > t$ pour lequel l'opérateur f est appliqué ($v_{t'} = f(v_{t'-1})$) ; or la stabilité prouve que $v_t = v_{t'-1}$ est un point fixe de f.

Proposition 3. *Toute itération chaotique se stabilise en un nombre fini d'étapes. De plus, posant :*

$$\Phi(x) = \bigsqcup_{i \in 1..k} f_i(x)$$

la limite de l'itération est $\Phi^d(v)$.

Démonstration. Notez que Φ est bien monotone et expansive. La stabilité est due, comme précédemment, à la finitude de $\mathbb{D}$: si l'itération ne se stabilise pas, on peut exhiber une séquence infinie de valeurs strictement croissantes de $\mathbb{D}$. La proposition est une simple conséquence des trois constats suivants :

- La limite de l'itération est un point fixe de Φ car elle est un point fixe de chacun des $f \in F$;
- La limite de l'itération est supérieure à v car chacun des opérateurs de F est expansif ;
- La limite de l'itération est inférieure à $\Phi^d(v)$ car, plus généralement, on a $\forall t\ v_t \sqsubseteq \Phi^t(v)$. Prouvons ce résultat par induction :
  - $v_0 = v = \Phi^0(v)$
  - Ayant $v_t \sqsubseteq \Phi^t(v)$, l'élément suivant de l'itération v_{t+1} est obtenu par application d'un certain opérateur $f \in F$. On a (en utilisant, notamment, la monotonie de Φ) :

$$v_{t+1} = f(v_t) \sqsubseteq \Phi(v_t) \sqsubseteq \Phi(\Phi^t(v)) = \Phi^{t+1}(v)$$

Ces trois constats caractérisent bien le plus petit point fixe supérieur à v de l'opérateur Φ. □

4.3.2.2 Algorithme d'itération chaotique

Disposant d'un ensemble d'opérateurs $f_1, \ldots f_k$ et partant d'un point initial I, un algorithme simple calculant le point obtenu en fin d'itération consiste à appliquer chacun des opérateurs tour à tour jusqu'à ce que plus aucun changement ne soit obtenu, auquel cas le point fixe est calculé. On peut très souvent faire mieux grâce à la propriété suivante, vérifiée par tous les opérateurs considérés dans cette thèse :

Définition 16. (idempotence)
Un opérateur f est idempotent s'il est inutile de l'appliquer plusieurs fois, c.-à-d. si $f(x) = f(f(x))$, pour tout x.

L'idempotence permet, une fois qu'un opérateur a été appliqué, de ne pas le reconsidérer immédiatement. On dispose par ailleurs en général d'une relation de *dépendance* entre opérateurs : il est possible de calculer statiquement (au préalable au lancement de l'itération chaotique) l'ensemble $\text{Deps}(f)$ des opérateurs devant être reconsidérés lorsque f a été appliqué avec succès. Dans le cas de l'arc-consistance, il est par exemple clair qu'une χ-contrainte n'a pas besoin d'être reconsidérée tant que le domaine d'une des variables de χ n'a pas été modifié. Dans ce cas, l'ensemble des dépendances d'un opérateur réduisant le domaine d'une certaine variable x est l'ensemble des opérateurs dépendant de cette variable. L'idempotence et la disposition d'une relation de dépendance permettent d'utiliser l'algorithme de propagation suivant (v est le point initial de l'itération) :

Algorithme *Itération_chaotique* ($f_1, \ldots, f_k$, v)

$Q \leftarrow \{f_1, \ldots, f_k\}$
tant que $Q \neq \emptyset$ **faire**
  $Q \leftarrow Q - \{f\}$ % *choix arbitraire d'un opérateur* %
  $v' \leftarrow f(v)$
  si $(v' \neq v)$ **alors**
    $v \leftarrow v'$
    $Q \leftarrow Q \cup \text{Deps}(f)$
  fin si
fin tant que

Fin Algorithme

La structure de *file* est en général choisie pour implémenter l'ensemble Q. L'algorithme est appelé AC3 [186]. Il est d'usage relativement général et peut, en particulier, être choisi lorsque les optimisations propres aux contraintes binaires ne sont pas applicables. On remarquera par ailleurs de fortes similarités avec l'algorithme de propagation booléenne vu précédemment.

4.4 Remarques

Les algorithmes de propagation de contraintes représentent des méthodes de déduction incomplètes permettant d'obtenir, à un coup calculatoire raisonnable, certaines réductions de l'espace de recherche du problème considéré. Intégrées dans un algorithme de recherche-élagage, elles permettent de définir un algorithme complet mais limitant en partie les problèmes d'explosion combinatoire constatés lors de l'emploi d'une énumération naïve. En fait, cette explosion combinatoire ne peut être que partiellement évitée, et seulement sur certains problèmes. Certaines classes de problème difficiles ont été identifiées. Par exemple, la *formule de trous de pigeons* de taille n encode en termes booléens le fait qu'il n'existe pas d'injection d'un ensemble à $n+1$ éléments dans un ensemble à n éléments ou, dit en termes imagés, qu'on ne peut pas "mettre $n+1$ pigeons dans n trous". Il a été démontré que toute réfutation de cette formule par résolution est de taille strictement exponentielle [137] (voir également la preuve simplifiée proposée par [16]). Or nous avons vu que les algorithmes présentés dans ce chapitre sont des variantes de résolution (*resolution-bounded*, au sens de [199]). Les formules telles celles des trous de pigeon permettent donc de construire des instances difficiles à résoudre pour tous ces algorithmes : on sait que le temps de résolution par DLL, MAC, ou d'autres variantes, sera strictement exponentiel[15].

Notons, enfin, que notre présentation, centrée autour des notions de résolution et d'itération chaotique, a seulement présenté les techniques que nous nous proposons par la suite d'utiliser ou d'adapter à d'autres types de contraintes. Nous n'avons ainsi pas pu mentionner les techniques de *consistance forte*. Issues des travaux de MONTANARI [202] et FREUDER [112], les consistances fortes sont des formes de propagation permettant d'atteindre des degrés de filtrage arbitrairement élevées — au prix d'une complexité croissante. Elles restent d'usage moins courant que l'arc-consistance, mais il est en général admis qu'elles présentent un intérêt pour la résolution de problèmes difficiles [85] ; il s'agit donc d'une perspective que nous gardons en vue pour la résolution de problèmes quantifiés.

[15] Ce constat montre une sérieuse limite commune à tout algorithme basé sur des techniques d'énumération et de propagation (par réduction de domaines ou de contraintes), incluant le *backtracking*, l'arc-consistance et toutes ses variantes (y compris les consistances fortes), et le *backtracking* intelligent et les autres méthodes d'apprentissage. D'autres techniques, différentes de celles utilisées classiquement en programmation par contraintes, permettent de résoudre ces instances difficiles, par exemple les diagrammes de décision binaires. Cependant, il existe des classes de problèmes pour lesquels des preuves concises par résolution existent alors que leur temps de résolution par BDD (pour tout ordre choisi) est exponentiel [136]. Il a également été suggéré (notamment dans [199]) de considérer d'autres systèmes de preuve, strictement plus concis que la résolution, comme alternatives et comme bases de nouveaux algorithmes de déduction. Citons par exemple les méthodes de génération de plans de coupe issues de la programmation en nombre entiers, ou la version booléenne des bases de GROEBNER [61].

PARTIE III

Résolution de contraintes quantifiées

Nous considérons dans ce chapitre un formalisme de définition de problèmes enrichissant le cadre CSP avec un nombre arbitraire d'alternances de quantificateurs. Un tel formalisme permet d'exprimer naturellement des problèmes complexes issus d'applications comme le model-checking *et le* planning*. Résoudre des contraintes quantifiées de ce type est un problème décidable. Il s'agit, en revanche, d'un problème* ***PSPACE****-complet.*

*Nous présentons successivement une description formelle du cadre, dûment motivé, puis une série de contributions à la résolution de ce type de contraintes. La principale, présentée dans le chapitre 6 et affinée dans le chapitre 7, est une généralisation de l'*arc-consistance*, technique essentielle dans la résolution des CSP non-quantifiés. Cette partie de la thèse se termine sur un aperçu des problèmes modélisés, qui nous permettent ainsi de valider notre approche sur le plan expérimental.*

CHAPITRE 5

Extension du cadre CSP aux contraintes quantifiées

Nous présentons ici le cadre général des problèmes de résolution de contraintes quantifiées. *Du point de vue du chapitre 3, ceux-ci correspondent au problème de satisfaisabilité de formules relationnelles (prénexes) de premier ordre sur des domaines finis. Nous particularisons le formalisme général présenté précédemment pour ce problème précis.*
Nous présentons le problème QCSP et spécifions sa sémantique en termes d'algèbre relationnelle. Nous motivons l'intérêt pour ce problème en décrivant ses grandes classes d'application, puis nous survolons certains des algorithmes de la littérature pouvant être considérés pour résoudre ce problème.

5.1 Présentation du problème

Si les problèmes de satisfaction de contraintes (CSP[1]) permettent d'exprimer et de résoudre une grande variété de problèmes combinatoires, les langages de programmation par contraintes ne sont cependant pas nécessairement des langages de programmation *complets* au sens des langages de programmation classique, qui permettent d'exprimer n'importe-quel algorithme. Bien évidemment, de nombreux systèmes de programmation par contraintes utilisent pour *langage hôte* un langage complet comme Prolog ou C++, mais l'expressivité du *noyau de résolution de contraintes* est néanmoins limitée, or c'est bien sur ce dernier qu'on se concentre lorsqu'on étudie les CSP.

De fait, il n'est pas difficile d'exhiber des problèmes qui ne peuvent pas raisonnablement être formulés comme des CSP. La raison en est simple: de nombreux problèmes centraux en Intelligence artificielle comme le calcul de plans d'actions, l'étude des jeux ou certaines formes de *model-checking*, sont des problèmes **PSPACE**-complets. Les CSP sont quant à eux un outil d'expression des problèmes **NP**, qui représentent une petite partie de **PSPACE** :

Théorème 1. (Stephen COOK [68]) *Tout problème p de la classe **P** peut être formulé comme un problème de résolution de contraintes dans le sens suivant : il est possible d'exhiber une fonction d'encodage f calculable en temps polynomial*[2] *associant à toute instance I du problème un problème de satisfaction de contraintes tel que I est vrai ssi $f(I)$ est satisfaisable.*

[1] Fidèles à la tradition régnant dans le domaine, nous utiliserons des acronymes basés sur la terminologie anglophone (e.g., *Constraint Satisfaction Problems*).

[2] On pourrait être plus restrictif sur les ressources exigées par les fonctions d'encodage : pour **NP** comme pour **PSPACE**, il est en fait possible d'utiliser seulement des fonctions de réduction **LOGSPACE** et même, comme montré dans [153], des fonctions de la classe $\mathbf{AC}_0$ (qui représente une classe de problèmes massivement parallélisables particulièrement peu coûteux

Que manque-t'il donc aux cadre si fructueux des CSP pour pouvoir modéliser les problèmes évoqués? La réponse se trouve dans un autre théorème de complexité :

Théorème 2. (Larry J. STOCKMEYER & Albert R. MEYER [241, 240]) *Tout problème **PSPACE** p peut être formulé comme un problème de résolution de contraintes quantifiées dans le sens suivant : il est possible d'exhiber une fonction d'encodage f calculable en temps polynomial associant à toute instance I du problème un problème de satisfaction de contraintes quantifiées tel que I est vrai ssi $f(I)$ est satisfaisable.*

C'est donc l'ajout des quantificateurs qui confère aux CSP l'expressivité nécessaire pour **PSPACE**[3]. Ce chapitre introduit les *problèmes de résolution de contraintes quantifiées* (QCSP[4]) comme une généralisation des CSP adaptée à la classe de problèmes **PSPACE**. Dans la formulation de ce type de notions, il est important de proposer une définition formelle du problème suffisamment générale pour en exprimer toute la richesse, mais tout en épurant le formalisme du maximum de détails non significatifs afin de lui conserver la plus grande simplicité possible. Un tel travail de formulation est perceptible dans la définition moderne des CSP rappelée au chapitre 4, qui est acceptée de manière relativement universelle malgré la diversité des auteurs et des styles de présentation. La formulation des CSP masque en particulier la distinction entre syntaxe et sémantique : si celle-ci est indispensable en logique classique, il est plus adéquat de considérer un CSP comme un simple ensemble de relations (sur un domaine donné) entre les variables du problème. Nous adoptons une approche similaire et proposons une généralisation simple du cadre CSP. Les rapprochements et différences avec les notions de logique introduites dans le chapitre 3 sont également soulignés. Dans le cadre simplifié, la sémantique formelle d'un CSP s'exprime de manière particulièrement élégante grâce à des notions d'algèbre relationnelle. Nous détaillons cette approche et discutons les propriétés algébriques des QCSP qu'elle permet de faire émerger.

Notons qu'une version préliminaire de certaines des contributions développées dans ce chapitre et le reste de cette partie a été publiée dans [33].

5.1.1 Problèmes de résolution de contraintes quantifiées

Nous avons vu au chapitre 3 que les problèmes quantifiés peuvent facilement être exprimés sous forme *prénexe*, c.-à-d. avec tous les quantificateurs en tête. Nous supposerons de plus que chaque variable du problème est quantifiée exactement une fois. Dans l'esprit de la formulation des CSP, on peut alors définir un QCSP comme un triplet $\langle \mathcal{X}, D, C \rangle$ où:

- $\mathcal{X}$ est une *séquence ordonnée* de noms de *variables* ; à chaque variable est associé un quantificateur, noté $quant(x) \in \{\exists, \forall\}$;
- D associe à chaque variable $x \in X$ un *domaine* D_x, représentant l'ensemble fini de valeurs que x est autorisé à prendre.

Pour préciser la définition de l'ensemble $\mathcal{C}$ dit *ensemble des contraintes*, il nous faut introduire les notions ensemblistes suivantes. Soit $\chi \subseteq \mathcal{X}$ un sous-ensemble des variables. Un *χ-tuple* t est une application de signature $\chi \rightarrow \mathbb{D}$, associant à chaque variable x une valeur (notée t_x) prise dans le domaine

en ressources calculatoires). On a donc véritablement affaire à des algorithmes d'encodage de coût calculatoire négligeable en comparaison des problèmes considérés.

[3] D'autres variantes de logique propositionelle possèdent une expressivité comparable, notamment la logique intuitioniste ou le fragment additif multiplicatif de logique linéaire [181].

[4] Pour *Quantified Constraint Solving Problems*. Le terme de *satisfaction* parait peu naturel pour les contraintes quantifiées, je préfère parler de *résolution* de contraintes quantifiées.

de x. Étant donné un sous-ensemble de variables $\chi' \subseteq \chi$, on définit la *restriction* de t à χ' (notée $t|_{\chi'}$) comme le χ'-tuple associant les mêmes valeurs que t aux variables de χ' et qui n'est pas défini sur les autres valeurs. Une χ-*contrainte* c est définie comme un ensemble de χ-tuples. Deux χ-contraintes remarquables sont 0^χ (relation vide, ne contenant aucun tuple) et 1^χ (relation "pleine", contenant l'ensemble des χ-tuples). Un $\mathcal{X}$-tuple t *satisfait* une χ-contrainte c si $t|_\chi \in c$, et une *solution* est un $\mathcal{X}$-tuple satisfaisant un ensemble de contraintes. On peut maintenant poser:

- $C = \{C_1, \ldots, C_m\}$ est un ensemble de contraintes, chacune étant une χ-relation sur un certain $\chi \subseteq \mathcal{X}$.

La notation $\sigma_{x=v}(\Phi)$, où Φ est un QCSP, désignera le QCSP obtenu en fixant le domaine de x à une valeur $v \in D_x$. Il est souvent utile d'analyser la séquence des variables quantifiées. On aura alors recours à la notation classique $Q_1 x_1, \ldots, Q_n x_n(C)$ (la lettre Q sera utilisée pour désigner un quantificateur choisi parmi $\{\exists, \forall\}$). Classiquement, le motif de quantificateurs est appelé *préfixe* et la partie non quantifiée du problème (C) est appelée *matrice*. L'abréviation $Q^\star$ désignera une séquence de variables quantifiées de longueur arbitraire. Par exemple, la notation $\exists x\, Q^\star(C)$ désigne un QCSP dont la première variable est x, qui est quantifiée existentiellement. Enfin, on aura parfois recours aux *quantificateurs bornés* ($Qx \in D_x$) pour représenter de manière concise à la fois la variable et son domaine.

Une manière simple de définir la notion de vérité et la sémantique d'un QCSP est par la biais de l'algèbre relationnelle (la conjonction correspondant à la jointure, l'existentielle à la projection et la négation au passage au complémentaire). L'approche relationnelle sera développée en Section 5.2.

5.1.2 Un exemple de QCSP

Un exemple de problème modélisé de manière naturelle par un QCSP est donné par le jeu suivant.

Exemple 6. *On considère un jeu à deux joueurs x et y, jouant alternativement. En début de partie, on dispose d'un tas de 100 jetons. Chaque tour de jeu se déroule comme suit : le joueur dont c'est le tour choisit un nombre i entre 1 et 10 et retire i jetons du tas. Le jeu s'arrête lorsque le tas est vide : le joueur qui a retiré le dernier jeton a gagné. Le fait que x possède une stratégie gagnante peut être modélisé par la formule quantifiée suivante :*

$$\exists x_0\ \forall y_1\, \exists x_1\ \ldots\ \forall y_9\, \exists x_9\ \left(\ x_0 + \textstyle\sum_{i\in\{1,\ldots,9\}}(y_i + x_i)\ \right) = 100$$

(la somme Σ est une abréviation pour une série de 100 additions répétées). En utilisant la technique de décomposition de formules par introduction de variables existentielles présentée dans le chapitre 3, on peut récrire la formule en une conjonction de contraintes. Il est en effet possible d'introduire une variable s_1 représentant la somme $s_0 + y_1$, puis une variable s_2 pour $s_1 + x_1$, etc.:

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{x_0}_{s_0} + y_1}_{s_1 = s_0 + y_1} + x_1}_{s_2 = s_1 + x_1} + \ldots}_{\ldots} + y_9}_{s_{17} = s_{16} + y_9} + x_9}_{s_{18} = s_{17} + x_9} = 100$$

Les variables introduites sont existentielles et il est correct de les introduire à la fin du préfixe de la formule. On obtient le QCSP suivant, qui utilise des contraintes d'addition et d'égalité :

$$\begin{array}{l} \exists x_0\ \forall y_1\ \exists x_1\ \ldots\ \forall y_9\ \exists x_9 \\ \quad \underline{\exists s_0\ \exists s_2\ \ldots\ \exists s_{18}} \quad s_0 = x_0\ \wedge\ s_{18} = 100\ \wedge \\ \qquad \bigwedge_{i\in 1..9} \left(\ s_{2i-1} = s_{2i-2} + y_i\ \wedge\ s_{2i} = s_{2i-1} + x_i\ \right) \end{array}$$

Ce QCSP est vrai, car le joueur x possède bel et bien une stratégie gagnante. En effet, x peut systématiquement choisir de se ramener à la situation dans laquelle le tas laissé à y est un multiple de 11. Au dernier tour, y devra donc retirer entre 1 et 10 éléments sur un tas de 11, et x pourra forcément vider le tas.

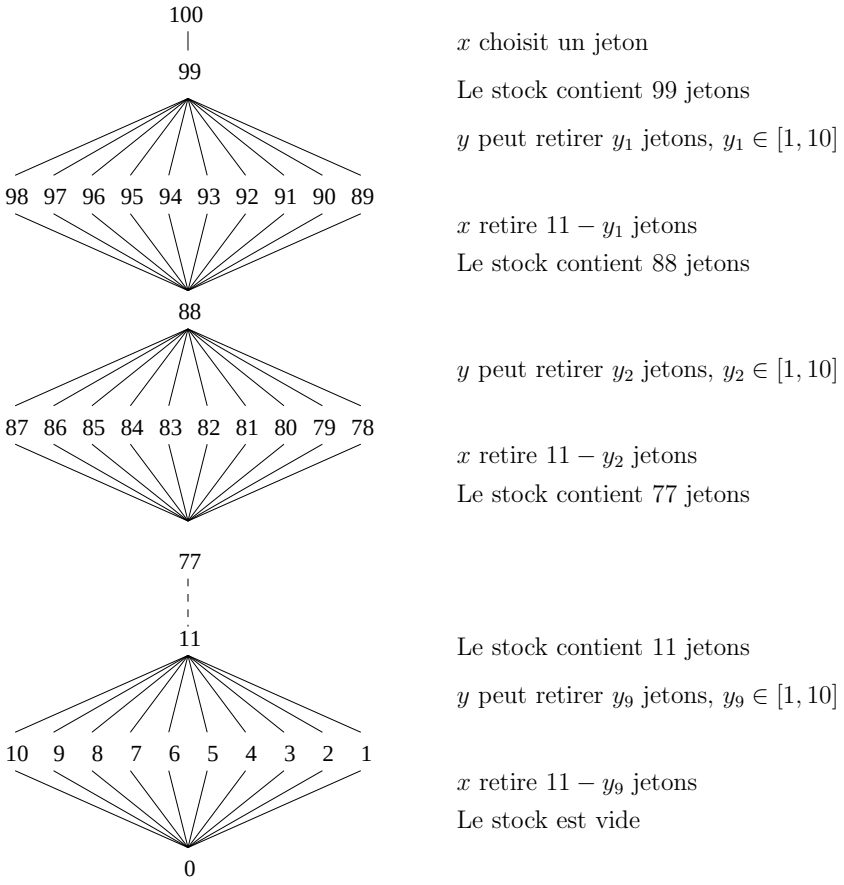


Figure 5.1 – La stratégie du joueur x gagnante pour le jeu à 100 jetons.

Cet exemple illustre le besoin, pour certains problèmes, de contraintes quantifiées sur les domaines *numériques*, et en particulier de contraintes linéaires. De tels problèmes, qui représentent une généralisation de la programmation linéaire en nombres entiers, seront appelés *problèmes de programmation linéaire quantifiée en nombre entiers*.

5.1.3 Notion de *solution* d'un QCSP

Dans notre étude des contraintes quantifiées nous nous contentons de chercher à déterminer si une formule bien formée (toutes les variables sont supposées quantifiées exactement une fois) est vraie ou non ; le *résultat* de nos algorithmes de résolution sera donc un simple booléen. Le problème exprimé sous cette forme est appelé *problème de décision*. Il est d'usage d'étudier les problèmes **NP** dans leur forme décisionnelle car celle-ci capture l'essentiel de ces problèmes : disposant d'un résolveur pour le problème de décision, il est facile d'utiliser celui-ci pour exhiber une *solution* d'un CSP car on peut instancier chaque variable à chacune de ses valeurs et conserver à chaque fois la première valeur vraie.

Il convient de justifier également la pertinence de l'étude du problème décisionnel pour QCSP. Une formule avec quantificateurs alternés exprime l'existence d'une certaine valeur pour tout choix dans une certaine autre valeur (et ainsi de suite). Idéalement, la notion de *solution* peut donc être identifiée à celle de *stratégie gagnante* : on attend de la machine qu'elle prévoie une combinaison de coups en réponse à l'ensemble des coups possibles pour l'adversaire. On obtient donc un *arbre de coups* prévoyant l'ensemble des parties possibles. Notez que cette notion peut également être vue sous l'angle de la *skolémisation* : une preuve de formule consiste à expliciter les fonctions associant les valeurs attribuées aux variables existentielles aux valeurs des variables universelles qui les précèdent. Cette notion de stratégie correspond également, d'une certaine manière, à celle de preuve.

La notion idéale de solution ne peut être retenue pour des questions de *taille* de la solution[5]. Une alternative est encore possible : celle consistant à déterminer une valeur pour les variables existentielles de tête de la formule. Par exemple, sur le problème :

$$\exists x_0 \ \forall y_1 \ \exists x_1 \ \ldots \ \forall y_9 \ \exists x_9 \ \left(\ x_0 + \textstyle\sum_{i \in \{1,\ldots,9\}} (y_i + x_i) \ \right) = 100$$

Il est intéressant de savoir que le nombre de jetons que x doit choisir dans l'immédiat est 1 (pour se ramener à la valeur 99). Notez qu'une variante consiste à rechercher une valeur pour les variables *libres* de la formule (le problème précédent pourrait être formulé avec x_0 non quantifié).

Il n'est pas difficile d'adapter un algorithme résolvant le problème décisionnel pour obtenir ce type de solution : il suffit d'instancier le problème (tour à tour) à toutes les valeurs possibles de la première variable existentielle, et de retenir la première valeur pour laquelle le sous-problème obtenu est vrai. On se satisfera donc dans cette thèse de l'étude du problème décisionnel.

5.2 Sémantique des QCSP

La sémantique des problèmes de résolution de contraintes quantifiées s'exprime de manière particulièrement simple et élégante en termes d'*algèbre relationnelle*. L'emploi direct de techniques d'évaluation d'opérateurs algébriques parait irréaliste car la taille des relations générées peut être exponentielle. Cependant, deux perspectives intéressantes se dégagent de l'approche relationnelle.

La première est d'ordre théorique : la vision relationnelle permet de mieux comprendre la difficulté réelle d'évaluation d'un QCSP particulier. Nous exhibons en effet un paramètre, nommé *largeur* d'un QCSP, qui donne une estimation de la performance des algorithmes d'évaluation. Ce paramètre est aisément calculable et il permet également de déterminer des transformations syntaxiques de QCSP permettant d'optimiser les performances de l'évaluation.

Deuxièmement, il est naturel de faire le rapprochement entre l'algèbre relationnelle et une catégorie de techniques utilisables pour représenter les relations et mettre en œuvre les opérations algébriques

[5] L'article [73] propose certaines techniques de *compression* de solution visant à répondre partiellement à ce problème.

les concernant : les *diagrammes de décision binaire ordonnées* (BDD). Nous donnons donc une brève présentation de ce type de structures de données et de leur application aux problèmes quantifiés.

5.2.1 Algèbre relationnelle

La sémantique des QCSP peut s'exprimer de manière formelle grâce à des notions d'*algèbre relationnelle* [1]. Cette sémantique est à la fois *dénotationnelle* dans le sens où elle donne une caractérisation mathématique des ensembles définis par un QCSP, et *opérationnelle* dans le sens où elle décrit un algorithme (naïf) d'évaluation. Notons que notre approche rejoint certains travaux de la communauté contraintes, notamment ceux de DECHTER et VAN BEEK [87]. Le rapprochement entre contraintes quantifiées et bases de données relationnelles est également connu de la communauté *bases de données*, voir par exemple [252, 253].

L'algèbre relationnelle est un formalisme utilisé essentiellement dans le domaine des bases de données (voir les nombreux ouvrages de référence sur le sujet, par exemple [1]). Nous en rappelons brièvement les principes et fixons les notations. L'entité de base considérée dans ce formalisme est la *relation* (et les contraintes quantifiées seront donc interprétées comme des relations entre certaines variables). Des *opérateurs relationnels* peuvent être définis dans le but de créer de nouvelles relations à partir de relations existantes. Parmi les exemples basiques d'opérateurs, on trouve notamment ceux d'*union* et d'*intersection* ($\cap$ et $\cup$, respectivement) ; appliqués à deux χ-relations, ces opérateurs définissent une nouvelle χ-relation. Pour rendre compte du sens des contraintes quantifiées, nous avons également besoin des opérateurs de *jointure*, *élimination*, *différence* et *sélection*. En fait, il est utile de distinguer entre les *expressions* qu'il est possible de construire à partir de ces opérateurs (e.g., $A \bowtie B$) et le résultat de l'*évaluation* de ces expressions (e.g., une relation portant sur les variables de A et de B). On présente donc successivement les notions d'*expression algébrique* et d'*évaluation* d'une telle expression.

5.2.1.1 Expressions algébriques

On dispose d'un ensemble $X = \{x_1, \ldots, x_n\}$ de variables, chacune possédant un domaine D_x. Intuitivement, une *expression algébrique* est une expression de forme :

$$E \quad ::= \quad Rel \mid E \bowtie E \mid \Pi_x(E) \mid E - E \mid \sigma_{x=v}(E)$$

Où Rel désigne une χ-relation sur un certain ensemble de variables $\chi \subseteq X$, x désigne une variable, et v désigne une valeur du domaine de calcul. Cependant, certaines constructions autorisées par cette grammaire doivent être interdites car n'ayant pas de sens : par exemple, les construction de type $E_1 - E_2$ ne sont autorisées que si les deux relations portent sur les mêmes variables ; de même on ne peut projeter sur une variable sur laquelle l'expression ne porte pas. Une définition formelle nécessite donc de préciser l'ensemble de variables sur lesquelles une expression porte :

Définition 17. (χ-expression algébrique)

- *Toute χ-relation est également une χ-expression. En particulier, 0^χ (χ-relation vide) et 1^χ (χ-relation "pleine") sont des χ-expressions ;*
- *Étant données une χ_1-expression E_1 et une χ_2-expression E_2, leur* jointure *$E_1 \bowtie E_2$ est une $(\chi_1 \cup \chi_2)$-expression ;*
- *Étant donnée une χ-expression E et une variable $x \in \chi$. L'*élimination *de x, notée $\Pi_x(R)$ est une $(\chi - \{x\})$-expression ;*

- *La* différence *de deux χ-expressions R_1 et R_2 est une χ-expression, notée $R_1 - R_2$. On abrégera l'expression $1^\chi - E$ en $-E$;*
- *Soit une χ-relation R, une variable $x \in \chi$ et une valeur $v \in D_x$, la* sélection $\sigma_{x=v}(R)$ *est également une χ-expression.*

Un exemple d'expression algébrique est $\sigma_{x=3}(\Pi_y(R))$ où R est, par exemple, la $\{x, y, z\}$-relation définie par $\langle x, y, z\rangle \in \{\langle 1, 3, 2\rangle, \langle 2, 2, 2\rangle\}$. On aura peu recours à la *sélection* dans ce chapitre, mais nous la présentons par soucis d'exhaustivité — elle sera de plus rencontrée dans la suite du manuscrit (remarquons de plus que la notation $\sigma_{x=v}(\Phi)$ est également utilisée pour représenter le fait d'instancier une variable d'un QCSP Φ). Notez enfin que, pour des raisons essentiellement techniques, nous avons remplacé la notion de *projection*, classique en algèbre relationnelle, par celle d'*élimination* de variables.

5.2.1.2 Évaluation des expressions algébriques

Une expression algébrique peut être *évaluée* en utilisant les définitions suivantes. Noter que l'évaluation d'une χ-expression donne une χ-relation.

Définition 18. (Évaluation de l'algèbre relationnelle)

*On notera $[\![E]\!]$ l'*évaluation *d'une expression algébrique E. Cette évaluation est définie comme suit :*

- *L'évaluation d'une χ-relation E est la relation E elle-même ;*
- *L'évaluation de la jointure $E_1 \bowtie E_2$ d'une χ_1-expression E_1 et une χ_2-expression E_2 est la $(\chi_1 \cup \chi_2)$-relation définie par :*

$$\{t \in (\chi_1 \cup \chi_2) \to \mathbb{D} \ \mid \ t|_{\chi_1} \in [\![E_1]\!] \ \ et \ \ t|_{\chi_2} \in [\![E_2]\!]\}$$

- *L'évaluation de l'élimination de x d'une χ-expression ($\Pi_x(E)$) est la $(\chi - \{x\})$-relation définie par :*

$$\{t|_{\chi-\{x\}} \ \mid \ t \in [\![E]\!]\}$$

  Lorsque $\chi = \{x\}$, l'unique variable de l'expression est éliminée et on obtient vrai *si $[\![E]\!] \neq \varnothing$,* faux *sinon ;*
- *La* différence $E_1 - E_2$ *de deux χ-expressions E_1 et E_2 s'évalue comme la χ-relation :*

$$\{t \in [\![E_1]\!] \ \mid \ t \notin [\![E_2]\!]\}$$

- *Soit une χ-expression E, une variable $x \in \chi$ et une valeur $v \in D_x$, l'évaluation de la* sélection $\sigma_{x=v}(E)$ *est l'ensemble des tuples de $[\![E]\!]$ ayant v pour valeur sur x :*

$$\{t \in [\![E]\!] \ \mid \ t_x = v\}$$

5.2.2 Traduction des QCSP dans l'algèbre relationnelle

L'interprétation relationnelle des QCSP est basée sur les équivalences logiques suivantes : la conjonction correspond à la jointure, la quantification existentielle à l'élimination de variable, etc. Considérons le jeu de connecteurs complet $\{\wedge, \neg, \exists\}$; intuitivement, la traduction d'une formule quantifiée en expression relationnelle s'effectue grâce aux réécritures suivantes :

$$\begin{array}{rcl} A \wedge B & \leadsto & A \bowtie B \\ \neg A & \leadsto & -A \\ \exists x \Phi & \leadsto & \Pi_x \Phi \end{array}$$

L'interprétation des autres connecteurs et quantificateurs s'exprime naturellement à partir du jeu $\{\wedge, \neg, \exists\}$. Par exemple une formule de type $\forall x\ A$ se traduit en une expression $-\Pi_x(-A)$. Nous décrivons de manière plus formelle l'expression algébrique associée à un QCSP :

Définition 19. (expression algébrique associée à un QCSP)

- *Un QCSP non quantifié de forme $C_1, \ldots, C_m$ se traduit en la jointure $C_1 \bowtie \cdots \bowtie C_m$;*
- *Un QCSP de forme $\exists x\ Q^\star(C_1, \ldots, C_m)$ se traduit en l'expression $\Pi_x(A)$, où A désigne l'expression algébrique associée à $Q^\star(C_1, \ldots, C_m)$;*
- *Un QCSP de forme $\forall x\ Q^\star(C_1, \ldots, C_m)$ se traduit en l'expression $-\Pi_x(-A)$, où A désigne l'expression algébrique associée à $Q^\star(C_1, \ldots, C_m)$.*

Par exemple, le QCSP :

$$\forall x\ \exists y\ \exists z\ (x + y = z, x = y)$$

se traduit sous forme algébrique par une expression

$$-\Pi_x(-\Pi_y(\Pi_z(\text{plus}(x, y, z) \bowtie \text{egal}(x, y))))$$

On dira, enfin, que le QCSP est *vrai* si la formule s'évalue à *vrai*. Notons qu'il a été imposé dans le chapitre précédent que les QCSP soient exprimés sous forme prénexe. Une fois convertis en expressions algébriques, cette restriction sera levée. Nous verrons en effet qu'une forme non-prénexe permet de mieux mettre en valeur la structure du problème, ce qui peut rendre son évaluation plus simple.

Il est instructif d'illustrer le rapport entre point de vue relationnel et logique sur un exemple d'évaluation.

Exemple 5.3. *Considérons le problème défini par*

$$\{x \mid \forall y\ \exists z\ (x + y = z)\}$$

où x, y et z ont pour domaine de valeurs l'ensemble $\{0, 1, 2\}$. La relation d'addition (add) sur ce domaine contient les tuples suivants :

x	0	0	0	1	1	2
y	0	1	2	0	1	0
z	0	1	2	1	2	2

Éliminer la variable z permet de calculer la table de la relation

$$\Pi_z(add) \ = \ \{\ \langle x, y\rangle \ \mid \ x + y \in 0 \ldots 2\ \}$$

c.-à-d. :

x	0	0	0	1	1	2
y	0	1	2	0	1	0

Le complémentaire de cette relation représente :

$$-\Pi_z(add) \ = \ \{\ \langle x, y\rangle \ \mid \ x + y \notin 0 \ldots 2\ \}$$

c.-à-d. :

x	1	2	2
y	2	1	2

Éliminons y pour calculer

$$\Pi_y \left(\ -\Pi_z(add) \ = \ \{ \ \langle x \rangle \ \mid \ \exists y \ (x + y \notin 0 \ldots 2) \ \} \ \right)$$

x	1	2

Reste à prendre le complément de cette relation pour obtenir :

$$-\Pi_y \left(\ -\Pi_z(add) \ = \ \{ \ \langle x \rangle \ \mid \ \forall y \ (x + y \in 0 \ldots 2) \ \} \ \right)$$

x	0

0 *est bien la seule valeur de x telle que, pour toute valeur de y, la somme $x + y$ soit dans $0 \ldots 2$.*

5.2.3 Complexité de l'évaluation d'un QCSP

La formulation algébrique d'un QCSP permet de tirer des enseignements précieux sur la complexité de ces problèmes. Il est clair que tous les opérateurs présentés peuvent être évalués en temps polynomial par rapport à la taille des tables sur lesquelles ils s'appliquent. Un paramètre important est la *dimension* des relations utilisées, c.-à-d. le nombre de variables de ces relations (cardinal de l'ensemble χ d'une χ-expression ; on notera ce cardinal $|\chi|$). On peut effectuer les constatations suivantes : les opérations de différence et de sélection ne changent pas la dimension du problème. L'opération de projection fait baisser le nombre de variables mises en cause (et donc la taille maximale des tables mises en jeu), alors que la jointure fait augmenter ce nombre de variables.

Il est donc clair que, partant de tables de dimension faible, l'opération de jointure est seule responsable de l'explosion de la taille des tables à calculer lors de l'évaluation. Par conséquent :

> *Pour minimiser la dimension des relations générées par l'algorithme d'évaluation, il est souhaitable d'effectuer les projections (correspondant aux quantificateurs, aussi bien existentiels qu'universels)* le plus tôt possible.

En d'autres termes, il est préférable d'éviter la *forme prénexe*, alors même que cette forme est en général imposée par la plupart des techniques de résolution existantes (la technique que nous proposerons dans la suite, l'*arc-consistance* quantifiée, ne faisant pas exception à la règle). L'exemple suivant montre que la différence de difficulté d'évaluation entre la forme prénexe et des formes plus appropriées peut être importante.

Exemple 5.4. *Considérons le QCSP suivant :*

$$\exists x_0 \ \forall y_1 \exists x_1 \ \ldots \ \forall y_9 \exists x_9 \quad \exists s_0 \exists s_1 \ \ldots \ \exists s_{18} \quad s_0 = x_0, s_{18} = 100, \ \bigwedge_{i \in 1..9} \begin{pmatrix} s_{2i-1} & = & s_{2i-2} + y_i, \\ s_{2i} & = & s_{2i-1} + x_i \end{pmatrix}$$

On a vu qu'il correspondait à la formule suivante (la transformation entre la formulation "fonctionnelle" et le QCSP est présentée dans le chapitre 3) :

$$\exists x_0 \ \forall y_1 \exists x_1 \ \ldots \ \forall y_9 \exists x_9 \ \left(\ x_0 + \textstyle\sum_{i \in 1..9} (y_i + x_i) \ \right) = 100$$

Évalué de manière naïve, ce problème requiert tout d'abord de construire une table de dimension 38, puis d'éliminer une par une toutes ses variables. En fait, le problème peut être ré-exprimé sous la forme suivante :

$$\exists x_0\ \forall y_1\ \exists s_1 \left(\begin{array}{l} s_1 = x_0 + y_1 \\ \wedge \\ \exists x_1\ \exists s_2 \left(\begin{array}{l} s_2 = s_1 + x_1 \\ \wedge \\ \forall y_2\ \exists s_3 \left(\begin{array}{l} s_3 = s_2 + y_2 \\ \wedge \\ \exists x_2\ \exists s_4 \left(\begin{array}{l} s_4 = s_3 + x_2 \\ \wedge \\ \ldots \exists x_9\ \exists s_{18} \left(\begin{array}{l} s_{18} = s_{17} + x_9 \\ \wedge \\ s_{18} = 100 \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right)$$

La valeur de vérité de ce QCSP est calculée par l'évaluation de l'expression suivante :

$$\Pi_{x_0}\ \text{–}\Pi_{y_1}\text{–}\Pi_{s_1} \left(\begin{array}{l} s_1 = x_0 + y_1 \\ \bowtie \\ \Pi_{x_1}\Pi_{s_2} \left(\begin{array}{l} s_2 = s_1 + x_1 \\ \bowtie \\ -\Pi_{y_2}\text{–}\Pi_{s_3} \left(\begin{array}{l} s_3 = s_2 + y_2 \\ \bowtie \\ \Pi_{x_2}\Pi_{s_4} \left(\begin{array}{l} s_4 = s_3 + x_2 \\ \bowtie \\ \ldots \Pi_{x_9}\Pi_{s_{18}} \left(\begin{array}{c} s_{18} = s_{17} + x_9 \\ \bowtie \\ s_{18} = 100 \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right)$$

L'évaluation de cette expression ne demande jamais la création la création d'une table de dimension supérieure à 3 : Partant d'une relation portant sur les variables s_{18}, s_{17} et x_9, on commence par éliminer x_9 et s_{18} pour obtenir une table de dimension 1, portant uniquement sur la variable s_{17}. Cette table est jointe avec les variables y_9 et s_{16}, puis y_9 et s_{17} sont éliminées (reste s_{16}), etc. (voir la figure 5.2).

Estimer la dimension maximale des contraintes à évaluer pour une expression donnée n'est pas difficile. Nous introduisons à cet effet la notion de *largeur* d'un QCSP, définie comme suit :

Définition 20. (Largeur d'un QCSP)

La largeur *d'une expression algébrique est le maximum des dimensions des relations construites lors de son évaluation, c.-à-d. :*

- *La largeur d'une χ-relation E non quantifiée est le nombre de variables de cette relation, c.-à-d. :*

$$w(E) \ = \ |\chi|$$

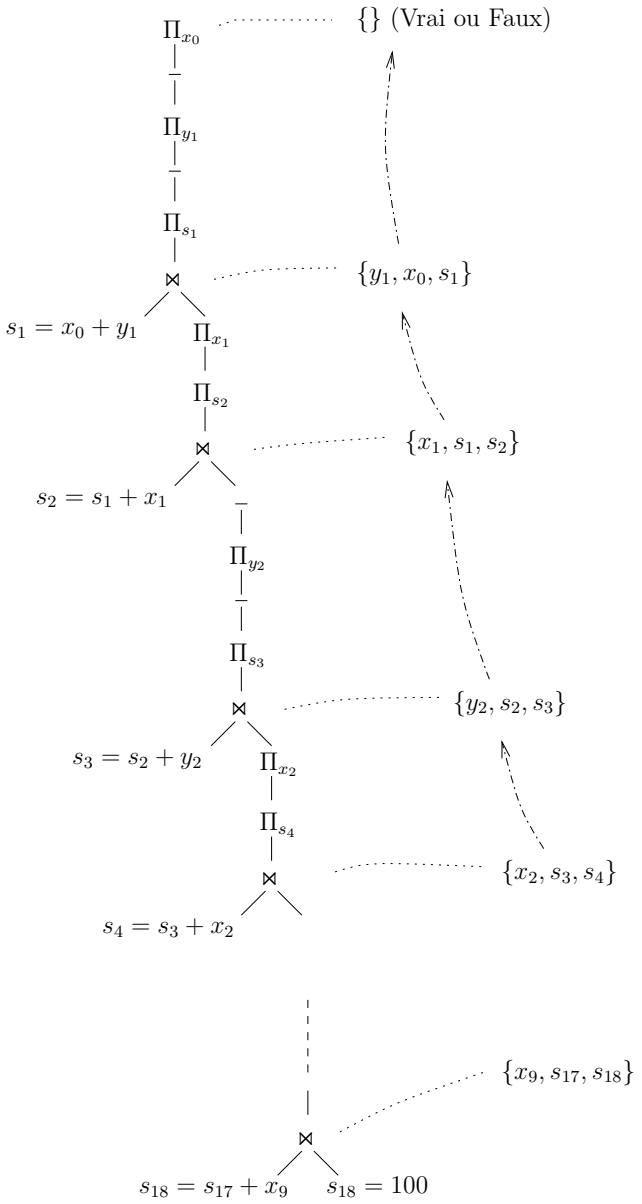


Figure 5.2 – Évaluation de la formule relationnelle correspondant au QCSP de l'exemple 5.4. Nous indiquons pour chaque jointure les variables sur lesquelles la relation correspondante porte.

- *La largeur de la jointure $R_1 \bowtie R_2$ entre une χ_1-expression R_1 et une χ_2-expression R_2 est le maximum entre la dimension de la relation produite et les largeurs des deux sous-expressions :*

$$w(E_1 \bowtie E_2) \;=\; \max\{|\chi_1 \cup \chi_2|, w(E_1), w(E_2)\}$$

- *La largeur d'une* élimination *est celle de l'expression dont on élimine une variable :*

$$w(\Pi_x(E)) \;=\; w(E)$$

- *La largeur de la* différence *est le maximum des largeurs des deux sous-expressions :*

$$w(E_1 - E_2) \;=\; \max\{w(E_1), w(E_2)\}$$

- *La largeur d'une* sélection *est la largeur de l'expression sur laquelle le sélection s'opère :*

$$w(\sigma_{x=v}(E)) \;=\; w(E)$$

Le paramètre w représente bien la dimension de la plus grande relation construite lors de l'évaluation d'une expression : pour évaluer une jointure par exemple, il est clair qu'on évalue ses deux sous-expressions (il faut donc prendre en compte le $\max$ des deux largeurs correspondantes), et que la relation construite peut être de dimension encore plus élevée que celles rencontrées lors du reste de l'évaluation. Notez de plus qu'un algorithme efficace de calcul de largeur (dirigé par la syntaxe) se déduit naturellement de la définition précédente. On peut démontrer que l'évaluation d'une formule de largeur w requiert un temps $\mathcal{O}(d^w)$, où d représente la taille du domaine. En particulier, si la largeur est faible, on a la garantie que le problème considéré sera résolu aisément par de simples techniques d'évaluation. Une perspective intéressante concerne donc l'utilisation de techniques symboliques permettant de réduire la largeur d'un QCSP. Des techniques similaires sont en fait classiques pour l'optimisation de requêtes sur les bases de données, et l'utilisation d'algorithmes issus de ce domaine de recherche est donc une piste intéressante. Je suggère, dans le schéma suivant, un exemple de réécriture possible permettant de réduire la largeur d'une formule relationnelle.

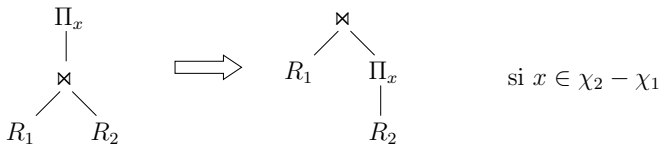


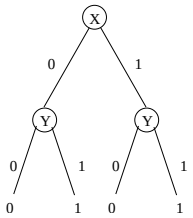
Figure 5.3 – Optimisation de requêtes par "projection d'abord".

Notez que le paramètre w joue un rôle assez similaire à certains paramètres introduits pour sur les réseaux de contraintes [86]. Enfin, puisque l'utilisation d'algèbre relationnelle pour résoudre des CSP a été étudiée par différents auteurs [135, 194, 229], il est probable que la plupart de ces travaux puissent être généralisés aux contraintes quantifiées par simple ajout de l'opérateur de négation.

