Apprentissage de problèmes de contraintes

Matthieu Lopez

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES

LABORATOIRE : LIFO

THÈSE présentée par :

Matthieu LOPEZ

soutenue le : 8 décembre 2011

pour obtenir le grade de : Docteur de l'université d'Orléans

Discipline/ Spécialité : Informatique

Apprentissage de problèmes de contraintes

THÈSE dirigée par :

Arnaud LALLOUET	Professeur, Université de Caen

RAPPORTEURS :

Lakhdar SAIS	Professeur d'Université, Université d'Artois
Michèle SEBAG	Directeur de Recherche, Université Paris Sud

JURY :

Laurent GRANVILLIERS	Professeur, Université de Nantes
Arnaud LALLOUET	Professeur, Université de Caen
Lionel MARTIN	Maître de Conférence, Université d'Orléans
Céline ROUVEIROL	Professeur, Université Paris Nord
Lakhdar SAIS	Professeur, Université d'Artois
Michèle SEBAG	Directeur de Recherche, Université Paris Sud
Christel VRAIN	Professeur, Université d'Orléans

Remerciements

Une thèse est le résultat d'un travail personnel important et passionné mais qui ne peut être réussi sans l'appui, le soutien et les conseils de nombreuses personnes. Cela s'est vérifié dans mon cas et (trop) nombreuses sont les personnes que j'aimerais remercier pour m'avoir accompagné pendant ces premières années de recherche.

Pour commencer, je tiens à remercier tous les membres de mon jury pour avoir montré un intérêt particulier pour mon travail. Merci donc à Laurent Granvilliers pour avoir accepté de présider ce jury et avoir réussi à animer la soutenance lui donnant une dimension à la fois intéressante et vivante. Ma gratitude s'adresse également à Michèle Sebag et Lakhdar Sais pour m'avoir fait l'honneur de rapporter ma thèse et pour les nombreuses remarques et conseils qu'ils m'ont donnés. Je remercie également Céline Rouveirol pour sa participation à mon jury et un très intéressant échange lors de la séance de questions.

Cette thèse n'aurait sans doute pas existé sans ma rencontre avec Arnaud Lallouet pendant mon master avec notamment mon stage de recherche qui a débouché sur mon doctorat sous sa direction. Même si les hasards de la vie ont fait qu'il a dû partir rejoindre l'Université de Caen, nous avons réussi à garder contact et continué à développer de nouvelles idées et travaux ensembles malgré cette contrainte. Je tiens également à remercier Lionel Martin qui a récupéré le « paquet » suite à la promotion d'Arnaud. Travailler avec Lionel a été un réel enrichissement et l'occasion de débats très animés mais aboutissant souvent à très bonnes choses. Ces débats ont d'ailleurs continué à se développer avec l'arrivée dans notre groupe de travail de Christel Vrain. Christel a su apporter une maturité à mes travaux en me donnant les clés pour prendre le recul nécessaire à l'accomplissement de ce doctorat. Je tiens également à la remercier pour le très bon travail de relecture et de correction qu'elle a fait sur ma thèse (même si je regrette d'avoir échappé au « rouge Christel »). Pour finir, je remercie mes récents collaborateurs de Montpellier, Christian Bessière, Rémi Coletta et Frédéric Koriche, pour leur conseil et leur expertise notamment sur le dernier chapitre de cette thèse.

Ces années n'auraient pas été aussi agréables sans la convivialité et le soutien apportés à la fois par le LIFO et le département informatique de l'Université d'Orléans dans lequel j'ai effectué de nombreuses heures d'enseignement. Je tiens donc à remercier tout le personnel, scientifique et administratif, par l'intermédiaire de leurs directeurs, Christel Vrain et Jérôme Durand-Lose du côté LIFO, Ali Ed-Bali du côté département. J'ai également une pensée à tous les étudiants qui se sont succédés dans mes cours et qui m'ont permis de sortir un peu la tête du guidon. Ces conditions ont également été sublimées par une ambiance entre doctorants/ATER/post-doc/jeunes docteurs d'une très grande qualité. Je tiens notamment à remercier Jérémie Vautard, mon « grand-frère » de thèse, pour les nombreux échanges scientifiques mais également peu scientifiques (Jérémie a un don pour trouver

une multitude de vidéos/musiques/images très surprenantes dans les méandres d'internet), et également pour les quelques missions que nous a eu la chance de faire ensemble (avec de très bons souvenirs de Lisbonne et Caen). Un grand merci également à Julien Tesson. D'abord pour notre dernière année de master recherche où malgré la « compétition » et pression engendrées par cette année particulière, Julien a toujours réussi à prendre de la hauteur et mettre une excellente ambiance dans la promotion. Ensuite et car notre parcours de doctorats possède de nombreuses similarités, pour avoir été toujours prêt à m'écouter (voire à me supporter) pendant nos nombreuses thérapies de groupe. Un merci particulier à Mathieu Chapelle pour avoir su animer la vie des doctorants du LIFO par l'intermédiaire de sorties (légendaires) et activités sportives permettant de prendre de très bons bols d'air.

Je tiens aussi à étendre les remerciements aux doctorants extérieurs au LIFO, je pense notamment à tous ceux que j'ai rencontrés grâce à l'ADSO. Les années passées dans cette association ont été l'occasion de très bonnes rencontres et d'ouvertures sur d'autres mondes que je n'aurai jamais soupçonnées.

Merci à ma famille (qui s'est fortement agrandie pendant ces années de thèse) pour avoir toujours montré de l'intérêt pour mon choix de continuer en doctorat, votre présence tout au long de cette épopée et malgré les questions dangereuses à poser à un doctorant : « Mais tu travailles sur quoi en fait ? », « Et ça va te mener où ? », « Tu soutiens bientôt ? ».

Et comme il est de coutume de finir par le plus important, je tiens à remercier très (TRÈS) fortement Marie, mon épouse. Toujours en première ligne dans les mauvais moments, elle s'est révélée une combattante merveilleuse capable de me redonner l'envie, la force et le courage de continuer dans les moments les plus durs. Ces années n'auraient jamais aussi merveilleuses sans ta présence à mes côtés !

Table des matières

Table des figures

Chapitre 1

Introduction

La programmation par contraintes est un cadre formel permettant l'expression de problèmes puis leur résolution par des solveurs. Ce domaine est un parfait exemple des succès qui peuvent être rencontrés en intelligence artificielle. Avec des applications réelles dans le planning de trafics aériens, l'optimisation de productions de véhicules ou encore l'ordonnancement de services de personnels, les solveurs actuels ont démontré leur efficacité et fait de la programmation par contraintes une discipline mature. Cependant, la modélisation des problèmes, étape consistant à exprimer son problème de manière formelle utilisable par les solveurs, reste une étape essentielle du processus qui, pour être réalisée avec succès, nécessite des connaissances et une expérience assez importantes. Ce fait reconnu par la communauté des problèmes de satisfaction de contraintes peut devenir une limitation importante pour la poursuite du développement de la discipline[Pug04]. D'où l'idée développée ces dernières années de proposer différentes approches pour rendre plus accessible la programmation par contraintes. Langage de modélisation haut niveau, aide à la modélisation, reformulation du modèle sont des sujets qui ont particulièrement intéressé la communauté ces dernières années.

En parallèle de cela, l'apprentissage automatique est un autre domaine de l'intelligence artificielle à avoir montré son efficacité. Cette thématique propose d'utiliser un processus d'apprentissage pour réussir des tâches ne pouvant être simplement automatisées. Un problème d'apprentissage va consister, à partir de données représentant le concept que l'on souhaite construire, à généraliser ces données pour en extraire une information plus générale. Ce qui nous intéresse dans cette thèse est la possibilité, en utilisant de telles techniques, d'automatiser, ou du moins de réduire l'implication de l'utilisateur humain, dans la modélisation des problèmes. Comment procède-t-on ? Nous partons de l'hypothèse que si l'utilisateur ne sait pas modéliser son problème formellement, il peut par contre identifier les solutions de son problème si on les lui présente. À partir des combinaisons qu'il aura identifiées comme des solutions, et de celles qu'il aura rejetées, la tâche consistera à reconstruire le modèle en trouvant les contraintes décrivant son problème. Cette tâche est connue sous le nom d'acquisition de modèle dans la littérature introduite par les concepteurs de CONACQ [BCKO05].

Les approches actuelles souffrent cependant de certaines limites. Dépasser ces limites est l'objectif fixé par cette thèse. Plus modestement, la problématique et l'enjeu sont de proposer des méthodes permettant de répondre aux différentes critiques concernant l'existant, comme la nécessité de fournir des solutions de son problème pour l'utilisateur ou encore d'apprendre un modèle si trouver des solutions peut être fait par une recherche quasi-aléatoire. Nous avons choisi de développer deux axes : le premier consiste à exploiter

des solutions de problèmes proches de celui considéré et le deuxième à rechercher une solution au problème, sans connaître les contraintes, mais en rendant interactif le processus de résolution.

Apprendre à partir de solutions de problèmes proches : Si l'on peut trouver peu raisonnable de demander à l'utilisateur des solutions de son problème, il faut par contre s'interroger sur ce qui peut lui être demandé. En partant d'un exemple comme les problèmes d'emploi du temps scolaire, nous nous sommes interrogé sur ce que pourrait fournir un utilisateur pour créer un emploi du temps pour la rentrée prochaine. En faisant l'hypothèse que jusqu'à maintenant les emplois du temps étaient faits à la main, nous avons potentiellement à notre disposition des exemples assez proches de ce que nous voudrions pour la prochaine rentrée. Proches car en effet, les professeurs ne sont peut être pas les mêmes, par exemple avec des créations de postes, ou encore la fermeture d'un bâtiment avec la perte de salles. Notre idée consiste dans un premier temps à construire une modélisation pour le concept général, ici les emplois du temps, qui pourrait ensuite permettre de créer des instances particulières comme le problème d'emploi du temps de la prochaine rentrée. Inspirés par les travaux récents sur les langages de spécifications haut-niveau, nous proposons dans cette thèse une approche visant à apprendre un modèle dans un langage plus abstrait que le classique tuple composé des ensembles de variables, domaines et contraintes. Notre cadre se décompose alors en deux parties : dans un premier temps, il s'agit d'apprendre un modèle dans notre langage à partir des données fournies par l'utilisateur puis dans un second temps, créer son instance particulière en fonction des caractéristiques de son problème réel (salles et professeurs à la prochaine rentrée).

Recherche interactive Le deuxième axe de recherche développé dans cette thèse consiste à reconsidérer les motivations de l'utilisateur. Que souhaite l'utilisateur ? Si son problème est de trouver un emploi du temps pour la rentrée prochaine, apprendre un modèle de son problème n'est qu'un moyen pour trouver sa solution. En effet, si l'objectif est de trouver une solution, l'apprentissage du modèle ne devient plus qu'une étape et il n'est même pas obligatoire d'achever cette tâche. Bien entendu, l'utilisateur ne sait toujours pas utiliser la programmation par contraintes et n'arrive pas à trouver de solutions à son problème par lui-même. Il peut par contre répondre à des questions sur des affectations/propositions de solutions, complètes ou partielles. En partant de ces hypothèses, nous proposons une approche visant à une résolution interactive. En commençant la résolution avec un modèle où aucune contrainte n'est exprimée, nous posons des questions à l'utilisateur dès que notre solveur n'est plus capable de décider par lui-même. Ces questions permettent d'apprendre une partie des contraintes du problème et ainsi augmenter l'automatisation du solveur.

Au delà des problématiques orientés utilisateurs, cette thèse développe aussi certains aspects concernant l'apprentissage automatique. Même si ces techniques sont envisagées comme un outil, résoudre nos problématiques nous ont amené à trouver des compromis entre l'efficacité du modèle à apprendre en terme d'expressivité et l'efficacité de ces techniques d'apprentissage, comme la rapidité, la précision ou encore l'interaction avec l'utilisateur. Un apport connexe de cette thèse est un nouveau regard sur les données dans le cadre de la programmation logique inductive utilisé dans notre premier axe. Pour dépasser les limites pathologiques dûes à la structure de nos modèles et de nos exemples, nous proposons d'exploiter les biais de langages, communément utilisés en programmation logique inductive, pour définir un nouvel opérateur de raffinement capable, dans notre cas de dépasser les phénomènes de recherche aveugle. En structurant nos exemples en plusieurs

couches, notre approche consiste à choisir dans chacune d'elles les éléments qui décriront au mieux le problème de l'utilisateur.

Cette thèse se découpe en plusieurs parties certaines dépendantes les unes des autres comme le montre la Figure 1. Suite à cette introduction, le chapitre 2 donne les notions de base autour de la programmation par contraintes allant de la modélisation des problèmes de satisfaction de contraintes à leur résolution. Ce sera également l'occasion de présenter une partie des notations utilisées dans les chapitres suivants. Cette partie introductive apporte également des définitions formelles des problématiques traitées dans cette thèse, en faisant également une présentation des approches ayant contribué à l'essor de l'automatisation de la tâche de modelage de CSP. Après avoir présenté formellement nos problématiques et avant les réponses que nous y avons apportées, il est indispensable d'introduire les concepts de bases de l'apprentissage automatique utilisés dans cette thèse. Au chapitre 3, nous présentons les problèmes d'apprentissage comme des problèmes de recherche souvent rencontrés en intelligence artificielle : description des espaces de recherche, des états de départ et des états finaux, puis les différentes méthodes usuelles pour les parcourir. Cette partie présentera également les différentes caractéristiques que peuvent posséder de telles approches. Ensuite, nous présentons les réponses aux problématiques proposées. Le chapitre 4 présente le premier axe : description du cadre permettant l'apprentissage de modèle abstrait à partir de données de problèmes proches puis la création du modèle représentant le problème réel de l'utilisateur. Il s'agira notamment de présenter notre langage de modélisation et la manière dont il peut être reformulé pour tenir compte des données de l'utilisateur. Ce chapitre montre aussi les difficultés rencontrées dans la phase d'apprentissage. Les techniques de la littérature échouant sur nos jeux de données, nous avons dû développer une nouvelle approche répondant aux spécificités de notre cadre. Notre approche, ainsi que sa caractérisation, est développée au chapitre 5. Le second axe, exploitant l'idée d'une résolution du problème en faisant appel à l'utilisateur pour progressivement le résoudre, est quant à lui présenté au chapitre 6. Ce chapitre commence par une présentation détaillée du système CONACQ qui est ensuite étendu. Le dernier chapitre de cette thèse est naturellement une conclusion générale sur les travaux menés et les opportunités d'ouverture offertes.

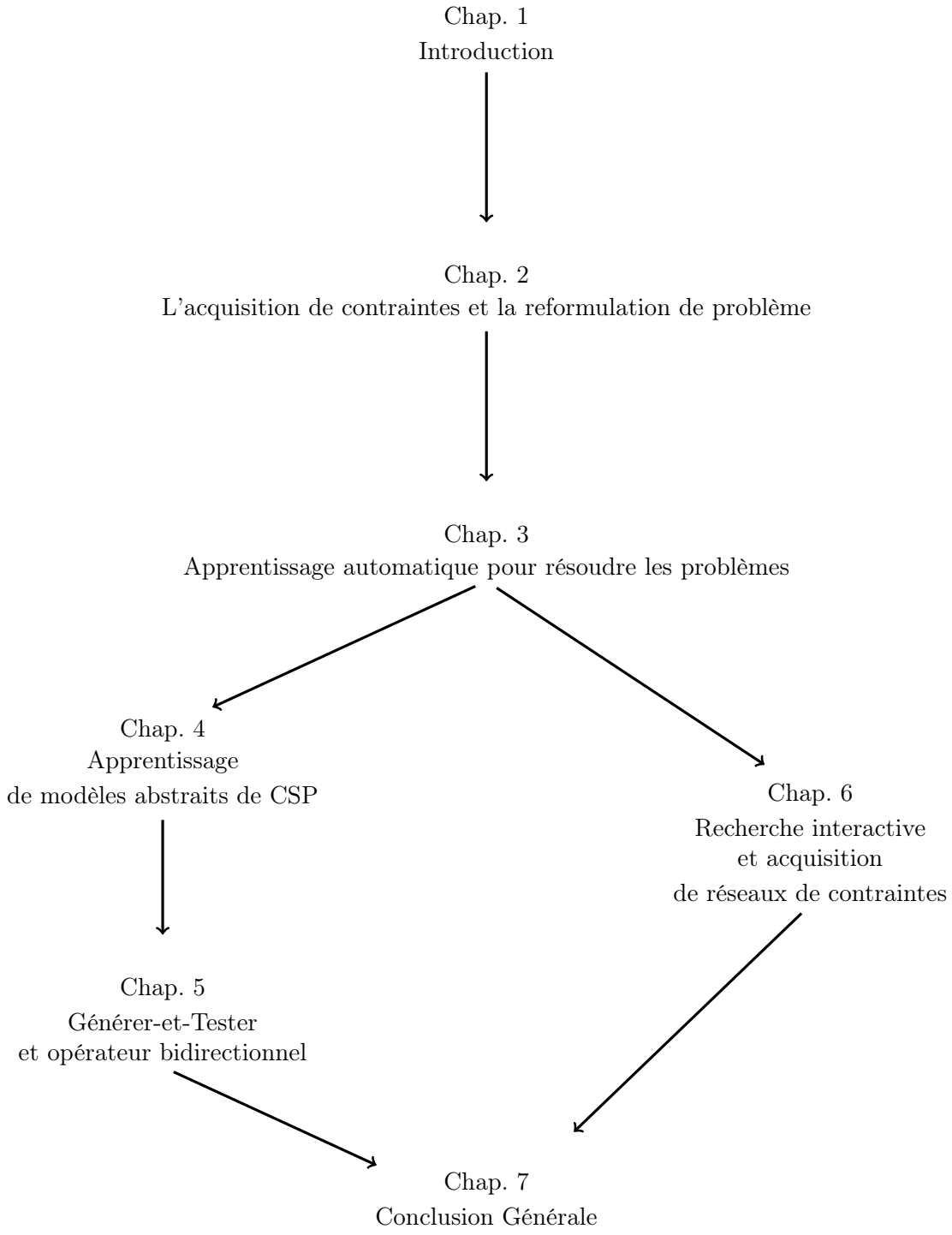

FIGURE 1.1 – Organisation des chapitres de la thèse

Chapitre 2

L'acquisition de contraintes et la reformulation de problème

2.1 Introduction

Pour que l'ordinateur résolve des problèmes, il lui faut deux choses. La première consiste en la description du problème. Si le problème est de trouver un emploi du temps, il faut que l'utilisateur humain réussisse à exprimer son besoin pour que l'ordinateur puisse le traiter. Pour cela, l'utilisateur doit utiliser un langage commun, suffisamment formel, avec l'ordinateur. Ensuite, il faut à l'ordinateur une méthode pour résoudre ce problème, et proposer une solution, voire la meilleure, à l'utilisateur.

Le sujet qui nous intéresse dans ce chapitre concerne principalement le premier besoin. Plus précisément, nous nous intéressons au formalisme des problèmes de satisfaction de contraintes (CSP) permettant de décrire un problème en exprimant ces contraintes. Reprenons l'exemple des emplois du temps. Une des contraintes concerne l'impossibilité pour un professeur d'enseigner dans deux endroits différents au même moment. Un autre exemple serait l'obligation, pour une classe d'avoir le bon enseignant avec le bon cours. En exprimant ainsi toutes les contraintes, nous obtenons une spécification représentant ce qui est un bon emploi du temps. Il reste alors à poser la question à la machine soit de trouver une solution en accord avec les contraintes, ce qui correspond au problème de *décision*, soit de trouver la meilleure solution selon un critère (par exemple, des journées peu chargée en cours), correspondant au problème d'*optimisation*. La programmation par contraintes (CP) est un paradigme permettant de concevoir, de modéliser, de tels problèmes. C'est un formalisme qui a connu un très grand succès à la fois dans la modélisation des problèmes mais également dans leur résolution avec de nombreuses applications allant des puzzles arithmétiques aux problèmes d'ordonnancement industriel. Cependant, la communauté de la CP reconnaît qu'il existe des difficultés grandissantes rencontrées dans la phase de modélisation [Pug04]. En effet, les utilisateurs débutants rencontrent des difficultés majeures pour terminer cette étape avec succès. Comment trouver les contraintes du problème ? Comment améliorer le modèle pour que sa résolution soit plus efficace ? Comment choisir les variables, c'est-à-dire les objets que l'on manipulera ? En effet, n'importe qui peut expliquer ce qu'est un emploi du temps, en revanche l'exprimer pour que la machine le résolve est plus complexe. Cela entraîne un ralentissement du développement du domaine hors du cadre de la recherche avec de nombreuses avancées scientifiques peu exploitées dans l'industrie. Pour répondre à ce problème, de nombreux travaux ont étudié la compréhension[Smi06] et l'automatisation[BCKO06] de la modélisation.

Dans ce chapitre, nous explorerons les approches sur l'automatisation de la modélisation

et l'aide automatique à la modélisation puis introduirons les deux problématiques étudiées dans cette thèse. Ainsi une première section décrira précisément ce qu'est un CSP et les différents éléments que l'utilisateur doit fournir. Puis, nous présenterons, dans un premier temps, les approches qui ont visé à la synthèse de tels problèmes, et dans un second, celles permettant d'améliorer des modèles existant. Enfin, nous finirons avec les deux nouvelles problématiques de ce cadre introduites dans cette thèse.

2.2 Problèmes de satisfaction de contraintes

La programmation par contraintes [Mon74, BCDP07, RBW06, Apt99] permet donc de modéliser un problème, de manière formelle, en décrivant les contraintes qui le caractérisent. Ces contraintes pourront également être exploitées pendant la recherche de solutions. Elles ont également permis à cette technologie de connaître de nombreux succès.

Nous définissons formellement dans la suite un problème de satisfaction de contraintes. Ensuite, nous introduisons des éléments pour comprendre comment ces problèmes peuvent être résolus.

2.2.1 Définition

Pour définir un problème avec cette méthode, il faut commencer par choisir les variables, c'est à dire les objets pour lesquels l'utilisateur souhaite trouver une valeur afin de construire une solution à son problème. Dans l'exemple de l'emploi du temps, les variables pourraient être les affectations de salles de chaque cours et les horaires où elles seront occupées. Une variable doit posséder un domaine, c'est-à-dire un ensemble de valeurs qu'elle peut prendre, les salles des bâtiments et les créneaux d'occupation possibles dans l'exemple. Ensuite, une contrainte permettra de restreindre les valeurs que pourront prendre un sous-ensemble de variables (deux salles doivent être différentes si elles sont utilisées par deux cours différents au même moment). Tout cela formera le modèle du problème, nommé dans le cadre de la programmation par contraintes un problème de satisfaction de contraintes (CSP). On parlera également de réseaux de contraintes. Les variables représentant les nœuds de ce réseau et les contraintes les liens entres ces nœuds. Il est à noter qu'un CSP représente un problème de décision. Si l'on souhaite modéliser un problème d'optimisation il faut ajouter une mesure, un critère permettant de comparer deux solutions entre elles. On parlera alors de problème d'optimisation de contraintes (COP).

Plus formellement, un CSP est un tuple $\langle X, D, C\rangle$, où X est un ensemble fini de variables, D un ensemble fini de domaines, et C un ensemble fini de contraintes. Un domaine D_i de D est associé à chaque variable X_i de X. Une contrainte est un couple $\langle S, R\rangle$, où S est la portée de la contrainte, c'est-à-dire un sous-ensemble de variables de X, et R est une relation d'arité $|S|$ sur les constantes de D.

Pour illustrer, considérons l'exemple suivant. Il y a trois variables X, Y et Z et leurs domaines sont constitués des mêmes valeurs $\{1, 2, 3\}$. Les contraintes du problème sont $X < Y$, $X \neq Z$ et $Y \neq Z$.

Nous dirons qu'une contrainte est satisfaite s'il existe une substitution θ des variables de S vers les constantes telles que $\theta(S)$ appartienne à R. Par exemple, $\{X/1, Y/2\}$ correspond à la substitution de X par 1 et Y par 2 et permet de satisfaire la contrainte $X < Y$ alors que $\{X/3, Y/2\}$ ne le permet pas. Une substitution σ des variables de X dans les valeurs respectives de D est une solution du CSP si toutes les contraintes sont satisfaites. On parlera de non-solutions quand une substitution σ ne satisfait pas toutes les contraintes. En d'autres termes, chaque substitution X_i/v_i doit vérifier que $v_i \in D_i$ et pour toute

contrainte $\langle S, R\rangle$, $\sigma(S)$ doit appartenir à R. Ainsi, une solution du problème de l'exemple pourrait être : $X = 1$, $Y = 2$, $Z = 3$.

2.2.2 Résolution

La résolution d'un problème de satisfaction de contraintes peut être vue comme un problème de recherche classique en intelligence artificielle. Un problème de recherche se caractérise par un espace de recherche représentant les différentes configurations possibles avec parmi elles, un état initial servant de point de départ à la recherche et la description des états finaux représentant les solutions du problème. Reste ensuite à définir un parcours de cet espace tel qu'il permette d'arriver rapidement à une solution afin d'éviter un parcours exhaustif, avec souvent un coût prohibitif, de l'espace.

Dans cette section, nous ne nous intéressons qu'à une catégorie de résolution : globale et structurée par un arbre de recherche. Dans le cas des CSP, les états de l'espace sont représentés par une affectation, partielle ou totale, des variables à des valeurs. Ainsi, l'état initial est celui où aucune variable n'a de valeur et les états finaux correspondent aux affectations complètes, solutions du problème telles que définies précédemment. Le principe de la recherche va consister à construire progressivement une solution en affectant des valeurs à des variables et en vérifiant la satisfaction des contraintes. Ces dernières seront également utilisées pour restreindre les choix d'affectations possibles. En effet, la résolution d'un CSP peut être organisée sous la forme d'un arbre où l'on alterne les phases d'élagage en utilisant les contraintes, étapes appelées propagation, et les phases choisissant une valeur pour une variables, appelées branchement.

La propagation permet, étant donné une affectation partielle, de restreindre les domaines des variables et ainsi d'éviter d'énumérer toutes les affectations. Par exemple, en partant de l'état initial où aucune variables n'a d'affectation, la contrainte $X < Y$ permet d'éliminer la valeur 3 du domaine de X puisqu'il n'y aura aucune solution avec $X = 3$. En procédant ainsi jusqu'à un point fixe, la phase de propagation va éliminer les valeurs n'appartenant à aucune solution. Après la phase initiale de propagation, on obtient comme domaine pour X, $\{1, 2\}$, pour Y $\{2, 3\}$ et celui de Z reste inchangé à $\{1, 2, 3\}$.

Ensuite pour pouvoir continuer l'exploration de l'espace de recherche, il faut choisir une affectation pour une variable, étape appelée un branchement. Ce choix est généralement guidé par une heuristique qui décidera quelles seront les affectations à explorer en priorité. Ainsi une nouvelle étape de propagation pourra élaguer à nouveau certaines branches de l'arbre de recherche. Prenons par exemple $X = 2$. Si on recommence l'étape de propagation, on obtient une solution du problème qui est $X = 2$, $Y = 3$ et $Z = 1$. Cependant, dans le cas général il faut répéter plusieurs fois l'alternance branchement/propagation.

La stratégie ainsi décrite correspond à une exploration en profondeur de l'arbre de recherche. La figure 2.1 présente l'arbre de recherche permettant d'atteindre toutes les solutions. Chaque branche représente une affectation (branchement) et le résultat de la propagation est donné dans le tableau de la figure 2.2. D'autres méthodes d'exploration existent, notamment pour répondre aux problèmes d'optimisation tel que le *Branch-and-bound* ou encore le *Restart*.

2.3 Synthèse de CSP et aide à la modélisation

Dans cette partie, nous présentons les travaux et approches existants dans les deux domaines que sont la synthèse de CSP, plutôt nommée dans la littérature « acquisition

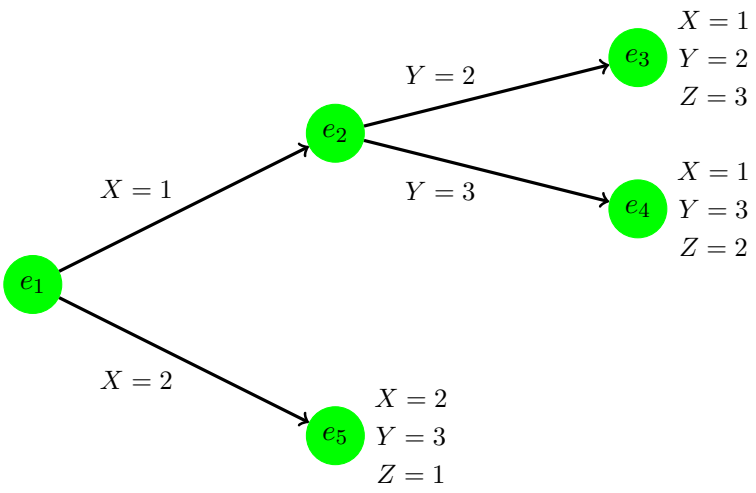

FIGURE 2.1 – Arbre de recherche des solutions d'un CSP

Nœud	D_X	D_Y	D_Z
e_1	$\{1, 2\}$	$\{2, 3\}$	$\{1, 2, 3\}$
e_2	$\{1\}$	$\{2, 3\}$	$\{1, 2, 3\}$
e_3	$\{1\}$	$\{2\}$	$\{3\}$
e_4	$\{1\}$	$\{3\}$	$\{2\}$
e_5	$\{2\}$	$\{3\}$	$\{1\}$

FIGURE 2.2 – États des domaines pour l'exemple à chaque nœud de recherche

de réseaux de contraintes », et l'aide à la modélisation, permettant d'aider l'utilisateur à améliorer son modèle.

2.3.1 Acquisition de réseaux de contraintes

Parmi les pistes explorées pour faciliter la modélisation des CSP, l'une étudie la possibilité d'acquérir, de construire le réseaux de contraintes à partir de solutions et de non-solutions du problème cible. Plus formellement :

Définition 2.3.1 (Problème d'acquisition de réseaux de contraintes)**.** *Étant donnés un CSP $\langle X, D, C\rangle$ où C est inconnu, une bibliothèque de contraintes C_X sur X, un ensemble de solutions E^+ et un autre ensemble de non-solutions E^- de ce problème, trouver C tel que :*

- *$C \subseteq C_X$;*
- *pour tout $\sigma \in E^+$, σ est une solution satisfaisant C ;*
- *pour tout $\sigma \in E^-$, σ ne satisfait pas C ;*

À notre connaissance, les seuls à avoir proposé une solution à ce problème sont [CBO+03, BCFO04, BCKO06, BCKO05] avec le système CONACQ. Ce système sera détaillé dans le chapitre 6.

La limite principale présentée par ce cadre est le besoin pour l'utilisateur de fournir des solutions et non-solutions de son problème. En effet, si l'utilisateur a à sa disposition des solutions de son problème, on peut s'interroger sur sa nécessité d'obtenir un modèle pour

son problème. Il peut cependant en avoir besoin pour passer à un problème d'optimisation et trouver la meilleure solution de son problème.

Pour passer au delà de ces limites, les auteurs proposent de faire évoluer la définition du problème d'acquisition de contraintes en utilisant des requêtes. Des requêtes sont des questions que le système d'acquisition peut poser à l'utilisateur afin de progresser dans son travail de conception. Ces requêtes prennent la forme d'affectation des variables du CSP et l'utilisateur devra uniquement répondre si elles correspondent à des solutions ou non-solutions du problème cible. La définition du problème d'acquisition devient alors :

Définition 2.3.2 (Problème d'acquisition de réseaux de contraintes avec requêtes)**.** *Étant donnés un CSP $\langle X, D, C\rangle$ où C est inconnu, une bibliothèque de contraintes C_X sur X, un oracle capable de déterminer si des affectations sont des solutions ou non du problème, trouver C tel que $C \subseteq C_X$ et C correspond aux contraintes du problème.*

Ainsi, en plus de reconstruire l'ensemble de contraintes, il faut également générer des requêtes pertinentes permettant au système d'avancer et surtout de converger. La version dirigée par les requêtes de CONACQ[BCOP07] répond à ces besoins. En effet, CONACQ est basé sur l'espace des versions[Mit97] décrit au chapitre 3, qui permet d'être sûr que le réseau de contraintes obtenu est celui représentant le problème et qu'aucune nouvelle requête n'est nécessaire.

Cependant cette dernière version possède également ses limites. Effectivement, pour pouvoir avancer dans l'acquisition, il lui faut générer des solutions du problème. Or, pour certains problèmes avec peu de solutions cela devient vite très compliqué.

Finalement, nous pouvons nous interroger sur la réelle motivation de ces approches, à quel besoin de l'utilisateur cela doit répondre. Cette thèse explorera deux manières de reconsidérer ces limites. Elles seront introduites à la fin de ce chapitre.

D'une manière plus large, nous évoquons également les travaux[BQR06] visant à acquérir un point de vue, c'est-à-dire l'ensemble X des variables et D des domaines. Ces travaux préliminaires explorent la possibilité, à partir de données historiques, de proposer à l'utilisateur différents ensembles possibles de variables de modélisation. Nous pouvons également citer les travaux autour de la découverte scientifique comme les systèmes Bacon[Lan77, LSBZ87, LSB87], Lagramge[TD97], ou encore Mechem[VP90]. Ces approches visent à modéliser des systèmes physiques ou chimiques grâce à des équations. En partant de mesures sur des variables (et dans certain cas d'un modèle incomplet), il vont chercher à construire un système d'équations permettant d'expliquer ces mesures en utilisant des opérateurs arithmétiques classiques ou comme c'est le cas dans Lagramge, avec des fonctions spécifiques aux domaines étudiés, définies par l'utilisateur. Une caractéristique de ces systèmes est également leur utilisation d'une manière interactive (voir par exemple [BSL02, TD01, VP90]) où l'utilisateur peut refuser un modèle proposé et demander à ce que le processus de découverte reprenne en tenant compte de ce refus. Ce processus interactif peut être comparé avec la problématique que nous proposons à la fin de ce chapitre (section 2.5) à la différence que nous ne proposons pas à l'utilisateur de valider un modèle mais de valider des affectations (partielles) des variables du CSP. Pour finir, les travaux de Page et Frisch[jF92] sur l'apprentissage de clauses contraintes visent à apprendre des contraintes sur les variables d'un atome. Pour cela, ils proposent un ordre de généralité, basé sur l'implication logique des deux modèles, permettant de comparer deux clauses contraintes entre elles, étant donnée une *théorie de contraintes*, c'est-à-dire une théorie du premier ordre définissant les prédicats/contraintes pouvant être utilisés.

2.3.2 Reformulation de problèmes

Une autre piste a été d'améliorer l'ensemble des contraintes afin de rendre la résolution du problème plus efficace. Effectivement, nous avons pu voir dans la section 2.2.2, que les contraintes jouent un rôle non négligeable dans la résolution. Pour un problème donné, plusieurs modélisations équivalentes sémantiquement peuvent donner des résultats très différents en terme d'efficacité. On peut citer aussi l'ajout de contraintes dites redondantes, car elles ne changent pas la sémantique du problème, mais permettent d'améliorer l'efficacité de la résolution grâce à des algorithmes de propagation efficace de ces contraintes. Un autre exemple est l'ajout de contraintes pour casser des *symétries*, c'est-à-dire des branches de l'arbre de recherche amenant à des solutions équivalentes. Ainsi un choix judicieux des contraintes utilisées peut amener à un meilleur modèle. Également, l'utilisation de contraintes globales[BCDP07, BH03], contraintes permettant une propagation globale d'un ensemble de variables avec un algorithme efficace, permet à la fois de simplifier l'écriture du modèle mais également de le rendre plus efficace.

Ce travail de reformulation du problème reste cependant très compliqué pour un débutant en CP et comme la modélisation, cette étape a fait l'objet d'études afin de l'automatiser. Nous pouvons citer les travaux de [CM01, CCM06] qui exploitent le système HR[Col98, CBW99] de découvertes mathématiques. En générant des solutions avec un modèle existant, ils cherchent à trouver des contraintes impliquées par celles existantes (et donc redondantes). L'ajout automatique de contraintes globales[BCP05, BCP07] a également fait l'objet d'intérêt. Il propose d'ajouter au modèle des contraintes globales à paramètres, afin de les rendre plus efficace. Il propose par exemple un cadre générique pour apprendre les paramètres de contraintes du genre NVALUE ou GCC[BCDP07].

2.4 Apprentissage de modèles abstraits de CSP

La première problématique étudiée dans cette thèse concerne la possibilité de fournir des solutions de problèmes proches. Nous disons qu'un CSP $\langle X_1, D_1, C_1\rangle$ est proche d'un autre $\langle X_2, D_2, C_2\rangle$, si malgré des ensembles de variables X_1 et X_2 et des ensembles de domaines D_1 et D_2 différents, le problème de fond exprimé par les contraintes reste le même.

Avec une approche comme CONACQ, les exemples sont des affectations des variables du problème cible. Or, nous avons soulevé la difficulté pour un utilisateur de fournir des solutions pour son problème. Cependant, il peut avoir à sa disposition des solutions pour des problèmes proches. Prenons l'exemple des emplois du temps, l'utilisateur veut obtenir une solution pour l'emploi du temps de l'année en cours. Jusqu'à maintenant il procédait manuellement et possède donc des données historiques correspondant aux emplois du temps des années précédentes. Ses variables et domaines sont différents mais le problème de fond reste le même. Il souhaite automatiser cette tâche de création. Pour cela, il serait intéressant d'exploiter ces données qui correspondent à des problèmes proches de ceux qu'il veut résoudre. En effet, le nombre de professeurs, de salles, les créneaux pour placer les cours, ont pu changer d'une année à l'autre. Par conséquent, même si la nature du problème reste la même (« trouver un emploi du temps »), l'ensemble des variables, des domaines du CSP correspondant sont différents. Ces données ne peuvent être exploitées avec une approche de type CONACQ, à cause de la variation des variables entre les solutions. Nous proposons un problème dérivé de l'acquisition de problèmes, proche de celui de l'acquisition de réseaux de contraintes.

Définition 2.4.1 (Problème d'acquisition de réseaux de contraintes avec exemples proches)**.** *Étant donnés un CSP $\langle X, D, C\rangle$ où C est inconnu, une bibliothèque de type de contraintes P, un ensemble de solutions E^+ et un autre de non-solutions E^- de problèmes proches, c'est-à-dire de problèmes où les ensembles de variables et de domaines sont différents, trouver C tel que $C \subseteq C_X$ et C correspond aux contraintes du problème.*

Pour cela, nous proposons de chercher un modèle suffisamment abstrait pour représenter toutes les solutions et non-solutions avant de l'« instancier » pour représenter le problème cible. Le cadre développé pendant cette thèse est présenté au chapitre 4. Comme discuté dans le chapitre 4, de nouvelles problématiques liées aux difficultés rencontrées avec les techniques d'apprentissage automatique, ont poussé au développement d'une nouvelle technique d'apprentissage pour répondre aux besoins de notre problème. Cette nouvelle approche est expliquée au chapitre 5.

2.5 Recherche Interactive

Dans un second temps, nous nous proposons de reprendre le besoin de l'utilisateur : « trouver une solution ». Pour cela, il n'a pas forcément de données historiques et par conséquent la problématique précédente ne s'applique pas. Nous avons vu dans la section 2.3.1 que l'utilisation de requêtes générées par le système d'acquisition du réseau pouvait permettre de répondre à ce besoin. Cependant pour être efficace, ce générateur doit produire des solutions du problème, ce qui est justement ce que souhaite obtenir notre utilisateur. Si les solutions sont faciles à générer, c'est-à-dire que la probabilité de construire une requête solution du problème est élevée, l'utilisateur n'a pas besoin de l'ordinateur pour résoudre son problème (à part dans le cas où il voudrait optimiser sa solution). On peut supposer que s'il souhaite faire appel à l'ordinateur, c'est qu'il ne trouve pas de solutions, ces dernières étant peu nombreuses. La recherche interactive va viser à obtenir une solution au problème de l'utilisateur sans connaître au préalable les contraintes spécifiant son problème et en demandant à l'utilisateur d'étiqueter des affectations, complètes ou partielles, des variables. Si une affectation partielle ne satisfait pas une contrainte l'utilisateur répondra « non », et « oui » sinon.

Définition 2.5.1 (Recherche interactive)**.** *Étant donnés un CSP $\langle X, D, C\rangle$ où C est inconnu, une bibliothèque de contraintes C_X sur X, un oracle capable de déterminer si des affectations, partielles ou complètes, sont des solutions ou non du problème, trouver une solution au problème telle que les contraintes $C \subseteq C_X$ soient satisfaites.*

La recherche interactive va donc chercher une solution en devinant progressivement les contraintes décrivant le problème à la manière du problème d'acquisition de réseaux de contraintes. Nous proposons une approche de résolution, biaisée par l'arbre de recherche décrit à la section 2.2.2, où pour chaque nœud de l'arbre, une requête sera posée à l'utilisateur si le système ne peut décider seul. Cette approche est décrite au chapitre 6. De plus, nous discuterons de son extension au problème d'acquisition de réseaux de contraintes auquel il peut se réduire.

Chapitre 3

Apprentissage automatique pour résoudre les problèmes

3.1 Introduction

Les problématiques qui nous intéressent, présentées au chapitre précédent, ont comme point commun de vouloir détecter automatiquement les contraintes d'un problème cible. Pour cela, nous avons à notre disposition soit des solutions/non-solutions de problèmes proches, soit la possibilité de générer des requêtes représentant des solutions/non-solutions partielles du problème cible. Ce cadre est celui des problèmes d'apprentissage automatique et par conséquent, celui utilisé tout au long de cette thèse.

En effet, l'apprentissage automatique vise à définir un concept à partir d'observations, des exemples, afin de pouvoir les classer. Une manière simple est de définir le concept comme l'ensemble des exemples appartenant à ce concept. Cependant cette définition est trop spécifique et ne permet pas de classer un nouvel exemple. Le but en apprentissage est donc de généraliser cette définition afin de pouvoir prédire si de futurs exemples appartiennent ou non au concept. Dans notre cas, le concept est le CSP ou un modèle plus abstrait de CSP représentant l'ensemble de ses solutions. Les exemples sont les solutions et non-solutions décrites plus haut. Nous ne nous intéressons qu'à l'apprentissage supervisé, où les classes des exemples sont connues, c'est-à-dire que l'on sait pour chacun de nos exemples s'il est une solution ou non. De même, nous ne considérerons que les données symboliques qui s'opposent aux données numériques. Ce choix s'explique par le fait que nous ne considérons que des variables de CSP à domaine fini.

Afin d'illustrer le principe de l'apprentissage, considérons l'exemple suivant, inspiré du problème des trains de Michalski. Dans cet exemple, chaque train est composé de voitures. Chaque voiture possède différentes caractéristiques telles que son nombre de roues et sa taille. Une voiture transporte des objets géométriques, décrits par leur forme et leur quantité. Le train t_1^+ de la Figure 3.1 représente un exemple que nous considérerons comme instance d'un concept général, une propriété caractérisant ce type de train. Toujours sur la même figure, le train t_2^-, quant à lui, ne respecte pas ce concept. Nous parlons alors d'un exemple positif pour t_1^+ et d'un exemple négatif pour t_2^-. L'objectif fixé par l'apprentissage est de trouver une propriété permettant de différencier ces deux trains. Cette propriété doit être suffisamment générale pour représenter d'autres exemples positifs que t_1^+ mais suffisamment spécifique pour ne pas être vérifiée par t_2^- voire d'autres exemples négatifs.

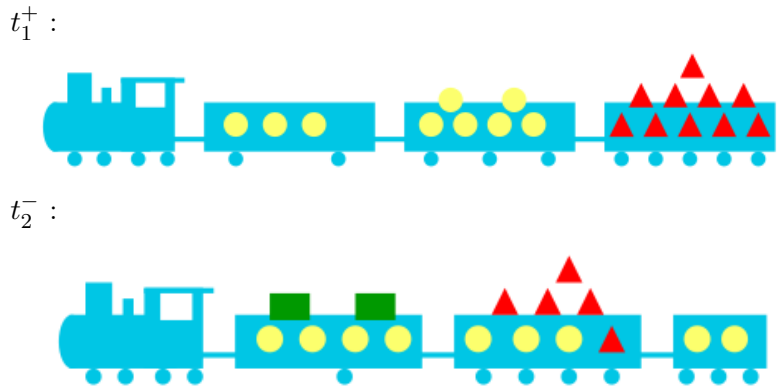

FIGURE 3.1 – Exemples de trains, instance et non-instance d'un concept

Ce chapitre a pour but de présenter les techniques de l'apprentissage automatique utilisées pour résoudre nos problématiques. Après avoir défini ce problème comme un problème de recherche classique d'Intelligence Artificielle, nous nous intéresserons aux opérateurs de raffinement. Ces opérateurs servent pour identifier dans l'espace de recherche, les nœuds pouvant être explorés après celui en cours. Nous considérerons deux types d'opérateurs : ceux raffinant une définition, en la généralisant ou en la spécialisant, et ceux raffinant deux définitions afin que le raffinement produit soit un couple de définitions plus « proches » dans l'espace de recherche. Pour finir, nous présenterons les différentes stratégies exploitant les opérateurs utilisées dans les chapitres suivants.

3.2 Apprentissage automatique vu comme un problème de recherche

3.2.1 Le langage des exemples

Avant de pouvoir définir les problèmes d'apprentissage, il est important de formaliser la représentation des exemples, les principales données en entrée du problème. Premièrement, il existe plusieurs types de langages, plus ou moins expressifs. Nous décrirons dans cette section principalement un sous-ensemble de la logique du premier ordre utilisée en programmation logique inductive(ILP)[NCW97, Rae96] et notamment aux chapitres 4 et 5.

Un *langage en logique du premier ordre* est composé d'un ensemble de *constantes*, d'un ensemble de *variables* et d'un ensemble de *prédicats*. Un prédicat peut être vu comme une fonction booléenne d'arité fixe. Un *terme* est soit une constante soit une variable. Un *atome* est une expression de la forme $p(t_1, t_2, \ldots, t_k)$ où p est un prédicat d'arité k et t_1, t_2, …, t_k sont des termes. On parlera d'*atome clos* quand t_1, t_2, …, t_k sont des constantes. Un *littéral* est soit un atome soit la négation d'un atome. Nous utiliserons dans la suite les notations pour les symboles logiques suivantes :

- $\wedge$ et $\vee$ pour la conjonction et la disjonction ;
- $\exists$ et $\forall$ pour les quantifieurs existentiels et universels ;
- $\rightarrow$ pour l'implication ;
- $\neg$ pour la négation.

Une *clause* est une disjonction finie de littéraux. Elle est parfois représentée par une implication séparant d'un côté les littéraux positifs de la disjonction des négatifs. Par exemple, la clause :

$$h_1 \vee h_2 \vee h_3 \vee \neg b_1 \vee \neg b_2$$

où h_1, h_2, h_3, b_1 et b_2 sont des atomes, peut être représentée également par

$$h_1 \vee h_2 \vee h_3 \leftarrow b_1 \wedge b_2$$

$h_1 \vee h_2 \vee h_3$ est la tête de la clause et $b_1 \wedge b_2$ son corps. Une *clause de Horn* est une clause dont au plus un littéral est positif, c'est-à-dire qu'il n'y a qu'un seul atome dans la tête de la clause. Nous parlerons de clauses conjonctives pour une conjonction de littéraux.

Pour finir, une formule logique est dite en *forme normale conjonctive* (CNF) si c'est une conjonction de clauses et en *forme normale disjonctive* (DNF) si c'est une disjonction de conjonctions.

Un *langage propositionnel* est assez proche de la logique du premier ordre. La différence réside dans son vocabulaire où les ensemble de termes et prédicats sont remplacés par un unique ensemble de variables propositionnelles. Ce type de langage sera utilisé au chapitre 6.

Nous pouvons maintenant décrire les exemples. Il existe communément deux manières de voir la description des exemples. La première, nommée *interprétation*, consiste à représenter un exemple comme un ensemble fini d'atomes clos. La seconde, nommée *par implication* (*entailment* en anglais), consiste à uniquement voir un exemple comme un atome clos où le prédicat de cet atome représente le concept que l'on souhaite trouver. La description de cet exemple pourra alors être retrouvée dans une base de connaissances.

Définition 3.2.1 (Base de connaissances)**.** *En ILP, une base de connaissances est un ensemble de clauses de Horn.*

Dans une telle base, nous pourrons à la fois trouver des descriptions des exemples dans la représentation par implication mais également des connaissances communes sur les exemples comme des prédicats utilisables dans le langage des hypothèses (voir section 3.2.2).

Nous proposons d'illustrer ces deux approches sur l'exemple des trains présentés dans l'introduction.

Exemple

Les deux trains (voir Figure 3.1) correspondant à un exemple positif t_1^+ et un exemple négatif t_2^-, pour le concept goodtrain peuvent être décrits par interprétation de la manière suivante :

$$\begin{aligned} t_1^+ : \{&has\_car(t_1^+, c_1), has\_car(t_1^+, c_2), has\_car(t_1^+, c_3), wheels(c_1, 2),\\ &wheels(c_2, 3), wheels(c_3, 5), long(c_1), long(c_2), long(c_3),\\ &load(c_1, circle, 3), load(c_2, circle, 6), load(c_3, triangle, 10)\} \end{aligned}$$

$$\begin{aligned} t_2^- : \{&has\_car(t_2^-, c_1), has\_car(t_2^-, c_2), has\_car(t_2^-, c_3), wheels(c_1, 1),\\ &wheels(c_2, 4), wheels(c_3, 3), long(c_1), long(c_2), short(c_3),\\ &load(c_1, circle, 4), load(c_1, rectangle, 2), load(c_2, triangle, 5)\\ &load(c_2, circle, 3), load(c_3, circle, 2)\} \end{aligned}$$

On peut représenter ces deux exemples par implication avec un seul atome clos : $goodtrain(t_1^+)$ et $goodtrain(t_2^-)$. Dans ce cas, les littéraux précédents (ceux des interprétations) seront placés dans la base de connaissances afin de pouvoir être liés à ces exemples comme nous le verrons par la suite.

Une fois les exemples représentés, l'objectif d'un problème d'apprentissage sera de dégager des propriétés séparant les exemples positifs des exemples négatifs et représentant le concept cible. Nous noterons dans la suite, $\mathcal{L}_e$ le langage des exemples, E^+ l'ensemble des exemples positifs et E^- celui contenant les exemples négatifs. Quand cela sera nécessaire nous représenterons par B la base de connaissances.

3.2.2 Le langage des hypothèses

Le langage d'hypothèse, noté $\mathcal{L}_h$ dans la suite, est très important puisqu'il décrit les propriétés pouvant être exprimées sur les exemples.

Dans le cadre de l'ILP considéré dans cette thèse, le langage des hypothèses est constitué de règles, c'est-à-dire des clauses de Horn. La tête de la règle est un atome non-clos utilisant le prédicat du concept cible (*goodtrain* dans l'exemple des trains). Le corps sera une conjonction de littéraux permettant d'exprimer une propriété.

Exemple (Suite)

En reprenant l'exemple des trains précédents, on pourrait également imaginer avoir à notre disposition dans la base de connaissances pour le langage d'hypothèses, le prédicat $\neq$ comparant les voitures ou les formes géométriques, et le prédicat $<$ comparant soit le nombre de roues, soit le nombre d'objets.

La règle suivante représente les trains ayant au moins deux wagons.

$goodtrain(T) : -has\_car(T, C_1), has\_car(T, C_2), C_1 \neq C_2,$

La suivante décrit les trains transportant des rectangles.

$goodtrain(T) : -has\_car(T, C_1), load(C_1, rectangle, L)$

Ces deux règles ne sont pas des règles décrivant le concept cible.

Dans le chapitre 6 où nous utiliserons un apprentissage propositionnel particulier pour apprendre un CSP, les hypothèses représentent un ensemble de contraintes modélisé par une conjonction de variables propositionnelles décrivant la présence ou l'absence des contraintes.

Afin de « juger » de la qualité des hypothèses, une opération permettant de savoir si un exemple e vérifie une hypothèse h est nécessaire. On parle d'un test de couverture.

Définition 3.2.2 (Test de couverture)**.** *Étant donnés une hypothèse h et un exemple e, nous dirons que h couvre e, noté $h \succeq e$, si la propriété h est vérifiée par l'exemple e.*

En ILP, nous dirons qu'une hypothèse h, une clause de Horn, couvre un exemple e, si e est une instance de h. Autrement dit, dans le cas où les exemples sont représentés par des interprétations, nous dirons qu'une hypothèse h couvre un exemple e, s'il existe un modèle pour h dans l'ensemble de littéraux e : $h \succeq e \Leftrightarrow e \models h$. Dans le cas où une hypothèse ne couvre pas un exemple, nous dirons qu'elle le rejette.

Bien entendu un langage riche permettra de définir une plus grande classe de concept. Mais en contre-partie, le problème deviendra plus complexe à résoudre. En effet, la taille de l'espace de recherche peut devenir importante et les tests de couvertures compliqués. Ainsi, si le test de couverture en propositionnel reste simple, il devient NP-difficile en ILP.

Quelque soit le langage utilisé, il faut l'organiser afin de structurer la recherche. Il faut notamment définir un ordre de généralité permettant d'identifier les hypothèses plus générales que d'autres. Ainsi, la propriété disant qu'un bon train doit avoir au moins deux wagons est plus générale que celle disant qu'un bon train à un wagon avec strictement plus de roues qu'un autre de ses wagons. En effet, dans la deuxième règle donné en exemple, il est sous-entendu que le train doit avoir deux wagons.

Définition 3.2.3 (Ordre de généralité)**.** *Une hypothèse h_1 est plus générale qu'une autre h_2, noté $h_1 \geq h_2$ par rapport à un ensemble d'exemples E si :*

$$\{e \mid \ h_2 \succeq e \ \wedge \ e \in E\} \subseteq \{e \mid \ h_1 \succeq e \ \wedge \ e \in E\}$$

En ILP, l'ordre de généralité entre deux hypothèses h_1 et h_2 est donc représentée par l'implication logique :

$$E \cup B \models h_1 \rightarrow h_2$$

où B est la base de connaissances potentielle. L'implication logique étant indécidable entre clauses de Horn, [Plo70] a proposé un test moins fort connu sous le nom de θ-subsomption.

Définition 3.2.4 (θ-subsomption)**.** *Étant données deux clauses C et D, C θ-subsume D s'il existe une substitution θ des variables de C telle que $\theta(C) \subseteq D$.*

Ce test étant NP-difficile, il représente une des limites de l'ILP. La θ-subsomption peut être à la fois utilisée pour ordonner le langage des hypothèses et pour effectuer le test de couverture sur le langage d'exemples liant ainsi ces deux langages. Ce cas apparaît quand le langage des exemples est un sous-ensemble du langage des hypothèses. Cet ordre permet également de structurer l'espace de recherche (par exemple sous la forme d'un treillis) et ainsi de rendre certaines approches plus efficaces.

Les difficultés rencontrées avec plusieurs langages en ILP concernent la taille potentiellement infinie de l'espace de recherche. Ainsi la communauté s'est très vite intéressée au moyen de limiter cette taille. Ces limites sont appelées des biais. D'après [NRA+96], les biais se divisent en trois catégories :

1. les biais de langage permettant de limiter le nombre d'hypothèses représentables par un langage d'hypothèses ;
2. les biais de recherche décrivant les hypothèses qui peuvent être testées par l'algorithme d'apprentissage et l'ordre dans lequel elles seront testées ;
3. les biais de validation représentant les critères d'arrêt de la recherche.

Seule la première catégorie nous intéresse dans cette section. Les autres seront évoquées dans la suite de ce chapitre. Les biais de langage sont assez nombreux. Allant de restrictions assez simples comme limiter le nombre de variables existentielles ou de littéraux utilisés dans les hypothèses à l'utilisation obligatoire des variables présentes dans la tête de la règle. Parmi les biais de langage, les modes des prédicats nous intéresseront particulièrement dans le chapitre 5. Ces modes indiquent pour un prédicat p, ses entrées et ses sorties. Ainsi le prédicat *sum* d'arité trois, avec la sémantique $sum(X, Y, Z) \Leftrightarrow X + Y = Z$, pourrait avoir le mode $sum(+, +, -)$, le $'+'$ indiquant que les deux premiers termes sont des entrées et le $'-'$ que le troisième est une sortie. De la même manière, nous pouvons également typer les termes autorisés pour utiliser un prédicat. Dans le cas de *sum*, on pourrait n'autoriser que les variables et constantes représentant des entiers par exemple. Il existe également des biais plus complexes, non considérés dans cette thèse, comme les clauses interdites où l'utilisateur donne des clauses qu'il faut retirer de l'ensemble des hypothèses ou encore les conjonctions interdites comme les tautologies ou les conjonctions d'atomes toujours fausses.

Une autre manière de délimiter l'espace de recherche (voir de le restreindre) est de le « propositionnaliser ». Cette méthode consiste à ramener un problème d'ILP à un problème d'apprentissage propositionnel en fixant avant la phase d'apprentissage un ensemble fini de littéraux pouvant être utilisés. Le système LINUS[LDG91] va ainsi générer tous les littéraux possibles étant donnés un ensemble de variables et un autre de prédicats. Il utilise ensuite des techniques habituelles d'apprentissage propositionnel pour construire les règles en indiquant pour chaque littéraux s'il fait partie de la règle ou non. Ce type de propositionnalisation est utilisé au chapitre 6 où nous construisons toutes les contraintes possibles étant donnés un ensemble de variables et des types de contraintes.

Nous verrons également au chapitre 5, l'utilisation d'un opérateur appelé la *saturation*[Rou92, Rou90] permettant de biaiser le langage des hypothèses avec un exemple.

3.2.3 Le problème

Comme pour la résolution des CSP, un problème d'apprentissage peut être vu comme un problème de recherche classique en Intelligence Artificielle [Mit82]. Pour cela il suffit de définir :

- l'espace de recherche, un ensemble d'états représentant les hypothèses (par ex. le langage d'hypothèses) ;
- un état initial, c'est-à-dire le point de départ de la recherche ;
- les états finaux, une description des états représentant des solutions du problème.

Une fois ces éléments définis, les algorithmes habituels d'exploration des espaces de recherche peuvent alors être considérés.

Premièrement, nous avons vu que les exemples, principales données du problème, sont représentés grâce à un langage d'exemples $\mathcal{L}_e$. L'espace de recherche est composé des définitions possibles pour le concept, appelées des hypothèses. Nous rappelons que le langage des hypothèses est noté $\mathcal{L}_h$. Nous verrons cependant que dans le cas des opérateurs bidirectionnels, l'espace de recherche peut être modifié (voir section 3.3.2) afin qu'un état représente un ensemble d'hypothèses plutôt qu'une seule. Enfin, il nous faut séparer les exemples en deux ensembles, les exemples positifs E^+ et négatifs E^-.

Le problème d'apprentissage qui nous intéresse se définit de cette manière :

Définition 3.2.5 (Problème d'apprentissage)**.** *Étant donnés des ensembles d'exemples E^+ et E^-, un langage d'hypothèses $\mathcal{L}_h$ et un test de couverture $\succeq$, le but est de trouver une hypothèse h de $\mathcal{L}_h$ telle que :*

(1) *h soit* complète*, c'est-à-dire* $\forall e^+ \in E^+,\ h \succeq e^+$ *;*

(2) *h soit* cohérente (consistance)*, c'est-à-dire* $\forall e^- \in E^-,\ h \not\succeq e^-$.

Ainsi, la complétude permet de garantir que tous les exemples positifs soient couverts et la cohérence que les négatifs soient rejetés. On peut cependant assouplir ces contraintes dans différents cas, comme par exemple, si les données sont bruitées et que certains exemples sont alors mal étiquetés. Ce cas là ne sera pas considéré dans la suite.

Exemple (suite)

Une solution complète et cohérente du problème des trains donné dans la section 3.2.1 représente les trains ayant au moins deux voitures contenant des objets de la même forme et telles que l'une des deux ait un nombre plus grand d'objet et un nombre plus important de roues que l'autre. Cela peut être représenté en une seule règle :

$$goodtrain(T) : -has\_car(T, C_1), has\_car(T, C_2), C_1 \neq C_2,$$

$$wheels(C_1, W_1), wheels(C_2, W_2),$$
$$load(C_1, O, L_1), load(C_2, O, L_2),$$
$$W_1 < W_2, L_1 < L_2$$

À noter qu'avec uniquement deux exemples, nous pouvons trouver des règles plus simples, surtout si nous autorisons les constantes dans les règles comme par exemple :

$goodtrain(T) : -has\_car(T, C_1), load(C_1, O, 3)$

Dans notre contexte, nous distinguons deux cas d'apprentissage : l'*apprentissage de règles* et l'*apprentissage disjonctif*. Dans le premier cas, l'hypothèse est composée d'une unique règle. En programmation logique inductive, nous avons vu précédemment que cette règle prend la forme d'une clause de Horn $h \leftarrow C$ où la tête h est le concept cible ($goodtrain(X)$ dans l'exemple des trains) et C une conjonction de littéraux. Dans le second cas, la définition recherchée pour le concept est une conjonction de règles de la forme :

$$(h \leftarrow C_1) \wedge (h \leftarrow C_2) \wedge \cdots \wedge (h \leftarrow C_n)$$

En extrayant la tête des clauses, on peut également voir cette définition de la manière suivante :

$$h \leftarrow C_1 \vee C_2 \vee \cdots \vee C_n$$

où le corps est alors une DNF.

Nous discuterons des stratégies pour rechercher une règle simple à la section 3.4.1. Nous verrons à la section 3.4.2, que l'apprentissage disjonctif peut être réduit à l'apprentissage successif de règles. Pour des raisons de complexité, l'apprentissage successif des règles est la seule approche que nous avons considérée dans cette thèse. Par conséquent, nous ne considérons dans la suite uniquement des langages d'hypothèses représentant des règles plutôt que des formules plus complexes.

Il existe une variante du problème utilisant une *base de connaissances*. Une telle base a pour but d'enrichir la description des exemples avec un ensemble de règles. Cela ne modifie que très peu la définition du problème où il faut considérer la base de connaissances au moment des tests de couverture.

La Figure 3.2 résume les données en entrées et sorties du problème du train illustrant ces sections. Nous retrouvons d'une part les ensembles d'exemples positifs et négatifs (E^+ et E^-) ainsi que la base de connaissances B pour les entrées et d'autre part la sortie du système d'apprentissage constituée d'une seule règle.

3.3 Les opérateurs de raffinement

La résolution du problème d'apprentissage va consister à explorer l'espace de recherche afin de trouver un état vérifiant les contraintes de cohérence et complétude. Pour ce faire, elle part d'une hypothèse initiale et voit comment la généraliser ou la spécialiser. Cette opération est appelée un raffinement. Un opérateur de raffinement va quant à lui retourner à partir d'une hypothèse l'ensemble des raffinements possibles, c'est à dire des états de recherche directement accessibles à partir de l'état courant. Par exemple, un opérateur dit *descendant* produira des raffinements plus spécifiques (en fonction de l'ordre de généralité), alors qu'un opérateur *ascendant* proposera des hypothèses plus générales. Il s'agit donc d'un élément clé d'un algorithme d'apprentissage puisque les opérateurs seront exploités lors de la résolution afin de choisir où se déplacer.

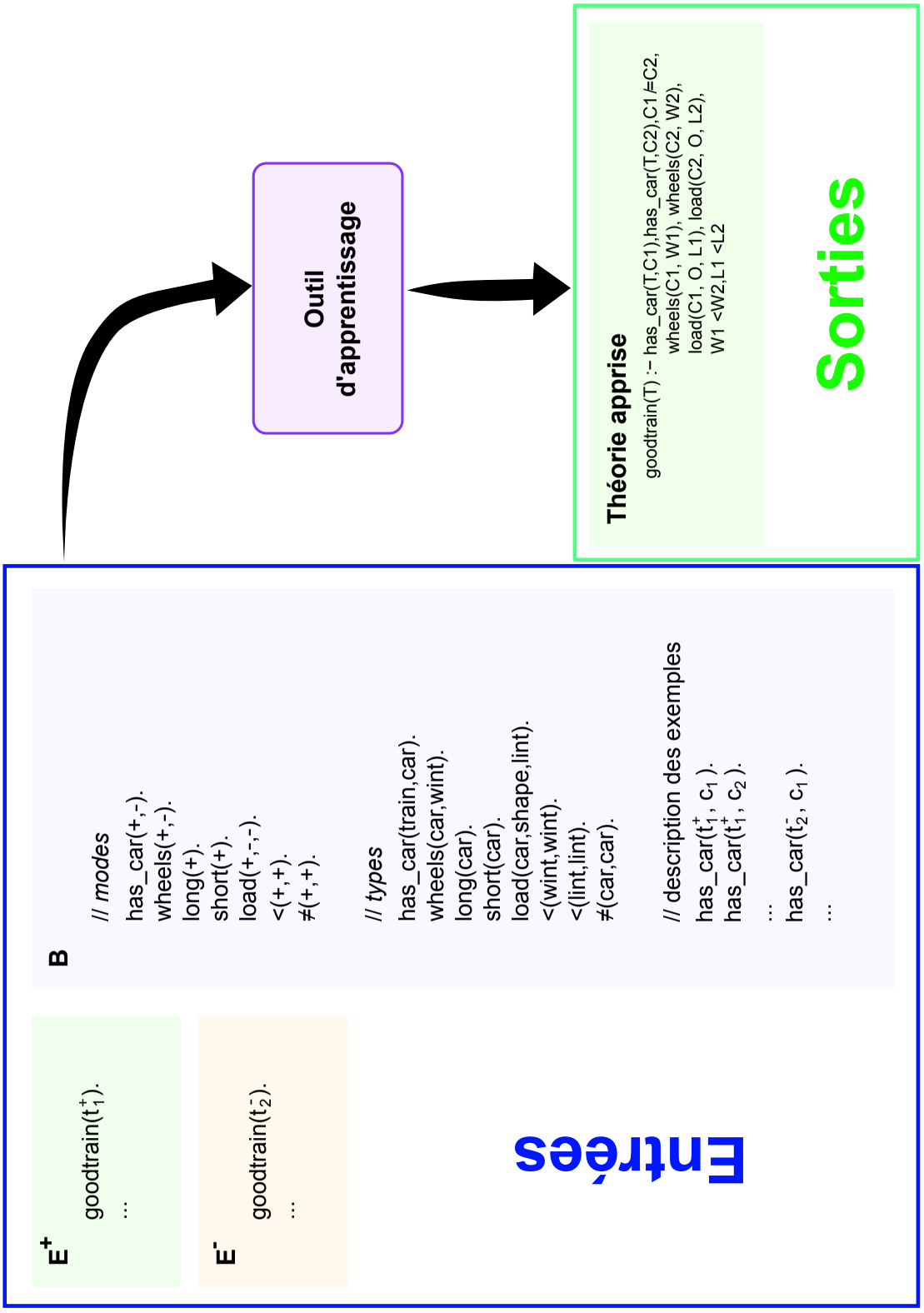

FIGURE 3.2 – Résumé des entrées et sorties d'un problème d'apprentissage en ILP

Dans cette section, nous détaillons ce qu'est un opérateur de raffinement et notamment le cas bi-directionnel en montrant les différences existant avec les opérateurs descendant (top-down) et ascendant (bottom-up), appelés par la suite mono-directionnel. En effet dans la littérature, on trouve généralement des définitions pour les opérateurs descendants et ascendants mais pas pour les opérateurs bi-directionnels. Il y a plusieurs façons de voir un tel opérateur. Par exemple, on peut le voir comme un opérateur pouvant produire des raffinements soit plus généraux et soit plus spécifiques, comparés à l'hypothèse courante. Cependant dans ce cas, on peut voir ce genre d'opérateur comme l'union de deux opérateurs mono-directionnels, l'un descendant l'autre ascendant comme c'est le cas dans l'algorithme Jojo[FW93]. Une autre manière, qui est celle qui nous intéresse, est de le voir comme un opérateur explorant l'espace des hypothèses dans les deux directions, descendante et ascendante, et pour ce faire maintenant un couple d'hypothèses plutôt qu'une unique. Dans ce sens, un des opérateurs bi-directionnels le plus connu est celui basé sur l'espace des versions de Mitchell[Mit97] que nous présenterons en section 3.4.4.

La suite est découpée en une première partie décrivant les familles d'opérateurs mono-directionnels, puis une seconde où nous introduirons les opérateurs bi-directionnels avec leurs particularités comme leur espace de recherche.

3.3.1 Les opérateurs mono-directionnels

L'espace de recherche dans ce cas correspond au langage des hypothèses $\mathcal{L}_h$. Ce langage est muni d'un ordre de généralité, noté $\succeq_h$ permettant de comparer les hypothèses entre elles. Quand ils existent, nous noterons $\top$, respectivement $\bot$, l'élément maximal, respectivement minimal, de $\mathcal{L}_h$.

Les opérateurs mono-directionnels se séparent en deux familles, les *descendants* et les *ascendants* introduits plus haut. De plus, les opérateurs peuvent être également divisés entre ceux de type *générer-et-tester* et ceux de type *dirigés par les données*. Les opérateurs *générer-et-tester* ne fonctionnent qu'à partir du langage des hypothèses et indépendamment des données (c'est-à-dire des exemples). Ils produiront donc des hypothèses sans considérer des contraintes de cohérence et de complétude. Ainsi quels que soient les exemples fournis pour l'apprentissage, un opérateur *générer-et-tester* produira les mêmes raffinements.

Définition 3.3.1 (Opérateur *générer-et-tester*)**.** *Étant donné un langage d'hypothèses $\mathcal{L}_h$ muni d'un ordre de généralité $\succeq_h$, un opérateur ρ est dit* générer-et-tester *s'il vérifie :*

- *dans le cas descendant :* $\forall h \in \mathcal{L}_h,\ \rho(h) \subseteq \{h' \mid h \succeq_h h'\}$
- *dans le cas ascendant :* $\forall h \in \mathcal{L}_h,\ \rho(h) \subseteq \{h' \mid h' \succeq_h h\}$

Les opérateurs *dirigés par les données*, quant à eux, utilisent les exemples pour produire des raffinements. Dans les définitions que nous donnons ci-dessous, nous considérons que ces opérateurs considèrent un unique exemple qu'il devra couvrir si c'est un exemple positif ou rejeter dans le cas où il serait négatif.

Définition 3.3.2 (Opérateur *dirigé par les données*)**.** *Étant donnés un langage d'hypothèses $\mathcal{L}_h$ muni d'un ordre de généralité $\succeq_h$, un exemple positif $e^+ \in E^+$, un exemple négatif $e^- \in E^-$ et le test de couverture $\succeq$, un opérateur ρ est dit* dirigé par les données *s'il vérifie :*

- *dans le cas descendant :* $\forall h \in \mathcal{L}_h,\ \rho(h, e^-) \subseteq \{h' \mid h \succeq_h h' \ \wedge\ h' \not\succeq e^-\}$
- *dans le cas ascendant :* $\forall h \in \mathcal{L}_h,\ \rho(h, e^+) \subseteq \{h' \mid h' \succeq_h h \ \wedge\ h' \succeq e^+\}$

Un opérateur *dirigé par les données* a l'avantage par rapport à un *générer-et-tester* de pouvoir limiter le nombre de raffinements produits. En effet, un raffinement candidat devra respecter une contrainte supplémentaire sur un exemple (dépendant du sens de l'opérateur). En contre partie, le calcul des raffinements est généralement plus complexe.

Pour finir, nous introduisons deux caractéristiques pour ces opérateurs : l'optimalité[RB93] et l'idéalité[vdLNC94]. Ces caractéristiques mettent en évidence si un opérateur permet d'explorer exhaustivement l'espace de recherche. Les définitions sont données pour le cas descendant et *générer-et-tester* mais peuvent être adaptées au cas ascendant et/ou *dirigé par les données*. Elles sont librement inspirées des travaux de [vdLNC94, TNM09].

Avant de donner la définition de ces deux familles d'opérateurs, il nous faut introduire différents concepts comme les opérateurs localement finis, non-redondants, propres et (faiblement) complets.

Définition 3.3.3 (Localement fini)**.** *Étant donné un langage d'hypothèses $\mathcal{L}_h$ muni d'un ordre de généralité $\succeq_h$, un opérateur descendant ρ est localement fini si pour tout $h \in \mathcal{L}_h$, $\rho(h)$ est fini et calculable.*

La propriété de propreté permet d'être sûr qu'un opérateur appliqué à une hypothèse h ne produira ni h ni une hypothèse équivalente (dans le cas descendant).

Définition 3.3.4 (Propre)**.** *Étant donné un langage d'hypothèses $\mathcal{L}_h$ muni d'un ordre de généralité $\succeq_h$ où $\succ_h$ est sa version stricte*[1]*, un opérateur descendant ρ est propre si pour tout $h \in \mathcal{L}_h$, $\rho(h) \subseteq \{h' | h \succ h'\}$.*

La non-redondance d'un opérateur permet de garantir que deux hypothèses ne seront pas produites plusieurs fois lors de la recherche. Pour définir la non-redondance, nous notons ρ^*, la clôture réflexive et transitive d'un opérateur ρ. Cet ensemble correspond aux états de l'espace de recherche atteignable par l'opérateur à partir d'un état de départ.

Définition 3.3.5 (Non-redondant)**.** *Étant donné un langage d'hypothèses $\mathcal{L}_h$ muni d'un ordre de généralité $\succeq_h$ où $\succ_h$ est sa version stricte, un opérateur descendant ρ est non-redondant si pour tous $C,\ D,\ E \in \mathcal{L}_h$, $E \in \rho^*(C)$ et $E \in \rho^*(D)$ impliquent soit $C \in \rho^*(D)$, soit $D \in \rho^*(C)$.*

Il existe deux notions de complétude pour un opérateur. Un opérateur descendant faiblement complet permet à partir de l'hypothèse $\top$ de produire par application répétée de l'opérateur l'ensemble de l'espace de recherche. Ainsi, un tel opérateur garantira à un algorithme qu'il existe un moyen d'atteindre toute hypothèse de $\mathcal{L}_h$. Il existe une notion plus forte qui consiste à dire qu'un opérateur *descendant* est complet si toute hypothèse plus spécifique que l'hypothèse courante peut être produite par application répétée de l'opérateur en partant de l'hypothèse courante.

Définition 3.3.6 (Faiblement complet)**.** *Étant donné un langage d'hypothèses $\mathcal{L}_h$ muni d'un ordre de généralité $\succeq_h$ où $\succ_h$ est sa version stricte, un opérateur descendant ρ est dit faiblement complet si $\rho^*(\top) = \mathcal{L}_h$.*

Définition 3.3.7 (Complet)**.** *Étant donné un langage d'hypothèses $\mathcal{L}_h$ muni d'un ordre de généralité $\succeq_h$ où $\succ_h$ est sa version stricte, un opérateur descendant ρ est dit complet si pour $C,\ D \in \mathcal{L}_h$, si $C \succ D$, alors $D \in \rho^*(C)$.*

1. c'est à dire $C \succ_h D \Leftrightarrow C \succeq_h D \ \wedge\ D \not\succeq_h C$

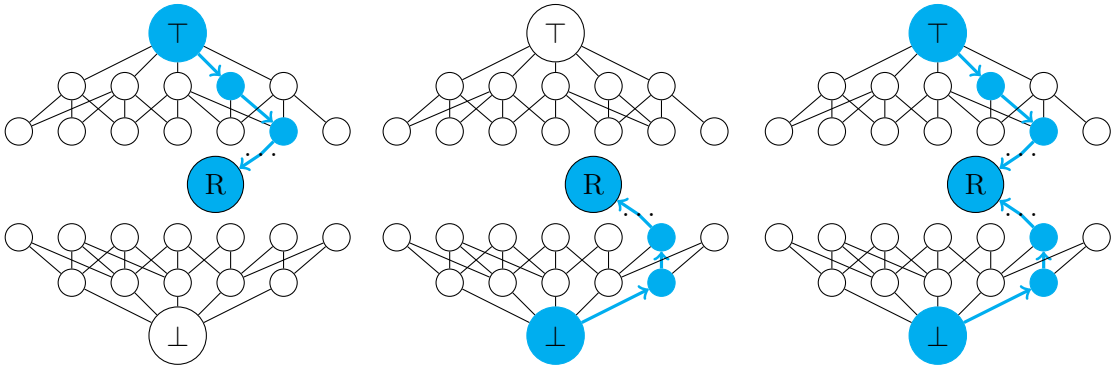

FIGURE 3.3 – Type de recherche pouvant être produite par un opérateur

Ces deux notions sont importantes car un opérateur ne peut être à la fois non-redondant et complet[NCW97]. De plus, elles permettent de garantir que l'opérateur permet d'explorer l'intégralité de l'espace de recherche et par conséquent que les solutions sont atteignables.

Définition 3.3.8 (Opérateur optimal et idéal)**.** *Étant donné un langage d'hypothèses $\mathcal{L}_h$ muni d'un ordre de généralité $\succeq_h$, un opérateur* descendant générer-et-tester *est dit :*

- *optimal s'il est localement fini, non-redondant et faiblement complet ;*
- *idéal s'il est localement fini, propre et complet.*

3.3.2 Les opérateurs bi-directionnels

Dans le cadre bi-directionnel, l'opérateur ne doit pas raffiner qu'une unique hypothèse mais un couple. En partant de deux hypothèses, l'application successive de l'opérateur doit permettre une exploration ordonnée par rapport à ces hypothèses de l'espace de recherche. À notre connaissance, il n'existe pas de cadre général dans la littérature pour décrire cette famille d'opérateurs. Nous proposons donc de reprendre les définitions précédentes en les adaptant à ce contexte. Par conséquent, il faut considérer soit qu'il y a, à chaque étape de recherche, deux hypothèses courantes, soit que l'espace de recherche n'est pas $\mathcal{L}_h$ mais un espace tirant partie de cette spécificité. Même si ces deux approches sont équivalentes, nous préfèrerons pour la suite utiliser la seconde. La Figure 3.3 donne l'idée intuitive du type de recherche que produit un tel opérateur comparé aux opérateurs mono-directionnels. L'espace de recherche, noté $\mathcal{E}$, sera donc composé de couples d'hypothèses de $\mathcal{L}_h$. Afin de limiter le nombre d'états, nous considérons que ces couples sont ordonnés. Ainsi, nous utilisons $\succeq_h$ afin de garantir que la première hypothèse du couple est plus générale que la seconde. Plus formellement, pour tout couple (C, D) de $\mathcal{E}$, la propriété $C \succeq_h D$ est vérifiée. Un état (C, D) de $\mathcal{E}$ représente un ensemble d'hypothèses candidates, à savoir $\{E \mid E \in \mathcal{L}_h \wedge C \succeq_h E \wedge E \succeq_h D\}$, contrairement à $\mathcal{L}_h$ où un état ne représente qu'une hypothèse.

Pour finir de définir cet espace de recherche, nous décrivons les états solutions dans le problème d'apprentissage que nous considérons (apprentissage d'une règle complète et cohérente). Pour rappel, les états solutions dans $\mathcal{L}_h$ sont ceux correspondant aux hypothèses satisfaisant les contraintes de complétude et de cohérence. Le but du problème de recherche (définition 3.2.5) étant de trouver une seule hypothèse de $\mathcal{L}_h$ décrivant le concept, les états

de $\mathcal{E}$ pouvant être solution font partie de l'ensemble $\{(C, C) \mid C \in \mathcal{L}_h\}$. Parmi ceux-là, les solutions sont les hypothèses C cohérentes et complètes vis-à-vis des exemples.

Définition 3.3.9 (États finaux de $\mathcal{E}$)**.** *Étant donné le problème d'apprentissage (définition 3.2.5) où le but est de trouver une hypothèse cohérerente et complète avec les ensembles d'exemples E^+ et E^-, nous définissons les états finaux de $\mathcal{E}$ comme les états $(C, C) \in \mathcal{E}$ défini par :*

- $C \in \mathcal{L}_h$;
- $\forall e^+ \in E^+,\ C \succeq e^+$;
- $\forall e^- \in E^-,\ C \not\succeq e^-$.

Comme pour $\mathcal{L}_h$, nous avons besoin d'un ordre afin de structurer l'espace de recherche. Parmi les ordres possibles, nous en utilisons un dépendant exclusivement de l'ordre sur $\mathcal{L}_h$.

Définition 3.3.10 (Ordre sur $\mathcal{E}$)**.** *On définit l'ordre $\succeq$ sur $\mathcal{E}$ de la manière suivante :*

$$\forall (C, D), (C', D') \in \mathcal{E}, (C, D) \succeq (C', D') \Leftrightarrow C \succeq_h C' \wedge D' \succeq_h D$$

Ainsi l'élément maximal de $\mathcal{E}$ correspond au couple d'hypothèses formé de l'hypothèse la plus générale $\top$ et la plus spécifique $\bot$ (si elles existent). Un élément inférieur « réduira la distance » entre les deux hypothèses. Les éléments minimaux sont tous les couples du type (C, C), où $C \in \mathcal{L}_h$.

Nous cherchons maintenant à définir de manière formelle ce que nous appellerons par la suite un opérateur bi-directionnel. L'objectif est de fournir, de la même manière que pour les opérateurs mono-directionnels, une définition suffisamment large pour englober la majorité de ces opérateurs. Nous proposerons deux approches une de type *générer-et-tester* et l'autre *dirigée par les données.*

Définition 3.3.11 (Opérateur bi-directionnel *générer-et-tester*)**.** *Étant donnés un langage d'hypothèse $\mathcal{L}_h$ muni d'un ordre $\succeq_h$, un espace $\mathcal{E}$ de couples d'hypothèses de $\mathcal{L}_h$ et un ordre $\succeq$ sur $\mathcal{E}$, un opérateur bi-directionnel ρ est dit* générer-et-tester *s'il respecte :*

$$\rho(C, D) \subseteq \{(C', D') \mid \ (C, D) \succeq (C', D')\}$$

Définition 3.3.12 (Opérateur bi-directionnel *dirigé par les données*)**.** *Étant donnés un langage d'hypothèse $\mathcal{L}_h$, un espace $\mathcal{E}$ de couples d'hypothèses de $\mathcal{L}_h$, un ordre $\succeq$ sur $\mathcal{E}$ et un exemple, soit positif e^+, soit négatif e^-, un opérateur bi-directionnel ρ est dit* dirigé par les données *s'il respecte :*

$$\rho(C, D, e^+) \subseteq \{(C', D') \mid \ (C, D) \succeq (C', D') \wedge D' \succeq_h e^+\}$$

$$\rho(C, D, e^-) \subseteq \{(C', D') \mid \ (C, D) \succeq (C', D') \wedge C' \not\succeq_h e^-\}$$

De la même manière que pour les opérateurs mono-directionnels, nous pouvons définir des propriétés d'optimalité et d'idéalité. Cependant les notions de complétude ne sont pas adaptées pour notre problème si nous les transférons telles quelles aux opérateurs bi-directionnels. En effet, pour un opérateur mono-directionnel, être complet ou faiblement complet permet de concevoir des algorithmes garantissant de traverser tous les états de recherche. Cela permet d'être sûr de trouver une hypothèse cohérente et complète (si elle existe) voire la meilleure. Or, nous avons caractérisé que, pour notre problème d'apprentissage, une solution est une hypothèse C complète et cohérente dans $\mathcal{L}_h$, compris dans l'ensemble des états finaux donnés à la définition 3.3.9. Par conséquent, la seule garantie

nécessaire aux algorithmes exploitant un opérateur bi-directionnel et résolvant notre problème d'apprentissage est de permettre de traverser tous les états $\{(C, C) \mid C \in \mathcal{L}_h\}$. Les autres états permettent uniquement de diriger la recherche voire d'élaguer des parties de l'espace. Nous définissons donc des notions de $\mathcal{L}_h$-complétude et faiblement $\mathcal{L}_h$-complétude, propres aux opérateurs bi-directionnels et qui préservent la motivation initiale des opérateurs mono-directionnels.