5.2.4 Diagrammes de décision ordonnés

S'il parait peu réaliste de mettre en œuvre l'évaluation de formules d'algèbre relationnelles de manière naïve, certaines techniques peuvent être appliquées afin de représenter les relations construites de

manière concise. Un outil particulièrement éprouvé est la technique des *Diagrammes de Décision Ordonnés*[6] [42], qu'il me parait donc pertinent de mentionner brièvement dans cette section. Les BDD sont une optimisation des arbres de décision ordonnés, dont voici un représentant (relation $x = y$ sur le domaine $\{0, 1\}$) :



Un arbre de décision est un arbre dont chaque nœud est étiqueté par une variable et possède autant de fils que le domaine de la variable en question possède de valeurs. Les feuilles sont étiquetées par des valeurs de vérité. Un arbre de décision représente une relation : pour connaître la valeur de vérité d'un tuple donné, il suffit, partant de la racine de l'arbre, de choisir la branche correspondant à la valeur de la variable représentée par le nœud courant. On recommence l'opération jusqu'à arriver finalement sur une feuille dont la valeur détermine si le couple appartient ou non à la relation.

Les arbres binaires de décision possèdent souvent un grand nombre de sous-arbres isomorphes. L'idée de base des BDD est de fusionner ces nœuds ; on obtient ainsi un graphe acyclique direct dont la taille peut être réduite de manière significative. Pour pouvoir être réalisée efficacement, la fusion des nœuds isomorphes nécessite de fixer un ordre sur les variable qui (à quelques subtilités près) impose aux différents fils d'un nœud d'être étiquetés par la même variable — d'où le nom de diagramme *ordonné*[7]. La conversion entre arbre de décision et BDD peut alors être calculée en temps linéaire suivant un algorithme partant des feuilles de l'arbre (cf. Fig. 5.4).

L'intérêt principal des BDD est qu'il est possible d'appliquer de nombreuses opérations relationnelles, notamment la composition et les opérations algébriques standard, directement sur ces représentations [43, 21]. On obtient ainsi directement (en général en temps linéaire par rapport à la taille des formules générées) une représentation compressée de l'intersection ou de l'union de deux relations, ou de la relation obtenue en ajoutant ou supprimant (par projection) une variable d'une autre relation, etc. On remarquera en particulier que l'opération de *complémentation* ensembliste peut être implémentée de manière particulièrement efficace. Cette qualité est précieuse puisque la complémentation est essentiellement l'unique opération qui distingue le cadre CSP classique (jointures, projections) du cadre QCSP. Cette opération est d'ailleurs problématique pour de nombreux autres algorithmes. Il est clair que l'emploi des BDD constitue donc une approche raisonnable[8] à l'implémentation des méthodes de résolution de contraintes quantifiées issues de l'algèbre relationnelle. Notons par ailleurs que de nombreuses améliorations ont été proposées [198, 54]. Remarquons, enfin, que l'utilisation des 2^k-arbres expérimentée dans [229] présente des similarités flagrantes avec les BDD. En fait, la principale différence concerne

[6] Il est en général fait mention de Diagrammes de Décision Ordonnés *Binaires* ((O)BDD), qui sont appliqués aux domaines booléens, mais l'idée peut être adaptée de manière évidente à des domaines arbitraires (BDD *généralisés*).

[7] Le choix de l'ordre des variables a évidemment une incidence importante sur la taille des diagrammes générés ; des heuristiques d'ordre et de réordonnancement dynamique ont été proposées.

[8] Le fait que les BDD permettent de résoudre les formules booléennes quantifiées est connu (voir e.g., [256]) et a été expérimenté par différentes équipes [73] (les techniques de construction de BDD utilisables ne consistent d'ailleurs pas nécessairement à mettre en œuvre les opérateurs d'algèbre relationnelle, d'autres approches sont possibles). Il semble cependant que la construction de BDD pour ce type de formules soit souvent difficile, et la plupart des approches récentes de résolution de QBF sont basées sur d'autres techniques, notamment des variantes de DLL.

l'absence de compression par factorisation de sous-arbres isomorphes, et le découpage de domaines de valeurs réelles en un nombre fini d'intervalles.

Les BDD ont été proposés dans [42] ; de nombreux articles [43, 218, 238] peuvent servir d'introduction à ce type de techniques.

5.3 Domaines d'applications

Nous achevons notre présentation des QCSP en donnant un aperçu de leur champ d'applications. Puisque les QCSP sont une généralisation du problème des formules booléennes quantifiées, qui a bénéficié récemment d'un vif intérêt de la communauté SAT, les QCSP s'appliquent naturellement aux problèmes pour lesquels les QBF ont été introduites. Nous motivons donc l'intérêt de la résolution de QCSP par rapport au cadre booléen pour certaines de ces applications. Enfin, nous donnons un rapide aperçu des algorithmes récemment proposés pour la résolution de QBF.

5.3.1 Quelques exemples d'applications

La résolution des QCSP peut apporter des solutions nouvelles à la résolutions de problèmes issus des grandes classes suivantes :

Jeux – Il est bien connu que les formules logiques quantifiées ont une interprétation en termes de théorie des jeux (par exemple les jeux d'EHRENFEUCHT-FRAÏSSÉ [110], qui offrent notamment un outil d'analyse des formules logiques de premier ordre à modèles finis [98, 153]). De fait, les jeux à deux joueurs sont naturellement modélisés par des QCSP dans lesquels chaque joueur correspond à l'un des quantificateurs. Une question typique posée en théorie des jeux est par exemple :

Existe-t'il un coup tel que,
pour tout coup adverse,
il existe un coup tel que ... je gagne?

dont le rapport avec les logiques à quantificateurs est évident. Une première application concerne les algorithmes de résolution de jeux, mais l'intérêt est en fait plus vaste. En effet, de nombreux systèmes comportant un ensemble d'agents dont on veut étudier l'interaction peuvent être modélisés en termes de jeux. Les applications à la prise de décision économique sont en particulier notables ; le point suivant est d'une certaine manière un cas particulier de ce type de jeux.

Raisonnement en présence d'incertitude – De nombreux problèmes de décision nécessitent la prise en compte de données difficiles à estimer. Un cas typique vient des paramètres dont la valeur est prévue avec une certaine marge d'erreur, et sera réellement connue seulement *dans le futur*. Par exemple, pour organiser la production d'un article donné sur un nombre donné de mois, le mieux serait de connaître à l'avance les ventes réalisées pour chacun des mois, or on dispose en général seulement d'estimations de ces paramètres. Le nombre d'articles à produire en Février dépend à la fois de la production et des ventes de janvier ; celui de Mars de la production et des ventes de février, etc. La dépendance entre les valeurs de vente et de production de chacun des mois est analogue à une alternance de quantificateurs, ce qui a poussé certains auteurs à formuler le problème de *satisfaisabilité stochastique* [209, 183, 182], dont les instances possèdent la syntaxe suivante:

$$\exists x_1 \, \text{Я} x_2 \, \ldots \, \exists x_{n-1} \, \text{Я} x_n \, \left(\; valeur\_attendue(\Phi(x_1, \ldots x_n)) \geq \theta \; \right)$$

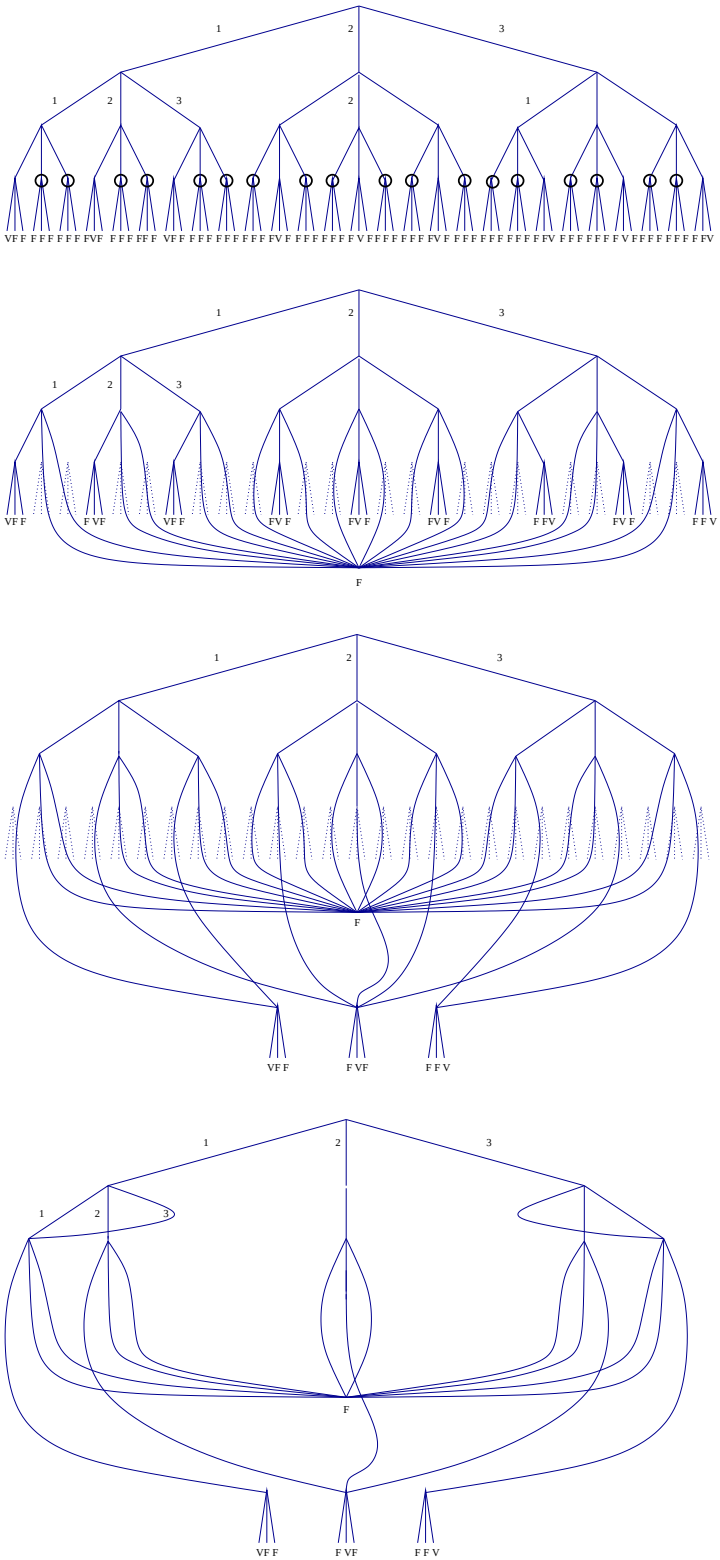


Figure 5.4 – Exemple de construction d'un diagramme de décision (non binaire en l'occurrence) par fusion des nœuds isomorphes. Un grand nombre de branches de l'arbre de décision (A) mènent à **F** dans tous les cas. On peut donc détruire les nœuds correspondants (et s'épargner ainsi le test de valeur sur la variable correspondant à ce nœud). On peut alors fusionner les branches isomorphes du niveau supérieur (C) et recommencer encore pour le niveau supérieur (D).

Une telle formule est lue "*existe-t'il x_1 telle que, pour des valeurs aléatoires*[9] *de y_1, il existe ... tel que la valeur attendue de Φ soir supérieure à θ?*" ; elle correspond presque directement à un QCSP.

Les problèmes de décision stochastique ont récemment attiré l'attention de la communauté contraintes, notamment avec l'introduction par T. WALSH de la *programmation par contraintes stochastique* [258, 188]), qui fait le lien entre la programmation par contraintes et la *programmation stochastique* (voir [29], ainsi que les nombreuses ressources accessibles depuis le site `http://www.stoprog.org/` pour une présentation de cette branche de la recherche opérationnelle).

Vérification de circuits et d'automates – Comme cette phrase de Lintao ZHANG[10] le résume, *pour la vérification de circuits, SAT est souvent insuffisant. De nombreux problèmes de vérification de circuit sont prouvés **PSPACE**-complets. Il s'agit d'une classe plus élevée que **NP** et, par conséquent, SAT n'est pas à même d'exprimer ce problème de manière efficace. L'évaluation de Formules Booléennes Quantifiées est mieux adaptée à ce type de problèmes.*

Cette opinion est partagée, notamment, par E. CLARKE qui, dans son exposé invité à CP 2002 [57], a explicitement appelé la communauté SAT et CP à proposer de nouvelles solutions au problème de satisfaisabilité quantifiée. De nombreux problèmes combinatoires **PSPACE**-complets sont en fait rencontrés dans différents aspects de la vérification de programmes. A propos de la preuve de **PSPACE**-complétude de la logique de HOARE propositionnelle [5], KOZEN écrit par exemple que "*cette construction est intéressante car elle permet de transformer un* model-checker *comme SMV en un vérifieur efficace pour l'équivalence d'expressions régulières"* [63]. Il est clair qu'un résolveur de contraintes quantifiées peut également fournir un outil pour ces divers problèmes d'équivalence et de preuve de programmes.

Calcul de plans d'actions – Domaine de recherche établi et important en Intelligence Artificielle, le calcul de plans d'actions (ou *planning*) consiste à déterminer une séquence d'actions permettant, depuis une situation initiale particulière, d'atteindre un but donné (voir [109] pour un article fondateur, [227] pour une introduction et [223] pour un état de l'art récent). Tous comme les problèmes de vérification de systèmes de transition, le calcul de plans d'actions est une forme de problème de recherche dans les graphes implicites, détaillé dans le chapitre 8. La plupart des problèmes de calcul de plans d'actions sont **PSPACE**-complets ; une approche à la modélisation de ces problèmes par des formules booléennes quantifiées est suggérée dans [222, 220].

Raisonnement dans les logiques non-standards – Certains problèmes calculatoires rencontrés dans ce type de logique sont de complexité supérieure à **NP** et s'expriment en tant que formules quantifiées avec un nombre faible d'alternances de quantificateurs. Il s'agit par exemple de certaines formes de logique non-monotone comme la *circumscription*, ou d'autres problèmes de classe Σ_i^p pour lesquels des approches basées sur les QBF ont été présentées [99].

Problèmes de décision en nombre réels – Un domaine d'application dont nous parlerons peu dans cette thèse consacrée aux problèmes *combinatoires* est celui de la résolution de problèmes quantifiés sur les nombres réels. De tels problèmes ont été étudiés dans la communauté programmation mathématique [157, 8, 148] et possèdent de nombreuses applications, dont une liste est dressée par S. RATSCHAN (voir `http://www.mpi-sb.mpg.de/~ratschan/appqcs.html` pour les références). Citons : l'ingénierie électrique et électronique, les problèmes de stabilité et de contrôle, la géométrie calculatoire et la preuve de théorèmes géométriques, la conception, le positionnement de caméras, la terminaison des systèmes de réécriture (!), etc.

[9] Le symbole Я vient du mot *random*.

[10] *Homepage*, `http://www.ee.princeton.edu/~lintaoz/`

Si ces problèmes sont de nature différente de ceux considérés dans cette thèse, dont les domaines sont finis, un point commun est cependant leur utilisation de contraintes *numériques* quantifiées. De même que les techniques de propagation d'intervalles ont permis d'apporter de nouvelles solutions aux problèmes d'optimisation non-linéaire, leur version quantifiée peut être appliquée à cette classe de problèmes difficiles [217, 17]. On remarquera à ce sujet que les techniques de propagation d'intervalles proposées dans le chapitre 7 sur les domaines finis peuvent se généraliser à certaines contraintes réelles — on obtient alors l'équivalent quantifié de la *Hull-consistance* [20].

Cette liste ne prétend nullement être exhaustive, et j'ai reporté seulement les problèmes pour lesquelles l'applicabilité des contraintes quantifiées est avérée. Pour citer un exemple de champ d'application plus prospectif, considérons la preuve de théorèmes de premier ordre. BAUMGARTNER [13] souligne le fait que la procédure de premier ordre "*DPLL réduit essentiellement la logique de premier ordre à de la logique propositionnelle. Le recherche de preuve est conduite en approximant une formule logique de premier ordre donnée par des ensembles, augmentés graduellement, de formules propositionnelles logiques*". DPLL a en général recours à la skolémisation pour éliminer les quantificateurs du théorème à démontrer ; un moyen possible d'éviter cette transformation est de prendre compte la quantification au niveau des formules propositionnelles construites. Enfin, le champ d'application de la logique propositionnelle quantifiée atteint des domaines plus variés encore puisque les protocoles interactifs introduits en cryptographie présentent de nombreuses analogies avec des jeux ou des problèmes quantifiés [234]. Bien entendu, il serait prématuré à l'heure actuelle de se hasarder sur un pronostic concernant l'intérêt des QCSP pour ce type d'applications ; celles-ci prouvent tout au moins que le problème présente de multiples facettes.

Les problèmes de raisonnement en présence d'incertitude nécessitent clairement l'utilisation de contraintes numériques et de domaines non booléens, qui justifie pleinement la définition d'un cadre de problèmes de contraintes quantifiées sur des domaines arbitraires. Les problèmes modélisant des systèmes de transition comme le calcul de plan d'action ou les problèmes de vérification peuvent en général être modélisés de manière booléenne. L'utilisation de domaines finis arbitraires peut cependant permettre d'améliorer leur formulation. Par exemple, si un problème tel que celui de la section 1.2.1 peut être modélisé de manière booléenne, il est plus naturel d'utiliser le domaine $\{\square, 0, X\}$, qui correspond directement aux valeurs possibles pour chaque case (occupée par l'un ou l'autre des pions ou vide). Ajoutons également que le cadre *programmation par contraintes* possède certains avantages par rapport aux techniques SAT : les contraintes globales, par exemple [14, 219], permettent la coopération d'algorithmes spécialisés avec les résolveurs d'usage général. De telles techniques donnent un avantage décisif au cadre CSP pour les applications dans lesquelles ce type de contraintes peut être identifié, et la perspective de fournir des contraintes globales utiles pour les domaines d'application des contraintes quantifiées est séduisante.

Remarquons enfin qu'une distinction importante existe entre les problèmes dont la modélisation requiert un certain motif de quantificateurs et ceux pour lequel le nombre d'alternances n'est pas borné. Le motif de quantificateurs $\exists^\star\forall^\star$ (formules dites de *Schönfinkel-Bernays*) permet par exemple de modéliser de nombreux problèmes, les plus connus étant par exemple des problèmes d'optimisation :

$$\exists \min\ \forall x \left(\ solution(\min) \ \wedge \ solution(x) \Rightarrow (\text{coût}(x) \geq \text{coût}(\min))\ \right)$$

Les problèmes possédant un motif de quantificateurs particulier sont ceux des classes intermédiaire de la Hiérarchie Polynomiale [212] (cf. chapitre 1). Une perspective intéressante consiste à prendre en compte les particularités de ces formules et de proposer des algorithmes adaptés à leur quantification particulière (c'est le cas, par exemple, de l'algorithme proposé dans [17] pour un certain type de formules de forme $\exists^\star\forall^\star$). A l'inverse, une caractéristique des problèmes **PSPACE**-complets est que le nombre d'alternances requises pour modéliser ces problèmes est, en général, impossible à borner.

5.3.2 Algorithmes de résolution existants

Nous avons déja mentionné l'intérêt récent de la communauté SAT pour le problème des contraintes booléennes quantifiées. De nombreux solveurs ont récemment été proposés pour ce problème.

Après des travaux consacrés à une forme particulière de QBF (Horn-SAT et 2-SAT quantifié) [4], le premier algorithme applicable aux QBF de manière générale a été proposé en 1995 [44]. Il s'agissait d'une méthode de *résolution* quantifiée, prouvée complète pour des clauses quantifiées arbitraires, et dont la version unitaire (analogue à la propagation unitaire rappelée dans le chapitre 4) est polynomiale et complète pour un certain type de formules Horn-SAT quantifiées. De même que la résolution classique (cf. chapitre 4), la résolution quantifiée est basée sur une représentation clausale de la matrice des problèmes considérés, et tire partie de certaines particularités de cette représentation (notamment : si une clause contient une variable universelle x telle qu'aucune variable existentielle de la clause n'apparaît avant x dans le préfixe, alors on peut simplifier la clause en supprimant x). Pour l'essentiel, l'algorithme est identique à la résolution classique, mais deux clauses peuvent générer un résolvant seulement si l'une d'elles contient un littéral x et l'autre un littéral $\neg x$, où on impose de plus que x *soit une variable existentielle*. La technique de résolution quantifiée a été reconsidérée récemment dans [55].

Quelques années après cette première contribution, plusieurs équipes ont entrepris de mettre en œuvre et d'expérimenter des techniques de résolution de QBF. La plupart des algorithmes proposés sont basés sur des techniques classiques pour SAT, et en particulier sur des variantes de l'algorithme DLL, dont les adaptations les plus remarquées sont probablement [220, 49, 50, 268]. L'algorithme le plus simple est probablement celui proposé dans [50] ; il s'agit d'un algorithme de *backtrack* dans lequel des tests de consistance et d'inconsistance sont utilisés pour éviter d'avoir recours à une exploration exhaustive (identification de clauses entièrement universelles non tautologiques, de clauses existentielles insatisfaisables, de littéraux monotones, etc.). D'autres références sont [9] ou [220, 221], qui utilise une technique de développement borné des quantificateurs. Dans le même esprit, d'autres équipes ont considéré l'emploi de techniques d'apprentissage et de *backtracking* intelligent [127, 126, 128, 129, 176, 266]. On trouvera également des contributions basées sur des méthodes de réécriture de formules quantifiées [215], l'utilisation d'outils de *model-checking* à base notamment de BDD [95], ou un algorithme parallèle [107]. Une contribution récente prometteuse est [120], qui marque l'apparition de techniques de recherche locale. Au rayon de l'étude théorique des algorithmes, on pourra consulter [261, 239, 55], ou encore l'étude des phénomènes de transition de phase des problèmes **PSPACE**-complets menée dans [122].

Tous ces algorithmes sont bien entendus restreints au domaine booléen (pour une présentation récente plus détaillée des algorithmes de résolution de formules booléennes quantifiées, consulter [175]). Bien que suggérés dans certains articles antérieurs de la littérature[11] [38, 39], les problèmes de contraintes discrètes quantifiées ont à notre connaissance été introduits par nos travaux personnels [33] ; le terme QCSP a depuis été utilisé par différents auteurs [55, 37]. Parmi les contributions proches du cadre QCSP, nous devons également mentionner les travaux sur la programmation stochastique par contraintes [258, 188], pour lesquels un algorithme de *forward-checking* est notamment proposé. Notons de plus que diverses contributions à des problèmes de contraintes quantifiées peuvent être identifiées dans la littérature pour des domaines divers : [67, 92] propose une méthode pour la théorie des arbres rationnels, [260] généralise la propagation de contraintes à des intervalles annotés qui correspondent à une forme exotique de quantification. [124] établit des liens entre les techniques de consistance et certaines formules quantifiées.

[11] Merci à E. FREUDER de m'avoir signalé ces références.

5.3.3 Illustration de la méthode proposée

Le reste de cette partie est consacré à la présentation d'une méthode de propagation de contraintes quantifiées. Il est clair que les QCSP peuvent être résolus par un algorithme d'élimination de quantificateurs remplaçant les universels et les existentiels par des conjonctions et disjonctions, respectivement. Cet algorithme peut alternativement être vu comme un algorithme de *backtrack* généralisé. Le but de la propagation de contraintes est de fournir un moyen d'élaguer l'arbre de recherche exponentiel généré par cet algorithme grâce à un raisonnement local, de coût polynomial, obtenant ainsi un algorithme de recherche-élagage (*branch & prune*) analogue aux algorithmes de type MAC ou DPLL. Nous nous contentons ici de décrire la partie non triviale de l'algorithme, c.-à-d. la partie *propagation*, dont nous donnons tout d'abord l'idée sur un exemple.

Considérons le problème suivant, qui modélise une version particulièrement simple (à deux tours!) du *jeu des allumettes* (chaque joueur prend tout à tour entre 1 et 10 allumettes, le tas en comporte 20, celui qui prend la dernière a gagné) :

$$\exists x_1 \in [1,10]\ \forall y_2 \in [1,10]\ \exists x_2 \in [1,10]\ \ (x_1 + y_2 + x_2 = 20)$$

Dans le cas de contraintes numériques arbitrairement complexes, il est en général nécessaire de *décomposer* le problème afin de pouvoir raisonner localement sur des sous-problèmes mettant en jeu un nombre limité de variables. Il est possible de remplacer chaque sous-terme complexe par introduction d'une variable existentielle, par exemple :

$$\exists x_1 \in [1,10]\ \forall y_2 \in [1,10]\ \exists x_2 \in [1,10]\ \underline{\exists s} \left(\begin{array}{rcl} x_1 + y_2 & = & s, \\ s + x_2 & = & 20 \end{array} \right)$$

D'autres alternatives sont parfois envisageables, mais il est toujours possible d'introduire ces quantificateurs en fin de préfixe. On raisonne donc sur deux sous-problèmes :

$$\exists x_1 \in [1,10]\ \forall y_2 \in [1,10]\ \exists s\ \ (x_1 + y_2 = s) \tag{5.1}$$

$$\exists x_2 \in [1,10]\ \exists s\ \ (s + x_2 = 20) \tag{5.2}$$

La propagation peut désormais intervenir. Considérons l'équation 5.2 : par propagation classique, il est facile de voir que s est borné par l'intervalle de valeurs $[10, 19]$. L'équation 5.1 devient :

$$\exists x_1 \in [1,10]\ \forall y_2 \in [1,10]\ \exists s \in [10,19]\ \ (x_1 + y_2 = s) \tag{5.3}$$

Déterminons maintenant les valeurs de $x_1 \in [1, 10]$ telles que, pour $y_2 \in [1, 10]$, la somme $x_1 + x_2$ appartienne à $[10, 19]$. Puisque y peut choisir de retirer une allumette, x doit choisir une valeur appartenant à l'intervalle $[10, 19] - 1 = [9, 18]$. y peut également choisir de retirer deux allumettes, et il faut donc également que le choix de x appartienne à $[10, 19] - 2 = [8, 17]$. En poursuivant le raisonnement, on déduit que x_1 doit appartenir à $[7, 16]$, $[6, 15]$, ..., et finalement à $[10, 19] - 10 = [0, 9]$. En fait, l'intersection ainsi définie se calcule très facilement : il suffit de raisonner uniquement sur les bornes de y et d'intersecter les intervalles $[9, 18]$ et $[0, 9]$. La seule bonne alternative pour x est donc de commencer par prendre 9 allumettes; il se trouve que cette stratégie est en effet gagnante. La règle de calcul utilisée se généralise à d'autres domaines que ceux considérés ici, et on peut donc définir un propagateur spécialisé pour les contraintes de forme $\exists x\ \forall y\ \exists z\ (x + y = z)$ (voir le chapitre 7).

CHAPITRE 6

Arc-consistance pour les contraintes quantifiées

*Nous présentons dans cette section une des principales contributions de cette thèse: la notion d'*arc-consistance quantifiée*, qui est une généralisation de l'arc-consistance permettant de gérer un nombre arbitraire d'alternations de quantificateurs.*
L'arc-consistance quantifiée a pour but de réduire les domaines des variables du problème. Elle permet ainsi d'éviter d'explorer certaines des branches rencontrées lors d'une recherche exhaustive, et de réduire l'explosion combinatoire des calculs.
Nous présentons tout d'abord la méthode dans un cadre restreint, visant uniquement à réduire les domaines de variables existentielles. Nous montrons comment intégrer l'arc-consistance à un algorithme de résolution complet et nous l'adaptons également à la réduction de domaines des variables universelles. Enfin, nous discutons les questions d'incrémentalité des calculs.

6.1 Notion de consistance

Les techniques d'élimination de quantificateurs suggérées dans le chapitre 4 permettent de déterminer la valeur de vérité d'un QCSP. L'algorithme peut être mis en œuvre en espace polynomial grâce à des techniques d'exploration/construction, qui consistent à explorer *en profondeur d'abord* les branches d'un arbre de recherche ET/OU. Cette technique correspond à une généralisation des algorithmes classiques de *backtrack*. Bien évidemment, le nombre de nœuds de l'arbre de recherche est cependant exponentiel, et il est donc irréaliste d'utiliser une telle technique d'énumération de manière naïve et exhaustive.

Pour éviter d'explorer certaines branches, il est nécessaire d'avoir recours à des moyens de déduction de coût limité (polynomial) permettant de détecter que certaines parties de l'arbre de recherche ne mènent pas à des solutions. Plusieurs approches ont été proposées pour atteindre ce but. L'approche utilisée en programmation par contraintes consiste à raisonner *localement* : on tentera de raisonner sur des sous-problèmes de taille bornée pour en tirer des conclusions sur le problème initial. Afin de proposer une telle approche, nous commençons par analyser le comportement d'algorithmes de recherche naïfs. L'exemple suivant nous accompagnera tout au long de cette section :

Exemple 6.5. *Considérons le QCSP suivant :*

$$\forall y_1 \, \exists x_1 \, \forall y_2 \, \exists x_2 \quad 8 - (y_1 + x_1 + y_2 + x_2) = 0$$

(toutes les variables ont pour domaine $\{1, 2, 3\}$). Cette formule peut être vue comme une version restreinte du jeu présenté dans le chapitre 5. Le jeu se joue en deux tours (y joue puis x puis y puis x). On dispose initialement de 8 jetons ; chaque joueur peut retirer à chaque tour entre 1 et 3 jetons et le but est d'être celui qui prendra le dernier jeton. On utilisera également la variante "plus simple" dans laquelle le but est de rendre la somme $8 - (y_1 + x_1 + y_2 + x_2)$ supérieure ou égale *à* 0.

6.1.1 Arbre de recherche et Arbre de preuve

Nous introduisons les notions d'*arbre de recherche* et d'*arbre de preuve*. Ces arbres peuvent être alternativement vus comme des *formules* logiques (nous utiliserons les deux points de vue).

Définition 21. (formule expansée)

On appelle formule expansée *d'un QCSP* Φ *la formule non quantifiée obtenue en remplaçant les quantificateurs universels et existentiels de* Φ *par (respectivement) des conjonctions et disjonctions :*

- *La formule expansée d'un QCSP non quantifié est le résultat du* test *de cette formule (remarquez qu'un QCSP non quantifié ne contient plus de variables et qu'il se réduit donc à un simple* test de contrainte*) ;*
- *La formule expansée d'un QCSP commençant par un quantificateur universel est la conjonction des QCSP obtenus en instanciant cette variable à toutes les valeurs possibles de son domaine.* [1]

$$expansion\ (\ \forall x \in \mathcal{D}\ Q^\star(C)\ )\ \stackrel{def}{=}\ \bigwedge_{v \in D} expansion\ (\ Q^\star(\sigma_{x=v}(C))\ )$$

- *La formule expansée d'un QCSP commençant par un quantificateur existentiel est la disjonction des QCSP obtenus en instanciant cette variable à toutes les valeurs possibles de son domaine.*

$$expansion\ (\ \exists x \in \mathcal{D}\ Q^\star(C)\ )\ \stackrel{def}{=}\ \bigvee_{v \in D} expansion\ (\ Q^\star(\sigma_{x=v}(C))\ )$$

Notez que les symboles $\bigvee$ et $\bigwedge$ désignent bien des *formules*, et non les valeurs logiques données par l'évaluation de ces dernières. De manière évidente, la formule expansée d'un QCSP peut être vue comme un arbre dont les nœuds sont étiquetés, selon le cas, par une conjonction ou une disjonction, et dont les feuilles sont des valeurs booléennes. On parlera d'*arbre de recherche* (aussi connu sous le nom d'*arbres ET/OU*) pour désigner l'arbre ainsi obtenu.

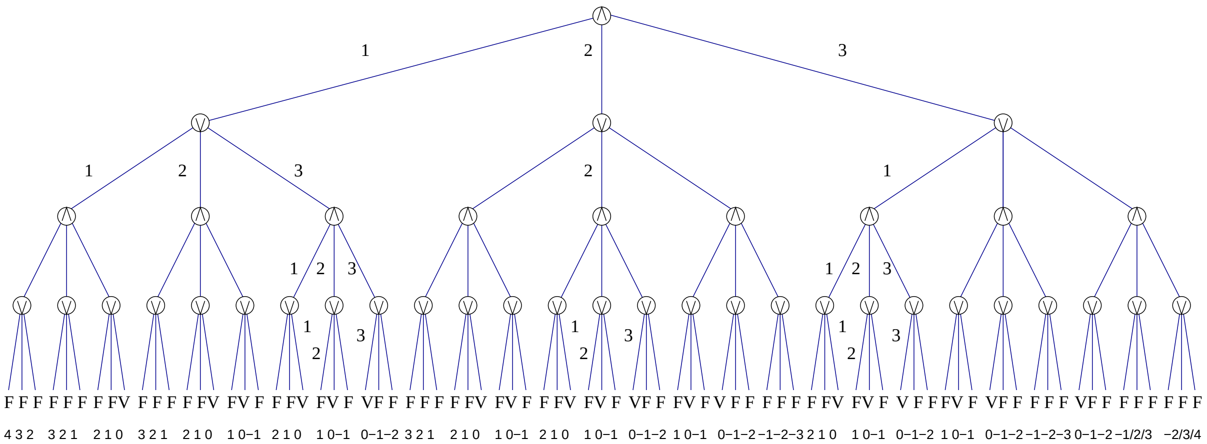


Figure 6.1 – Arbre de recherche pour le jeu présenté en exemple 6.5. Le rapport avec une formule booléenne (*formule expansée*) est immédiat. A chaque niveau, les branches correspondent, de gauche à droite, à la prise de 1, 2 ou 3 jetons. On a indiqué pour chaque fin de partie (feuilles de l'arbre) la valeur de vérité (**V**rai/**F**aux) de la formule $8-(y_1+x_1+y_2+x_2)=0$, ainsi que le nombre de jetons restants.

[1] Rappelons que $Q^\star(C)$ désigne un QCSP — la notation $Q^\star$ désigne une séquence arbitraire de quantificateurs.

De manière tout aussi évidente, l'*évaluation* de la formule expansée donne la *valeur de vérité* d'un QCSP, et cette valeur correspond au résultat de l'algorithme naïf d'*évaluation* suivant (celui-ci correspond directement à une exploration *en profondeur*, et donc en espace polynomial, de l'arbre de recherche) :

Algorithme *Evaluation_Naïve (qcsp) : booléen*

si $(qcsp = \exists x \in D\ Q^\star(C))$ **alors**
  % *évaluation de* $\bigvee_{v \in D}(Evaluation\_Naive(\sigma_{x=v}(C)))$ %
  pour tout $v \in D$ **faire**
    si $Evaluation\_Naive(\sigma_{x=v}(C))$ **then retourner** $vrai$
  fin pour
  retourner $faux$
fin si
si $(qcsp = \forall x \in D\ Q^\star(C))$ **alors**
  % *évaluation de* $\bigwedge_{v \in D}(Evaluation\_Naive(\sigma_{x=v}(C)))$ %
  pour tout $v \in D$ **faire**
    si non $Evaluation\_Naive(\sigma_{x=v}(C))$ **then retourner** $faux$
  fin pour
  retourner $vrai$
fin si
  % *qcp est non quantifié* %
retourner $test(qcsp)$

Fin Algorithme

Il est ainsi possible d'attribuer une valeur de vérité *à chaque nœud* de l'arbre de recherche. La valeur de vérité d'un nœud père se calcule en fonction de la valeur de ses fils par simple application de l'opération associée au nœud.

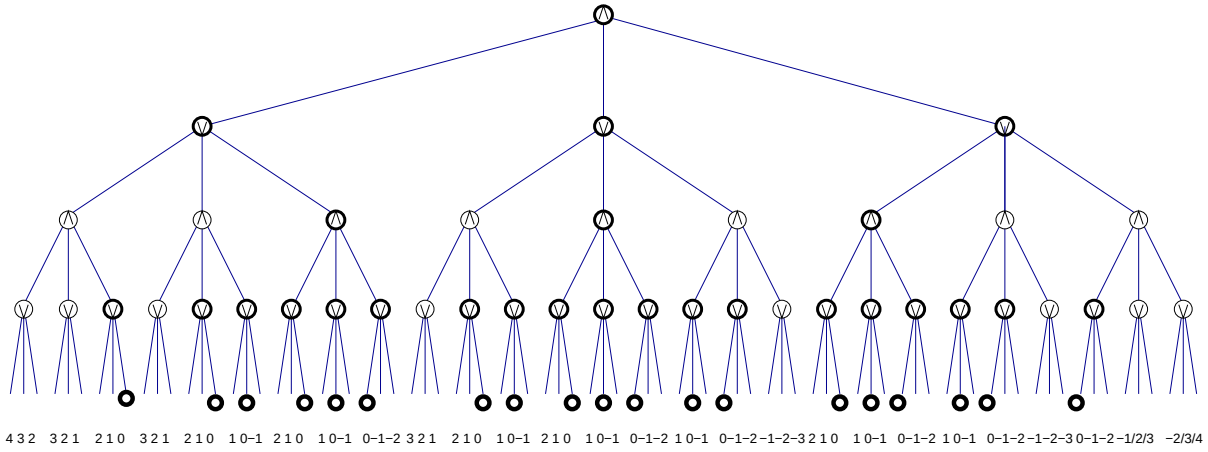


Figure 6.2 – Illustration de la notion de *valeur de vérité* : Les nœuds entourés par un cercle épais correspondent aux sous-formules *vraies* ; les autres sont *fausses*.

Il est clair qu'il est possible d'éviter le parcours de certaines branches de l'arbre de recherche : on peut en effet prouver qu'un QCSP est *vrai* simplement en exhibant l'*une* des branches satisfaisantes

pour chaque nœud *ou*. L'arbre simplifié ainsi obtenu est appelé *arbre de preuve*.

Définition 22. (arbre de preuve)
On appelle arbre conjonctif *(ou formule conjonctive) un sous-arbre de l'arbre de recherche dans lequel chaque nœud* ou *possède exactement un fils. Plus précisément :*

- *L'arbre conjonctif associé à un QCSP non quantifié est unique : il s'agit du booléen obtenu par simple test sur ce QCSP.*
- *Un arbre conjonctif associé à un QCSP $\forall x \in \mathcal{D}\ Q^\star(C)$ est obtenu par conjonction de $|D|$ arbres conjonctifs pour chacune des valeurs de x, c.-à-d.une conjonction :*

$$\bigwedge_{v \in D} T_v$$

où chaque T_v est un arbre conjonctif du QCSP $Q^\star\ \sigma_{x=v}(C)$.
- *Un arbre conjonctif associé à un QCSP $\exists x \in \mathcal{D}\ Q^\star(C)$ est donné par n'importe quel T_v où T_v est un arbre conjonctif du QCSP $Q^\star\ \sigma_{x=v}(C)$.*

(notez que, ainsi défini en termes de formules, un arbre conjonctif ne comporte effectivement aucune disjonction). Un arbre de preuve *est un arbre conjonctif s'évaluant à* vrai.

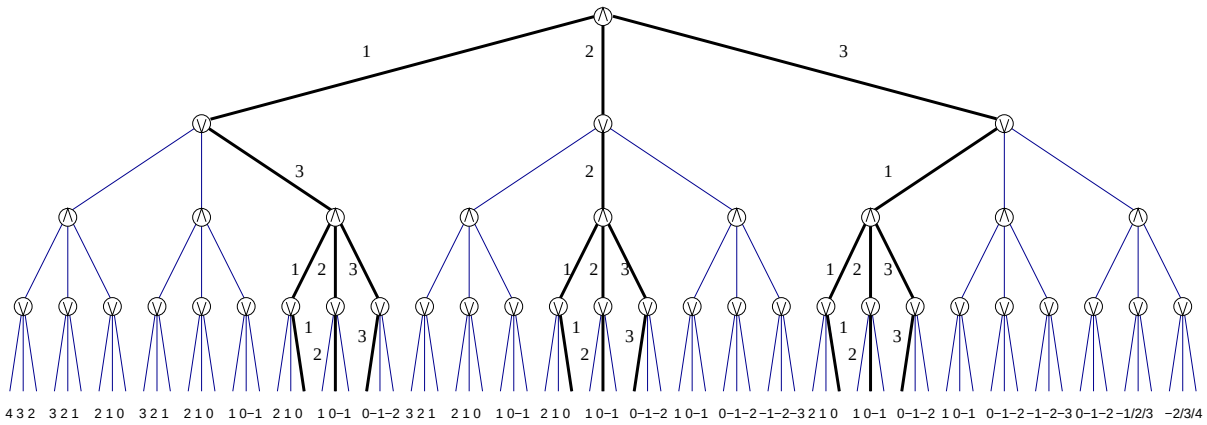


Figure 6.3 – Arbre de preuve pour le problème considéré en exemple 6.5. Cet arbre est conjonctif dans le sens où chaque nœud *ou* possède un seul fils.

La notion d'*arbre de preuve* correspond assez fidèlement à celles de *solution* ou de *stratégie gagnante*, discutées dans le chapitre 5. Il est par ailleurs évident qu'un QCSP est vrai si et seulement si on peut construire un arbre de preuve.

6.1.2 Valeurs consistantes & arbre des preuves possibles

Le but de l'exécution d'un algorithme de résolution de QCSP est de parcourir-construire une preuve s'il en existe une ou de s'assurer qu'il n'en existe pas dans le cas contraire. Pour chaque nœud *ou* de l'arbre, il s'agit donc de trouver une branche fille satisfaisante, et il n'est pas nécessaire de les parcourir toutes : dès qu'une branche satisfaisante est rencontrée, il est possible d'arrêter la recherche dans le sous-arbre considéré car on sait que celui-ci est *vrai*. Une première solution pour prendre en compte

cette remarque est d'utiliser une *évaluation paresseuse* de l'arbre de recherche, retournant *vrai* sur un nœud *ou* dès qu'une branche satisfaisante a été rencontrée ; cependant, comme le montre la figure 6.4, cet algorithme est loin d'éviter toute branche incorrecte.

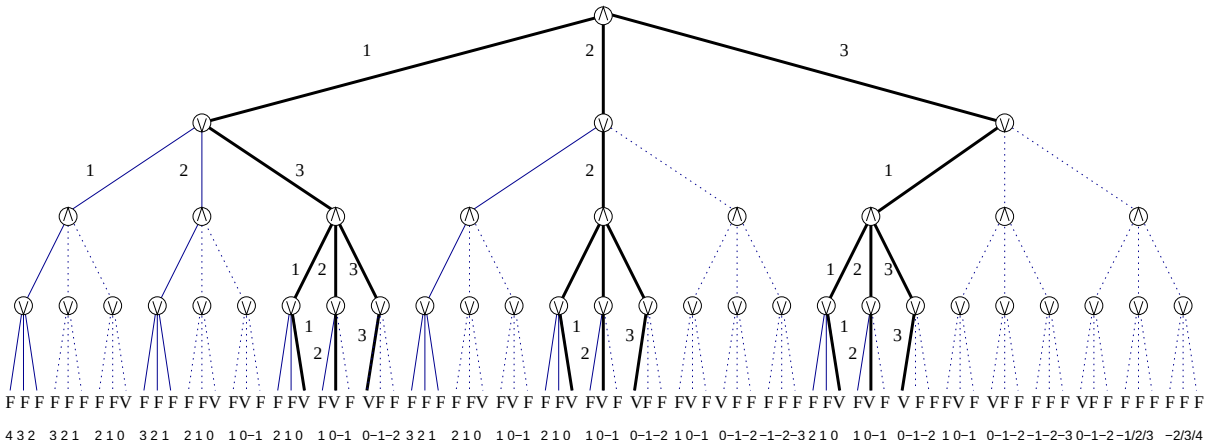


Figure 6.4 – Branches visitées (traits non pointillés) par l'algorithme d'évaluation paresseuse en profondeur basé sur un ordre de gauche à droite (ordre croissant du nombres de jetons considérés) ; on voit les bénéfices de l'évaluation paresseuse. Le nombre de nœuds parcourus est cependant plus important que dans l'unique arbre de preuve du QCSP (traits épais).

Le problème est que la recherche naïve parcourt parfois des branches pour lesquelles aucun arbre de preuve n'est possible, car elle considère systématiquement toutes les branches filles d'un nœud *ou* dans un certain ordre de parcours, jusqu'à trouver une éventuelle branche *vraie*. L'idéal serait bien entendu d'éviter le maximum de branches existentielles de valeur *faux*, et donc de prévoir à l'avance que certaines branches ne permettront jamais de construire une preuve. On désigne par *arbre des preuves possibles* la construction suivante, qui représente l'ensemble des branches réellement intéressantes de l'arbre de recherche.

Définition 23. (arbre des preuves possibles)
Une formule fausse *ne possède pas d'arbre de preuve possible. Dans le cas d'une formule* vraie *:*

- *L'arbre des preuves possibles associées à un QCSP non quantifié est le booléen obtenu par le test de ce QCSP.*
- *L'arbre des preuves possibles d'un QCSP $\forall x \in \mathcal{D}\ Q^\star(C)$ est la conjonction des arbres des preuves possibles de chacun des $Q^\star\ \sigma_{x=v}(C)$, pour $x \in D$;*
- *L'arbre des preuves possibles d'un QCSP $\exists x \in \mathcal{D}\ Q^\star(C)$ est la disjonction des arbres des preuves possibles de tous les $Q^\star\ \sigma_{x=v}(C)$ s'évaluant à* vrai*, pour $x \in D$.*

On cherchera donc à éviter d'effectuer une recherche dans les branches ne faisant pas partie de l'arbre des preuves possibles. Certains QCSP possèdent une preuve unique et, dans ce cas, la notion d'arbre des preuves possibles et celle d'arbre de preuve coïncident ; c'est le cas du problème considéré dans l'exemple 6.4, dont l'arbre des preuves possibles se réduit à l'arbre présenté en Figure 6.3. Dans d'autres cas, l'arbre de preuves possibles peut comporter un grand nombre de branches, reflétant l'existence de nombreux choix sur les nœuds existentiels (cf. Figure 6.5).