Définition 3.3.13 (Faiblement $\mathcal{L}_h$-complet)**.** *Étant donné un espace $\mathcal{E}$ de couple d'hypothèses de $\mathcal{L}_h$ muni d'un élément maximal $(\top, \bot)$, où $\top$ et $\bot$ sont respectivement les éléments maximal et minimal de $\mathcal{L}_h$ s'ils existent, un opérateur bi-directionnel ρ est dit faiblement $\mathcal{L}_h$-complet si $\forall C \in \mathcal{L}_h,\ (C, C) \in \rho^*(\top, \bot)$.*

Définition 3.3.14 ($\mathcal{L}_h$-complet)**.** *Étant donné un espace $\mathcal{E}$ de couples d'hypothèses de $\mathcal{L}_h$, un opérateur bi-directionnel ρ est dit $\mathcal{L}_h$-complet si pour tout $C \in \mathcal{L}_h$ et $(D, E) \in \mathcal{E}$ tel que $(D, E) \succ (C, C)$, alors (C, C) est dans $\rho^*(D, E)$.*

Dans le cadre bi-directionnel, nous substituerons ces définitions à celles présentes dans les définitions des opérateurs idéaux et optimaux.

3.4 Se déplacer dans l'espace avec ces opérateurs

Nous allons maintenant discuter des différentes manières dont ces opérateurs peuvent être utilisés pour trouver une définition. Nous présenterons également deux algorithmes utilisés dans les chapitres suivants. Le premier, appelé *separate-and-conquer*[Fur99], cherche à apprendre progressivement un ensemble d'hypothèses pour définir le concept. Le second est l'algorithme d'élimination des candidats dans l'espace des versions de Tom Mitchell [Mit82]. Le but de cette section n'est pas d'avoir un paysage exhaustif des nombreuses approches développées dans l'apprentissage de règles mais de présenter les outils utilisés dans la suite de cette thèse.

3.4.1 Recherche de règles

Nous avons vu que chercher la définition d'un concept cible revenait à parcourir un espace de recherche composé d'hypothèses. On peut bien entendu procéder à une recherche complète, c'est-à-dire parcourir l'ensemble des états de l'espace jusqu'à trouver une solution satisfaisant les contraintes de cohérence et complétude. Cependant cela n'est pas toujours possible en pratique. En effet, certains problèmes proposent un espace de recherche d'une taille suffisamment grande pour rendre ce genre d'approche impraticable. C'est notamment le cas de nombreux problèmes en ILP dont celui qui nous intéressera au chapitre 4.6. Il existe cependant d'autres approches, incomplètes, mais capables de trouver des solutions intéressantes en pratique. Ces stratégies sont des approches gloutonnes ne garantissant pas de trouver de solutions. Par exemple, la *beam search* (recherche par faisceaux en français) propose à chaque étape de raffinement de limiter le nombre d'hypothèses explorées ; cette limite est appelée la taille de la *beam*. Si la limite d'une *beam search* est fixée à l'infini et que l'opérateur de raffinement est complet ou faiblement complet, alors ce type de recherche revient à une recherche complète. La stratégie *hill climbing* ne retiendra qu'une seule hypothèse par étape de raffinement. Cette stratégie est un cas particulier de la *beam search* où la taille de la *beam* est fixée à 1. L'algorithme de la Figure 3.4 exécute une recherche de règle par faisceau avec un opérateur descendant pour résoudre notre problème d'apprentissage. Il commence par initialiser le faisceau avec l'hypothèse $\top$ la plus générale de l'espace de

Algorithm : LEARNRULE($E^+, E^-, \mathcal{L}_h$)

```
1.  beam ← {⊤} //⊤ element of L_h
2.  while beam ≠ ∅
3.    candidates ← ∅
4.    for each r ∈ beam do
5.      candidates ← candidates ∪ ρ(r)
6.    beam ← BESTCANDIDATES(beamsize, candidates)
7.    if  ∃r ∈ beam such that
8.          ∀e+ ∈ E+,  r ⪰ e+
9.          ∀e− ∈ E−,  r ⋡ e−
10.       return  r
11. throw no solution
```

FIGURE 3.4 – Recherche par faisceau par un opérateur mono-directionnel descendant

recherche. Ensuite, il recherche une règle complète et cohérence par application successive de l'opérateur de raffinement ρ sur les règles du faisceau. À noter que nous détaillerons ci-dessous comment la fonction BESTCANDIDATES sélectionne les meilleures raffinements pour restreindre la taille du faisceau.

Comme ces approches ne garantissent pas d'explorer l'ensemble de l'espace de recherche, il est nécessaire de les diriger vers les zones de l'espace susceptibles de contenir des solutions, voire les plus intéressantes. Ceci correspond à un biais de recherche. En effet, quand on généralise la définition d'un concept à partir d'exemples, nous n'avons aucune garantie qu'une solution soit meilleure qu'une autre[Sch94]. Prenons par exemple deux ensembles d'exemples E^+ et E^- que l'on divise chacun en deux. Nous avons donc quatre ensembles E_1^+, E_2^+, E_1^- et E_2^-. Si nous résolvons le problème d'apprentissage discriminant E_1^+ de E_1^-, nous obtenons une règle r que nous espérons suffisamment générale pour également discriminer E_2^+ de E_2^-. Comme l'apprentissage n'a pas porté sur ces deux ensembles rien ne le garantit cependant. Nous pouvons même imaginer inverser les ensembles E_2^+ et E_2^- : si nous apprenons encore sur E_1^+ de E_1^- la même règle r sera apprise mais aura un résultat totalement différent sur les ensembles E_2^+ et E_2^- inversé. Les ensembles d'exemples ne donnent qu'une partie de l'univers étudié par le système d'apprentissage et ne permettent pas de garantir l'efficacité des propriétés apprises. Cependant, la qualité d'une hypothèse peut être jugée comparée aux exemples connus du concept. Ainsi plusieurs heuristiques ont été développées afin de pouvoir préférer une hypothèse à une autre. Elles ont comme point commun de dépendre des mêmes critères : les nombres d'exemples positifs couverts et d'exemples négatifs couverts. Dans certains cas, la taille de l'hypothèse est prise en compte. L'heuristique consistera à choisir l'hypothèse avec la meilleure valeur d'une fonction d'évaluation prenant en compte ces critères. Nous allons maintenant donner quelques exemples d'heuristiques afin d'illustrer ce biais de recherche. Les exemples d'heuristiques choisies ont été sélectionnés dans [Fur99] et nous utiliserons également les mêmes notations :

- P : le nombre d'exemples positifs (égal à $|E^+|$) ;
- N : le nombre d'exemples négatifs (égal à $|E^-|$) ;
- r : la règle candidate (un des raffinements possible) ;
- r' : la règle précédente (donc $r \in \rho(r')$ dans le cas d'une recherche mono-directionnelle) ;
- p : le nombre d'exemples positifs couverts par r (égal à $|\{e|e \in E^+ \ \wedge \ r \succeq e\}|$;
- n : le nombre d'exemples négatifs couverts par r (égal à $|\{e|e \in E^- \ \wedge \ r \succeq e\}|$;
- l : le nombre de conditions dans r (c'est-à-dire le nombre de littéraux dans le corps d'une règle dans le cas de l'ILP).

Accuracy Notamment utilisée en partie dans Progol (voir section 3.4.3), cette heuristique calcule le taux d'exemples correctement couverts par la règle :

$$acc(r) = \frac{p + (N - n)}{P + N} \simeq p - n$$

À noter que Progol utilise plutôt la différence entre les positifs et et les négatifs couverts. Cela se justifie comme les nombres P et N sont constants lors de la recherche d'une règle.

Purity La pureté quantifie la portion d'exemples positifs couverts par la règle parmi tous les exemples couverts :

$$purity(r) = \frac{p}{p + n}$$

Information Content Équivalent à la pureté, l'heuristique information content est souvent utilisée à l'intérieur d'autres heuristiques :

$$ic(r) = -log\frac{p}{p + n}$$

Weigthed Information Gain Cette heuristique est utilisée la première fois dans Foil (voir section 3.4.3) :

$$wig(r) = -p \times (ic(r) - ic(r'))$$

Dans sa forme originale, Quinlan considérait le nombre de modèles de chaque règle comme facteur plutôt que p.

Le lecteur pourra se reporter à [FF05] pour une étude des heuristiques les plus souvent utilisées.

3.4.2 Separate-and-conquer

Comme dit précédemment, certains langages ne sont pas suffisamment riches pour décrire certains concepts. Pour remédier à cela, il suffit d'augmenter l'expressivité d'un langage. Mais cela à un coût en terme d'efficacité et ne permet pas toujours d'obtenir des approches réussissant en pratique. Une idée pour augmenter l'expressivité du langage, sans pour autant « exploser » la complexité, est d'assouplir la contrainte de cohérence, forçant une règle à couvrir tous les exemples positifs et d'apprendre plusieurs règles telles que l'union des exemples couverts corresponde à l'ensemble des exemples positifs et par conséquent formant une définition cohérente du problème.

Ainsi, un cadre *separate-and-conquer*(diviser pour régner en français) a été largement utilisé en apprentissage de règles [Fur99]. L'intuition est la suivante : on commence par apprendre une règle, on supprime les exemples qu'elle couvre de l'ensemble E^+ et on recommence jusqu'à ce que E^+ soit vide. L'algorithme de la Figure 3.5 détaille l'approche. Il part avec une définition vide.

Cette augmentation de l'expressivité est cependant à relativiser. L'apprentissage successif de règles pose des problèmes de fragmentation des données et il existe un risque d'obtenir un certain nombre de règles spécifiques à de petits ensembles d'exemples. On peut cependant trouver dans la littérature (voir [Dom97] par exemple) des systèmes visant à combiner les différentes règles apprises.

Tant qu'il y a des exemples positifs non couverts, il apprend une règle qu'il ajoute à la définition et retire de E^+ les exemples positifs couverts.

Cette approche est utilisée dans le cadre du chapitre 4 uniquement dans sa version ILP.

Algorithm : SEPARATEANDCONQUER($E^+, E^-, \mathcal{L}_h$)

```
1. definition ← ∅
2. while E+ ≠ ∅
3.    rule ←LEARNRULE(E+, E−, L_h)
4.    definition ← definition ∪ {rule}
5.    E+ ← E+\COVER(rule, E+)
6. return definition
```

FIGURE 3.5 – Squelette d'un algorithme de type *separate-and-conquer*

3.4.3 Systèmes d'apprentissage de règles ou d'apprentissage disjonctif

Dans cette partie, nous décrivons les différents systèmes ILP utilisés dans les chapitres 4 et 5. Il s'agit plus de donner les clés pour comprendre comment ils fonctionnent et comment les distinguer notamment de notre approche (chapitre 5) que de détailler avec précision le fonctionnement technique de ces systèmes.

Foil

Proposé par Ross Quinlan[QCJ93], cet algorithme est basé sur un opérateur descendant *générer-et-tester*. En partant de la clause $p(X_1, X_2, \ldots, X_k) \leftarrow$, sans corps et où p est le prédicat représentant le concept cible, l'algorithme raffine la règle en ajoutant un littéral à la règle courante.

Plus formellement, l'opérateur de raffinement s'écrit :

$$\rho_{foil}(h) = \{h' \mid h' = h + \text{un littéral dans le corps } \wedge\ h' \in \mathcal{L}_h\}$$

Par souci d'efficacité, Foil utilise une recherche *hill-climbing*. Comme dit précédemment, il utilise l'heuristique *Weigthed Information Gain* pour sélectionner un raffinement à chaque étape.

Progol

Progol[Mug95] utilise les biais de langages comme les modes et les types pour ordonner les littéraux de la clause $\perp$ en une séquence. Un littéral est placé dans cette séquence de telle manière que ses variables en entrées apparaissent dans les littéraux précédents. L'opérateur de raffinement va alors considérer un à un les littéraux dans l'ordre de cette séquence et modifiera une règle en ajoutant ou non le littéral considéré. Les littéraux seront ajoutés cependant à un renommage près des variables. En effet, progol offre la possibilité d'ajouter un littéral en *divisant* ses variables en sortie c'est à dire en utilisant une variable non présente dans $\perp$. Ce système utilise une recherche descendante de type A^*. Une étude détaillée de cet opérateur peut être trouvée dans [TNM09].

Aleph

Aleph[Sri] est une plateforme permettant d'émuler un grand nombre d'algorithmes d'ILP. Permettant de fixer une *beam* de 1 à l'infini, d'utiliser des opérateurs descendants ou ascendants assez variés, de choisir les heuristiques évaluant les recherches, Aleph permet de simuler des algorithmes comme Foil et Progol. Il permet également de procéder à des saturations d'exemples, opération servant régulièrement pour définir une clause $\perp$ et décrite en détail au chapitre 5.

ICL

ICL[RL95] a la particularité de ne pas chercher à apprendre un ensemble de règles sous la forme de clauses de Horn mais une CNF. Utilisant une approche *separate-and-conquer*, ce système apprend des règles ayant la forme de disjonctions de littéraux progressivement en supprimant les exemples négatifs non couverts par la règle (contrairement à l'approche standard supprimant les exemples positifs couverts). La construction de ces règles se fait avec un opérateur descendant proche de celui de Foil, ajoutant, un à un, des littéraux à la disjonction. Comme signalé dans [Lae02], cette approche est le dual de l'approche standard revenant à apprendre le concept négatif quand on considère des clauses de Horn.

Beth

Beth[Tan03] se distingue d'autres approches en proposant de calculer la clause $\bot$ à la volée lors de la recherche de la règle et éviter le calcul complet de la saturation d'un exemple. À noter que cela peut également être simulé par Aleph.

Propal

Propal[AR06] est basé sur un opérateur descendant dirigé par les données. Un raffinement d'une règle r doit rejeter au moins un exemple négatif e couvert par r. Pour cela, les auteurs utilisent un CSP pour chercher le raffinement le plus proche de r en nombre de littéraux rejettant e.

Progolem

Progolem[MSTN09] est un système ascendant dirigé par les données combinant :
– la clause $\bot$ comme défini dans Progol (séquence de littéraux) ;
– des généralisations successives inspirées du système golem[MF90].

Étant donné un exemple positif e, l'idée est de chercher les généralisations possibles de la règle courante couvrant e en se limitant aux sous-séquences de $\bot$.

Still

STILL[SR00] est un système utilisant une inférence stochastique pour construire des règles. Pour cela, il sélectionne, pour chaque exemple positif, un exemple négatif puis, au hasard, une substitution des termes de l'exemple négatif vers le positif. Enfin, il raffine la règle de manière à ce que la règle obtenue rejette la substitution.

Une chose intéressante dans ce système est la possibilité d'ajouter des contraintes sur les domaines d'une variable (par exemple en ajoutant X<5,4 comme littéral d'une règle) pour séparer un exemple positif d'un négatif.

Le tableau de la Figure 3.6 résume les particularités de ces systèmes.

Nom du système	Foil	Progol	Aleph	ICL	Beth	Propal	Progolem	Still
Ascendant/**D**escendant	D	D	*	D	D	D	A	D
***G**énérer-et-**T**ester/**D**irigé par les **D**onnées*	GT	GT	GT	GT	GT	DD	DD	DD
***H**ill-**C**limbing/**B**eam-**S**earch/A^**	HC	A^*	*	BS	BS	BS	BS	HC

FIGURE 3.6 – Comparatif des systèmes d'ILP

3.4.4 Espace des versions

L'espace des versions [Mit97] et l'algorithme d'élimination des candidats sont utilisés par CONACQ (voir section 2.3.1) et dans le chapitre 6. L'espace des versions représente l'ensemble des hypothèses satisfaisant la cohérence et la complétude et donc les solutions du problème.

Définition 3.4.1 (Espace des versions)**.** *Étant donnés un ensemble d'exemples positifs E^+, un ensemble d'exemples négatifs E^- et un langage d'hypothèses $\mathcal{L}_h$, l'espace des versions est le sous-ensemble d'hypothèses EV de $\mathcal{L}_h$ tel que pour chaque hypothèse h de EV, h couvre les exemples de E^+ et rejette ceux de E^-. Plus formellement :*

$$EV = \{h \in \mathcal{L}_h \mid \forall e \in E^+,\ h \succeq e \ \wedge\ \forall e \in E^-,\ h \not\succeq e\}$$

Cet espace étant trop grand pour être énuméré, Tom Mitchell propose un algorithme d'élimination des candidats qui permet, plutôt que de construire cet espace, de le délimiter par deux bornes. La première, notée G, regroupe les hypothèses les plus générales et l'autre, notée S, les plus spécifiques. Si une hypothèse h est cohérente et complète avec les exemples, alors il existe $g \in G$ et $s \in S$ telle que $g \geq h \geq s$.

Dans l'algorithme d'élimination, les exemples sont traités de manière incrémentale. En partant avec $G = \{\top\}$ et $S = \{\bot\}$, on raffine les deux bornes en fonction de la classe de l'exemple. Si c'est un exemple positif, il faut retirer les généralisations qui ne le couvrent pas dans G et généraliser au minimum les hypothèses de S ne le couvrant pas. S'il s'agit d'un négatif, il faut spécialiser dans G les hypothèses le couvrant et supprimer de S celles le couvrant. Il nous faut donc deux fonctions : une produisant les généralisations minimales d'une hypothèse, et l'autre les spécialisations maximales.

Définition 3.4.2 (Gen)**.** *Soit* Gen*, la fonction qui étant donnés une hypothèse h, un exemple e et G retourne l'ensemble des généralisations minimales de h, pour lesquelles il existe une hypothèse dans G plus générales.*

Plus formellement :

$$\begin{aligned}\textsc{Gen}(e, h, G) = \{h' \mid\ & h' \succeq e \\ & \wedge\ h' \geq h \\ & \wedge\ \nexists\ h'' \in \mathcal{L}_h,\ h' \geq h'' \geq h \\ & \wedge\ \exists\ h'' \in G,\ h'' \geq h'\}\end{aligned}$$

Définition 3.4.3 (Spe)**.** *Soit* Spe*, la fonction qui étant donnés une hypothèse h, un exemple e et S, retourne l'ensemble des spécialisations maximales de h telle que pour chacune des généralisations, il existe dans S une hypothèse plus spécifique.*

Plus formellement :

$$\begin{aligned}\textsc{Spe}(e, h, S) = \{h' \mid\ & h' \not\succeq e \\ & \wedge\ h \geq h' \\ & \wedge\ \nexists\ h'' \in \mathcal{L}_h,\ h \geq h'' \geq h' \\ & \wedge\ \exists\ h'' \in S,\ h' \geq h''\}\end{aligned}$$

Algorithm : CandidateElimination$(E^+, E^-, \mathcal{L}_h)$

1. $S \leftarrow \{\bot\}$ //$\bot$ element of $\mathcal{L}_h$
2. $G \leftarrow \{\top\}$ //$\top$ element of $\mathcal{L}_h$
3. **for each** $e \in E^+ \cup E^-$
4. **if** $e \in E^+$ **then**
5. //Update of S and G when e is a positive example
6. **for each** $h \in G$ **do if** $h \not\geq e$ **then** $G \leftarrow G - \{h\}$
7. $S' \leftarrow \emptyset$
8. **for each** $h \in S$ **do**
9. **if** $h \not\geq e$ **then** $S \leftarrow S - \{h\}$
10. $S' \leftarrow S' \cup \text{Gen}(e, h, G)$
11. **for each** $h \in S'$
12. **if** $\forall h' \in S \cup S' - \{h\},\ h \not\geq h'$ **then** $S \leftarrow \{h\}$
13. //Update of S and G when e is a negative example
14. **if** $e \in E^-$ **then**
15. **for each** $h \in S$ **do if** $h \not\geq e$ **then** $S \leftarrow S - \{h\}$
16. $G' \leftarrow \emptyset$
17. **for each** $h \in G$ **do**
18. **if** $h \not\geq e$ **then** $G \leftarrow G - \{h\}$
19. $G' \leftarrow G' \cup \text{Spe}(e, h, S)$
20. **for each** $h \in G'$
21. **if** $\forall h' \in G \cup G' - \{h\},\ h' \not\geq h$ **then** $G \leftarrow \{h\}$
22. **return** (S, G)

FIGURE 3.7 – Algorithme d'élimination des candidats

L'algorithme de la Figure 3.7 présente l'algorithme d'élimination des candidats de Tom Mitchell[Mit82], qui sépare les traitements selon que l'exemple reçu par l'algorithme soit positif ou négatif.

Dans cet algorithme, le but n'est pas simplement de chercher une règle cohérente et complète mais l'ensemble des solutions. Même si cela change par rapport au problème d'apprentissage considéré dans ce chapitre, l'algorithme conserve la même structure avec l'application répétée d'un opérateur de raffinement. En effet, étant donnés l'état courant (S, G) et un exemple e, l'opérateur, se cachant derrière l'algorithme de la Figure 3.7, consiste à généraliser S et spécialiser G. En pratique, cette approche n'est applicable qu'à des langages limités. Il a été par exemple utilisé avec succès dans CONACQ où il est encodé dans un problème SAT (problème de satisfaction de formules propositionnelles) détaillé dans le chapitre 6.

3.4.5 Apprentissage actif

L'apprentissage actif [Ang88, Set09], ou apprentissage par requête, cherche à répondre à une faiblesse de l'apprentissage comme définit précédemment (appelé apprentissage passif) : fournir les exemples étiquetés n'est pas toujours une tâche aisée pour l'utilisateur. D'autre part, sur certains problèmes d'apprentissage, peu d'exemples permet d'arriver à un résultat suffisamment précis. En partant de cette constatation, l'apprentissage actif propose que le système d'apprentissage choisisse en fonction du langage d'exemple, des instances non étiquetés. Celles-ci, appelé requêtes, sont soumises à l'utilisateur, plus généralement un oracle capable d'étiqueter ces exemples. Une fois étiquetées ces exemples servent à raffiner l'hypothèse courante construite par l'utilisateur. La Figure 3.8 résume cette approche.

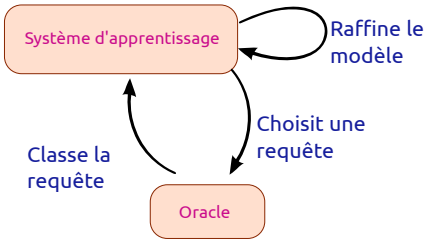

FIGURE 3.8 – Apprentissage actif

Dans cette thèse, nous nous sommes uniquement intéressés à la synthèse de requêtes d'appartenance, il existe cependant d'autres types de requêtes[Ang88] comme les requêtes de sous-ensemble qui prend la forme d'un ensemble d'exemple — si l'utilisateur répond oui cela signifie que ce sont tous des exemples positifs sinon il en identifie un comme étant un exemple négatif.

L'objectif est de limiter le travail de l'oracle en ciblant un petit nombre de requêtes à lui faire étiquetées. En contrepartie, il est parfois difficile de trouver une manière pertinente pour le système d'apprentissage de sélectionner les requêtes à soumettre à l'oracle. Les requêtes intéressantes font partie des requêtes admises par l'hypothèses courantes, c'est-à-dire l'espace de versions, qui comme dit à la section précédente est trop grand pour être énuméré. Une stratégie simple serait de sélectionner ces requêtes au hasard (suivant une loi de distribution) mais on peut trouver dans la littérature des approches plus complexe comme celle développé dans [LG94], où le système cherche à sélectionner un exemple pour lequel l'étiquette courante est le moins fiable possible.

Chapitre 4

Apprentissage de modèles abstraits de CSP

4.1 Introduction

La première approche proposée dans cette thèse concerne l'acquisition automatique de problèmes à partir d'exemples proches (Définition 2.4.1 du chapitre 2). Cette problématique répond aux limites soulevées par les approches existantes d'acquisition de modèle de CSP. Dans ces approches, l'utilisation de solutions et non-solutions du problème recherché par l'utilisateur pose des problèmes quant à la motivation même de ces techniques. En effet, la seule méthode d'acquisition de réseaux de contraintes à notre connaissance est CONACQ qui souffre de certaines limitations (voir chapitres 2 et 6). Comme montré au chapitre 2 cette approche amène plusieurs questions. On peut notamment se demander pourquoi un utilisateur souhaiterait un modèle pour un problème où il possède déjà des solutions. En contraste et d'un point de vue cognitif, l'utilisateur souhaiterait sans doute modéliser un nouveau problème, ayant en sa possession seulement des solutions et non-solutions de problèmes proches résolus antérieurement. Ces problèmes proches sont les mêmes que le problème réel de l'utilisateur mais ne possèdent pas les mêmes domaines et surtout le même nombre de variables. Ainsi dans l'exemple des emplois du temps illustrant cette approche en section 2.4, les nombres de professeurs, de groupes, de salles peuvent être différents.

En parallèle de l'acquisition de contraintes et pour répondre aux limitations de la programmation par contraintes, des langages de modélisation comme Essence[FGJ+07] et Zinc[MNR+08] fournissent un cadre pour modéliser les problèmes par contraintes avec un haut-niveau d'abstraction. De nombreux autres langages ont été proposés [Hen99, MTW+99, FGK89, Hni03, FPÅ03, Hen03, AA, RA04, CIP+01, RAA03] montrant ainsi le grand intérêt suscité par ce sujet. L'utilisateur fournit les règles écrites dans ces langages décrivant les contraintes du problème et les paramètres de son instance qui sont ensuite combinés dans une phase de réécriture pour générer un CSP adapté au problème à résoudre. Apprendre une telle spécification à partir de problèmes résolus (des données historiques par exemple) fournirait un modèle qui pourrait être réutilisé dans un nouveau contexte avec différents paramètres. Par exemple, après la génération d'une spécification d'un emploi du temps scolaire, le modèle pourrait être paramétré avec les données actuelles comme le nombre de classes, d'enseignants, les nouvelles salles, etc.

Dans ce chapitre, nous présentons un cadre d'acquisition d'une telle spécification en utilisant la programmation logique inductive (ILP). Les exemples et contre-exemples pour le concept à apprendre sont définis comme des interprétations dans un langage logique que nous appellerons le *langage de description* et le CSP en sortie est exprimé par des

contraintes dans un *langage de contraintes*. La spécification est quant à elle écrite avec des règles du premier ordre qui sont associées à un ensemble de prédicats du langage de description (le corps de la règle) et un ensemble de prédicats du langage de contraintes (la tête de la règle). Nous n'utilisons pas directement de langage haut-niveau pour rester proche d'un système de règles mais les règles que nous apprenons ont le même niveau d'abstraction que ceux des langages mi-niveau comme OPL [Hen99], Essence' [FGJ+07] ou MiniZinc [NSB+07] (ces langages seront présentés dans la suite). En particulier, ils autorisent les paramètres et sont réécrits pour générer des CSP de différentes tailles. Nous verrons que trouver de telles règles est un vrai problème pour les techniques d'ILP classiques puisque nous tombons systématiquement dans des cas pathologiques de plateaux et de recherche aveugle.

La suite est organisée de la manière suivante. Nous commençons avec une vision informelle de notre cadre (section 4.2), introduisons le langage de règles (section 4.3), puis nous présentons le cadre d'apprentissage et donnons les clés pour comprendre comment le ramener à un cadre d'ILP classique (section 4.4). Nous verrons comment nous avons procédé pour évaluer les différentes méthodes dans la section 4.5 et détaillerons les limites rencontrées dans la section suivante. Finalement, nous verrons comment traduire une spécification écrite dans notre langage en un CSP avant de conclure.

Ce chapitre est inspiré des travaux que nous avons publiés dans [LLMV10].

4.2 Présentation générale du cadre

Afin de combler le vide entre les langages de programmation par contrainte et l'ILP, nous utilisons un cadre plus complexe. Dans cette section, nous donnons un rapide aperçu de notre cadre représenté dans le diagramme de flux de la Figure 4.1, ainsi que des justifications pour nos choix. Pour illustrer ces concepts, nous utiliserons l'exemple très simple de la coloration de graphe.

Premièrement, les exemples et contre-exemples sont représentés par une interprétation logique, qui peut être vu comme un ensemble d'atomes clos. Prenons l'exemple de la coloration de graphe où le but est de colorer chaque sommet d'un graphe de telle sorte que deux sommets voisins n'aient pas la même couleur. Deux graphes sont représentés dans la Figure 4.2, l'un, appelé $g1$, avec une mauvaise coloration, et un autre, appelé $g2$, correctement coloré. Pour le premier graphe $g1$, sa description logique est donnée par l'interprétation { *n(g1,a), n(g1,b), n(g1,c), n(g1,d), adj(g1,a,b), adj(g1,a,d), adj(g1,b,c), adj(g1,b,d), adj(g1,c,d), col(g1,a,blue), col(g1,b,green), col(g1,c,green), col(g1,d,red)...* } où le prédicat n représente les nœuds du graphe, *adj* la relation de voisinage et *col* pour la couleur. Nous pouvons également donner une description similaire pour le graphe $g2$. Notons que les exemples et contre-exemples peuvent utiliser des ensembles de constantes différents, avoir plus ou moins de nœuds et une relation de voisinage différente. Cela nous donne un cadre plus flexible que le précédent système de l'état de l'art CONACQ. Pour simplifier le langage de description et ainsi être capable d'inférer un CSP, les constantes sont typées. Dans l'exemple, il y a les types *nœud* et *couleur*.

La règle que nous souhaitons apprendre pour la coloration de graphe est :

$$col(G, X, A) \wedge col(G, Y, B) \wedge adj(G, X, Y) \rightarrow A \neq B$$

où chaque variable est implicitement universellement quantifiée. Cette règle décrit les conditions que doivent vérifier un graphe bien coloré indépendamment du graphe à colorer dans

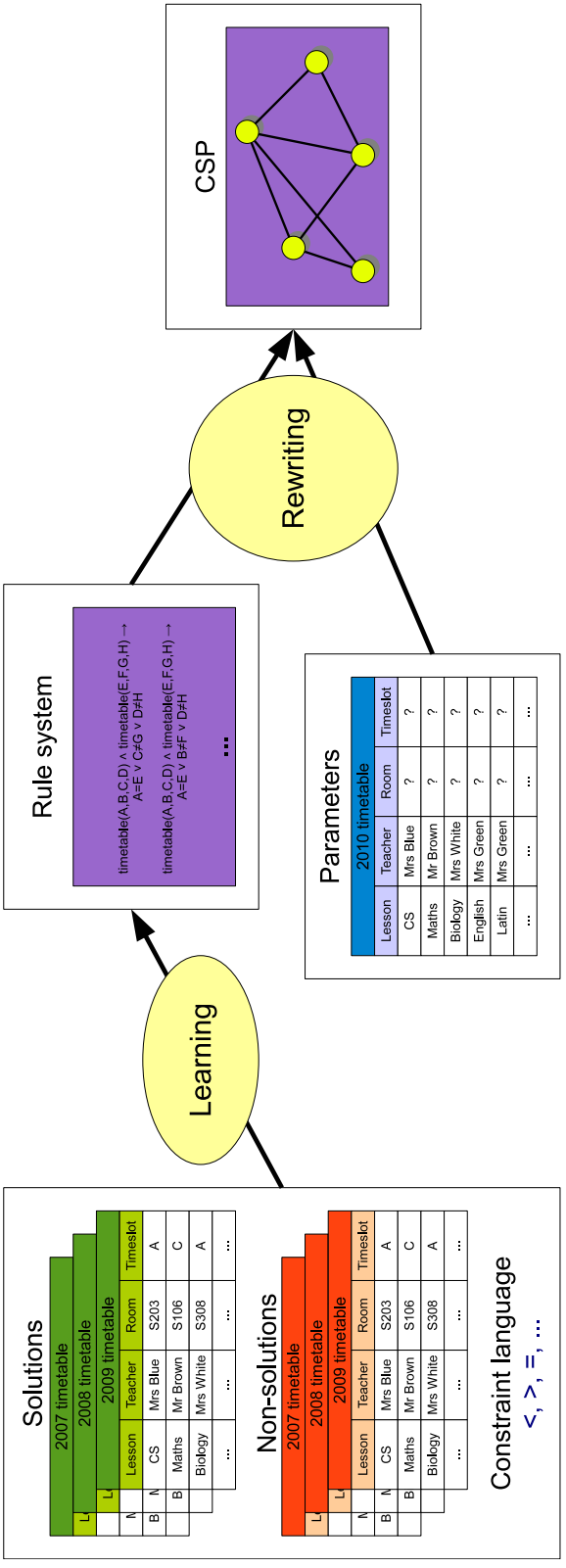

FIGURE 4.1 – Flux d'apprentissage de problèmes contraints

le problème réel. Nous pourrons plus tard utiliser la règle pour construire un CSP pour

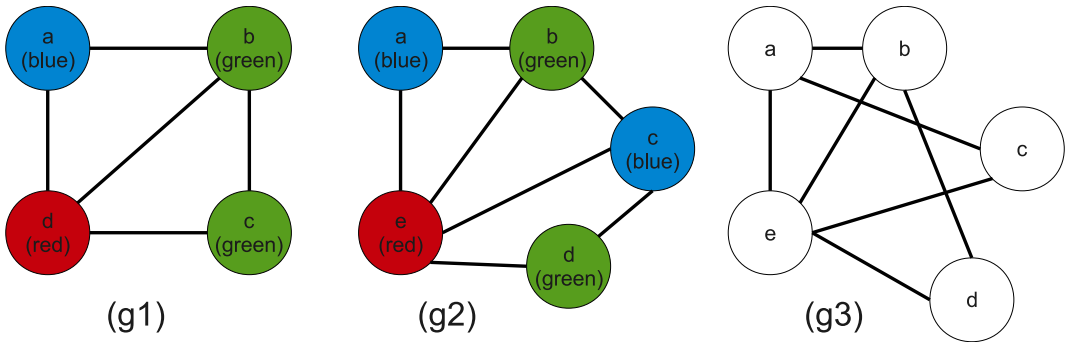

FIGURE 4.2 – (g1) mauvaise coloration, (g2) bonne coloration, (g3) à colorer

$$
\begin{aligned}
X &= \{ColA, ColB, ColC, ColD, ColE\} \\
D &= \{red, yellow, green, blue\} \\
C &= \{ \\
&\quad ColA \neq ColB \\
&\quad ColA \neq ColC \\
&\quad ColA \neq ColE \\
&\quad ColB \neq ColD \\
&\quad ColB \neq ColE \\
&\quad ColC \neq ColE \\
&\quad ColD \neq ColE \\
&\} \\
\end{aligned}
$$

FIGURE 4.3 – CSP construit pour résoudre le problème de coloration de graphe de la Figure 4.2

colorer un graphe spécifique comme le graphe $g3$ de la Figure 4.2. Nous supposons que l'utilisateur fournit une description du graphe qu'il souhaite coloré, en donnant une description similaire mais incomplète du graphe : { *n(g3,a), n(g3,b), n(g3,c), n(g3,d), n(g3,e), col(g3,a, ?), col(g3,b, ?), col(g3,c, ?), col(g3,d, ?), col(g3,e, ?), adj(a,b), adj(a,c), adj(a,e), adj(b,d), adj(b,e), adj(c,e), adj(d,e)* }[1]. Les points d'interrogation représentent les données inconnues que l'utilisateur souhaite compléter. Il faudra également que l'utilisateur renseigne les valeurs que pourront prendre ces données inconnues (c'est-à-dire le domaine). Ces données inconnues seront remplacées par des variables afin d'obtenir *col(g3,a,ColA), col(g3,b,ColB), col(g3,c,ColC), col(g3,d,ColD), col(g3,e,ColE)*. Si nous parvenons à apprendre la règle ci-dessus, toutes les substitutions possibles entre la partie gauche de la règle et la description pourront alors être calculées. Par exemple la substitution { *G/g3, X/a, A/ColA, Y/b, B/ColB* } permet d'ajouter la contrainte $ColA \neq ColB$ au CSP en appliquant la substitution sur la partie droite de la règle. En considérant toutes les substitutions autorisées par les types de variables, nous obtenons le CSP définissant la *colorabilité* de $g3$ qui est détaillé dans la Figure 4.3.

Exprimé comme une clause, la règle ci-dessus donne :

$$\neg col(G, X, A) \vee \neg col(G, Y, B) \vee \neg adj(G, X, Y) \vee A \neq B$$

Une spécification est composée d'une conjonction de clauses comme celle-ci. Comme présenté au chapitre 3, la plupart des systèmes d'ILP apprennent des formules en forme nor-

1. Il faudrait également fournir les symétries du prédicat *adj* comme $adj(b, a)$

male disjonctive (DNF) plutôt qu'en forme normale conjonctive (CNF). Pour pouvoir utiliser ces systèmes, nous considérons simplement la négation des règles à apprendre, les variables universelles deviennent existentielles, et nous échangeons les exemples et contre-exemples. Une transformation similaire peut être trouvée dans [Lae02]. La conjonction à apprendre sera comme suit et décrit une mauvaise coloration :

$$col(G, X, A) \wedge col(G, Y, B) \wedge adj(G, X, Y) \wedge A = B$$

En théorie, nous pouvons simplement donner ces problèmes à des outils d'apprentissage relationnel [QCJ93, AR06, Mug95, MSTN09, MF90, Lae02, RL95, Sri, Tan03] mais en pratique, nous verrons que ces systèmes échouent : soit les approches descendantes échouent à cause d'un plateau rendant la recherche aléatoire, soit les approches ascendantes font face à des tests de couverture ou de subsomption trop coûteux.

4.3 Décrire un modèle

Un langage de modélisation donne un moyen de spécifier un CSP de manière abstraite. Nombreux sont les langages de modélisation existants utilisant des variables haut-niveaux, comme les ensembles, des fonctions, des opérateurs arithmétiques, des quantificateurs existentiels et universels ou des arguments pour paramétrer leur modèle. En pratique, les paramètres sont habituellement fournis dans un fichier séparé et joints au modèle pour produire un CSP. Comme décrit précédemment, nous souhaitons apprendre un modèle avec un tel niveau d'abstraction pour le problème cible. Cependant, même si nous aimerions apprendre une spécification dans de tels langages, il est difficile de traiter directement de tels langages dû à leur excès d'expressivité. Nous continuons cette section avec un présentation des principaux langages de modélisation existants puis nous détaillerons le langage que nous avons défini, adapté à notre problématique d'apprentissage.

4.3.1 Langages de modélisation

Nous avons vu au chapitre 2 que pour écrire un CSP, il suffisait de donner son ensemble de variables, leurs domaines et les contraintes décrivant le problème souhaité. Cette manière d'exprimer un modèle est certes simple à comprendre pour un débutant mais devient vite compliquée quand il s'agit d'exprimer des problèmes plus complexes. Prenons par exemple le problème des 8-reines, où le but est de placer 8 reines sur un plateau d'échec de taille 8×8 tel qu'aucune reine n'attaque une autre. Si nous choisissons de représenter nos reines de la manière suivante : 8 variables X_1, X_2, ..., X_8 ayant comme domaine [1..8] avec pour sémantique, la reine i est placé à la ligne X_i et la colonne i. En choisissant ces variables, nous avons déjà exprimé que 2 reines ne peuvent pas être sur la même colonne. Exprimer les contraintes qui interdisent à deux reines d'être sur la même ligne n'est pas non plus compliqué, il suffit d'ajouter les contraintes $X_i \neq X_j$ où i, j varient entre 1 et 8 et $i < j$. Cependant, décrire la contrainte sur la diagonale est plus complexe, si nous n'avons à notre disposition que des contraintes basiques comme on peut le constater dans la Figure 4.4. La nécessité d'utiliser des variables auxiliaires (les variables T_i) n'est pas forcément évidente et naturelle pour un utilisateur débutant surtout quand le problème devient plus complexe.

Le CSP construit de cette manière n'est donc pas toujours facile et peu réutilisable. Notamment la nécessité de spécifier de manière exacte les variables empêche de formuler le problème général des n-reines. Ainsi à la fin des années 90, la communauté de programmation par contraintes a proposé des langages avec un niveau d'abstraction plus élevé,

Modèle bas-niveau

$X = \{X_1, X_2, \ldots, X_8, T_1, T_2, \ldots\}$

$D = [1, , 8]$

$C = \{X_1 \neq X_2, X1 \neq X_3, \ldots$
$X_1 + 1 = T_1, X_2 + 2 = T_2, T_1 \neq T_2$
$X_1 - 1 = T_3, X_2 - 2 = T_4, T_3 \neq T_4$
$\ldots\}$

Modèle en OPL

```
int n=... ;
range Domain=1..n ;
var Domain queen[Domain] ;

subject to{
    forall(ordered i, j in Domain){
        queen[i]<>queen[j] :
        queen[i]+i<>queen[j]+j :
        queen[i]-i<>queen[j]-j :
        }
    }
```

Modèle en Essence

Given $n : int$
Where $n > 0$
Letting $Index$ *be domain* $int(*1..n*)$
Find $arg : function\ Index \rightarrow (*bijective*)Index$
Such that
$\forall_{q_1,q_2 \in Index}\ q_1 \neq q_2 \rightarrow$
$|arg(q_1) - arg(q_2)| \neq |q_1 - q_2|$

Modèle en Zinc

```
int : n ;
type Domain=1..n ;
array[Domain] of var Domain : q ;

predicate noattack(Domain : i,j,
  var Domain : qi,qj) =
    qi !=qj / qi+i != qj+j / qi-i !=qj-j ;

constraint forall (i,j in Domain where i<j)
    noattack(i,j,q[i],q[j]) ;
```

Figure 4.4 – Modélisation des n-reines dans différents langages

permettant d'une part de paramétrer ses modèles (en indiquant par exemple que l'on a n variables dans un tableau X), ou encore de décrire les contraintes dans un langage mathématique plus naturel que les contraintes basiques. Ainsi un modèle décrit un ensemble de CSP et pour en obtenir un en particulier il suffit d'instancier les paramètres du modèle. Un modèle est alors caractérisé par trois parties :

- la première décrit les paramètres utilisés (les entrées permettant de l'instancier) ;
- la seconde déclare les variables et leur domaine ;
- la troisième exprime les contraintes du CSP.

OPL[Hen99] est un des premiers langages proposant ces possibilités. Le problème des n-reines peut s'exprimer en paramétrant n et en exprimant plus simplement la contrainte de la diagonale comme on peut le voir à la Figure 4.4. La plupart des solveurs proposent désormais ce genre de langage en plus de celui bas-niveau décrit au début de la section.

La communauté s'est depuis intéressée à concevoir des langages avec un niveau d'abstraction de plus en plus élevé. On trouve maintenant des langages de modélisation dit haut-niveau, voire également appelé langage de spécification. Ces langages proposent, pour se différencier des langages mi-niveau, des types plus complexes que les variables élémentaires ou les tableaux comme par exemple les ensembles, les séquences, les permutations, les partitions... De plus, ils offrent la possibilité de combiner ces types en les imbriquant les uns dans les autres (par exemple une séquence de permutations de variables entières).

Ces types représentent des collections où les objets les composant sont qualifiés de non-nommés[FM06], permettant ainsi de manipuler ces types sans distinguer les éléments qui les composent, toujours dans l'esprit d'augmenter l'abstraction du langage. Essence[FGJ+07] et Zinc[MNR+08] sont des exemples de tels langages. Ainsi l'utilisation d'une bijection en Essence (voir Figure 4.4) permet d'exprimer à la fois les contraintes sur les lignes et sur les colonnes des reines. En effet, une bijection des lignes vers les colonnes garantit que les lignes sont toutes différentes entre elles et de même pour les colonnes. Il ne reste alors plus qu'à exprimer la contrainte sur la diagonale. On peut également remarquer la possibilité en Zinc de définir ses propres prédicats (par exemple, le prédicat `noattack` dans l'exemple des reines).

Que le modèle soit écrit dans un langage mi-niveau ou haut-niveau, il faut pouvoir à un moment le compiler, le traduire, pour retrouver la forme basique des CSP (le triplet $\langle X, D, C\rangle$). Si traduire un modèle mi-niveau reste relativement simple : déplier des boucles et décomposer des contraintes complexes avec des contraintes basiques, la traduction des modèles haut-niveaux est plus complexe. Derrière les types complexes proposés par ces langages se cachent des contraintes qu'il faut exprimer, en ajoutant des contraintes au modèle ou en proposant des encodages de variables particulières (un peu comme le choix des variables des 8-reines permettant de décrire la contrainte sur les colonnes sans poser de contraintes). Les créateurs de Zinc et Essence ont choisi de proposer un langage mi-niveau, respectivement FlatZinc et Essence' ([FGJ+07, MNR+08]), permettant ensuite de se rapprocher des compilateurs développés pour les langages tel OPL. Une première étape est donc la traduction d'un modèle haut-niveau en un modèle mi-niveau puis la seconde d'un langage mi-niveau au langage bas-niveau d'un solveur.

4.3.2 Notre langage

Dans cette partie, nous proposons de nous concentrer sur un langage plus simple mais permettant la définition d'une large gamme de problèmes. Nous choisissons un sous-ensemble de la logique du premier ordre en retenant la notion de paramètre. Nous avons écarté les caractéristiques comme les fonctions trouvées communément dans les langages de modelage ciblés pour les humains, mais nous avons gardé la capacité de générer des CSP pour un ensemble d'instances du problème.

Une des contributions connexes de nos travaux [LLMV10] a été de proposer un langage de modélisation intermédiaire basé sur un système de règles pensé pour une utilisation en apprentissage automatique. Basé sur un sous-ensemble de la logique du premier ordre (voir Chapitre 3), une spécification de problèmes contraints, appelé dans la suite CPS pour *Constraint Problem Specification*, dans ce langage consiste en un ensemble de règles décrivant quand une contrainte doit être posée dans une instance du problème.

Soient T un ensemble de types, $V = (V_t)_{t\in T}$ et $(Const_t)_{t\in T}$ respectivement un ensemble typé de variables et de constantes. Nous rappelons qu'un terme est soit une variable, soit une constante. Les prédicats ont également des types et sont divisés en deux groupes disjoints, P_D et P_C, correspondant respectivement aux prédicats pouvant être utilisés dans le corps et la tête d'une règle. Les prédicats du corps forment le *langage de description*. Ils sont utilisés pour exprimer des exemples et contre-exemples et pour introduire les variables des règles. Ils ont également une déclaration de mode (voir Chapitre 3 section 3.2.2) : chaque argument a un mode décrivant s'il est utilisé comme une entrée ou une sortie. Les entrées sont notées '+' et les sorties '-'. Par exemple, le prédicat *sum(X,Y,Z)* avec la sémantique

$X + Y = Z$ et le mode *sum(+,+,-)* définit que le dernier argument est la somme des deux premiers. Les prédicats de la tête forment le *langage de la tête* et sont utilisés pour définir les contraintes qui devront être posées quand les prédicats du corps sont vrais. Ces prédicats sont les précurseurs des contraintes et seront transformés en contraintes pendant la phase de réécriture (voir Figure 4.1 et section 4.7). Nous rappelons qu'un atome est une expression de la forme $P(t_1, \ldots, t_k)$, où P est un prédicat d'arité k et $t_1, \ldots, t_k$ sont des termes.

La syntaxe de nos règles est la suivante :

$$
\begin{aligned}
\textit{rule} &::= \forall\ \textit{variables} :\ \textit{body}\ \rightarrow \textit{head} \\
\textit{variables} &::= \textit{vs} \in \texttt{TYPE} \mid \textit{variables}, \textit{variables} \\
\textit{vs} &::= \texttt{VARIABLE} \mid \textit{vs}, \textit{vs} \\
\textit{body} &::= \texttt{BODY\_ATOM} \mid \textit{body}\ \wedge\ \textit{body} \\
\textit{head} &::= \texttt{HEAD\_ATOM} \mid \neg\texttt{HEAD\_ATOM} \\
&\quad \mid \textit{head}\ \vee\ \textit{head}
\end{aligned}
$$

Le terminal `BODY_ATOM` correspond à un atome avec un prédicat de *description* et le terminal `HEAD_ATOM` de *tête* à un atome avec un prédicat de *tête*.

Une règle est donc formée d'un corps sous forme d'une conjonction de littéraux positifs (sans négation) et d'une tête prenant la forme d'une disjonction de littéraux.

La Figure 4.5 présente quelques exemples de problèmes spécifiés dans notre langage. Les quantificateurs universels ont été omis pour rendre la figure plus lisible.

Le premier exemple correspond à la coloration de graphe décrit précédemment. Le second est un emploi du temps scolaire simplifié. $\mathit{timetable}(L, T, R, S)$ signifie qu'un cours L est donné par un enseignant T dans la salle R sur le créneau S. La première règle impose que deux cours ne puissent avoir lieu au même moment s'ils sont dans la même salle. La seconde assure qu'un enseignant ne puisse donner deux cours différents pendant la même période. Le troisième exemple est un problème d'ordonnancement de tâches où $\mathit{schedule}(J, T, B, E, M)$ signifie que la tâche J de type T est effectuée sur la machine M entre les dates B et E. La première règle spécifie que le début d'une tâche doit être avant sa fin ; la seconde que deux tâches ne peuvent être effectuées au même moment par la même machine ; la dernière que certaines tâches doivent être faites avant d'autres en respect de leur types (*prev* décrit l'ordre sur les types). Enfin, le problème des n-reines, comme dit précédemment, consiste à placer n reines sur un plateau d'échec de taille $n \times n$ de manière à ce qu'aucune reine n'attaque une autre. Autrement dit, une reine ne peut être placée sur la même ligne ou la même colonne qu'une autre (les deux premières règles) ou encore être placée sur la même diagonale qu'une autre (troisième règle). Le prédicat *gap* donne l'écart entre deux entiers, c'est-à-dire $\mathit{gap}(X, Y, Z) \Leftrightarrow |X - Y| = Z$.

En contraste avec les langages de modélisation classiques, la présence des disjonctions rend difficile la compréhension pour un utilisateur humain. Cependant, un modèle dans ce langage est automatiquement compilé dans un CSP et dans des expressions simplifiées dans lesquelles beaucoup de disjonctions disparaissent.

4.4 Apprendre un CPS comme un problème d'ILP

La première étape de notre cadre consiste en l'apprentissage d'un CPS décrivant le problème cible. Dans cette section, nous présentons comment l'apprentissage d'un tel modèle peut être vu comme un problème d'apprentissage classique en ILP.

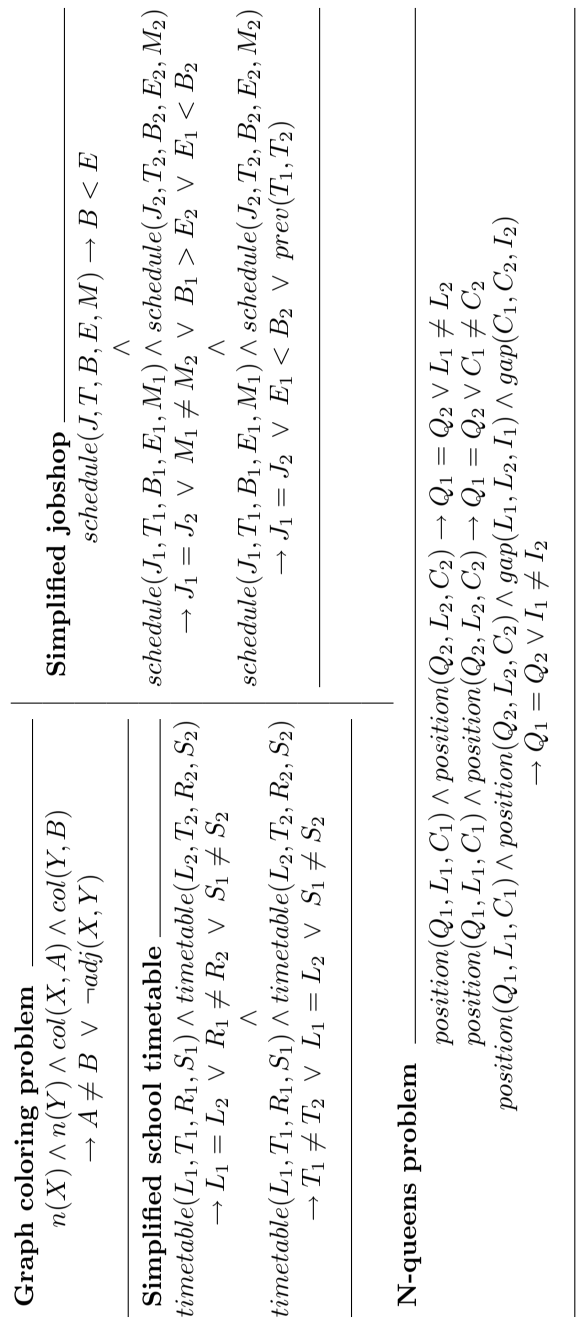

FIGURE 4.5 – Quelques exemples de CPS : toutes les variables sont quantifiées universellement

Premièrement, nous présentons les entrées nécessaires au traitement. Si nous nous référons à la Figure 4.1, la phase d'apprentissage nécessite un ensemble de solutions, par exemple dans le cas du problème des emplois du temps, des exemples d'emplois du temps corrects, un ensemble de non solutions, qui sont des exemples incorrects d'emplois du temps, et une base de connaissances donnant les types de contraintes pouvant être utilisés. Ces descriptions de contraintes sont soit données en extension soit sous la forme de clauses de Horn. De plus, la déclaration des modes et les types est également donnée dans cette base.

Le but est d'apprendre, dans notre langage, une définition pour un CPS, notée CS dans la suite, qui discrimine correctement les solutions des non-solutions. La discrimination d'une règle est calculée avec le test de couverture : informellement une règle C couvre un exemple e si e ne viole pas les contraintes exprimées par C.

Les solutions et non-solutions sont données sous forme de tables. Pour illustrer, nous utiliserons l'exemple de l'emploi du temps, pour lequel il n'y a besoin que d'une table donnant les caractéristiques de chaque cours. Cette table peut être vue comme une relation *timetable* d'arité 4, les arguments étant le nom du cours, l'enseignant, la salle et le créneau horaire. Pour la coloration de graphe, il y aurait deux tables n et col correspondant aux nœuds du graphes et aux couleurs de chacun de ces nœuds.

Un exemple positif d'un emploi du temps pourrait être :

$$\begin{aligned} e^+ : \{ \quad & timetable(ComputerScience, MrsBlue, s203, A), \\ & timetable(Maths, MrsBrown, s106, C), \\ & timetable(Biology, MrsWhite, s308, A), \\ & timetable(Maths, MrsBrown, s106, B)\} \end{aligned}$$

alors qu'un exemple négatif serait :

$$\begin{aligned} e^- : \{ \quad & timetable(ComputerScience, MrsBlue, s203, A), \\ & timetable(Maths, MrsBrown, s106, C) \\ & timetable(Biology, MrsWhite, s203, A), \\ & timetable(Maths, MrsBrown, s105, C)\} \end{aligned}$$

On peut remarquer que le contre-exemple e^- contient deux erreurs. $MrsBrown$ ne peut enseigner deux cours différents pendant le créneau C et la salle $s203$ est utilisée deux fois pendant le même créneau par des personnes différentes.

La base de connaissances contient des prédicats comme l'égalité et la différence permettant de comparer les cours, les enseignants, les salles et les créneaux horaires.

Un ensemble de règles définissant un CPS est équivalent à une CNF. Pour passer à une DNF et ainsi utiliser les systèmes d'apprentissage disjonctif décrits au Chapitre 3 section 3.4.3, il suffit de rechercher la négation du concept cible(voir [Lae02], section 3.4.3).

Afin d'illustrer ce point, considérons l'exemple de l'emploi du temps, la CNF correspondant au CPS de la Figure 4.5 est :

$$\begin{aligned} \forall \quad & L_1, L_2 \in Lessons, T_1, T_2 \in Teachers, \ldots : \\ & \neg timetable(L_1, T_1, R_1, S_1) \vee \neg timetable(L_2, T_2, R_2, S_2) \end{aligned}$$

$$\vee L_1 = L_2 \ \vee \ R_1 \neq R_2 \ \vee \ S_1 \neq S_2$$
$$\bigwedge$$
$$\forall \ L_1, L_2, \ldots :$$
$$\neg timetable(L_1, T_1, R_1, S_1) \vee \neg timetable(L_2, T_2, R_2, S_2)$$
$$\vee T_1 \neq T_2 \ \vee \ L_1 = L_2 \ \vee \ S_1 \neq S_2$$

La négation produit la DNF :

$$\exists \ L_1, L_2 \in Lessons, T_1, T_2 \in Teachers, \ldots :$$
$$timetable(L_1, T_1, R_1, S_1) \wedge timetable(L_2, T_2, R_2, S_2)$$
$$\wedge L_1 \neq L_2 \ \wedge \ R_1 = R_2 \ \wedge \ S_1 = S_2$$
$$\bigvee$$
$$\exists \ L_1, L_2, \ldots :$$
$$timetable(L_1, T_1, R_1, S_1) \wedge timetable(L_2, T_2, R_2, S_2)$$
$$\wedge T_1 = T_2 \ \wedge \ L_1 \neq L_2 \ \wedge \ S_1 = S_2$$

Pour retrouver le cadre du Chapitre 3 et l'apprentissage disjonctif de clauses de Horn, il suffit d'ajouter une tête à cette conjonction représentant notre modèle de contraintes.

Pour apprendre ce concept, les ensembles d'exemples positifs et négatifs doivent être inversés. Les exemples positifs deviennent les négatifs et vice versa. Pour retrouver le concept cible (la définition du CPS) il suffira d'inverser les connecteurs, $\wedge$ devient $\vee$ et $\vee$ devient $\wedge$, et les quantificateurs, $\exists$ devient $\forall$.

4.5 Évaluation de la phase d'apprentissage

Une fois le cadre d'apprentissage donné se pose le problème de l'évaluation des différents systèmes d'ILP. Pour évaluer un système comme CONACQ, les auteurs ont testé leur système sur plusieurs exemples : des CSP aléatoires, des CSP aléatoires avec des motifs et des problèmes ciblés comme le SUDOKU. Les problèmes aléatoires ne peuvent être exploités avec notre approche. En effet, pour qu'elle ait un intérêt, notre approche doit s'appliquer sur des problèmes structurés où des problèmes proches peuvent être générés. Nous nous sommes donc plutôt intéressés à des problèmes particuliers pouvant être modélisés par un CPS. Nous avons retenu la coloration de graphe, un emploi du temps simplifié, un ordonnancement de tâches et les n-reines. Ces problèmes ont des difficultés progressives comme nous pouvons le voir en Figure 4.5.

Pour produire les jeux de données, nous avons généré aléatoirement les ensembles d'exemples positifs et négatifs nécessaires à la phase d'apprentissage. Pour produire des solutions, nous avons choisi au hasard les différentes tailles pour chaque exemple (par exemple, le nombre de sommets pour la coloration de graphe), et nous avons résolu le CSP associé avec une heuristique de branchement choisissant la variable et la valeur aléatoirement. Pour produire des non-solutions, nous avons procédé d'une manière similaire mais les contraintes sont réifiées, c'est-à-dire qu'il n'est pas obligatoire qu'elles soient vérifiées. Pour réifier une contrainte c_i, il suffit d'ajouter une variable booléenne b_i au modèle et de remplacer c_i par $c_i \Leftrightarrow b_i$. Nous nous sommes cependant assurés qu'il y avait au moins une contrainte non satisfaite (sinon, nous aurions produit une solution) ce qui revient à s'assurer qu'au moins une des variables b_i est égale à 0 (ou faux). Pour résumer, le protocole de génération est le suivant :

- Construire un modèle mi-niveau paramétré
- Pour chaque solution :

  - Choisir des valeurs aléatoires pour les paramètres
  - Résoudre le CSP avec une heuristique de branchement aléatoire (choix de la variable et de la valeur à lui affecter au hasard)
- Pour chaque non-solution :
  - Choisir des valeurs aléatoires pour les paramètres
  - Réifier les contraintes
  - Ajouter la contrainte $\sum_{c_i \in C} b_i < |C|$
  - Résoudre le CSP avec une heuristique de branchement aléatoire

Comme nous le verrons par la suite, nous avons choisi pour résoudre la tâche d'apprentissage du CPS des approches de type *separate-and-conquer* (voir chapitre 3, section 3.4.2). Si nos motivations étaient ainsi de limiter la complexité et ainsi rendre notre approche possible en pratique, ce type d'apprentissage ne permet d'apprendre pour une non-solution qu'une seule des contraintes qu'elle viole. Pour illustrer ce phénomène, nous avons généré deux types de jeux de données par problème. Un premier où les exemples sont complètement aléatoires, version appelée par la suite *alea* et un autre où seulement 10 exemples sont complètement aléatoires et pour chacune des règles décrivant le problème, 10 exemples violant uniquement cette règle sont générés, cette version est ensuite nommé $alea + rules$. Par exemple, pour les n-reines, les jeux de données comptent 10 non-solutions aléatoires, violant quasiment toutes les règles, 10 non-solutions ne violant que la contraintes sur les lignes, 10 autres la contrainte sur les colonnes et enfin 10 non-solutions avec des reines s'attaquant en diagonale. La Figure 4.6 décrit les caractéristiques des jeux de données. La ligne des paramètres correspond à la taille des ensembles utilisés par les règles (par exemple l'ensemble des sommets pour les graphes). Pour générer une solution ou non-solution, une valeur aléatoire est choisie dans la plage donnée entre crochets. Les deux lignes suivantes donnent le nombre de solutions/non-solutions générées pour l'ensemble servant à l'apprentissage. Ce nombre dépend du nombre de règles (10 exemples par règle + 10 aléatoire) sauf pour le cas particulier de la coloration de graphe où il n'y a qu'une seule règle. Pour la version *alea* des jeux de données, le même nombre a été généré sans contraintes sur les règles à violer (soit 40 non-solutions complètement aléatoires pour les n-reines par exemple).

Jeux de données	Graph coloring	School Timetable	Jobshop	n-queens
Paramètres	sommets [1..20]	cours [2..30] enseignants [2..10] salles [2..10] créneaux [2..10]	tâches [2..10] types [2..10] machines [2..10] temps [2..10]	reines [2..20]
Nombre de solutions	10	30	40	40
Nombre de non-solutions	10	30	40	40
Taille de l'ensemble de test	180	180	180	180

FIGURE 4.6 – Jeux de données générés

Nous avons également généré des ensembles de solutions/non-solutions pour évaluer la qualité des règles apprises par les différents systèmes. Ces exemples sont générés de la même manière que précédemment. Cela évite un apprentissage par cœur du problème, c'est-à-dire un CPS trop spécifique aux exemples fournis au système d'apprentissage. Le résultat de ce calcul sera appelé dans la suite la *précision*. Plus cette valeur sera haute, plus nous jugerons que le CPS appris est de qualité. Pour générer, ces jeux de données nous avons procédé de manière similaire aux jeux de données $alea + rules$.

Nous avons testé les algorithmes d'ILP (de type *separate-and-conquer*) usuel.

– des approches top-down, les plus répandues dans la littérature, avec plusieurs configurations du système Aleph[Sri] ou des systèmes comme Progol[Mug95, MSTN09], Foil[QCJ93], ICL[Lae02, RL95] et Propal[AR06] ;
– des approches bottom-up comme golem[MF90] ou progolem[MSTN09].

Ces systèmes n'ont pas amené à des résultats satisfaisants. La plupart échouaient sur les problèmes plus complexes comme l'ordonnancement de tâches ou les n-reines. Seules des approches complètes, très coûteuses en temps, nous ont permis d'obtenir des résultats satisfaisants. Cela nous a poussé à étudier la structure de nos problèmes et les difficultés qu'ils généraient pour les systèmes d'apprentissage classique. Nous avons proposé une nouvelle approche qui sera détaillée au chapitre 5. Les résultats obtenus avec ces systèmes (dans le cas où ils ont pu terminer l'apprentissage) seront également détaillés dans ce chapitre afin de les comparer avec notre approche.

Nous continuons avec une présentation des difficultés rencontrées avec nos jeux de données expliquant pourquoi les systèmes existants échouent.

4.6 Limite rencontrée

Nous avons rencontré de sérieuses difficultés lors de la phase d'expérimentations, liées à l'étape d'apprentissage de CPS.

D'une part les approches ascendantes échouaient à cause du coût trop important des tests de couverture ou de subsomption (comme évoqué au chapitre 3, ce test est NP-difficile). En effet, la taille des exemples est trop importante pour être traitée.

Quant aux approches descendantes, il y a deux cas. D'abord les approches complètes, énumérant progressivement les hypothèses de l'espace de recherche jusqu'à en trouver une satisfaisante, ne sont guère utilisables à cause de la taille trop grande de l'espace de recherche. Elles ont donc un coût trop élevé. Nous nous sommes, par conséquent, intéressés aux approches incomplètes de type *hill-climbing* et *beam search* permettant dans certains cas de trouver des règles sans risquer de parcourir l'ensemble des hypothèses possibles. Seulement pour être efficaces, ces méthodes utilisent des heuristiques basées sur la couverture des hypothèses. Cependant, il y a peu de règles discriminant les exemples dans l'espace de recherche. D'autre part, les règles des CPS sont très sensibles et si on enlève un seul littéral, elles ne sont plus du tout discriminantes. Prenons par exemple la coloration de graphe. La règle cible est la suivante : $CS(I) \leftarrow n(I,X) \wedge n(I,Y) \wedge col(I,X,A) \wedge col(I,Y,A) \wedge adj(I,X,Y)$. Quel que soit le littéral qu'on retire de cette règle, elle risque de couvrir tous les exemples mêmes les graphes colorés (si on suppose que tous les nœuds des graphes sont colorés). Par conséquent, les approches descendantes ne peuvent localiser la zone de l'espace de recherche où se situent les règles discriminantes et donc se déplacer pour trouver une règle satisfaisant les contraintes de complétude et correction.

On parle dans ce cas de *recherche aveugle*. Que l'approche soit complète ou non, l'algorithme n'est pas efficace et l'apprentissage peut échouer. Dans notre cas, cela est dû à un phénomène de plateau où de grandes zones dans l'espace de recherche contiennent des hypothèses avec la même valeur de couverture rendant aléatoire l'utilisation des heuristiques. Prenons l'exemple sur la coloration de graphe suivant. L'hypothèse courante est $CS(I) \leftarrow n(I,X) \vee n(I,Y)$. Si on ne considère que les raffinements ajoutant un littéral à la règle parmi $\{col(X,A), col(Y,A), adj(X,Y)\}$, rien ne permet de distinguer un raffinement parmi d'autres. Tous les nœuds sont colorés et des nœuds adjacents se trouvent autant dans les solutions que les non-solutions. Imaginons maintenant qu'il y ait d'autres littéraux possibles produisant le même effet mais n'appartenant pas à une règle discriminante,

les stratégies de recherches incomplètes seront incapables de sélectionner un raffinement correct et le choix se fera de manière aléatoire. Ces phénomènes ont fait l'objet en ILP de récents travaux assez conséquents montrant les limites des approches de l'état de l'art [BGSS03, SGS01, AO08, RM92, SP91, OdCDPC05, Qui91].

En effet, et comme pour nos travaux, les approches de programmation logique inductive ont tendance à être conçues pour des données particulières correspondant à des problèmes précis et ne permettant pas la création d'algorithmes efficaces et donc réutilisables dans les cas généraux.

4.7 Traduction d'un CPS en CSP

Considérons à nouveau l'exemple de l'emploi du temps. Nous supposons que la phase d'apprentissage a permis d'obtenir un CPS décrivant le concept général d'emploi du temps. Il nous faut maintenant produire un CSP pour le problème réel : l'emploi du temps de cette année. Pour cela, l'utilisateur doit fournir des informations sur son problème. Ces informations prennent la forme de table incomplète représentant le prédicat *timetable* (Figure 4.1). D'une manière générale, il peut y avoir plusieurs tables, nous les appellerons dans la suite des extensions partielles. Ces extensions permettent de fixer les « paramètres » du CPS. L'objectif, pour l'utilisateur, est d'obtenir un CSP qui pourra, une fois résolu, compléter les *extensions partielles*. Dans la Figure 4.1, le problème consiste à déterminer les salles et les créneaux où les enseignants pourront faire leur cours. L'utilisateur doit alors fournir, en plus des extensions partielles décrivant les cours et leurs enseignants associés, les domaines correspondant aux données inconnues : les salles à sa disposition et les créneaux horaires pouvant être utilisés. Ces données sont très naturelles à fournir étant donné le problème actuel à résoudre.

Afin de décrire l'algorithme de traduction plus formellement, nous introduisons les définitions suivantes :

Définition 4.7.1 (Extension partielle)**.** *Étant donné un prédicat p d'arité k, une extension partielle de p est un couple (p, E) où E est un ensemble de tuples $\langle x_1, x_2, \ldots, x_k \rangle$ où x_i est :*

- *soit une constante ;*
- *soit le symbole ? signifiant que la valeur de x_i est inconnue.*

Ces extensions partielles seront fournies par l'utilisateur et permettent d'instancier le CPS afin de créer le problème de satisfaction de contraintes pour trouver les valeurs manquantes de ces extensions, (celles représentées par ?). Pour créer ce CSP, notre algorithme va progressivement instancier les règles en accord avec les extensions partielles fournies par l'utilisateur. Notre algorithme cherchera alors à trouver les substitutions des atomes des règles avec les constantes et les variables du CSP.

Définition 4.7.2 (Satisfaction d'un atome)**.** *Étant donnés un atome $p(X_1, \ldots, X_k)$, une extension partielle (p, E) et une substitution $\sigma = \{X_1/x_1, \ldots, X_k/x_k\}$, x_i étant soit une constante soit ?, nous dirons que σ satisfait p si $\langle x_1, \ldots, x_k \rangle \in E$.*

La traduction d'un CPS en CSP se déroule en deux étapes. Premièrement, les extensions partielles sont complétées avec des variables de CSP qui auront les domaines correspondant aux types d'arguments qu'elles représentent. Ensuite, pour chaque règle du CPS, toutes les substitutions possibles du corps seront produites et les contraintes correspondant à la tête seront ajoutées au CSP. L'algorithme 4.7 calcule ces deux étapes.

Algorithm : TRANSLATE($CS, ext, domains$)

```
1.  //Complète l'extension partielle
2.  // avec des variables de CSP et le domaine correspondant
3.  ext ← COMPLETE(ext, domains)
4.  //Initialise l'ensemble des variables
5.  //avec celles de ext
6.  vars ← GETVAR(ext)
7.  constraints ← ∅
8.  for each G → C ∈ CS
9.  // Génère toutes les substitutions du corps
10.    subst ← GENERATEALLSUBST(G, ext)
11.    for each σ ∈ subst
12. //Si il y a des atomes sans apparaiement
13. // dans ext, ajoute des variables auxiliaires
14. // et la contrainte
15.      for each atoms p(t_1, ..., t_k) ∈ G
16.      such that (p, _) ∉ ext
17.        vars.add(GETVAR(σ(p(t_1, ..., t_k))))
18.        constraints.add(σ(p(t_1, ..., t_k)))
19.  //Ajoute la contrainte
20.  //correspondant à la tête
21.      constraint.add(σ(C))
22.  //Retourne le CSP
23.  return CSP(vars, domains, constraints)
```

FIGURE 4.7 – Traduction d'un CPS en CSP

La première étape (ligne 3) consiste à compléter ext, en remplaçant les ? par des variables de CSP avec le bon domaine (donné par l'utilisateur). La seconde étape produit les contraintes à partir du CPS en fonction de l'instance du problème de l'utilisateur. Pour cela, l'algorithme génère toutes les substitutions possibles du corps G le satisfaisant dans ext. Étant donnée une substitution σ, le corps est satisfait si chaque atome $p(t_1, \ldots, t_k)$ de G est satisfait avec σ. Un atome $p(t_1, \ldots, t_k)$ est satisfait si $\sigma(p(t_1, \ldots, t_k))$ a un support dans ext ou, dans le cas où p est défini intentionellement, si $\sigma(p(t_1, \ldots, t_k))$ est valide selon la définition. Une difficulté peut apparaître quand p est intentionel. Quand il y a des variables de CSP en entrée d'un littéral, il faut générer des variables auxiliaires en sortie. Par exemple, considérons l'atome $sum(X, Y, Z)$ et la substitution $\{X/2, Y/v_1, Z/?\}$ où le domaine de v_1 est [2..6]. La substitution est complétée avec Z/v_2 et le domaine de v_2 est [4..8]. Une fois toutes les substitutions calculées, l'algorithme substitue la tête de la règle pour produire les contraintes du CSP (ligne 21). Ces contraintes sont des disjonctions de contraintes. Cependant, si nous considérons l'exemple de l'emploi du temps, plusieurs variables des contraintes ont déjà une valeur simplifiant alors la contrainte. Par exemple, la substitution $\{L_1/Latin, T_1/Mrs\ Green, R_1/v_1, S_1/v_2, L_2/English, T_2/Mrs\ Green, R_2/v_3, S_2/v_4\}$ de la seconde règle (Figure 4.5 permet de calculer la contrainte $Mr\ Green \neq Mrs\ Green \vee Latin = English \vee v_2 \neq v_4$ qui se simplifie en $v_2 \neq v_4$. Dans le cas général, toutes les contraintes ne peuvent être simplifiées. Pour illustrer, en prenant la substitution précédente et en l'appliquant à la première règle, il restera une disjonction.

4.8 Bilan et ouvertures

La motivation de ce nouveau cadre d'acquisition de problèmes par contraintes est dans le but de dépasser les limites des cadres de la littérature. Le but était de voir comment nous pouvions, à partir de problèmes proches du problème réel de l'utilisateur, généraliser un modèle décrivant à la fois ces problèmes et celui ciblé par l'utilisateur. Ces problèmes proches ont déjà été résolus par l'utilisateur d'une autre manière, par exemple à la main, mais leurs données, telles les variables et les domaines diffèrent par rapport à son nouveau problème. Nous avons donc proposé de passer par un langage de modélisation plus expressif que le langage de base bas-niveau des CSP consistant à fournir les ensembles X de variables, D des domaines et C des contraintes. Dans ce langage, X et D ne sont pas donnés. Il y a bien des variables et des domaines mais leurs descriptions sont fournies par l'utilisateur uniquement au moment où il souhaite créer une instance du problème, pouvant être résolue par un solveur. Nous avons choisi comme langage un sous-ensemble de la logique du premier ordre permettant d'exprimer une large classe de problème par contraintes. Ce choix a été préféré dans un souci de réutilisation des techniques, voire des systèmes, de programmation logique inductive. Cependant, comme le montrent les expérimentations présentées dans le chapitre suivant, les problèmes représentés par les CPS posent de sérieuses difficultés aux systèmes d'apprentissage ce qui nous a conduit à développer un algorithme tirant parti de la structure de ces modèles.

Ainsi, nous obtenons un cadre où l'utilisateur doit fournir des solutions et non-solutions de problèmes, proches mais suffisamment différents de son problème cible. Une fois le modèle appris, l'utilisateur fournit, sous forme de tables, les variables et domaines du problème dont il cherche une solution. Notre système construit alors un CSP qui peut être résolu par les nombreux solveurs existant en programmation par contraintes.

Ce cadre offre plusieurs pistes de développement. La première consiste à enrichir le langage des CPS afin de pouvoir traiter plus de types de problèmes. Ajouter la quantification existentielle, les agrégats pour proposer des contraintes plus complexes (comme Σ par exemple)... sont des pistes à étudier. Cela doit cependant se faire en gardant à l'esprit un souci de complexité pour la partie apprentissage. Cependant exploiter une structure plus forte dans notre langage, nous éloignant ainsi du cadre ILP traditionnel, pourrait permettre ce genre d'enrichissement. On pourrait considérer des *objets*, plus expressifs que les variables, mais pouvant simplifier à la fois l'expression des contraintes et l'apprentissage des règles. Dans les jeux de données que nous avons utilisés, nous pouvons constater la présence d'objets comme les reines, les tâches ou encore les cours, auxquels sont liés des caractéristiques, les positions des reines par exemples, sur lesquelles sont ensuite posées les contraintes. Nous pourrions avoir une approche consistant à chercher les contraintes uniquement en considérant un unique objet. Ensuite, il s'agirait de voir si on peut trouver des contraintes entre plusieurs objets de manière incrémentale. D'un tout autre côté, un reproche pouvant être fait à notre cadre concerne la nécessité par l'utilisateur de fournir des non-solutions de problèmes proches. En effet, si les solutions peuvent être facilement trouvées grâce à des données historiques, il est difficile d'imaginer que soient gardées des non-solutions. Nous sommes partis de l'idée que si l'utilisateur souhaitait utiliser notre cadre, c'est qu'il n'arrivait pas à trouver de solutions à son problème actuel. Nous pouvons, de notre point de vue, supposer qu'il soit alors facilement capable de produire des non-solutions. Cependant, cette manière de générer des contres-exemples peut produire des non-solutions très riches en informations qui d'une part ne seront pas toutes exploitées

(dès qu'une règle est apprise permettant de dire que c'est une non-solution, cet exemple est « jeté ») et d'autre part qui pourront induire en erreur les algorithmes d'apprentissage (en mélangeant deux règles par exemple) et ainsi perdre en efficacité. Nous pouvons alors sortir deux pistes. La première consisterait à mieux exploiter ces exemples, en les « compressant » pour faire ressortir des singularités[KZ10]. Une autre consisterait à ne demander que des solutions à l'utilisateur et ensuite trouver un moyen de générer de petits exemples que l'utilisateur aurait à étiqueter, c'est-à-dire des requêtes. De tel exemples permettraient de créer un espace de recherche plus petit dans la phase d'apprentissage et par conséquent d'utiliser des approches complètes.

Chapitre 5

Générer-et-Tester et opérateur bidirectionnel

5.1 Introduction

Pour répondre aux problèmes rencontrés dans la phase d'apprentissage, nous avons développé une nouvelle approche d'apprentissage, bi-directionnelle et de type générer-et-tester[LMV10]. Dans cette approche, nous proposons de réduire progressivement l'espace de recherche en ciblant la zone où pourrait se trouver une règle discriminante. Pour cela, nous maintenons deux hypothèses, $(\top_i, \bot_i)$, bornant l'espace de recherche. Notre opérateur de raffinement produira de nouvelles bornes $(\top_{i+1}, \bot_{i+1})$ telles que $\top_{i+1}$ sera plus spécifique que $\top_i$ et $\bot_{i+1}$ sera plus générale que $\bot_i$. La recherche s'arrêtera alors quand $\top_i = \bot_i$ si possible, couple représentant une unique hypothèse, la règle retournée par notre approche.

Ces deux hypothèses sont liées entre elles par une relation de saturation [Rou92, Rou90] permettant d'assurer que tous les littéraux de $\bot_i$ sont *atteignables* à partir de $\top_i$, c'est-à-dire peuvent être ajoutés à $\top_i$ en respectant les entrées/sorties des prédicats.

L'intérêt de maintenir ces deux bornes est qu'elles permettent de rendre plus efficace l'utilisation d'approche incomplète grâce à des heuristiques plus précises sur des données structurées comme les CPS.

Ce chapitre se divise de la manière suivante. La première partie décrit le langage des exemples utilisé dans notre approche avec notamment une description de l'opération de saturation[Rou92, Rou90] permettant de compléter les exemples avec des informations présentes dans la base de connaissances. Ensuite, nous caractériserons notre espace de recherche en commençant par la description du langage des hypothèses (section 5.3) puis l'espace de recherche que nous utilisons (section 5.4). Nous pourrons alors présenter notre opérateur de raffinement avec ses principales caractéristiques puis la stratégie d'utilisation dans le cadre de l'apprentissage de CPS. Nous finirons par les expériences liés au chapitre précédent, montrant l'efficacité de notre technique dans la tâche d'apprentissage de notre cadre d'acquisition de CSP.

Ce chapitre est inspiré des travaux publiés dans [LMV10] en collaboration avec Lionel Martin et Christel Vrain.

5.2 Langage des exemples

Nous avons vu dans le chapitre précédent que nous pouvions représenter nos exemples par une interprétation, c'est-à-dire par des ensembles de littéraux (exemple de l'emploi du

temps décrit avec un ensemble d'instances du prédicat *timetable*). Dans ce chapitre, nous préfèrerons la représentation par implication, plus souvent utilisée dans les systèmes d'ILP. Nos exemples sont donc décrits par un unique littéral, noté $c(Id)$ dans la suite, où Id est un identifiant permettant de retrouver les caractéristiques de l'exemple dans la base de connaissances (l'ensemble des instances de *timetable* lié à l'emploi du temps).

Pour illustrer, nous utiliserons l'exemple des trains déjà présenté dans le chapitre 3. Nous le rappelons avec la nouvelle représentation.

Exemple

Dans cet exemple chaque train est composé de voitures. Chaque voiture possède différentes caractéristiques telles que son nombre de roues et sa taille. Une voiture transporte des objets géométriques, décrits par leur forme et leur quantité. Le concept cible est goodtrain qui définit un « train » et nous avons deux trains identifiés par t_1^+ et t_2^-. Nous avons un exemple positif du concept cible $goodtrain(t_1^+)$ et un autre négatif $goodtrain(t_2^-)$. Pour décrire ces exemples, il faut ajouter dans la base de connaissances les littéraux clos suivants :

$$\begin{aligned}
\textit{Pour } t_1^+ \ : \{&has_car(t_1^+, c_1), has_car(t_1^+, c_2), has_car(t_1^+, c_3), wheels(c_1, 2),\\
&wheels(c_2, 3), wheels(c_3, 5), long(c_1), long(c_2), long(c_3),\\
&load(c_1, circle, 3), load(c_2, circle, 6), load(c_3, triangle, 10)\}
\end{aligned}$$

$$\begin{aligned}
\textit{Pour } t_2^- \ : \{&has_car(t_2^-, c_1), has_car(t_2^-, c_2), has_car(t_2^-, c_3), wheels(c_1, 1),\\
&wheels(c_2, 4), wheels(c_3, 3), long(c_1), long(c_2), short(c_3),\\
&load(c_1, circle, 4), load(c_1, rectangle, 2), load(c_2, triangle, 5)\\
&load(c_2, circle, 3), load(c_3, circle, 2)\}
\end{aligned}$$

On ajoute également à la base de connaissances le prédicat $\neq$ comparant les voitures ou les formes géométriques, et le prédicat $<$ comparant soit le nombre de roues, soit le nombre d'objets.

Le concept cible représente les trains ayant au moins deux voitures contenant des objets de la même forme et telles que l'une des deux ait un nombre plus grand d'objets et un nombre plus important de roues que l'autre. Cela peut être représenté en une seule règle :

$$\begin{aligned}
goodtrain(T) : -&has_car(T, C_1), has_car(T, C_2), C_1 \neq C_2,\\
&wheels(C_1, W_1), wheels(C_2, W_2),\\
&load(C_1, O, L_1), load(C_2, O, L_2),\\
&W_1 < W_2, L_1 < L_2
\end{aligned}$$

La principale différence entre les deux représentations d'exemples réside dans la nécessité, quand on utilise celle par implication, de reconstruire l'exemple en retrouvant les littéraux le caractérisant. À noter que cette opération est également nécessaire dans le cas des interprétations pour trouver notamment les instances des prédicats donnés en intention, dans l'exemple des trains $\neq$ et $<$ qui sont satisfaits avec notre exemple. Cette opération, consistant à trouver les littéraux satisfaisant l'exemple, souvent utilisée en ILP, est appelée la *saturation*[Rou92, Rou90].

Par exemple, pour le train t_1^+, la saturation ajouterait d'une part, les littéraux clos le décrivant comme $has\_car(t_1^+, c_1)$ ou $wheels(c_3, 5)$, et d'autre part, les littéraux $3 < 4$ ou encore $rectangle \neq triangle$ construits à partir des prédicats donnés dans la bases de connaissances.

Dans la suite de cette section, nous formalisons l'opération de saturation et nous montrons comment elle peut être utilisée pour réduire le langage des hypothèses.

Nous commençons par introduire formellement la notation utilisée pour les modes des prédicats.