Se pose la question de la *représentation* des branches utiles ; clairement, le but étant d'augmenter l'efficacité, il n'est pas question de s'autoriser un stockage coûteux de structures de données exponentielles — pas question donc de stocker des parties entières de l'arbre de preuve. Nous opterons pour

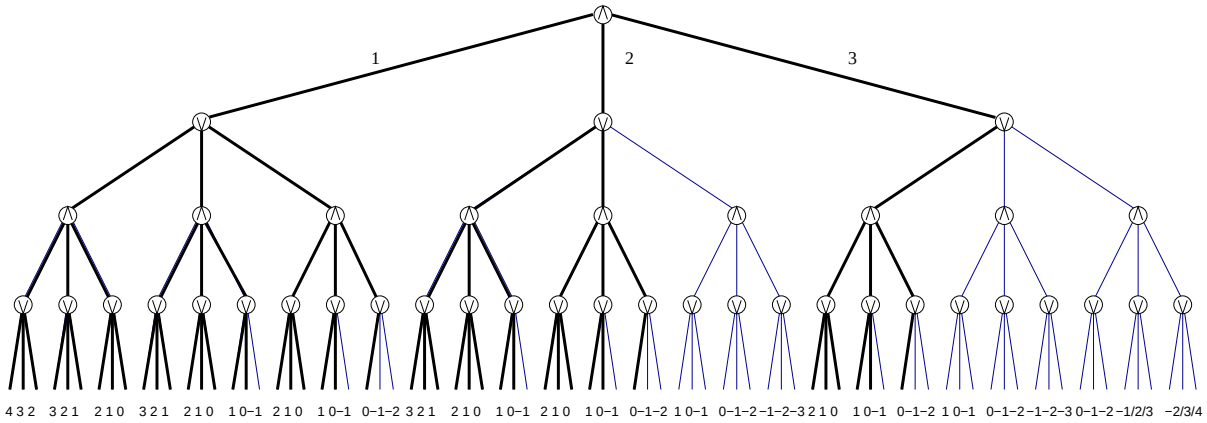


Figure 6.5 – Arbre des preuves possibles du QCSP $\forall y_1\ \exists x_1\ \forall y_2\ \exists x_2 \quad 8 - (y_1 + x_1 + y_2 + x_2) \geq 0$.

l'alternative éprouvée pour SAT et CSP : celle consistant à calculer, pour chaque variable, l'ensemble des valeurs que celle-ci sera amenée à prendre dans l'ensemble des branches de l'arbre. On a donc une représentation par *domaines* dans le sens des CSP, et on cherche à déterminer quelles valeurs de chaque domaine sont *consistantes*.

Définition 24. (valeurs consistantes)

On appelle affectation *un* couple variable/valeur, *noté* $(x : v)$. *L'ensemble des affectations* consistantes *d'un QCSP* faux *est vide, sinon il est défini comme suit :*

- *L'ensemble des affectations consistantes d'un QCSP non quantifié est vide ;*
- *L'ensemble des affectations consistantes d'un QCSP universel est définie par :*

$$consist(\forall x \in \mathcal{D}\ Q^\star(C)) \quad \stackrel{def}{=} \quad \Big(\ \{(x : v) \mid v \in D\} \ \cup \ \bigcup_{v \in \mathcal{D}} consist(Q^\star(\sigma_{x=v}(C))) \ \Big)$$

- *L'ensemble des affectations consistantes d'un QCSP existentiel est définie par :*

$$consist(\exists x \in \mathcal{D}\ Q^\star(C)) \quad \stackrel{def}{=} \quad \left(\begin{array}{r} \{(x : v) \ \mid \ Q^\star(\sigma_{x=v}(C)) \textit{ est vrai}\} \\ \cup \\ \bigcup_{v \in \mathcal{D}} \{consist(Q^\star(\sigma_{x=v}(C)) \ \mid \ Q^\star(\sigma_{x=v}(C)) \textit{ est vrai}\} \end{array} \right)$$

L'ensemble des valeurs consistantes *pour une variable x (noté $consist(x)$) est l'ensemble des valeurs des affectations consistantes portant sur x, c.-à-d.l'ensemble :*

$$val(x) \quad \stackrel{def}{=} \quad \{v \mid (x : v) \textit{ est une affectation consistante}\}$$

Enfin, on appelle inconsistantes *les affectations non consistantes d'un problème P, et on note l'ensemble de ces valeurs $inconsist(P)$.*

Une affectation consistante $(x : v)$ représente donc le fait que x peut prendre la valeur v dans une branche d'un arbre de preuve possible : la définition précédente signifie qu'on collecte les valeurs prises par toutes les branches partant vers des nœuds fils *vrais* (on collecte bien, ainsi, l'ensemble des preuves possibles). Pour un nœud universel toutes ces branches sont vraies (sinon la formule est fausse et on se trouve dans le cas où l'ensemble de valeurs consistantes est vide). Notez que conserver les valeurs

consistantes de variables universelles est sans intérêt : soit cet ensemble représente le domaine complet de la variable, soit la formule universelle est fausse. Un ensemble d'exemples illustrant la notion de consistance est donné en Figure 6.7. Un algorithme de construction des valeurs consistantes d'un QCSP s'obtient directement depuis la définition précédente (l'algorithme *Collect_Consist* collecte uniquement les valeurs consistantes pour les variables *existentielles*).

Algorithme *Collect_Consist (qcsp)*

% *Le QCSP est supposé* vrai — *Initialement on a* $Consist = \emptyset$ %
si $(qcsp = \forall x \in D\ Q^\star(C))$ **alors**
  For all $v \in D$ **do** $Collect\_Consist(\sigma_{x=v}(C))$
fin si

si $(qcsp = \exists x \in D\ Q^\star(C))$ **alors**
  pour tout $v \in D$ **faire**
    si $(\sigma_{x=v}(C)$ *est vrai*$)$ **alors**
      $Consist \leftarrow Consist \cup \{(x : v)\}$
      $Collect\_Consist(\sigma_{x=v}(C))$
    fin si
  fin pour
fin si
% *Si qcp est non quantifié, on ne fait rien* %

Fin Algorithme

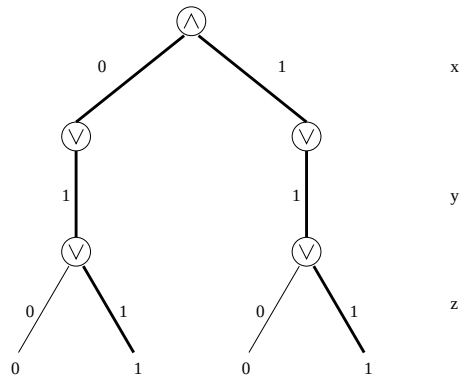
Complexité

Afin d'éviter les recalculs, on suppose que les valeurs de vérité de chaque nœud sont calculées par une phase préliminaire d'évaluation qui nécessite un temps $\mathcal{O}(md^n)$, où d est la taille du plus grand domaine de variables, n est le nombre de variables quantifiées, et en faisant la supposition que le test de chacune des m contraintes se fait en temps constant. La phase de collection des valeurs se fait alors en $\mathcal{O}(d^n)$ sous réserve d'une représentation de l'ensemble $Consist$ autorisant l'ajout d'éléments en temps constant. Le coût total est donc $\mathcal{O}(md^n)$.

6.2 Raisonnement local

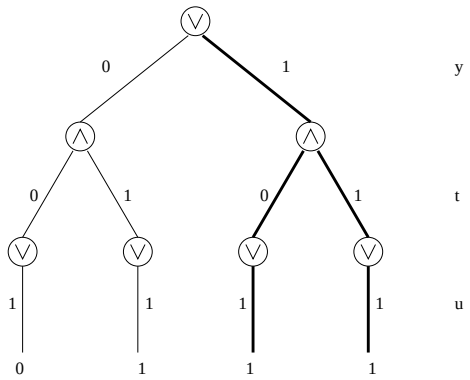
Déterminer si une valeur est consistante est un problème complexe : si nous raisonnons au niveau d'un QCSP entier, calculer les valeurs globalement consistantes revient simplement à résoudre ce QCSP. Pour limiter ce coup calculatoire, on cherchera à déterminer des conditions *suffisantes*, permettant de garantir l'inconsistance de certaines valeurs, mais on n'exigera pas que ces conditions soient *nécessaires*. L'utilisation d'un raisonnement *local*, restreint à des sous-parties du problème, donne de telles garanties. Le but de ce raisonnement est d'éliminer des valeurs des *domaines* des variables.

La correction du raisonnement local repose sur une propriété de *monotonie de la valeur de vérité d'une formule par ajout de contraintes*, qui stipule que (comme l'intuition le suggère dans le cas non quantifié) l'ajout d'une contrainte fait baisser la valeur de vérité d'un QCSP :



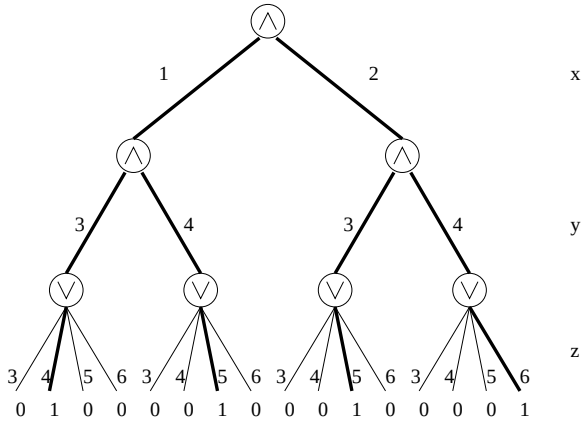
$\forall x \in \{0,1\}\ \exists y \in \{1\}\ \exists z \in \{0,1\}\ (x \vee y = z)$

$(z : 0)$ est une valeur inconsistante.



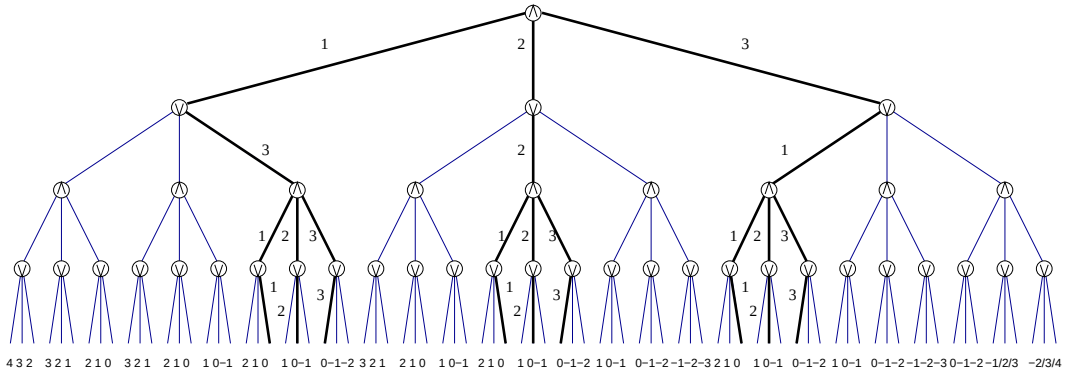
$\exists y \in \{0,1\}\ \forall t \in \{0,1\}\ \exists u \in \{1\}\ (y \vee t = u)$

$(y : 0)$ est une valeur inconsistante.



$\forall x \in \{1,2\}\ \forall y \in \{3,4\}\ \exists z \in \{3,4,5,6\}\ (x + y = z)$

$(z : 3)$ est une valeur inconsistante.



Arbres des preuves possibles du QCSP présenté en début de chapître.

Toutes les valeurs sont consistantes ! (x, y et z prennent chacune des valeurs 1, 3 dans une des branches)

Figure 6.6 – Ilustrations de la notion de *consistance*. L'ensemble des valeurs consistantes pour une variable est calculé en parcourant l'ensemble des branches d'arbres de preuves possibles et en collectant les différentes valeurs prises par cette variable le long de ces différentes branches.

Lemme 1. (Monotonie de la vérité d'une formule par ajout de contraintes)

Si un QCSP $A = Q^\star\ (C_1, \ldots C_m)$ est faux, alors pour toute contrainte supplémentaire C_{m+1}, le QCSP $B = Q^\star\ (C_1, \ldots, C_m, C_{m+1})$ est également faux.

> *Démonstration.* Chacune des feuilles de l'arbre *et*/*ou* de la formule B est fausse si la feuille correspondant dans l'arbre A est fausse. Par *monotonie des opérateurs* $\wedge$ *et* $\vee$, la valeur de vérité de l'arbre B est donc *faux* si A est *faux*. □

6.2.1 Élimination de valeurs

Le fait d'éliminer une valeur du domaine d'une variable permet d'une certaine manière de simplifier ce problème. On utilise la notation suivante pour exprimer le retrait d'une valeur du domaine d'une variable d'un QCSP :

Définition 25. (opérateur d'élimination de valeur)

Soit un QCSP $P = \langle \mathcal{X}, D, C \rangle$. Le QCSP obtenu par élimination *de la valeur $(x : v)$ est :*

$$P \backslash (x : v) \stackrel{def}{=} \langle \mathcal{X}, D', C \rangle$$

où $D'_x = D_x - \{v\}$ et les domaines des autres variables demeurent inchangés. En généralisant cette notation, on autorisera aussi l'écriture $P \backslash E$ où E est un ensemble de valeurs éliminées.

Par exemple, soit $P = \forall x \in \{1,2\}\ \exists y \in \{1,2\}\ C(x,y)$, on a $P \backslash (y : 2) = \forall x \in \{1,2\}\ \exists y \in \{1\}\ C(x,y)$. La suppression de valeurs inconsistantes permet de simplifier un QCSP car le problème ainsi obtenu est équivalent au problème initial :

Proposition 4. *Si une affectation $(x : v)$ est inconsistante pour un problème P, et si x est une variable* existentielle*, alors on a :*

$$P \text{ est vrai} \quad \text{si et seulement si} \quad P \backslash (x : v) \text{ est vrai}$$

> *Démonstration.* Les valeurs inconsistantes correspondent à des branches qui ne font pas partie de l'arbre des preuves possibles de P. Tous les arbres de preuve construits pour P seront donc également des arbres de preuve de $P \backslash (x : v)$. □

Cette proposition est bien évidemment fausse pour les variables *universelles*. A chaque fois qu'une valeur inconsistante est détectée, on peut donc remplacer le QCSP considéré par le QCSP simplifié.

6.2.2 Opérateur de réduction associé à une contrainte

La possibilité de raisonner *localement* vient du fait que les déductions obtenues depuis un sous-problème restent vraies dans le problème initial. On a donc besoin de résultats de conservation de propriétés : la proposition suivante garantit que les déductions opérées en raisonnant isolément sur chacune des contraintes du problème seront également valides pour le QCSP initial considéré dans son intégralité :

Proposition 5. (correction du raisonnement local)

Considérons un QCSP $P = Q^\star\ (C_1, \ldots, C_m)$. Si une valeur est inconsistante pour un certain QCSP $Q^\star(C_i)$, alors elle est également inconsistante pour P. En d'autres termes, on a :

$$consist(Q^\star\ (C_1, \ldots, C_m)) \quad \subseteq \quad consist(Q^\star(C_i))$$

pour tout $i \in \{1 \ldots m\}$.

Démonstration. On procède par induction sur le nombre de quantificateurs du QCSP. Si $P' = Q^\star(C_i)$ est faux, alors P est également faux (lemme 1). Dans ce cas comme dans le cas où les problèmes sont non quantifiés, on a $consist(P) = consist(P') = \varnothing$.
Si $P = \forall x\ Q^\star\ (C_1, \ldots, C_m)$ et P est vrai, alors $\forall x\ Q^\star\ (C_i)$ est également vrai ; si de plus on a $consist(\sigma_{x=v}(Q^\star\ (C_1, \ldots, C_m))) \subseteq consist(\sigma_{x=v}(Q^\star\ (C_i)))$ pour tout $v \in D_x$, alors on a bien :

$$\bigcup_{x \in D_x} consist(\sigma_{x=v}(Q^\star\ (C_1, \ldots, C_m))) \quad \subseteq \quad \bigcup_{x \in D_x} consist(\sigma_{x=v}(Q^\star\ (C_i)))$$

et (par ajout des $\{(x : v) \mid v \in D_x\}$ à ces deux ensembles), on a obtient :

$$consist(\forall x\ Q^\star\ (C_1, \ldots, C_m)) \subseteq consist(\forall x\ Q^\star\ (C_i))$$

Si $P = \exists x\ Q^\star\ (C_1, \ldots, C_m)$ et P est vrai, alors $\exists x\ Q^\star\ (C_i)$ est également vrai ; si de plus on a $consist(\sigma_{x=v}(Q^\star\ (C_1, \ldots, C_m))) \subseteq consist(\sigma_{x=v}(Q^\star\ (C_i)))$ pour tout $v \in D_x$, alors on a bien :

$$\{A = \sigma_{x=v}(Q^\star(C_1, \ldots, C_m)) \mid A \text{ est vrai } \} \quad \subseteq \quad \{B = \sigma_{x=v}(Q^\star(C_i)) \mid B \text{ est vrai } \}$$

$$\bigcup_{x \in D_x} \left\{ consist(A) \;\middle|\; \begin{array}{c} A = \sigma_{x=v}(Q^\star\ (\bigwedge C_i)) \\ A \text{ est vrai} \end{array} \right\} \subseteq \bigcup_{x \in D_x} \left\{ consist(B) \;\middle|\; \begin{array}{c} B = \sigma_{x=v}(Q^\star\ (C_i)) \\ B \text{ est vrai} \end{array} \right\}$$

et (par ajout des $\{(x : v) \mid v \in D_x\}$ à ces deux ensembles), on a bien :

$$consist(\exists x\ Q^\star\ (C_1, \ldots, C_m)) \subseteq consist(\exists x\ Q^\star\ (C_i))$$

□

Cette propriété permet de déterminer des valeurs inconsistantes par analyse de chacun des sous-problèmes $Q^\star\ C_i$ du QCSP initial. L'intérêt de raisonner sur un QCSP portant sur une seule contrainte est que le nombre de variables mises en jeu est moins important (les techniques d'introduction de variables existentielles introduites dans le chapitre 3 permettent de décomposer n'importe quel problème numérique ou booléen en contraintes *ternaires*). La complexité du calcul de valeurs inconsistantes est ainsi réduite. On se repose en fait sur le résultat suivant :

Proposition 6. (déquantification)

Soit un QCSP $P = \langle \mathcal{X}, D, C \rangle$, où C est une χ-relation. Si $x \notin \chi$, alors on peut ignorer cette variable du problème et raisonner sur le QCSP $Q = \langle \mathcal{X}', D, C \rangle$, où $\mathcal{X}'$ est obtenu par suppression de la variable x. On a alors :

$$inconsist(P) \quad = \quad inconsist(Q)$$

Démonstration. Si P est de forme $Qx\ Q^\star(C)$ où C ne porte pas sur la variable x, *toutes les branches* $Q^\star\sigma_{x=v}(C)$ *ont la même valeur de vérité*. Que le quantificateur soit existentiel ou universel, soit cette valeur de vérité est vraie et toutes les $(x : v)$ sont consistants, soit cette valeur est fausse et la formule est alors fausse. □

Par exemple, pour calculer l'ensemble des valeurs inconsistantes du QCSP $\forall x\ \exists y\ \forall z\ \exists u\ (y = 1)$, il suffit de raisonner sur $\exists y\ (y = 1)$. L'exemple suivant récapitule les bénéfices de l'approche par décomposition de problème.

Exemple 6.6. *Considérons le QCSP :*

$$\begin{array}{l}\exists x_0\ \forall y_1\ \exists x_1\ \ldots\ \forall y_9\ \exists x_9 \\ \quad \underline{\exists s_0\ \exists s_1\ \ldots\ \exists s_{18}} \quad s_0 = x_0\ \wedge\ s_{18} = 100\ \wedge \\ \qquad \bigwedge_{i\in 1..9}\left(\ s_{2i-1} = s_{2i-2} + y_i\ \wedge\ s_{2i} = s_{2i-1} + x_i\ \right)\end{array}$$

On a vu qu'il correspondait à la formule suivante (la transformation entre la formulation "fonctionnelle" et le QCSP est présentée dans le chapitre 5) :

$$\exists x_0\ \forall y_1\ \exists x_1\ \ldots\ \forall y_9\ \exists x_9\ \left(\ x_0 + \textstyle\sum_{i\in 1..9}(y_i + x_i)\ \right) = 100$$

Le QCSP comportant 38 variables dont les domaines possèdent au minimum 10 éléments, déterminer l'ensemble des valeurs consistantes du QCSP requière l'exploration d'un arbre de recherche énorme, comportant plus de 10^{38} *nœuds. En revanche, il est possible de raisonner localement sur chacun des "sous-problème" suivants :*

$$\begin{array}{ll}\exists x_0\ \exists s_0\ (x_0 = s_0), & \exists s_{18}\ (s_{18} = 100), \\ \exists s_0\ \forall y_1\ \exists s_1\ (s_1 = s_0 + y_1), & \exists s_1\ \exists x_1\ \exists s_2\ (s_2 = s_1 + x_1) \\ \exists s_2\ \forall y_2\ \exists s_3\ (s_3 = s_2 + y_2), & \exists s_3\ \exists x_2\ \exists s_4\ (s_4 = s_3 + x_2) \\ \ldots & \\ \exists s_{16}\ \exists y_9\ \exists s_{17}\ (s_{17} = s_{16} + y_9) & \exists s_{17}\ \exists x_9\ \exists s_{18}\ (s_{18} = s_{17} + x_9)\end{array}$$

Ces problèmes correspondent à chacune des 20 contraintes $x+y = z$ *de la formulation QCSP. Chacune comporte 3 variables, et il est donc possible d'en construire l'arbre des preuves possibles. Or les valeurs consistantes de chaque QCSP ainsi obtenu donnent une approximation de celles du QCSP.*

On voit bien, également, le rôle de la technique de "déquantification" : inutile de prendre en compte les quantificateurs sur lesquels la contrainte ne porte pas. On associera donc à chacune des m contraintes d'un QCSP $Q^\star(C_1, \ldots, C_m)$ un *opérateur de réduction de problème* dont le rôle est de supprimer les valeurs inconsistantes par rapport à cette contrainte.

Définition 26. (réduction d'un QCSP)

On appellera opérateur de réduction *d'indice* i *la fonction qui, prenant un QCSP* P *en entrée, retourne le QCSP duquel les valeurs inconsistantes par rapport à la* i*-ème contrainte de* P *ont été supprimées :*

$$reduc_i\left(\ Q^\star\ (C_1, \ldots, C_m)\ \right) \quad \stackrel{def}{=} \quad Q^\star\ (C_1, \ldots, C_m) \setminus inconsist(Q^\star(C_i))$$

6.2.3 Propagation

Nous abordons maintenant la notion de *propagation*, centrale dans l'approche programmation par contraintes. On a vu qu'il est possible d'associer à chaque contrainte d'un QCSP un opérateur permettant de supprimer certaines valeurs inconsistantes des domaines des variables existentielles. Reste à savoir comment utiliser au mieux l'*ensemble* de ces opérateurs pour simplifier le problème. Nous commençons par étudier un exemple.

Exemple 6.7. *Chacune des étapes de notre raisonnement est illustrée de manière graphique par la figure 6.7. Considérons le QCSP booléen suivant :*

$$P : \quad \forall x \in \{0,1\}\ \exists y \in \{0,1\}\ \exists z \in \{0,1\}\ \forall t \in \{0,1\}\ \exists u \in \{1\}\ \ (x \vee y = z, y \vee t = u)$$

On peut raisonner séparément sur chacun des QCSP correspondant aux deux contraintes du problème :

$$\begin{array}{ll} A : & \forall x \in \{0,1\}\ \exists y \in \{0,1\}\ \exists z \in \{0,1\} \quad (x \vee y = z) \\ B : & \exists y \in \{0,1\}\ \forall t \in \{0,1\}\ \exists u \in \{1\} \quad (y \vee t = u) \end{array}$$

Dans une première étape, nous pouvons commencer par éliminer les valeurs inconsistantes par rapport au premier sous-problème. Il se trouve que toutes les valeurs sont consistantes par rapport à A, et on n'obtient donc aucune réduction par la première contrainte :

$$reduc_1(P) = P$$

Si, dans une seconde étape, nous considérons maintenant B, nous remarquons que la valeur $(y : 0)$ est inconsistante (voir Figure 6.7). On obtient donc le QCSP simplifié suivant, qui est équivalent à P :

$$P' = reduc_2(P) : \quad \forall x \in \{0,1\}\ \exists y \in \{1\}\ \exists z \in \{0,1\}\ \forall t \in \{0,1\}\ \exists u \in \{1\}\ \ (x \vee y = z, y \vee t = u)$$

Celui-ci se décompose en deux problèmes A' et B' :

$$\begin{array}{ll} A' : & \forall x \in \{0,1\}\ \exists y \in \{1\}\ \exists z \in \{0,1\} \quad (x \vee y = z) \\ B' : & \exists y \in \{1\}\ \forall t \in \{0,1\}\ \exists u \in \{1\} \quad (y \vee t = u) \end{array}$$

Reconsidérons désormais A' (étape 3), nous pouvons éliminer une nouvelle valeur, $(z : 0)$. Le problème se simplifie donc finalement en :

$$P'' = reduc_1(P') : \quad \forall x \in \{0,1\}\ \exists y \in \{1\}\ \exists z \in \{1\}\ \forall t \in \{0,1\}\ \exists u \in \{1\}\ \ (x \vee y = z, y \vee t = u)$$

Les possibilités de réduction de domaines par raisonnement sur chacune des contraintes s'arrêtent là, on a en effet :

$$reduc_1(P'') \ = \ reduc_2(P'') \ = \ P''$$

Notez que nous avons obtenu des réductions de domaine en raisonnant uniquement sur des sous-problèmes sans avoir à étudier le QCSP dans sa globalité. Cette technique se généralise à des problèmes arbitrairement gros. De plus, remarquons la nécessité de *reconsidérer* certaines contraintes plusieurs fois. Par exemple, s'il était initialement impossible d'effectuer la moindre déduction en utilisant A, le fait de supprimer une valeur nous a permis de réutiliser cette contrainte pour calculer des inconsistances.

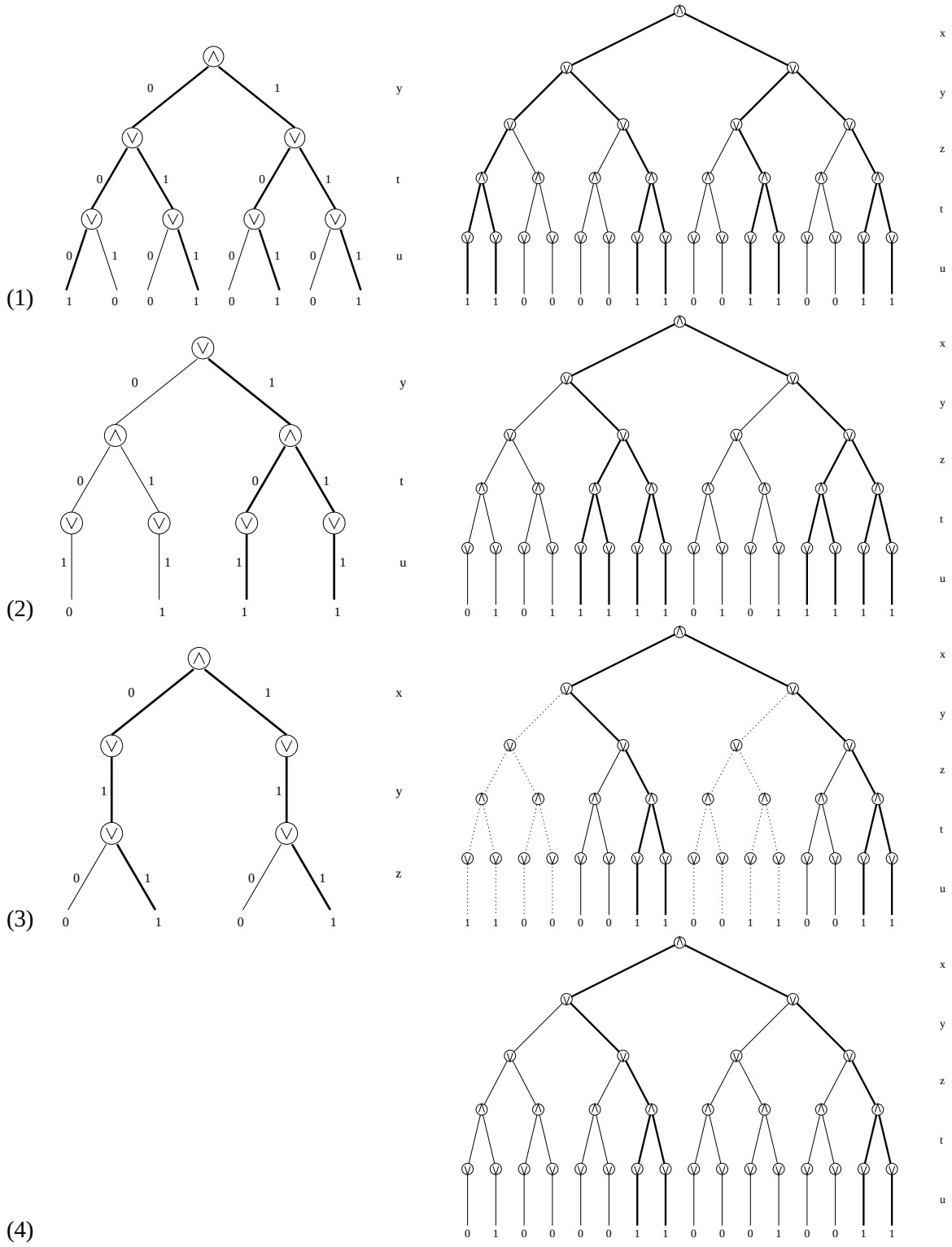


Figure 6.7 – Un exemple de *propagation de contraintes quantifiées* pour le problème $\forall x \in \{0,1\}\ \exists y \in \{0,1\}\ \exists z \in \{0,1\}\ \forall t \in \{0,1\}\ \exists u \in \{1\}\ \ (x \vee y = z, y \vee t = u)$ (voir l'exemple 6.7 pour des commentaires détaillés). En partie droite du dessin on fait apparaître les calculs de valeurs consistantes ; en partie gauche, les calculs réellement nécessaires du fait de la *déquantification*. La dernière figure (4) correspond à l'arbre des preuves possibles du problème, qui se trouve correspondre à l'arbre réduit obtenu par propagation.

Pour utiliser au mieux l'ensemble des opérateurs de réduction associés à un problème, on essaiera donc d'appliquer ceux-ci *tant qu'il est possible d'obtenir de nouvelles réductions de domaine*. On en arrive naturellement à l'algorithme suivant :

Algorithme *Simplifier_par_propagation* $(P = Q^\star(C_1, \ldots, C_m))$

$prop \leftarrow \{Q^\star(C_i) \mid i \in 1 \ldots m\}$
tant que $prop \neq \varnothing$ **faire**
  $prop \leftarrow prop - \{Q^\star(C_i)\}$ % *choix arbitraire d'une contrainte* %
  $Suppr \leftarrow incons(Q^\star(C_i))$
  si $(Suppr \neq \varnothing)$ **alors**
    $P \leftarrow P \backslash Suppr$ % *on applique* $reduce_i$ %
    pour tout $\{x \mid (x : v) \in Suppr\}$ **faire**
      $prop \leftarrow prop \cup \{Q^\star(C_i) \mid Q^\star(C_i)$ "dépend de" $x\}$
    fin pour
  fin si
fin tant que

Fin Algorithme

On n'a pas spécifié ici la structure de données utilisée pour mettre en œuvre le *propagateur* $prop$ — on utilisera en général une *file*. La relation "dépend de" signifie que la contrainte $Q^\star(C_i)$ doit être reconsidérée si le domaine de la variable x est modifiée. Ce doit être le cas si C_i est une χ-relation et $x \in \chi$; sinon, la déquantification nous permet de ne pas reconsidérer les valeurs consistantes de $Q^\star(C_i)$.

Il est évident que cet algorithme permet d'obtenir une certaine réduction de domaines (on n'a cependant aucune garantie que celle-ci ne soit pas nulle). Chacune de ces réductions correspond à l'application d'un opérateur de réduction qui préserve l'équivalence avec le problème initial, on a donc bien une réduction *correcte* dans ce sens. Il est également assez simple de montrer que cet algorithme termine ; nous allons faire mieux en analysant sa complexité. Reste une question à résoudre : celle du non-déterminisme de l'algorithme, qui ne précise pas l'ordre dans lequel les opérateurs sont appliqués. Or on peut se demander si tous les ordres d'application aboutissent au même résultat. La réponse est *oui*.

6.2.3.1 Confluence

On rappelle que la *confluence* est la propriété d'un système spécifié de façon non-déterministe de calculer en fait un résultat unique. Un cadre idéal pour démontrer la confluence des algorithmes de propagation est celui des *itérations chaotiques* de APT [6], dont nous avons rappelé un certain nombre de résultats dans le chapitre 4. Ce cadre consiste à identifier les propriétés des opérateurs mis en jeu par rapport à une certaine relation d'ordre. Nous utiliserons la relation suivante :

Définition 27. (ordre existentiel sur les QCSP)

*On définit la relation d'*ordre (partiel) existentiel *entre QCSP par :*

$$\langle \mathcal{X}, D, C \rangle \quad \preceq_\exists \quad \langle \mathcal{X}', D', C' \rangle$$

ssi $\mathcal{X} = \mathcal{X}'$, $C = C'$, *et si, pour toute variable* x, *on a :*

$$\begin{array}{lcll} D_x & \subseteq & D'_x & \text{si } x \text{ est existentielle} \\ D_x & = & D'_x & \text{si } x \text{ est universelle} \end{array}$$

Cette relation modélise le fait que $A \preceq_\exists B$ si A est obtenu par *réduction des domaines existentiels* de B. Par exemple, ayant deux QCSP $A = \forall x \in \{1,3\}\ \exists y \in \{1,3\}\ (x+y=4)$ et $B = \forall x \in \{1,3\}\ \exists y \in \{1,3,5\}\ (x+y=4)$ on a bien $A \preceq_\exists B$. Il est évident que $\preceq_\exists$ est bien une relation d'ordre *partiel*. Les opérateurs de réduction possèdent les propriétés suivantes, classiques en programmation par contraintes :

Proposition 7. *Les opérateurs $reduc_i$ (définition 26) sont des opérateurs de clôture idempotents, c.-à-d. qu'ils possèdent les caractéristiques suivantes :*

$$\begin{array}{rcll} reduc_i(A) & \preceq_\exists & A & (\textit{contraction}) \\ reduc_i(reduc_i(A)) & = & reduc_i(A) & (idempotence) \\ A \preceq_\exists B & \rightarrow & reduc_i(A) \preceq_\exists reduc_i(B) & (monotonie) \end{array}$$

Démonstration. Il est clair que les opérateurs de *réduction* de domaines sont contractants. De même, l'idempotence est évidente puisqu'on peut facilement se convaincre que

$$inconsist(A \backslash inconsist(A)) = inconsist(A)$$

Nous détaillons la preuve de monotonie, qui est un peu moins triviale et que l'on pourra de plus comparer aux résultats de non-monotonie sur les variables universelles. Procédons par induction sur le nombre de quantificateurs des QCSP. Si A et B sont *faux* ou non quantifiés, l'inclusion est trivialement vraie.

Comparons deux QCSP $A = \forall x\ A'$ et $B = \forall x\ B'$, avec $A \preceq_\exists B$. Pour chaque valeur $v \in \mathcal{D}$, on a $\sigma_{x=v}(A') \preceq_\exists \sigma_{x=v}(B')$ et donc, par application de l'hypothèse d'induction, $consist(\sigma_{x=v}(A')) \subseteq consist(\sigma_{x=v}(B'))$. Il s'en suit que :

$$\bigcup_{v \in D_x} consist(\sigma_{x=v}(A')) \subseteq \bigcup_{v \in D_x} consist(\sigma_{x=v}(B'))$$

Par ajout des $\{(x:v) \mid v \in D_x\}$ des deux côtés de cette inclusion, on a bien $consist(A) \subseteq consist(B)$ et donc $reduc_i(A) \subseteq reduc_i(B)$.

Le cas existentiel est similaire, en remarquant de plus que :

$$\{\sigma_{x=v}(A') \mid \sigma_{x=v}(A') \text{ est vrai } \} \subseteq \{\sigma_{x=v}(B') \mid \sigma_{x=v}(B') \text{ est vrai } \}$$

□

On a vu dans le chapitre 4 que tout algorithme calculant une itération d'opérateurs contractants et monotones est confluente : on calcule toujours le *plus grand point fixe du jeu d'opérateurs* inférieur au QCSP de départ de l'itération. En d'autres termes, en définissant la notion d'*arc-consistance* comme suit :

Définition 28. (Arc-consistance quantifiée)

Un QCSP $P = Q^\star(C_1, \ldots, C_m)$ est arc-consistant *si pour tout $i \in 1 \ldots m$, on a*

$$P \preceq_\exists reduc_i(P)$$

le résultat de la réduction (ou *filtrage*) d'un QCSP P par l'ensemble des m opérateurs de réduction est le plus grand QCSP $P' \preceq_\exists P$ qui soit arc-consistant.

6.2.3.2 Complexité

Soit d la taille du plus grand domaine de variables, k le nombre maximum de variables mises en jeu dans une contrainte du problème et m le nombre de ces contraintes, et n le nombre de variables existentielles du problème. Dans le pire des cas, l'algorithme supprime une à une les nd valeurs des différents domaines de variables, et dans le pire des cas il faut reconsidérer chacune des contraintes pour détecter l'inconsistance d'une valeur. On devra alors calculer mnd fois les valeurs consistantes d'une des contraintes, ce qui nécessite à chaque fois une exploration de coût $\mathcal{O}(d^k)$. Le coût total est donc $\mathcal{O}(mnd^{k+1})$ soit, par exemple, $\mathcal{O}(mnd^4)$ dans le cas de contraintes ternaires obtenues par une décomposition standard de contraintes numériques et booléennes.

Notons que l'algorithme est non optimisé, on pourrait probablement s'épargner de nombreux recalculs grâce au stockage d'information supplémentaires. On pourra notamment s'inspirer de la littérature existant sur l'arc-consistance classique, binaire (AC4 [200], AC5 [249], AC6 [24], AC7 [25], et les récentes variantes optimales d'AC3 [264, 27]) ou généralisée ([201, 23]).

6.3 Algorithme de recherche-élagage

L'arc-consistance quantifiée ne constitue pas un algorithme de *résolution* de contraintes quantifiées. Lorsqu'elle détermine qu'aucune valeur n'est arc-consistante, le problème est prouvé inconsistant ; en revanche, le fait que certaines valeurs ne soient pas éliminées ne signifie pas que le QCSP considéré soit vrai, car on s'est contentés de raisonner localement : l'algorithme est dit *incomplet*.

L'algorithme d'évaluation naïve présenté en début de ce chapitre est quant à lui complet, puisqu'il explore intégralement l'espace de solutions. On peut tirer le meilleur des deux algorithmes en entrelaçant arc-consistance et exploration exhaustive de l'arbre de recherche du problème. À chaque nœud de l'arbre de recherche, on propage ainsi les inconsistances causées par la dernière instantiation de variable ; ce raisonnement étant peu coûteux, on espère que le surcoût en temps de calcul sera marginal. En revanche, on espère ainsi élaguer le maximum de branches et éviter d'avoir à développer le nombre exponentiel de nœuds de l'arbre de recherche global. Ce type d'algorithmes, déjà mentionnés dans de chapitre 4, est dit de *recherche-élagage*[2].

6.3.1 Élagage des domaines universels

Une limitation importante de notre algorithme de recherche-élagage est qu'il permet uniquement d'éliminer les branches correspondant à des valeurs existentielles. Il est indispensable de réduire également les domaines des variables universelles, sinon le temps d'exploration est amené à croître de manière exponentielle par rapport au nombre de quantificateurs universels.

Il est en fait simple d'adapter la technique de propagation existentielle aux variables universelles : il suffit d'appliquer la technique à la *négation* du problème initial. On rappelle que la négation d'une formule prénexe s'obtient par simple inversion des quantificateurs et négation de la matrice (cf. chapitre 3). Dans cette négation, les variables universelles deviennent existentielles et réciproquement, et l'arc-consistance s'applique donc directement[3]. On manipulera donc en parallèle deux QCSP : le premier, dit

[2] *Branch and prune.*

[3] Il me parait intéressant de reporter ici une remarque d'un relecteur anonyme de la conférence CP, faisant part d'un certain scepticisme à l'égard d'une approche "réfutationnelle" des quantificateurs. Ce scepticisme est légitime car il est vrai qu'il est plus difficile de prouver un énoncé universel qu'un énoncé existentiel. Cependant, j'aimerais apporter les éléments de réponse suivants :

QCSP *primal*, représente le problème tel qu'il est formulé ; le second représente la négation du primal, il sera appelé QCSP dual.

Proposition 8. *Si une affectation* $(x : v)$ *est inconsistante dans le dual, il est inutile d'explorer le sous-arbre correspondant : celui-ci est* vrai.

Démonstration. Quelle que soit la branche explorée, la branche correspondante dans l'arbre du problème dual correspond à la *négation* du sous-problème. Si la négation est fausse, la sous-formule primale — et la branche correspondante — est donc vraie. □

6.3.2 Incrémentalité

6.3.2.1 Élagage des domaines universels et monotonie

Dans notre présentation de l'arc-consistance quantifiée, nous nous sommes pour l'instant prudemment gardés d'autoriser la réduction des domaines des variables universelles. La raison en est simple : de telles réductions ne possèdent pas les propriétés de monotonie exigées par la propagation de contraintes, et il faut donc la plus grande méfiance à l'égard de ce type d'opérations. Les problèmes pouvant apparaître sont illustrés par l'exemple suivant :

Exemple 6.8. *Considérons le QCSP :*

$$\forall x \in 1..3\ \exists y \in 1..4\ (x = y + 1)$$

Ce QCSP est vrai, *et seule la valeur* $(y : 1)$ *est inconsistante. Réduisons maintenant le domaine de la variable universelle* x *:*

$$\forall x \in 1..2\ \exists y \in 1..4\ (x = y + 1)$$

La formule demeure vraie, *mais il y a désormais deux valeurs inconsistantes :* $(y : 1)$ *et* $(y : 4)$*. Il serait toutefois naïf de penser que le nombre de valeurs inconsistantes augmente systématiquement lorsqu'on réduit les domaines universel. Par exemple, élargissons le domaine de* x *:*

$$\forall x \in 1..4\ \exists y \in 1..4\ (x = y + 1)$$

- Si les énoncés universels (par exemple des instances SAT insatisfaisables) sont a priori difficiles, l'approche consistant à appliquer un résolveur complet sur la négation de ces problèmes et à montrer l'absence de solutions est pourtant parfaitement adaptée. Par exemple, certaines des instances utilisées lors des récentes compétitions SAT sont des instances insatisfaisables de grande taille modélisant notamment des problèmes de *model-checking*. Or les résolveurs complets basés sur la recherche-élagage, comme `zchaff` [205, 267], s'avèrent très compétitifs sur ce type d'instances (merci à Inês Lynce, Lyndon Drake, Laurent Simon et Lintao Zhang pour les renseignements qu'ils m'ont apportés sur les résolveurs SAT au cours de nos discussions).
- Pour ce qui concerne les problèmes arbitrairement quantifiés, prouver ou réfuter la négation d'un QCSP n'est de toute façon ni plus difficile, ni plus facile que le problème initial. Contrairement au cas des contraintes classiques, existentielles, le passage à la négation ne change pas la nature du problème (**PSPACE** est clos par complément!) — il est vrai qu'en contrepartie, les contraintes quantifiées représentent un problème à la fois plus dur que les problèmes universels et que les problèmes existentiels.
  On a imposé que la matrice (ensemble de contraintes) d'un QCSP soit exprimée sous forme d'une conjonction et, par passage à la négation, on obtient donc essentiellement une disjonction de contraintes (celle-ci pouvant bien entendu à nouveau être exprimée sous forme conjonctive grâce aux techniques présentées dans le chapitre 3). Les disjonctions sont plus faciles à prouver vraies et les conjonctions à réfuter ; selon le type de quantification, rien n'indique que l'une ou l'autre soit préférable.

Enfin, se pose le problème d'exprimer la *négation* d'une contrainte. Ce problème, critique dans le cas des domaines continus (sur lesquels la contrainte $\neq$ est notamment difficile à utiliser), ne se pose pas sur les domaines discrets.

on obtient une formule fausse : aucune valeur n'est consistante!

Il serait donc vain de rechercher des résultats de monotonie ou d'anti-monotonie formulés de manière générale pour les variables universelles d'un QCSP. On préconisera donc l'approche consistant à *ne jamais réduire le domaine des variables universelles* pendant la propagation. Dans le problème primal, on conserve ainsi les domaines universels initiaux, et la propagation a pour but de réduire les domaines des variables existentielles du problème, dont on sait qu'elles possèdent les bonnes propriétés de monotonie. De même, dans le problème dual, on conservera toujours les domaines non réduits des variables universelles (existentielles du primal), et on réduira les existentielles de ce problème (universelles du primal). Chaque variable possède donc deux domaines : l'un dans le problème primal, l'autre dans le problème dual ; un seul de ces domaines est affecté par la propagation (le domaine primal si la variable est existentielle, le domaine dual si elle est universelle). Le tableau suivant récapitule l'effet de la propagation de contraintes sur chacun des domaines maintenus :

	problème primal	problème dual
Variable existentielle	suppression des valeurs inconsistantes	domaine inchangé
Variable universelle	domaine inchangé	suppression des valeurs tautologiques

Remarquons par ailleurs qu'il est légitime de ne pas supprimer du problème primal une valeur d'une variable universelle si celle-ci est éliminée du dual : les valeurs supprimées dans le problème dual ne sont pas inconsistantes, la propagation duale montre simplement qu'*il est inutile d'explorer la branche correspondant à cette valeur*, car le sous-arbre obtenu est *vrai* (on appelle *tautologiques* de telles sous-formules).

6.3.2.2 Incrémentalité

Si la propagation de contraintes laisse les domaines universels inchangés (que ce soit dans le problème primal ou le dual), l'algorithme d'exploration procède en instanciant la variable du quantificateur de tête, que celle-ci soit existentielle ou universelle. L'interaction entre l'algorithme d'exploration et celui de propagation soulève encore certaines questions liées à l'*incrémentalité* des calculs, c.-à-d. au fait que les réductions de domaine obtenues par propagation demeurent valides après instantiation d'une variable.

Si la variable de tête x d'une formule est quantifiée existentiellement, il est clair que le fait d'instancier x fait augmenter le nombre de valeurs inconsistantes, et que les réductions de domaine déjà obtenues peuvent être conservées. Dans le cas d'un quantificateur universel, les propriétés de monotonie ne sont plus vérifiées (cf. exemple 6.8), ce qui suggère *a priori* une plus grande prudence. Rappelons cependant la définition de l'ensemble des valeurs consistantes d'un QCSP dont la variable de tête est universelle :

$$consist(\forall x \in D\ Q^\star(C)) \stackrel{\text{def}}{=} \left(\{(x:v) \mid v \in D\} \cup \bigcup_{v \in D} consist(Q^\star(\sigma_{x=v}(C))) \right)$$

L'ensemble des valeurs consistantes de $Q^\star(\sigma_{x=v}(C))$ est bien inclus dans l'ensemble des valeurs consistantes du problème initial $\forall x \in D\ Q^\star(C)$, ce qui garantit la propriété d'incrémentalité désirée. Notons qu'il est indispensable que le processus d'élimination de quantificateurs suive l'ordre de leur apparition dans le préfixe. On obtient finalement l'algorithme de recherche-élagage présenté en Figure 6.3.2.2.