Définition 5.2.1 (Mode d'un prédicat)**.** *Un mode d'un prédicat sépare ses arguments en deux catégories : des entrées, notées +, et des sorties notées -. Étant donnés un littéral l et un mode m pour le prédicat de l, nous introduisons les notations suivantes :*

- *$input(l, m)$, l'ensemble des termes en entrées de l selon le mode m*
- *$output(l, m)$, l'ensemble des termes en sorties de l selon le mode m*

Le mode indique aussi les types de constantes ou variables attendues comme arguments du prédicat. Étant donné le prédicat p d'arité k, nous notons un mode de p, $p(m_1, m_2, \ldots, m_k)$ où m_i est soit $+type$ ou $-type$ avec $type$ le type de l'argument.

Dans la suite, nous considérons dans la formalisation qu'il n'existe qu'un seul mode par prédicat et par conséquent allégeons la notation en retirant le paramètre m des définitions de *input* et *output*. Cependant, avoir plusieurs modes par prédicat ne pose pas de problème. Une manière simple pour s'en rendre compte est de dupliquer les prédicats ayants plusieurs modes. Imaginons avoir dans la base de connaissance le prédicat p avec les modes $p(+, +, -)$ et $p(+, -, -)$, il suffirait alors de créer le prédicat p_1 avec le mode $p_1(+, +, -)$ et p_2 avec le mode $p_2(+, -, -)$.

Exemple (suite)

Dans l'exemple des trains, les types de constantes et variables sont :

- *$train$, par exemple les constantes t_1^+ ;*
- *car, par exemple c_1 et c_2 ;*
- *$nbwheels$, par exemple 3 et 5 ;*
- *$shape$, par exemple $circle$;*
- *$nbobjects$, par exemple 2 et 4.*

Nous avons choisi les modes suivants :

- *$has\_car(+train, -car)$;*
- *$wheels(+car, -nbwheels)$;*
- *$long(+car)$;*
- *$load(+car, -shape, -nbobjects)$;*
- *$\neq (+car, +car)$;*
- *$\neq (+shape, +shape)$;*
- *$< (+nbwheels, +nbwheels)$;*
- *$< (+nbobjects, +nbobjects)$;*

Ainsi $input(load(c_1, circle, 3))$ est égal à $\{c_1\}$ et $output(load(c_1, circle, 3))$ à $\{circle, 3\}$.

Définition 5.2.2 (Entrées/Sorties d'un ensemble de littéraux)**.** *Étant donné un ensemble de littéraux E, nous définissons ses entrées, respectivement ses sorties, par $input(E) = \cup_{l \in E} input(l)$, respectivement $output(E) = \cup_{l \in E} output(l)$.*

Nous définissons la saturation de manière simplifiée, ne considérant pas les nombres de rappel (*recall number* en anglais) permettant de limiter le nombre de fois qu'un prédicat peut être utilisé. Nous ne considérons pas les symboles de fonctions dans notre langage permettant de construire des termes plus complexes. La saturation d'un exemple est obtenue en ajoutant, autant que possible, des littéraux en respect de la base de connaissances tel que l'ensemble forme une formule connectée. La saturation est basée sur les informations liées aux modes des prédicats. C'est un ensemble organisé en plusieurs couches : un littéral est dans la couche k si toutes les entrées nécessaires à son introduction ont été ajoutées dans les couches précédentes. Cette opération pouvant être infinie, elle est souvent paramétrée par une profondeur maximale correspondant à la dernière couche. Cette profondeur maximale est noté i.

Définition 5.2.3 (Saturation). *Étant donné un exemple $c(e)$ et une base de connaissance BK, la saturation $sat(i, c(e))$ de $c(e)$ à la profondeur i est définie par :*

$$sat(i, c(e)) = \quad sat(i, \{c(e)\})$$

$$\begin{aligned} & sat(0, S) = && S \\ k > 0 \quad & sat(k, S) = && sat(k-1, S \cup \{l \in BK \mid input(l) \subseteq terms(S)\}) \end{aligned}$$

où S est un ensemble de littéraux et $terms(S)$ l'ensemble des termes apparaissant dans S.

Le paramètre k de $sat(k, S)$ ne représente pas le calcul de la k^e couche de la clause mais le nombre de couches restant à calculer, S étant les couches déjà calculé.

La dernière étape de la saturation consiste à remplacer chaque constante par une variable. L'exemple « *variabilisé* » est la tête de la règle et les autres littéraux de la saturation forment le corps.

Nous illustrons la saturation sur l'exemple du train.

Exemple (suite)

La saturation de l'exemple $goodtrain(t_1^+)$ avec comme profondeur maximale 3 est :

Layer	*Literals*
0	$goodtrain(t_1^+)$
1	$has\_car(t_1^+, c_1), has\_car(t_1^+, c_2), has\_car(t_1^+, c_3)$
2	$wheels(c_1, 2), wheels(c_2, 3), wheels(c_3, 4), long(c_1), long(c_2), long(c_3),$ $load(c_1, circle, 3), load(c_2, circle, 5), load(c_3, triangle, 10),$ $c_1 \neq c_2, c_1 \neq c_3, c_2 \neq c_3$
3	$2 < 3, 2 < 4, 3 < 4, triangle \neq circle, 3 < 5, 3 < 10, 5 < 10$

La saturation est généralement utilisée pour limiter la taille de l'espace de recherche. En prenant un exemple positif $c(e)$ appelé une graine, la saturation représente une règle trop spécifique car elle ne couvrira sans doute que l'exemple $c(e)$ mais correcte à condition que la saturation soit suffisamment profonde et que la base de connaissances contienne suffisamment d'information pour discriminer $c(e)$ des exemples négatifs. Ainsi, la saturation peut être utilisée comme borne $\perp$ (voir section 3.3.1) du langage des hypothèses pour en limiter la taille. Ce sera notre cas comme décrit dans la section suivante.

5.3 Langage des hypothèses

Dans notre langage, nous nous intéressons aux clauses de Horn de la forme $h \leftarrow B$, où h est la *tête* de la clause et B une conjonction de littéraux formant le *corps*. La tête représente le concept cible et donc sera la même pour toutes les hypothèses, $goodtrain(Id)$ dans l'exemple des trains. Dans la suite, nous considérons parfois ces clauses comme des ensembles de littéraux. Ainsi, nous considérerons la clause $h \leftarrow b_1, b_2, \ldots, b_n$ comme l'ensemble de littéraux $\{h, b_1, b_2, \ldots, b_n\}$ permettant d'utiliser les notations d'inclusion ($\subseteq$) ou d'appartenance ($\in$).

Nous ne souhaitons que des règles respectant les modes, et donc l'introduction correcte des variables (comme sortie des littéraux) comme défini ci-dessous.

Définition 5.3.1 (Clause respectant un ensemble de modes)**.** *Nous dirons qu'une clause C respecte un ensemble de modes M, s'il existe un ordonnancement $S = b_1, b_2, \ldots, b_n$ des littéraux du corps de C tel que :*

$$\forall b_i \in S, input(b_i) \subseteq \big(output(\cup_{1 \leq j < i} b_j) \cup terms(h)\big)$$

où h est la tête de C.

Comme dit dans la section précédente, nous utilisons la saturation pour limiter la taille de notre langage d'hypothèses.

Définition 5.3.2 ($\perp$)**.** *Étant donné une profondeur maximale i et un exemple positif $c(e)$ appelé une graine, la clause $\perp$, la plus spécifique de notre langage d'hypothèses, est défini par : $vars(sat(c(e), i))$ où $vars$ est l'opération consistant à substituer chaque constante par une variable unique.*

Exemple (suite)

Ainsi dans l'exemple des trains, il suffit à partir de la saturation calculée précédemment de remplacer chaque constante par une unique variable.

Layer	*Literals*
0	$goodtrain(Vt_1^+)$
1	$has\_car(Vt_1^+, Vc_1), has\_car(Vt_1^+, Vc_2), has\_car(Vt_1^+, Vc_3)$
2	$wheels(Vc_1, VW2), wheels(Vc_2, VW3), wheels(Vc_3, VW4),$ $long(Vc_1), long(Vc_2), long(Vc_3),$ $load(Vc_1, Vcircle, VL3), load(Vc_2, Vcircle, VL5),$ $load(Vc_3, Vtriangle, VL10),$ $Vc_1 \neq Vc_2, Vc_1 \neq Vc_3, Vc_2 \neq Vc_3$
3	$VW2 < VW3, VW2 < VW4, VW3 < VW4,$ $Vtriangle \neq Vcircle, VL3 < VL5, VL3 < VL10, VL5 < VL10$

Ici, on a par exemple remplacé la constante c_1 par la variable Vc_1.

Nous pouvons maintenant définir notre langage d'hypothèses.

Définition 5.3.3 ($\mathcal{L}_\perp$)**.** *Étant donnés $\perp$ défini dans la définition 5.3.2 et un ensemble de modes M, une clause C est dans $\mathcal{L}_\perp$ si et seulement si C respecte les modes de M et les littéraux de C sont dans $\perp$.*

À noter que nous ne considérons pas les renommages de variables. Ainsi dans l'exemple du train, la règle

$$goodtrain(Vt_1^+) \leftarrow has\_car(Vt_1^+, Vc_1)$$

et

$$goodtrain(Vt_1^+) \leftarrow has\_car(Vt_1^+, Vc_2)$$

sont deux hypothèses différentes. Cette manière de considérer les hypothèses se retrouve dans la propositionalisation [AR00]. On peut remarquer que par définition $\mathcal{L}_\perp$ est un sous-ensemble de $2^\perp$ représentant l'ensemble des parties de $\perp$.

Nous allons maintenant étudier les manières de structurer un tel langage. Premièrement, nous verrons que l'on peut organiser n'importe quelle clause de ce langage sous forme d'une structure en couches comme illustré dans les exemples de saturation. Ceci nous servira dans les futures sections à définir notre opérateur de raffinement. Ensuite nous proposerons un ordre de généralité dans ce langage et enfin montrerons qu'avec cet ordre, le langage forme un treillis.

Nous commençons donc par montrer que l'on peut organiser une clause de $\mathcal{L}_\perp$ en plusieurs couches. Pour cela, nous allons définir la notion de profondeur de variables et de profondeur de littéraux. Ces deux notions étant mutuellement récursives, nous les définissons en même temps. Afin de simplifier la définition, il faut remarquer que nous considérons que les arguments du prédicat servant à donner les exemples (*goodtrain* pour l'exemple des trains) sont des sorties (alors qu'avec la sémantique donnée aux modes il serait plutôt des entrées).

Définition 5.3.4 (Profondeur d'une variable et d'un littéral)**.** *Étant donnés un littéral l et une variable X, tous deux présents dans une clause $C \in \mathcal{L}_\perp$, nous définissons la profondeur $depth(X, C)$ de X et la profondeur $depth(l, C)$ de l par :*

$$\begin{array}{ll} depth(X, C) = 0 & \textit{Si } X \textit{ est dans la tête de } C \\ depth(X, C) = min_{\ l \in C \ \mid \ X \in output(l)} depth(l, C) & \textit{Sinon} \end{array}$$

$$\begin{array}{ll} depth(l, C) = 0 & \textit{Si } l \textit{ est la tête de } C \\ depth(l, C) = max_{X \in input(l)} depth(X, C) + 1 & \textit{Sinon} \end{array}$$

Proposition 5.3.1. *Pour toute clause C de $\mathcal{L}_\perp$, les profondeurs des variables et des littéraux sont définies.*

Démonstration. Nous montrons que pour toute clause $C \in \mathcal{L}_\perp$, pour tout littéral $l \in C$ et pour chaque variable $X \in C$, $depth(l, C)$ et $depth(X, C)$ sont définies.

Comme $C \in \mathcal{L}_\perp$, il existe une séquence $S = b_1, b_2, \ldots, b_n$ des littéraux du corps de C telle que définie dans la définition 5.3.1. Nous définissons la clause C_k comme la clause contenant une sous-séquence S_k de S commençant par b_1 et terminant par b_k. Nous définissons également C_0 comme la clause $h \leftarrow$ prenant une sous-séquence vide de S. Nous avons donc $C_n = C$ et par construction n'importe qu'elle C_k appartient à $\mathcal{L}_\perp$.

Nous montrons par induction que pour chaque valeur de k, les profondeurs $depth(X, C_k)$ et $depth(l, C_k)$ sont définies.

Pour $k = 0$, alors $C = h \leftarrow$ et par définition $depth(h, C_0) = 0$ et $depth(X, C_0) = 0$, pour toute variable X de h.

Si *depth* est bien définie pour C_k, nous montrons qu'elle l'est aussi pour C_{k+1}.

On a, par hypothèse, $S_k = b_1, b_2, \ldots, b_k$ où $depth(l, C_k)$ est définie pour tout b_i et $depth(X, C_k)$ est définie pour toute variable X apparaissant dans b_i avec $1 \leq i \leq k$. Pour i compris entre 1 et k, $depth(b_i, C_{k+1})$ est définie soit avec la même valeur que $depth(b_i, C_k)$ soit avec une valeur plus petite si l'ajout de b_{k+1} le permet.

Pour b_{k+1}, comme $C_{k+1} \in \mathcal{L}_\perp$, on a la propriété : $input(b_{k+1}) \subseteq \big(output(\cup_{1 \leq j \leq k} b_j) \cup terms(h)\big)$ avec $1 \leq i \leq k$. Donc, $depth(l, C_{k+1})$ est définie pour tout l apparaissant dans C_{k+1}.

Ensuite comme $depth(l, C_{k+1})$ est définie, on peut également définir $depth(X, C_{k+1})$ pour tout $X \in C_{k+1}$.

□

Proposition 5.3.2. *Pour toute clause C de $\mathcal{L}_\perp$, un littéral ou une variable a une profondeur unique.*

Démonstration. De la même manière que précédemment, $C \in \mathcal{L}_\perp$ implique qu'il existe une séquence $S = b_1, b_2, \ldots, b_k$ des littéraux du corps de C telle que la définition 5.3.1 soit vérifiée. Supposons qu'il existe $depth_1$ et $depth_2$, tel que $\exists\, l \in C$, $depth_1(l) \neq depth_2(l)$. Nous montrerons que ces deux définitions de *depth* ne peuvent exister. On procède également par induction sur k.

Si $k = 0$, alors $C = h \leftarrow$ et $depth_1(h) = depth_2(h) = 0$.

On a par définition :

$$depth_1(l_{k+1}) = max_{X \in input(l_{k+1})} depth_1(X) + 1$$

$$depth_2(l_{k+1}) = max_{X \in input(l_{k+1})} depth_2(X) + 1$$

Comme $C \in \mathcal{L}_\perp$, $input(l_{k+1}) \subseteq \big(output(\cup_{1 \leq i \leq k} l_i) \cup terms(h)\big)$ et donc $depth_1(X) = depth_2(X)$, et donc $\forall X \in input(l_{k+1})$, on a :

$$max_{X \in input(l_{k+1})} depth_1(X) = max_{X \in input(l_{k+1})} depth_2(X)$$

Par conséquent, $depth_1(l_{k+1}) = depth_2(l_{k+1})$ et $\forall X \in output(l_{k+1}), depth_1(X) = depth_2(X) = min_{X \in output(l)} depth_{1/2}(l)$

□

Définition 5.3.5 (Profondeur maximale d'une clause)**.** *Étant donnée une clause $C \in \mathcal{L}_\perp$, nous définissons la profondeur maximale de C comme :*

$$maxdepth(C) = max_{l \in \mathcal{L}_\perp} depth(l, C)$$

Nous nous intéressons maintenant à la manière d'ordonner les clauses de $\mathcal{L}_\perp$ les unes par rapport aux autres. Pour cela, nous choisissons un ordre de généralité.

Définition 5.3.6 (Ordre de généralité)**.** *Étant données deux clauses C et D de $\mathcal{L}_\perp$, nous dirons que C est plus générale que D si les littéraux de C sont également présents dans D. Cet ordre revient donc aux sous-ensembles et nous noterons par conséquent $C \subseteq D$ le fait que C soit plus générale que D.*

Dans $\mathcal{L}_\perp$, les propriétés de réflexivité, transitivité et antisymétrie de $\subseteq$ sont préservées et donc il s'agit d'un ordre. Par conséquent, $\langle \mathcal{L}_\perp, \subseteq \rangle$ est un ensemble ordonné. Nous allons maintenant montrer qu'il s'agit d'un treillis. Pour cela, nous montrons qu'il existe pour chaque couple de clauses de $\mathcal{L}_\perp$ un plus grand généralisé et un plus petit spécialisé.

Définition 5.3.7 (*mgs* dans $\langle \mathcal{L}_\perp, \subseteq \rangle$)**.** *Nous définissons le plus grand spécialisé de deux clauses C et D de $\mathcal{L}_\perp$, noté $mgs(C, D)$, par :*

$$mgs(C, D) = C \cup D$$

Lemme 5.3.1. *Pour tout C et D de $\mathcal{L}_\perp$, $mgs(C, D)$ existe et appartient à $\mathcal{L}_\perp$.*

Démonstration. Comme C et D appartienent à $\mathcal{L}_\perp$, il existe respectivement les séquences ordonnant les littéraux de leur corps :

$$S_C = b_{1,C}, b_{2,C}, \ldots, b_{n,C}$$

$$S_D = b_{1,D}, b_{2,D}, \ldots, b_{m,D}$$

vérifiant la définition 5.3.1. Par conséquent, la séquence $S_C, (S_D \backslash S_C)$ montre qu'il existe une séquence respectant les modes et par conséquent que $mgs(C, D) = C \cup D$ est bien défini et appartient à $\mathcal{L}_\perp$. □

Alors que définir un opérateur pour le plus grand spécialisé est plutôt simple, le plus petit généralisé est plus complexe. En effet, une simple intersection entre les deux clauses ne suffit pas car il faut vérifier que les littéraux respectent bien les modes. Nous proposons une définition inductive pour cet opérateur.

Définition 5.3.8 (*lgg* dans $\langle \mathcal{L}_\perp, \subseteq \rangle$)**.** *Nous commençons par définir les opérateurs lgg^0, lgg^k, lgg^* pour deux clauses C et D de $\mathcal{L}_\perp$ de la manière suivante :*

$$lgg^0(C, D) = h \text{ où } h \text{ est la tête de } C \text{ et } D$$

$$lgg^k(C, D) = \{l \mid \ l \in C \cap D \ \wedge \ input(l) \subseteq \big(output(\bigcup_{i=0}^{i=k-1} lgg^i(C, D)) \cup terms(h)\big)\}$$

Ces opérateurs permettent de construire couche par couche le plus grand généralisé. Ainsi $lgg^k(C, D)$ représente tous les littéraux pouvant être ajoutés et respectant les modes si on applique l'opérateur k fois.

Finalement, nous définirons le plus petit généralisé de deux clauses C et D de $\mathcal{L}_\perp$, noté $lgg(C, D)$, par :

$$lgg(C, D) = lgg^m(C, D) \quad tel\ que \quad m = min(maxdepth(C), maxdepth(D))$$

Lemme 5.3.2. *Pour tout C et D de $\mathcal{L}_\perp$, $lgg(C, D)$ existe et appartient à $\mathcal{L}_\perp$.*

Démonstration. Nous montrons d'abord que $lgg^k(C, D)$, forme une clause appartenant à $\mathcal{L}_\perp$ pour tout $k \geq 0$.

Commençons par rappeler que comme C et D appartient à $\mathcal{L}_\perp$, il existe respectivement les séquences comme dans les preuves précédentes :

$$S_C = b_{1,C}, b_{2,C}, \ldots, b_{n,C}$$

$$S_D = b_{1,D}, b_{2,D}, \ldots, b_{m,D}$$

Pour $k = 0$, $lgg^0(C, D) = \{h\}$ et donc appartient à $\mathcal{L}_\perp$.

Supposons qu'il existe un k tel que $lgg^k(C, D) \in \mathcal{L}_\perp$. On montre que $lgg^{k+1}(C, D) \in \mathcal{L}_\perp$. Premièrement si b appartient à $lgg^{k+1}(C, D)$ et à $lgg^k(C, D)$, alors b respecte bien les modes. Si b_p appartient à $lgg^{k+1}(C, D) \backslash lgg^k(C, D)$, par définition on peut construire la séquence $S = b_1, b_2, \ldots b_m, b_p, b_{p+1} \ldots b_n$ tel que b_i avec $1 \leq i \leq m$ constitue les littéraux de $lgg^k(C, D)$ et tous les littéraux après b_p sont ceux de $lgg^{k+1}(C, D) \backslash lgg^k(C, D)$. Comme par définition du lgg, pour tout b_j, $p \leq j \leq n$, on a $input(b_p) \subseteq \big(output(lgg^k(C, D)) \cup terms(h)\big)$, cette séquence vérifie bien la définition 5.3.1 et $lgg^{k+1}(C, D)$ appartient à $\mathcal{L}_\perp$.

Par conséquent, $lgg(C, D) = lgg^*(C, D)$ appartient bien à $\mathcal{L}_\perp$ pour tout C et D de $\mathcal{L}_\perp$. □

Le théorème suivant découle directement des deux précédents lemmes.

Théorème 5.3.3. $\langle \mathcal{L}_\perp, \subseteq \rangle$ *est un treillis.*

Nous finissons cette section avec la proposition suivante qui permet d'exhiber un cas où $\langle \mathcal{L}_\perp, \subseteq \rangle$ est fini.

Proposition 5.3.3. *Si $\perp$ est fini, alors $\langle \mathcal{L}_\perp, \subseteq \rangle$ est fini.*

Démonstration. Cela découle directement de la définition 5.3.3. En effet puisque $\langle \mathcal{L}_\perp, \subseteq \rangle$ est un sous-ensemble de $\langle 2^\perp, \subseteq \rangle$, si ce dernier est fini, alors notre langage également. Nous rappelons que nous ne considérons pas les symboles de fonctions dans notre langage. □

5.4 Espace de recherche

Dans la prochaine section, nous définissons notre opérateur de raffinement bi-directionnel. Cet opérateur a la particularité de produire des raffinements non pas uniquement à partir d'une hypothèse comme il est régulièrement le cas dans la littérature mais à partir d'un couple d'hypothèses ayant les particularités :

- l'une est plus générale que l'autre ;
- les littéraux qu'ils ont en commun ont la même profondeur ;
- la clause la plus spécifique contient seulement les mêmes couches que la plus générale (pas de littéraux supplémentaires dans ces couches) et des couches plus profondes (sauf dans le cas particulier où ces clauses sont égales).

Dans cette section, après avoir défini formellement notre espace de recherche, nous nous intéressons à la possibilité d'y retrouver toutes les hypothèses de $\mathcal{L}_\perp$.

Définition 5.4.1 ($\mathcal{L}_{\perp,2}$)**.** *Soit $\mathcal{L}_{\perp,2}$, le langage de couples d'hypothèses défini par :*
$(C, D) \in \mathcal{L}_{\perp,2}$ *si et seulement si :*

- $C,\ D \in \mathcal{L}_\perp$
- *il existe* md_C *tel que* $0 \leq md_C \leq maxdepth(C)$, $l \in C \Leftrightarrow l \in D \wedge depth(l, D) \leq md_C$

Afin de montrer que ce langage est bien défini nous montrons que pour tout $(C, D) \in \mathcal{L}_{\perp,2}$ les propriétés suivantes sont correctes :

1. $C \subseteq D$ (C est plus générale que D) ;
2. Pour tout littéral l de C, $depth(l, C) = depth(l, D)$ et pour toute variable X de C, $depth(X, C) = depth(X, D)$;
3. $maxdepth(C) = md_C$

Démonstration de 1. Ceci est directement impliqué par $l \in C \Leftrightarrow l \in D \wedge depth(l, D) \leq md_C$ □

Démonstration de 2. Nous le montrons par récurrence sur $depth(l, C)$.

Si $depth(l, C) = 0$, l est donc la tête de C qui est la même que D et donc $depth(l, D) = 0$. On a également que pour toute variable X de l, $depth(X, C) = depth(X, D) = 0$.

Supposons qu'il existe k tel que pour tout $j \leq k$, si $depth(l, C) = j$ alors $depth(l, C) = depth(l, D)$ et si $depth(X, D) = j$ alors $depth(X, C) = depth(X, D)$. Nous montrons que cela est également le cas si $depth(l, C) = k + 1$. On a par définition $depth(l, C) = max_{X \in input(l)} depth(X, C) + 1$ et donc que pour toute variable X dans $input(l)$, $depth(X, C) \leq k$ puisque C respecte les modes. Grâce à l'hypothèse de récurrence, on sait que $\forall X \in input(l)$, $depth(X, C) = depth(X, D)$ et donc $depth(l, C) = depth(l, D)$. De même, on a $depth(X, C) = depth(X, D)$. □

Démonstration de 3. Supposons que $maxdepth(C) \neq md_C$. Alors par définition on a $md_C < maxdepth(C)$. Or $maxdepth(C) = max_{l \in C} depth(l, C)$ et donc il existe $l \in C$ tel que $depth(l, C) > md_C$. Par la seconde propriété, $depth(l, C) = depth(l, D)$ et donc il existe $l \in C$ tel que $depth(l, D) > md_C$ ce qui forme une contradiction avec la définition. □

Les états finaux de notre espace de recherche seront les états du type (C, C), représentant l'unique hypothèse C, et respectant les contraintes de complétude et de correction. L'existence de tels états dépend bien entendu de l'existence d'une clause C, solution du problème d'apprentissage, dans $\mathcal{L}_\perp$. D'où le théorème suivant qui permet d'être sûr que pour toute hypothèse de $\mathcal{L}_\perp$, il y a un état correspondant à une solution dans l'espace de recherche.

Théorème 5.4.1. *Pour toute clause C de $\mathcal{L}_\perp$, il existe $(C, C) \in \mathcal{L}_{\perp,2}$.*

Démonstration. La preuve est triviale puisqu'en remplaçant D par C, la définition 5.4.1 est correcte. Ainsi, $C \in \mathcal{L}_\perp$ et $l \in C \Leftrightarrow l \in C \ \wedge \ depth(l, C) \leq md_C$ est vrai. □

Nous définissons maintenant un ordre partiel sur $\mathcal{L}_{\perp,2}$ afin de pouvoir comparer les couples entre eux.

Définition 5.4.2 (Ordre sur $\mathcal{L}_{\perp,2}$)**.** *On définit l'ordre $\succeq$ sur $\mathcal{L}_{\perp,2}$ de la manière suivante :*

$$\forall (C, D), (C', D') \in \mathcal{L}_{\perp,2}, (C, D) \succeq (C', D') \Leftrightarrow C \subseteq C' \wedge D' \subseteq D$$

Avec cet ordre, l'élément maximal de l'espace de recherche est $(\top, \perp)$ et les élément minimaux sont les couples de la forme (C, C) où $C \in \mathcal{L}_\perp$.

Cet espace de recherche est intéressant car il offre des états de recherche plus riche en informations avec notamment une clause plus spécifique que l'autre pouvant guider plus précisément une recherche descendante quand des phénomènes de plateaux se présentent. Mais il souffre d'une grande faiblesse : sa taille. En effet, pour un nombre d'hypothèses déjà assez grand on crée un espace de recherche en ajoutant de nombreux états qui ne font pas partie des solutions potentielles (tout les états (C, D) où $C \neq D$). Par conséquent, les approches complètes pouvant être quasi-exhaustives sont, en pratique, inutilisables. Cependant, l'utilisation de technique d'élagage, comme c'est le cas dans la résolution des CSP, pourraient potentiellement réduire la taille de l'espace de recherche et ainsi rendre praticable les approches complètes.

5.5 Opérateur de raffinement

L'opérateur de raffinement bidirectionnel $\rho(H_\top, H_\perp)$ où $(H_\top, H_\perp) \in \mathcal{L}_{\perp,2}$ se décompose en deux étapes. Dans un premier temps, il ajoute à $H_\top$ une nouvelle couche de littéraux présent dans $H_\perp$ de profondeur $maxdepth(H_\top) + 1$. Puis, supprime de $H_\perp$, les littéraux de profondeur $maxdepth(H_\top) + 1$ qui n'ont pas été ajoutés à la nouvelle couche de $H_\top$, et enfin retire les littéraux ne respectant pas les modes jusqu'à obtenir une hypothèse de $\mathcal{L}_\perp$.

Pour définir cet opérateur, il nous faut deux opérations : une fournissant les littéraux candidats à être ajoutés à $H_\top$ afin de la spécialiser, l'autre pour généraliser $H_\perp$ en supprimant certains littéraux. Ces deux opérations dépendent de deux hypothèses de $\mathcal{L}_\perp$ dont le couple n'est pas nécessairement présent dans $\mathcal{L}_{\perp,2}$. La première, nommée *next*, consiste à récupérer les littéraux de $H_\perp$ de la couche $maxdepth(H_\top)+1$ tel que si ces littéraux étaient

ajoutées à $H_\top$, ils respecteraient les modes. La seconde, nommée *narrow*, retirera les littéraux qui ne respectent plus les modes dans $H_\bot$ si on réduit la couche $maxdepth(H_\top)+1$ de $H_\bot$ à un certain ensemble de littéraux. Ainsi l'opérateur de raffinement produira des candidats en ajoutant une nouvelle couche de littéraux issus de $H_\bot$ à $H_\top$ et réduira $H_\bot$ en supprimant les littéraux qui ne pourront plus être intéressants dans de futurs raffinements.

Nous commençons cette section en définissant formellement ces opérations ainsi que l'opérateur de raffinement bidirectionnel que nous avons proposé dans [LMV10]. Enfin, nous nous intéressons aux différentes propriétés que cet opérateur.

Définition 5.5.1 (*next*)**.** *Étant données les clauses $C, D \in \mathcal{L}_\bot$, $next(k, C, D)$ est l'ensemble des littéraux présents dans la couche k de D qui pourraient être ajoutés à C en respectant les modes. Plus formellement :*

$$\begin{aligned} next(k, C, D) = \{l \mid \quad & l \in D \\ \wedge \quad & depth(l, D) = k \\ \wedge \quad & input(l) \subseteq output(C)\} \end{aligned}$$

Exemple (suite)

Dans l'exemple des trains, le résultat de $next(2, C, D)$, où C et D sont définis par :
C :

Layer	*Literals*
0	$goodtrain(Vt_1^+)$
1	$has\_car(Vt_1^+, Vc_1), has\_car(Vt_1^+, Vc_2)$

et
D :

Layer	*Literals*
0	$goodtrain(Vt_1^+)$
1	$has\_car(Vt_1^+, Vc_1), has\_car(Vt_1^+, Vc_2)$
2	$wheels(Vc_1, VW2), wheels(Vc_2, VW3),$ $long(Vc_1), long(Vc_2),$ $load(Vc_1, Vcircle, VL3), load(Vc_2, Vcircle, VL5),$ $Vc_1 \neq Vc_2,$
3	$VW2 < VW3, VL3 < VL5$

sera l'ensemble des littéraux de la couche 2 de D.

Définition 5.5.2 (*narrow*)**.** *Étant données les clauses $C, D \in \mathcal{L}_\bot$, $narrow(k, C, D)$ retourne une clause E de $\mathcal{L}_\bot$ telle que E contient les littéraux de C plus les littéraux des couches supérieures ou égale à k de D qui respectent les modes. Plus formellement :*

$$narrow(k, C, D) = narrow(k, maxdepth(D), C, D)$$

où $narrow(k, n, C, D)$ est définit par :

$$\begin{aligned} & narrow(n, n, C, D) = \quad C \\ \forall n > k \quad & narrow(k, n, C, D) = \quad narrow(k, n-1, C, D) \\ & \qquad\qquad\qquad\quad \cup\, next(n, narrow(k, n-1, C, D), D) \end{aligned}$$

Exemple (suite)

Le résultat de $narrow(2, C, D)$*, où* C *et* D *sont définis par :*

C :

Layer	*Literals*
0	$goodtrain(Vt_1^+)$
1	$has\_car(Vt_1^+, Vc_1)$

et

D :

Layer	*Literals*
0	$goodtrain(Vt_1^+)$
1	$has\_car(Vt_1^+, Vc_1), has\_car(Vt_1^+, Vc_2)$
2	$wheels(Vc_1, VW2), wheels(Vc_2, VW3),$ $long(Vc_1), long(Vc_2),$ $load(Vc_1, Vcircle, VL3), load(Vc_2, Vcircle, VL5),$ $Vc_1 \neq Vc_2,$
3	$VW2 < VW3, VL3 < VL5$

sera :

Layer	*Literals*
0	$goodtrain(Vt_1^+)$
1	$has\_car(Vt_1^+, Vc_1)$
2	$wheels(Vc_1, VW2), long(Vc_1),$ $load(Vc_1, Vcircle, VL3),$

Notre opérateur de raffinement fournit pour un couple $(H_\top, H_\bot)$ de $\mathcal{L}_{\bot,2}$ un ensemble de raffinements possibles tels que $H_\top$ soit spécialisé et $H_\bot$ soit généralisé. L'idée principale est de pouvoir ainsi à chaque étape « rapprocher » les deux hypothèses pour au final obtenir la même. Pour cela, $\rho(H_\top, H_\bot)$ va proposer des raffinements $(H'_\top, H'_\bot)$ tels que $H'_\top$ a une couche de plus que $H_\top$. $H_\bot$ est quant à lui généralisé de manière à éliminer les littéraux qui ne sont plus intéressants car leur ajout dans $H'_\top$ ne respecterait pas les modes.

Définition 5.5.3 (ρ)**.** *Pour tout couple* $(H_\top, H_\bot) \in \mathcal{L}_{\bot,2}$*, l'opérateur de raffinement bidirectionnel* ρ *de [LMV10] est défini de la manière suivante :*

Si $H_\top = H_\bot$ *:*

$$\rho(H_\top, H_\bot) = \emptyset$$

Si $H_\top \neq H_\bot$ *:*

$$\begin{aligned} \rho(H_\top, H_\bot) = \{(H'_\top, H'_\bot) \mid \quad & H'_\top = H_\top \cup S \\ \wedge \quad & S \subseteq next(maxdepth(H_\top) + 1, H_\top, H_\bot) \\ \wedge \quad & H'_\bot = narrow(maxdepth(H_\top) + 1, H'_\top, H_\bot)\} \end{aligned}$$

En pratique, nous retrouvons la limite décrite dans la section précédente concernant la taille trop importante de l'espace de recherche. Effectivement, il existe pour un couple un nombre de raffinements très grand, exponentiel en fonction de la taille de la couche

de $H_\perp$ dans lequel sont pris les littéraux ajoutés à $H_\top$. Cependant, nous ne produisons pas tous les raffinements possibles en pratique. Motivé par le principe du rasoir d'Occam, consistant à choisir les hypothèses les plus simples possibles, nous commençons par produire les raffinements ajoutant un unique littéral à $H_\top$, puis 2, puis 3... et ainsi de suite jusqu'à trouver un raffinement satisfaisant. Nous décrirons plus particulièrement notre stratégie dans la section suivante.

Exemple (suite)

Pour illustrer cet opérateur de raffinement, nous considérons le couple d'hypothèses : $(H_\top, H_\perp) = (vars(\{goodtrain(t_1^+)\}), vars(sat(goodtrain(t_1^+), 3))$ *et les raffinements produits par* $\rho(H_\top, H_\perp)$.

En ajoutant une couche de taille 1 à $H_\top$*, tous les raffinements sont équivalents à un renommage de variables près, mais seront tous considérés par l'opérateur :*

Layer	*Literals*
0	$\mathbf{goodtrain(Vt_1^+)}$
1	$\mathbf{has\_car(Vt_1^+, Vc_1)}$
2	$wheels(Vc_1, VW2), load(Vc_1, Vcircle, VL3), long(Vc_1)$
3	

où $H'_\top$ *est en gras et* $H'_\perp$ *contient tous les littéraux. Ce couple n'est pas consistant car* $H'_\perp$ *couvre l'exemple négatif* t_2^-*. Par conséquent, aucune solution ne pourra être trouvé à partir de ce couple.*

Avec une couche de taille 2, nous obtenons les couples :

Layer	*Literals*
0	$\mathbf{goodtrain(Vt_1^+)}$
1	$\mathbf{has\_car(Vt_1^+, Vc_1)}, \mathbf{has\_car(Vt_1^+, Vc_2)}$
2	$wheels(Vc_1, VW2), wheels(Vc_2, VW3), load(Vc_1, Vcircle, VL3),$ $load(Vc_2, Vcircle, VL5), long(Vc_1), long(Vc_2), Vc_1 \neq Vc_2$
3	$VW2 < VW3, VL3 < VL5$

Layer	*Literals*
0	$goodtrain(Vt_1^+)$
1	$has\_car(Vt_1^+, Vc_2), has\_car(Vt_1^+, Vc_3)$
2	$wheels(Vc_2, VW3), wheels(Vc_3, VW4), long(Vc_2), long(Vc_3),$ $load(Vc_2, Vcircle, VL5), load(Vc_3, Vtriangle, VL10), Vc_2 \neq Vc_3$
3	$VW3 < VW4, Vtriangle \neq Vcircle, VL5 < VL10$

Layer	*Literals*
0	$\mathbf{goodtrain(Vt_1^+)}$
1	$\mathbf{has\_car(Vt_1^+, Vc_1)}, \mathbf{has\_car(Vt_1^+, Vc_3)}$
2	$wheels(Vc_1, VW2), wheels(Vc_3, VW4), long(Vc_1), long(Vc_3),$ $load(Vc_1, Vcircle, VL3), load(Vc_3, Vtriangle, VL10), Vc_1 \neq Vc_3$
3	$VW2 < VW4, Vtriangle \neq Vcircle, VL3 < VL10$

Seule la première clause $H'_\perp$ *rejette les exemples négatifs.*

Nous allons maintenant nous assurer que les raffinements produits font bien partie de l'espace de recherche. Pour cela, nous commençons par vérifier que les opérations permettant de construire $H'_\top$ et $H'_\perp$ permettent d'obtenir une clause de $\mathcal{L}_\perp$.

Lemme 5.5.1. *Pour tout* $C, D \in \mathcal{L}_\perp$ *et* $S \subseteq next(maxdepth(C)+1, C, D)$*,* $C \cup S$ *est dans* $\mathcal{L}_\perp$.

Démonstration. $C \in \mathcal{L}_\perp$ et donc il existe une séquence $T_1 = l_1, l_2, \ldots, l_k$ des littéraux de

C telle que définie dans la définition 5.3.1. Comme

$$\begin{aligned} S \subseteq \{l \mid \quad & l \in D \\ \wedge \quad & depth(l, D) = maxdepth(C) + 1 \\ \wedge \quad & input(l) \subseteq output(C)\} \end{aligned}$$

la séquence T_1, T_2 où T_2 est n'importe quelle séquence des littéraux de S respecte également la définition 5.3.1. et donc $C \cup S$ est bien dans $\mathcal{L}_\perp$. □

Lemme 5.5.2. *Pour tout $C, D \in \mathcal{L}_\perp$ tel que $C \subseteq D$, pour tout k avec $maxdepth(C) \leq k \leq maxdepth(D)$, $narrow(k, C, D)$ est dans $\mathcal{L}_\perp$.*

Démonstration. On rappelle que :

$$narrow(k, C, D) = narrow(k, maxdepth(D), C, D)$$

Comme $C \subseteq D$, on a $maxdepth(D) \geq maxdepth(C)$. On procède avec une démonstration par récurrence pour montrer que $narrow(k, n, C, D) \in \mathcal{L}_\perp$ pour $n \geq k$.

Pour $n = k$, $narrow(n, n, C, D) = C$ et donc appartient à $\mathcal{L}_\perp$.

Supposons qu'il existe $n \geq k$ tel que $narrow(k, n, C, D) \in \mathcal{L}_\perp$. Il existe donc une séquence S_1 des littéraux de $narrow(k, n, C, D)$ comme définie dans la définition 5.3.1. Comme

$$narrow(k, n+1, C, D) = narrow(k, n, C, D) \cup\ next(n, narrow(k, n, C, D), D)$$

on peut d'une manière similaire à la preuve du lemme 5.5.1 dire que n'importe quelle séquence S_1, S_2 où S_2 est une sous séquence quelconque des littéraux de $next(n, narrow(k, n, C, D), D)$, respecte la définition 5.3.1. Donc $narrow(k, n+1, C, D)$ appartient à $\mathcal{L}_\perp$.

Par conséquent, $narrow(k, C, D)$ appartient à $\mathcal{L}_\perp$ pour tout $C, D \in \mathcal{L}_\perp$ tel que $C \subseteq D$. □

Théorème 5.5.3. *Pour tout $(H_\top, H_\perp) \in \mathcal{L}_{\perp,2}$, $\rho(H_\top, H_\perp)$ est un sous-ensemble de $\mathcal{L}_{\perp,2}$.*

Démonstration. Nous montrons que pour tout $(H_\top, H_\perp)$ de $\mathcal{L}_{\perp,2}$, quelque soit le couple $(H'_\top, H'_\perp) \in \rho(H_\top, H_\perp)$, il vérifie :

- $H'_\top, H'_\perp \in \mathcal{L}_\perp$;
- $(H'_\top, H'_\perp) \in \mathcal{L}_{\perp,2}$.

Comme $H'_\top = H_\top \cup S$, le lemme 5.5.1 nous assure que $H'_\top$ est dans $\mathcal{L}_\perp$. De même, $H'_\perp = narrow(maxdepth(H_\top) + 1, H'_\top, H_\perp)$ et $H'_\top \subseteq H_\top \cup next(maxdepth(H_\top) + 1, H_\top, H_\perp) \subseteq H_\perp$, et donc le lemme 5.5.2 nous permet de dire que $H'_\perp$ est dans $\mathcal{L}_\perp$.

Nous montrons maintenant que pour tout l tel que $0 \leq depth(l, H'_\perp) \leq maxdepth(H'_\top)$, $l \in H'_\top \Leftrightarrow l \in H'_\perp$.

Par construction de $H'_\perp$, on a $H'_\top \subseteq H'_\perp$ et donc $l \in H'_\top \Rightarrow l \in H'_\perp$.

Nous montrons maintenant l'autre sens de l'implication. Supposons qu'il existe $l \in H'_\perp$ tel que $depth(l, H'_\perp) \leq maxdepth(H'_\perp)$ et $l \notin H'_\top$. l n'appartient donc pas à $narrow(k, k, H'_\top, H'_\perp)$, où $k = maxdepth(H_\top)+1$, et doit appartenir à un des ensembles $next(n, narrow(k, n, H'_\top, H'_\perp), H'_\perp)$ où $n \geq k$. Nous montrons que quelque soit la valeur de n, l ne peut appartenir à ces ensembles. Il faut remarquer que par construction $H'_\top = narrow(k, k, H'_\top, H'_\perp)$ est contenu dans $narrow(k, n, H'_\top, H'_\perp)$. Par conséquent,

$$maxdepth(narrow(k, n, H'_\top, H'_\perp)) \geq maxdepth(H'_\top)$$

et donc,

$$\forall l' \in next(n, narrow(k, n, H'_\top, H'_\bot), H'_\bot),\ depth(l') > maxdepth(H'_\top)$$

Finalement, on peut dire que l ne peut appartenir à aucun de ces ensembles et donc être présent dans $H'_\bot$ prouvant ainsi l'équivalence.

En conclusion, on a bien que $\forall (H'_\top, H'_\bot) \in \rho(H_\top, H_\bot)$, $(H'_\top, H'_\bot)$ appartient à $\mathcal{L}_{\bot,2}$.

□

Nous allons maintenant nous intéresser aux différentes propriétés que possède l'opérateur vis-à-vis de l'espace de recherche. Pour chacune de ces propriétés, nous rappelons la définition, adaptée à notre opérateur puis nous montrons qu'il les possède.

Définition 5.5.4 (Localement fini)**.** *ρ est localement fini si pour tout $(C, D) \in \mathcal{L}_{\bot,2}$, $\rho(C, D)$ est fini et calculable.*

Définition 5.5.5 (Propre)**.** *ρ est propre si pour tout $(C, D) \in \mathcal{L}_{\bot,2}$, $\rho(C, D) \subseteq \{(C', D') \mid (C, D) \succ (C'D')\}$.*

Définition 5.5.6 (Non redondant)**.** *ρ est non redondant si pour chaque (C, D), (E, F) et (G, H) de $\mathcal{L}_{\bot,2}$, $(G, H) \in \rho^*(C, D)$ et $(G, H) \in \rho^*(E, F)$ impliquent que $(C, D) \in \rho^*(E, F)$ ou $(E, F) \in \rho^*(C, D)$.*

Théorème 5.5.4. *ρ est localement fini.*

Démonstration. Comme l'espace de recherche est fini, cela suit directement de la définition 5.5.3. □

Théorème 5.5.5. *ρ est propre.*

Démonstration. Si $C = D$, $\rho(C, D)$ est vide et donc trivialement propre. Sinon, on a $C \subset D$. Supposons qu'il existe (C', D') dans $\rho(C, D)$ tel que $(C', D') \succeq (C, D)$. Alors, $C' \subseteq C \wedge D \subseteq D'$. On rappelle que par construction $C' = C \cup S$ et $D' = narrow(maxdepth(C)+1, C, D)$. Si $S \neq \emptyset$, $C \subset C'$ et on a donc une contradiction avec l'hypothèse. Si $S = \emptyset$ alors $C' = C$ et $D' = C$. Par conséquent, $D \subset D'$ et cela forme également une contradiction. ρ est donc propre. □

Théorème 5.5.6. *ρ n'est ni $\mathcal{L}_\bot$-complet, ni faiblement $\mathcal{L}_\bot$-complet*[1]*.*

Démonstration. Prenons l'exemple graine suivant :

Layer	Literals
0	$P(X)$
1	$R(X, Y), R(X, Z)$
2	$R(Y, Z), R(Z, T)$

où R a pour mode $(+, -)$.

Si la règle cible est : $P(X) \leftarrow R(X, Y), R(Y, Z), R(T, U)$, celle-ci ne pourra être trouvée grâce à notre opérateur malgré qu'elle couvre bien la graine. En effet, si dans la première couche, nous sélectionnons uniquement $R(X, Y)$ ($R(X, Z)$ ne faisant pas partie de la règle cible), alors $R(Z, T)$ sera supprimé de la borne $H_\bot$ et ne pourra plus faire partie de notre règle.

□

1. Voir définitions 3.3.13 et 3.3.14

Théorème 5.5.7. *ρ est non redondant.*

ρ n'étant pas faiblement $\mathcal{L}_\perp$-complet, cette définition peut sembler trop générale. Nous souhaitons montrer que l'application de notre opérateur organise l'espace de recherche sous forme d'un arbre, imposant que chaque état de l'espace ne puisse être atteint que d'une seule manière.

Pour démontrer cela, nous allons commencer par démontrer le lemme ci-dessous.

Lemme 5.5.8. *Pour tout $E, F \in \rho^*(\top, \perp)$, $(C, D) \in \rho(E, F)$ implique que :*

$$D = narrow(maxdepth(E) + 1, C, F) = narrow(maxdepth(E) + 1, C, \perp)$$

.

Démonstration. Nous démontrer ce lemme par induction. Si $(E, F) = (\top, \perp)$ alors, le lemme est trivialement vrai. Supposons maintenant qu'il existe (E, F), $(G, H) \in \rho^*(\top, \perp)$ où $(E, F) \in \rho(G, H)$ implique que

$$F = narrow(maxdepth(G) + 1, E, H) = narrow(maxdepth(G) + 1, E, \perp)$$

Nous montrons que $\forall (C, D) \in \rho(E, F)$, nous avons :

$$D = narrow(maxdepth(E) + 1, C, F) = narrow(maxdepth(E) + 1, C, \perp)$$

Nous montrons que tout $l \in narrow(maxdepth(E) + 1, C, F)$ est également dans $narrow(maxdepth(E) + 1, C, \perp)$. Par définition, l appartient à F et donc appartient également à $narrow(maxdepth(E) + 1, C, \perp)$.

Nous montrons maintenant la réciproque. Si l appartient à $narrow(maxdepth(E) + 1, C\perp)$ alors, il existe une séquence $S = l_0, l_1, \dots l_n$ telle que

- $l_n = l$;
- pour tout i entre 0 et n, $l_i \in \perp$;
- $l_0 \in C$;
- pour tout i entre 0 et n, $input(l_i) \subseteq output(\bigcup_{1 \leq j < i} l_j)$.

Or comme E est contenu dans C par construction, il existe également un chemin similaire menant à l en partant d'un littéral de E. Dans cas l est dans F, (E, F) appartenant à $\mathcal{L}_{\perp,2}$, donc $l \in narrow(maxdepth(E) + 1, C, F)$ et finalement le lemme est vérifié. □

Nous allons montrer que ρ organise bien l'espace de recherche sous forme d'un arbre et donc que la propriété ci-dessous est vraie :

$$\forall (C, D), (E, F), (G, H) \in \rho^*(\top, \perp),\ (C, D) \in \rho(E, F) \wedge (C, D) \in \rho(G, H) \rightarrow (E, F) = (G, H)$$

Cette propriété représente le fait qu'un couple appartenant à $\mathcal{L}_{\perp,2}$ et atteignable en partant de $(\top, \perp)$ avec ρ ne peut être atteint de deux manières différentes.

Démonstration. Nous détaillons la preuve en fonction de la profondeur de C, E et G, sachant que ρ soit ajoute une couche non vide et donc C à une couche de plus que E et/ou G soit ajoute une couche vide et donc C est de même profondeur que E et/ou G.

1er cas $maxdepth(C) = maxdepth(E) + 1 = maxdepth(G) + 1$

Dans ce cas, les couches de E sont identiques aux $maxdepth(E)$ premières couches de C et un raisonnement similaire peut être fait avec G ce qui conduit à $E = G$. Or par le lemme 5.5.8, nous avons :

$$F = narrow(maxdepth(E), E, \perp)$$

$$H = narrow(maxdepth(G), G, \perp)$$

F est égale à H.

2ème cas $maxdepth(C) = maxdepth(E) = maxdepth(G)$

Alors la couche ajoutée pour créer C est vide et donc $E = G$ et de même que précédemment $F = H$.

3ème cas $maxdepth(C) = maxdepth(E) = maxdepth(G) + 1$

C est alors égal à E et donc $C = D$. D est par définition de ρ obtenue avec

$$narrow(maxdepth(E) + 1, C, \bot)$$

et

$$narrow(maxdepth(G) + 1, C, \bot)$$

Cela est alors impossible par définition de *narrow*.

4ème cas $maxdepth(C) = maxdepth(E) + 1 = maxdepth(G)$

La preuve est similaire au cas précédent.

□

L'opérateur ρ organise bien la recherche sous forme d'un arbre permettant d'être sûr de ne pas considérer deux fois le même état dans l'espace de recherche.

5.6 Stratégies

Nous discutons dans cette section de la manière dont nous avons utilisé cet opérateur de raffinement.

Dans un premier temps, nous allons décrire comment nous avons évalué les raffinements pour ordonner la recherche. Puis, nous discuterons des différentes stratégies pouvant être mises en œuvre.

Nous avons vu qu'il était difficile d'envisager le calcul complet des raffinements produits par $\rho(H_\top, H_\bot)$. Par conséquent, nous avons cherché à ordonner les raffinements afin de pouvoir arrêter le calcul dès qu'un raffinement conviendrait à la stratégie de recherche.

La première remarque que nous pouvons faire concerne la complétude de la clause $H'_\bot$ du couple produit par l'opérateur de raffinement. En effet, si $H'_\bot$ n'est pas correcte alors aucune des hypothèses plus spécifiques que cette hypothèse ne le sera. On peut alors ne pas considérer les raffinements violant cette contrainte. Ensuite, nous avons utilisé deux critères pour ordonner les raffinements restants :

- d'une part le nombre de littéraux ajoutés à $H_\top$, un $H'_\top$ plus petit sera privilégié à un plus grand ;
- puis, le nombre d'exemples positifs couverts servira à faire la différence.

Plus ce nombre est grand, plus la règle sera efficace dans une approche de type *separate-and-conquer*.

Pour définir formellement cet ordre, nous avons besoin d'introduire les notations pour le test de couverture.

Définition 5.6.1 (Nombre d'exemples positifs et négatifs couverts)**.** *Étant donnés une hypothèse h, le test de couverture $\succeq$ et les ensembles d'exemples positifs E^+ et négatifs E^-,*

Algorithm : LEARN$(c(e), i)$

1. $H_\top = \{vars(c(e))\}$ // the *top* hypothesis
2. $H_\bot = vars(sat(c(e), i))$ // the *bot* hypothesis
3. **while** $\rho(H_\top, H_\bot) \neq empty$
4. $\quad (H_\top, H_\bot) \leftarrow best(\rho(H_\top, H_\bot))$ // the best refinement
5. **return** $H_\top$

FIGURE 5.1 – Exemple d'approche *hill-climbing* utilisant l'opérateur ρ

nous noterons $p(h)$, le nombre d'exemples positifs couverts et et $n(h)$ le nombre de négatifs couverts définis par :

$$\begin{aligned} p(h) = &\ |\{e \mid e \in E^+ \ \wedge \ h \succeq e\}| \\ n(h) = &\ |\{e \mid e \in E^- \ \wedge \ h \succeq e\}| \end{aligned}$$

Définition 5.6.2 (Ordre sur les raffinements)**.** *Soit $\preceq_\rho$, l'ordre défini sur :*

$$\mathcal{L}_{\bot,2} \backslash \{(C, D) \mid n(D) \neq 0\}$$

permettant d'ordonner les raffinements produits par ρ et défini par :

$$\begin{aligned} (C, D) \preceq_\rho (C', D') \Leftrightarrow \ & |C| \leq |C'| \\ & \vee \ (|C| = |C'| \ \wedge \ p(D) > p(D')) \end{aligned}$$

Cet ordre nous permet de ne pas avoir à générer tous les raffinements. Ainsi, nous commençons par ceux ajoutant un seul littéral dans la couche suivante, puis deux, et ainsi de suite jusqu'à en obtenir un rejettant l'ensemble des exemples négatifs.

Cette stratégie n'a d'intérêt qu'avec les approches incomplètes, sinon il est indispensable de produire tous les raffinements. Nous nous sommes donc intéressés uniquement à des stratégies de type *hill-climbing* et *beam search*. La raison est toujours la nécessité d'éviter un parcours exhaustif de l'espace de recherche qui affecterait notre approche en pratique. Nous verrons dans la section suivante qu'un *hill-climbing* est suffisant pour réussir l'apprentissage d'un CPS. Cela s'explique par la particularité du problème où il y a peu de solutions et où notre approche permet rapidement de cibler l'endroit de l'espace de recherche où une contrainte est violée. L'approche consiste à partir du couple $(\top, \bot)$ et à appliquer progressivement l'opérateur de raffinement en fixant les couches les unes après les autres.

L'algorithme de la Figure 5.1 présente l'approche *hill-climbing* utilisé dans la section suivante. Elle commence par initialiser le couple $(H_\top, H_\bot)$ respectivement à l'exemple graine $c(e)$ et à la saturation de cet exemple correspondant à l'hypothèse $\bot$ du langage d'hypothèses. Ensuite, on raffine progressivement le couple, en ajoutant progressivement des couches de littéraux à $H_\top$ jusqu'à ce que ρ ne retourne plus de raffinement, c'est-à-dire que $H_\top = H_\bot$. On retourne alors une des deux hypothèses. Cet algorithme est une version simplifiée ne traitant pas, par exemple, le cas où il n'y a pas de raffinements corrects avec les exemples négatifs.

Il est clair que l'algorithme de la figure 5.1, même dans une version *beam search*, a des limites. Dû aux nombres exponentiels de raffinements, il n'est pas efficace dans le cas général de l'apprentissage de programme logique. Pour être efficace, cet algorithme doit s'arrêter avant d'avoir produit tous les raffinements de ρ. Il est donc dédié à des problèmes où le nombre des littéraux appartenant à la règle cible par couche est petit. C'est effectivement le cas des CPS, où par exemple le nombre de reines impliquées dans les règles ne dépasse jamais deux dans le problème des n-reines (voir Figure 4.5). Par contre, l'algorithme n'est pas adapté au problème mutagénèse[LM05] : la saturation d'un exemple est composée seulement de deux couches et tous les littéraux permettant de discriminer apparaissent dans la première.

5.7 Expériences et résultats

Nous avons testé différents systèmes de l'état de l'art en ILP sur les jeux de données de CPS (voir chapitre 4, section 4.5) : Foil[QCJ93], Beth[Tan03], Aleph[Sri], ICL[RL95], Propal[AR06], Progolem[MSTN09].

Propal est une approche descendante dirigé par les données, utilisant une *beam search* (valeur par défaut dans nos expériences). Pour Aleph, nous avons utilisé des configurations simulant des approches complètes, où nous limitions cependant le nombre de nœuds visités pour limiter la durée de l'expérience (et donc la recherche devenait incomplète). La première configuration, nommée Aleph1, correspond à une recherche en largeur avec un maximum de 200 000 nœuds visités et une liste ouverte infinie. Nous avons en fait augmenté ce nombre de nœuds jusqu'à ce que le CPS appris soit correct et complet pour les jeux de données $alea + rules$. La seconde configuration, nommée Aleph2, diffère uniquement dans la stratégie d'exploration où une recherche heuristique (type A) est utilisée. Enfin, Progolem est une approche ascendante dirigé par les données inspirée de [MF90] sur le principe d'ordonnancement des littéraux de Progol [Mug95]. Nous avons fixé une limite de temps de 10 heures pour chacun des tests.

La Figure 5.2 donne les résultats obtenus sur les jeux de données complètement aléatoires. Sans surprise, aucun système n'arrive à trouver complètement les règles décrivant le problème. Comme expliqué au chapitre 4, les approches de type *separate-and-conquer* auxquelles nous nous intéressons se contentent d'une seule règle pour discriminer un exemple. Si nous prenons notre approche, la première règle apprise est bien une règle du CPS mais il manque ensuite les autres règles du modèles pour décrire toutes les non-solutions possibles.

	Propal			Algorithm of Figure 5.1		
benchmark	# learned rules	time (s)	acc.	# learned rules	time (s)	acc.
School timetable	2	118	77.78%	1	0.19	77.78%
Job-shop	4	114	73.33%	2	2.51	68.89%
N-queens	-	-	-	1	1.82	66,67%
	Aleph1			Aleph2		
School timetable	1	0.35	77.78%	1	0.34	100%
Job-shop	2	235,63	66.67%	3	155.6	65.56%
N-queens	1	0.41	71.11%	2	205.47	71.11%
	Progolem					
School timetable	1	15.96	100%			
Job-shop	1	71.78	61.11%			
N-queens	1	63.44	61.11			

FIGURE 5.2 – Expériences sur les jeux de données *alea*

Sur les jeux de données *alea + rules*, seulement Propal (la version de [AR00], plus rapide que la version de [AR06]) et certaines configuration d'Aleph et Progolem ont réussi. La Figure 5.3 présente les résultats avec ces systèmes et l'algorithme de la Figure 5.1. La première remarque concerne les approches ascendantes. Seul Progolem a réussi mais uniquement sur les deux premiers jeux de données où les règles sont plus simples. Pour les autres, le système a été arrêté après 10 heures de calcul. La raison de cet échec vient de la taille des exemples une fois saturés qui peut devenir très importante avec une forte combinatoire pour les tests de subsomption. Plus le problème devient complexe, plus Propal et Aleph2 peinent à trouver un CPS valable. Propal a été arrêté après 10 heures de calcul pour le problème des n-reines. Cela illustre bien la difficulté rencontrée par les recherches descendantes face aux plateaux apparaissant dans nos jeux de données. Aleph1 réussit naturellement mais le temps de calcul nécessaire est très important pour les jeux de données plus complexes. Enfin notre méthode réussit sur tous les jeux de données : elle trouve des CPS précis en peu de temps. Ces résultats ne sont pas surprenants étant donnée la structure des CPS très propice à l'approche.

	Propal			Algorithm of Figure 5.1		
benchmark	# learned rules	time (s)	acc.	# learned rules	time (s)	acc.
Graph coloring	1	0	100%	1	0.17	100%
School timetable	3	11	98,33%	2	0.69	100%
Job-shop	6	103	87,78%	5	7.37	100%
N-queens	-	-	-	3	29.11	100%
	Aleph1			Aleph2		
Graph coloring	1	0.24	100%	1	0.14	100%
School timetable	1	1.24	100%	1	0.31	100%
Job-shop	3	1051.03	100%	6	1130.88	96%
N-queens	3	489.49	100%	3	560.76	61.67%
	Progolem					
Graph coloring	1	0.18	100%			
School timetable	1	11.91	100%			
Job-shop	-	-	-			
N-queens	-	-	-			

FIGURE 5.3 – Expériences sur les jeux de données *alea + rules*

5.8 Bilan et ouvertures

Suite aux difficultés dans la phase d'apprentissage décrite au chapitre 4, nous avons souhaité comprendre quelle était l'influence de nos jeux de données. La sensibilité des règles cibles engendrant des plateaux, la grande taille de l'espace de recherche et le faible nombre de solutions dans cette espace sont les raisons des échecs rencontrés par les systèmes de l'état de l'art. Cependant, nos jeux de données présentaient une singularité : leur saturation formait une structure en couches permettant, si les premiers littéraux ajoutés à une règle étaient bien choisis de vite élaguer l'espace de recherche et permettre ainsi aux approches descendantes de trouver une règle correcte. Malheureusement, les phénomènes de plateaux, fortement présents dans ces jeux de données, empêchent les heuristiques usuelles de fonctionner correctement et seules des approches complètes, coûteuses en temps, peuvent être utilisées.

Sur ce constat, nous avons développé une approche basée sur la structure en couche pouvant être induite par l'opération de saturation d'un exemple graine. En sélectionnant

couche après couche, les littéraux devant appartenir à la règle, notre système donne l'impression d'une approche descendante basée uniquement sur une hypothèse à spécialiser. Cependant les informations contenues par une telle hypothèse ne sont pas suffisante. En effet, en partant de la clause $\perp$ de l'espace de recherche, notre algorithme va progressivement supprimer les littéraux ne pouvant appartenir à la règle car inaccessibles par les littéraux que nous avons fixé comme appartenant à la règle, où encore ne pouvant être atteints en respectant la profondeur de saturation. Ainsi, nous maintenons au fil de la recherche deux bornes, représentant la partie de l'espace de recherche restant à explorer, à la manière d'un espace des versions. Pour cette raison, nous décrivons notre opérateur de raffinement comme un opérateur bi-directionnel. On peut critiquer ce choix de nommage car les deux bornes dépendent l'une de l'autre et on pourrait écrire une formalisation dépendant d'une seule de ces bornes et d'un ensemble de littéraux, la dernière couche fixée. De notre point de vue, cela est insuffisant pour dire que notre approche est ascendante ou descendante car notre idée de départ est bien de réduire l'espace de recherche en raffinant les deux bornes. D'ailleurs l'utilisation de ces deux bornes est également un frein pour notre approche qui ne peut être $\mathcal{L}_\perp$-complet et peut par conséquent passer à côté des règles correctes. *A contrario*, le fait de maintenir une borne plus spécifique offre la perspective de nouvelles heuristiques comme celles utilisées dans notre approche permettant de dépasser les problèmes liés aux phénomènes de plateaux. Nous avons ainsi réussi à trouver un compromis entre les qualités souhaitées par notre opérateur, trouver des solutions dans $\mathcal{L}_\perp$, et l'efficacité, y arriver dans un temps raisonnable et avec une bonne précision.

Notre opérateur réussit efficacement sur les CPS mais souffre de limitations dès que les exemples utilisés ne possèdent pas une certaine structure. Il serait intéressant d'étudier l'impact de ces couches sur notre algorithme mais également sur les autres systèmes de l'état de l'art. Peut-on raisonnablement développer des approches performantes pour des exemples structurés ? Comment évaluer de tels systèmes et les comparer avec l'existant ? Les études développées autour de la transition de phase tendent vers un cadre général d'évaluation des systèmes basé sur une génération aléatoire de problèmes d'apprentissage. Cependant ces modèles génèrent uniquement des exemples qui seront structurés en une seule couche et donc inintéressants pour nous. Nous pensons que proposer des modèles permettant de sérieusement étudier les biais de langage choisis de manière régulière en ILP, comme c'est le cas du langage $\mathcal{L}_\perp$ et de l'utilisation de mode et type pour les prédicats, serait un premier pas intéressant pour réfléchir à l'impact de la structure des exemples graines dans la recherche. Un premier pas serait de proposer un modèle de CPS aléatoires suffisamment objectif, pour rendre crédibles les résultats obtenus, et souple, pour permettre simplement de modifier le nombre de couches, leur taille et les liens entre littéraux (*c.-à-d.* les ensembles *input* et *output*). Ensuite, il faudrait pouvoir généraliser à des langages avec une expressivité plus grande. Une autre ouverture consisterait à étudier la possibilité d'avoir un opérateur $\mathcal{L}_\perp$-complet efficace en pratique. Les possibilités offerts par la structuration en couche sont intéressantes. On pourrait imaginer réordonner dynamiquement des littéraux afin qu'ils soient à nouveau atteignables.

Chapitre 6

Recherche interactive et acquisition de réseaux de contraintes

6.1 Introduction

Tout au long du chapitre 2, nous nous sommes interrogés sur le problème d'acquérir des modèles de CSP et ses motivations. Par souci d'accessibilité de la programmation par contraintes, l'étape cruciale de modélisation des réseaux à fait l'objet de nombreux travaux de recherche allant de l'acquisition complète d'un réseau à sa reformulation en vue de le rendre plus efficace. Mais ces problématiques ont souvent été abordées d'un point de vue utilisation de la programmation par contraintes. Comment trouver les variables de mon CSP ? Quels sont les contraintes le représentant ? Quelles sont celles qui seraient plus efficaces ? Il en est ressorti un certain nombre de critiques qui ne voyaient pas forcément la réelle utilité de certaines approches. Pourquoi apprendre un réseau de contraintes, si on peut trouver simplement des solutions à ce problème ? En effet, un utilisateur verra un intérêt à se servir d'un ordinateur que si son problème ne peut être résolu simplement en s'armant uniquement d'un crayon et d'une feuille de papier. Nous proposons dans ce chapitre un nouvel objectif : se placer en tant qu'utilisateur et souhaiter obtenir une solution à un problème pour lequel nous ne sommes pas capables de trouver de solutions. Certes, ce problème peut être réduit à l'acquisition d'un réseau de contraintes. Avec ce réseau, on peut trouver une solution au problème. Nous verrons qu'effectivement nous pouvons réutiliser une partie des travaux existants, notamment les théories sur lesquelles sont basées le système CONACQ[BCKO06, BCKO05]. Nous développerons également les spécificités et atouts qu'apportent ce nouveau cadre.

Notre problématique est définie dans la définition 2.5.1 du chapitre 2. Celle-ci met à notre disposition un CSP incomplet composé uniquement des variables et des domaines, les contraintes étant inconnues. L'utilisateur doit également fournir un ensemble de contraintes pouvant être utilisées. L'objectif est ensuite de trouver une solution, en se donnant la possibilité de questionner l'utilisateur sur des affectations, potentiellement partielles, des variables. Notre approche consistera alors à apprendre progressivement les contraintes du CSP afin d'éviter une énumération complète des affectations possibles ce qui seraient très contraignant pour l'utilisateur. Afin d'illustrer cette problématique considérons le problème du zèbre, dont la création est attribuée à Albert Einstein ou parfois à Lewis Caroll. Ce problème a la particularité d'une part de n'avoir qu'une seule solution, très difficile à trouver, et d'autre part une description consistant en plusieurs déclarations. Voici comment il se décrit. Dans une rue où il y a cinq maisons se différenciant par leur couleur, chaque

habitant possède une nationalité différente. Chacun possède sa propre boisson et fume sa propre marque de cigarette. Ils ont également un animal de compagnie différent. Le reste des contraintes s'expriment de la manière suivante :

1. L'anglais vit dans la maison rouge.
2. L'espagnol a un chien.
3. Le café est bu dans la maison verte.
4. L'ukrainien boit du thé.
5. La maison verte est directement à droite de la maison ivoire.
6. Le fumeur de Old Gold possède des escargots.
7. Les Kools sont fumés dans la maison jaune.
8. Le lait est bu dans la maison centrale.
9. Le norvégien vit dans la première maison.
10. Un des voisins de l'homme fumant des Chesterfields possède un renard.
11. Celui qui fume les Kools vit à côté de celui ayant un cheval.
12. La personne fumant des Lucky Strike boit du jus d'orange.
13. Le japonais fume des Parliaments.
14. Le norvégien vit à côté de la maison bleue.

La question est alors la suivante : où vit le zèbre ? Ce problème est intéressant dans notre approche car il permet à n'importe qui de vérifier si une affectation (partielle) des variables vérifient la définition sans pour autant avoir une idée de la manière dont formaliser ce problème. Trouver sa solution représente un challenge pour n'importe quel humain non muni d'outil de résolution de problème comme la programmation par contraintes. Notre approche consistera à construire une solution en posant progressivement des questions du type : « L'anglais, peut-il posséder le cheval ? ». La réponse ici serait « oui » puisqu'aucune déclaration ne semble s'y opposer et même si la solution infirme cette possibilité (l'ukrainien est le seul propriétaire possible de cet animal). Ces questions et leurs réponses nous permettent de trouver une partie des contraintes du problème et peuvent être ainsi utilisées pour éviter un parcours complet des affectations possibles pour trouver la solution.

Dans un premier temps, nous présenterons le système CONACQ et les clés pour comprendre son fonctionnement. Nous continuerons par une première extension de ce système afin qu'il puisse gérer les exemples partiels. Ensuite, une explication détaillée de notre approche consistant à résoudre un problème sans connaître a priori ses contraintes sera donnée en expliquant chaque point constituant notre démarche. Finalement, nous évaluerons notre approche sur un ensemble de problèmes n'offrant que peu de solutions.

6.2 CONACQ et génération de requêtes

Dans un premier temps, nous commençons par décrire CONACQ, un système visant à acquérir grâce à des exemples étiquetés, les contraintes d'un problème. Nous débuterons en donnant une intuition du fonctionnement du système qui est basé sur l'espace des versions. Nous décrirons comment les auteurs ont encodé leur approche dans un problème SAT et la manière dont les exemples sont traités. Pour rendre ce système plus efficace, CONACQ possède des mécanismes comme les règles de redondance que nous détaillerons dans la partie suivante. Enfin, nous verrons l'extension de CONACQ à l'utilisation de requêtes partielles.

6.2.1 Acquisition de contraintes, espace des versions et formulation par un problème SAT

CONACQ est donc un système visant à répondre au problème d'acquisition de réseaux de contraintes. Pour ce faire, il propose de construire le réseau en se basant sur des solutions et non-solutions du problème à résoudre. Un exemple positif pourra éliminer certaines contraintes car cet exemple ne peut être rejeté par le réseau, alors qu'un exemple négatif informe qu'il existe au moins une contrainte qui le rejette. Cela revient à maintenir un espace des versions, indiquant d'une part les contraintes ne pouvant pas être dans le réseau, raffiné grâce aux exemples positifs, et d'autre part en représentant les contraintes encore possibles, construites à partir des exemples négatifs.

Mais un des atouts principaux de CONACQ est la possibilité d'encoder le problème grâce à une formulation SAT à la fois efficace et intuitive. Ainsi dans [BCKO06, BCKO05], les auteurs montrent qu'en représentant les deux bornes par une seule théorie clausale, on peut à la fois représenter le réseau de contraintes et rendre le traitement des exemples simples à exprimer. Nous allons détailler cette formalisation dans la suite et nous nous en servirons dans les sections suivantes pour décrire notre approche.

Nous utilisons les notations propres aux CSPs déjà introduites au chapitre 2. Nous représentons donc un CSP par un triplet $\langle X, D, C\rangle$, où X est l'ensemble des variables, D les domaines et enfin C les contraintes. Cet ensemble C est dans le cas de CONACQ inconnu et doit être construit grâce aux exemples fournis par l'utilisateur et une bibliothèque de contraintes, que nous notons C_X, représentant toutes les contraintes autorisées sur les variables X. Il s'agit alors de trouver un sous-ensemble C de C_X, tel que les exemples positifs soient solutions de C et les exemples négatifs non-solutions.

Dans CONACQ, la présence d'une variable de C_X est représentée par une variable binaire qui, si elle est vraie, indique que la contrainte appartient au réseau et, si elle est fausse, qu'elle n'y est pas. Il faut être vigilant sur le fait qu'un booléen à faux représente l'absence de la contrainte et non sa négation. Ainsi, pour le CSP où $C = \emptyset$, toute affectation des variables est solution et les booléens associés aux contraintes sont à faux. Afin de simplifier la notation, nous ne considérons que des contraintes binaires, cependant la formalisation peut être étendue aux contraintes n-aires. Pour chaque contrainte $c(X_i, X_j)$ de C_X, nous associons donc une variable binaire que nous noterons b^c_{ij}. Nous utiliserons également une notation simplifiée pour $c(X_i, X_j)$ dans les exemples qui est c_{ij} comme utilisées dans [BCKO06, BCKO05].

Exemple

Considérons l'ensemble des variables $\{X_0, X_1, X_2, X_3\}$ ayant toutes comme domaine {0,1,2,3}. Nous utiliserons comme types de contraintes $\{\leq, \neq, \geq\}$. Nous supposons que le problème cible est $X_0 \leq X_3$, $X_1 \neq X_2$ et $X_2 \geq X_3$. Les variables binaires associées à ces contraintes, $\leq_{03}$, $\neq_{12}$ et $\geq_{23}$, devront être mise à vrai à la fin de l'acquisition du réseau alors que les autres seront à faux.

Les exemples nécessaires à CONACQ sont des solutions et non-solutions du problème cible. Autrement dit, ce sont des affectations pour chacune des variables du problème d'une valeur de son domaine. Une substitution peut se représenter comme une fonction qui pour chaque variable retourne la valeur qui lui est attribuée, $\sigma = (X_0 = v_0, X_1 = v_1, \ldots, X_n =$

v_n) où $X = \{X_i \mid 0 \leq i \leq n\}$ et $v_i \in D_i$ pour tout i de 0 à n. Dans la suite, nous simplifierons cette notation en indiquant uniquement les valeurs : $(v_0, v_1, \ldots, v_n)$.

Exemple (suite)

Dans l'exemple précédent, $e_1 = (0, 0, 0, 0)$ représente un exemple négatif ne satisfaisant pas la contrainte $X_1 \neq X_2$ alors que $e_2 = (0, 1, 0, 0)$ est un exemple positif.

Maintenant que les exemples sont représentés, il nous reste à présenter comment un exemple apporte une connaissance sur les valeurs des booléens associés aux contraintes. En fonction que l'exemple soit positif ou négatif, nous avons dit que le traitement serait différent. Avant de présenter ces traitements, il est important de définir un ensemble utilisé également dans la suite, il s'agit de l'ensemble des contraintes non satisfaites. Nous rappelons qu'étant donnée une substitution σ des variables d'un CSP, une contrainte $c(X_i, X_j)$ est satisfaite si $(\sigma(X_i), \sigma(X_j))$ appartient à la relation de satisfaction de la contrainte (voir section 2.2.1).

Définition 6.2.1 ($\mathcal{K}(e)$)**.** *Étant donné un exemple e, on définit l'ensemble $\mathcal{K}(e)$ comme l'ensemble des variables booléennes b^c_{ij} telles que la contrainte $c(X_i, X_j)$ ne soit pas satisfaite dans e.*

Plus formellement,

$$\mathcal{K}(e) = \{b^c_{ij} \mid c(X_i, X_j) \in C_X \ \wedge \ c(X_i, X_j) = \langle S, R\rangle \ \wedge \ (e(X_i), e(X_j)) \notin R\}$$ [1]

Pour chaque exemple, CONACQ va exploiter cet ensemble pour apporter une information complémentaire. L'idée est que CONACQ part d'une théorie clausale K vide au départ de l'algorithme. Il ajoutera dans cette théorie des clauses sur les variables b^c_{ij}. L'apprentissage sera terminé quand il n'y aura plus qu'une seule solution au problème SAT correspondant à K. Plus précisément, CONACQ exploitera l'ensemble $\mathcal{K}(e)_{[K]}$ restreint aux variables booléennes non encore fixées dans K.

Exemple (suite)

Pour les exemples e_1 et e_2, on a :

$$\mathcal{K}(e_1)_{[K]} = \{\neq_{01}, \neq_{02}, \neq_{03}, \neq_{12}, \neq_{13}, \neq_{23}\}$$

et

$$\mathcal{K}(e_2)_{[K]} = \{\geq_{01}, \neq_{02}, \neq_{03}, \leq_{12}, \leq_{13}, \neq_{23}\}$$

CONACQ traite les exemples les uns après les autres en ajoutant des clauses selon que l'exemple est une solution ou une non-solution.

Dans le cas où l'exemple est une solution, on peut déduire que les contraintes représentées dans $\mathcal{K}(e)_{[K]}$ ne peuvent être présentes dans le réseau. En effet, si une de ces contraintes étaient présentes dans le CSP, alors par définition, cet exemple ne pourrait être une solution. On ajoutera donc pour chaque b^c_{ij} de $\mathcal{K}(e)_{[K]}$, $\neg b^c_{ij}$ à K.

Quand l'exemple est une non-solution, alors il existe une contrainte représentée dans $\mathcal{K}(e)_{[K]}$ qui appartient au réseau. En effet, si l'ensemble C_X est suffisant pour décrire

1. Une contrainte est représentée par $\langle S, R\rangle$ où S est la portée de la contrainte et R sa relation de satisfaction.

le problème, au moins une des contraintes non satisfaites doit être présente pour rejeter l'exemple. On ajoutera donc la clause $\bigvee_{b_{ij}^c \in \mathcal{K}(e)_{[K]}} b_{ij}^c$ à la théorie clausale K.

Exemple (suite)

Ainsi, si CONACQ reçoit les exemples e_1 et e_2 dans cet ordre, on obtient progressivement les théorie K ci-dessous.

e	$\mathcal{K}(e)_{[K]}$	*Label*	K
$e_1 = (0,0,0,0)$	$\{\neq_{01}, \neq_{02}, \neq_{03}, \neq_{12}, \neq_{13}, \neq_{23}\}$	$-$	$(\neq_{01} \vee \neq_{02} \vee \neq_{03}$ $\vee \neq_{12} \vee \neq_{13} \vee \neq_{23})$
$e_2 = (0,1,0,0)$	$\{\geq_{01}, \neq_{02}, \neq_{03}, \leq_{12}, \leq_{13}, \neq_{23}\}$	$+$	$(\neq_{01} \vee \neq_{12} \vee \neq_{13})$ $\wedge\neg \geq_{01} \wedge\neg \neq_{02} \wedge\neg \neq_{03}$ $\wedge\neg \leq_{12} \wedge\neg \leq_{13} \wedge\neg \neq_{23}$

À remarquer que e_2 permet de fixer des valeurs pour plusieurs variables booléennes et par conséquent simplifie la première clause apprise avec e_1.

CONACQ va ainsi traiter tous les exemples fournis par l'utilisateur. Il pourra cependant s'arrêter si toutes les variables booléennes sont fixées. En effet, dans ce cas il n'y a plus rien à apprendre et l'espace des versions a convergé sur une unique hypothèse : le réseau cible.

Ceci est un point très intéressant de CONACQ puisqu'il permet de garantir à l'utilisateur que l'algorithme d'acquisition est complètement terminé et que le réseau retourné est exactement celui ciblé par l'utilisateur. Nous verrons cependant dans la section suivante que certaines optimisations sont nécessaires pour permettre, dans certains cas, d'assurer que CONACQ a bien convergé.

6.2.2 Générateurs de requêtes

Comme expliqué au chapitre 2, CONACQ a évolué pour répondre aux limites qui lui était reprochées vers la résolution d'un problème d'acquisition de réseaux de contraintes avec requêtes (Définition 2.3.2 du chapitre 2 et [BCOP07] pour la version dirigée par les requêtes). Au lieu de demander à l'utilisateur de fournir les exemples, le système doit poser des questions à l'utilisateur : des requêtes. Ce cadre d'apprentissage est assez large[Ang88] et nous ne nous intéressons qu'aux requêtes d'appartenance, consistant, dans le cas de l'acquisition de réseaux de contraintes, en des affectations des variables du CSP.

Définition 6.2.2 (Requête complète)**.** *Dans le cas de la résolution d'un problème d'acquisition d'un CSP $\langle X, D, C\rangle$, où C est inconnu, on appelle* requête complète *une affectation de toutes les variables de X à une valeur de leur domaine.*

Nous disons que la requête est positive, si l'utilisateur (l'oracle) répond que l'affectation est une solution du problème. Nous parlons sinon de requête négative.

Il faut remarquer que nous définissons ici une requête *complète* alors que dans [BCOP07] il est question simplement de requêtes. Nous avons ajouté cette qualification de « complète » pour faire la différence dans la suite avec les *requêtes partielles* que nous introduirons.

Les requêtes complètes sont donc des exemples générés par le système et étiquetés par l'utilisateur. L'intérêt est que l'utilisateur n'a plus à chercher par lui même ces exemples ce qui rend ainsi son travail plus simple. Les exemples précédemment décrits sont remplacés par ces requêtes dans le processus d'apprentissage de CONACQ. Le traitement des exemples reste le même et seule la génération de requêtes est à ajouter au système.

Pour générer des requêtes, les auteurs proposent différentes approches : affectation aléatoire des variables, génération d'une requête avec un « petit » $\mathcal{K}(e)_{[K]}$ ou encore une requête avec un « gros » $\mathcal{K}(e)_{[K]}$. Il font également remarquer qu'en générant les requêtes de cette manière, il se peut que certaines requêtes ne soient pas intéressantes. En effet, il peut arriver qu'une requête ne contienne pas de nouvelle information car déjà rejetée de l'espace des versions. On parlera alors de requête redondante.

Définition 6.2.3 (Requête redondante)**.** *Une requête redondante est une requête qui ne permet pas d'apprendre de nouvelles informations. Étant donnée une requête e, il y a deux cas de requêtes redondantes :*

- *soit $\mathcal{K}(e)_{[K]} = \emptyset$ et dans ce cas on peut automatiquement détecter que e est une solution ;*
- *soit $\mathcal{K}(e)_{[K]}$ est un sur-ensemble des littéraux d'une clause présente dans K et donc e est une non-solution.*

Ainsi ces requêtes ne présentent aucun intérêt et il n'est pas nécessaire de demander à l'utilisateur de les étiqueter.

Le dernier point à évoquer dans cette section concerne un biais utilisé dans l'apprentissage par requêtes consistant à commencer l'apprentissage en fournissant au système d'apprentissage, un exemple positif (voir [Ang88]). D'un point de vue cognitif, cela s'explique par le fait que l'enseignant doit déjà montrer un exemple positif de ce qu'il veut faire apprendre : « je veux que tu généralises les choses ressemblant à cela ». Le système d'apprentissage commence alors à poser des questions en essayant d'apprendre le concept se cachant derrière ce premier exemple. C'est également le cas dans CONACQ car les auteurs commencent également avec une solution du problème cible. Cela pose à nouveau des difficultés car cette solution doit être fournie par l'utilisateur. On retombe alors dans un des reproches fait à la première version du problème d'acquisition. Nous verrons cependant que ce cas n'est pas nécessaire pour la *recherche interactive* décrite dans ce chapitre.

6.2.3 Optimisations

Pour améliorer CONACQ et notamment réussir à faire converger l'espace des versions, les auteurs de ce système ont proposé plusieurs techniques permettant de fixer des valeurs de variables booléennes étant donnée une théorie K.

La première consiste à ajouter au problème SAT des règles de redondance. Ces règles sont des clauses sur les variables booléennes permettant d'exprimer des redondances entre les contraintes. Par exemple, pour exprimer la transitivité de $<$ sur les variables X_0, X_1 et X_2, on ajoutera, entre autre, la règle $<_{01} \wedge <_{12} \rightarrow <_{02}$ représentant la redondance entre $X_0 < X_1$, $X_1 < X_2$ et $X_0 < X_2$.

Une autre approche consiste à maintenir un *backbone* (« colonne vertébrale » en français). Ceci consiste à trouver, étant donné un problème SAT, l'ensemble des littéraux qui ne peuvent être faux et donc par conséquent fixer la valeur de ces booléens à vrai dans K. Pour cela, après chaque exemple modifiant K, on teste pour chaque littéral b_{ij}^c non fixé s'il existe encore une solution au problème $K \cup \{b_{ij}^c\}$. S'il n'y a pas de solution, on ajoute alors b_{ij}^c au *backbone*.