Algorithme *Recherche_elagage (primal, dual) : booléen*

% *Simplification du problème (élagage)* %
$primal \leftarrow primal \backslash inconsist(primal$
$dual \leftarrow dual \backslash inconsist(primal$

si $(consist(primal) = \varnothing)$ **alors**
  retourner *faux*
fin si
si $(consist(dual) = \varnothing)$ **alors**
  retourner *vrai*
fin si

% *Décomposition du problème (recherche)* %
si $(primal = \exists x \in \mathcal{D}\ Q^{\star}(C))$ **alors**
  retourner $\bigvee_{v \in \mathcal{D}}(Recherche\_elagage(\sigma_{x=v}(primal), \sigma_{x=v}(dual)))$
sinon
  % $dual = \exists x \in \mathcal{D}\ Q^{\star}(C)$ %
  % *Notez qu'on utilise le domaine de* x *issu du problème dual* %
  retourner $\bigwedge_{v \in \mathcal{D}}(Recherche\_elagage(\sigma_{x=v}(primal), \sigma_{x=v}(dual)))$
fin si

% *Cas d'arrêt : le qcsp est non quantifié* %
% $primal$ *et* $dual$ *sont tous deux réduits à la matrice entièrement instanciée* %
retourner $test$(matrice)

Fin Algorithme

6.3.2.3 Complexité

Une borne de compléxité naïve de l'algorithme de recherche-élagage est obtenue à partir de celle de la propagation : le nombre de nœuds de l'arbre de recherche est borné par d^n ($d = |\mathbb{D}|$ est la taille du domaine), et ce coût est multiplié par celui de la propagation ($\mathcal{O}(mnd^4)$), celle-ci étant utilisée à chaque nœud.

Cette borne correspond bien entendu à une estimation grossière du pire cas. Cependant, comme dans le cas des méthodes de propagation de contraintes non quantifiées, on peut exhiber une famille infinie d'instances sur lesquelles le temps de calcul est strictement exponentiel. Un exemple est donné par l'encodage en (Q)CSP des *formules des trous de pigeons* décrites brièvement dans le chapitre 4 — il est clair que notre algorithme de propagation ne se comporte pas mieux que les formes classiques de propagation pour ce type de problème.

6.4 Conclusion et remarques

Dans ce chapitre, nous avons présenté une notion d'*arc-consistance quantifiée* permettant d'éliminer certaines des valeurs des variables existentielles correspondant à des branches dont l'exploration est inutile. Du fait de la possibilité d'effectuer un calcul similaire sur la négation du problème initial, l'algorithme s'applique également à la réduction de domaines universels.

Insistons sur le fait que la technique présentée est bien une généralisation de l'arc-consistance classique, avec laquelle elle coïncide dans le cas de contraintes existentielles : dans ce cas, l'algorithme de calcul de valeurs consistances collecte bien pour chaque variable l'ensemble des valeurs appartenant à une solution. Il est évident que d'autres notions de consistances "fortes" [202, 112, 85], et notamment les consistances "relationnelles" [87] peuvent également être envisagées. Le fait de relier les contraintes quantifiées à un cadre "à la CSP" permet plus généralement d'espérer intégrer au cadre de résolution de contraintes quantifiées la batterie d'outils qui font l'intérêt des CSP par rapport à SAT, parmi lesquelles on peut mentionner les contraintes globales [14, 219]. Un autre avantage essentiel du cadre (Q)CSP par rapport à SAT est la meilleure gestion de l'arithmétique grâce notamment à la propagation d'intervalles ; cet apport est moins prospectif puisque des propositions de techniques de propagation d'intervalles pour les contraintes quantifiées seront proposées dans le chapitre suivant.

Remarquons, enfin, que la définition que nous avons donnée de l'arc-consistance quantifiée est relativement technique (basée sur une analyse des branches utiles de l'arbre de recherche) et moins naturelle que dans le cadre CSP habituel. Contrairement au cas existentiel, la nature des déductions effectuées est en effet plus difficile à exprimer en termes logiques. Le fait de supprimer une valeur v d'un domaine D_x ne saurait être exprimé sous la forme $Q^\star(R \rightarrow x \neq v)$ puisque l'implication $\exists x \in \{0,1\}\ \exists y \in \{0,1\}\ \exists z \in \{0\}\ ((x \vee y = z) \rightarrow x \neq 0)$ est par exemple vraie, alors que la valeur $(x : 0)$ n'est pas inconsistante pour la contrainte $\exists x \in \{0,1\}\ \exists y \in \{0,1\}\ \exists z \in \{0\}\ (x \vee y = z)$[4]. De même, $\forall x \in \{1,3\}\ \forall y \in \{1,2\}\ \exists z\ (x + y = 3)$ est faux alors que $(z : 3)$ n'est pas une valeur inconsistance du problème $\forall x \in \{1,3\}\ \forall y \in \{1,2\}\ \exists z \in \{1,2,3,4,5\}\ (x + y = z)$; une valeur inconsistante $(x : v)$ n'est donc pas caractérisée par le fait que $Q^\star(\sigma_{x=v}(C))$ soit faux.

[4] Merci à François FAGES pour sa lecture attentive qui a permis de déceler un certain nombre d'erreurs dans ces exemples.

CHAPITRE 7

Améliorations de l'arc-consistance quantifiée

L'algorithme de propagation de contraintes proposé en section précédente permet de raisonner localement contrainte par contrainte en parcourant intégralement l'espace de solution de chacune de ces contraintes. Cette méthode n'est pas optimale en général. En particulier, dans le cas des contraintes numériques, dont les variables possèdent des domaines (non booléens) parfois larges, l'énumération exhaustive des valeurs possibles des variables d'une contrainte peut s'avérer coûteuse.
Une solution consiste à reformuler la propagation de contraintes en termes de règles de calculs simples à appliquer. On obtient un ensemble fini de propagateurs *qui servent de langage cible vers lequel les QCSP peuvent être "compilés". Dans le cas des problèmes quantifiés numériques, on peut ainsi remplacer l'emploi de domaines de valeurs représentés en extension par des* intervalles *plus compacts en mémoire, et sur lesquels la plupart des opérations peuvent être mises en œuvre en temps constant, même si elles sacrifient une partie de la précision initiale et calculent en fait une approximation de l'arc-consistance.*

7.1 Propagateurs pour les contraintes quantifiées

Il existe différentes façons de mettre en œuvre les techniques de consistance locale. La plus immédiate consiste, pour chaque contrainte, à calculer par énumération l'ensemble des valeurs inconsistantes par rapport à cette contrainte suivant l'approche proposée en section précédente. Pour certaines contraintes, notamment sur les domaines numériques, cette technique n'est cependant pas la plus appropriée. Le langage de contraintes utilisé dans le cas des QCSP arithmétiques ou booléens comporte un petit nombre d'opérations de base : addition, multiplication, conjonction, etc., et le fait de connaître à l'avance le jeu de contraintes primitives utilisables permet de déterminer des règles de calcul spécifiques à ces primitives particulières.

7.1.1 Règles de propagation

De nombreuses variantes de propagation de contraintes peuvent être exprimées en termes de *règles de propagation* (également appelées *propagateurs*). Les règles que nous utiliserons seront de simples clauses de Horn, dont la forme sera donc :

$$\texttt{condition}_1\texttt{, ..., condition}_q \Longrightarrow \texttt{conclusion}$$

Une règle s'*applique* si chacune de ses parties gauches est vérifiée (la virgule est donc lue comme une conjonction), auquel cas la *conclusion* de la règle peut être inférée. La nature exacte des constructions utilisables en condition et en conclusion sera donnée ultérieurement en fonction du domaine considéré. À ce point de vue opérationnel s'ajoute une deuxième interprétation, déclarative, consistant à voir une règle comme un opérateur monotone sur un certain domaine de calcul.

La *programmation par règles* est un style de programmation qui trouve notamment des applications dans les bases de données déductives ou autres systèmes de déduction [254]. Différents cadres à base de règles ont été proposés pour exprimer la propagation par contraintes, les plus connus étant probablement les CHR (*Constraint Handling Rules* [116]). Des idées assez proches peuvent également être remarquées, notamment, dans le langage Claire [51], ou dans les travaux sur les *contraintes concurrentes* [251] ou réactives [187], dans lesquels un lien explicite est établi entre propagation de contraintes et programmation événementielle. Remarquons que les clauses de Horn que nous nous proposons d'utiliser dans ce chapitre sont un cas particulièrement restreint de règles[1]. Notre but sera d'extraire pour chaque contrainte primitive du problème un ensemble de règles permettant d'exprimer les déductions obtenues par propagation de contraintes. Les clauses de Horn forment en fait un langage-cible assez proche de ceux utilisés pour la compilation de contraintes "classiques" (quantifiées existentiellement) sur les domaines finis par exemple dans GNU-Prolog[2].

7.1.2 Contraintes primitives

Dans la suite de ce chapitre, nous allons présenter des règles de propagation pour la résolution de contraintes quantifiées sur nos deux principaux domaines d'application : le domaine *booléen* puis celui des *contraintes numériques*. Nous avons vu dans le chapitre 4 qu'il est possible de décomposer ce type de contraintes par introduction de variables existentielles. Cette approche est similaire à celle utilisée dans certains résolveurs comme GNU-Prolog [62, 91], dans lequel un petit nombre de contraintes primitives est utilisé (de forme $x + y = z$, $x \wedge y = z$, etc.).

[1] Les CHR permettent d'exprimer plusieurs types de règles, certaines représentant l'équivalence, d'autres l'implication logique. Se restreindre à des clauses de Horn masque une subtilité opérationnelle (d'importance mineure dans le cas des règles que nous considérerons) : dans le cas d'équivalences, il est possible de *remplacer* un ensemble de faits par la conclusion (et non pas seulement d'ajouter des conclusions de manière monotone). Mais le principal apport des CHR en termes d'expressivité provient de la possibilité d'associer des règles à un *ensemble de contraintes*, alors que dans notre cas chaque règle correspond à une seule contrainte. Par exemple, les CHR permettent d'exprimer le raisonnement suivant : *s'il existe dans l'ensemble de contraintes du problème deux contraintes $x \leq y$ et $y \leq x$, alors on peut (sans condition) déduire que x et y sont en fait égales*. En syntaxe CHR, ce raisonnement s'exprime comme suit :

$$\texttt{x} \leq \texttt{y, y} \leq \texttt{x | true ==> x} = \texttt{y}$$

Notez que cette règle de déduction est en fait précisément un exemple de règle d'équivalence : il est possible de *remplacer* les deux contraintes d'inégalité par l'égalité ; on pourrait donc utiliser une règle de *remplacement*, utilisant le symbole `<=>`.

[2] GNU-Prolog [62] permet de décrire les algorithmes de propagation de contraintes en spécifiant comment les domaines de chaque variable sont (re)calculés en fonction des autres variables de problème ; par exemple, la contrainte $x \leq y$ est compilée de la manière suivante :

$$\texttt{x in } -\infty \texttt{ .. } \max(\texttt{y})$$
$$\texttt{y in } \min(\texttt{x}) \texttt{ .. } +\infty$$

Les dernières versions du système intègrent certaines extensions permettant par exemple de décrire des contraintes globales et des contraintes réifiées [91]. L'ajout des primitives de type "`x in r`" au jeu d'instructions de la *machine abstraite de Warren* permet ainsi une véritable *compilation* de programmes CLP(FD) en code machine [3]. Les expressions de type `x in r` correspondent en fait à la partie *déduction* des clauses de Horn que nous utiliserons dans ce chapitre.

Dans notre cas, une particularité vient s'ajouter, puisque nous devons gérer les quantificateurs. On doit donc multiplier le nombre de primitives du langage de contraintes, pour tenir compte des différents motifs de quantificateurs qu'il est possible d'obtenir. En nous ramenant à des contraintes (au plus) ternaires, nous obtenons 8 quantifications possibles (par exemple pour la primitive d'addition) :

$$\begin{array}{l}\exists x\ \exists y\ \exists z\ \ (x+y=z)\\ \exists x\ \exists y\ \forall z\ \ (x+y=z)\\ \exists x\ \forall y\ \exists z\ \ (x+y=z)\\ \exists x\ \forall y\ \forall z\ \ (x+y=z)\\ \forall x\ \exists y\ \exists z\ \ (x+y=z)\\ \forall x\ \exists y\ \forall z\ \ (x+y=z)\\ \forall x\ \forall y\ \exists z\ \ (x+y=z)\\ \forall x\ \forall y\ \forall z\ \ (x+y=z)\end{array}$$

Nous allons donc proposer, pour chaque opération booléenne ou arithmétique élémentaire et pour chaque quantification possible des contraintes primitives liées à cette opération, un ensemble de règles de propagation permettant de réduire les domaines de ses valeurs existentielles.

7.1.3 Correction et complétude

Deux propriétés essentielles permettent de garantir l'adéquation entre un ensemble de règles et la définition originale de l'arc consistance (voir [7] pour une présentation de ces propriétés dans le contexte de la génération automatique de règles) :

La correction d'une règle signifie que les valeurs détruites par déclenchement des règles sont bien des valeurs inconsistantes ;

La complétude d'un *ensemble* de règles est vérifiée si toutes les valeurs inconsistantes sont détectées par application de l'ensemble de règles.

Déterminer des jeux de règles corrects et complets permet de remplacer une contrainte par un jeu de règles et d'effectuer tous les raisonnements de propagation à partir du jeu de règles en question — cette condition est donc nécessaire pour pouvoir parler de *compilation*.

7.2 Propagation booléenne

Des exemples de déduction obtenues par la propagation de contraintes booléennes sont les suivantes (le symbole $\leadsto$ représente ici une instantiation due à la détection d'une valeur non arc-consistante) :

$$\begin{array}{lcl}\exists x \in \{0,1\}\ \forall y \in \{0,1\}\ \exists z \in \{1\}\ (x \vee y = z) & \leadsto & x = 1\\ \forall x \in \{0,1\}\ \exists y \in \{1\}\ \exists z \in \{0,1\}\ (x \vee y = z) & \leadsto & z = 1\\ \exists x \in \{0,1\}\ \exists y \in \{1\}\ \forall z \in \{0,1\}\ (x \vee y = z) & \leadsto & \textit{faux}\end{array}$$

Deux types de déduction sont possibles, l'une consiste à détecter que la contrainte est *fausse*, l'autre consiste à *instantier* une variable à une certaine valeur, excluant ainsi l'autre valeur booléenne de son domaine. Les règles de calcul des contraintes booléennes s'obtiennent directement depuis les tables donnant, pour chaque domaine de chacune des variables de la contrainte, l'ensemble des valeurs inconsistantes.

7.2.1 Traduction des primitives booléennes

La faible taille des domaines booléens permet de précalculer les conséquences obtenues pour chaque contrainte primitive et pour chaque sous-domaine possible. On obtient des tables de calcul donnant, pour chaque domaine courant de chacune des trois variables, soit la valeur *faux* si la contrainte obtenue est inconsistante pour ces domaines, soit les éventuelles réductions de domaine touchant des variables existentielles du problème (soit rien si la contrainte est consistante et qu'aucune déduction n'est possible).

D_x	D_y	$(f = x_\neg)\ f_\forall\, x_\forall$	$(f = x_\neg)\ f_\exists\, x_\forall$	$(f = x_\neg)\ f_\forall\, x_\exists$	$(f = x_\neg)\ f_\exists\, x_\exists$
$\{0\}$	$\{0\}$	*faux*	*faux*	*faux*	*faux*
	$\{1\}$				
	$\{0,1\}$	*faux*	$y = 1$	*faux*	$y = 1$
$\{1\}$	$\{0\}$				
	$\{1\}$	*faux*	*faux*	*faux*	*faux*
	$\{0,1\}$	*faux*	$y = 0$	*faux*	$y = 0$
$\{0,1\}$	$\{0\}$	*faux*	*faux*	$x = 1$	$x = 1$
	$\{1\}$	*faux*	*faux*	$x = 0$	$x = 0$
	$\{0,1\}$	*faux*		*faux*	

Figure 7.1 – Table des déductions obtenues pour les différentes variantes de la contrainte de négation.

La table des différentes variantes de la contrainte de négation est donnée par la figure 7.1 ; celle des variantes de la disjonction par la figure 7.2. (nous nous contenterons de décrire ici le jeu d'opérateurs complet $\{\neg, \vee\}$, le principe étant le même pour les autres contraintes). On remarquera que le contenu de ces tables, laborieux à générer à la main, peut être automatisé pour chaque contrainte par énumération des sous-domaines possibles et calcul des valeurs inconsistantes[3].

7.2.2 Expression de la propagation booléenne sous forme de règles

Le calcul d'une table de déductions permet d'exprimer les critères de correction et de complétude de la manière suivante :

Correction : si les domaines des variables permettent d'appliquer une règle, les déductions de celle-ci doivent apparaître dans la table à la case correspondante ;

Complétude : toutes les déductions obtenues pour une certaine case de la table doivent être obtenues par application de l'ensemble des règles appliquées sur les mêmes domaines.

Fixons par ailleurs le langage utilisé pour les règles de propagation booléenne. Celui-ci utilise essentiellement des *tests d'appartenance*, validés si une certaine valeur booléenne est incluse ou exclue du domaine d'une variable. La grammaire des règles utilisées est la suivante (ϵ désigne le mot vide) :

[3] Une perspective intéressante est donc d'appliquer les techniques de génération décrites dans [7]. Celles-ci présentent un intérêt dans le cas d'applications utilisant des contraintes sur des domaines finis de taille limitée.

D_x	D_y	D_z	$(z = f \wedge x)\ z^{\forall}\ f^{\forall}\ x^{\forall}$	$(z = f \wedge x)\ z^{\exists}\ f^{\forall}\ x^{\forall}$	$(z = f \wedge x)\ z^{\forall}\ f^{\exists}\ x^{\forall}$	$(z = f \wedge x)\ z^{\exists}\ f^{\exists}\ x^{\forall}$	$(z = f \wedge x)\ z^{\forall}\ f^{\forall}\ x^{\exists}$	$(z = f \wedge x)\ z^{\exists}\ f^{\forall}\ x^{\exists}$	$(z = f \wedge x)\ z^{\forall}\ f^{\exists}\ x^{\exists}$	$(z = f \wedge x)\ z^{\exists}\ f^{\exists}\ x^{\exists}$
$\{0\}$	$\{0\}$	$\{0\}$								
		$\{1\}$	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*
		$\{0,1\}$	*faux*	$z = 0$	*faux*	$z = 0$	*faux*	$z = 0$	*faux*	$z = 0$
	$\{1\}$	$\{0\}$	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*
		$\{1\}$								
		$\{0,1\}$	*faux*	$z = 1$	*faux*	$z = 1$	*faux*	$z = 1$	*faux*	$z = 1$
	$\{0,1\}$	$\{0\}$	*faux*	*faux*	$y = 0$	$y = 0$	*faux*	*faux*	$y = 0$	$y = 0$
		$\{1\}$	*faux*	*faux*	$y = 1$	$y = 1$	*faux*	*faux*	$y = 1$	$y = 1$
		$\{0,1\}$	*faux*		*faux*		*faux*		*faux*	
$\{1\}$	$\{0\}$	$\{0\}$	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*
		$\{1\}$								
		$\{0,1\}$	*faux*	$z = 1$	*faux*	$z = 1$	*faux*	$z = 1$	*faux*	$z = 1$
	$\{1\}$	$\{0\}$	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*
		$\{1\}$								
		$\{0,1\}$	*faux*	$z = 1$	*faux*	$z = 1$	*faux*	$z = 1$	*faux*	$z = 1$
	$\{0,1\}$	$\{0\}$	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*
		$\{1\}$								
		$\{0,1\}$	*faux*	$z = 1$	*faux*	$z = 1$	*faux*	$z = 1$	*faux*	$z = 1$
$\{0,1\}$	$\{0\}$	$\{0\}$	*faux*	*faux*	*faux*	*faux*	$x = 0$	$x = 0$	$x = 0$	$x = 0$
		$\{1\}$	*faux*	*faux*	*faux*	*faux*	$x = 1$	$x = 1$	$x = 1$	$x = 1$
		$\{0,1\}$	*faux*		*faux*		*faux*		*faux*	
	$\{1\}$	$\{0\}$	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*
		$\{1\}$								
		$\{0,1\}$	*faux*	$z = 1$	*faux*	$z = 1$	*faux*	$z = 1$	*faux*	$z = 1$
	$\{0,1\}$	$\{0\}$	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	$x = y = 0$	$x = y = 0$
		$\{1\}$	*faux*	*faux*	$y = 1$	$y = 1$	$x = 1$	$x = 1$		
		$\{0,1\}$	*faux*		*faux*		*faux*		*faux*	

Figure 7.2 – Table des déductions obtenues pour chaque variante de la contrainte de disjonction.

```
    const  ::=  0 | 1
condition  ::=  const ∈ x | const ∉ x
    garde  ::=  ϵ | condition [, condition]*
conclusion ::=  x = const | faux
    règle  ::=  garde ⟹ conclusion
```

La *garde* de chaque règle est une conjonction (éventuellement vide) de tests d'appartenance d'une constante à une variable ; la conclusion obtenue est soit l'inconsistance, soit l'instantiation d'une variable. On remarquera que la propagation booléenne sur des contraintes existentielles est classiquement exprimée par un jeu de règles similaires mais n'utilisant pas les conditions de forme `cond` $\notin x$ (voir [116, 7]). L'ajout de ce type de constructions soulève quelques subtilités liées à la propriété de *monotonie* que nous discuterons à la fin de cette section.

Exemple 7.9. *La contrainte* $\exists x\, \forall y\, \exists z\, (x \vee y = z)$ *est représentée par l'ensemble de règles suivantes :*

$$\begin{array}{lrcl}
(1) & 1 \in D_y, 1 \notin D_z & \Longrightarrow & \textit{faux} \\
(2) & 1 \notin D_z & \Longrightarrow & x = 0 \\
(3) & 0 \in D_y, 0 \notin D_z & \Longrightarrow & x = 1 \\
(4) & 1 \notin D_x, 1 \notin D_y & \Longrightarrow & z = 0 \\
(5) & 0 \notin D_x & \Longrightarrow & z = 1 \\
(6) & 0 \notin D_y & \Longrightarrow & z = 1
\end{array}$$

Vérifier que ce jeu de règles est bien correct et complet nécessite de considérer chacun des domaines possibles pour x, y *et* z *:*

D_x	D_y	D_z	$1 \in D_y, 1 \notin D_z$	$1 \notin D_z$	$0 \in D_y, 0 \notin D_z$	$1 \notin D_x, 1 \notin D_y$	$0 \notin D_x$	$0 \notin D_y$	*conclusion*
		$\{0\}$		$x = 0$		$z = 0$			
	$\{0\}$	$\{1\}$			$x = 1$	$z = 0$			*faux*
		$\{0,1\}$				$z = 0$			$z = 0$
		$\{0\}$	*faux*	$x = 0$				$z = 1$	*faux*
$\{0\}$	$\{1\}$	$\{1\}$						$z = 1$	
		$\{0,1\}$						$z = 1$	$z = 1$
		$\{0\}$	*faux*	$x = 0$					*faux*
	$\{0,1\}$	$\{1\}$			$x = 1$				*faux*
		$\{0,1\}$							
		$\{0\}$		$x = 0$			$z = 1$		*faux*
	$\{0\}$	$\{1\}$			$x = 1$		$z = 1$		
		$\{0,1\}$					$z = 1$		$z = 1$
		$\{0\}$	*faux*	$x = 0$			$z = 1$	$z = 1$	*faux*
$\{1\}$	$\{1\}$	$\{1\}$					$z = 1$	$z = 1$	
		$\{0,1\}$					$z = 1$	$z = 1$	$z = 1$
		$\{0\}$	*faux*	$x = 0$			$z = 1$		*faux*
	$\{0,1\}$	$\{1\}$			$x = 1$		$z = 1$		
		$\{0,1\}$					$z = 1$		$z = 1$

D_x	D_y	D_z	$1 \in D_y, 1 \notin D_z$	$1 \notin D_z$	$0 \in D_y, 0 \notin D_z$	$1 \notin D_x, 1 \notin D_y$	$0 \notin D_x$	$0 \notin D_y$	*conclusion*
		$\{0\}$		$x = 0$					$x = 0$
	$\{0\}$	$\{1\}$			$x = 1$				$x = 1$
		$\{0,1\}$							
		$\{0\}$	*faux*	$x = 0$				$z = 1$	*faux*
$\{0,1\}$	$\{1\}$	$\{1\}$						$z = 1$	
		$\{0,1\}$						$z = 1$	$z = 1$
		$\{0\}$	*faux*	$x = 0$					*faux*
	$\{0,1\}$	$\{1\}$			$x = 1$				$x = 1$
		$\{0,1\}$							

Dans le tableau précédent, on a reporté dans chaque case la conclusion de la règle (colonne) si la garde de cette règle est vérifiée par le domaine considéré (ligne). La conjonction de chacune des conclusions obtenues pour une ligne donne bien l'ensemble des valeurs inconsistantes pour la contrainte $\exists x\, \forall y\, \exists z\, (x \vee y = z)$ *sur le domaine considéré.*

Aux propriétés de correction et de complétude vient s'ajouter une troisième propriété, désirable pour des raisons opérationnelles, celle de *minimalité* du jeu de règles obtenu [7]. Une règle r est dite *redondante* par rapport à un ensemble de règles E si, dans chaque cas où la garde de r est vérifiée, sa conclusion peut être obtenue par application des règles de E. Vérifier la minimalité d'un jeu de règles consiste donc à vérifier qu'aucune règle ne peut être supprimée sans compromettre la propriété de complétude. Les figures 7.3 et 7.4 donnent respectivement les jeux de règles minimaux obtenus pour chacune des variantes des contraintes de négation et de disjonction.

contrainte	garde		action
$\forall x\, \forall y\ \ (\neg x = y)$	$0 \in D_x, 0 \in D_y$	$\Longrightarrow$	*faux*
	$1 \in D_x, 1 \in D_y$	$\Longrightarrow$	*faux*
$\forall x\, \exists y\ \ (\neg x = y)$	$0 \in D_x, 1 \notin D_y$	$\Longrightarrow$	*faux*
	$1 \in D_x, 0 \notin D_y$	$\Longrightarrow$	*faux*
	$0 \notin D_x$	$\Longrightarrow$	$y = 0$
	$1 \notin D_x$	$\Longrightarrow$	$y = 1$
$\exists x\, \forall y\ \ (\neg x = y)$	$1 \in D_y$	$\Longrightarrow$	$x = 0$
	$0 \in D_y$	$\Longrightarrow$	$x = 1$
$\exists x\, \exists y\ \ (\neg x = y)$	$0 \notin D_y$	$\Longrightarrow$	$x = 0$
	$1 \notin D_y$	$\Longrightarrow$	$x = 1$
	$0 \notin D_x$	$\Longrightarrow$	$y = 0$
	$1 \notin D_x$	$\Longrightarrow$	$y = 1$

Figure 7.3 – Règles de propagation pour $\neg$.

contrainte	action		garde
$\forall x\ \forall y\ \forall z\ \ (x \vee y = z)$	$0 \in D_z, 1 \in D_z$	$\Longrightarrow$	*faux*
	$1 \in D_x, 1 \notin D_z$	$\Longrightarrow$	*faux*
	$1 \in D_y, 1 \notin D_z$	$\Longrightarrow$	*faux*
	$0 \in D_x, 0 \in D_y, 0 \notin D_z$	$\Longrightarrow$	*faux*
$\forall x\ \forall y\ \exists z\ \ (x \vee y = z)$	$1 \in D_y, 1 \notin D_z$	$\Longrightarrow$	*faux*
	$1 \in D_x, 1 \notin D_z$	$\Longrightarrow$	*faux*
	$0 \in D_x, 0 \in D_y, 0 \notin D_z$	$\Longrightarrow$	*faux*
	$1 \notin D_x, 1 \notin D_y$	$\Longrightarrow$	$z = 0$
	$0 \notin D_x$	$\Longrightarrow$	$z = 1$
	$0 \notin D_y$	$\Longrightarrow$	$z = 1$
$\forall x\ \exists y\ \forall z\ \ (x \vee y = z)$	$0 \in D_z, 1 \in D_z$	$\Longrightarrow$	*faux*
	$1 \in D_x, 1 \notin D_z$	$\Longrightarrow$	*faux*
	$1 \notin D_z$	$\Longrightarrow$	$y = 0$
	$0 \in D_x, 0 \notin D_z$	$\Longrightarrow$	$y = 1$
$\forall x\ \exists y\ \exists z\ \ (x \vee y = z)$	$1 \in D_x, 1 \notin D_z$	$\Longrightarrow$	*faux*
	$1 \notin D_z$	$\Longrightarrow$	$y = 0$
	$0 \in D_x, 0 \notin D_z$	$\Longrightarrow$	$y = 1$
	$1 \notin D_x, 1 \notin D_y$	$\Longrightarrow$	$z = 0$
	$0 \notin D_x$	$\Longrightarrow$	$z = 1$
	$0 \notin D_y$	$\Longrightarrow$	$z = 1$
$\exists x\ \forall y\ \forall z\ \ (x \vee y = z)$	$0 \in D_z, 1 \in D_z$	$\Longrightarrow$	*faux*
	$1 \in D_y, 1 \notin D_z$	$\Longrightarrow$	*faux*
	$1 \notin D_z$	$\Longrightarrow$	$x = 0$
	$0 \in D_y, 0 \notin D_z$	$\Longrightarrow$	$x = 1$
$\exists x\ \forall y\ \exists z\ \ (x \vee y = z)$	$1 \in D_y, 1 \notin D_z$	$\Longrightarrow$	*faux*
	$1 \notin D_z$	$\Longrightarrow$	$x = 0$
	$0 \in D_y, 0 \notin D_z$	$\Longrightarrow$	$x = 1$
	$1 \notin D_x, 1 \notin D_y$	$\Longrightarrow$	$z = 0$
	$0 \notin D_x$	$\Longrightarrow$	$z = 1$
	$0 \notin D_y$	$\Longrightarrow$	$z = 1$
$\exists x\ \exists y\ \forall z\ \ (x \vee y = z)$	$0 \in D_z, 1 \in D_z$	$\Longrightarrow$	*faux*
	$1 \notin D_z$	$\Longrightarrow$	$x = 0$
	$1 \notin D_y, 0 \notin D_z$	$\Longrightarrow$	$x = 1$
	$1 \notin D_z$	$\Longrightarrow$	$y = 0$
	$1 \notin D_x, 0 \notin D_z$	$\Longrightarrow$	$y = 1$
$\exists x\ \exists y\ \exists z\ \ (x \vee y = z)$	$1 \notin D_z$	$\Longrightarrow$	$x = 0$
	$1 \notin D_y, 0 \notin D_z$	$\Longrightarrow$	$x = 1$
	$1 \notin D_z$	$\Longrightarrow$	$y = 0$
	$1 \notin D_x, 0 \notin D_z$	$\Longrightarrow$	$y = 1$
	$1 \notin D_x, 1 \notin D_y$	$\Longrightarrow$	$z = 0$
	$0 \notin D_x$	$\Longrightarrow$	$z = 1$
	$0 \notin D_y$	$\Longrightarrow$	$z = 1$

Figure 7.4 – Règles de propagation pour $\vee$.

7.2.2.1 Exemple d'application

Pour illustrer l'application de la propagation booléenne par règles sur un cas simple, reprenons l'exemple 6.7 (chapitre précédent).

$$\begin{array}{lll} A: & \forall x \in \{0,1\} \ \exists y \in \{0,1\} \ \exists z \in \{0,1\} & (x \vee y = z) \\ B: & \exists y \in \{0,1\} \ \forall t \in \{0,1\} \ \exists u \in \{1\} & (y \vee t = u) \end{array}$$

Aucune des règles correspondant à la contrainte A ne s'applique. En revanche, il est possible d'appliquer une règle correspondant à la deuxième contrainte, à savoir :

$$0 \in D_t, 0 \notin D_u \implies y = 0$$

On peut donc reconsidérer le première contrainte ; du fait de la réduction du domaine de y, une règle s'applique désormais :

$$0 \notin D_y \implies z = 1$$

Plus aucune règle ne s'applique.

7.2.3 Propriétés des propagateurs booléens

Du point de vue opérationnel, une règle est vue de manière naturelle comme une procédure qui, si les conditions d'application sont respectées, provoque la suppression d'éléments de certains domaines de variables existentielles. Si une suppression permet de déduire qu'un domaine est vide, on génère implicitement un *échec*.

7.2.3.1 Monotonie et confluence

Les règles de calcul utilisées pour décrire nos algorithmes de propagation présentent une particularité indésirable : elles permettent d'exprimer des raisonnements *non monotones*. Par exemple, une condition de type $1 \in x$ peut être vérifiée à un instant donné puis être invalidée si 1 est éliminée du domaine de x. La restriction syntaxique suivante permet d'éviter cet écueil :

Définition 29. (règle sûre)

Une règle est dite sûre *si*
- *les tests d'inclusion (*$\mathsf{cond} \in x$*) portent exclusivement sur des variables quantifiées universellement ;*
- *si la conclusion de la règle est une instantiation, celle-ci concerne une variable quantifiée existentiellement.*

Toutes les règles utilisées pour exprimer les contraintes de négation et de disjonction vérifient cette propriété de sûreté. La propriété de sûreté est une condition suffisante pour assurer que l'opérateur ainsi décrit est *monotone*, ce qui rend possible la définition suivante :

Définition 30. (opérateur associé à une règle)

L'opérateur associé à une règle est la fonction associant à un QCSP :
- *soit ce même QCSP non modifié si les domaines fournis en entrée ne vérifient pas la garde,*
- *soit faux si la conclusion indique l'inconsistance,*

- *soit le QCSP initial dans lequel l'instantiation de domaine indiquée en conclusion a été effectuée,*
- *dans le cas où un domaine devient vide, on retourne faux.*

Il est clair que l'opérateur ainsi défini est section est *contractant* au sens défini dans le chapitre 4 : la deuxième condition de sûreté impose en effet que les domaines des variables existentielles sont toujours réduits par application des opérateurs. La *monotonie* vient de la première condition de sûreté : enfin chaque règle est *idempotente* (cf. Section 6.2.3.1) puisque la conclusion qu'elle permet d'obtenir est une instantiation de variable, qu'il serait inutile d'effectuer plusieurs fois[4]. La propriété de confluence d'application des règles est donc garantie par les théorèmes données en Section 4.3.2.

Notez que cette définition donne à la fois une méthode opérationnelle de calcul (l'algorithme d'arc-consistance consiste à appliquer les règles jusqu'à saturation suivant une itération chaotique classique) et une caractérisation déclarative du calcul (le résultat est le plus grand point fixe du jeu d'opérateurs).

7.2.3.2 Complexité

La présentation de l'algorithme sous forme de règles permet de déterminer une borne optimale de complexité. On utilise pour cela l'algorithme de propagation unitaire présenté au chapitre 4. Il est clair que les règles de calcul utilisées sont en fait des règles propositionnelles, c.-à-d. un cas particulier de clauses. Pour mettre en évidence cette nature propositionnelle, on associe à chaque variable x du problème initial *deux* variables propositionnelles :

$$\begin{array}{lll} X_0 & \text{signifiant} & 0 \in D_x \\ X_1 & \text{signifiant} & 1 \in D_x \end{array}$$

Il est alors simple de convertir une règle en clause : les tests d'inclusion et la conclusion des règles se traduisent en littéraux, et une règle se traduit donc en clause en utilisant les transformations suivantes :

$$\begin{array}{rcl} traduction(const \in x) & \equiv & X_{\text{const}} \\ traduction(const \notin x) & \equiv & \neg X_{\text{const}} \\ traduction(x = const) & \equiv & \neg X_{(1-const)} \\ traduction(c_1, \ldots, c_q \implies \textit{tête}) & \equiv & (\overline{traduction(c_1)} \vee \cdots \vee \overline{traduction(c_q)} \vee \textit{tête}) \end{array}$$

Le surlignement désigne la négation de littéral, définie par $\overline{x} = \neg x$ et $\overline{\neg x} = x$. Par exemple, la règle

$$1 \in D_x, 0 \notin D_z \implies y = 1$$

se traduit par la clause

$$\neg(1 \in D_x) \vee \neg(0 \notin D_z) \vee y = 1 \quad \text{c.-à-d.} \quad \neg X_1 \vee Z_0 \vee \neg Y_0$$

Il est clair que l'algorithme de propagation unitaire sur cet ensemble de règles propositionnelles permet d'activer les règles dès que l'ensemble de leurs gardes sont validées (on a vu dans le chapitre 4 que la propagation unitaire est un algorithme optimal pour Horn-SAT). Cet algorithme fonctionne en temps linaire par rapport à la longueur de l'ensemble de clauses considéré. Remarquons de plus que la taille du problème SAT généré est linéaire par rapport au nombre de contraintes primitives quantifiées issues du QCSP original. On conclut que *l'arc-consistance quantifiée peut être mise en œuvre en temps linéaire par rapport au nombre de contraintes* sur le domaine booléen[5].

[4] En fait, les règles de calcul sur les booléens présentent la particularité de ne pouvoir être appliquées qu'une seule fois : une fois que leur conclusion a été inférée, on peut supprimer l'opérateur.

[5] L'algorithme mis en œuvre dans notre prototype est cependant légèrement moins efficace, puisqu'il est à l'heure actuelle basé sur une propagation de type AC3, permettant de gérer de manière plus homogène la propagation booléenne et la propagation d'intervalles.

7.3 Propagation d'intervalles

Nous proposons maintenant des propagateurs pour les contraintes numériques (addition, multiplication par une constante, etc.) sur les domaines finis. Si les techniques d'énumération présentées dans le chapitre précédent permettent de réduire les domaines de ce type de contraintes, il est possible, par l'utilisation d'intervalles, d'obtenir des règles de réduction nécessitant un temps de calcul constant, indépendant de la taille des domaines. Les règles de raisonnement que nous allons introduire dans cette section permettent par exemple d'obtenir les déductions suivantes :

$$\begin{array}{lll} \exists x \in [1,100]\ \forall y \in [1,100]\ \exists z \in [51,170]\ (x+y=z) & \rightsquigarrow & x \in [50,70] \\ \forall x \in [13,15]\ \exists y \in [50,70]\ \exists z \in [10,500]\ (x+y=z) & \rightsquigarrow & z \in [63,85] \\ \forall x \in [1,100]\ \exists y \in [1,100]\ \exists z \in [1,100]\ \ (x+y=z) & \rightsquigarrow & \textit{faux} \end{array}$$

7.3.1 Calcul d'intervalles

Il est d'usage de considérer que le travail fondateur introduisant l'utilisation des intervalles pour une utilisation sûre de l'arithmétique en machine est l'ouvrage de R. MOORE [204]. Le calcul par intervalles présente un certain nombre d'avantages, il permet notamment de prendre en compte des données imprécises et de garantir la correction des calculs opérés sur la représentation des nombres réels en machine, dont la nature finie (encodage en nombres *flottantes*) est parfois source d'erreurs numériques[6]. Une référence moderne sur l'arithmétique d'intervalles et son application à l'optimisation globale sous contraintes est [139].

7.3.1.1 Propagation d'intervalles

La propagation de contraintes basée sur des intervalles a été introduite dans la fin des années 1980 [78, 60, 150]. Les développements des années 1990 ont permis son application aux contraintes discrètes [247] et continues [179, 106, 20][7]. La plupart des résolveurs de contraintes en domaines finis utilisent une forme de raisonnement pas intervalles pour certaines contraintes (voir par exemple [91]).

Un intervalle est un couple de points $[a,b]$ représentant l'ensemble $\{x \mid a \leq x \leq b\}$. On notera I_x l'intervalle de valeurs associé à la variable x. Étant donné un intervalle I, on notera I^- sa borne gauche et I^+ sa borne droite ; on abrégera ${I_x}^-$ et ${I_x}^+$ en x^- et x^+, respectivement. L'intervalle associé à une variable x est *vide* si $x^- > x^+$. Différentes opérations peuvent être approximées par un calcul d'intervalles, celles de comparaison, d'addition, de multiplication, etc. Nous rappelons tout d'abord celles d'*union* et d'*intersection* :

$$I_x \cap I_y \ = \ \left[\ \max(x^-,y^-), \min(x^+,y^+)\ \right] \qquad I_x \cup I_y \ = \ \left[\ \min(x^-,y^-), \max(x^+,y^+)\ \right]$$

[6] Il serait inexact de considérer que les erreurs de calcul dues aux arrondis opérés sur les réels dans leur représentation machine peuvent être facilement bornées, ni qu'on puisse systématiquement les corriger par une augmentation de la précision de la représentation : des exemples ont montré que certaines évaluations de fonctions pouvaient mener à des évaluations parfaitement incorrectes que seules les méthodes par intervalles avec arrondis "vers l'extérieur" sont capables de détecter.

[7] Un grand nombre de méthodes de calcul sont utilisées par les outils actuels de résolution de contraintes continues et d'optimisation, notamment des techniques de manipulation symbolique et des résolveurs spécialisés sur certains types de contraintes (linéaires, quadratiques, etc.) [32]. Concernant la propagation proprement dite, on considère en général que la *box-consistance* [19] est la technique la plus performante, notamment dans la version présentée dans [18]. Nous ne pouvons malheureusement faire qu'effleurer ce sujet, cher à l'équipe (précisément dénommée *Contraintes Continues et Applications*, ou CoCoA) au sein de laquelle cette thèse s'est déroulée.

Notre but dans ce chapitre est d'introduire les propagateurs quantifiés correspondant à chacune des opérations arithmétiques de base. Nous choisissons une présentation par type d'opération, qui permet de procéder par niveau croissant de difficulté technique.

7.3.1.2 Syntaxe des règles de calcul

La propagation de contraintes d'intervalles consiste à réduire les bornes de certaines variables (existentielles) en fonction des réductions touchant d'autres variables. L'opération de base est donc l'*intersection d'intervalles,* consistant à éliminer de l'intervalle de valeurs d'un domaine des valeurs n'appartenant pas à un intervalle calculé en fonction d'autres variables. La notation $I_x \subseteq A$, où A est un intervalle, sera utilisée pour indiquer qu'on élimine de l'intervalle associé à x les valeurs non incluses dans A. Certaines réductions d'intervalles étant soumises à des conditions sur d'autres bornes, nous adoptons la syntaxe suivante pour les règles de calcul de borne (x désigne une variable, $\diamond$ un opérateur arithmétique binaire choisi parmi $\{+, -, \times, \div\}$ et $\sim$ une relation binaire choisie parmi $\{<, \leq, =, \neq, \geq, >\}$)[8] :

```
  val  ::=  x⁻ | x⁺ | +∞ | −∞ | entier | val ◇ val
  itv  ::=  I_x | [val, val]
 test  ::=  val ∼ val
corps  ::=  ϵ | test [, test]*
règle  ::=  corps ⟹ I_x ⊆ itv
```

Dans les cas simples, nous aurons uniquement besoin de règles dont le corps utilise des tests sur les bornes des intervalles ; l'utilisation de calculs d'intervalles à ce niveau (tests de forme `val` $\in$ `itv` | `val` $\notin$ `itv`) sera utile pour exprimer les contraintes de multiplication et de division en Section 7.3.4. Notons enfin que, contrairement au cas booléen ou aux autres domaines finis, il nous est impossible d'avoir recours à des tables précompilant les résultats des calculs de consistance, celles-ci étant dépendantes du domaine de calcul utilisé. La *correction* de nos algorithmes de propagation devra donc être démontrée par un raisonnement analytique basé sur la définition de l'arc-consistance, que nous rappelons ici (la fonction $consist_x(\phi)$ dénote l'ensemble des valeurs consistantes[9] du domaine de x) :

$$consist_x\left(\ \exists x\ Q^\star(C)\ \right) \stackrel{\text{def}}{=} \left\{\ v \in D_x\ \middle|\ \sigma_{x=v}(Q^\star(C)) \text{ est vrai}\ \right\}$$

$$consist_x\left(\ \forall x\ Q^\star(C)\ \right) \stackrel{\text{def}}{=} \begin{cases} D_x & \text{si } \sigma_{x=v}(Q^\star(C)) \text{ est vrai pour tout } v \in D_x \\ \emptyset & \text{sinon} \end{cases}$$

$$consist_x\left(\ Qy\ Q^\star(C)\ \right),\ x \neq y \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{si } Qy\ Q^\star \text{ est } \textit{faux} \\ \bigcup_{v \in D_y} consist_x(\sigma_{y=v}(Q^\star(C))) & \text{sinon} \end{cases}$$

7.3.2 Intersection, égalité, comparaison, différence

Les contraintes d'ordre (de forme $\leq$, $<$), d'égalité ($=$) et d'inégalité ($\neq$) sont les plus simples à exprimer. La figure 7.5 présente les règles obtenues pour ces différents opérateurs.

[8] On s'autorisera à noter les intervalles ouverts avec les crochets vers l'intérieur, écrivant par exemple $[-\infty, 2]$.

[9] Nous adoptons ici une notation légèrement différente de celle utilisée dans le chapitre précédent : nous nous intéressons en effet aux valeurs consistantes *d'un domaine particulier,* alors que la définition originale concernait l'ensemble des affectations consistantes de forme $(x : v)$ du problème.

contrainte	garde		action
$\forall x\, \forall y\ \ (x \leq y)$	$x^+ > y^-$	$\Longrightarrow$	*faux*
$\forall x\, \exists y\ \ (x \leq y)$	$x^+ > y^+$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_y \subseteq [x^-, +\infty]$
$\exists x\, \forall y\ \ (x \leq y)$		$\Longrightarrow$	$I_x \subseteq [-\infty, y^-]$
$\exists x\, \exists y\ \ (x \leq y)$		$\Longrightarrow$	$I_x \subseteq [-\infty, y^+]$
		$\Longrightarrow$	$I_y \subseteq [x^-, +\infty]$
$\forall x\, \forall y\ \ (x < y)$	$x^+ \geq y^-$	$\Longrightarrow$	*faux*
$\forall x\, \exists y\ \ (x < y)$	$x^+ \geq y^+$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_y \subseteq [x^- + 1, +\infty]$
$\exists x\, \forall y\ \ (x < y)$		$\Longrightarrow$	$I_x \subseteq [-\infty, y^- - 1]$
$\exists x\, \exists y\ \ (x < y)$		$\Longrightarrow$	$I_x \subseteq [-\infty, y^+ - 1]$
		$\Longrightarrow$	$I_y \subseteq [x^- + 1, +\infty]$
$\forall x\, \forall y\ \ (x = y)$	$x^- \neq y^+$	$\Longrightarrow$	*faux*
	$x^+ \neq y^-$	$\Longrightarrow$	*faux*
$\forall x\, \exists y\ \ (x = y)$	$x^- < y^-$	$\Longrightarrow$	*faux*
	$x^+ > y^+$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_y \subseteq [x^-, x^+]$
$\exists x\, \forall y\ \ (x = y)$	$y^- \neq y^+$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_x \subseteq [y^-, y^+]$
$\exists x\, \exists y\ \ (x = y)$		$\Longrightarrow$	$I_x \subseteq [y^-, y^+]$
		$\Longrightarrow$	$I_y \subseteq [x^-, x^+]$
$\forall x\, \forall y\ \ (x \neq y)$	$x^+ \geq y^-, y^+ \geq x^-$	$\Longrightarrow$	*faux*
$\forall x\, \exists y\ \ (x \neq y)$	$x^- \leq y^-, y^- = y^+, y^+ \leq x^+$	$\Longrightarrow$	*faux*
$\exists x\, \forall y\ \ (x \neq y)$	$x^+ \leq y^+$	$\Longrightarrow$	$I_x \subseteq [-\infty, y^- - 1]$
	$x^- \geq y^-$	$\Longrightarrow$	$I_x \subseteq [y^+ + 1, +\infty]$
$\exists x\, \exists y\ \ (x \neq y)$	$x^- = x^+, x^- = y^-$	$\Longrightarrow$	$I_y \subseteq [y^- + 1, +\infty]$
	$x^- = x^+, x^- = y^+$	$\Longrightarrow$	$I_y \subseteq [-\infty, y^+ - 1]$
	$y^- = y^+, y^- = x^-$	$\Longrightarrow$	$I_x \subseteq [x^- + 1, +\infty]$
	$y^- = y^+, y^- = x^+$	$\Longrightarrow$	$I_x \subseteq [-\infty, x^+ - 1]$

Figure 7.5 – Règles de propagation pour $\leq$, $<$, $=$ et $\neq$.