Dans le cas de la génération de requêtes, les auteurs de CONACQ ont proposé d'essayer de « casser » des clauses présentes dans K. En d'autres termes, il s'agit de générer une requête dont le $\mathcal{K}(e)_{[K]}$ contiendrait un sous-ensemble strict de littéraux présents dans la clause qu'on souhaiterait casser. Si cette requête est étiquetée comme une non-solution, on obtient une clause plus petite en ne gardant que les variables présentes dans le $\mathcal{K}(e)_{[K]}$ sinon on la diminue en mettant tous les littéraux du $\mathcal{K}(e)_{[K]}$ à faux. À remarquer que si on cherche une requête avec un $\mathcal{K}(e)_{[K]}$ de taille 1, et si la requête est fausse, alors la clause simplifiée a une seule variable qui doit par conséquent être vraie.

La dernière optimisation proposée par les auteurs concerne un cas se produisant si on essaye de casser des clauses. Ce cas arrive quand on force les variables non fixées, n'apparaissant pas dans la clause qu'on souhaite simplifier, à être vraies pour qu'elles n'apparaissent pas dans $\mathcal{K}(e)_{[K]}$. Dans ce cas, il est possible qu'aucune requête n'existe. Ils proposent d'extraire un ensemble de littéraux en conflit S et d'ajouter la clause $\bigvee_{b^c_{ij} \in S} \neg b^c_{ij}$ forçant au moins une des variables à être fausse. En pratique, il y a plusieurs méthodes pour produire de tels ensembles de conflit. Une méthode consiste pour chaque variable b^c_{ij} de $\mathcal{K}(e)_{[K]}$ à tester si en la fixant à vrai, alors K devient inconsistant Dans ce cas cette variable doit être fausse.

6.3 Extension de CONACQ aux requêtes partielles

Dans le cadre de la recherche interactive, les requêtes peuvent prendre la forme d'affectation partielle des variables du CSP cible. Cela présente plusieurs intérêts : les requêtes étant plus petites, elles sont plus faciles à étiqueter pour l'utilisateur, il est plus facile de produire des requêtes qui correspondent à des solutions partielles pour les problèmes ayant peu de solutions, ou encore elles peuvent être plus simples à générer dans le cas par exemple de la simplification de clauses. De plus dans le cas de notre approche basée sur l'arbre de résolution d'un CSP, ces requêtes sont très naturelles à générer puisqu'il s'agit pour l'utilisateur d'étiqueter les nœuds de l'arbre de recherche que le système ne sera pas capable d'étiqueter seul (dans le cas des requêtes redondantes, le système peut décider seul).

Étant donné un CSP $\langle X, D, C\rangle$, on parle d'affectation partielle des variables X pour une substitution incomplète des variables de X à une valeur de leur domaine. Une solution partielle est une affectation partielle où les contraintes concernées par les variables affectées de X sont satisfaites.

Une requête partielle est donc une question posée à l'utilisateur sous la forme d'une affectation partielle. Si c'est une solution partielle, elle est étiquetée positivement, sinon négativement. Pour les variables non-substituées dans une requête partielle nous utiliserons la notation Ω signifiant que la valeur pour cette variable est inconnue. Par exemple, en considérant un CSP avec trois variables X_1, X_2 et X_3 partageant le même domaine $\{1, 2\}$, une requête complète pourrait être $(1, 2, 1)$ et une requête partielle où X_2 n'est pas substituée, pourrait être $(1, \Omega, 1)$.

Pour finir avec les notations, nous introduisons une notion intéressante appelée la partie pertinent d'une requête.

Définition 6.3.1 (Partie pertinente d'une requête)**.** *Étant donnée une requête $e = (c_1, c_2, \ldots, c_n)$, où les c_i sont des constantes ou le symbole Ω, nous disons que $c_i \in e$ est pertinente si $c_i \neq \Omega$ et qu'il existe $b^c_{ij} \in \mathcal{K}(e)_{[K]}$ ou $b^c_{ji} \in \mathcal{K}(e)_{[K]}$.*

La partie pertinente $r(e)$ d'une requête e est le sous-ensemble minimal de e contenant strictement les constantes pertinentes.

L'intérêt d'une telle notion est que la partie pertinente est la seule chose utile à montrer à l'utilisateur et donc nous pouvons réduire la taille de la requête de cette manière. Par la suite, nous ne considérons que les parties pertinentes quand nous parlerons de la taille d'une requête.

Pour éviter une énumération de toutes les affectations possibles, notre approche consiste à apprendre progressivement les contraintes du problème cible. La recherche se fera alors d'une manière habituelle en programmation par contraintes, alternant les phases de branchement, affectant une valeur à une variable de X, et les phases de propagation, supprimant des domaines les valeurs ne pouvant faire partie d'une solution. Après chaque étape de propagation, notre algorithme, décrit dans la section 6.4, évalue si l'affectation courante est une solution (partielle). S'il n'est pas capable de le déterminer seul, il soumet une requête à l'utilisateur et, dépendant de la réponse, ajoute des contraintes au CSP. La redondance d'une requête partielle est identique à celle d'une requête complète. On peut donc déterminer automatiquement si une requête redondante est une solution partielle ($\mathcal{K}(e)_{[K]}$ vide) ou s'il s'agit d'une non-solution ($\mathcal{K}(e)_{[K]}$ est un sur-ensemble d'une clause de K).

Les traitements de CONACQ pour les requêtes complètes dépendent uniquement de l'ensemble $\mathcal{K}(e)_{[K]}$. Pour étendre ces traitements aux requêtes partielles, il suffit de redéfinir cet ensemble afin qu'il prenne en compte le fait que certaines variables ne sont pas substituées. Plus précisément, il faut redéfinir $\mathcal{K}(e)$ de la manière suivante :

$$\begin{aligned}\mathcal{K}(e) = \{b^c_{ij} \mid \quad & c(X_i, X_j) \in C_X \ \wedge \ c(X_i, X_j) = \langle S, R\rangle \\ & \wedge\, e(X_i) \neq \Omega \ \wedge \ e(X_j) \neq \Omega \ \wedge \ (e(X_i), e(X_j)) \notin R\}\end{aligned}$$

Alors qu'une requête partielle est plus facile à étiqueter pour l'utilisateur, elle contient en contre-partie potentiellement moins d'information. Or, les auteurs de CONACQ évaluent leur approche en regardant le nombre de requêtes nécessaires pour faire converger leur système. Cette manière d'évaluer ne prenant pas en compte la taille des requêtes, nous proposons une mesure correspondant à $\#q \times |q|$, un genre d'aire, de surface, qui sera noté $m(q)$ dans la suite. Elle permet de relativiser le nombre de requêtes avec leur taille.

6.4 Génération de requêtes biaisée par la résolution du CSP

Nous décrivons notre approche de recherche interactive pour trouver une solution dans un premier temps puis discutons de l'intérêt de à chercher toutes les solutions.

6.4.1 Rechercher une solution

Notre algorithme est une recherche usuelle alternant la propagation des contraintes et les branchements. Pour éviter une exploration complète, nous apprenons les contraintes du problèmes grâce à des requêtes. Pour chaque nœud de l'arbre de recherche, il existe plusieurs cas. Nous rappelons qu'un nœud représente une affectation partielle des variables du CSP. Si l'affectation (partielle) représentée par le nœud est une requête redondante, nous pouvons décider sans l'utilisateur si l'exploration peut continuer dans cette branche dans

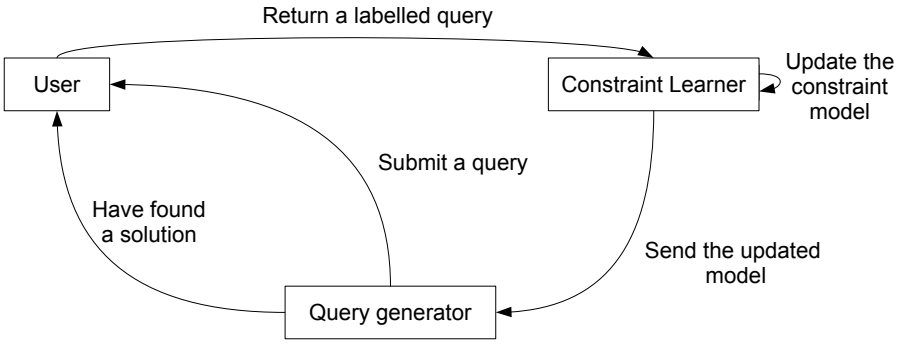

FIGURE 6.1 – Protocole de Recherche Interaction

le cas où nous avons une solution partielle ou bien s'arrêter et continuer l'exploration dans une nouvelle branche. Dans le cas où la requête n'est pas redondante, il faut demander à l'utilisateur de l'étiqueter pour continuer la recherche. Nous commençons donc par résoudre un problème sans ses contraintes et ajoutons les contraintes apprises lors de la résolution. Cette démarche est similaire à celle que l'on peut trouver dans les algorithmes de recherche *branch-and-bound*. Dans ces derniers, à chaque solution trouvée dans l'arbre de recherche une contrainte est ajouté pour forcer la prochaine solution à être meilleure. Dans notre cas, à chaque requête nous ajoutons une contrainte pour éviter de parcourir des états de recherche similaires. Pour mettre en avant cette analogie, nous appelons notre approche « *branch-and-ask* ». Ce n'est évidemment pas la seule façon d'exploiter les requêtes partielles mais celle-ci à l'avantage de fournir un cadre clair à l'utilisateur. La machine cherche à résoudre le problème dès le départ et interagit avec l'utilisateur pour cerner le modèle jusqu'à trouver une solution.

Pour généraliser, nous voyons notre processus comme un générateur de requête pour un système comme CONACQ qui serait capable de gérer les requêtes partielles. Dans [BCOP07], les auteurs proposent de générer les requêtes à la volée en fonction de l'état du processus d'apprentissage. Leur générateur ne produit que des requêtes non redondantes, utiles pour améliorer la théorie clausale courante. Les différentes étapes d'un tel système sont :

- `Génération d'une requête` q ;
- `Étiquetage de` q `par l'utilisateur` ;
- `Mise à jour de` K `avec` q `en fonction de la réponse.`

Nous présentons notre algorithme de résolution de ce point de vue. Donc, comme représenté dans la Figure 6.1, notre approche consiste en deux modules, un premier recherchant une solution dans l'arbre de résolution et qui retourne une requête quand cela est nécessaire retourne une requête, et un autre mettant à jour le modèle de contraintes du problème à la manière de CONACQ mais gérant les requêtes partielles.

En utilisant comme biais de génération de requêtes l'arbre de résolution du CSP, nous commençons par la racine de l'arbre décrivant une affectation partielle vide (aucune variable n'a de substitution). Ensuite, un nœud n_2 est obtenu par un nœud n_1 si l'affectation partielle représentée par n_2 est un sur-ensemble de celle de n_1. La structure de l'arbre dépend en fait de la manière dont sont produits les nouveaux nœuds à partir d'un existant. Par exemple, une stratégie usuelle consiste à fixer une valeur pour une variable. Il y a alors

deux nouveaux nœuds : celui où la variable a la valeur et celui où cette valeur est retirée du domaine de la variable. Mais d'autres stratégies existent comme par exemple fixer plus d'une variable. Ces stratégies forment les stratégies de branchement et pour généraliser nous dirons qu'une stratégie ajoute seulement des contraintes aux nouveaux nœuds. Ainsi, pour fixer une variable X_i à la valeur a, on ajoutera la contrainte $X_i = a$ pour créer le nœud. L'autre nœud où l'on retire a du domaine de X_i aura la contrainte $X_i \neq a$.

Comme le CSP est progressivement appris par le module de type CONACQ, il est important, avant de traiter un nœud, de mettre à jour ses contraintes. En effet, nous partons d'un nœud où il n'y a aucune contrainte et il faut les ajouter dès que celles-ci sont connues. Pour se faire nous proposons un encodage proche de celui proposé par CONACQ. Le CSP de départ de notre arbre est composé des variables X associées à leur domaine D. Nous ajoutons également les variables booléennes b^c_{ij} comme défini dans CONACQ. Pour résumer, les variables du CSP sont les suivantes :

$$\forall X_i \in X, X_i \in D_{X_i}$$
$$\forall c(X_i, X_j) \in C_X, b^c_{ij} \in \{0, 1\}$$

Les contraintes de départ de ce CSP sont alors les suivantes :

$$\forall c(X_i, X_j) \in C_X, b^c_{ij} \Rightarrow c(X_i, X_j)$$

Ainsi, une affectation d'un booléen décidera si une contrainte sera « activée » ou pas. Ces variables booléennes ayant la même sémantique que pour CONACQ, si l'une d'elles est mis à faux alors la contrainte n'est pas présente ce qui ne signifie pas que sa négation l'est. Cela explique pourquoi nous avons une implication plutôt qu'une équivalence.

La mise à jour d'un nœud revient à fixer à vrai ou faux les variables booléennes dont la valeur est connue dans le module d'apprentissage des contraintes. On ajoutera également les clauses apprises afin de limiter les requêtes redondantes.

Maintenant que les nœuds sont définis, le dernier point concerne la manière dont le générateur de requêtes va les explorer. Plusieurs stratégies peuvent être utilisées comme la recherche en profondeur (DFS), le *restart*, le *branch-and-bound*... Nous nommons ces stratégies les stratégies d'exploration.

L'algorithme de la Figure 6.2 résume le processus de génération de requêtes biaisé par l'arbre de résolution. Premièrement, introduisons quelques notations. Soit *nodes* l'ensemble ordonné des nœuds qui attendent d'être traités. Son implémentation dépend de la stratégie d'exploration. L'opération *next* de *node* retourne le premier nœud de l'ensemble et *erase* supprime le nœud en argument de l'ensemble. Pour un nœud n, l'opération *update* met à jour le modèle de contraintes du nœud en fonction de la théorie clausale courante K, *propagate* réalise la propagation des contraintes et *query* extrait la requête représentée par le nœud. Finalement, l'opération *branching*, dépendant de la stratégie de branchement, produit tous les fils possible du nœud donné en paramètre.

L'algorithme commence par récupérer le prochain nœud. Après la mise à jour des contraintes et leur propagation, la requête q est extraite du nœud. Si la requête n'est pas redondante, il faut la soumettre à l'utilisateur et l'algorithme s'arrête en la retournant. Sinon, nous pouvons déterminer s'il s'agit d'une solution (partielle) ou non. L'algorithme produit alors de nouveaux nœuds si la requête n'est pas complète et procède à un appel récursif pour explorer les nœuds suivants. Reste le cas particulier où il y a une solution

Algorithm : GENERATE

```
1.  cur ← nodes.next()
2.  cur.update(K)
3.  cur.propagate()
4.  q ← cur.query()
5.  if q is non-redundant then
6.    return q
7.  else
8.    nodes.erase(cur)
9.    if q is a (partial) solution then
10.     if q is complete query then
11.       cur is a solution
12.     else
13.       nodes ← nodes ∪ branching(cur)
14.     end if
15.   end if
16.   GENERATE
17. end if
```

FIGURE 6.2 – Génération de requêtes biaisées par l'arbre de résolution

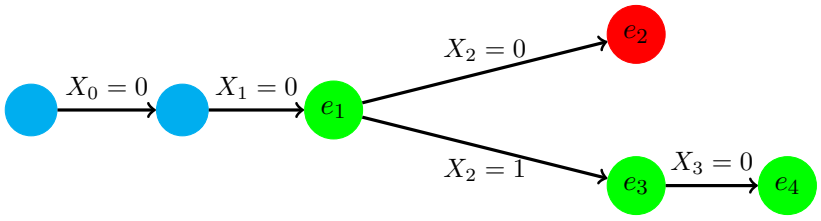

FIGURE 6.3 – Recherche d'une solution

(requête complète redondante positive). Dans le cas où nous cherchons justement une solution le processus s'arrêtera en fournissant cette solution à l'utilisateur.

Exemple

Pour illustrer notre algorithme, nous l'appliquons au problème suivant. Le problème a quatre variables $\{X_0, X_1, X_2, X_3\}$ *avec comme domaine* $\{0, 1, 2, 3\}$ *et les types de contraintes sont* $\{\leq, \neq, \geq\}$. *Les contraintes du problème cible sont* $X_0 \leq X_3$, $X_1 \neq X_2$ *et* $X_2 \geq X_3$.

Nous utilisons une recherche en profondeur d'abord, en branchant sur les variables dans l'ordre suivant X_0, X_1, X_2 *et* X_3 *avec la plus petite valeur possible. La Figure 6.3 représente l'arbre de recherche exploré par notre algorithme. Les nœuds bleus représentent des requêtes redondantes et ne sont pas soumis à l'utilisateur. Les nœuds verts et rouges représentent respectivement des requêtes positives et négatives étiquetées par l'utilisateur.*

Le tableau de la Figure 6.4 montre les contraintes apprises avec les requêtes successives. La première colonne donne les requêtes soumises à l'utilisateur. La seconde montre les $\mathcal{K}(e)_{[K]}$ *correspondant aux requêtes et la troisième la réponse de l'utilisateur,* $+$ *signifiant positif et* $-$ *négatif. La dernière montre les contraintes ajoutées à* K *qui seront également ajoutées au CSP en construction lors de la mise à jour.*

La recherche commence avec deux requêtes redondantes où le $\mathcal{K}(e)_{[K]}$ *est vide. Comme ces requêtes correspondent à des solutions (partielles) la recherche continue. La première*

e	$\mathcal{K}(e)_{[K]}$	Label	Added constraints
$e_1 = (0, 0, \Omega, \Omega)$	$\{\neq_{01}\}$	$+$	$\neg \neq_{01}$
$e_2 = (0, 0, 0, \Omega)$	$\{\neq_{02}, \neq_{12}\}$	$-$	$\neq_{02} \vee \neq_{12}$
$e_3 = (0, 0, 1, \Omega)$	$\{\geq_{02}, \geq_{12}\}$	$+$	$\neg \geq_{02} \wedge \neg \geq_{12}$
$e_4 = (0, 0, 1, 0)$	$\{\neq_{03}, \geq_{23}, \neq_{13}\}$	$+$	$\neg \neq_{03} \wedge \neg \geq_{23} \vee \neg \neq_{13}$

FIGURE 6.4 – Contraintes apprises pendant la recherche interactive d'une solution

requête soumise est e_1 qui est une solution partielle permettant d'éliminer la contrainte $X_0 \neq X_1$. Pour la seconde, l'utilisateur répond négativement et notre système apprend la clause $\neq_{02} \vee \neq_{12}$. La recherche retourne alors en arrière et branche sur $X_2 = 1$. Ensuite, nous avons deux requêtes positives pour obtenir une solution au problème.

Il est clair que sur un exemple jouet comme celui-ci, il est difficile de voir l'intérêt apporté par l'apprentissage de contraintes, qui est censé éviter de parcourir exhaustivement tout l'espace de recherche. Nous verrons cependant dans la section suivante qu'en continuant la recherche sur ce problème les contraintes apprises permettent d'élaguer une bonne partie de l'arbre de recherche. Sur des problèmes vraiment complexes, où peu de solutions existent, les contraintes apprises permettent également de converger plus efficacement vers une solution.

6.4.2 Rechercher toutes les solutions

Dans la section précédente, nous avons proposé une approche permettant de résoudre le problème de recherche interactive décrit au chapitre 2. Cependant, l'utilisateur peut souhaiter plusieurs solutions, si la première ne lui convenait pas tout à fait (même si elle respecte les contraintes de son problème). Dans ce cas, nous pouvons continuer la recherche là où elle avait été arrêtée et rechercher la prochaine solution, voire toutes les solutions. Pour cela, nous poursuivons l'exploration de l'arbre de recherche de la même manière que précédemment.

En faisant ainsi, nous pouvons nous demander s'il est possible d'apprendre le modèle exacte du problème. Comme nous le verrons dans la partie concernant les expériences que nous avons menées, notre méthode ne permet pas de garantir une convergence de l'espace des versions comme c'était le cas dans CONACQ. Cela est dû à la manière dont nous générons nos requêtes et à l'absence de règles de redondance dans notre système. D'une part, nous sommes biaisés par l'exploration de l'arbre et d'autre part par les différentes stratégies utilisées. L'ordre de branchement sur l'affectation des variables peut « éviter » (ne pas poser) certaines requêtes pourtant nécessaires pour converger. Cependant les contraintes apprises auxquelles on ajoute les clauses sur les variables non fixées sont suffisantes pour reproduire la résolution du CSP sans aucune requête pour l'utilisateur. Nous montrerons à la fin de cette section que les ensembles de solutions du CSP cible et celles du CSP construit à partir des contraintes et clauses apprises pendant la résolution sont les mêmes.

Exemple (suite)

Nous continuons la résolution du problème précédent. La Figure 6.5 et le tableau de la Figure 6.6 présentent l'arbre exploré et les contraintes apprises comme dans l'exemple précédent. Les points de suspension signifient que la recherche continue sans requête soumise à l'utilisateur.

Nous pouvons remarquer que la majeure partie de l'arbre de recherche ne nécessite pas de soumettre de requêtes à l'utilisateur, les contraintes apprises suffisant pour poursuivre.

L'utilisateur doit étiqueter seulement 4 solutions sur les 90 solutions existantes pour ce problème et trouvées avec notre approche.

On peut voir d'autres intérêts à cette approche, comme la possibilité d'utiliser la recherche interactive dans un cadre de résolution de problème d'optimisation où les contraintes sont *a priori* inconnues.

Nous finissons cette section par la preuve du théorème qui permet d'affirmer que notre approche est suffisante pour apprendre un modèle complet du CSP. Avant cela nous introduisons quelques notations nécessaires. Dans la suite, nous noterons le CSP que cible l'utilisateur CSP_c. L'ensemble des solutions d'un CSP nommé P sera noté $sol(P)$. Enfin, étant donnés le $\text{CSP}_c = \langle X, D, C\rangle$, la bibliothèque de contraintes C_X et une théorie clausale K, nous définissons le CSP CSP_K de la manière suivante :

- $\text{CSP}_K = \langle X', D', C'\rangle$
- $X' = X \cup \{b^c_{ij} \mid c(X_i, X_j) \in C_X\}$
- $D' = D \cup \{D_{b^c_{ij}} = \{0, 1\} \mid c(X_i, X_j) \in C_X\}$
- $\forall c(X_i, X_j) \in C_X, (b^c_{ij} \Rightarrow c(X_i, X_j)) \in C'$
- Pour toute clause m de K, $m \in C'$

Ce CSP possède les mêmes variables et domaines que le CSP cible, les mêmes contraintes réifiées permettant de faire le lien entre les variables booléennes et les contraintes de C_X ainsi que les clauses de la théorie clausale sous forme de contraintes réifiées quand les clauses contiennent plus d'un littéral.

Théorème 6.4.1. *Étant donnés un CSP cible* CSP_c *et la théorie clausale* K *obtenue en cherchant toutes les solutions avec un parcours complet de l'arbre de recherche, nous avons* $sol(\text{CSP}_c) = sol(\text{CSP}_K)$.

Démonstration. Pour obtenir K, notre méthode consiste de partir d'une théorie vide K_0 et d'ajouter à chaque requête, soit une clause m si la requête est négative, soit un ensemble de clauses unitaires de la forme $\neg b^c_{ij}$ si elle est positive.

Nous avons donc une séquence de raffinements $K_0, K_1, \ldots, K_n = K$ vérifiant l'implication logique $K_{i+1} \rightarrow K_i$.

Montrons que $sol(\text{CSP}_c) \subseteq sol(\text{CSP}_K)$. Supposons qu'il existe une requête complète e telle que $e \in sol(\text{CSP}_c)$ et $e \notin sol(\text{CSP}_K)$. Il existe donc un K_i rejettant e ce qui est une contradiction avec le parcours complet de l'arbre de recherche.

Montrons que $sol(\text{CSP}_K) \subseteq sol(\text{CSP}_c)$. Trivialement, nous avons $sol(\text{CSP}_c) \subseteq sol(\text{CSP}_{K_0})$. Supposons maintenant qu'il existe i compris entre 0 et n tel que $\forall s \in sol(\text{CSP}_c), s \in sol(\text{CSP}_{K_i})$. Nous montrons maintenant que $\forall s \in sol(\text{CSP}_c), s \in sol(\text{CSP}_{K_{i+1}})$. Dans le cas d'une requête positive en passant de K_i à K_{i+1}, les solutions de $\text{CSP}_{K_{i+1}}$ restent les mêmes que celles de CSP_{K_i}, la différence entre les deux théories résidant uniquement dans l'élimination de contraintes (ajout de clauses unitaires du type $\neg b^c_{ij}$). On a donc $\forall s \in sol(\text{CSP}_c), s \in sol(\text{CSP}_{K_{i+1}})$. Dans le cas d'une requête négative, nous avons $K_{i+1} = K_i \cup \{m\}$ où m est une clause. Alors, $sol(\text{CSP}_{K_{i+1}}) = sol(\text{CSP}_{K_i}) \cap (\bigcup_{b^c_{ij} \in m} sol(c(X_i, X_j))$, où $sol(c(X_i, X_j))$ est l'ensemble des affectations de X satisfaisant la contrainte $c(X_i, X_j)$. Par hypothèse $sol(\text{CSP}_c) \subseteq sol(\text{CSP}_{K_i})$, il nous reste donc à montrer $sol(\text{CSP}_c) \subseteq (\bigcup_{b^c_{ij} \in m} sol(c(X_i, X_j))$. Supposons qu'il existe une solution s de $sol(\text{CSP}_c)$ telle que pour tout b^c_{ij} de m, s n'est pas dans $sol(c(X_i, X_j))$. Or d'après la définition des traitements des requêtes (voir section 6.3), les b^c_{ij} de m devraient être à 0 comme $s \in sol(\text{CSP}_c)$. La requête aurait donc dû être positive ce qui mène à une contradiction.

Nous obtenons donc bien l'égalité entre ces deux ensembles de solutions.

□

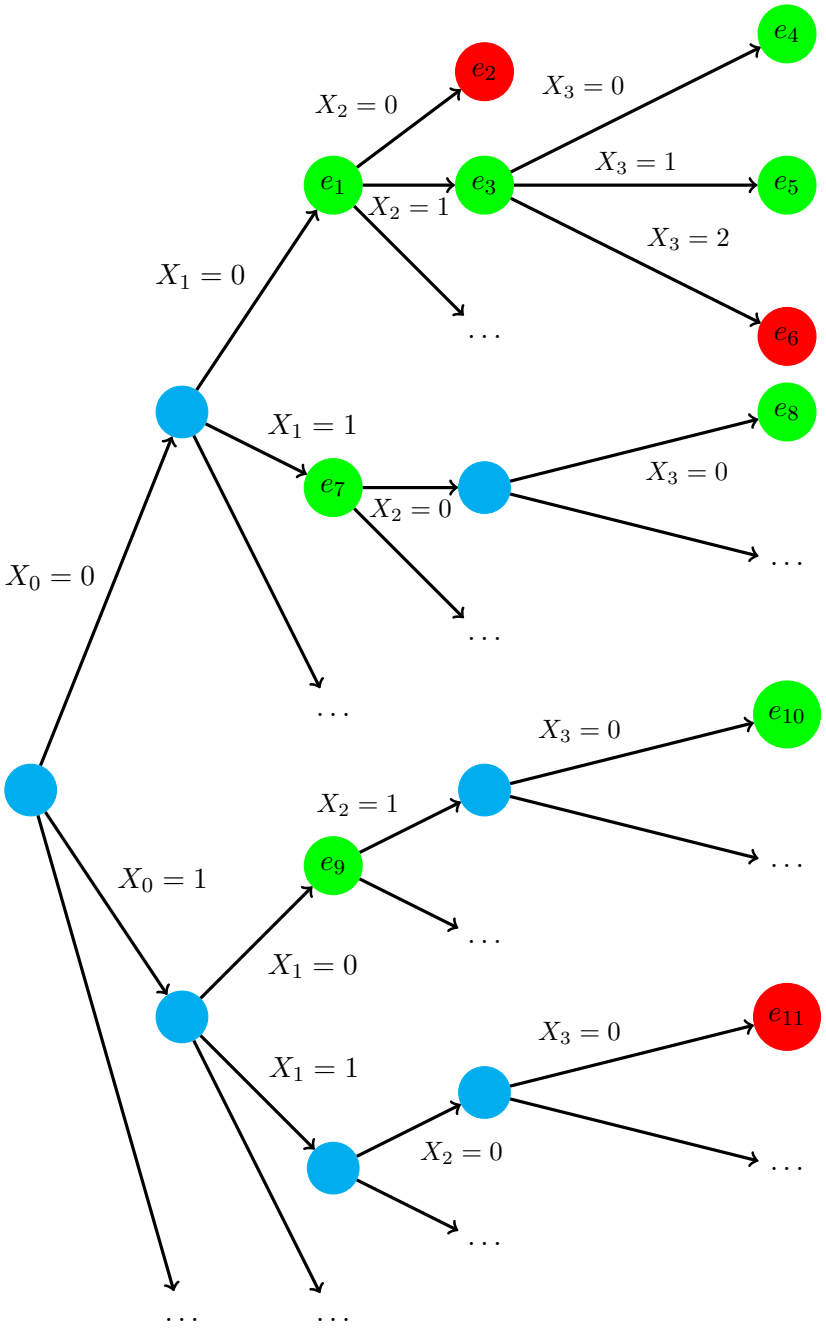

FIGURE 6.5 – Recherche de toutes les solutions

e	$\mathcal{K}(e)_{[K]}$	Label	Added constraints
$e_1 = (0, 0, \Omega, \Omega)$	$\{\neq_{01}\}$	$+$	$\neg \neq_{01}$
$e_2 = (0, 0, 0, \Omega)$	$\{\neq_{02}, \neq_{12}\}$	$-$	$\neq_{02} \vee \neq_{12}$
$e_3 = (0, 0, 1, \Omega)$	$\{\geq_{02}, \geq_{12}\}$	$+$	$\neg \geq_{02} \wedge \neg \geq_{12}$
$e_4 = (0, 0, 1, 0)$	$\{\neq_{03}, \geq_{23}, \neq_{13}\}$	$+$	$\neg \neq_{03} \wedge \neg \geq_{23} \vee \neg \neq_{13}$
$e_5 = (0, 0, 1, 1)$	$\{\neq_{23}, \geq_{03}, \geq_{13}\}$	$+$	$\neg \neq_{23} \wedge \neg \geq_{03} \vee \neg \geq_{13}$
$e_6 = (0, 0, 1, 2)$	$\{\geq_{23}\}$	$-$	$\geq_{23}$
$e_7 = (0, 1, \Omega, \Omega)$	$\{\geq_{01}\}$	$+$	$\neg \geq_{01}$
$e_8 = (0, 1, 0, 0)$	$\{\neq_{02}, \leq_{12}, \leq_{13}\}$	$+$	$\neg \neq_{02} \wedge \neg \leq_{12} \vee \neg \leq_{13}$
$e_9 = (1, 0, \Omega, \Omega)$	$\{\leq_{01}\}$	$+$	$\neg \leq_{01}$
$e_{10} = (1, 0, 1, 0)$	$\{\leq_{03}\}$	$+$	$\neg \leq_{03}$
$e_{11} = (1, 1, 0, 0)$	$\{\leq_{02}\}$	$-$	$\leq_{02}$

FIGURE 6.6 – Contraintes apprises pendant la recherche de toutes les solutions

6.4.3 Stratégies

Notre approche dépend donc de deux stratégies permettant de personnaliser le générateur de requêtes. La première est de définir une manière de créer de nouveaux nœuds à explorer. Nous présentons dans cette section, deux stratégies basées sur l'affectation d'une valeur à une variable. La seconde quant à elle décrit la manière et l'ordre dont les nœuds sont explorés.

Pour produire de nouveaux nœuds, nous avons dans un premier temps utilisé une heuristique usuelle en programmation par contraintes, appelé *dom* dans la suite. L'idée est de sélectionner la variable avec le plus petit domaine n'ayant pas encore de valeur fixe et de lui attribuer la plus petite valeur de son domaine. La seconde, nommé *maxb* que nous avons testé est plus compliquée. L'intuition est de choisir la variable qui produit une requête avec un $\mathcal{K}(e)_{[K]}$ de grosse taille. Si la requête produite est étiquetée comme positive alors le système d'apprentissage pourra fixer beaucoup de contraintes (*c.-à-d.* de variables binaires b^c_{ij}). À l'opposé s'il s'agit d'une requête négative, une clause de taille importante et peu efficace sera apprise. Cependant, alors que les générateurs de requêtes complètes ont de forte probabilité de produire une requête négative quand le problème ne compte pas beaucoup de solutions, cela n'est plus forcément le cas quand on considère les requêtes partielles. Pour sélectionner la variable, nous associons à chaque variable du problème (celle de X) une valeur correspondant au nombre de contraintes où la variable est présente dans le *scope* et dont les booléens associés ne sont pas encore fixés.

Exemple

Pour illustrer cette stratégie, considérons le problème suivant composé de trois variables X_1, X_2 et X_3 et la bibliothèque de contraintes $\{=_{12}, =_{13}, =_{23}\}$. Au début, les variables ont le même score qui est 2 (aucune contrainte n'est connue). Si $=_{12}$ est fixé à vrai, alors la prochaine variable sera X_3 avec le score 2 alors que X_1 et X_2 ont un score de 1.

Pour explorer les nœuds, nous avons la possibilité d'utiliser une recherche en profondeur (DFS). Mais nous pouvons imaginer d'autres stratégies plus complexes. Par exemple, quand une requête est étiquetée comme négative, nous pouvons vider la liste des nœuds explorés en recommençant à la manière d'un *restart* du début. Le but est d'obtenir des requêtes plus petites en repartant avec des nœuds où peu de variables sont affectées. Le dernier point consiste dans le « cassage » de clauses. Après que l'utilisateur ait étiqueté une

requête négativement, une clause est ajoutée au modèle. Nous pouvons essayer de casser ces clauses de la manière décrite précédemment en recherchant par exemple une requête fixant une contrainte apparaissant dans la plus petite clause ou en extrayant un conflit. Pour résumer, nous utilisons trois stratégies d'exploration : DFS, *DFS + clauses breaking* et *Restart + clauses breaking*.

6.5 Évaluation de la recherche interactive et de son extension à l'acquisition de modèles

Nous avons implémenté notre système avec le solveur de contraintes Gecode[httb] pour le module de génération de requêtes et le solveur Minisat[htta] pour le module d'apprentissage de contraintes. Les stratégies précédemment décrites ont été intégrées. La méthode de *backbone* provenant de CONACQ a été ajoutée mais pas l'utilisation de règles de redondance qui peut poser des problèmes (précisé en section 6.6).

Pour étudier le comportement de notre approche, nous avons utilisé des problèmes ayant peu de solutions comme le problème du zèbre. Les spécifications de ces problèmes sont données en annexes (voir Annexe A). Notre motivation initiale étant d'aider l'utilisateur à trouver des solutions à des problèmes qu'il ne peut résoudre facilement nous avons donc préféré ce genre de problème. La plupart n'ayant qu'une solution, une comparaison entre notre approche et CONACQ n'a pas réellement de sens. De plus CONACQ ne gérent pas les requêtes partielles et son objectif diffère (trouver une ou toutes les solutions pour nous, trouver les contraintes pour CONACQ). On peut cependant faire remarquer que sur les jeux de données testés dans [BCOP07] notre approche nécessite plus de requêtes. Cela est à relativiser car la taille de nos requêtes est naturellement plus petite que celles de CONACQ qui n'utilise pas de requêtes partielles. Nous proposons de concentrer notre analyse sur notre approche.

Nous avons collecté pour chaque jeux de données les statistiques suivantes : le nombre $\#q$ de requêtes soumises, la taille moyenne des requêtes notée $|q|$ et le temps nécessaire pour trouver une solution et toutes les solutions. L'étiquetage se fait automatiquement par une simulation de l'utilisateur. Ces statistiques nous ont permis de calculer $m(q)$ (voir section 6.3) qui permet de donner une idée de la qualité de l'approche testée avec un compromis entre la taille et le nombre de requêtes. Enfin, la dernière information récoltée (dans les colonnes $Conv.$) donne le nombre de variables booléennes fixées, c'est-à-dire de contraintes apprises ou rejetées, sur le nombre total de variables booléennes, c'est à dire la taille de C_X.

Les Figures 6.7, 6.8, 6.9 et 6.10 donnent les résultats obtenus sur les quatre jeux de données et les différentes heuristiques. Sans grande surprise, l'heuristique $Maxb$ est meilleure que Dom. Alors que Dom est utilisée usuellement en programmation par contraintes, $Maxb$ a été conçu dans l'optique de notre approche et privilégiant un certain type de requêtes. L'intérêt de comparer ces deux heuristiques est de montrer qu'il est possible de produire des heuristiques plus intéressantes selon le type de problème qu'on souhaite résoudre. D'un autre côté, on peut constater que comme annoncé précédemment, la recherche de toutes les solutions, possible avec un parcours complet de l'arbre de recherche, ne permet pas de faire converger. Les contraintes et clauses apprises sont cependant suffisantes pour caractériser l'ensemble des solutions et pourraient être réutilisées pour retrouver les solutions

ou faire de l'optimisation. Un fait plus étonnant concerne l'inexistence de corrélation entre le nombre de variables booléennes fixées et le nombre de requêtes. Le cassage de clauses devraient logiquement permettre de fixer plus de contraintes, ce qui est le cas, mais ne permet pas nécessairement de décroître le nombre de requêtes nécessaires. Cela peut sans doute s'expliquer par la présence des clauses dans le CSP. Même si les contraintes ne sont pas toutes fixées, les clauses apprises peuvent permettre d'améliorer l'exploration de l'arbre de recherche, évitant certains nœuds pourtant nécessaires pour faire converger l'espace des versions. Il est cependant difficile de généraliser avec les jeux de données considérés, car l'heuristique peut être déterminante dans ces cas précis. La dernière remarque concerne le nombre de requêtes qui peut paraître important, notamment si on regarde celui du problème du zèbre. Répondre à 403 questions est une tâche fastidieuse et rebutante pour un utilisateur même si les questions semblent assez simples (taille moyenne de 4). Cependant, ce problème reste un cas très difficile et il serait intéressant de chercher le score minimal, en terme de $m(q)$, qu'on pourrait espérer obtenir. Il reste que l'espace de recherche exploré contient 1790 nœuds et donc que nous n'avons pas besoin de l'utilisateur pour un bon nombre de nœuds.

Strategy	Dom					Maxb				
	$\#q$	$\|q\|$	$m(q)$	Conv.	Times	$\#q$	$\|q\|$	$m(q)$	Conv.	Times
DFS										
1 solution	109	5	545	121/198	1.88 s	116	4	464	145/198	2.60 s
all solutions	115	5	575	123/198	2.66 s	123	4	492	164/198	2.21s
DFS+clauses breaking										
1 solution	88	6	528	118/198	2.34 s	105	4	**420**	145/198	3.00 s
all solutions	93	6	558	120/198	2.83 s	111	4	**444**	153/198	2.54 s
Restart+clauses breaking										
1 solution	100	5	500	120/198	3.44 s	134	4	536	165/198	3.01s
all solutions	105	5	525	122/198	3.42 s	141	4	564	175/198	3.49 s

FIGURE 6.7 – Résultats pour Prudey's general store

Strategy	Dom					Maxb				
	$\#q$	$\|q\|$	$m(q)$	Conv.	Times	$\#q$	$\|q\|$	$m(q)$	Conv.	Times
DFS										
1 solution	60	6	360	99/198	1.40 s	88	4	**352**	131/198	1.74 s
all solutions	67	6	402	102/198	1.54 s	97	4	**388**	146/198	1.80 s
DFS+clauses breaking										
1 solution	64	6	384	103/198	2.42 s	95	4	380	131/198	2.49 s
all solutions	69	6	414	104/198	2.40 s	109	4	436	146/198	2.82s
Restart+clauses breaking										
1 solution	65	6	390	102/198	1.75 s	133	4	532	162/198	2.91 s
all solutions	69	6	414	102/198	2.12 s	140	4	560	171/198	3.13 s

FIGURE 6.8 – Résultats pour allergic reactions

Strategy	Dom					Maxb				
	$\#q$	$\|q\|$	$m(q)$	Conv.	Times	$\#q$	$\|q\|$	$m(q)$	Conv.	Times
DFS										
1 solution	132	7	924	237/360	7.53 s	160	5	**800**	285/360	7.31 s
all solutions	164	6	1148	280/360	9.18 s	175	5	875	309/360	7.71 s
DFS+clauses breaking										
1 solution	143	7	1001	247/360	8.58 s	163	5	815	287/360	8.41 s
all solutions	178	6	1002	287/360	10.05 s	177	4	**708**	309/360	8.96 s
Restart+clauses breaking										
1 solution	144	7	1008	248/360	9.09 s	166	5	830	288/360	8.59 s
all solutions	178	6	1068	284/360	11.61 s	177	4	**708**	310/360	9.37 s

FIGURE 6.9 – Résultats pour miffed millionaires

Strategy	Dom					Maxb				
	$\#q$	$\|q\|$	$m(q)$	Conv.	Times	$\#q$	$\|q\|$	$m(q)$	Conv.	Times
DFS										
1 solution	405	9	3645	1535/2100	449.41 s	400	5	2000	1973/2100	449.41 s
all solutions	511	9	4599	1741/2100	463.22 s	400	5	2000	1973/2100	463.22 s
DFS+clauses breaking										
1 solution	464	9	4176	1543/2100	523.88 s	400	6	2400	1955/2100	523.75 s
all solutions	565	9	5085	1761/2100	493.48 s	400	6	2400	1955/2100	493.48 s
Restart+clauses breaking										
1 solution	452	9	4068	1570/2100	1189.35 s	403	4	**1612**	1991/2100	495.75 s
all solutions	561	9	5049	1777/2100	1475.67 s	419	4	**1676**	2020/2100	483.32 s

FIGURE 6.10 – Résultats pour le problème du zèbre

6.6 Bilan et ouvertures

La deuxième piste étudiée pendant cette thèse consistait à reconsidérer depuis le départ la problématique d'acquisition de CSP. Que souhaite réellement l'utilisateur ? S'il n'est pas capable d'écrire un modèle de contraintes pour son problème, on peut avoir des doutes sur l'utilité des approches existantes. En se penchant plutôt sur le besoin de résoudre un problème, nous avons proposé une nouvelle problématique, baptisée la recherche interactive *branch-and-ask*. Le but est de trouver une solution à un problème que l'utilisateur ne sait modéliser mais pour lequel il peut, par contre, répondre à des questions simples. Pour cela, nous avons choisi de nous baser sur l'existant et notamment le système CONACQ afin de profiter des techniques déjà conçues ayant démontré leur efficacité. Les requêtes partielles, des questions plus simples que celles utilisées dans CONACQ, ont montré qu'elles pouvaient très facilement s'intégrer à ce système. Nous avons souhaité utiliser une approche basée sur la résolution usuelle d'un CSP afin de générer ces requêtes. Notre idée est de résoudre le problème et d'interroger l'utilisateur quand notre système ne peut décider seul. Cela permet de proposer un cadre assez souple donnant l'opportunité de développer plusieurs stratégies. Ainsi, nous avons développé plusieurs heuristiques influençant la structure de l'arbre de recherche. Les expériences menées sur des problèmes comptant peu de solutions montrent que l'approche est possible même si le nombre de requêtes nécessaires peut s'avérer au premier regard important.