Nous présentons les preuves des différentes règles de propagation utilisées pour l'opérateur $\leq$, en considérant chacune des 4 quantifications possibles :

- $\forall x \, \forall y \;\; x \leq y$: Cette contrainte est vraie si et seulement si :

$$\bigwedge_{v_x \in I_x} \bigwedge_{v_y \in I_y} v_x \leq v_y \qquad \text{c.-à-d. ssi } x^+ \leq y^-$$

- $\forall x \, \exists y \;\; x \leq y$: Cette contrainte est vraie si et seulement si :

$$\bigwedge_{v_x \in I_x} \bigvee_{v_y \in I_y} v_x \leq v_y \qquad \text{c.-à-d. ssi } \bigwedge_{v_x \in I_x} v_x \leq y^+ \qquad \text{c.-à-d. ssi } x^+ \leq y^+$$

  L'ensemble des valeurs consistantes pour y est donc :

$$\bigcup_{v_x \in I_x} \{v_y \in I_y \mid v_x \leq v_y\} \;\; = \;\; \bigcup_{v_x \in I_x} [v_x, y^+] \;\; = \;\; [x^-, y^+]$$

- $\exists x \, \forall y \;\; x \leq y$: Les valeurs consistantes de I_x vérifient :

$$\bigwedge_{v_y \in I_y} x \leq v_y \qquad \text{c.-à-d. } x \leq y^-$$

- $\exists x \, \exists y \;\; x \leq y$: Les valeurs consistantes de x sont celles inférieures à une valeur de I_y (et donc à y^+) ; les valeurs consistantes de y sont celles supérieures à une valeur de I_x (et donc à x^-).

7.3.3 Addition, soustraction

Le raisonnement par intervalles sur les courbes des fonctions d'addition et de soustraction tire partie de leur propriété de *monotonie*, exprimée comme suit :

Définition 31. (monotonie d'une fonction n-aire[10])

- *Une fonction f est* croissante *sur son i-ème argument si, en augmentant la valeur de cet argument, la valeur du résultat augmente également c.-à-d. si (pour tout $(x_1, \ldots x_n)$) :*

$$\forall \epsilon \geq 0 \quad f(x_1, \ldots, x_{i-1}, x_i + \epsilon, x_{i+1}, \ldots, x_m) \geq f(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_m)$$

- *Une fonction f est* décroissante *sur son i-ème argument si, en augmentant la valeur de cet argument, la valeur du résultat diminue :*

$$\forall \epsilon \geq 0 \quad f(x_1, \ldots, x_{i-1}, x_i + \epsilon, x_{i+1}, \ldots, x_m) \leq f(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_m)$$

- *Une fonction f est* indépendante *de son i-ème argument si, elle est à la fois croissante et décroissante, c.-à-d. si, en modifiant la valeur de cet argument, la valeur du résultat demeure inchangée :*

$$\forall \epsilon \geq 0 \quad f(x_1, \ldots, x_{i-1}, x_i + \epsilon, x_{i+1}, \ldots, x_m) = f(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_m)$$

  Une fonction est monotone *sur son i-ème argument si elle est soit croissante, soit décroissante (soit indépendante) sur cet argument. Une fonction est dite* monotone *si elle est monotone sur tous ses arguments.*

[10] Cette propriété est appelée *multi-monotonie* dans [66].

La monotonie est une propriété importante pour le calcul par intervalles car elle permet de déterminer la borne inférieure (resp. supérieure) d'une telle fonction. Par exemple, la soustraction étant croissante sur x et décroissante sur y, il est clair que la borne inférieure de $I_x - I_y$ est obtenue lorsque x prend la valeur x^- et que y prend la valeur y^+. On obtient les règles de calcul suivantes :

	Règle de calcul	Exemple
$x \pm y$	$I_x + I_y = [\, x^- + y^-, x^+ + y^+ \,]$	$[-1,10] + [-5,-2] = [-6,+8]$
	$I_x - I_y = [\, x^- - y^+, x^+ - y^- \,]$	$[-1,10] - [-5,-2] = [1,15]$

Un ensemble de règles de propagation pour les différentes variantes quantifiées de la contrainte d'addition est présenté en Figure 7.7. Nous détaillons une des preuves les plus intéressantes, celle de la méthode de calcul des valeurs consistantes pour x de la contrainte $\exists x\ \forall y\ \exists z\ (x + y = z)$, illustrée par la figure suivante : l'ensemble des valeurs consistantes pour x sont telles que, pour chaque valeur v du domaine de y, on a $\exists y\ (x + v = y)$. Chacune des valeurs consistantes de x doit donc vérifier la relation :

$$\bigwedge_{v_y \in D_y} \exists z\ (x + v_y = z)$$

c.-à-d. :

$$\bigwedge_{v_y \in D_y} \left(\bigvee_{v_z \in D_z} (x = v_z - v_y) \right)$$

Cette propriété se réécrit :

$$\bigwedge_{v_y \in D_y} \left(x \in\ (I_z - v_y) \right)$$

L'intersection ainsi représentée est illustrée par la figure suivante :

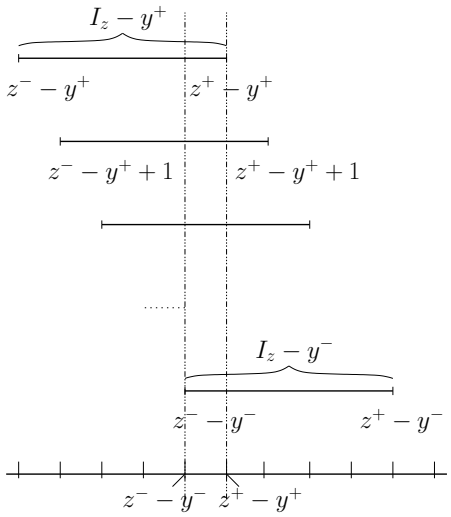


Figure 7.6 – Valeurs de x telles que, pour chacune des valeurs de y prises dans I_y, on vérifie $x + y \in I_z$.

x doit appartenir *à la fois* à chacun des ensembles $(I_z - y^+)$, $(I_z - y^+ + 1)$, $\ldots$, $(I_z - y^-)$, et donc à :

$$\bigcap_{v_z \in D_z} (I_z - v_y)$$

La borne inférieure de chacun des intervalles $I_z - v_y$ est atteinte quand z prend sa valeur minimale z^-. La valeur de x doit donc être à la fois supérieure à $z^- - y^+$, à $z^- - y^+ + 1$, à $z^- - y^-$, ce qui revient à être supérieur à $z^- - y^-$. Un raisonnement similaire permet de déterminer la borne droite. On obtient finalement la règle de propagation suivante :

$$x \subseteq [z^- - y^-, z^+ - y^+]$$

L'intervalle peut être vide si la borne droite ainsi calculée est inférieure à la borne gauche, ce qui correspond bien au cas où les intervalles $I_y - y^+$ et $I_z - y^-$ sont disjoints.

contrainte	garde		action
$\forall x\, \forall y\, \forall z\ \ (x + y = z)$	$x^- \neq x^+$	$\Longrightarrow$	*faux*
	$y^- \neq y^+$	$\Longrightarrow$	*faux*
	$z^- \neq z^+$	$\Longrightarrow$	*faux*
	$x^- + y^- \neq z^-$	$\Longrightarrow$	*faux*
$\forall x\, \forall y\, \exists z\ \ (x + y = z)$	$z^- > x^- + y^-$	$\Longrightarrow$	*faux*
	$z^+ < x^+ + y^+$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_z \subseteq [x^- + y^-, x^+ + y^+]$
$\forall x\, \exists y\, \forall z\ \ (x + y = z)$	$z^- \neq z^+$	$\Longrightarrow$	*faux*
	$z^- > x^- + y^-$	$\Longrightarrow$	*faux*
	$z^+ < x^+ + y^+$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_y \subseteq [z^- - x^+, z^+ - x^-]$
$\forall x\, \exists y\, \exists z\ \ (x + y = z)$	$x^- < z^- - y^+$	$\Longrightarrow$	*faux*
	$x^+ > z^+ - y^-$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_y \subseteq [z^- - x^+, z^+ - x^-]$
		$\Longrightarrow$	$I_z \subseteq [x^- + y^-, x^+ + y^+]$
$\exists x\, \forall y\, \forall z\ \ (x + y = z)$	$y^- \neq y^+$	$\Longrightarrow$	*faux*
	$z^- \neq z^+$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_x \subseteq [z^- - y^+, z^+ - y^-]$
$\exists x\, \forall y\, \exists z\ \ (x + y = z)$		$\Longrightarrow$	$I_x \subseteq [z^- - y^-, z^+ - y^+]$
		$\Longrightarrow$	$I_z \subseteq [x^- + y^-, x^+ + y^+]$
$\exists x\, \exists y\, \forall z\ \ (x + y = z)$	$z^- \neq z^+$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_x \subseteq [z^- - y^+, z^+ - y^-]$
		$\Longrightarrow$	$I_y \subseteq [z^- - x^+, z^+ - x^-]$
$\exists x\, \exists y\, \exists z\ \ (x + y = z)$		$\Longrightarrow$	$I_x \subseteq [z^- - y^+, z^+ - y^-]$
		$\Longrightarrow$	$I_y \subseteq [z^- - x^+, z^+ - x^-]$
		$\Longrightarrow$	$I_z \subseteq [x^- + y^-, x^+ + y^+]$

Figure 7.7 – Règles de propagation pour $+$.

On remarque que toutes les contraintes de forme $Qx\ Q'y\ \forall z\ (x + y = z)$ sont systématiquement fausses dès lors que le domaine de z n'est pas un singleton, ceci est dû au fait que l'addition est une relation *fonctionnelle,* et que l'image z est unique pour tous x et y fixés.

Des règles de propagation similaires pour la contrainte de soustraction peuvent être obtenues par un raisonnement similaire. Cependant, une alternative est possible : l'emploi de l'opération de soustraction unaire, qui permet de décomposer $x - y = z$ en $x + (-y) = z$. Plus précisément, partant d'une addition quantifiée de forme $Q_x x\ Q_y y\ Q_z z\ (x + y = z)$, l'introduction de la variable supplémentaire nécessaire donne le QCSP suivant :

$$Q_x x\ Q_y y\ Q_z x\ \exists u\ (-y = u, x + u = z)$$

La variable introduite est nécessairement existentielle et elle ne précède jamais la variable qu'elle renomme dans l'ordre des quantificateurs. Les deux cas reportés dans la figure 7.8 suffisent donc à exprimer la soustraction binaire en toute généralité.

contrainte	garde		action
$\forall x\ \exists y\ \ (-x = y)$	$y^- > -x^+$	$\Longrightarrow$	*faux*
	$y^+ < -x^-$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_y \subseteq [-x^+, -x^-]$
$\exists x\ \exists y\ \ (-x = y)$		$\Longrightarrow$	$I_x \subseteq [-y^+, -y^-]$
		$\Longrightarrow$	$I_y \subseteq [-x^+, -x^-]$

Figure 7.8 – Règles de propagation pour la soustraction.

7.3.4 Multiplication, division

Les fonctions de multiplication et de division sont plus délicates à manipuler pour plusieurs raisons, l'une d'elles étant le fait qu'elles sont *non-monotones*. Plus précisément, elles sont formées d'un petit nombre de parties monotones (x/y est monotone lorsque x et y prennent toutes deux des valeurs entièrement positives ou entièrement négatives).

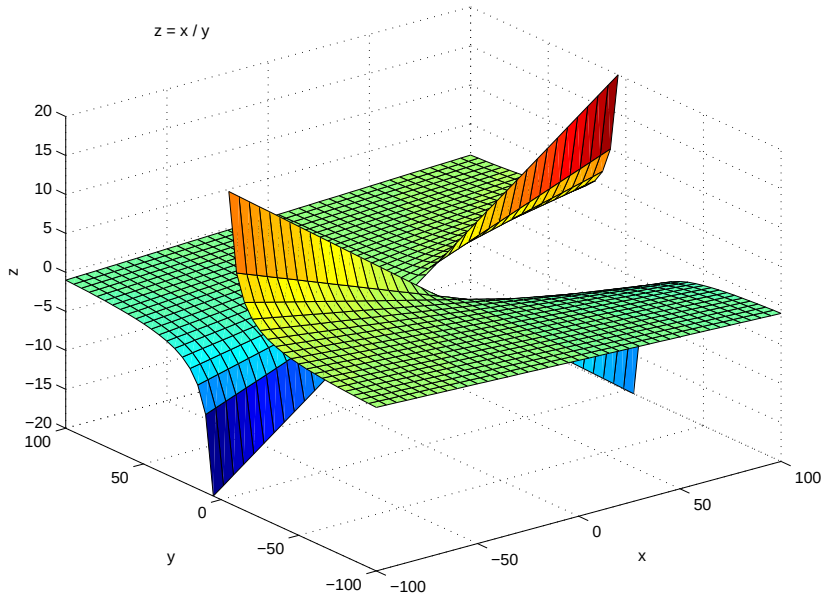


Figure 7.9 – La courbe de la contrainte de division (courbe obtenue sous `Matlab`).

7.3.4.1 Évaluation intervalle de la multplication et de la division

La figure 7.9 montre que la courbe obtenue pour la division se décompose en quatre parties monotones ; par exemple, si x est négatif et y est strictement positif, la fonction x/y est croissante sur x comme sur y. Les règles de calcul suivantes représentent l'évaluation par intervalles de la multiplication et de la division sur leurs 4 parties monotones :

x/y

Règle de calcul	Exemple
$I_x/I_y = [\ x^+/y^-, x^-/y^+\ ]$ si $x^+ \leq 0$ et $y^+ < 0$	$[-10,-1]/[-5,-2] = [+\frac{1}{5}, +5]$
$I_x/I_y = [\ x^-/y^-, x^+/y^+\ ]$ si $x^+ \leq 0$ et $y^- > 0$	$[-10,-1]/[+2,+5] = [-5, -\frac{1}{5}]$
$I_x/I_y = [\ x^+/y^+, x^-/y^-\ ]$ si $x^- \geq 0$ et $y^+ < 0$	$[+1,+10]/[-5,-2] = [-5, -\frac{1}{5}]$
$I_x/I_y = [\ x^-/y^+, x^+/y^-\ ]$ si $x^- \geq 0$ et $y^- > 0$	$[+1,+10]/[+2,+5] = [+\frac{1}{5}, +5]$

$x.y$

Règle de calcul	Exemple
$I_x.I_y = [\ x^+.y^+, x^-.y^-\ ]$ si $x^+ \leq 0$ et $y^+ \leq 0$	$[-10,-1].[-5,-2] = [+2,+50]$
$I_x.I_y = [\ x^-.y^+, x^+.y^-\ ]$ si $x^+ \leq 0$ et $y^- \geq 0$	$[-10,-1].[+2,+5] = [-50,-2]$
$I_x.I_y = [\ x^+.y^-, x^-.y^+\ ]$ si $x^- \geq 0$ et $y^+ \leq 0$	$[+1,+10].[-5,-2] = [-50,-2]$
$I_x.I_y = [\ x^-.y^-, x^+.y^+\ ]$ si $x^- \geq 0$ et $y^- \geq 0$	$[+1,+10].[+2,+5] = [+2,+50]$

Sur des intervalles quelconques, l'évaluation par intervalles de la multiplication et de la division est basée sur l'évaluation en un certain nombre de *points extrémaux*, et est calculée comme suit[11] :

$$I_x.I_y = [\min(Ext), \min(Ext)], \qquad \text{où } Ext = \{x^-.y^-, x^-.y^+, x^+.y^-, x^+.y^+\}$$

$$I_x/I_y = [\lceil \min(Ext) \rceil, \lfloor \max(Ext) \rfloor], \quad \text{où } Ext = \{v_x/v_y \mid v_x \in Ext_x, v_y \in Ext_y\}$$

$$\text{et} \begin{cases} Ext_x = \{x^-, x^+\} \\ Ext_y = (\ \{y^-, y^+\} \cup (\{-1,1\} \cap I_y)\ ) \end{cases}$$

La division par intervalles ainsi obtenue donne bien une approximation de l'ensemble des v_z tels que $v_y.v_z = v_x$, ce qui correspond à la version "relationnelle" de la division par intervalles.

[11] Rappelons que nous manipulons des valeurs *entières*, et que les règles de calcul sont donc légèrement différentes des règles d'arithmétique d'intervalles classiques, sur les réels ou les nombres flottants. Dans notre cas, les seules valeurs proches de 0 ayant un sens sont 1 et -1 alors que, sur les réels, la division par un intervalle contenant 0 fait tendre les valeurs possibles vers les infinis (division par des valeurs infiniment petites). Par exemple, $[-3,5]/[-7,8]$ s'évalue en $[-\infty, +\infty]$ sur les réels, mais en $[\lceil \frac{5}{-1} \rceil, \lfloor \frac{5}{1} \rfloor] = [-5,5]$ sur les entiers. Il suffit donc de prendre en compte les valeurs -1 et 1 de y — si celles-ci sont incluses dans I_y — pour calculer les bonnes bornes. Cette technique n'est, à ma connaissance, pas documentée dans les systèmes de programmation par contraintes classiques (GNU-Prolog [62] utilise des domaines positifs, CLP(BNR) [20] utilise un calcul d'intervalles réels dont les bornes sont tronquées pour obtenir des valeurs entières, on obtiendrait donc $[-\infty, +\infty]$).

7.3.4.2 Multiplication par une constante

Un cas simple et important d'utilisation de la multiplication est celui de la multiplication par une constante. En effet, les fonctions *linéaires* sont exprimées sous la forme $\sum_i a_i.x_i$, et la contrainte de multiplication par une constante est donc suffisante pour gérer les problèmes de *programmation linéaire en nombres entiers quantifiée*. De même que pour les contraintes numériques classiques, la plupart des problèmes se modélisent naturellement en termes de contraintes linéaires. Les règles de propagation dépendent du signe de la constante, elles sont présentées en Figure 7.10.

contrainte	garde		action
$\forall x\ \forall y\ \ (a.x = y)$	(si $a > 0$) $\begin{cases} a.x^+ \neq y^- \\ a.x^- \neq y^+ \end{cases}$	$\Longrightarrow$ $\Longrightarrow$	*faux* *faux*
	(si $a < 0$) $\begin{cases} a.x^- \neq y^- \\ a.x^+ \neq y^+ \end{cases}$	$\Longrightarrow$ $\Longrightarrow$	*faux* *faux*
$\forall x\ \exists y\ \ (a.x = y)$	(si $a > 0$) $\begin{cases} a.x^- < y^- \\ a.x^+ > y^+ \\ \end{cases}$	$\Longrightarrow$ $\Longrightarrow$ $\Longrightarrow$	*faux* *faux* $I_y \subseteq [a.x^-, a.x^+]$
	(si $a < 0$) $\begin{cases} a.x^+ < y^- \\ a.x^- > y^+ \\ \end{cases}$	$\Longrightarrow$ $\Longrightarrow$ $\Longrightarrow$	*faux* *faux* $I_y \subseteq [a.x^+, a.x^-]$
$\exists x\ \forall y\ \ (a.x = y)$	$y^- \neq y^+$	$\Longrightarrow$ $\Longrightarrow$	*faux* $I_x \subseteq [\lceil y^-/a \rceil, \lfloor y^-/a \rfloor]$
$\exists x\ \exists y\ \ (a.x = y)$	(si $a > 0$)	$\Longrightarrow$ $\Longrightarrow$	$\begin{cases} I_x \subseteq [\lceil y^-/a \rceil, \lfloor y^+/a \rfloor] \\ I_y \subseteq [a.x^-, a.x^+] \end{cases}$
	(si $a < 0$)	$\Longrightarrow$ $\Longrightarrow$	$\begin{cases} I_x \subseteq [\lceil y^+/a \rceil, \lfloor y^-/a \rfloor] \\ I_y \subseteq [a.x^+, a.x^-] \end{cases}$

Figure 7.10 – Règles de propagation pour la multiplication par une constante non nulle.

7.3.4.3 Multiplication : cas général

Définir un ensemble de règles de propagation pour la multiplication est beaucoup moins naturel dans le cas général, et la solution que nous proposons n'est que partiellement satisfaisante ; nous la proposons pour montrer que leur définition est *a priori* possible (voir Figure 7.11), mais ces règles n'ont pas été expérimentées. Nous avons utilisé des calculs d'intervalles dans les gardes et les conclusions des règles pour alléger l'écriture. Des propagateurs pour la division peuvent être formulés de manière similaire à ceux ainsi proposés pour la multiplication, mais nous ne les présenterons pas ici.

Les propagateurs ainsi formulés appellent différentes remarques. Premièrement, il nous faut prendre en compte le cas où l'une des variables x ou y prend ses valeurs dans $[0, 0]$. Par exemple, la contrainte $\forall x\ \forall y\ \exists z\ (x.y = z)$ est vraie si $I_y = [0, 0]$, auquel cas I_z est également restreint à $[0, 0]$. Ce fait est dû au changement de monotonie en 0. Deuxièmement, si toutes les règles que nous avons formulées sont correctes, seules certaines sont complètes, pour des raisons que nous détaillerons dans la section suivante. Les propagateurs proposés pour différentes contraintes sont le fruit d'un compromis visant à

obtenir des déductions pertinentes sans multiplier le nombre de règles considérées. Prenons par exemple la contrainte $\exists x\ \forall y\ \exists z\ (x.y = z)$. Pour chacune des valeurs $v_y \in I_y$, il serait correct d'ajouter la règle :

$$I_x \subseteq I_z / v_y$$

On obtient cependant un nombre de règles à appliquer linéaire en la taille du domaine de y, or l'un de nos buts dans cette section était précisément que le coût d'application d'une règle soit indépendant de la taille des domaines considérés. On a donc restreint les valeurs considérées aux bornes de y, ainsi qu'aux valeurs -1 et 1, si toutefois elles appartiennent au domaine.

Les propagateurs basés sur un raisonnement par intervalles pour les contraintes de multiplication et de division fournissent une approximation relativement faible de l'arc-consistance (en revanche, la confluence de la propagation de règles est préservée). L'incomplétude du raisonnement par intervalles est illustré par l'exemple suivant. Considérons la contrainte[12] existentielle $\exists x\ \forall y \in [4,5]\ \exists z \in [7,9]\ (x = y.z)$. x doit appartenir aux ensembles suivants :

$$\begin{array}{rcl} 4.[7,9] & = & \{28, 32, 36\} \\ 5.[7,9] & = & \{35, 40, 45\} \end{array}$$

Ces deux ensembles sont disjoints, et le domaine de x est donc vide. Or si on procède par intervalles, l'intersection de $[28, 36]$ et de $[35, 45]$ donne $[35, 36]$, on n'a donc pas détecté la vacuité. Cet exemple prouve que, contrairement au cas existentiel, la propagation d'intervalles quantifiés ne calcule pas les bornes des valeurs consistantes de la contrainte primitive (il faudrait pour cela effectuer une énumération). En contrepartie, il est clair que le fait de calculer seulement sur les deux bornes à la place d'un calcul par élément de I_y est moins coûteux.

7.3.5 Propriétés des propagateurs par intervalles

Nous concluons notre présentation de la propagation d'intervalles, en discutant les questions de confluence et en déterminant la complexité de l'algorithmes proposé.

7.3.5.1 Confluence

Il est clair que toutes les règles de propagation d'intervalles proposées dans cette thèse sont *contractantes*, les conclusions consistant soit à déduire *faux*, soit à réduire les bornes d'un intervalle. En revanche, la *monotonie* des opérateurs par rapport à l'ordre d'inclusion des variables existentielles est moins aisée à démontrer, et il nous faut raisonner au cas par cas.

Une première difficulté vient du fait que la condition d'application d'une règle R peut ne pas être monotone, c.-à-d. que la condition peut être vérifiée pour un certain contexte C, et ne pas l'être pour $C' \subset C$. Or supposons par exemple que la règle aie pour conclusion *faux*, il est clair que la monotonie n'est dans ce cas pas respectée ($R(C) = \emptyset$, or $R(C') = C' \supset \emptyset$). Toutes les contraintes proposées dans ce chapitre ayant pour conclusion *faux* entrent dans le cas suivant :

- Une règle est monotone si elle a pour conclusion *faux* et que sa condition est monotone.

En guise d'illustration, commentons brièvement un exemple pouvant soulever des doutes. Considérons :

$$\forall x\ \exists y\ (x < y) \qquad\qquad x^+ \geq y^+ \Longrightarrow \textit{faux}$$

[12] L'exemple considéré ne fait pas partie des contraintes gérées dans la table 7.11, mais il présente l'intérêt d'être plus lisible.

Cette règle est monotone par rapport au domaine de y, car si la borne inférieure de l'intervalle est inférieure à une certaine valeur, elle le sera également pour un intervalle I_y réduit. En revanche, la règle est anti-monotone par rapport au domaine de x : il est possible qu'en réduisant la borne supérieure de x on invalide l'inégalité alors que celle-ci était auparavant respectée. Ceci n'est cependant pas surprenant puisque la variable x est quantifiée universellement, or on a vu dans le chapitre 6 que ce type de variables peuvent être la cause d'un comportement non monotone. Fort heureusement, comme nous l'avons mentioné, nous avons uniquement besoin de la monotonie par rapport à l'ordre d'inclusion sur les variables existentielles — les variables universelles demeurent inchangées.

Le deuxième élément à prendre en compte dans notre étude est la monotonie du calcul de réduction de domaine effectué en conclusion de la règle. Il est possible de formuler des conclusions non monotones, par exemple la règle :

$$\implies \quad I_x \subseteq [0, 100 - x^+]$$

Cette règle réduit l'intervalle $[0, 100]$ à $[0, 0]$, alors qu'elle réduit $[0, 99]$ à $[0, 1]$, et même $[0, 50]$ à $[0, 50]$. Les règles que nous avons proposées évitent cet écueil car elles appartiennent à la catégorie suivante :

- Une règle est monotone si sa condition est vide et que sa conclusion est monotone.

Là encore, il nous faudrait raisonner au cas pas cas, ce qui serait long et répétitif. Nous nous contentons donc de commenter brièvement un exemple. Une règle pouvant sembler suspecte est la suivante :

$$\exists x\ \forall y\ \exists z\ (x + y = z) \qquad \implies I_x \subseteq [z^- - y^-, z^+ - y^+]$$

z peut être réduit de deux manières : en augmentant sa borne inférieure (auquel cas celle de x est également augmentée) et en réduisant sa borne supérieure (auquel cas celle de x est également diminuée). Si la règle est par conséquent monotone par rapport au domaine de z, elle est anti monotone par rapport à y, puisque une augmentation de sa borne inférieure fait par exemple diminuer celle de x. Là encore, y étant universelle, nous vérifions cependant la monotonie par rapport à l'ordre d'inclusion sur les variables existentielles.

Toutes les règles utilisées pour les contraintes d'égalité, d'addition et de multiplication par une constante appartiennent à l'une ou l'autre des catégories décrites précédemment. Il reste deux cas résistant à cette classification, et ils apparaissent (sans surprise) pour la contrainte de différence. La première règle est la suivante :

$$\exists x\ \forall y\ (x \neq y) \quad \left\{ \begin{array}{lll} x^+ \leq y^+ & \implies & I_x \subseteq [-\infty, y^- - 1] \\ x^- \geq y^- & \implies & I_x \subseteq [y^+ + 1, +\infty] \end{array} \right.$$

Les conditions sont monotones par rapport au domaine de x (dès lors que $x^+ \leq y^+$, cette inéquation restera valide pour une borne x^+ réduite), et la conclusion, qui dépend seulement de y, est donc monotone par rapport à x. Enfin, la contrainte $\exists x\ \exists y\ (x \neq y)$ présente des règles dont ni les conditions, ni les conclusions ne semblent *a priori* monotones. Chacune des règles laisse les domaines inchangés tant que l'une des variables (disons x) n'est pas instanciée. Le rôle de ces règles consiste, si une valeur est instanciée (ce qui est vrai pour tout sous-domaine non vide), à éliminer cette valeur du domaine de l'autre variable ; or l'opération consistant à éliminer une valeur donnée d'un ensemble est monotone.

7.3.5.2 Complexité

La complexité des résolveurs numériques sur les domaines finis peut bénéficier du type d'analyse proposé par O. LHOMME dans son travail de thèse [179] et repris ultérieurement dans [34]. Cette complexité s'exprime en fonction de la taille d du plus grand intervalle, du nombre m de contraintes.

Remarquons tout d'abord que chaque règle de propagation dépend d'un nombre borné (3) de variables. Lors de la propagation, chacune des règles n'est ajouté à la file des règles à réactiver que lorsque l'intervalle de l'une des variables dont elle dépend est modifié. Ceci peut survenir au plus $\mathcal{O}(d)$ fois, il y a donc au plus $\mathcal{O}(md)$ insertions de contraintes dans la file. Chaque contrainte étant extraite de la file et appliquée en temps constant, $\mathcal{O}(md)$ est également la complexité globale de l'algorithme.

7.4 Remarques

Nous avons proposé deux variations sur le thème de l'arc-consistance quantifiée. Dans le cas des domaines booléens, on retrouve un type de règles de propagation assez proche de l'esprit des CHR [116] ; dans le cas des contraintes numériques (en particulier linéaires), les règles de calcul sont l'équivalent de celles utilisées en domaines finis [62] ou des techniques de propagation d'intervalles réels, en particulier la *Hull-consistance* de [20] (également appelée 2B-consistance dans [179]). La généralisation de la propagation d'intervalles que nous avons proposé présente également des similarités avec les travaux sur les intervalles modaux [93].

L'avantage principal de l'arc-consistance quantifiée est de prendre en compte le motif exact de quantification et de tirer ainsi pleinement profit des quantificateurs universels. Ceux-ci sont en fait extrêmement contraignants puisqu'ils ont en quelque sorte pour rôle de "dupliquer" certaines contraintes. Par exemple, la contrainte universellement quantifiée :

$$\forall y \in [0,10]\ \exists z \in [0,10]\ =\ (x = y + z)$$

peut (et doit!) être vue comme un ensemble extrêmement contraignant :

$$\exists z\ (x = 0 + z) \wedge \cdots \wedge \exists z\ (x = 10 + z)$$

Enfin, ajoutons que, de même que pour les contraintes classiques, il est possible de définir des propagateurs *réifiés* pour les contraintes relationnelles ($\leq, <, =, >, \geq$), dans lesquels le résultat booléen du prédicat est rendu explicite par une variable supplémentaire (cf. chapitre 3). La réfication permet de gérer les problèmes mixtes booléen/numérique. L'utilisation de disjonctions, par exemple, est ainsi facilité. La propagation pour les contraintes réifiées $\leq$ et $=$ sont décrites en Figure 7.12.

contrainte	garde		action
$\forall x\,\forall y\,\forall z\ \ (x.y=z)$	$I_y \neq [0,0], x^- \neq x^+$	$\Longrightarrow$	*faux*
	$I_x \neq [0,0], y^- \neq y^+$	$\Longrightarrow$	*faux*
	$I_x \neq [0,0], I_y \neq [0,0], z^- \neq z^+$	$\Longrightarrow$	*faux*
	$x^-.y^- \neq z^-$	$\Longrightarrow$	*faux*
	$x^+.y^+ \neq z^+$	$\Longrightarrow$	*faux*
$\forall x\,\forall y\,\exists z\ \ (x.y=z)$	$z^- > (I_x.I_y)^+$	$\Longrightarrow$	*faux*
	$z^+ < (I_x.I_y)^-$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_z \subseteq I_x.I_y$
$\forall x\,\exists y\,\forall z\ \ (x.y=z)$	$x^- \neq x^+, I_y \neq [0,0]$	$\Longrightarrow$	*faux*
	$z^- \neq z^+, I_y \neq [0,0]$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_y \subseteq I_z/I_x$
$\forall x\,\exists y\,\exists z\ \ (x.y=z)$	$x^- \notin I_z/I_y$	$\Longrightarrow$	*faux*
	$x^+ \notin I_z/I_y$	$\Longrightarrow$	*faux*
	$1 \in I_x,\quad 1 \notin I_z/I_y$	$\Longrightarrow$	*faux*
	$-1 \in I_x, -1 \notin I_z/I_y$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_y \subseteq I_z/I_x$
		$\Longrightarrow$	$I_z \subseteq I_x.I_y$
$\exists x\,\forall y\,\forall z\ \ (x.y=z)$	$I_x \neq [0,0], y^- \neq y^+$	$\Longrightarrow$	*faux*
	$I_x \neq [0,0], z^- \neq z^+$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_x \subseteq I_z/I_y$
$\exists x\,\forall y\,\exists z\ \ (x.y=z)$	$1 \in I_y$	$\Longrightarrow$	$I_x \subseteq +I_z$
	$-1 \in I_y$	$\Longrightarrow$	$I_x \subseteq -I_z$
		$\Longrightarrow$	$I_x \subseteq I_z/y^-$
		$\Longrightarrow$	$I_x \subseteq I_z/y^+$
		$\Longrightarrow$	$I_z \subseteq I_x.I_y$
$\exists x\,\exists y\,\forall z\ \ (x.y=z)$	$I_x \neq [0,0], I_y \neq [0,0], z^- \neq z^+$	$\Longrightarrow$	*faux*
		$\Longrightarrow$	$I_x \subseteq I_z/I_y$
		$\Longrightarrow$	$I_y \subseteq I_z/I_x$
$\exists x\,\exists y\,\exists z\ \ (x.y=z)$		$\Longrightarrow$	$I_x \subseteq I_z/I_y$
		$\Longrightarrow$	$I_y \subseteq I_z/I_x$
		$\Longrightarrow$	$I_z \subseteq I_x.I_y$

Figure 7.11 – Règles de propagation pour la multiplication.

contrainte	garde		action
$\forall x\ \forall y\ \exists b\ \ (x \leq y\ \leftrightarrow b)$	$x^+ \leq y^-$	$\Longrightarrow$	$b = 1$
	$x^- > y^+$	$\Longrightarrow$	$b = 0$
$\forall x\ \exists y\ \exists b\ \ (x \leq y\ \leftrightarrow b)$	$b = 1, x^+ > y^+$	$\Longrightarrow$	*faux*
	$b = 0, x^- \leq y^-$	$\Longrightarrow$	*faux*
	$x^+ \leq y^-$	$\Longrightarrow$	$b = 1$
	$x^- > y^+$	$\Longrightarrow$	$b = 0$
	$b = 1$	$\Longrightarrow$	$I_y \subseteq [x^-, +\infty]$
	$b = 0$	$\Longrightarrow$	$I_y \subseteq [-\infty, x^+ - 1]$
$\exists x\ \forall y\ \exists b\ \ (x \leq y\ \leftrightarrow b)$	$b = 0$	$\Longrightarrow$	$I_x \subseteq [y^+ + 1, +\infty]$
	$b = 1$	$\Longrightarrow$	$I_x \subseteq [-\infty, y^-]$
	$x^+ \leq y^-$	$\Longrightarrow$	$b = 1$
	$x^- > y^+$	$\Longrightarrow$	$b = 0$
$\exists x\ \exists y\ \exists b\ \ (x \leq y\ \leftrightarrow b)$	$b = 1$	$\Longrightarrow$	$I_x \subseteq [-\infty, y^+]$
	$b = 1$	$\Longrightarrow$	$I_y \subseteq [x^-, +\infty]$
	$b = 0$	$\Longrightarrow$	$I_x \subseteq [y^- + 1, +\infty]$
	$b = 0$	$\Longrightarrow$	$I_y \subseteq [-\infty, x^+ - 1]$
	$x^+ \leq y^-$	$\Longrightarrow$	$b = 1$
	$x^- > y^+$	$\Longrightarrow$	$b = 0$
$\forall x\ \forall y\ \exists b\ \ (x = y\ \leftrightarrow b)$	$b = 1, x^- < y^+$	$\Longrightarrow$	*faux*
	$b = 1, y^- < x^+$	$\Longrightarrow$	*faux*
	$x^+ \leq y^-, y^+ \leq x^-$	$\Longrightarrow$	$b = 1$
	$x^- > y^+$	$\Longrightarrow$	$b = 0$
	$x^+ < y^-$	$\Longrightarrow$	$b = 0$
$\forall x\ \exists y\ \exists b\ \ (x = y\ \leftrightarrow b)$	$b = 1, x^- < y^-$	$\Longrightarrow$	*faux*
	$b = 1, x^+ > y^+$	$\Longrightarrow$	*faux*
	$b = 0, x^- \leq y^-, y^- = y^+, y^+ \leq x^+$	$\Longrightarrow$	*faux*
	$b = 1$	$\Longrightarrow$	$I_y \subseteq [x^-, x^+]$
	$x^+ \leq y^-, y^+ \leq x^-$	$\Longrightarrow$	$b = 1$
	$x^- > y^+$	$\Longrightarrow$	$b = 0$
	$x^+ < y^-$	$\Longrightarrow$	$b = 0$
$\exists x\ \forall y\ \exists b\ \ (x = y\ \leftrightarrow b)$	$b = 1, y^- \neq y^+$	$\Longrightarrow$	*faux*
	$b = 0, x^+ \leq y^+$	$\Longrightarrow$	$I_x \subseteq [-\infty, y^- - 1]$
	$b = 0, x^- \geq y^-$	$\Longrightarrow$	$I_x \subseteq [y^+ + 1, +\infty]$
	$b = 1$	$\Longrightarrow$	$I_x \subseteq [y^-, y^+]$
	$x^+ \leq y^-, y^+ \leq x^-$	$\Longrightarrow$	$b = 1$
	$x^- > y^+$	$\Longrightarrow$	$b = 0$
	$x^+ < y^-$	$\Longrightarrow$	$b = 0$
$\exists x\ \exists y\ \exists b\ \ (x = y\ \leftrightarrow b)$	$b = 1,$	$\Longrightarrow$	$I_x \subseteq [y^-, y^+]$
	$b = 1,$	$\Longrightarrow$	$I_y \subseteq [x^-, x^+]$
	$b = 0, x^- = x^+, x^- = y^-$	$\Longrightarrow$	$I_y \subseteq [y^- + 1, +\infty]$
	$b = 0, x^- = x^+, x^- = y^+$	$\Longrightarrow$	$I_y \subseteq [-\infty, y^+ - 1]$
	$b = 0, y^- = y^+, y^- = x^-$	$\Longrightarrow$	$I_x \subseteq [x^- + 1, +\infty]$
	$b = 0, y^- = y^+, y^- = x^+$	$\Longrightarrow$	$I_x \subseteq [-\infty, x^+ - 1]$
	$x^+ \leq y^-, y^+ \leq x^-$	$\Longrightarrow$	$b = 1$
	$x^- > y^+$	$\Longrightarrow$	$b = 0$
	$x^+ < y^-$	$\Longrightarrow$	$b = 0$

Figure 7.12 – Règles de propagation pour les contraintes réifiées.

CHAPITRE 8

Résolution de problèmes modélisés par des contraintes quantifiées

Les contraintes quantifiées représentent un outil de formulation et de résolution de problèmes combinatoires complexes issus, notamment, de jeux ou de problématiques d'intelligence artificielle ou de vérification *de programmes. Notre but dans cette section est double : déterminer un ensemble d'exemples de problèmes quantifiées permettant de valider notre approche sur un plan expérimental, et donner un aperçu de l'étendue des applications des contraintes quantifiées.*
Un argument notable en faveur de l'emploi des QCSP est qu'ils permettent d'exprimer des problèmes portant sur des graphes implicites*, c.-à-d. des graphes représentés par la description en termes de contraintes d'une relation d'adjacence entre des sommets représentant en général les configurations d'un système. La taille de ces graphes est typiquement exponentielle par rapport à la longueur de la description du problème par des contraintes.*
De tels graphes présentent des applications importantes et difficiles, et représentent donc une classe d'applications significative des contraintes quantifiées. Nous commençons par présenter ce type de problèmes de manière générale puis en dérivons plusieurs classes d'applications. Dans une seconde section, nous décrivons d'autres types de problèmes auxquels les contraintes quantifiées apportent des réponses intéressantes, et discutons nos résultats expérimentaux.

8.1 Graphes implicites

De nombreuses situations peuvent être modélisées par des problèmes de *graphes* et notamment par des problèmes d'*atteignabilité* dans les graphes, c.-à-d. des problèmes de recherche de chemin permettant d'atteindre un sommet *but* depuis un sommet initial. Par exemple, si les sommets représentent des stations de métro et les arcs des lignes, il est clair que les problèmes d'atteignabilité permettent de trouver un chemin menant d'une station à une autre. Dans le cas de plans de métro ou de lignes aériennes, la donnée du problème est une représentation *explicite* du graphe : on dispose par exemple de la matrice d'adjacence du problème ou d'une base de données stockant la liste des vols prévus. Il existe bien évidemment des algorithmes polynomiaux *en la taille du graphe ainsi représenté* permettant de résoudre ce type de problèmes, comme les algorithmes d'exploration en largeur ou en profondeur. D'autres algorithmes polynomiaux permettent même de minimiser la taille du chemin obtenu, comme celui de DIJKSTRA (dans le cas de graphes à arêtes de poids positifs) [133, 213].

Il est également possible d'utiliser une modélisation par graphes de problèmes dans lesquels les sommets sont les descriptions formelles d'*environnements* d'un ensemble d'agents, ou de *configurations* d'un

système. Les chemins représentent alors, par exemple, des séquences d'actions à exécuter pour passer d'une configuration à une autre, ou la séquence des états successivement pris par un système de transitions au cours du temps. Dans ce type d'applications, une différence notable apparaît : les graphes ne sont plus stockés explicitement mais de manière *implicite* — on dispose seulement d'une description, dans un certain langage formel, des conditions dans lesquelles une action ou une transition sont réalisables. Il serait possible de construire explicitement le graphe correspondant au problème, mais celui-ci serait de taille exponentielle[1].

8.1.1 Définition formelle

Nous proposons une formalisation de la notion de graphe implicite dans le but de montrer que les contraintes quantifiées permettent de modéliser les problèmes d'atteignabilité dans ce type de graphes. Nous détaillerons dans la suite plusieurs classes d'application des graphes implicites. La présentation choisie est basée sur la notion de contrainte. On suppose qu'il existe un certain ensemble de variables $V = \{x_1, \ldots, x_n\}$ (représentant les paramètres décrivant un certain environnement), chaque variable x pouvant prendre une valeur choisie dans un domaine D_x. Une *configuration* (également appelée *sommet*) attribue à chaque variable une valeur de son domaine. La notion de transition entre deux configurations est traduite par ensemble de contraintes entre les variables des deux configurations :

Définition 32. (graphe implicite)

Un graphe implicite est un couple (V, G) où :

- $V = \{x_1, \ldots, x_n\}$ *est un ensemble de* variables, *chaque variable x pouvant prendre une valeur-choisie dans un domaine D_x. Un* sommet *est représenté par un V-tuple, c'est à dire une valuation associant à chaque $x \in V$ une valeur $v_x \in D_x$.*
- *G est un problème de satisfaction de contraintes portant sur $2n$ variables, qu'on notera $\{x_1, \ldots x_n\}$ et $\{x'_1, \ldots x'_n\}$, chaque x et chaque x' ayant le même domaine D_x (intuitivement, deux sommets $\langle v_1, \ldots, v_n\rangle$ et $\langle v'_1, \ldots, v'_n\rangle$ sont reliés par un arc si et seulement si $\langle v_1, \ldots v_n, v'_1, \ldots v'_n\rangle$ est une solution de G). Plus précisément, $G = \{C_1, \ldots C_m\}$ est un ensemble de contraintes, chaque contrainte étant une χ-relation avec $\chi \subseteq \{x_1, \ldots x_n, x'_1, \ldots x'_n\}$.*

Un graphe implicite est donc un graphe dont les sommets sont encodés comme un mot d'une certaine longueur sur un alphabet, et dont *la relation d'adjacence n'est pas donné explicitement comme une matrice d'adjacence, mais comme un CSP*. Notons que si l'ensemble de variables est de taille $\{x_1, \ldots x_n\}$, le graphe généré possède $|D_{x_1}| \times \ldots \times |D_{x_n}|$ sommets, ce qui est exponentiel en n.

[1] La différence en termes de complexité entre les problèmes exprimés sur des données implicites et explicites est parfaitement établie : le problème d'atteignabilité dans les graphes stockés en extension est l'exemple le plus classique de problème **NLOGSPACE**-complet [151, 243], alors que ce problème est **PSPACE**-complet dans le cas de graphes implicites. L'utilisation de représentations implicites ou *succinctes* pour représenter les données d'un problème augmente en général sa complexité de manière exponentielle ; ce phénomène est reporté dans [211, 212, 12] (il est bien connu [230] que **PSPACE** = **NPSPACE**, qui est exactement la classe située une exponentielle au dessus de **NLOGSPACE**).

Historiquement, de nombreuses approches de résolution des problèmes de graphes implicites par construction explicite du graphe de transition associé ont été initialement proposées, par exemple aux premières heures du *model-checking*. Ces approches ont, bien entendu, rapidement montré leurs limites et l'introduction de techniques visant à éviter autant que possible cette construction a considérablement élargi le champ d'application de ces techniques. Le terme couramment employé de *model-checking symbolique* désigne précisément l'utilisation de représentations implicites des graphes de transition de systèmes (le terme est resté, bien que plus personne à l'heure actuelle ne soit tenté d'utiliser des formes non symboliques de *model-checking*!).

Exemple 8.10. *Des exemples simples de graphes implicites sont donnés par les graphe de transition de nombreux jeux de plateaux. Prenons par exemple un jeu de taquin de dimension 3×3 : les 9 variables ($x_1, \ldots, x_9$) du problème correspondent aux cases du plateau. Celles-ci peuvent recevoir une valeur prise dans le domaine $\{A, \ldots, H, \square\}$, selon que la case considérée est occupée par le carré A, le carré B, etc., ou n'est pas occupée du tout (case blanche). Un sommet du graphe correspond donc à un configuration de jeu ; il y a $9! = 362880$ configurations possibles.*
Un arc relie deux sommets s'il est possible, en un seul mouvement, de passer de la configuration correspondant au premier sommet à celle correspondant au deuxième. On peut effectuer 4 types de déplacements : vers le haut, vers le bas, la droite ou la gauche.

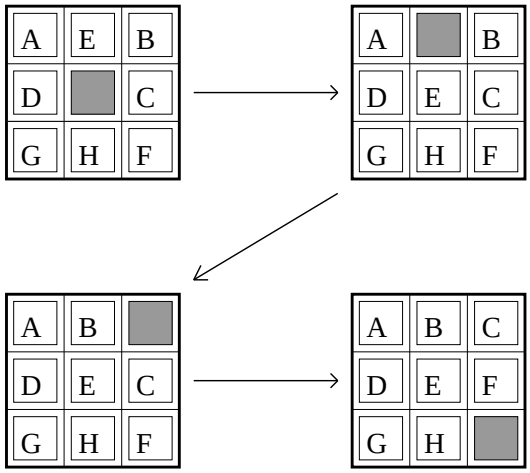


Figure 8.1 – Un exemple de système de transition naturellement modélisé par un graphe implicite.

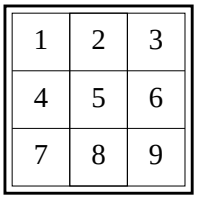


Figure 8.2 – Conventions de numérotation des plateaux de jeu.

Dans tous les cas, les mouvements ne sont possibles que s'ils sont dirigés vers la case vide. Par exemple, les mouvements vers le haut consistent, si l'une des cases 1 à 6 est libre, à échanger la valeur de cette case avec celle située juste au dessous ; on peut modéliser ce type de mouvements par une contrainte portant sur deux vecteurs de variables $X = \{x_1, \ldots, x_9\}$ et $X' = \{x'_1, \ldots, x'_9\}$:

$$h(X, X') \equiv \begin{pmatrix} (x_1 = \square, x'_1 = x_4, x'_4 = \square) \vee \\ (x_2 = \square, x'_2 = x_5, x'_5 = \square) \vee \\ (x_3 = \square, x'_3 = x_6, x'_6 = \square) \vee \\ (x_4 = \square, x'_4 = x_7, x'_7 = \square) \vee \\ (x_5 = \square, x'_5 = x_8, x'_8 = \square) \vee \\ (x_6 = \square, x'_6 = x_9, x'_9 = \square) \end{pmatrix}$$

De même, il est possible d'exprimer les mouvements vers le bas, la droite et la gauche par des contraintes $b(X,Y)$, $g(I,G)$, $d(X,Y)$. *La transition entre deux configurations de jeux s'exprime alors :*

$$arc(X,X') \equiv \left(\begin{array}{l} d(X,X') \vee g(X,X') \vee \\ h(X,X') \vee b(X,X') \end{array} \right)$$

Notez que, bien que cet ensemble de contraintes soit exprimé sous forme disjonctive, il est possible de se ramener à une conjonction par décomposition.

8.1.2 Atteignabilité dans un graphe implicite

Le problème combinatoire le plus typique lié aux graphes implicites est celui d'*atteignabilité*. Celui-ci consiste, étant donnés un graphe, un sommet initial et un sommet but, à déterminer s'il existe une séquence d'arcs menant de l'état initial au sommet but dans le graphe. Ce problème apparaît sous les formes les plus diverses, et permet notamment de résoudre des jeux (taquin), des problèmes de *calcul de plans d'actions*, ou de déterminer si un système de transition est susceptible d'atteindre un état indésirable[2]. Le problème d'atteignabilité consiste à déterminer s'il existe un chemin entre un sommet initial et un sommet but, c.-à-d. :

Définition 33. (problème d'atteignabilité dans un graphe implicite)

Étant donné un graphe implicite (V,G), *un sommet* s' *est dit* immédiatement atteignable *depuis un sommet* s *si le couple* (s,s') *satisfait les contraintes de* G, *ce qu'on notera alors* $s \rightarrow s'$. *La relation d'*atteignabilité *est la clôture transitive de la relation d'atteignabilité immédiate, c.-à-d. la plus petite relation* $\stackrel{\star}{\rightarrow}$ *vérifiant :*

- *Si* $s \rightarrow s'$ *alors* $s \stackrel{\star}{\rightarrow} s'$ *;*
- *Si* $s \stackrel{\star}{\rightarrow} s'$ *et* $s' \stackrel{\star}{\rightarrow} s''$ *alors* $s \stackrel{\star}{\rightarrow} s''$.

Un sommet s_k *est donc atteignable depuis un sommet initial* s_1 *s'il existe une séquence de sommets* $s_1, s_2, \ldots s_k$ *telle que chaque sommet* s_t *soit relié par un arc à* s_{t+1} *(pour* $t \in 1..k-1$*). Une telle séquence est appelée un* chemin. *On peut définir la* longueur *d'un chemin comme le nombre de sommets qui le composent, la* distance *entre deux sommets comme la longueur du plus court chemin les reliant (ou* $+\infty$ *s'ils ne sont pas reliés), et le* diamètre *d'un graphe comme le maximum des distances obtenues pour chaque couple de points atteignables du graphe.*

[2] Il est remarquable que le problème soit apparu dans des domaines aussi divers que l'*intelligence artificielle* (particulièrement le *calcul de plans d'actions*) et la vérification automatique de systèmes d'états/transitions, et il me semble encore plus remarquable que ces communautés aient développé indépendamment des algorithmes de résolution finalement similaires, essentiellement basés sur un petit nombre de techniques. La technique de base consiste en une exploration de graphe par construction partielle [208, 96, 109]. De nombreuses améliorations ont été proposées, visant par exemple à éviter le parcours de certains arcs (par exemple grâce à des fonctions heuristiques dans le cas de la procédure générique de recherche A* [141, 214]). Il semble que les techniques appliquées à l'un des problèmes soient en général applicables aux autres (par exemple les diagrammes de décision binaires, qu'on associe en général aux problèmes de *vérification*, ont récemment été appliqués avec succès au *calcul de plans d'action* [156]), et les avancées récentes sur l'utilisation de techniques SAT ou CSP [164, 163, 130, 58, 94, 57, 114, 41] permettent d'espérer une unification des techniques utilisées, dans laquelle la notion de *contrainte* occupera fort probablement une place centrale. On trouvera dans la partie *graph-search and STRIPS-planning* du cours du MIT de MCALLESTER [195] une introduction accessible aux diverses facettes du problème de recherche dans les graphes implicites en IA. A noter également un *survey* récent [223] et les notes de cours de J. RINTANEN sur le *calcul de plans d'action* [222]. Le livre de référence sur le *model-checking* est [59]. A l'heure actuelle, je ne connais pas d'ouvrage abordant ces divers aspects des techniques de recherche dans les graphes implicites de manière unifiée.

8.1.2.1 Encodage des problèmes d'atteignabilité par CSP ; problème de diamètre

Dans notre modélisation des graphes implicites, la relation d'adjacence est modélisée par des contraintes, et il n'est pas difficile d'utiliser des contraintes pour exprimer des relations plus complexes. On notera $\text{arc}(x_1, \ldots, x_n, y_1, \ldots, y_n)$ l'ensemble de contraintes modélisant les arcs du graphe. Introduisant n variables supplémentaires (disons $z_1, \ldots, z_n$), il est clairement possible d'exprimer l'existence de chemins de longueur 2 par le CSP suivant :

$$\text{arc}(x_1, \ldots, x_n, z_1, \ldots, z_n) \ \wedge \ \text{arc}(z_1, \ldots, z_n, y_1, \ldots, y_n)$$

La configuration $Z = \langle z_1, \ldots z_n \rangle$ représente le sommet intermédiaire, par lequel on passe pour aller de X à Y. Plus généralement, posant k ensembles de n variables $X^t = \{x_1^t, \ldots, x_n^t\}$ (avec $t \in 1..k$), il est possible d'exprimer l'existence d'un chemin de longueur k (l'ensemble X^1 correspond à l'état initial, l'état X^k correspond à l'état but) par une conjonction de contraintes[3] :

$$\bigwedge_{t \in 1..k-1} \text{arc}(X^t, X^{t+1})$$

En général, on n'est cependant pas spécialement intéressés par des chemins de longueur *exactement* k, mais on souhaite plutôt déterminer s'il existe des chemins de longueur *au plus* k. Notons *chemin_k* l'ensemble de contraintes spécifiant cette propriété ; on a :

$$\begin{aligned}
&\text{chemin_1}(X, Y) \equiv \\
&\quad X = Y \ \vee \ \text{arc}(X, Y) \\
&\\
&\text{chemin_2}(X, Y) \equiv \\
&\quad X = Y \ \vee \ \text{chemin_1}(X, Y) \\
&\ldots
\end{aligned}$$

Où l'égalité entre deux états X et Y correspond à la conjonction :

$$\bigwedge_{i \in 1..n} X_i = Y_i$$

On parvient donc à écrire cette propriété en termes de contraintes, en dupliquant k fois l'ensemble des variables et des contraintes les reliant. On peut maintenant se demander quelle limite fixer à k : que se passe-t'il en effet si on souhaite déterminer s'il existe un chemin entre deux sommets, sans limite sur la longueur de celui-ci? Le problème est qu'il est difficile de borner la longueur des chemins possibles. Dans le pire des cas, un chemin pourrait passer par tous les sommets possibles du graphe implicite, or on sait que leur nombre est exponentiellement élevé par rapport au nombre de paramètres du problème (rappelons qu'il existe $|D_{x_1}| \times \ldots \times |D_{x_n}|$ sommets). En fait, dans le cas général, il est impossible de proposer un encodage du problème d'atteignabilité par un CSP. La justification est que le problème d'atteignabilité est **PSPACE**-complet, et qu'une réduction polynomiale à SAT ou CSP prouverait que **NP** = **PSPACE**, ce que personne ne pense exact.

[3] La traduction de l'atteignabilité bornée dans un graphe implicite en termes de contraintes proposée ici a été introduite indépendamment sous le terme *planning as satisfiability* [164, 163] dans la communauté *Intelligence Artificielle* et sous le terme *bounded model-checking* par la communauté vérification [28, 58, 57, 114].

8.1.2.2 Encodage des problèmes d'atteignabilité par QCSP

Il est possible d'utiliser des contraintes quantifiées pour exprimer le problème d'atteignabilité de manière concise. Rappelons que le but est d'exprimer l'existence d'un chemin de longueur n où n est exponentiel par rapport au nombre de paramètres du problème. On cherche donc un encodage de longueur *logarithmique* en n. L'idée de cet encodage est de découper le problème en 2 : un chemin de longueur $2.n$ passe par un certain sommet situé en son *milieu*, et exprimer l'existence d'un tel chemin nécessite donc de savoir tester l'existence d'un chemin de longueur n. On peut ainsi exprimer la contrainte chemin_$2n$ à partir de la contrainte chemin_n, et on espère ainsi avoir recours à $\log(n)$ contraintes. Une application naïve de cette idée est la suivante :

$$\text{chemin\_}2n(X,Y) \;\equiv\; \exists M \;\big(\; \text{chemin\_}n(X,M) \wedge \text{chemin\_}n(M,Y) \;\big)$$

Cependant cet encodage ne résout en rien le problème : en formulant ainsi le prédicat chemin_$2n$, on a dû recopier *deux fois* l'ensemble de contraintes exprimant chemin_n, ce qui ruine l'économie réalisée en divisant par deux la longueur testée. Il nous faudrait en fait utiliser la même contrainte chemin_n pour exprimer l'existence du chemin entre X et M et de celui entre M et Y. Les quantificateurs permettent cette réutilisation ; on obtient la définition récursive suivante :

$$\begin{array}{l}\text{chemin\_}1(X,Y) \equiv \\ \qquad X = Y \vee \text{arc}(X,Y)\end{array}$$

$$\begin{array}{l}\text{chemin\_}2n(X,Y) \equiv \\ \qquad \exists M\; \forall A\; \forall B \left(\begin{array}{l}(A = X \wedge B = M) \vee \\ (A = M \wedge B = Y)\end{array} \right) \rightarrow \text{chemin\_}n(A,B)\end{array}$$

La technique consiste à considérer tous les couples de sommets et à tester s'il s'agit des extrémités de l'un ou l'autre des demi-chemins, auquel cas on imposera qu'il soient connectés grâce à la contrainte d'existence d'un chemin de longueur n. Il est clair que cet encodage[4] atteint les objectifs fixés précédemment, et que la taille de la formule ainsi générée est polynomiale par rapport au nombre de paramètres du graphe implicite. Elle peut de plus facilement être mise sous forme prénexe. En guise d'exemple, le QCSP suivant, utilisant 3 copies du motif de quantificateurs $\exists M\; \forall A\; \forall B$, exprime l'existence d'un chemin de longueur 8 :

$$\begin{array}{l}chemin_8(X_0,Y_0) \equiv \\ \quad \exists M_1\; \forall X_1\; \forall Y_1\; \exists M_2\; \forall X_2\; \forall Y_2\; \exists M_3\; \forall X_3\; \forall Y_3 \\ \qquad ((X_1 = X_0 \wedge Y_1 = M_1) \vee (X_1 = M_1 \wedge Y_1 = Y_0)) \rightarrow \\ \qquad\quad ((X_2 = X_1 \wedge Y_2 = M_2) \vee (X_2 = M_2 \wedge Y_2 = Y_1)) \rightarrow \\ \qquad\qquad ((X_3 = X_2 \wedge Y_3 = M_3) \vee (X_3 = M_3 \wedge Y_3 = Y_2)) \rightarrow \;\; (X_3 = Y_3 \vee arc(X_3,Y_3))\end{array}$$

8.2 Problèmes de calcul de plans d'action

Le *calcul de plans d'action*[5] est un domaine de recherche important en Intelligence Artificielle consistant à déterminer une séquence d'actions permettant d'atteindre un certain *but* à partir d'une *situation* donnée. Le problème du taquin présenté en figure 8.10 ou des variantes comme *Rubick's cube*

[4] Cet encodage est en fait à la base de la preuve de **PSPACE**-complétude de QCSP [241], qui est une transposition en termes logiques par des quantificateurs de l'idée de *réutilisation de l'espace* utilisée par W. SAVITCH [230] pour prouver que le problème d'atteignabilité est soluble en espace $\mathcal{O}(log^2 n)$. Ma présentation de cette partie s'inspire de [212].

[5] Traduction du terme anglais *Planning* dans son acceptation usuelle en Intelligence Artificielle.

ou les *tours de Hanoï* sont des exemples de problème de calcul de plans d'action. Des applications plus sérieuses sont fournies par la robotique (calcul de mouvements) ou la prise de décision (calcul d'un plan d'action permettant de transporter un ensemble de marchandises entre différents lieux).

L'environnement est représenté par un certain nombre de variables, et les situations initiale et finale sont modélisés par un ensemble de faits portants sur ces variables. Quant aux *actions* qu'il est possible d'appliquer, elles sont modélisées comme des *fonctions de transition* modifiant la base de faits représentant une situation donnée. Le non-déterminisme provient du fait que plusieurs actions sont applicables à chaque instant — on a donc bien affaire à une *relation* de transition entre les différents environnements possibles.

8.2.1 STRIPS

Un des formalismes les plus communément adoptés pour l'expression de problèmes de calcul de plans d'action est STRIPS[6] [109]. Nous le présentons rapidement en nous basant sur un exemple simple de problème. Notre but étant uniquement de suggérer l'expression de problèmes de calcul de plans d'action en termes de contraintes quantifiées, nous renvoyons le lecteur aux articles fondateurs [109] ou aux textes introductifs [227, 222] et *états de l'art* [223] plus récents pour de plus amples détails sur le sujet.

L'exemple considéré est le classique *problème des cubes*[7] : un certain nombre de cubes (identifiés par des lettres différentes) sont disposés sur une table, certains étant empilés sur d'autres. Le but est de trouver une séquence de déplacements de cubes permettant d'obtenir (par exemple) un tas respectant un certain ordre d'empilement. Faisant abstraction de nombreux détails concernant les coordonnées spatiales des objets, on réduit l'observation d'un environnement à un petit nombre paramètres comportant essentiellement la présence d'un cube sur un autre ou sur la table, ou le fait que le dessus d'un cube soit occupé ou non. Notez que ces observations sont essentiellement *propositionnelles*.

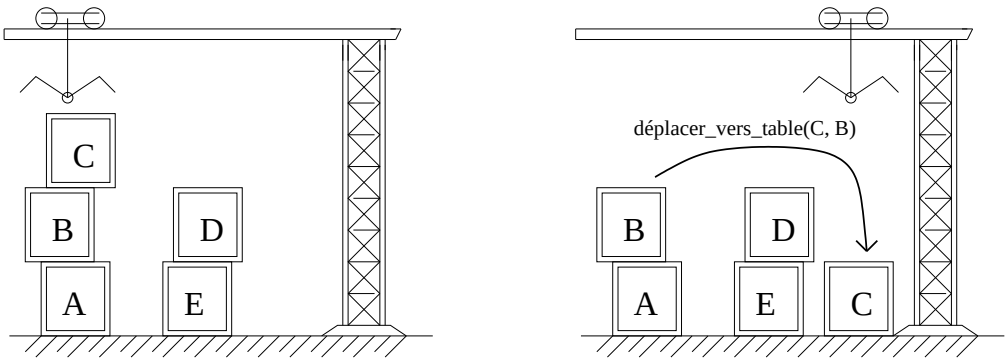


Figure 8.3 – Le problème de déplacement de cubes.

En STRIPS, l'ensemble des objets composant une situation est représenté par un ensemble de constantes (ici $A..E$ et $table$), et la situation elle-même est décrite par un ensemble de faits portant sur ces constantes. Par exemple, la situation initiale de notre exemple est représentée par :

$$sur(C, B), sur(B, A), sur(D, E), libre(C), libre(D)$$

Le prédicate $libre$ exprime le fait que le dessus d'un objet est libre, et que celui-ci peut-donc recevoir un autre objet. D'autres prédicats peuvent être introduits afin, par exemple, de distinguer les différents types

[6] Stanford Research Institute Problem Solver.
[7] Blocks-world.

d'objets (dans notre exemple, les cubes doivent être distingués de la table par le biais d'un prédicat $cube$ et des faits $cube(A), \ldots, cube(D)$).

Les actions applicables dans une situation donnée sont spécifiées par des règles composées d'une *précondition*, d'une liste d'*ajouts* de faits et d'une liste de *suppressions* de faits. La précondition correspond aux hypothèses déterminant si la règle peut être appliquée ; les ajouts correspondent aux faits qui sont validés par l'action ; les suppressions aux faits qui sont invalidés. Il est clair en effet que le rôle des actions présente un aspect *non monotone*, puisque certaines observations (présence du cube C sur le cube B) peuvent devenir fausses après certaines actions (déplacement du cube B vers la table). Les suppressions sont d'ailleurs parfois écrites comme des négations par certains auteurs.

Deux actions sont applicables au problème des cubes : il est possible de déplacer un cube vers la table ou de déplacer un cube pour le déposer sur un autre cube[8] :

déplacer$(x, orig, dest)$:

Précondition	$sur(x, orig), libre(x), libre(dest), cube(x)$
Ajouts	$sur(x, dest), libre(orig)$
Suppressions	$sur(x, orig), libre(dest)$

déplacer_vers_table$(x, orig)$:

Précondition	$sur(x, orig), libre(x), cube(x)$
Ajouts	$sur(x, table), libre(orig),$
Suppressions	$sur(x, orig)$

On peut juger utile d'ajouter certaines préconditions d'inégalité (comme $orig \neq dest$), mais le seul danger de notre omission est que l'espace de recherche obtenu s'en retrouve plus grand — sa correction n'est pas compromise.

8.2.2 Traduction de STRIPS en QCSP

Il est bien connu que la formulation d'un problème en STRIPS peut être traduite en termes de contraintes — cette technique est à la base de l'approche *planning as satisfiability* [163, 164]. La démarche est la suivante :

Les variables propositionnelles modélisent chacun des *faits* possibles du langage de description. Chaque prédicat p est traduit par un *tableau* de variables logiques associant à chaque constante c une variable $p[c]$. Par exemple, une variable booléenne indiquera si A est au dessus de B. Cette variable, notée $sur[A, C]$, correspond au fait $sur(A, C)$ en termes STRIPS. Les variables correspondant au prédicat sur possèdent initialement les valeurs suivantes (le problème possède d'autres variables issues des prédicats $cube$ et $libre$) :

sur	A	B	C	D	E	$table$
A	0	1	0	0	0	0
B	0	0	1	0	0	0
C	0	0	0	0	0	0
D	0	0	0	0	0	0
E	0	0	0	1	0	0
$table$	1	0	0	0	1	0

[8] Ma spécification de ce problème est, pour l'essentiel, tirée de [227].

Les contraintes correspondent à chacune des actions qu'il est possible d'effectuer. Par exemple, il est possible de déplacer le cube A sur B, ou de le cube B sur la table, etc. Chaque règle STRIPS permet donc de dériver autant de relations de transition qu'il existe d'instantiations possibles de la règle par des objets de l'environnement (en indiquant les valeurs de chaque variable avant et après l'action[9]. Le problème de contraintes obtenu au bout du compte est la disjonction des contraintes correspondant aux différentes actions possibles.

On modélise bien ainsi la relation de transition entre deux sommets, c.-à-d. les arcs du graphe implicite correspondant au problème de calcul de plans. L'encodage est en général naturellement booléen, mais des variantes utilisant pleinement l'expressivité des contraintes sur les domaines finis ont été proposées : la conclusion de [94] est que "*CSP est un meilleur substrat pour la compilation [des problèmes de* planning*], qu'il soit comparé à SAT ou à la programmation linéaire en nombres entiers*".

Reste à estimer le diamètre du graphe ainsi obtenu afin de borner la nombre de répétitions du bloc de variables dans la formule quantifiée. Il est toujours suffisant de prendre une formule avec un nombre de motifs $\exists M\ \forall A\ \forall B$ inférieur au nombre de variables (ayant n variables, on a 2^n états et donc un diamètre inférieur à 2^n. Une formule quantifiée formée sur une répétition de n motifs suffit à exprimer l'atteignabilité dans ce type de graphes).

Exemple 8.11. *Considérons le problèmes à deux cubes A et B avec pour configuration initiale A posé sur B, et comme but B sur A. Une configuration est décrite par les variables propositionnelles suivantes (T représente la table) :*

$$\mathcal{X} = \begin{pmatrix} sur\_A\_A & sur\_A\_B & sur\_A\_T \\ sur\_B\_A & sur\_B\_B & sur\_B\_T \\ sur\_T\_A & sur\_T\_B & sur\_T\_T \\ cube\_A & cube\_B & cube\_T \\ libre\_A & libre\_B & libre\_T \end{pmatrix}$$

Les actions possibles sont les suivantes :

déplacer_vers_table_A_A	*déplacer_vers_table_A_B*	*déplacer_vers_table_A_B*
déplacer_vers_table_B_A	*déplacer_vers_table_B_B*	*déplacer_vers_table_B_B*
déplacer_vers_table_T_A	*déplacer_vers_table_T_B*	*déplacer_vers_table_T_B*
déplacer_A_A_A	...	...

Il est évident que certaines des variables sont en fait constantes ($libre\_T$ n'est modifiée par aucune action et sera donc toujours vraie*) et qu'un grand nombre des actions sont impossibles à réaliser du fait des préconditions posées (toutes les actions de déplacement de la table sont interdites du fait de la contrainte imposant que l'objet déplacé soit un cube : $cube\_T$ ne sera jamais vrai). Ces problèmes proviennent du fait que notre modèle possède une seule classe d'objets, utiliser des variantes "typées" de STRIPS, permettant de préciser que les cubes et la table sont des objets de nature différentes sur lesquels des actions différentes s'appliquent, permettrait d'affiner notre modèle. Le principe resterait toutefois essentiellement le même.*
Chacune de ces actions est une relation entre un ensemble de variables initial et un ensemble final. Par exemple, l'action d'enlever A du dessus de B et de le déposer sur la table est modélisée comme suit :

[9] Il est nécessaire de préciser explicitement les égalités correspondant aux variables laissées inchangées par l'action. Ce besoin a posé de nombreux problèmes logiques à la communauté IA (problème du *frame*) mais ne pause pas de problème particulier dans notre cas.

déplacer_vers_table$\_A\_B(\mathcal{X}, \mathcal{X}') \equiv$

$$\left(\begin{array}{ccccc} sur\_A\_B = 1 & \wedge & libre\_A = 1 & \wedge & cube\_A = 1 \\ sur\_A\_B' = 0 & \wedge & libre\_B' = 1 & \wedge & sur\_A\_T = 1 \\ \multicolumn{5}{c}{\bigwedge_{x \in (\mathcal{X} - \{sur\_A\_B, libre\_B, sur\_A\_T\})} x = x'} \end{array} \right) \wedge$$

Où $\mathcal{X}$ et $\mathcal{X}'$ représentent, l'ensemble des variables de l'environnement avant et après l'action (il s'agit donc d'une contrainte portant sur 2 $\times$15 variables).
Finalement, la formule quantifiée exprimant l'atteignabilité de la configuration but *à partir de la configuration initiale nécessite 15 répétitions du bloc de contraintes, et donc 30 alternations de quantificateurs.*

8.3 Problèmes de vérification de systèmes de transition

De nombreux automates, protocoles et systèmes asynchrones peuvent être modélisés en termes de graphes[10] et, là encore, le graphe de la relation de transition entre deux états du système considéré est souvent décrit de manière implicite. Nous présentons maintenant un aperçu de l'utilisation des contraintes quantifiées pour les problèmes de *vérification* de systèmes finis, en particulier ceux pour lesquels les techniques de *model-checking* [59] sont utilisables. Comme pour le calcul de plans d'actions, notre présentation s'appuie essentiellement sur un exemple et tente simplement de donner une idée suffisamment précise des techniques de conversion utilisables pour modéliser ce type de problèmes par des QCSP. Deux ouvrages de référence donnant de plus amples détails sur le sujet sont [59] pour le *model-checking* et [189] pour la spécification de systèmes[11].

8.3.1 Modélisation de systèmes concurrents

Considérons en guise d'exemple un groupe de n processus devant accéder à un certain nombre de ressources de manière exclusive[12]. Supposons, par exemple, que chacun des n processus exécute le même code, qui consiste à effectuer en boucle des opérations mobilisant la ressource critique. Une solution simple (encore très insatisfaisante) au problème d'exclusion mutuelle consiste à utiliser une variable

[10] Il est intéressant de mentionner ici deux autres problèmes en rapport à la fois avec la vérification de systèmes concurrents et avec les contraintes quantifiées : le problème d'équivalence d'expressions régulières et celui de preuve en logique de HOARE propositionnelle. Les expressions régulières sont une notation permettant de décrire des automates finis (ceux-ci reconnaissent les langages réguliers, qui coïncident exactement avec la classe des langages reconnaissables en *espace constant* [212]). Déterminer l'équivalence d'automates finis présente un intérêt évident pour les problèmes de vérification de systèmes de transition, et permet par exemple de vérifier l'équivalence entre une spécification et sa mise en œuvre. Quant à la logique de HOARE [5], c'est un formalisme de preuve de programme basé sur les notions de *précondition* et *postcondition*.

Les deux problèmes sont **PSPACE**-complets [212, 63] et peuvent donc faire appel au même type d'algorithmique que les problèmes considérés dans cette section — ceci n'est finalement guère surprenant vues que ces deux problèmes sont, là encore, fortement liés aux graphes implicites.

[11] Les outils de *model-checking* trouvent également des applications dans d'autres domaines que les systèmes concurrents, voir par exemple [52] pour une application à la modélisation de processus biologiques.

[12] Bien qu'une présentation plus poussée des problématiques de programmation concurrente dépasse à la fois la portée de cette thèse et le champ de compétence de son auteur, on rappelle qu'une *section critique* est une séquence d'instructions qui requiert que le processus l'exécutant se voit attribuer un certain nombre de ressources de manière exclusive : aucun autre proccessus ne doit pouvoir opérer de lecture ou d'écriture sur les emplacements mémoire ainsi attribués, ceci jusqu'à la fin de l'exécution de la séquence d'instructions. Il s'agit là d'un problème classique particulièrement important dans les systèmes d'exploitation.

partagée (*tour*), dont la valeur donne le numéro du processus ayant le droit d'accès aux dites ressources. Chaque processus est responsable de rendre la main en fin de section critique. L'algorithme exécuté par chaque processus est le suivant :

Processus (id **: entier)**
1 **tant que** $vrai$ **faire**
2 | **tant que** $tour \neq id$ **faire**
3 | | **rien**
| | **fin tant que**
| | {utilisation de la ressource critique}
4 | | $tour \leftarrow (tour + 1)$ **mod** $nb\_processus$
5 **fin tant que**
fin processus

On exécutera en parallèle un certain nombre n de processus, fonctionnant tous sur le même algorithme ; on fournit en paramètre de chaque processus un numéro distinct permettant de l'identifier. On exécute donc au final le programme suivant :

$$\textbf{\textit{processus}}(0) \parallel \ldots \parallel \textbf{\textit{processus}}(n-1)$$

Pour donner une idée plus précise des problématiques de vérification posées par ce programme, supposons par exemple que les processus sont exécutés sur un unique processeur attribuant tour à tour une partie de son temps de calcul à chacun d'entre eux. A chaque instant, le processeur exécute l'instruction courante du processus actif puis a le choix entre continuer sur le même processus ou désigner un nouveau processus comme actif. Un exemple de propriété souhaitable est de garantir l'absence de *blocage*.

8.3.2 Traduction d'un programme concurrent en termes de contraintes

Nous ébauchons brièvement une façon d'exprimer la relation de transition de ce système en termes de contraintes ; la quantification permettra alors d'exprimer l'atteignabilité dans le graphe de transition correspondant[13]. Il est important d'expliciter clairement le principal prérequis de notre méthode : celle-ci suppose que les domaines des variables soient *finies* (ce qui arrive naturellement dans un grand nombre de problèmes de vérification de circuits ou machines asynchrones). [14]

L'état du système comporte la variable partagée *tour* et le contexte d'exécution de chacun des programmes. En faisant abstraction des traitements effectués par chacun de ces programmes, on pourra se contenter de représenter leur contexte par leur pointeur de code, identifié au numéro de ligne (de 1 à 7) de l'instruction en cours d'exécution dans le programme abstrait représenté ci-dessus. La relation de transition est de forme suivante :

$$transition(tour, l_0, \ldots, l_{n-1}, tour', l'_0, \ldots, l'_{n-1})$$

Les variables l_i et l'_i représentent le contexte d'exécution du processus i, respectivement avant et après la transition. La relation de transition dépend d'un choix effectué de manière non-déterministe par le processeur, celui-ci pouvant choisir à tout instant d'attribuer la main à un processus différent. Elle est

[13] En logique temporelle, on emploie en général le terme *structure de* KRIPKE pour désigner les graphes implicites dans lesquels les énoncés sont interprétés.

[14] Cette restriction est également imposée par les outils classiques de *model-checking*, qu'ils soient basées sur des diagrammes de décision binaires ou sur des résolveurs SAT ; elle ne doit donc être considérée un défaut majeur de l'approche QCSP.

donc formée de la disjonction de n contraintes (une par choix de processeur à qui donner la main). Par exemple, la deuxième de ces relations, qui correspond au cas où le processeur choisit d'exécuter une instruction du processeur 2,

$$transition_2 \quad (tour, l_0, \ldots, l_{n-1}, tour, l'_0, \ldots, l'_{n-1})$$

$$\equiv \quad \bigvee \left(\begin{array}{l} (l_2 = 1 \wedge l'2 = 2) \\ (l_2 = 2 \wedge tour = 2 \wedge l'2 = 4) \\ (l_2 = 3 \wedge l'_2 = 2) \\ \bigvee_{i=0..n-1} (l_2 = 4 \wedge tour' = i + 1 \wedge l_2 = 1) \end{array} \right)$$

Pour être plus précis, nous avons seulement mentionné dans chacune des disjonctions les valeurs subissant un changement ; comme toujours il faut ajouter un ensemble de contraintes à chaque élément de la disjonction pour préciser que les variables non mentionnées demeurent inchangées.

8.4 Expérimentations

L'algorithme de résolution de contraintes quantifiées par recherche-élagage basé sur les règles de propagation présentées dans le chapitre 7 a été implanté en OCAML ; le code est disponible à l'URL `http://www.sciences.univ-nantes.fr/info/perso/permanents/bordeaux/OCAML/`. Les méthodes implantées comportent la propagation booléenne (jeu de règles pour les contraintes $\vee$ et $\neg$ réifiées), les comparaisons réifiées ($\leq$ et $=$) et l'addition et la multiplication par une constante. Le prototype permet donc d'exprimer les problèmes quantifiés linéaires (propagateurs pour le jeu complet de contraintes booléennes $\{\neg, \vee\}$, pour l'addition et la multiplication par une constante, et pour le jeu complet de contraintes de comparaison $\{=, \leq\}$, en version réifiée). L'implantation comporte à l'heure actuelle environ 2500 lignes de code, dont la partie la plus importante (fichier `narrowing.ml` de plus de 800 lignes) est consacrée à la traduction des règles de propagation (traduction des tableaux présentés dans le chapitre 7). Le moteur de propagation est un AC3 classique.

L'intérêt de la technique est testé sur des instances d'un jeu similaire à celui présenté dans le début de cette partie de la thèse (t représente le nombre de tours, a le nombre d'allumettes de jeu, le domaine est $1..d$) :

$$\exists x_0 \ \forall y_1 \, \exists x_1 \ \ldots \ \forall y_t \, \exists x_t \ \left(\ x_0 + \textstyle\sum_{i \in \{1,\ldots,9\}} (y_i + x_i) \ \right) = a$$

Il existe des instances satisfaisables de ce jeu pour des tailles arbitraires ; par exemple, fixant le nombre d'allumettes à $1..10$, on utilise les jeux ($t = 5, a = 60$), ($t = 6, a = 70$), ($t = 7, a = 80$), etc. Ce type de jeux correspond à une version simplifiée de problèmes de gestion de stock, et on peut construire des variantes modélisant les situations dans lesquelles on produit chaque mois i une quantitié p_i d'un produit (celle-ci est bornée par une capacité maximale de production). La demande pour le produit au cours de chaque mois est une variable d_i dont une estimation des bornes est disponible. Le stock de chaque mois i est noté s_i. L'ensemble des évolutions possibles du stock est modélisé par la formule :

$$\exists p_0 \ \forall d_1 \, \exists s_1 \exists p_1 \ \ldots \ \forall d_t \, \exists s_t \exists p_t \quad \bigwedge_{i \in 1..t} \left(\ s_i = s_{i-1} + p_i - d_i \ \right)$$

L'ajout de contraintes de positivité des p_i permet, par exemple, d'obtenir une production suffisante, pour garantir l'approvisionnement. Des exemples typiques de temps de calcul donnés par le prototype sont les suivants, sur une version du jeu d'allumettes (la taille de l'instance utilisée est exprimée en nombre de variables, et les temps sont en secondes) :

taille de l'instance	11	13	15
recherche + propagation	14.2	147.7	1536
recherche	67.2	6079	> 60000

La propagation diminue la vitesse de croissance de la courbe, et elle joue donc bien son rôle. En revanche, il est clair que les temps de calcul demeurent importants et que la technique ne permet pas de résoudre des instances de taille élevée. Un des points faibles de notre implantation est à l'heure actuelle l'algorithme naïf de décomposition en contraintes primitives. Nous introduisons les quantificateurs existentiels en fin de préfixe, ce qui aboutit souvent à la création de contraintes de motif $\forall\exists$ ou $\forall\exists\exists$. Une décomposition plus fine permettrait de produire des contraintes de type $\exists\forall\exists$, qui offrent un filtrage plus fort et semblent plus adaptées à ce type de problèmes.

On reporte également l'importance de l'utilisation d'une propagation incrémentale de type AC3 ; dans une première version utilisant un calcul de point fixe naïf, les temps de calcul étaient multipliés par un facteur d'environ 10 (l'utilisation de la propagation ralentit alors la résolution de certaines instances de petite taille). De plus, on constate que la propagation duale permet d'obtenir des réductions de domaine. Cependant, pour les problèmes utilisant des contraintes d'égalité, le gain obtenu en termes de temps est quasiment nul : l'intérêt faible des réductions de domaine semble dans ce cas contrebalancé par le surcoût induit par la propagation duale.

8.5 Conclusion et remarques

Les contraintes quantifiées permettent d'exprimer certains problèmes combinatoires qui ne peuvent en général pas être traduits directement comme des CSP, ceux-ci permettant seulement d'en exprimer des versions bornées. Il est clair que l'intérêt de ce gain d'expressivité dépend du type d'application. On peut par exemple s'interroger sur le sens de plans d'action de longueur déraisonnable : si un nombre astronomique d'actions est nécessaire pour parvenir à un but donné, il importe finalement peu que l'ordinateur soit capable de produire un tel plan, fût-ce sous une représentation compacte — il sera de toute façon impossible d'atteindre le but fixé. Il peut donc être *préférable,* pour certaines applications, de rechercher uniquement des chemins de taille raisonnable dans le graphe implicite modélisant le problème. À l'inverse, cette limitation est insatisfaisante si on considère certains problèmes pour lesquels *l'absence de chemin doit être garantie*. C'est souvent le cas, par exemple des problèmes de vérification, dans lesquels il peut ne pas être raisonnable de garantir l'absence de panne seulement jusqu'à une borne donnée de temps d'exécution du système.

Pour être totalement explicites et exhaustifs, insistons sur le fait que notre méthode n'est pas la seule possible pour répondre au problème de diamètre ; d'autres techniques, notamment basés sur un calcul de points-fixes, peuvent nous en affranchir [113, 72, 89, 114]. Un argument en faveur de l'expression de problèmes d'atteignabilité par des contraintes quantifiées et que leurs algorithmes de résolution ont naturellement une consommation en espace poynomiale, la quantification exprimant pour l'essentiel l'idée de l'algorithme de SAVITCH. Une telle garantie ne me parait pas évidente pour de nombreux algorithmes d'exploration de graphes.

Malgré l'intérêt potentiel de notre méthode, les résultats expérimentaux sont seulement probants pour des instances de taille limitée. L'élagage obtenu par l'arc-consistance est réel et son intérêt augmente avec la taille des instances. Un traitement symbolique permettant d'optimiser la décomposition des QCSP en contraintes primitives devrait permettre de tirer d'avantage profit des contraintes universelles des problèmes.

PARTIE IV

Extraction de programmes à partir de spécifications logiques

Nous nous sommes intéressés précédemment aux problèmes de satisfaisabilité liés aux contraintes quantifiés. Dans ce type de problèmes, il est possible de traduire n'importe quelle instance d'un problème ***PSPACE*** *en une formule quantifiée dont la satisfaisabilité répond au problème. L'alternation de quantificateurs dans ce type de formules rend le problème difficile.*
Un autre aspect des logiques quantifiées est lié aux problèmes de vérification de modèles. Dans ce type d'approche, une formule logique exprime un problème et le fait de vérifier si une structure finie est une solution correspond au problème de vérification de modèles. Un exemple intéressant de ce type de logique est la logique existentielle de second ordre, qui permet d'exprimer exactement l'ensemble des problèmes ***NP****. Dans cette logique, les quantificateurs de premier ordre jouent un simple rôle de* boucles *permettant de répéter un motif de contraintes sur un ensemble de variables.*
La logique existentielle de second-ordre permet d'exprimer de nombreux problèmes ***NP*** *de manière extrêmement naturelle, en particulier ceux liés aux graphes. Il est possible d'obtenir automatiquement de la spécification un algorithme non-déterministe polynomial pour les problèmes exprimés dans cette logique. Un algorithme polynomial peut parfois être déduit des spécifications. Nous proposons une technique simple basée sur un raisonnement propositionnel pour obtenir cette garantie dans certains cas.*

CHAPITRE 9

Extraction de programmes à partir de spécifications logiques

Nous étudions la compilation de problèmes exprimés en logique existentielle de second ordre*, un langage expressif dans lequel une large classe de problèmes combinatoires peut être exprimée. On entend par "compilation" le fait de rendre explicites certaines règles de déduction qui sont implicites dans la formulation initiale du problème. Le problème initial est ainsi traduit dans une forme soluble par des méthodes de raisonnement simples. Notre approche est une adaptation de techniques de* compilation de connaissances*, qui ont récemment été proposées pour accélérer les déductions dans les bases de connaissances. Nous l'illustrons sur des problèmes simples issus de la théorie des graphes. Nous montrons que certains algorithmes classiques peuvent être extraits automatiquement de la formulation logique du problème. La technique de compilation est nécessairement incomplète et son succès repose sur une propriété de localité. Nous proposons un test à base de compilation de connaissances permettant de tester si cette propriété est vérifiée.*

9.1 Introduction

Comprendre la manière dont un algorithme est obtenu à partir de la spécification logique d'un problème calculatoire est une question centrale pour de nombreux domaines liés à la logique et la combinatoire, incluant la programmation par contraintes, la programmation logique, ou les méthodes formelles. La possibilité d'exprimer des problèmes dans un langage de haut niveau et d'obtenir automatiquement un programme exécutable représente l'idéal de la programmation déclarative. Cependant, en raison de nombreux résultats d'indécidabilité, nous savons que ce but ne sera jamais entièrement réalisé, et nous devons concentrer nos recherches sur des fragments restreints du langage logique, et sur des classes restreintes de problèmes.

Nous considérons ici plus spécifiquement le problème de génération d'un algorithme efficace (ce qu'on entendra ici au sens de "*polynomial* en temps") à partir d'un langage de spécification de haut niveau permettant d'exprimer de nombreux problèmes combinatoires. Le langage en question est un fragment du langage logique à la fois expressif et naturel appelé "*logique existentielle du second ordre*". Les problèmes que nous considérons sont issus de la théorie des graphes, et essentiellement de problèmes **NP**-complets liés aux graphes (bien que la méthode présentée ne soit *a priori* pas intrinsèquement limitée aux problèmes de graphes, il semble qu'elle trouve dans ce domaine ses exemples d'application les plus naturels).

9.1.1 Aperçu et mise en perspective de la méthode proposée

En guise d'illustration de notre approche, considérons le problème de *bipartition*, qui consiste à déterminer si un graphe est 2-coloriable. Une manière extrêmement naturelle et formelle à la fois de spécifier ce problème est la suivante[1] :

$$\begin{array}{ll} \exists \mathrm{R}\backslash 1 \;\; \exists \mathrm{B}\backslash 1 & \\ \quad \forall x & \left(\begin{array}{l} \mathrm{R}(x) \vee \mathrm{B}(x), \\ \neg(\mathrm{R}(x) \wedge \mathrm{B}(x)) \end{array} \right), \\ \quad \forall x \; \forall y & \left(\begin{array}{l} \neg(\mathrm{Arc}(x,y) \wedge \mathrm{R}(x) \wedge \mathrm{R}(y)), \\ \neg(\mathrm{Arc}(x,y) \wedge \mathrm{B}(x) \wedge \mathrm{B}(y)) \end{array} \right) \end{array} \tag{9.1}$$

En d'autres termes (et cela ne surprendra personne), un graphe est biparti si et seulement s'il existe deux couleurs R (pour *rouge*) et B (pour *bleu*) telles que tout sommet se voit attribuer exactement une de ces couleurs et qu'aucune paire de sommets adjacents ne reçoive la même couleur.

Un *modèle* de l'énoncé S donné en (9.1) est défini comme un ensemble V (de sommets) et par une relation $\mathrm{Arc} \subseteq V \times V$, vraie si deux sommets sont connectés. Notez que dans l'énoncé S, les variables universellement quantifiées x et y prennent valeur dans le domaine V (il s'agit donc de quantificateurs du *premier ordre*), alors que les variables quantifiées existentiellement R et B représentent des *relations*, dont l'arité est indiquée "à la Prolog" sous la forme *Relation* $\backslash$ *arité*. Déterminer si un graphe est biparti est donc un problème de *vérification de modèle*[2] : il s'agit en effet de déterminer si l'énoncé S est vrai dans le modèle G, ce qu'on écrira $G \models S$. L'approche utilisée ici consiste donc à identifier les problèmes à des *requêtes* (ou énoncés) exprimées dans un certain langage logique, et les structures de données correspondant à une instance à des *modèles finis*. Cette approche correspond à une branche de la logique calculatoire appelée *théorie des modèles finis* [152, 98, 153, 104]. De plus amples détails sur le langage particulier considéré dans notre exemple viendront en Section 9.2.

Le problème de vérification de modèles du type de formules considéré peut être résolu automatiquement de la manière suivante. En présence d'un graphe particulier G (possédant n sommets) dont nous voulons déterminer s'il est un modèle de S, il est facile d'*éliminer les quantificateurs* de la formule : définissant des vecteurs de variables booléennes pour chacun des faits $\mathrm{Arc}[x,y]$, $\mathrm{R}[x]$ et $\mathrm{B}[x]$ ($x, y \in 1 \ldots n$), le problème de vérification de modèle devient un problème classique de satisfaisabilité propositionnelle (dont nous donnons ici la forme normale conjonctive — ou CNF) :

$$\bigwedge_{x,y \in V} \left(\begin{array}{l} (\mathrm{R}[x] \vee \mathrm{B}[x]), (\neg\mathrm{R}[x] \vee \neg\mathrm{B}[x]), \\ (\neg\mathrm{Arc}[x,y] \vee \neg\mathrm{R}[x] \vee \neg\mathrm{R}[y]), \\ (\neg\mathrm{Arc}[x,y] \vee \neg\mathrm{B}[x] \vee \neg\mathrm{B}[y]) \end{array} \right) \tag{9.2}$$

En quelque sorte, les quantificateurs de premier ordre sont vus comme de simples *boucles* qu'il est possible de dérouler. La partie propositionnelle (ou *matrice*) de la formule est alors utilisée comme un

[1] Certaines améliorations peuvent rendre cette formule encore plus concise et lisible, au prix d'extensions syntaxiques du langage logique utilisé, par exemple de notations *fonctionnelles*. Le problème peut par exemple être exprimé sous la forme $\exists couleur : V \to \{r, b\} \;\; \forall x \, \forall y \; edge(x,y) \to couleur(x) \neq couleur(y)$. Pour préserver la minimalité du langage utilisé, nous éviterons toute extension de ce genre; celles-ci peuvent être considérées comme des macros faciles à ajouter au langage.