Il reste de nombreuses pistes à explorer dans cette partie. Il faut comprendre avec plus de précision l'impact des contraintes apprises, et notamment les clauses, sur l'exploration de l'espace de recherche et la résolution du problème. Jusqu'à quel point simplifier une clause est-il intéressant ? Avec une meilleure compréhension des phénomènes se produisant, nous pourrions développer des heuristiques plus efficaces et intéressantes pour les types de problèmes qui nous intéressent. Il est également possible de s'éloigner du cadre proposé et s'intéresser à des générateurs de requêtes qui ne seraient plus dirigés par l'arbre de recherche mais pas d'autres données permettant de converger plus rapidement et, nous pouvons espérer, plus rapide à résoudre (en terme de nombre de requêtes). Un premier pas serait d'adapter les générateurs proposés par CONACQ[BCOP07] pour qu'ils puissent produire des requêtes partielles. D'un tout autre point de vue, nous pourrions aussi étendre la problématique aux problèmes d'optimisation. Les questions seraient alors plus complexes, puisqu'il faudrait par moment demander à l'utilisateur d'ordonner des requêtes (celle-ci est meilleure que l'autre).

Chapitre 7

Conclusion Générale

7.1 Bilan

L'intelligence artificielle a su évoluer dans notre histoire et aboutir de nos jours a de nombreuses techniques ayant montré, pour certaines d'entre elles, de belles réussites dans leur mise en pratique. La programmation par contraintes, étudiée dans cette thèse, est une de ces méthodes ayant démontré son efficacité comme par exemple dans les applications d'ordonnancement. Mais ce succès est à relativiser aux difficultés liées à son utilisation. Ainsi, il est clairement identifié par la communauté que ce manque d'accessibilité dont souffre la programmation par contraintes peut être responsable de ralentissement dans la poursuite du développement de cette thématique. D'où l'idée, à la manière de la synthèse de programmes logiques, de concevoir des techniques permettant de construire des CSP en limitant l'intervention humaine. La conception d'un CSP se divise en plusieurs étapes : choix des variables et de leur domaine, représentation formelle sous forme d'un modèle de contraintes, et enfin la reformulation de ce modèle afin de rendre sa résolution plus efficace. Il reste bien entendu la configuration des solveurs qui dans ce document n'a pas été considérée car ne faisant pas réellement partie de la phase de modélisation. Chacune de ces étapes a déjà fait l'objet de recherches en vue de son automatisation ou du moins de limiter l'intervention humaine. La plus développée est sans doute la reformulation de modèle qui va de l'ajout de contraintes redondantes à la recherche de contraintes globales, contrairement à la recherche de point de vue (variables+domaines) qui n'a pour l'instant reçu que peu d'attention. D'autre part, la construction automatique de modèle a particulièrement été étudiée dans les travaux de thèse de Rémi Coletta [Col06] introduisant le système CONACQ. Cette thèse a comme objectif de continuer les travaux concernant cette acquisition automatique de modèle en proposant d'éviter les défauts relevés dans l'état de l'art dont le principal est le besoin de solutions du problème dont on souhaite obtenir le modèle. Il est effectivement raisonnable de soulever ce point car il s'agit finalement de se reposer la question du « pourquoi a-t-on besoin d'automatiser cette étape de modélisation ? ». Cette question a servi de point de départ. La réponse qui nous a semblé la plus naturelle de notre point de vue est : « Pour ensuite résoudre le problème cible » et par conséquent il n'est plus raisonnable de demander à l'utilisateur de fournir des solutions. Nous avons alors choisi deux axes à étudier :

- demander des solutions et non-solutions de problèmes proches ;
- assister l'utilisateur dans sa recherche de solutions et améliorer la recherche en apprenant progressivement le modèle.

Nous avons donc reconsidéré le problème d'acquisition de modèle en redéfinissant les entrées du problèmes. Nous avons dans un premier temps étudié la possibilité d'utiliser des solutions/non-solutions de problèmes proches de celui ciblé par l'utilisateur. Le problème de fond restant le même mais les variables et domaines du CSP pouvant changer selon les instances. Cela nous a mené à poser un cadre se divisant en deux phases : une première construisant un modèle généralisant le problème de l'acquisition de CSP et une seconde créant le CSP pour le problème réel de l'utilisateur. Nous avons choisi d'utiliser un langage relativement simple se ramenant à un sous-ensemble des CNF en logique du premier ordre. Ce langage a été motivé par les langages de spécification fleurissant abondemment ces dernières années en programmation par contraintes. Notre but était de trouver un compromis entre l'expressivité permettant de couvrir une large famille de problèmes et l'efficacité nécessaire aux techniques d'apprentissage. Sur ce point, nous avons rencontré de sérieuses difficultés à apprendre des modèles dans notre langage. Espace de recherche trop grand, phénomènes de plateaux avec recherche aveugle et le peu de règles correctes existantes sont les explications aux échecs des systèmes et approches de la littérature. Nous avons par conséquent dû développer un nouvel algorithme tirant parti des spécificités de notre langage. Le dernier apport concerne l'instanciation d'un modèle en CSP. Cette phase de réécriture est grandement inspirée des travaux dans les langages de spécifications. Le résultat final est donc un système complet exploitant des solutions et non-solutions de problèmes proches. Certes l'objectif initial était d'enrichir le langage afin d'étendre son expressivité mais les difficultés inattendues ont empêché d'atteindre cet objectif. Le cadre actuel est cependant suffisamment ouvert pour permettre la poursuite de ces travaux.

La seconde approche consistait à repartir du souhait de base de l'utilisateur : trouver une solution à un problème. L'hypothèse que nous avons choisie est que l'utilisateur ne sait pas se servir de la programmation par contraintes mais peut par contre répondre à des questions sur l'affectation des variables. Cette idée s'inscrivait directement dans la suite de la version de CONACQ dirigée par les requêtes. La première différence était de proposer des requêtes partielles permettant ainsi d'adresser notre approche à des problèmes avec peu de solutions. La seconde se situe dans la manière dont sont générées les requêtes. En partant d'un CSP sans contrainte, nous lui cherchons une solution en affectant progressivement des valeurs aux variables. Quand notre système ne sait pas s'il doit continuer l'exploration d'une branche ou retourner en arrière, une requête est générée. À chaque requête des contraintes sont apprises et ajoutées au CSP.

Ce système a été implémenté et les résultats obtenus sur des problèmes comptant peu de solutions sont encourageants. Le nombre de requêtes nécessaires peut sembler important au premier regard mais les premières heuristiques ont montré qu'il était possible d'obtenir des ordres de grandeur plus raisonnables.

Les derniers apports de cette thèse concernent finalement les nouveaux besoins et problématiques de recherche que peuvent créer un tel sujet à mi chemin entre la programmation par contraintes et l'apprentissage automatique. Effectivement, alors que les techniques d'apprentissage devaient être pour nous de simples outils, nous avons pu constater grâce aux difficultés rencontrées qu'il nous faudrait rechercher de nouvelles approches pour parvenir à nos fins. C'est ainsi que nous avons vu qu'apprendre un CPS, un modèle dans notre langage mi-niveau, était quelque chose de complexe, soulevant un certain nombre de questions. Malgré l'existence de biais de langage, les approches tombaient dans les limites pathologiques de l'ILP. Pourtant en utilisant la structure particulière qu'offrait les CPS, nous avons proposé une nouvelle approche résolvant très rapidement nos problèmes d'ap-

prentissage, basée sur l'idée de cibler progressivement la zone de l'espace de recherche où se trouvé la ou les règles discriminantes, nous raffinons ainsi deux bornes, dans le même esprit que les algorithmes exploitant l'espace des versions, à la différence que nous ne sommes pas dirigés par les données (l'utilisation d'exemple graine mis à part). Entre ces bornes, il existe un lien, une sorte de saturation, assurant la cohérence de l'ensemble des hypothèses qu'elles représentent. Nous avons montré que nous pouvions organiser les hypothèses de notre espace de recherche en plusieurs couches. À chaque pas, notre approche va consister à ajouter une nouvelle couche à la borne la plus générale en sélectionnant un sous ensemble de littéraux présents dans la couche correspondante dans la borne la plus spécifique. Ensuite, il faut supprimer de cette dernière les littéraux ne pouvant plus être connectés s'ils étaient ajoutés à la borne spécifique. Malgré le succès sur nos jeux de données, il faut reconnaître que notre opérateur présente certains défauts. N'étant pas $\mathcal{L}_\perp$-complet, il peut ne pas trouver la solution même si elle existe dans le langage des hypothèses. De plus, le nombre de raffinements produits par notre opérateur est exponentiel par rapport à la taille de la couche considérée dans la borne spécifique. Cependant en trouvant le bon compromis, nous avons pu exploiter ces deux bornes pour utiliser une recherche incomplète capable, malgré la présence de plateaux, de trouver les bonnes hypothèses. En effet, la possibilité de concevoir des heuristiques exploitant ces deux bornes apporte une nouvelle richesse et l'opportunité de dépasser certaines limites de la programmation logique inductive.

7.2 Perspectives

Nous avons donc développé dans cette thèse deux axes autour de l'acquisition de modèles et un troisième a émergé autour de la programmation logique inductive et l'utilisation de la structure des exemples pour éviter les plateaux. Chacun de ces axes offrent des possibilités de développement futurs, déjà évoqué au fil des chapitres. Nous les rappelons puis évoquons des perspectives plus générales autour de l'acquisition de CSP.

7.2.1 Exploiter des exemples de problèmes proches

Nous avons posé un cadre qui est basé sur l'apprentissage d'un modèle général décrivant le problème de fond voulu par l'utilisateur. Le langage que nous avons introduit limite pour l'instant les problèmes qui peuvent être appris. Prenons l'exemple du problème des carrés magiques. Ce problème consiste à placer les nombres de 1 à n^2 dans une matrice de taille $n \times n$ tel que la somme des colonnes, lignes et diagonales soient égales. Ce problème offre les mêmes particularités que les problèmes que nous avons étudiés avec un problème de fond paramétré par n et des problèmes réels correspondant à ceux où une valeur est attribuée à n. Ce problème ne peut cependant pas être exprimé dans notre langage. Il nous manque effectivement la possibilité d'agréger des variables avec une opération (ici l'addition) ou encore l'utilisation de listes de variables. Il serait donc intéressant d'étudier la possibilité d'enrichir notre langage et ainsi augmenter les types de problèmes que nous pouvons traiter. Il faut cependant garder à l'esprit de ne pas trop augmenter la complexité de notre langage notamment pour permettre aux approches d'apprentissage une certaine efficacité. Nous avons évoqué au chapitre 4 la possibilité d'utiliser des objets plus complexes permettant d'éviter le plateau engendré par l'introduction des variables (hors les variables auxiliaires).

Un défaut de notre approche concerne la nécessité pour l'utilisateur de donner des non-solutions de son problème. Cette tâche peut sembler complexe, même s'il est raisonnable

de penser que l'utilisateur peut produire facilement des non-solutions en essayant de résoudre son problème. Une idée serait alors de voir comment générer de « petites » non-solutions qui pourrait ainsi être facilement étiquetées par l'utilisateur et ainsi servir de graines aux algorithmes d'apprentissage produisant de petits espaces de recherches où les opérateurs complets pourraient se montrer très intéressants. L'autre point concernant les non-solutions, quand elles sont produites par l'utilisateur, est qu'elles contiennent souvent plus d'informations que celles obtenues avec l'apprentissage de type *separate-and-conquer*. L'idée serait d'une certaine manière de faire ressortir certains motifs fréquents ou peu fréquents dans les non-solutions en faisant l'hypothèse que certains contiennent l'information suffisante pour exprimer une règle. Une nouvelle fois, on obtiendrait ainsi des exemples petits permettant de réduire significativement l'espace de recherche et ainsi se risquer à des parcours exhaustifs.

7.2.2 Rechercher une solution avec l'aide de l'utilisateur

Cette piste étant la dernière étudiée dans cette thèse, le recul n'est pas aussi fort que pour les autres apports. Mais c'est également et *a fortiori* une problématique offrant le plus d'opportunité de poursuite. Premièrement, une meilleure compréhension des mécanismes en jeu dans notre approche est nécessaire pour proposer des techniques réduisant le nombre d'interactions avec l'utilisateur, à la manière dont cela a été fait dans CONACQ. Autant les règles de redondances et simplification de clauses avaient montré leur intérêt dans CONACQ, autant dans notre recherche interactive leur impact semble moins important. D'une part les clauses apprises peuvent déjà servir dans le CSP en cours de résolution, d'autre part l'utilisation de requêtes partielles et la redondance induite par le CSP, base de notre générateur, rend plus complexe l'utilisation et la puissance des règles de redondances. Il y a cependant d'autres pistes à envisager comme le fait de réordonner l'affectation des variables pour permettre un apprentissage d'une autre partie du réseau (grâce à de petites requêtes). On peut également envisager de se séparer du biais induit par l'arbre de résolution du CSP et de partir sur des générateurs produisant des requêtes partielles de manière intelligente pour faire converger rapidement l'espace des versions. On pourrait également s'interroger sur d'autres types de requêtes, car pour l'instant seules les requêtes d'appartenances ont été étudiées. Effectivement d'autres types de requêtes, telles les requêtes de sous-ensemble ou sur-ensemble[Ang88], pourraient être intéressants pour la simplification des clauses. Il reste cependant la difficulté dans ce genre de cadre de poser un protocole général permettant de mesurer l'intérêt des approches. En fonction des types de requêtes, de leur taille, de leur forme, de l'effort à fournir pour l'utilisateur, *etc*, une approche sera plus ou moins performante mais il semble difficile de comparer les approches entre elles.

D'un tout autre point de vue, étendre la recherche interactive aux problèmes d'optimisation pourrait être une piste intéressante. Il s'agirait alors pour l'utilisateur de classer toute ou une partie des requêtes avec un score et ainsi essayer d'apprendre la notion d'optimisation se cachant derrière cela. Il y aurait deux visions, soit l'optimisation du nombre de contraintes satisfaites et du poids de chaque contrainte apprise, soit la valeur d'une solution par rapport à une autre et donc chercher à apprendre un ordre entre les solutions sous la forme par exemple d'une ou plusieurs contraintes.

7.2.3 Exploiter la structure des exemples

Notre réponse devant les difficultés rencontrées dans l'apprentissage de modèle abstrait de CSP a été d'exploiter la structure des exemples. Cette approche part naturellement d'un biais choisi pour répondre à un problème particulier. Cependant, il soulève la question de

la structure des exemples. Y a-t-il réellement des structures apparaissant dans les exemples caractérisant des classes de problèmes ? Bien entendu, ce n'est pas vrai de manière générale mais les exemples rencontrés dans cette thèse montrent bien que ce genre de cas de figure peut arriver. Nous pouvons nous interroger alors sur l'impact qu'auraient ces structures sur l'apprentissage classique en ILP, un peu comme la présence de cliques dans les réseaux de contraintes.

Par rapport à notre approche, il faudrait pouvoir mesurer l'impact des couches sur son efficacité, voire d'étendre cette étude aux autres systèmes de l'état de l'art. Un peu dans l'esprit des générateurs de problèmes d'apprentissage, on pourrait produire un générateur de problème produisant des exemples organisés en couche et le contrôler en fixant des tailles et un nombre de couches. Bien entendu, ce genre de modèle serait fait pour étudier des biais particuliers et non le cadre général comme les travaux autour de la transition de phase en ILP [SS10, AO09]. Il serait également intéressant de caractériser différents types de plateaux engendrés pour des raisons différentes liées à des caractéristiques des exemples ou de biais choisis.

7.2.4 Développer les recherches bi-directionnelles ou l'opération de saturation

Un autre point intéressant dans nos apports en programmation logique inductive concerne les opérateurs bi-directionnels. Nous avons vu que pour celui que nous proposons, la vision bi-directionnelle peut être critiquée. Mais peut-on construire d'autres opérateurs exploitant deux bornes délimitant l'espace de recherche restant à explorer ? Existe-t-il un tel opérateur à la fois efficace et $\mathcal{L}_{\perp}$-complet ? Peut-on fournir des biais intéressant pouvant limiter la taille de l'espace de recherche $\mathcal{L}_{\perp,2}$? Toutes ces questions expriment un certain nombre de pistes qu'il serait intéressant de fouiller. De tels opérateurs pourraient se montrer très efficaces pour éviter des cas pathologiques de l'ILP.

En perdant le côté bi-directionnel mais en préservant l'idée d'avoir une deuxième borne, obtenue grâce à une sorte de saturation de l'hypothèse courante, nous pourrions essayer de définir des approches permettant de prévoir la qualité d'une hypothèse afin d'améliorer les heuristiques. On peut trouver cette idée dans les approches[RM92] en ILP inspirée du *path finding* (recherche de chemin). Peut être pourrons nous envisager d'estimer un nombre de solutions possibles dans certaines parties de l'espace de recherche et ainsi privilégier l'exploration de zones plus prometteuse.

7.2.5 L'acquisition de modèles de contraintes, pourquoi et comment ?

Pour finir cette dernière partie, nous proposons une courte réflexion sur la forme et le fond de l'apprentissage de modèle de contraintes, des risques, des difficultés et des limites de cette thématique. Né d'un besoin d'augmenter l'accessibilité de la programmation par contraintes à un plus grand nombre et de types d'utilisateur, nous avons cependant vu dans cette thèse, la difficulté de bien définir les problématiques que l'on souhaite étudier. En effet, les motivations qui nous ont poussé à proposer de nouveaux cadres ont été de se rapprocher des besoins, des préoccupations réelles que pourraient avoir l'utilisateur. Tout ceci est à prendre au conditionnel, car ce problème est plutôt venu de la maturité de la communauté de la programmation par contraintes et de son envie de diffuser ses techniques, voire sa peur d'un ralentissement du développement si cela n'arrivait pas. Le piège est donc de définir les besoins, source de la problématique, d'un point de vue de programmeur par

contraintes plutôt que de se mettre dans la peau de celui qui aurait besoin de recourir à de telles techniques. On peut d'ailleurs s'interroger sur la nécessité d'utiliser ensuite la programmation par contraintes plutôt que d'autres techniques comme la programmation linéaire ou la programmation mixant les variables entières et réelles. D'ailleurs, le point de vue des concepteurs de langages de spécification est justement de s'affranchir de la méthode de résolution et de proposer des modèles pouvant ensuite être résolus par différentes techniques. Enfin, nous avons montré qu'avec la recherche interactive, nous pouvions reconsidérer l'acquisition de modèle et en faire, plutôt qu'une problématique, finalement un moyen de résoudre un autre problème. Il existe sans doute d'autres manières d'appréhender ce besoin de limiter l'intervention humaine et il serait particulièrement intéressant d'essayer de proposer un ensemble de nouveaux problèmes dépendant des entrées et des types d'utilisateurs, classés en fonction de leurs capacités et de leurs besoins.

Mais la multiplication des cadres délimitant les besoins, les entrées ou encore les capacités attribuées à l'utilisateur complique énormément l'évaluation d'une approche. Une technique est-elle satisfaisante pour répondre à la problématique ? Comment déterminer qu'une approche est meilleure qu'une autre quand les entrées ne sont pas identiques ? Une réflexion de fond sur ce qui nous semble raisonnable dans ce cadre d'acquisition de CSP permettrait une meilleure lecture des résultats et une conduite plus précise des recherches à mener.

Une autre leçon ressortant de cette thèse est la nécessité de trouver un compromis entre les besoins et l'efficacité. Si on prend la recherche interactive, il n'est pas raisonnable de s'attendre à voir chuter le nombre d'interactions avec l'utilisateur si nous partons d'un CSP sans aucune contrainte exprimée. La tâche de l'utilisateur restera fastidieuse. Si on prend l'acquisition basée sur des solutions de problèmes proches, le langage de représentation du problème ne doit pas être trop expressif pour que la phase d'apprentissage soit couronnée de succès.

Pour finir, l'opportunité de lier deux thématiques de l'intelligence artificielle est une chose très intéressante car elle donne l'occasion aux deux communautés de se rencontrer et d'échanger. On peut d'ailleurs remarquer une certaine attirance entre la programmation par contraintes et l'apprentissage. Non seulement comme présenté dans cette thèse mais également dans l'autre sens où la programmation par contraintes est utilisée pour fournir un cadre plus général voire des techniques plus rapides. C'est le cas par exemple dans la recherche d'ensemble d'items fréquents [RGN10] ou encore des algorithmes de theta-subsomption[MS04] en passant par la découverte de motif n-aires[KBC10]. Tous ces travaux montrent les possibilités d'échange entre les deux communautés et laisse présager de futurs développements.

Bibliographie

[AA] Hatem Ahriz and Ines Arana. Specifying constraint problems with z.

[Ang88] Dana Angluin. Queries and concept learning. *Mach. Learn.*, 2 :319–342, April 1988.

[AO08] Érick Alphonse and Aomar Osmani. On the connection between the phase transition of the covering test and the learning success rate in ilp. *Machine Learning*, 70(2-3) :135–150, 2008.

[AO09] Érick Alphonse and Aomar Osmani. Empirical study of relational learning algorithms in the phase transition framework. In Wray L. Buntine, Marko Grobelnik, Dunja Mladenic, and John Shawe-Taylor, editors, *ECML/PKDD (1)*, volume 5781 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2009.

[Apt99] Krzysztof R. Apt. The essence of constraint propagation. *Theor. Comput. Sci.*, 221(1-2) :179–210, 1999.

[AR00] Érick Alphonse and Céline Rouveirol. Lazy propositionalisation for relational learning. In Werner Horn, editor, *ECAI*, pages 256–260. IOS Press, 2000.

[AR06] Érick Alphonse and Céline Rouveirol. Extension of the top-down data-driven strategy to ilp. In Stephen Muggleton, Ramón P. Otero, and Alireza Tamaddoni-Nezhad, editors, *ILP*, volume 4455 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2006.

[BCDP07] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue : Past, present and future. *Constraints*, 12(1) :21–62, 2007.

[BCFO04] Christian Bessière, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2004.

[BCKO05] Christian Bessière, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In Gama et al. [GCB+05], pages 23–34.

[BCKO06] Christian Bessière, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. Acquiring constraint networks using a sat-based version space algorithm. In *AAAI*. AAAI Press, 2006.

[BCOP07] Christian Bessière, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In Veloso [Vel07], pages 50–55.

[BCP05] Christian Bessière, Remi Coletta, and Thierry Petit. Acquiring parameters of implied global constraints. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 747–751. Springer, 2005.

[BCP07] Christian Bessière, Remi Coletta, and Thierry Petit. Learning implied global constraints. In *IJCAI*, pages 44–49, 2007.

[BGSS03] Marco Botta, Attilio Giordana, Lorenza Saitta, and Michèle Sebag. Relational learning as search in a critical region. *Journal of Machine Learning Research*, 4 :431–463, 2003.

[BH03] Christian Bessière and Pascal Van Hentenryck. To be or not to be ... a global constraint. In Rossi [Ros03], pages 789–794.

[BQR06] Christian Bessiere, Joil Quinqueton, and Gilles Raymond. Mining historical data to build constraint viewpoints. In *Proceedings CP'06 Workshop on Modelling and Reformulation*, pages 1–16, 2006.

[BSL02] Stephen D. Bay, Daniel G. Shapiro, and Pat Langley. Revising engineering models : Combining computational discovery with knowledge. In *Proceedings of the 13th European Conference on Machine Learning*, ECML '02, pages 10–22, London, UK, 2002. Springer-Verlag.

[CBO+03] Remi Coletta, Christian Bessière, Barry O'Sullivan, Eugene C. Freuder, Sarah O'Connell, and Joël Quinqueton. Semi-automatic modeling by constraint acquisition. In Rossi [Ros03], pages 812–816.

[CBW99] Simon Colton, Alan Bundy, and Toby Walsh. Automatic concept formation in pure mathematics. In Thomas Dean, editor, *IJCAI*, pages 786–793. Morgan Kaufmann, 1999.

[CCM06] John Charnley, Simon Colton, and Ian Miguel. Automatic generation of implied constraints. In *ECAI*, pages 73–77, 2006.

[CIP+01] Marco Cadoli, Giovambattista Ianni, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. Np-spec : An executable specification language for solving all problems in np. In *in NP. Computer Languages*, pages 16–30. Springer-Verlag, 2001.

[CM01] Simon Colton and Ian Miguel. Constraint generation via automated theory formation. In Toby Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, pages 575–579. Springer, 2001.

[Col98] Simon Colton. Hr - automatic concept formation in finite algebras. In *AAAI/IAAI*, page 1170, 1998.

[Col06] Rémi Coletta. *Acquisition de Contraintes*. PhD thesis, Université de Montepllier II, 2006.

[Dom97] Pedro Domingos. Knowledge acquisition form examples vis multiple models. In Douglas H. Fisher, editor, *ICML*, pages 98–106. Morgan Kaufmann, 1997.

[FF05] Johannes Fürnkranz and Peter A. Flach. Roc 'n' rule learning-towards a better understanding of covering algorithms. *Machine Learning*, 58(1) :39–77, 2005.

[FGJ+07] Alan M. Frisch, Matthew Grum, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The design of essence : A constraint language for specifying combinatorial problems. In Veloso [Vel07], pages 80–87.

[FGK89] Robert Fourer, David M. Gay, and Brian W. Kernighan. Ampl : A mathematical programming language. Technical report, MANAGEMENT SCIENCE, 1989.

[FM06] Alan M Frisch and Ian Miguel. The concept and provenance of unnamed, indistinguishable types, September 2006.

[FPÅ03] Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing esra, a relational language for modelling combinatorial problems. In *LOPSTR*, pages 214–232, 2003.

[Fur99] Johannes Furnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13 :3–54, 1999.

[FW93] Dieter Fensel and Markus Wiese. Refinement of rule sets with jojo. In *Proceedings of the European Conference on Machine Learning*, pages 378–383, London, UK, 1993. Springer-Verlag.

[GCB+05] João Gama, Rui Camacho, Pavel Brazdil, Alípio Jorge, and Luís Torgo, editors. *Machine Learning : ECML 2005, 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings*, volume 3720 of *Lecture Notes in Computer Science*. Springer, 2005.

[Hen99] P. Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[Hen03] L Henocque. Modeling object oriented constraint programs in z. *CoRR*, cs.AI/0312020, 2003.

[Hni03] Brahim Hnich. *Function variables for constraint programming*. PhD thesis, Uppsala University, Amsterdam, 2003.

[htta] http ://minisat.se/. Minisat solver.

[httb] http ://www.gecode.org/. Gecode solver.

[jF92] C. D. Page jr and A. M. Frisch. Generalization and learnability : A study of constrained atoms. In S. Muggleton, editor, *Inductive Logic Programming*, pages 29–61. Academic Press, London, 1992.

[KBC10] Mehdi Khiari, Patrice Boizumault, and Bruno Crémilleux. Constraint programming for mining n-ary patterns. In David Cohen, editor, *CP*, volume 6308 of *Lecture Notes in Computer Science*, pages 552–567. Springer, 2010.

[KZ10] Ondrej Kuzelka and Filip Zelezny. Seeing the world through homomorphism : An experimental study on reducibility of examples. In *Inductive Logic Programming, 20th International Conference, ILP 2010, poster*, 2010.

[Lae02] Wim Van Laer. *From Propositional to First Order Logic in Machine Learning and Data Mining*. PhD thesis, Katholieke Universiteit Leuven, June 2002.

[Lan77] Pat Langley. Bacon : A production system that discovers empirical laws. In *IJCAI*, pages 344–344, 1977.

[LDG91] Nada Lavrac, Saso Dzeroski, and Marko Grobelnik. Learning nonrecursive definitions of relations with linus. In *Proceedings of the European Working Session on Machine Learning*, pages 265–281, London, UK, 1991. Springer-Verlag.

[LG94] David D. Lewis and William A. Gale. A sequential algorithm for training text classifiers. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '94, pages 3–12, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[LLMV10] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *ICTAI 2010, 22th IEEE International Conference on Tools with Artificial Intelligence*, 2010.

[LM05] Huma Lodhi and Stephen Muggleton. Is mutagenesis still challenging. In *ILP - Late-Breaking Papers*, 2005.

[LMV10] Matthieu Lopez, Lionel Martin, and Christel Vrain. Learning discriminant rules as a minimal saturation search. In *ILP*, 2010.

[LSB87] P. Langley, H. A. Simon, and G. L. Bradshaw. *Heuristics for empirical discovery*, pages 21–54. Springer-Verlag, London, UK, 1987.

[LSBZ87] Pat Langley, Herbert A. Simon, Gary L. Bradshaw, and Jan M. Zytkow. *Scientific discovery : computational explorations of the creative process*. MIT Press, Cambridge, MA, USA, 1987.

[MF90] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *ALT*, pages 368–381, 1990.

[Mit82] Tom M. Mitchell. Generalization as search. *Artif. Intell.*, 18(2) :203–226, 1982.

[Mit97] Tom M. Mitchell. *Machine learning*, chapter Concept learning and the general-to-specific ordering, pages I–XVII, 1–414. Number 2 in McGraw Hill series in computer science. McGraw-Hill, 1997.

[MNR+08] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the zinc modelling language. *Constraints*, 13(3) :229–267, 2008.

[Mon74] Ugo Montanari. Networks of constraints : fundamental properties and application to picture processing. In *Information Science 7*, 1974.

[MS04] Jérôme Maloberti and Michèle Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Mach. Learn.*, 55(2) :137–174, 2004.

[MSTN09] Stephen Muggleton, José Carlos Almeida Santos, and Alireza Tamaddoni-Nezhad. Progolem : A system based on relative minimal generalisation. In Luc De Raedt, editor, *ILP*, volume 5989 of *Lecture Notes in Computer Science*, pages 131–148. Springer, 2009.

[MTW+99] Patrick Mills, Edward Tsang, Richard Williams, John Ford, James Borrett, and Wivenhoe Park. Technical report csm-324 1 eacl 1.5 : An easy abstract constraint optimisation programming language, 1999.

[Mug95] Stephen Muggleton. Inverse entailment and progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4) :245–286, 1995.

[NCW97] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[NRA+96] C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, 1996.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc : Towards a standard cp modelling language. In *CP*, pages 529–543, 2007.

[OdCDPC05] Irene M. Ong, Inês de Castro Dutra, David Page, and Vítor Santos Costa. Mode directed path finding. In Gama et al. [GCB+05], pages 673–681.

[Plo70] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5, 1970.

[Pug04] Jean-Francois Puget. Constraint programming next challenge : Simplicity of use. In Mark Wallace, editor, *International Conference on Constraint Programming*, volume 3258 of *LNCS*, pages 5–8, Toronto, CA, 2004. Springer. Invited paper.

[QCJ93] J. Ross Quinlan and R. Mike Cameron-Jones. Foil : A midterm report. In Pavel Brazdil, editor, *ECML*, volume 667 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 1993.

[Qui91] J. Ross Quinlan. Determinate literals in inductive logic programming. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 746–750, Sydney, Austalia, 1991.

[RA04] Gerrit Renker and Hatem Ahriz. Building models through formal specification. In Jean-Charles Régin and Michel Rueher, editors, *CPAIOR*, volume 3011 of *Lecture Notes in Computer Science*, pages 395–401. Springer, 2004.

[RAA03] Gerrit Renker, Hatem Ahriz, and Inés Arana. A synergy of modelling for constraint problems. In Vasile Palade, Robert J. Howlett, and Lakhmi C. Jain, editors, *KES*, volume 2773 of *Lecture Notes in Computer Science*, pages 1030–1038. Springer, 2003.

[Rae96] De Raedt. *Advances in Inductive Logic Programming*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1st edition, 1996.

[RB93] Luc De Raedt and Maurice Bruynooghe. A theory of clausal discovery. In *IJCAI*, pages 1058–1063, 1993.

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[RGN10] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for data mining and machine learning. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[RL95] Luc De Raedt and Wim Van Laer. Inductive constraint logic. In *ALT*, pages 80–94, 1995.

[RM92] Bradley L. Richards and Raymond J. Mooney. Learning relations by pathfinding. In *AAAI*, pages 50–55, 1992.

[Ros03] Francesca Rossi, editor. *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*. Springer, 2003.

[Rou90] Céline Rouveirol. Saturation : Postponing choices when inverting resolution. In *ECAI*, pages 557–562, 1990.

[Rou92] C. Rouveirol. Extensions of inversion of resolution applied to theory completion. In S. Muggleton, editor, *ILP*, pages 63–92. AP, 1992.

[Sch94] Cullen Schaffer. A conservation law for generalization performance. In *ICML*, pages 259–265, 1994.

[Set09] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wincousin–Madison, 2009.

[SGS01] Alessandro Serra, Attilio Giordana, and Lorenza Saitta. Learning on the phase transition edge. In *IJCAI*, pages 921–926, 2001.

[Smi06] Barbara M. Smith. Modelling. In T. Walsh F. Rossi, P. van Beek, editor, *Handbook of Constraint Programming*, chapter 11, pages 377–406. Elsevier, 2006.

[SP91] Glenn Silverstein and Michael J. Pazzani. Relational clichés : Constraining induction during relational learning. In *ML*, pages 203–207, 1991.

[SR00] Michèle Sebag and Céline Rouveirol. Resource-bounded relational reasoning : Induction and deduction through stochastic matching. *Mach. Learn.*, 38 :41–62, January 2000.

[Sri] Ashwin Srinivasan. *A learning engine for proposing hypotheses (Aleph).*

[SS10] Lorenza Saitta and Michèle Sebag. Phase transitions in machine learning. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning*, pages 767–773. Springer, 2010.

[Tan03] Lap Poon Rupert Tang. *Integrating top-down and bottom-up approaches in inductive logic programming : applications in natural language processing and relational data mining.* PhD thesis, Department of Computer Sciences, University of Texas, 2003. Supervisor-Mooney, Raymond J.

[TD97] Ljupco Todorovski and Saso Dzeroski. Declarative bias in equation discovery. In *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, pages 376–384, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[TD01] Ljupco Todorovski and Saso Dzeroski. Theory revision in equation discovery. In Klaus P. Jantke and Ayumi Shinohara, editors, *Discovery Science*, volume 2226 of *Lecture Notes in Computer Science*, pages 389–400. Springer, 2001.

[TNM09] Alireza Tamaddoni-Nezhad and Stephen Muggleton. The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. *Mach. Learn.*, 76(1) :37–72, 2009.

[vdLNC94] Patrick R. J. van der Laag and Shan-Hwei Nienhuys-Cheng. Existence and nonexistence of complete refinement operators. In Francesco Bergadano and Luc De Raedt, editors, *ECML*, volume 784 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1994.

[Vel07] Manuela M. Veloso, editor. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007.

[VP90] R.E. Valdés-Pérez. *Machine discovery of chemical reaction pathways.* Number vol. 90 à 191 in Research paper. Carnegie Mellon University, Computer Science Dept., 1990.

Annexe A

Énoncés des problèmes étudiés au chapitre 6

A.1 Purdey's General Store

Ce problème a été posé par Jo Mason dans le Dell Logic Puzzles d'avril 1997.

Quatre familles, les Boyd, Garvey, Logan et Navarro, achètent quatre produits différents dans une boutique. Ils payent de différentes manières : une en espèce, une par crédit et les deux autres par troc. Le but est de trouver qui a acheté quoi et de quelle manière sachant que :

1. Les Boyd viennent dans la boutique pour la première fois.
2. La famille (pas les Logan) qui a échangé les pois ne l'a pas fait pour du pétrole.
3. Les Boyd et les Garvey ont acheté le pétrole et la mousseline dans un certain ordre.
4. Une famille a échangé du jambon fumé contre un sac de farine.
5. La boutique ne fait crédit qu'aux clients fidèles comme ceux ayant acheté la mousseline par crédit.

A.2 Allergic Reactions

Ce problème a été posé par Eliot George dans le Dell Logic Puzzles d'avril 1997.

Quatre amis, deux femmes appelées Debra et Janet et deux hommes Hugh et Rick, sont chacun allergiques à quelque chose de différent. Le problème est d'assortir chaque personne à son allergie et à son nom de famille (Baxter, Lemmon, Malone et Van Fleet) sachant que :

1. Rick n'est pas allergique aux moules.
2. Baxter est allergique aux œufs.
3. Hugh ne s'appelle pas Lemmon ou Van Fleet.
4. Debra est allergique à l'ambroisie.
5. Janet (dont le nom n'est pas Lemmon) n'est allergique ni aux œufs ni aux moules.

A.3 Miffed Millionaires

Ce problème a été posé par Stephanie Inglehart dans le Dell Logic Puzzles d'avril 1997.

Quatre personnes fortunées, deux femmes appelées Elena et Nina et deux hommes appelés Adam et Fritz, ont pour nom de famille : Cahill, Gantry, Ruiz et Voss dans un certain ordre. Ils ont fait fortune de différentes manières (pétrole, marché de l'or, logiciel et lotterie) et ont chacun un sujet de fixation qui les énerve. Le but est d'assortir chaque millionnaire avec son nom, la source de sa fortune et sa discussion énervante sachant

1. M. Gantry (qui n'a pas fait fortune dans le pétrole) se plaint sur les tranches d'imposition élevées.
2. L'homme qui a gagné à la lotterie se lamente au sujet de la publicité non voulue.
3. La femme (qui n'est pas M$^{\text{lle}}$ Cahill) qui a fait fortune sur le marché de l'or ne s'intéresse pas à son taux d'assurance.
4. Nina ne s'est jamais plainte des gens dépensant beaucoup d'argent.
5. Fritz (qui n'est pas Voss) ne connaît rien des ordinateurs.

Matthieu LOPEZ

Apprentissage de problèmes de contraintes

La programmation par contraintes permet de modéliser des problèmes et offre des méthodes de résolution efficaces. Cependant, sa complexité augmentant ces dernières années, son utilisation, notamment pour modéliser des problèmes, est devenue limitée à des utilisateurs possédant une bonne expérience dans le domaine. Cette thèse s'inscrit dans un cadre visant à automatiser la modélisation. Les techniques existantes ont montré des résultats encourageants mais certaines exigences rendent leur utilisation encore problématique. Dans une première partie, nous proposons de dépasser une limite existante qui résident dans la nécessité pour l'utilisateur de fournir des solutions du problème qu'il veut modéliser. En remplacement, il nous fournit des solutions de problèmes proches, c'est-à-dire de problèmes dont la sémantique de fond est la même mais dont les variables et leur domaine peuvent changer. Pour exploiter de telles données, nous proposons d'acquérir, grâce à des techniques de programmation logique inductive, un modèle plus abstrait que le réseau de contraintes. Une fois appris, ce modèle est ensuite transformé pour correspondre au problème initial que souhaitait résoudre l'utilisateur. Nous montrons également que la phase d'apprentissage se heurte à des limites pathologiques et qui nous ont contraint à développer un nouvel algorithme pour synthétiser ces modèles abstraits. Dans une seconde partie, nous nous intéressons à la possibilité pour l'utilisateur de ne pas donner d'exemples du tout. En partant d'un CSP sans aucune contrainte, notre méthode consiste à résoudre le problème de l'utilisateur de manière classique. Grâce à un arbre de recherche, nous affectons progressivement des valeurs aux variables. Quand notre outil ne peut décider si l'affectation partielle courante est correcte ou non, nous demandons à l'utilisateur de guider la recherche sous forme de requêtes. Ces requêtes permettent de trouver des contraintes à ajouter aux modèles du CSP et ainsi améliorer la recherche.
Mots clés : Acquisition de modèle, CSP, programmation par contrainte, programmation logique inductive

Constraint problems learning

Constraint programming allows to model many kind of problems with efficient solving methods. However, its complexity has increased these last years and its use, notably to model problems, has become limited to people with a fair expertise in the domain. This thesis deals with automating the modeling task in constraint programming. Methods already exist, with encouraging results, but many requirements are debatable.

In a first part, we propose to avoid the limitation consisting, for the user, in providing solutions of the problem she aims to solve. As a replacement of these solutions, the user has to provide solutions of closed problem, i.e problem with same semantic but where variables and domains can be different. To handle this kind of data, we acquire, thanks to inductive logic programming, a more abstract model than the constraint network. When this model is learned, it is translated in the very constraint network the user aims to model. We show the limitations of learning method to build such a model due to pathological problems and explain the new algorithm we have developed to build these abstract models. In a second part, we are interesting in the possibility to the user to not provide any examples. Starting with a CSP without constraints, our method consists in solving the problem the user wants in a standard way. Thanks to a search tree, we affect to each variable a value. When our tool cannot decide if the current partial affectation is correct or not, we ask to the user, with yes/no queries, to guide the search. These queries allow to find constraints to add to the model and then to improve the quality of the search.
Keywords : model acquisition, CSP, constraint programming, inductive logic programming

Laboratoire d'Informatique Fondamentale d'Orléans
Batiment IIIA, Rue Léonard de Vinci
B.P. 6759 F-45067 ORLEANS Cedex 2