[2] Il s'agit bien du même sens que dans l'anglicisme *model-checking*, généralement réservé à la vérification de programmes. Dans ce cas particulier, les énoncés sont formulés dans une logique temporelle permettant d'exprimer les propriétés de systèmes de transition. En fait, de nombreux problèmes calculatoires sont, d'un point de vue logique, des problèmes de vérification de modèle, un autre exemple notable étant donné par les requêtes dans les bases de données [1].

motif dupliqué par le déroulement des boucles. Ce "motif" joue un rôle central. Regardant de plus près la formule générée, nous voyons que des règles simples de forme :

$$\mathrm{Arc}[x, y],\ \mathrm{R}[y]\ \rightarrow\ \neg\mathrm{R}[x]$$

peuvent être obtenues pour chaque paire $x, y \in 1 \ldots n$, en "orientant" chacune des clauses pour en faire des implications. En fait, il n'est pas absolument nécessaire d'expliciter ces règles car celles-ci peuvent également être simulées à la volée par la règle de *propagation unitaire*, qui est une version non orientée du type de raisonnement représenté par cette implication. Cette technique, qui est au cœur de la plupart des résolveurs SAT (voir, par exemple, [265]), infère naturellement le littéral $\mathrm{R}[x]$ si chacun des littéraux $\mathrm{Arc}[x, y]$ et $\mathrm{R}[y]$ est vrai.

D'autres types de déductions, un peu moins apparentes, peuvent être obtenues à partir de la formule (2), par exemple :

$$\mathrm{Arc}[x, y],\ \mathrm{R}[y]\ \rightarrow\ \mathrm{B}[x]$$

Cette règle est obtenue par simple *résolution* entre la clause $\neg\mathrm{Arc}[x, y] \vee \neg\mathrm{R}[x] \vee \neg\mathrm{R}[y]$ et la clause $\mathrm{R}[x] \vee \mathrm{B}[x]$. Il s'agit donc d'un *implicat* de l'ensemble de clauses originales. Là encore, il n'est toutefois pas nécessaire de produire explicitement la clause $(\neg Arc[x, y] \vee \neg R[y] \vee B[x])$ car sa conséquence $B[x]$ peut être obtenue par simple propagation unitaire à partir des règles $Arc[x, y], R[y] \rightarrow \neg R[x]$ et $\neg R[x] \rightarrow B[x]$. Notez que la résolution utilisée ici est un peu particulière puisqu'elle manipule des littéraux formés à partir d'*atomes* de type $B[x]$, et non de variables classiques; nous appelons cette technique *résolution de motifs* (cf. Section 9.4.3).

Il est clair que le raisonnement propositionnel sur la formule logique introduite précédemment présente de fortes similarités avec l'algorithme usuel de reconnaissance de graphes bipartis[3]. De même, il n'est pas difficile d'identifier dans d'autres algorithmes comme les parcours de graphe des mécanismes proches de ceux utilisés en déduction propositionnelle, notamment des mécanismes de propagation. Notre but dans ce chapitre est d'étudier les relations entre des formes de raisonnement propositionnel comme le *chaînage avant* ou la *propagation unitaire* et des algorithmes issus de domaines apparemment disjoints, en l'occurrence la théorie des graphes.

La technique utilisée se contente de considérer la partie *propositionnelle* des énoncés considérés (la forme des quantificateurs autorisés est d'ailleurs fixée), ce qui rend possible l'emploi de techniques de *compilation de connaissances* propositionnelles [47].

9.1.2 Plan du chapitre

Dans la prochaine section (9.2), nous rappelons un minimum de bases sur le langage logique considéré, SO$(\exists)$. Ensuite vient une brève présentation des techniques de *compilation de connaissances* dont nous avons besoin (Section 9.3). La confrontation entre SO$(\exists)$ et compilation de connaissances a lieu en section 9.4, où sont présentés successivement une utilisation naïve de la compilation de formules et un algorithme permettant de détecter la complétude de raisonnements propositionnels. Nous illustrons le cadre obtenu en section 9.5. Nous concluons alors en Section 9.6.

[3] Celui-ci fixe un sommet de départ à une couleur arbitraire, attribue la même couleur aux sommets situés à distance paire et l'autre couleur aux sommets à distance impaire, et utilise ce marquage pour détecter les éventuels cycles impairs.

9.2 La logique existentielle du second ordre

La logique existentielle du second ordre est le fragment logique dont nous considérons l'emploi pour l'expression de problèmes combinatoires. Nous rappelons brièvement la description formelle de ce langage (voir également le chapitre 3) et illustrons son utilisation pour la modélisation de problèmes.

9.2.1 Description formelle du langage

Définition 34. [Syntaxe de SO$(\exists)$]

- *Un* vocabulaire relationnel *est défini comme un ensemble de* symboles de prédicat *(écrits en majuscule)* $P_1, \dots P_m$*, chaque symbole ayant une* arité *fixée, précisée éventuellement en notation* Prolog *(cf.* $\exists R \backslash 1, \dots$*).*
- *Soient* $C = \{c_1, \dots, c_p\}$ *un ensemble fini de* symboles de constantes *et* $\mathcal{X} = \{x_1, \dots, x_n\}$ *un ensemble fini de* noms de variables*. Un* atome *est une expression de forme* $P(v_1, \dots, v_k)$*, où* P *est un symbole de prédicat d'arité* k *et chaque* v_i *est soit une constante, soit une variable.*
- *Une* formule du premier ordre *est soit un atome, soit la* conjonction *(*$\wedge$*, parfois simplement notée par une virgule) de deux formules de premier ordre, soit la* négation *(*$\neg$*) d'une formule, soit une* formule quantifiée universellement *de forme* $\forall x \Phi$ *où* Φ *est une formule et* x *une variable (*$\vee$*,* $\rightarrow$*,* $\exists$*, ... sont vus comme des abréviations).*
- *Une* formule existentielle de second-ordre *est une expression de forme* $\exists P_1, \dots, \exists P_m\ \Phi$*, dans laquelle chaque* P *est un symbole de prédicat et* Φ *est une formule de premier ordre.*

Une *instance* du problème spécifié par une formule SO$(\exists)$ est définie par un *domaine* fini $\mathbb{D}$ et une interprétation de chaque symbole de constante c par une valeur v du domaine et de chaque symbole de prédicat P par une relation R sur $\mathbb{D}$ (c.-à-d. une partie de $\mathbb{D}^k$, où k est l'arité de P). Une instance $I = \langle \mathbb{D}, v_1, \dots, v_p, R_1, \dots, R_m \rangle$ qui satisfait une requête Q est appelée *modèle*, ce qu'on note $I \models Q$.

Notez que nous n'avons pas jugé utile d'introduire de *symboles de fonctions* dans la syntaxe présentée pour SO$(\exists)$. Contrairement à d'autres types de logique calculatoire, telles Prolog, les structures de données ne sont pas représentées en SO$(\exists)$ par des termes, mais par des structures relationnelles finies qui correspondent à des *tableaux*, proches de la représentation physique des données dans la mémoire d'une machine. Le symbole d'égalité, invariablement interprété comme la relation $\{(x, x) \mid x \in \mathbb{D}\}$, est supposé présent dans l'ensemble de symboles de prédicat considérés.

9.2.2 Utiliser SO$(\exists)$ pour exprimer des problèmes combinatoires

Les structures relationnelles finies fournissent une représentation naturelle d'objets communs en informatique comme les mots, les graphes, les nombres, ou les tables d'une base de données. Par exemple, encoder des mots est possible grâce à l'utilisation d'une relation d'ordre total $\leq \backslash 2$ sur le domaine et d'un prédicat unaire indiquant si la valeur de chaque position est 0 ou 1. Les requêtes de premier ordre permettent d'exprimer des propriétés simples telles la symétrie, la clôture par transitivité, ou la triangulation (graphes C_4-libres) :

$$\begin{aligned} &\text{triangulé}(\text{Arc}) \equiv \\ &\quad \forall x\ \forall y\ \forall z\ \forall t \begin{pmatrix} \text{Arc}(x,y), \text{Arc}(y,z), \\ \text{Arc}(z,t),\ \text{Arc}(t,x) \end{pmatrix} \rightarrow \begin{pmatrix} \text{Arc}(x,z) \vee \\ \text{Arc}(y,t) \end{pmatrix} \end{aligned} \tag{9.3}$$

L'expressivité de la logique du premier ordre sur des structures finies est en fait équivalente à celle de l'algèbre relationnelle [1]. Le langage ne permet donc pas d'exprimer certaines requêtes, comme l'*atteignabilité,* qui est en revanche expressible en logique existentielle de second-ordre (pour des raisons techniques, la *non*-atteignabilité est plus simple à exprimer[4]) :

$$\begin{array}{l} \text{non\_atteignable}(\text{Arc}\backslash 2, s, t) \equiv \\ \quad \exists \text{Chemin}\backslash 2 \\ \quad\quad \neg \text{Chemin}(s,t), \\ \\ \quad\quad \forall x \forall y \quad \big(\ \text{Arc}(x,y) \rightarrow \text{Chemin}(x,y)\ \big), \\ \\ \quad\quad \forall x \forall y \forall z \quad \left(\begin{array}{r} \text{Chemin}(x,y), \text{Chemin}(y,z) \\ \rightarrow \text{Chemin}(x,z) \end{array} \right) \end{array} \tag{9.4}$$

L'expressivité de $\mathsf{SO}(\exists)$ dépasse en fait les problèmes polynomiaux, puisqu'il est très facile d'exprimer le problème d'isomorphisme de graphe qui (bien que vraisemblablement non **NP**-complet) n'est pas réputé être dans **P** :

$$\begin{array}{l} \text{isomorphes}\ (\text{Arc}\backslash 2, \text{Arc}'\backslash 2) \equiv \\ \quad \exists \text{Iso}\backslash 2 \\ \quad\quad \text{bijection}(Iso), \\ \quad\quad \forall x\ \forall y\ \forall x' \forall y' \quad \left(\begin{array}{r} \text{Iso}(x,x'), \text{Iso}(y,y') \rightarrow \\ (\text{Arc}(x,y) \leftrightarrow \text{Arc}'(x',y')) \end{array} \right) \end{array} \tag{9.5}$$

(*bijection* est une macro représentant une formule de premier ordre). L'utilisation de quantificateurs de premier ordre correspond à de simples boucles "`pour tout`", ce qui facilite l'utilisation de structures de données de type tableaux. En fait, $\mathsf{SO}(\exists)$ est un langage très proche de langages existants de programmation par contraintes, comme OPL [248] ou ALICE [172]. En particulier, cette logique présente la même séparation claire entre la partie *spécification* du problème et la parties *données* du problème. Cette caractéristique, qui se retrouve également dans les langages de modélisation mathématique, permet une meilleure réutilisabilité.

9.2.3 Expressivité

Il est clair que le problème de vérification de modèle ($I \models Q$) pour $\mathsf{SO}(\exists)$ est dans **NP** (voir l'algorithme de la Section 9.4). En fait, il est également vrai que tout problème **NP** sur une structure finie peut être exprimé comme le problème de vérification de modèle d'une certaine formule $\mathsf{SO}(\exists)$:

Théorème 3. *[105] L'ensemble des requêtes expressibles en $\mathsf{SO}(\exists)$ est exactement **NP**. Plus précisément, toute propriété vérifiable en temps non-déterministe polynomial peut être exprimée comme un énoncé $\mathsf{SO}(\exists)$ dont tous les quantificateurs de premier ordre sont* universels *et situés en tête de la formule de premier ordre (forme* prénexe*)*[5].

[4] Il serait naïf d'essayer d'exprimer l'atteignabilité par l'existence d'une relation transitive joignant s et t, car la relation $chemin(x,y)$ vraie pour tout couple (x,y) vérifie cette spécification.

[5] Dans notre soumission d'une présentation de ce chapitre à la conférence JFPLC, nous affirmions que l'emploi de techniques de *skolémisation* permet de remplacer les quantificateurs existentiels de premier ordre par des quantificateurs universels, au prix de l'introduction de relations par quantification existentielle de second-ordre. D'après un relecteur, que nous remercions pour

Voir par exemple [105], ou [153] page 116 pour une preuve[6]. Dans la suite de ce chapitre, nous nous restreignons donc aux requêtes SO($\exists$) construites sur une formule de premier ordre universellement quantifiée prénexe, et dont la matrice (partie propositionnelle de la formule) est mise sous forme CNF (la mise sous forme CNF, classique en SAT, étant facile à obtenir par introduction de variables existentielles et décomposition en contraintes ternaires).

9.3 Raisonnement propositionnel et compilation de connaissances

Nous mettons provisoirement de côté les quantificateurs de premier et second ordre et nous nous concentrons sur des techniques de raisonnement propositionnel. Nous effectuons quelques rappels des points utiles déjà présentés dans le chapitre 4, auquel nous renvoyons plus de détails sur la résolution (en particulier la preuve de la propriété de complétude utilisée pour la compilation de connaissances) et la propagation unitaire.

9.3.1 Raisonnement propositionnel

Rappelons qu'un *littéral* est soit une variable soit sa négation. Les littéraux représentent des faits comme "le sommet x est (ou n'est *pas*) bleu". Une *clause* est une disjonction de littéraux; une formule CNF est une conjonction de clauses, par exemple $(\neg\mathrm{Arc}[x,y]\vee\neg\mathrm{R}[x]\vee\neg\mathrm{R}[y]), (\mathrm{R}[x]\vee\mathrm{B}[x])$. La question que nous nous posons ici est de calculer l'ensemble des conséquences d'un ensemble de littéraux L et d'une base de connaissances B, c.-à-d. l'ensemble de littéraux $\{c \mid \models F \wedge KB \rightarrow c\}$), ou de détecter l'inconsistance. Par exemple, partant de la base mentionnée précédemment, la conséquence $\mathrm{B}[x]$ peut être déduite des faits $\mathrm{Arc}[x,y]$ et $\mathrm{R}[y]$. Un algorithme simple et efficace pour ce type de déductions est la *propagation unitaire*, qui est basée sur la règle suivante :

Règle 1. ***[Propagation unitaire]*** *Si tous les littéraux d'une clause sont faux excepté un, alors ce dernier littéral peut être inféré.*

La propagation unitaire permet d'inférer des littéraux de manière non-déterministe jusqu'à ce que la règle 1 ne puisse plus s'appliquer. Il est clair, par exemple, que la propagation unitaire permet de déduire $\neg\mathrm{R}[x]$ à partir des littéraux $\mathrm{Arc}[x,y]$ et $\mathrm{R}[y]$ grâce à la clause $\neg\mathrm{Arc}[x,y] \vee \neg R[x] \vee \neg\mathrm{R}[y]$. Il est alors possible d'inférer $\mathrm{B}[x]$ grâce à la clause $\mathrm{R}[x] \vee \mathrm{B}[x]$. La propagation unitaire est similaire à l'arc-consistance, [186] c'est également une généralisation du *chaînage avant* dans laquelle l'orientation des implications n'a pas besoin d'être précisée[7]. La propagation unitaire peut être mise en œuvre en temps linéaire par rapport à la longueur des clauses considérées [265]. Cependant, il ne s'agit pas d'une méthode de raisonnement *complète* :

sa vigilance, cette affirmation est fausse en général, et les techniques applicables ne relèvent pas de la skolémisation [100]. De plus, bien que le théorème de FAGIN soit vrai sur les structures non ordonnées, il semble la forme particulière indiquée ici repose au contraire sur l'existence d'une relation d'ordre sur la structure considérée — une exigence par ailleurs souvent implicite en complexité descriptive.

Par ailleurs, il est en général nécessaire (et en particulier pour les langages exprimant la classe **P**) de disposer d'une *relation d'ordre* sur le domaine d'interprétation.

[6] Voir également [152] pour une introduction *grand public* aux résultats de complexité descriptive, et [153] pour un traité complet sur le sujet.

[7] Remarquons que la propagation unitaire est un algorithme optimal pour Horn-SAT (ainsi que antiHorn-SAT 2-SAT) et qu'il n'y a donc pas d'intérêt pratique à orienter les règles.

Exemple 9.12. *[47] Considérons la base $(\neg l \vee a \vee c), (l \vee b \vee c)$. Si a et b sont faux, alors c est vrai. Cependant, ceci ne peut être déduit par propagation unitaire.*

9.3.2 Compilation de connaissances

Une méthode permettant de s'affranchir de l'incomplétude de la propagation unitaire est de compiler une base de connaissances B en une base B' logiquement équivalente sur laquelle un certain nombre de règles de déduction implicites ont été rendues explicites, rendant ainsi la propagation unitaire complète. Cette approche a été proposée par [88] et (pour une variante basée sur le chaînage avant) par [193]. Elle repose sur les notions d'*implicat* et de *résolution*. Un implicat est une clause qui est une conséquence logique de la base. On qualifie de *premier* un implicat minimal. La résolution est la méthode de calcul d'implicats suivante (d'autres algorithmes de calcul d'implicats peuvent être trouvés, par exemple, dans [192, 237]):

Règle 2. ***[Résolution]*** *[225, 84] Si une CNF contient deux clauses $A \vee x$ and $\neg x \vee B$ (x étant une variable), la clause $A \vee B$ est un implicat appelé* résolvant.

Il est bien connu que la propagation unitaire, mais aussi les algorithmes de branchement, sont des cas particuliers de résolution. La résolution est une méthode de déduction complète mais pour laquelle des instances requérant un temps de calcul exponentiel ont été identifiées [137]. Nous dirons qu'une base de connaissances est *saturée* lorsqu'elle est close par ajout d'implicats premiers. La proposition suivante est à la base du type de compilation de connaissances discuté dans la suite :

Proposition 9. *[193, 88] La propagation unitaire sur une base CNF saturée est complète.*

Démonstration. Supposons qu'un littéral c soit une conséquence d'un ensemble de littéraux L pour une certaine base B, c.-à-d. $\{c \mid \models L \wedge B \rightarrow c\}$. L étant une conjonction, sa négation est une disjonction qui peut être écrite comme une clause $F^{\neg}$. Ayant $(L \wedge \neg c) \wedge B \rightarrow \bot$, la clause $(L^{\neg} \vee c)$ est un implicat de B. B étant supposée saturée, une de ses clauses contient un sous-ensemble des littéraux de $(L^{\neg} \vee c)$. Soit cette clause contient directement le littéral c, qui peut donc être déduit par propagation unitaire sur la clause dès lors que les autres littéraux de l'implicat sont faux, soit elle ne contient pas c et on peut alors inférer *faux*. □

Exemple 9.1. *(suite) À partir de la base $B = (\neg l \vee a \vee c), (l \vee b \vee c)$, on peut déduire le résolvant $R = (a \vee b \vee c)$, et la base $B \wedge R$ est saturée. La propagation permet alors de déduire c à partir des littéraux $\neg a$ et $\neg b$.*

La résolution permet de produire l'ensemble des implicats d'une formule CNF (il s'agit là d'une propriété de *complétude* plus forte que la complétude réfutationnelle généralement démontrée dans les traités); notez toutefois que le nombre d'implicats d'un SAT peut être exponentiel, [53] et que la compilation de connaissances peut donc faire exploser la taille de la base. En pratique, l'approche consistant à produire *tous* les implicats premiers est un peu naïve: on peut éviter une création d'implicat si tous les littéraux de celui-ci peuvent déjà être inférés à partir de la base courante [88].

9.4 Compilation de la matrice d'une formule SO$(\exists)$

Nous décrivons un mécanisme d'exécution des spécifications SO$(\exists)$ basé sur la production d'un *store de contraintes* dont la satisfaisabilité permet de déterminer si la structure considérée est un modèle[8]. Ce store de contraintes est obtenu par simple déroulement des quantificateurs de premier ordre. Un ensemble d'observations sur le rapport entre la *matrice* (c.-à-d. la partie non quantifiée) de la formule et la difficulté de résolution du problème nous permet alors de présenter des applications des techniques de compilation.

9.4.1 Génération d'un store à partir d'une requête SO$(\exists)$

Étant donnée une requête Q et une instance I, il est facile d'éliminer les quantificateurs (universels) de premier ordre : comme suggéré en exemple d'introduction, les quantificateurs universels peuvent être vus comme des simples boucles, dont le rôle est de dupliquer la matrice du problème. La transformation opérée est donc la suivante :

$$\forall x\ \Phi \ \leadsto \ \bigwedge_{v \in \mathbb{D}} \Phi[x \leftarrow v]$$

Notez qu'une fois le domaine $\mathbb{D}$ fixé pour une certaine structure, une relation $R\backslash k$ est naturellement vue comme un tableau à k dimensions à valeurs booléennes, c.-à-d. comme un ensemble de $|\mathbb{D}|^k$ variables logiques.

Dans cette approche, la matrice (partie propositionnelle) de la formule est donc vue comme un *motif* dupliqué pour chaque combinaison de valeurs prises dans le domaine. Par exemple, considérons la requête :

$$Q \ : \ \exists P\backslash 1\ \forall x \ \left(\ \neg \mathrm{Arc}(1,x) \vee \neg P(1) \vee P(x) \ \right)$$

vérifier le modèle correspondant à un graphe a deux sommets ($\mathbb{D} = \{1,2\}$) possédant une unique arête (non orientée) ($\mathrm{Arc} = \{(1,2)\}$) est équivalent à résoudre le store de contraintes booléennes :

$$(\neg \mathrm{Arc}[1,1] \vee \neg \mathrm{P}[1] \vee \mathrm{P}[1]), (\neg \mathrm{Arc}[1,2] \vee \neg \mathrm{P}[1] \vee \mathrm{P}[2])$$

Le tableau Arc est donné par le modèle, le tableau P correspond à la relation unaire sur le domaine $\{1,2\}$. Il est facile de se convaincre qu'avec la technique suggérée (qui correspond assez fidèlement au mécanisme d'exécution d'OPL, D'ALICE, ou de langages équivalents), le problème initial est vrai si et seulement si le store produit est satisfaisable.

[8] On retrouve ici le mécanisme d'exécution du paradigme CLP [247, 154], dans lequel l'exécution d'un programme consiste en la construction incrémentale d'un "store" qui est traité par un résolveur sur un domaine de calcul particulier (ici les booléens), l'approche est illustrée par l'algorithme générique de résolution de problème de [247] :

```
solveProblem(...) ←
   generateVariables(...),
   stateConstraints(...),
   stateSurrogateConstraints(...),
   makeChoices(...). % résolution
```

L'approche a également été reconsidérée récemment dans [48] et, en un certain sens, elle a été inaugurée par le système ALICE [172].

9.4.2 Compilation d'énoncés SO$(\exists)$

Il est clair que les techniques de compilation de connaissances permettent d'améliorer la forme des requêtes SO$(\exists)$: enrichir la matrice en rendant explicites certains de ses implicats permet d'augmenter l'efficacité des méthodes de propagation. Comme le motif ainsi enrichi est dupliqué un grand nombre de fois durant la résolution du problème, chaque clause ajoutée à la matrice ajoutera un grand nombre de redondances dans le store de contraintes produit, et il est bien connu que la production de redondances facilite la résolution de contraintes.

Comme suggéré en exemple d'introduction, la requête correspondant au problème de bipartition peut par exemple être "compilée" par ajout d'une clause (4) obtenue par résolution :

$$\exists \mathrm{R}\backslash 1 \;\; \exists \mathrm{B}\backslash 1 \;\; \forall x \, \forall y \;\; \left(\begin{array}{ll} 1 & (\mathrm{R}[x] \vee \mathrm{B}[x]), (\neg \mathrm{R}[x] \vee \neg \mathrm{B}[x]), \\ 2 & (\neg \mathrm{Arc}[x,y] \vee \neg \mathrm{R}[x] \vee \neg \mathrm{R}[y]), \\ 3 & (\neg \mathrm{Arc}[x,y] \vee \neg \mathrm{B}[x] \vee \neg \mathrm{B}[y]) \\ 4 & (\neg \mathrm{Arc}[x,y] \vee \mathrm{R}[x] \vee \neg \mathrm{B}[y]), \ldots \end{array} \right) \tag{9.6}$$

Notez que la "résolution" utilisée ici opère sur des *faits* et non sur des variables propositionnelles; en fait il suffit de remplacer chaque occurrence *syntaxiquement égale* d'un fait par une même variable et d'utiliser la résolution classique (dans notre exemple, nous avons effectué une résolution sur la "variable" $\mathrm{R}[x]$, celle-ci étant différente de la "variable" $\mathrm{R}[y]$).

Si ce type de compilation permet d'espérer améliorer la forme des requêtes de manière automatique, le plus intéressant est en fait de comprendre pourquoi cette technique ne fonctionne pas toujours, c.-à-d. pourquoi elle ne permet pas systématiquement de rendre la propagation unitaire complète. Considérons pour ce faire une requête correspondant au problème de 3-coloration, obtenue par une légère modification de la requête pour la bipartition (formule 9.1). La matrice du problème, qui utilise une troisième couleur G (pour *vert*), est la suivante :

$$\begin{array}{ll} 1 & (\mathrm{R}[x] \vee \mathrm{B}[x] \vee \mathrm{G}[x]), \\ 2 & (\neg \mathrm{R}[x] \vee \neg \mathrm{B}[x]), (\neg \mathrm{B}[x] \vee \neg \mathrm{G}[x]), (\neg \mathrm{G}[x] \vee \neg \mathrm{R}[x]), \\ 3 & (\neg \mathrm{Arc}[x,y] \vee \neg \mathrm{R}[x] \vee \neg \mathrm{R}[y]), \ldots \end{array}$$

La troisième ligne est répétée pour les couleurs G et B. Nous pouvons inférer deux sortes de résolvants pour ce motif. Ceux issus de la clause (1) et d'une clause de la ligne (2) sont de type $(\mathrm{G}[x] \vee \mathrm{B}[x] \vee \neg \mathrm{B}[x])$ et peuvent ne pas être conservés car ils contiennent une variable et sa négation et sont donc toujours vérifiés. Ceux issus de la clause (1) avec la clause (3) ou son équivalent pour les couleurs G et B sont de forme $(\neg \mathrm{Arc}[x,y] \vee \neg \mathrm{B}[x] \vee \mathrm{R}[x] \vee \mathrm{R}[y])$ et ne sont pas non plus nécessaires car chacun de leurs littéraux peut déjà être obtenu par d'autres clauses à partir de la négation des trois autres littéraux. Le motif est donc saturé.

Pourtant, si nous considérons les 222 clauses générées par cette formule sur un graphe à 6 sommets, dont un aperçu est donné en Figure 9.1, on découvre que celles-ci présentent l'implicat $(\neg \mathrm{Arc}[2,5] \vee \neg \mathrm{Arc}[4,5] \vee \neg \mathrm{R}[2] \vee \neg \mathrm{G}[4] \vee \mathrm{B}[5])$[9], dont la conclusion ne peut être obtenue par propagation unitaire. Intuitivement, il était impossible sur cet exemple de déduire par un raisonnement purement local que si les deux voisins d'un sommet x prennent des couleurs distinctes, x prend la 3ème couleur. Le fait qu'une matrice soit saturée ne permet donc pas en général de garantir que le store généré le sera également.

[9] Résoudre sur 1 et 3 (on obtient 5), sur 2 et 3 (6), sur 4 et 5 (7) et finalement sur 6 et 7.

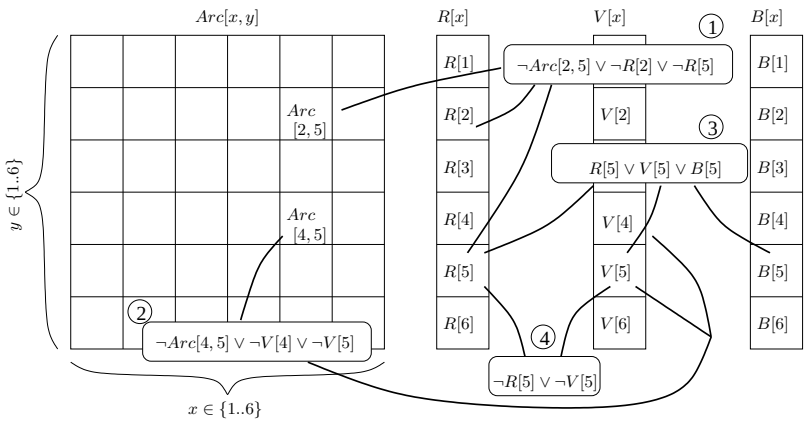


Figure 9.1 – Un aperçu du store de contraintes.

9.4.3 Détection de conflits

L'exemple précédent met en avant le fait que des résolvants peuvent apparaître lors de la *duplication* de la matrice. Par exemple, les clauses générées par les motifs $\neg\text{Arc}[x, y] \vee \neg\text{R}[x] \vee \neg\text{R}[y]$ et $\text{R}[x] \vee \text{G}[x] \vee \text{B}[x]$ produisent un résolvant lorsque les motifs sont instanciés, par $(x = 2, y = 5)$ et $(x = 5)$, respectivement.

Nous appelons *renommage* une fonction partielle de l'ensemble $\mathcal{X}$ des variables vers lui-même; par exemple $r = \{x \to z, y \to z\}$. Un renommage s'applique à un atome (resp. une clause) en l'appliquant à chacune de ses variables (resp. chacun de ses atomes). Par exemple, soit C la clause $\text{Arc}[x, y] \vee \neg\text{R}[y]$, on a $r(C) = (\text{Arc}[z, z] \vee \neg\text{R}[z])$. Deux clauses S_1 et S_2 sont dites *égales à un renommage près* s'il existe un renommage r tel que $r(C_1) = C_2$ (il s'agit ici d'une égalité ensembliste, et l'ordre des littéraux n'est donc pas significatif). La résolution peut être étendue aux motifs de la façon suivante :

Règle 3. ***[Résolution de motifs]*** *Si la matrice considérée comporte deux clauses $A \vee S_1$ et $\neg S_2 \vee B$ et si S_1 et S_2 sont égaux à un renommage près, inférer la clause $r(A) \vee B$.*

Des exemples d'application de cette règle sont donnés en section suivante. La résolution de motifs permet de générer les modèles de tous les résolvants pouvant apparaître dans le store de contraintes :

Proposition 10. *Tout résolvant pouvant apparaître sur le store généré par une requête sur un modèle particulier est une instance d'un motif produit par* résolution de motifs*.*

Démonstration. (esquissée) Soient deux clauses $(A \vee x)$ et $(\neg x \vee B)$, dérivées de deux motifs P_1 et P_2. Le résolvant $A \vee B$ est une instance d'un des résolvants des deux motifs. □

La conséquence de la proposition 10 est la suivante : si la résolution de motifs termine sans exhiber de *nouveau* motif déjà présent dans la matrice du problème, alors la compilation a réussi, et le store de contraintes généré sera saturé pour tout modèle candidat. Le problème peut alors être résolu par simple propagation unitaire.

9.5 Quelques exemples

Nous illustrons finalement notre approche sur quelques exemples de formules modélisant des problèmes classiques de théorie des graphes. Pour la bipartition, tous les résolvants proviennent du motif $(\mathrm{R}[x] \vee \mathrm{B}[x])$, contenant les seuls littéraux positifs. Par renommage, $\mathrm{R}[y]$ peut être résolu aussi bien avec $\neg\mathrm{R}[x]$ que $\neg\mathrm{R}[y]$. On obtient finalement :

$$\begin{array}{llll} 1 & \mathrm{R}[x] \vee \neg\mathrm{R}[x] & 2 & \mathrm{B}[x] \vee \neg\mathrm{B}[x] \\ 3 & \neg\mathrm{Arc}[x,y] \vee \neg\mathrm{R}[x] \vee \mathrm{B}[y] & 4 & \neg\mathrm{Arc}[x,y] \vee \mathrm{B}[x] \vee \neg\mathrm{R}[y] \\ 5 & \neg\mathrm{Arc}[x,y] \vee \neg\mathrm{B}[x] \vee \mathrm{R}[y] & 6 & \neg\mathrm{Arc}[x,y] \vee \mathrm{R}[x] \vee \neg\mathrm{B}[y] \end{array}$$

Les motifs de type (1) et (2) sont tautologiques et peuvent être écartés; les autres produisent des clauses dont chaque littéral peut être retrouvé par propagation unitaire sur les clauses originales du problème. Il est donc impossible, en répétant les motifs de la matrice de bipartition, d'obtenir de nouveaux résolvants; la base générée sera saturée quelque soit le modèle, ce qui prouve la complétude de la propagation unitaire pour ce problème.

Considérant le problème de non-atteignabilité, défini par la requête 9.4, on obtient des variantes de motifs comme $\neg\mathrm{Arc}(s,t)$, ou encore $\neg\mathrm{Chemin}[x,y] \vee \neg\mathrm{Arc}[y,z] \vee \mathrm{Chemin}[x,z]$, dont chacun des littéraux peuvent être obtenus par propagation, ce qui prouve également la complétude de la propagation pour ce problème.

Certains problèmes moins classiques se prêtent également au même type de raisonnement; par exemple, le problème de reconnaissance des graphes de comparabilité, exprimé comme suit :

$$\begin{array}{l} \mathrm{graphe\_de\_comparabilité}(\mathrm{Arc}) \equiv \\ \quad \exists \mathrm{Dir} \backslash 2 \\ \qquad \forall x\, \forall y\, \forall z \left(\begin{array}{l} \neg\mathrm{Dir}(x,x), \\ \neg\mathrm{Dir}(x,y) \vee \mathrm{Arc}(x,y), \\ \neg\mathrm{Dir}(x,y) \vee \neg\mathrm{Dir}(y,x), \\ \neg\mathrm{Dir}(x,y) \vee \neg\mathrm{Dir}(y,z) \vee \mathrm{Dir}(x,z) \end{array} \right) \end{array}$$

La résolution de motifs produit un nombre fini de clauses pouvant, là encore, être écartées (par exemple $\mathrm{Dir}(x,y) \vee \neg\mathrm{Dir}(y,z) \vee \mathrm{Arc}(x,z)$, ou encore $\mathrm{Dir}(x,y) \vee \neg\mathrm{Dir}(y,z) \vee \neg\mathrm{Dir}(z,x)$). Nous ne pouvons détailler les résolvants obtenus pour deux autres exemples intéressants, celui de *couplage parfait* et celui de *reconnaissance des graphes d'intervalles*, dont nous suggérons une formulation, et laissons le lecteur vérifier qu'il est également saturé :

$$\begin{array}{l} \mathrm{graphe\_d\_intervalles}(\mathrm{Arc}) \equiv \\ \quad \exists \mathrm{Comp} \left(\begin{array}{l} \mathrm{graphe\_de\_comparabilité}(\mathrm{Comp}), \\ \forall x,y,z,t \left(\begin{array}{l} \mathrm{sans\_cycle}(\mathrm{Arc},x,y,z,t) \\ \mathrm{Comp}(x,y) \leftrightarrow \neg\mathrm{Arc}(x,y) \end{array} \right) \end{array} \right) \end{array}$$

9.6 Conclusion et discussion

La logique existentielle du second ordre est un langage expressif et naturel permettant d'exprimer les problèmes combinatoires sur les structures finies. Son expressivité est clairement déterminée, et elle est proche de langages de programmation par contraintes existants [248, 172]. Déterminer si une instance satisfait une spécification dans ce langage revient à générer un store de contraintes à partir d'un

motif de contraintes. Nous avons proposé une application d'une forme de *compilation de connaissances* à base de résolution sur des énoncés $\mathsf{SO}(\exists)$. Basé sur le fait bien connu que la clôture par génération d'implicats rend la propagation unitaire complète dans le cas propositionnel, nous avons proposé un test basé sur l'énumération de tous les motifs de résolvants pouvant apparaître dans le store. Cette technique peut, dans des cas simples dont nous ne sommes pas encore parvenus à donner de caractérisation précise, garantir qu'aucun résolvant ne sera généré par duplication de motifs, garantissant ainsi la complétude de la propagation unitaire sur le store généré. Notons que notre algorithme d'identification de classes polynomiales de formules $\mathsf{SO}(\exists)$ doit être considéré comme complémentaire d'autres approches existantes, notamment celles basées sur l'analyse des préfixes [134].

Affiner des énoncés de logique à modèle finis (et particulièrement des énoncés $\mathsf{SO}(\exists)$) en les traduisant vers des formes plus propices à une exécution efficace est une perspective déjà suggérée dans [153] (§15.1.2). Bien entendu, notre approche est loin de représenter une avancée définitive vers le but fixé par Immerman, et la pertinence de cette approche pour des problèmes plus pratiques (en particulier son application aux CSP) reste à valider. Cependant, une approche de $\mathsf{SO}(\exists)$ basée sur la compilation de connaissances est intéressante pour plusieurs raisons. Elle montre que les techniques de compilation de connaissances, jusqu'alors réservées à la logique propositionnelle, peuvent également servir à l'analyse de formules logiques. Par ailleurs, l'utilisation de motifs de contraintes pour générer un store est typique de la programmation par contraintes; l'emploi de méthodes à base de résolution permettant d'analyser les conflits pouvant survenir lors de la création du store est donc potentiellement intéressante pour ce type de programmation. Enfin, une motivation sous-jacente à notre travail est de préciser l'analogie entre certaines méthodes de raisonnement propositionnel et d'autres types d'algorithmes classiques. Ganzinger et Mc Allester [117] ont mis en évidence l'intérêt de "modèles algorithmiques de programmation logique" pour ré-exprimer certains algorithmes standards; il nous semble que $\mathsf{SO}(\exists)$ ouvre des perspectives intéressantes par rapport à cette problématique.

PARTIE V

Conclusion

CHAPITRE 10

Conclusion générale

Every student should write a bad PhD thesis, so that he won't spend the rest of his life working on his thesis.

— Alan J. PERLIS, cité dans [236].

10.1 Contributions principales

Dans le cadre de la résolution de problèmes combinatoires exprimés dans des langages issus de la logique, et en particulier de la logique propositionnelle ou sur des domaines finis, le sujet central de cette thèse a été la prise en compte des *quantificateurs* (universels et existentiels), notion généralement absente des outils de programmation par contraintes. Nous avons motivé cette extension en montrant qu'elle permet d'étendre l'expressivité des langages de contraintes. Elle les rend ainsi aptes à la modélisation et la résolution de problèmes pour lesquels peu de méthodes sont actuellement disponibles (notamment les problèmes de décision en présence d'incertitude, dans lesquels une forme de quantification émerge naturellement), ou pour lesquels les outils disponibles manquent de généricité (des algorithmes, souvent similaires, sont développés, par exemple, pour le *model-checking* ou le calcul de plans d'action, mais il n'existe pas d'outil permettant de résoudre l'ensemble de la classe d'applications représentée par ces problèmes, et qui possède la souplesse d'utilisation d'un résolveur de contraintes).

Du point de vue technique, la principale contribution de cette thèse a été de proposer une méthode de propagation de contraintes basée sur une notion d'arc-consistance quantifiée. Cette généralisation de la technique classique de résolution de CSP n'est pas triviale, car la quantification empêche une caractérisation simple des valeurs inutiles des domaines des variables. Partant d'un certain nombre d'observations sur les arbres de résolution de QCSP, nous avons défini une notion de *consistance* caractérisant les valeurs pouvant être supprimées. Nous avons montré que la consistance peut être approximée par un raisonnement local appliqué à chacune des contraintes du problème. La consistance locale obtenue est bien une généralisation de l'arc-consistance classique, qui correspond au cas particulier de quantificateurs existentiels. Nous avons de plus montré qu'il est possible de dériver de cette notion générale d'arc-consistance des règles de propagation sur les booléens ou sur les intervalles. On peut ainsi reformuler pour les contraintes quantifiées l'essentiel du cadre usuel pour les CSP, et exhiber une notion d'opérateur contractant, caractériser le filtrage atteint en termes de point fixe de ces opérateurs, démontrer les propriétés de confluence des algorithmes, etc.

La méthode a été expérimentée et testée sur des exemples académiques de petite taille. Si les performances sont loin d'être pleinement satisfaisantes, des accélérations importantes sont cependant observées par rapport à une résolution naïve, ce qui permet de valider l'approche. Une meilleure propagation nécessite une gestion fine de la décomposition d'expressions en contraintes primitives, ce qui suppose un traitement symbolique non encore expérimenté. Par ailleurs, d'autres techniques dérivées de l'arc-consistance peuvent être envisagées et faire l'objet de recherches ultérieures. Les consistances fortes, qui

s'avèrent parfois supérieures à l'arc-consistance pour les problèmes difficiles, présentent en particulier un intérêt potentiel pour la classe particulièrement ardue que représentent les problèmes quantifiées.

Nous ne prétendons pas avoir réussi à aborder tous les aspects du problème de contraintes quantifiées, ni à avoir exploré leurs applications de manière exhaustive. Nous nous sommes en particulier contentés d'effleurer les problèmes liés au *planning* et au *model-checking*, deux domaines de recherche suffisamment vastes pour avoir suscité des ouvrages entiers et être les thèmes principaux de plusieurs conférences internationales annuelles. Il serait bien entendu illusoire de penser que les techniques de résolution de contraintes quantifiées proposées dans cette thèse s'imposeront *en remplacement* de techniques de résolution existantes de problèmes **PSPACE**-complets ; il est clair que toute nouvelle proposition de technique de résolution de problèmes s'inscrit désormais dans un cadre de coopération et d'intégration aux techniques existantes. Nous espérons donc que les techniques de propagation de contraintes quantifiées trouveront leur place parmi les techniques utiles pour ce type d'applications pratiques, et qu'elles permettront ainsi de contribuer à l'application de ces techniques. Un autre domaine d'application particulièrement récent et prometteur, dont nous n'avons fait que suggérer brièvement les enjeux, concerne la résolution de problèmes en présence de données incertaines, et en particulier le cadre récent des problèmes de contraintes stochastiques, apparu durant la fin de cette thèse [258, 188].

10.2 Contributions mineures

Au cours de cette thèse, nous avons également tenté d'exprimer nos points de vue sur la problématique de recherche dans laquelle la thèse s'inscrit et de faire transparaître de manière diffuse dans le manuscrit des contributions mineures. Je tente ici d'en résumer l'essentiel.

- Nous avons proposé un cadre dans lequel les différents aspects de la programmation par contraintes apparaissent comme des variantes de problèmes logiques. En particulier, certains langages de programmation par contraintes peuvent être vus comme des variantes syntaxiques de la logique existentielle de second ordre; l'exécution de ces programmes est un problème de *vérification de modèles*.
  Si le fait de réduire les langages utilisés en pratique au noyau logique les constituant peut sembler une simple curiosité intellectuelle, cette approche permet également de clarifier les notions d'*expressivité* des langages de contraintes. Il est surprenant de constater que la question a été très peu étudiée dans la communauté contraintes (un article de Marco CADOLI [46] et une étude par FAGES *et al.* [103] de l'expressivité des langages concurrents faisant exception) — on peut comparer cette situation à la communauté des bases de données relationnelles, dans laquelle l'expressivité des différents langages de requête est en général parfaitement déterminée [1].
  Comme nous l'avons souligné dans cette thèse, les résultats de *complexité descriptive* permettent de caractériser précisément l'expressivité des langages comme OPL et Alice : ceux-ci permettent d'exprimer tous les problèmes **NP**. Cette conclusion théorique me parait bienvenue car elle confirme la pratique de la programmation par contraintes et de la programmation linéaire — les spécialistes "savent bien" qu'il est toujours plus ou moins possible d'exprimer un problème combinatoire classique avec ces outils de programmation.

- Nous avons tenté de regrouper et ré-exprimer, dans un contexte *programmation par contraintes*, les résultats de manipulation des formules quantifiées qui peuvent servir de base aux outils de résolution de contraintes quantifiées. Ces aspects sont en général connus de la communauté *démonstration automatique*. Il nous semble cependant, si l'étude de la résolution de contraintes quan-

tifiées se développe comme nous le pensons, qu'un travail de reformulation de la vaste littérature logique existante en fonction des particularités du domaine en émergence est nécessaire. Nous espérons que ce travail de reformulation permettra aux lecteurs de ce document non familiers des problèmes quantifiés de comprendre rapidement les particularités induites par l'introduction de quantificateurs dans les problèmes de contraintes.

- Nous avons proposé une vision relationnelle des contraintes quantifiées. Une telle vision n'est pas totalement neuve, puisqu'elle est liée à des résultats de complexité sur les bases de données [252]. Le lien pratique entre algèbre relationnelle et résolution de contraintes quantifiées n'avait cependant, à notre connaissance, pas été formulé dans la littérature. Une telle vision peut être riche d'enseignements et des algorithmes de résolution peuvent émerger de la confrontation entre Bases de Données et Programmation par Contraintes, il nous parait donc important de documenter cette approche.

- Nous avons suggéré les limites des formules en forme prénexe pour l'étude de certains problèmes. Nous avons exhibé des formules en forme non prénexe pour lesquelles l'évaluation naïve donne des performances polynomiales que ne garantissent pas les méthodes d'énumération. Ce point soulève des questions intéressantes ; en particulier, la mise systématique sous forme prénexe imposée par l'ensemble des outils existants parait, en théorie du moins, peu légitime. Plus généralement, l'étude de l'ordre des quantificateurs et des dépendances entre les variables liées est un problème central qui devra, dans l'avenir, faire l'objet de travaux approfondis (une contribution récente à des problématiques assez proches semble d'ailleurs être [180]).

Les perspectives issues du cadre QCSP et des techniques de propagation quantifiée sont nombreuses, et la marge d'amélioration des techniques proposées est plus importante encore que le travail déjà accompli. Un bon nombre de pistes de recherche possibles émergent directement des points suggérés précédemment et des points non entièrement résolus dans cette thèse. J'aimerais de plus mentionner les perspectives suivantes :

- L'identification de contraintes globales a permis d'intégrer des algorithmes spécialisés améliorant la résolution de contraintes issues de certaines classes d'application ; les contraintes globales ont eu un rôle important dans le développement de la programmation par contraintes, qu'elles ont ont rendues matures pour les applications industrielles. Il est donc légitime de chercher, de même, à identifier des algorithmes spécialisés permettant de traiter certains aspects de classes de problèmes données, et d'ajouter ainsi des contraintes globales au cadre QCSP ;
- La technique de skolémisation suggère une alternative à la représentation des quantificateurs alternés : les variables peuvent être vues comme des fonctions des variables quantifiées de rang inférieur dans le préfixe de la formule. Il est possible que le fait d'expliciter ces fonctions permette certaines clarifications du cadre : ce sont en fait des fonctions qui sont réduites par les techniques de consistances locales. Des liens avec le cadre CLP(F), proposé au début des années 1990 [146], doivent ainsi être explorés ;
- Enfin, pour les applications dans lesquelles un motif particulier de quantificateur apparaît, il est possible d'envisager des améliorations *ad-hoc* des techniques de propagation. Nous avons vu que de tels motifs correspondent à des classes particulières et restreintes de problèmes, mais ceux-ci possèdent néanmoins des applications remarquables, qui justifient leur étude approfondie.

Bibliographie

[1] S. ABITEBOUL, R. HULL et V. VIANU. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. AGARWAL, N. SAXENA et N. KAYAL. PRIMES is in P. Rapport technique, Indian Institute of Technology of Kanpur, 2002.

[3] H. AÏT-KACI. *Warren's Abstract Machine – a tutorial reconstruction*. MIT Press, 1999. `http://www.vanx.org/archive/wam/wam.html`,

[4] B. APSWALL, M. F. PLASS et R. E. TARJAN. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[5] K. R. APT. Ten years of Hoare's logic: A survey - part 1. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[6] K. R. APT. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.

[7] K. R. APT et E. MONFROY. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming*, 1 — special Issue in honor of Alain Colmerauer,(6):713–750, 2001.

[8] D. S. ARNON, G. E. COLLINS et S. MAC CALLUM. Cylindrical Algebraic Decomposition I: The basic algorithm. *SIAM J. of Computing*, 13(4):865–877, 1984.

[9] A. AYARI et D. A. BASIN. QUBOS: Deciding quantified boolean logic using propositional satisfiability solvers. In *Proc. of the 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 187–201, Portland, Oregon, 2002. Springer.

[10] F. BACCHUS, X. CHEN, P. VAN BEEK et T. WALSH. Binary vs. non-binary constraints. *Artificial Intelligence*, 140(1-2):1–37, 2002.

[11] L. BACHMAIR et H. GANZINGER. Resolution theorem proving. In J. A. ROBINSON et A. VORONKOV, réds., *Handbook of Automated Reasoning*, chapter 2, pages 19–99. Elsevier, 2001.

[12] J. L. BALCÁZAR. The complexity of searching implicit graphs. *Artificial Intelligence*, 86(1):171–188, 1996.

[13] P. BAUMGARTNER. A first-order Davis-Putnam-Longeman-Loveland procedure. In G. LAKEMEYER et B. NEBEL, réds., *Exploring Artificial Intelligence in the New Millennium — distinguished papers from the 17th IJCAI*. Morgan Kaufmann, 2001.

[14] N. BELDICEANU et E. CONTEJEAN. Introducing global constraints in CHIP. *Mathematical Computing and Modelling*, 20(12):87–123, 1994.

[15] R. BELLMAN. *Dynamic Programming*. Princeton University Press, 1957.

[16] E. BEN-SASSON et WIGDERSON. Short proofs are narrow - resolution made simple. *J. of the ACM*, 48(2):149–169, 2001.

[17] F. BENHAMOU et F. GOUALARD. Universally quantified interval constraints. In *Proc. of the 6th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, LNCS, pages 67–82, Singapore, 2000. Springer.

[18] F. BENHAMOU, F. GOUALARD, L. GRANVILLIERS et J.-F. PUGET. Revising hull and box consistency. In *Proc. of the 16th Int. Conf. on Logic Programming*, pages 230–244, Las Cruces, New Mexico, 1999. MIT Press.

[19] F. BENHAMOU, D. A. MAC ALLESTER et P. VAN HENTENRYCK. CLP(Intervals) revisited. In *Proc. of the Int. Symp. on Logic Programming*, pages 124–138, Ithaca, NY, 1984. MIT Press.

[20] F. BENHAMOU et W. J. OLDER. Applying interval arithmetic to real, integer, and boolean constraints. *J. of Logic Programming*, 32(1):1–24, 1997.

[21] R. BERGHAMMER, B. LEONIUK et U. MILANESE. Implementation of relational algebra using binary decision diagrams. In *Proc. of the 6th Int. WS on Relational Methods in Computer Science (RelMiCS)*, pages 260–274, Oisterwijk, Niederlande, 2001. Springer.

[22] C. BESSIERE, E. HÉBRARD et Walsh T.. Local consistencies in SAT. In *Proc. of the 6th Int. Conf. on Satisfiability Testing*, Portofino (Italy), 2003. Springer.

[23] C. BESSIÈRE, P. MESEGUER, E. C. FREUDER et J. LARROSA. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141(1-2):205–224, 2002.

[24] Ch. BESSIÈRE. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[25] Ch. BESSIÈRE, E. C. FREUDER et J.-C. RÉGIN. Using inference to reduce arc consistency computation. In *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 592–599, Montréal, Québec, 1995. Morgan Kaufmann.

[26] Ch. BESSIÈRE et J.-C. RÉGIN. Arc consistency for general constraint networks: Preliminary results. In *Proc. of the 15th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 398–404, Nagoya, Japan, 1997. Morgan Kaufmann.

[27] Ch. BESSIÈRE et J.-C. RÉGIN. Refining the basic constraint propagation algorithm. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 309–315, Seattle, WA, 2001. Morgan Kaufmann.

[28] A. BIERE, A. CIMATTI, E. M. CLARKE et Y. ZHU. Symbolic model checking without BDDs. In *Proc. of the 5th. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 193–207, Amsterdam, The Netherlands, 1999. Springer.

[29] J. R. BIRGE et F. LOUVEAUX. *Introduction to Stochastic Programming*. Springer Verlag, 1997.

[30] S. BISTARELLI, U. MONTANARI, F. ROSSI, T. SCHIEX, G. VERFAILLIE et H. FARGIER. Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. *Constraints*, 4(3):1999, 1999.

[31] R. BIXBY. The new generation of Integer Programming codes. In N. JUSSIEN et F. LABURTHE, réds., *Proc. of the 4th Int. WS on CP-AI-OR (invited talk)*, Le Croisic, France, 2002.

[32] C. BLIEK, P. SPELLUCI, L. N. VICENTE, A. NEUMAIER, L. GRANVILLIERS, E. MONFROY, F. BENHAMOU, E. HUENS, P. VAN HENTENRYCK, D. SAM-HAROUD et B. FALTINGS. Algorithms for solving nonlinear constraint optimization problems: the state of the art. *Deliverable of the COCONUT IST project*, 2001.

[33] L. BORDEAUX et E. MONFROY. Beyond NP: Arc-consistency for quantified constraints. In *Proc. of the 8th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, pages 371–386, Ithaca, USA, 2002. Springer.

[34] L. BORDEAUX, E. MONFROY et F. BENHAMOU. Improved bounds on the complexity of kB-consistency. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 303–308, Seattle, US, 2001. Morgan Kaufmann.

[35] L. BORDEAUX, E. MONFROY et F. BENHAMOU. Raisonnement automatique sur les propriétés de contraintes numériques. In *Actes des 11èmes J. Franc. de Programmation en Logique et par Contraintes (JFPLC)*, Nice, France, 2002. Hermès. in french,

[36] L. BORDEAUX, E. MONFROY et F. BENHAMOU. Towards automated reasoning on the properties of numerical constraints. In B. O'SULLIVAN, réd., *Recent advances in constraints*, LNAI, Cork, Ireland, 2003. Springer. Selected papers from the 2002 ERCIM/Colognet Workshop on Constraints,

[37] F. BÖRNER, A. KROKHIN, A. BULATOV et P. JEAVONS. Quantified constraints and surjective polymorphisms. Rapport technique, Oxford University, UK, 2002.

[38] J. BOWEN et D. BAHLER. Conditional existence of variables in generalised constraint networks. In *Proc. of the 9th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 215–220, Anaheim, CA, 1991. AAAI Press / MIT Press.

[39] J. BOWEN, R. LAI et D. BAHLER. Lexical imprecision in fuzzy constraint networks. In *Proc. of the 10th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 616–621, San Jose, CA, 1992. AAAI Press / MIT Press.

[40] A. BRAUNSTEIN, M. MÉZARD, R. WEIGT et R. ZECCHINA. Constraint satisfaction by survey propagation. *To appear*, 2003.

[41] P. BRUCKER. Scheduling and constraint propagation. *Discrete Applied Mathematics*, 123:227–256, 2002.

[42] R. E. BRYANT. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[43] R. E. BRYANT. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[44] H. K. BUENING, M. KARPINSKI et A. FLOGEL. Resolution for quantified boolean formulas. *Information and Computation*, 117(1):12–18, 1995.

[45] S. R. BUSS. An introduction to proof theory. In S. BUSS, réd., *Handbook of Proof Theory*, chapter 1, pages 1–78. Elsevier, 1998.

[46] M. CADOLI. The expressive power of binary linear programming. In *Proc. of the 7th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, pages 570–574, Paphos, Cyprus, 2001. Springer.

[47] M. CADOLI et F. M. DONINI. A survey on knowledge compilation. *AI Communications*, 10(3-4):137–150, 1997.

[48] M. CADOLI, I. GIOVAMBATTISTA, L. PALOPOLI, A. SCHAERF et D. VASILE. NP-SPEC: an executable specification language for solving all problems in NP. *Computer Languages*, 26(2-4):165–195, 2000.

[49] M. CADOLI, A GIOVANARDI et M. SCHAERF. An algorithm to evaluate quantified boolean formulae. In *Proc. of the 15th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 262–267, Madison, USA, 1999. AAAI/MIT Press.

[50] M. CADOLI, M. SCHAERF, A. GIOVANARDI et M. GIOVANARDI. An algorithm to evaluate quantified boolean formulae and its experimental evaluation. *J. of Automated Reasoning*, 28(2):101–142, 2002.

[51] Y. CASEAU, F.-X. JOSSET et F. LABURTHE. CLAIRE: Combining sets, search and rules to better express algorithms. *Theory and Practice of Logic Programming*, 2(6):769–805, 2002.

[52] N. CHABRIER, M. CHIAVERINI, V. DANOS, F. FAGES et V. SCHÄCHTER. Modeling and querying biochemical networks. *Theoretical Computer Science*, to appear, 2003.

[53] A. K. CHANDRA et G. MARKOWSKY. On the number of prime implicants. *Discrete Mathematics*, 24:7–11, 1978.

[54] P. CHATALIC et L. SIMON. Multiresolution for SAT checking. *International Journal on Artificial Intelligence Tools (IJAIT)*, 10(4):451–481, 2001.

[55] H. CHEN. Quantified constraint satisfaction problems: Closure properties, complexity, and proof systems. Rapport technique, Cornell University, 2003.

[56] V. CHVATAL. *Linear Programming*. W.H. Freeman Co, 1983.

[57] E. CLARKE. Bounded model-checking and the diameter problem. Invited talk, 8th int. conf. on Principles and Practice of Constraint Programming (CP), 2002,

[58] E. M. CLARKE, A. BIERE, R. RAIMI et Y. ZHU. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[59] E. M. CLARKE, O. GRUMBERG et D. PELED. *Model Checking*. The MIT Press, 2000.

[60] J. G. CLEARY. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[61] M. CLEGG, J. EDMONDS et R. IMPAGLIAZZO. Using the groebner basis algorithm to find proofs of unsatisfiability. In *Proc. of the 28th ACM Symp. on the Theory of Computing (STOC)*, pages 174–183, Philadelphia, PE, 1996. ACM Press.

[62] P. CODOGNET et D. DIAZ. Compiling constraints in CLP(FD). *J. of Logic Programming*, 27(3):185–226, 1996.

[63] E. COHEN et D. KOZEN. A note on the complexity of propositional Hoare logic. *ACM Transactions on Computational Logic*, 1(1):171–174, 2000.

[64] P. COHEN. *Set theory and the continuum hypothesis*. Benjamin, 1966.

[65] A. COLMERAUER. Equations and inequations on finite and infinite tress. In *Proc. of the Int. Conf. on Fifth Generation Computing*, pages 85–99, Tokyo, Japan, 1984. ICOT.

[66] A. COLMERAUER. Espaces d'approximation. In *10èmes Journées Francophones de Programmation en Logique et par Contraintes (JFPLC) (Invited talk)*, Paris, France, 2001.

[67] A. COLMERAUER et T. B. H. DIEP-DAO. Expressiveness of full first order constraints in the algebra of finite or infinite trees. In *Proc. of the 6th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, LNCS, pages 172–186, Singapore, 2000. Springer.

[68] S. A. COOK. The complexity of theorem-proving procedures. In *Conf. Record of the 3rd Symp. on Theory of Computing (STOC)*, pages 151–158, Shaker Heights, Ohio, 1971. ACM.

[69] S. A. COOK et D. MITCHELL. Finding hard instances of the satisfiability problem: A survey. *DIMACS Series in Discrete Math. and Theoretical Computer Science*, 35:1–17, 1997.

[70] M. C. COOPER. An optimal k-consistency algorithm. *Artificial Intelligence*, 41(1):89–95, 1989.

[71] G. CORNUÉJOLS et M. DAWANDE. A class of hard small 0-1 programs. In *6th Int. Conf. on Integer Programming and Combinatorial Optimization (IPCO)*, LNCS, pages 284–293, Houston, TX, 1998. Springer.

[72] M.-M. CORSINI et A. RAUZY. Toupie: The mu-calculus over finite domains as a constraint language. *J. of Automated Reasoning*, 19(2):143–171, 1997.

[73] S. COSTE-MARQUIS, H. FARGIER, J. LANG, D. LE BERRE et P. MARQUIS. Résolution de problèmes booléens quantifiés : problèmes et algorithmes. In *Actes des 13èmes Renc. Francophones d'Intelligence Artificielle (RFIA)*, Angers, France, 2002. in french,

[74] P. COUSOT et R. COUSOT. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the 4th Annual ACM Symp. on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York, NY.

[75] P. COUSOT et R. COUSOT. Abstract interpretation and application to logic programs. *J. of Logic Programming*, 13(2-3):103–179, 1992.

[76] E. DANTSIN, A. GOERDT, E. A. HIRSCH, R. KANNAN, J. KLEINBERG, C. PAPADIMITRIOU, P. RAGHAVAN et U. SCHÖNING. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.

[77] G. DANTZIG. *Linear Programming and Extensions*. Princeton University Press, 1963.

[78] E. DAVIS. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, 1987.

[79] M. DAVIS. *Computability and Unsolvability*. Academic Press, 1958. Dover edition,

[80] M. DAVIS. *Game Theory – A non-technical introduction*. Basic books, Inc, 1970. 3rd edition, Dover, 1997,

[81] M. DAVIS. The early history of automated deduction. In J. A. ROBINSON et A. VORONKOV, réds., *Handbook of Automated Reasoning*, volume 1, pages 3–15. Elsevier and MIT Press, 2001.

[82] M. DAVIS. *Engines of Logic: Mathematicians and the Origin of the Computer*. Norton publishing, 2001. also published under the title *The universal computer: the road from Leibniz to Turing*,

[83] M. DAVIS, G. LOGEMANN et D. LOVELAND. A machine program for theorem-proving. *Communications of the ACM*, 5(7):393–397, 1962.

[84] M. DAVIS et H. PUTNAM. A computing procedure for quantification theory. *J. of the ACM*, 7(3):201–215, 1960.

[85] R. DEBRUYNE et Ch. BESSIÈRE. Domain filtering consistencies. *J. of Artificial Intelligence Research*, 14:205–230, 2001.

[86] R. DECHTER. Constraint networks. In *Encyclopedia of Artificial Intelligence, 2nd ed.*, pages 276–285. John Wiley and Sons, 1992.

[87] R. DECHTER et P. van BEEK. Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308, 1997.

[88] A. del VAL. Tractable databases: How to make propositional unit resolution complete through compilation. In *Proc. of the 4th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 551–561, Bonn, Germany, 1994. Morgan Kaufmann.

[89] G. DELZANNO et A. PODELSKI. Model-checking in CLP. In *Proc. of the 5th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 223–239, Amsterdam, the Netherlands, 1999. Springer.

[90] D. DEUTSCH. *The Fabric of Reality*. Penguin, 1989.

[91] D. DIAZ et P. CODOGNET. Design and implementation of the GNU-Prolog system. *J. of Functional and Logic Programming*, 6, 2001.

[92] T.-B.-H. DIEP-DAO. *Résolution de contraintes du premier ordre dans la théorie des arbres finis ou infinis*. Thèse de Doctorat, Université de la méditerranée, Aix-Marseille II, 2000.

[93] N. DIMITROVA, S. M. MARKOV et E. POPOVA. Extended interval arithmetics: New results and applications. In L. ATANASSOVA et J. HERZBERGER, réds., *Computer Arithmetic and Enclosure Methods*, pages 225–232. Elsevier, 1992.

[94] M. B. DO et S. KAMBHAMPATI. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132:151–182, 2001.

[95] F. M. DONINI, P. LIBERATORE, F. MASSACCI et M. SCHAERF. Solving QBF with SMV. In *Proc. of the 8th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 578–589, Toulouse, France, 2002. Morgan Kaufmann.

[96] J. DORAN et D. MICHIE. Experiments with the graph-traverser program. *Proc. of the Royal Society of London*, 294(2):235–259, 1966.

[97] O. DUBOIS et G. DEQUEN. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 248–253, Seattle, WA, 2001. Morgan Kaufmann.

[98] H.-D. EBBINGHAUS et J. FLUM. *Finite Model Theory*. Springer, 1999.

[99] U. EGLY, T. EITER, H. TOMPITS et S. WOLTRAN. Solving advanced reasoning tasks using quantified boolean formulas. In *Proc. of the 17th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 417–422, Austin, TX, 2001. AAAI/MIT Press.

[100] T. EITER, G. GOTTLOB et Y. GUREVICH. Normal forms for second-order logic over finite structures, and classification of NP optimization problems. *Annals of Pure and Applied Logic*, 78(1-3):111–125, 1996.

[101] E. A. EMERSON. Temporal and modal logic. In J. van LEEUWEN, réd., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages –1072. North-Holland Pub. Co./MIT Press, 1990.

[102] E. A. EMERSON. Model-checking and the mu-calculus. In N. IMMERMAN et P. KOLAITIS, réds., *Descriptive Complexity and Finite Models*, pages 185–214. American Mathematical Society, 1997.

[103] F. FAGES, S. SOLIMAN et V. VIANU. Expressiveness and complexity of concurrent constraint programming: a finite model theoretic approach. Rapport technique 98-14, LIENS, 1998.

[104] R. FAGIN. Finite-model theory - a personal perspective. *Theoretical Computer Science*, 116(1-2):3–31, 1993.

[105] R. A. FAGIN. Generalized first-order spectra and polynomial-time recognizable sets. In R. KARP, réd., *Complexity of Computation*, pages 43–73. SIAM-AMS, 1974.

[106] B. FALTINGS. Arc-consistency for continuous variables. *Artificial Intelligence*, 65(2):363–376, 1994.

[107] R. FELDMANN, B. MONIEN et S. SCHAMBERGER. A distributed algorithm to evaluate quantified boolean formulae. In *Proc. of the 17th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 285–290, Austin, TX, 2001. AAAI/MIT Press.

[108] R. FIKES. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1(1/2):27–120, 1970.

[109] R. FIKES et N. J. NILSSON. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[110] R. FRAÏSSÉ. *Sur les Classifications des Systems de Relations*. Publications Scientifiques de l'Université d'Alger I, 1954.

[111] E. FREUDER. Constraint programming: In pursuit of the holy grail. *Constraints*, 2(1):57–61, 1997.

[112] E. C. FREUDER. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[113] L. FRIBOURG. Constraint logic programming applied to model checking. In *Proc. of the 9th Int. WS on Logic Programming Synthesis and Transformation (LOPSTR)*, pages 30–41, Venezia, Italia, 1999. Springer.

[114] A. M. FRISCH, D. SHERIDAN et T. WALSH. A fixpoint based encoding for bounded model checking. In *Proc. of the 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 238–255, Portland, OR, 2002. Springer.

[115] T. FRÜHWIRTH. Constraint solving with constraint handling rules. In *Intensional Programming II — papers based on the ISLIP 1999 Conference*, pages 14–30. World Scientific Singapore, 2000.

[116] T. W. FRÜHWIRTH. Theory and practice of Constraint Handling Rules. *J. of Logic Programming*, 37(1-3):95–138, 1998.

[117] H. GANZINGER et D. A. MAC ALLESTER. Logical algorithms. In *Proc. of the 18th Int. Conf. on Logic Programming (ICLP)*, pages 209–223, Copenhagen, Denmark, 2002. Springer.

[118] M. GAREY et D.S. JOHNSON. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA., 1979.

[119] I. P. GENT. Arc consistency in SAT. In *Proc. of the 15th European Conf. on Artificial Intelligence (ECAI)*, pages 121–125, Lyon, France, 2002. IOS Press.

[120] I. P. GENT, H. H. HOOS, A. G. D. ROWLEY et K. SMYTH. Using stochastic local search to solve quantified boolean formulae. In *Proc. of the 9th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, Kinsale, Ireland, 2003. to appear,

[121] I. P. GENT et B. M. SMITH. Symmetry breaking in constraint programming. In *Proc. of the 14th European Conf. on Artificial Intelligence (ECAI)*, pages 599–603, Berlin, Deutschland, 2000. IOS Press.

[122] I. P. GENT et T. WALSH. Beyond NP: the QSAT phase transition. In *Proc. of the 15th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 648–653, Madison, USA, 1999. AAAI/MIT Press.

[123] M. L. GINSBERG. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.

[124] M. L. GINSBERG et A. J. PARKES. Satisfiability algorithms and finite quantification. In *Proc. of the 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 690–701, Breckenridge, Colorado, 2000. Morgan Kaufmann.

[125] J.-Y. GIRARD. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[126] E. GIUNCHIGLIA, M. NARIZZANO et A. TACCHELLA. Backjumping for quantified boolean logic satisfiability. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 275–281, Seattle, 2001. Morgan Kaufmann.

[127] E. GIUNCHIGLIA, M. NARIZZANO et A. TACCHELLA. QUBE: A system for deciding quantified boolean formulas satisfiability. In *Proc. of the 1st Int. Joint Conf. on Automated Reasoning (IJCAR)*, LNCS, pages 364–369, Siena, Italy, 2001. Springer.

[128] E. GIUNCHIGLIA, M. NARIZZANO et A. TACCHELLA. Learning for quantified boolean logic satisfiability. In *Proc. of the 18th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 649–654, Edmonton, Alberta, 2002. AAAI/MIT Press.

[129] E. GIUNCHIGLIA, M. NARIZZANO et A. TACCHELLA. Backjumping for quantified boolean logic satisfiability. *Artificial Intelligence*, 145(1-2):199–2003, 2003.

[130] F. GIUNCHIGLIA et P. TRAVERSO. Planning as model-checking. In *Proc. of the 5th European Conf. on Planning (ECP)*, pages 1–20, Durham, UK, 1999. Springer. Invited talk,

[131] K. GÖDEL. Über Formal Unentscheidbare Sätze der Pincipia Mathematica und Verwandter Systeme i. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. eng. translation entitled "On Formally Undecidable Propositions in Principia Mathematica and Related Systems" appears in [207],

[132] C. GOMES et B. SELMAN. Satisfied with physics. *Science*, 287:784–785, 2002. Comment on the paper [206],

[133] M. GONDRAN et M. MINOUX. *Graphes et Algorithmes*. Eyrolles, 2nd édition, 1985.

[134] G. GOTTLOB, T. KOLAITIS et T. SCHWENTICK. Existential second-order logic over graphs: Charting the tractability frontier. In *Proc. of the 41st Symp. on Foundations of Computer Science (FOCS) (final version to appear in JACM, 2003)*, pages 664–674, Redondo Beach, California, 2000. IEEE Computer Society.

[135] G. GOTTLOB, N. LEONE et F. SCARCELLO. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[136] J. F. GROOTE et H. ZANTEMA. Resolution and binary decision diagrams cannot simulate each other polynomially. In *Proc. of the 4th Int. Andrei Ershov Memorial Conf. on Perspectives of System Informatics (PSI)*, pages 33–38, Novosibirsk, Russia, 2001. Springer.

[137] A. HAKEN. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[138] J. Y. HALPERN et M. VARDI. Model checking vs. theorem proving: A manifesto. In *Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 325–334, Cambridge, MA, USA, 1991. Morgan Kaufmann.

[139] E. R. HANSEN. *Global Optimization Using Interval Analysis*. Pure and Applied Mathematics. Marcel Dekker Inc., 1992.

[140] J.-K. HAO, P. GALINIER et M. HABIB. Métaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes. *Revue d'Intelligence Artificielle*, 13(2):283–324, 1999.

[141] P. E. HART, N. J. NILSSON et B. RAPHAEL. A formal basis for the heuristic determination of minimum cost paths in graphs. *IEEE Trans. on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[142] J. HARTMANIS et R. E. STEARNS. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117(5):285–306, 1965.

[143] W. D. HARVEY et M. L. GINSBERG. Limited discrepancy search. In *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 607–615, Montréal, Québec, 1995. Morgan Kaufmann.

[144] B. HAYES. On the threshold. *American Scientist*, 91(1):12–17, 2003.

[145] M. HENZ et J. WÜRTZ. Using oz for college timetabling. In *Proc. of the 1st Int. Conf. on Practice and Theory of Automated Timetabling (PATAT)*, pages 162–177, Edimburgh, UK, 1995. Springer.

[146] T. HICKEY. Functional constraints inCLP languages. In *Constraint Logic Programming, Selected Research — proc. of the Workshop*, pages 355–381, Marseille, France, *The MIT Press*. 1991.

[147] E. A. HIRSCH et A. KOJEVNIKOV. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *submitted*.

[148] H. HONG. An improvement of the projection operator in Cylindrical Algebraic Decomposition. In *Proc. of the Int. Symp. on Symbolic and Algebraic Computation (ISSAC)*, pages 261–264, Tokyo, Japan, 1990. ACM Press and Addison-Wesley.

[149] H. HOOS. On the run-time behaviour of stochastic local search methods for SAT. In *Proc. of the 16th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 661–666, Orlando, Florida, 1999. AAAI Press / The MIT Press.

[150] E. HYVÖNEN. Constraint reasoning based on interval arithmetic. In N. S. SRIDHARAN, réd., *Proc. of the 11th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 1193–1198, Detroit, MI, 1989. Morgan Kaufmann.

[151] N. IMMERMAN. Nondeterministic space is closed under complementation. *SIAM J. of Computing*, 17(5):935–938, 1988.

[152] N. IMMERMAN. Descriptive complexity: A logician's approach to computation. *Notices of the American Mathematical Society*, 42(10):1127–1133, 1995.

[153] N. IMMERMAN. *Descriptive Complexity*. Springer, 1999.

[154] J. JAFFAR et J.-L. LASSEZ. Constraint Logic Programming. In *Conf. Record of the 14th Annual ACM Symp. on Principles of Programming Languages (POPL)*, pages 111–119, Munich, Germany, 1987.

[155] J. JAFFAR et M.-J. MAHER. Constraint logic programming: A survey. *J. of Logic Programming*, 19(20):503–581, 1994.

[156] R. M. JENSEN, R. E. BRYANT et M. M. VELOSO. SetA*: An efficient BDD-based heuristic search algorithm. In *Proc. of the 18th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 668–673, Edmonton, Alberta, 2002. AAAI Press/MIT Press.

[157] M. JIRSTRAND. Cylindrical Algebraic Decomposition - an introduction. Rapport technique, Dpt. of electrical engineering, Linköping University, 1995.

[158] N. KAMARKAR. A new polynomial-time algorithm for linear programming. *Combinatorics*, 4:373–395, 1984.

[159] P. KANELLAKIS, H. G. MAIRSON et J. C. MITCHELL. Unification and ML type reconstruction. In J.-L. LASSEZ et G. PLOTKIN, réds., *Computational Logic: essays in honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[160] R. M. KARP. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. Plenum Press, 1972.

[161] R. M. KARP. Combinatorics, complexity and randomness. *Communications of the ACM*, 29:98–111, 1986. Turing Award Lecture,

[162] S. KASIF. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):99–118, 1990.

[163] A. A. KAUTZ et B. SELMAN. Unifying SAT-based and graph-based planning. In *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 318–325, Stockolm, Sweden, 1999. Morgan Kaufmann.

[164] H. A. KAUTZ, D. A. MAC ALLESTER et B. SELMAN. Encoding plans in propositional logic. In *Proc. of the 5th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 374–384, Cambridge, MA, 1996. Morgan Kaufmann.

[165] L.G. KHACHIAN. A polynomial time algorithm for linear programming. *Soviet Mathematics Dokadly*, 20:191–194, 1979.

[166] C. KIRCHNER et H. KIRCHNER. *Rewriting, Solving, Proving*. unpublished, 2001. `http://www.loria.fr/~ckirchne/rsp.ps.gz`,

[167] S. C. KLEENE. *Introduction to metamathematics*. North Holland, 1952.

[168] A. N. KOLMOGOROV et S. V. FOMIN. *Introductory Real Analysis*. Dover Publications, 1975. Translated and edited from Russian by Richard A. Silverman,

[169] R. A. KOWALSKI. Predicate logic as programming language. In *Proc. of 6th IFIP World Computer Congress*, pages 569–574, Stockholm, 1974.

[170] R. A. KOWALSKI. *Logic for problem solving*. North-Holland, 1979.

[171] R. E. LADNER. On the structure of polynomial time reducibility. *J. of the ACM*, 22(1):155–171, 1975.

[172] J.-L. LAURIÈRE. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[173] D. LEIVANT. Higher-order logic. In D. M. GABBAY, C. J. HOGGER et J. A. ROBINSON, réds., *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2: Deduction Methodologies, vol. 2: Deduction methodologies*, pages 229–321. Clarendon Press, 1994.

[174] D. LESAINT. Inferring constraint types in constraint programming. In *Proc. of the 8th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, LNCS, pages 492–507, Ithaca, NY, 2002. Springer.

[175] F. LETOMBE. Algorithmique pour les formules booléennes quantifiées. Rapport technique, Centre de Recherche en Informatique de Lens, 2003.

[176] R. LETZ. Lemma and model caching in decision procedures for quantified boolean formulas. In *Proc. of the 11th Int. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, pages 160–175, Copenhagen, 2002. Springer.

[177] L. LEVIN. Randomness and nondeterminism. In *Proc. of the Int. Congress of Mathematicians*, Zurich, 1994. Birkhauser Verlag. invited talk,

[178] L. A. LEVIN. Universal sorting problems. *Problems of Information Transmission*, 9:265–266, 1973.

[179] O. LHOMME. Consistency techniques for numeric CSPs. In *Proc. of the 13th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 232–238, Chambéry, France, 1993. Morgan Kaufmann.

[180] L. LIBKIN. Variable independence for first-order definable constraints. *ACM Transactions on Computational Logic (TOCL)*, to appear, 2003.

[181] P. LINCOLN, J. C. MITCHELL, Scedrov A. et N. SHANKAR. Decision problems for propositional linear logic. In *proc. of the 31st IEEE Conf. on Foundations of Computing Science*, pages 662–671. IEEE Press, 1990.

[182] M. L. LITTMAN et MAJERCIK. Contingent planning under uncertainty via stochastic satisfiability. *Artificial Intelligence*, 2003. To appear,

[183] M. L. LITTMAN, S. M. MAJERCIK et T. PITASSI. Stochastic boolean satisfiability. *J. of Automated Reasoning*, 27(3):251–296, 2001.

[184] L. LOVÁSZ. Discrete and continuous: Two sides of the same? *Geom. Funt. Anal.*, pages 359–382, 2000.

[185] A. MACKWORTH. On reading sketch maps. In *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 598–606, Cambridge, MA, 1977. William Kaufman.

[186] A.K. MACKWORTH. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[187] M. J. MAHER. Propagation completeness of reactive constraints. In *Proc. of the 18th Int. Cont. on Logic Programming (ICLP)*, pages 148–162, Copenhagen, Denmark, 2002. Springer.

[188] S. MANANDHAR, A. TARIM et T. WALSH. Scenario-based stochastic constraint programming. In *Proc. of the 18th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, page to appear, Acapulco, Mexico, 2003. Morgan Kaufmann.

[189] Z. MANNA et A. PNUELI. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1991.

[190] A. MARGARIS. *First-Order Mathematical Logic*. Dover, 1967. Reprint. 1990,

[191] J. P. MARQUES SILVA et K. A. SAKALLAH. Conflict analysis in search algorithms for satisfiability. In *Proc. of the 8th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, pages 467–469, Toulouse, France, 1996. IEEE Press.

[192] P. MARQUIS. Consequence-finding algorithms. In D. GABBAY et Ph. SMETS, réds., *Handbook of defeasible reasoning and uncertainty management systems. Vol 5: algorithms for defeasible and uncertain reasoning*, chapter 2, pages 41–145. Kluwer academic publishers, 2000.

[193] P. MATHIEU et J.-P. DELAHAYE. A kind of logical compilation for knowledge bases. *Theoretical Computer Science*, 131(1):197–218, 1994.

[194] J. MAUSS, F. SEELISCH et M. M. TATAR. A relational constraint solver for model-based engineering. In *Proc. of the 8th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, pages 696–701, Ithaca, NY, 2002. Springer.

[195] D. A. MC ALLESTER. Notes for course 6.824 on AI — chapters on CSPs, graph search, games. Rapport technique, MIT, 1995.

[196] Z. MICHALEWICZ. A survey of constraint handling techniques in evolutionary computation methods. In *Proc. of the 4th Int. Conf. on Evolutionary Programming (EP)*, pages 135–155, San Diego, CA, 1995. MIT Press.

[197] R. MILNER. Elements of interaction. *Communications of the ACM*, 36(1):78–89, 1993. Turing award lecture,

[198] S.-I. MINATO. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of the 30th Design Automation Conference (DAC)*, pages 272–277, Dallas, Texas, 1993. ACM Press.

[199] D. G. MITCHELL. Hard problems for CSP algorithms. In *Proc. of the 15th Nat. Conf on Artificial Intelligence (AAAI)*, pages 398–405, Madison WI, 1998. AAAI Press/MIT Press.

[200] R. MOHR et T. C. HENDERSON. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[201] R. MOHR et G. MASINI. Good old discrete relaxation. In *Proc. of the 8th European Conf. on Artificial Intelligence (ECAI)*, pages 651–656, Munich, Germany, 1988. Pitmann Publishing, London.

[202] U. MONTANARI. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):85–132, 1974.

[203] G. MOORE. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[204] R. E. MOORE. *Interval Analysis*. Prentice-Hall, 1966.

[205] M. W. MOSKEWICZ, C. F. MADIGAN, Y. ZHAO, L. ZHANG et S. MALIK. Chaff: Engineering an efficient SAT solver. In *Proc. of the 38th Int. Conf. on Design Automation (DAC)*, pages 530–535, Las Vegas, NV, 2001. ACM.

[206] M. MÉZARD, G. PARISI et R. ZECCHINA. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297:812–815, 2002.

[207] E. NAGEL, J. R. NEWMAN, K. GÖDEL et J.-Y. GIRARD. *Le théorème de Gödel*. Le Seuil, 1989.

[208] A. NEWELL, J. C. SHAW et H. A. SIMON. Report on a general problem-solving program. In *Proc. of the 1st IFIP World Computer Congress*, pages 256–264, Paris, France, 1960. UNESCO.

[209] C. PAPADIMITRIOU. Games against nature. *J. of Computer and System Sciences (JCSS)*, 31(2):288–301, 1985.

[210] C. PAPADIMITRIOU. NP-completeness: A retrospective. In *24th Int. Col. on Automata, Languages and Programming*, LNCS, pages 2–6, Bologna, Italy, 1997. Springer.

[211] C. PAPADIMITRIOU et M. YANNAKIS. A note on succint representations of graphs. *Information and Control*, 71(3):181–185, 1986.

[212] Ch. H. PAPADIMITRIOU. *Computational Complexity*. Addison Wesley, 1994.

[213] Ch. H. PAPADIMITRIOU et K. STEIGLIZ. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982. Dover edition, 1998,

[214] J. PEARL. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[215] D. PLAISTED, A. BIERE et Y. ZHU. A satisfiability procedure for quantified boolean formulae. *Discrete Applied Mathematics*, 130(2):291–328, 2003.

[216] P. PROSSER. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[217] S. RATSCHAN. Convergent approximate solving of first-order constraints by approximate quantifiers. *ACM Transactions on Computational Logic TOCL*, To appear, 2003.

[218] A. RAUZY. A brief introduction to binary decision diagrams. *RAIRO-APII-JESA*, 30(8):1033–1051, 1996.

[219] J.-C. RÉGIN. A filtering algorithm for constraints of difference in CSPs. In *Proc. of the 12th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 362–367, Seattle, WA, 1994. AAAI Press.

[220] J. RINTANEN. Improvements to the evaluation of quantified boolean formulae. In *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 1192–1197, Stockholm, Sweden, 1999. Morgan Kaufmann.

[221] J. RINTANEN. Partial implicit unfolding in the Davis-Putnam procedure for quantified boolean formulae. In *Proc. of the 8th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250 of *LNCS*, pages 362–376, Havana, Cuba, 2001. Springer.

[222] J. RINTANEN. *Introduction to Automated Planning (unpublished course notes.* Freiburg Institut für Informatik, 2003.

[223] J. RINTANEN et J. HOFFMAN. An overview of recent algorithms for AI planning. *Künstliche Intelligenz*, 2(5-11):2001, 2001.

[224] R. L. RIVEST, L. M. ADLEMAN et A. SHAMIR. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 2(1978):120–126, 21.

[225] J. A. ROBINSON. A machine-oriented logic based on the resolution principle. *J. of the ACM*, 12(1):23–41, 1965.

[226] J. A. ROBINSON. Computational logic: Memories of the past and challenges for the future. In *Proc. of the 1st Int. Conf. on Computational Logic*, LNCS, pages 1–24, London, 2000. Springer.

[227] S. J. RUSSEL et P. NORVIG. *Artificial Intelligence: a Modern Approach, 2nd ed.* Prentice Hall, 2002.

[228] D. SABIN et E. FREUDER. Contradicting conventional wisdom in constraint satisfaction. In *Proc. of the 11th European Conf. on Artificial Intelligence (ECAI)*, pages 125–129, Amsterdam, the Netherlands, 1994. John Wiley and Sons.

[229] D. SAM-HAROUD et B. FALTINGS. Consistency techniques for continuous constraints. *Constraints*, 1(1-2):85–118, 1996.

[230] W. J. SAVITCH. Relationships between nondeterministic and deterministic tape complexities. *J. of Computer and System Sciences (JCSS)*, 4(2):177–192, 1970.

[231] J. SCHOENFIELD. *Mathematical Logic*. Addison-Wesley, 1967.

[232] B. SELMAN, H. KAUTZ et B. COHEN. Noise strategies for improving local search. In *Proc. of the 12th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 337–343, Seattle, WA, 1994. AAAI Press.

[233] B. SELMAN, H. J. LEVESQUE et D. G. MITCHELL. A new method for solving hard satisfiability problems. In *Proc. of the 10th Nat. Conf. on Artificial Intelligence (AAAI)*, pages 440–446, San Jose, CA, 1992. AAAI Press / MIT Press.

[234] A. SHAMIR. IP = PSPACE. In *Proc. of the 31st Int. Symp. on Foundations of Computer Science (FOCS)*, pages 11–15, St. Louis, Missouri, 1990. IEEE.

[235] P. W. SHOR. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. on Computing*, 26(5):1484–1509, 1997.

[236] L. SIKLÓSSY. The way it was at carnegie: 1965-1968. *Revue d'Intelligence Artificielle — Numéro spécial: Hommage à Herbert Simon*, 16(1-2):17–27, 2002.

[237] L. SIMON et A. DEL VAL. Efficient consequence finding. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 359–365, Seattle, WA, 2001. Morgan Kaufmann.

[238] A. SRINIVASAN, T. KAM, S. MALIK et R. K. KURSHAN. Algorithms for discrete function manipulation. In *Proc. of the 1st Int. Conf. on Computer Aided Design ICCAD*, pages 92–95, NY, 1990. IEEE.

[239] R. E. STEARNS et H. B. HUNT. Exploiting structure in quantified formulas. *J. of Algorithms*, 43(2):220–263, 2002.

[240] L. J. STOCKMEYER. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[241] L. J. STOCKMEYER et A. R. MEYER. Word problems requiring exponential time: Preliminary report. In *Conf. Record of the 5th Symp. on the Theory of Computing (STOC)*, pages 1–9, Austin, TX, 1973. ACM.

[242] I. E. SUTHERLAND. *Sketchpad, A Man-Machine Graphical Communication System*. Garland Publishing, 1963.

[243] R. SZELEPCSÉNYI. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[244] A. TARSKI. *A decision method for elementary algebra and geometry*. Univ. of California Press, 1951.

[245] A. TARSKI. A lattice-theoretic fixpoint theorem and its applications. *Pacific J. of Mathematics*, 5:285–309, 1955.

[246] A. M. TURING. On computable numbers, with an application to the entscheidungsproblem. *Proc. of the London Mathematical Society*, 2(42):230–265, 1937.

[247] P. VAN HENTENRYCK. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[248] P. VAN HENTENRYCK. *The OPL Optimization Programming Language*. the MIT press, 1999.

[249] P. VAN HENTENRYCK, Y. DEVILLE et C.-M. TENG. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321, 1992.

[250] P. VAN HENTENRYCK, L. PERRON et J.-F. PUGET. Search and strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):285–320, 2000.

[251] P. VAN HENTENRYCK, V. A. SARASWAT et Y. DEVILLE. Design, implementation, and evaluation of the constraint language cc(fd). *J. of Logic Programming*, 37(1-3):139–164, 1998.

[252] M. Y. VARDI. The complexity of relational query languages. In *Proc. of the 14th Symp. on Theory of Computing (STOC)*, pages 137–146, San Francisco, CA, 1982. ACM.

[253] M. Y. VARDI. Constraint satisfaction and database theory: a tutorial. In *Proc. of the 19th Symp. on Principles of Database Systems (PODS)*, pages 76–85, Dallas, TX, 2000. ACM.

[254] V. VIANU. Rule-based languages. *Annals of Mathematics and Artificial Intelligence*, 19(1-2):215–259, 1997.

[255] J. VON NEUMANN et O. MORGENSTERN. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[256] A. VORONKOV. Course notes on logic in computer science. Rapport technique, University of Manchester, 2001.
`http://rpc25.cs.man.ac.uk/cgi-bin/teaching/lics/2001/index.cgi`,

[257] M. WALLACE. Search in AI - escaping the CSP straightjacket. In *Proc.*, 14th European Conf. on Artificial Intelligence (ECAI) — invited talk, 2000. IOS Press.

[258] T. WALSH. Stochastic constraint programming. In *Proc. of the 15th European Conf. on Artificial Intelligence (ECAI)*, pages 111–115, Lyon, France, 2002. John Wiley and Sons.

[259] D. L. WALTZ. Generating semantic descriptions from drawings of scenes with shadows. Rapport technique MAC-AI-TR-271, MIT, 1972.

[260] A. C. WARD, T. LOZANO-PEREZ et W. P. SEERING. Extending the constraint propagation of intervals. In *Proc. of the 11th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 1453–1460, Detroit, Michigan, 1989. Morgan Kaufmann.

[261] R. WILLIAMS. Algorithms for quantified boolean formulas. In *Proc. of the 13th Int. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 299–307, San Francisco, CA, 2002. ACM/SIAM.

[262] S. WOLFRAM. *A new kind of science*. Wolfram Media, 2002.

[263] B. YAMNITSKY et L. A. LEVIN. An 1965 linear programming algorithm runs in polynomial time. In *Proc. of the Symp. on Foundations of Computer Science (FOCS)*, pages 327–328. IEEE, 1982.

[264] Z. YUANLIN et R. H. C. YAP. Making AC3 an optimal algorithm. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 316–321, Seattle, WA, 2001. Morgan Kaufmann.

[265] H. ZHANG et M. E. STICKEL. Implementing the Davis-Putnam method. *J. of Automated Reasoning*, 24(1/2):277–296, 2000.

[266] L. ZHANG et S. MALIK. Conflict driven learning in a quantified boolean satisfiability solver. In *Proc. of the 13th Int. Conf. on Computer Aided Design (ICCAD)*, San Jose, CA, 2002. ACM/IEEE.

[267] L. ZHANG et S. MALIK. The quest for efficient boolean satisfiability solvers. In *Proc. of the 18th Int. Conf. on Computer Aided Deduction (CADE) and Proc. of the 14th Conf. on Computer Aided Verification (CAV)*, pages 295–313 (CADE), 17–36 (CAV), Copenhagen, Denmark, 2002. Springer. Common invited talk to CADE and CAV,

[268] L. ZHANG et S. MALIK. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *Proc. of the 8th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, pages 200–215, Ithaca, USA, 2002. Springer.

[269] W. ZHANG. *State Space Search: Algorithms, Complexity, Extensions, and Applications*. Springer, 1999.

Table des matières

Partie I — Introduction et contexte de la thèse

Partie II — Logique et déduction

Partie III — Résolution de contraintes quantifiées

Index

Résolution de problèmes combinatoires modélisés par des contraintes quantifiées

Lucas BORDEAUX

Résumé

Cette thèse s'inscrit dans le contexte de la *programmation par contraintes* sur les domaines finis, un paradigme de programmation qui consiste à exprimer des problèmes combinatoires par le biais de langages formels. L'emploi d'algorithmes de résolution de formules logiques permet ainsi de résoudre une grande variété de problèmes. Les résolveurs de contraintes actuels sont basés sur une logique propositionnelle de laquelle la notion de quantification ("pour tout", "il existe") est absente.
Le sujet principal de la thèse est le problème de résolution de contraintes discrètes quantifiées. L'étude de la restriction booléenne de ce problème a récemment fait l'objet d'une intense recherche dans la communauté SAT. A priori, cette restriction n'est cependant pas justifiée et de nombreuses applications s'expriment grâce à une extension du cadre des problèmes de satisfaction de contraintes (CSP) à domaines finis avec quantificateurs. Nous proposons donc d'étudier le problème des CSP quantifiés ; notre principale contribution est de formuler une technique d'*arc-consistance quantifiée*, généralisant la technique classique de résolution de CSP. On montre ainsi que l'essentiel du cadre classique de résolution de CSP (notion d'opérateurs de réduction, propriétés de confluence, propagation d'intervalles) peut être adapté à la résolution de problèmes quantifiés.
Enfin, nous terminons la thèse en ouvrant une problématique plus prospective : l'utilisation de techniques de compilation logique pour déterminer si les problèmes décrits dans certaines logiques quantifiées peuvent être résolus de manière efficace.

Mots-clés : Programmation par contraintes, contraintes quantifiées, problèmes PSPACE-complets

Abstract

The research area of this thesis is *constraint programming* over finite domains, a programming paradigm in which combinatorial problems are stated using formal languages. Algorithms for solving logical formulae can hence be used to solve a wide variety of problems. Existing constraint solvers are based on a propositional logic which excludes quantifiers ("forall", "exists").
The main topic of the thesis is the problem of solving quantified discrete constraints. The restriction of this problem to the domain of booleans has recently received much attention from the SAT community; nevertheless, this restriction does not always seem legitimate, and many applications can be expressed more satisfactorily in a generalization of the framework of constraint satisfaction problems with arbitrary quantifiers. We propose to study the problem of quantified CSP; our main contribution is to propose *quantified arc-consistency* as a generalization of the well-known constraint-solving technique. Therefore, we show that the bulk of the classical CSP framework (notions of narrowing operators, confluence properties, interval propagation) can be adapted to quantified constraints.
We conclude the thesis on a more open topic: the use of logical compilation techniques to determine whether problems described in a particular form of quantified logic can be solved efficiently.

Keywords: Constraint programming, quantified constraints, PSPACE-complete problems