Problèmes d'apprentissage de contraintes

Arnaud Lallouet, Matthieu Lopez, Lionel Martin

Problèmes d'apprentissage de contraintes

Arnaud Lallouet[1], Matthieu Lopez[2], Lionel Martin[2]
[1]GREYC, Université de Caen, [2]LIFO, Université d'Orléans
[1]prenom.nom@info.unicaen.fr, [2]prenom.nom@univ-orleans.fr

Abstract

Il est reconnu que la création d'un modèle de réseaux de contraintes requiert une bonne expérience du domaine. Pour cette raison, des outils pour générer automatiquement de tels réseaux ont gagné en intérêt ces dernières années. Ce papier présente un système basé sur la programmation logique inductive capable de construire un modèle de contraintes à partir de solutions et non-solutions de problèmes proches. Le modèle est exprimé dans un langage mi-niveau. Nous montrons que les approches de PLI classique ne sont pas capables de résoudre cette tâche d'apprentissage et nous proposons une nouvelle approche basée sur le raffinement d'une solution appelée graine. Nous présentons des résultats expérimentaux sur des jeux de données allant des puzzles aux problèmes d'emploi du temps.

1 Introduction

La programmation par contraintes (CP) est un formalisme pour modéliser et résoudre une large gamme de problèmes de décision, des puzzles arithmétiques aux emplois du temps en passant par les problèmes de programmation de tâches industrielles. Cependant, il est reconnu par la communauté [16] que le modelage en CP requiert une connaissance approfondie pour être produit avec succès. Des problèmes majeurs pour les utilisateurs débutants sont leur connaissance très limitée sur le choix des variables, comment trouver les contraintes et comment améliorer leur modèle pour le rendre efficace. Dans ce processus, trouver les bonnes contraintes est crucial et il existe beaucoup de travaux sur la compréhension[18] et l'automatisation[4] de la tâche de modelage.

A notre connaissance, seulement le système CONACQ dans [9] et les papiers ultérieurs [4, 5] cherche à apprendre un réseau de contraintes avec un algorithme basé sur l'espace des versions. Cependant, une limitation principale de l'approche est que l'utilisateur doit fournir l'ensemble exact des variables ainsi que des solutions et des non-solutions pour son problème. Qu'un utilisateur veuille construire un modèle alors qu'il a déjà des solutions est discutable. En contraste, et d'un point de vue cognitif, il est plus intéressant pour un utilisateur d'obtenir un modèle pour un nouveau problème en ayant des solutions et non-solutions de problèmes liés. Pour illustrer, nous considérons qu'un utilisateur veut modéliser un problème d'emploi du temps scolaire en ayant seulement des solutions et non-solutions, produites à la main, pour des instances historiques des précédentes années. Malgré la généralisation à un cadre apprentissage actif[5], l'utilisateur doit fournir à CONACQ un jugement sur des solutions potentielles du problème actuel.

Certains langages de modelage comme OPL [12], Essence' [10] ou MiniZinc[15] fournissent un cadre pour modéliser des problèmes de contraintes avec un niveau d'abstraction moyen. L'utilisateur fournit des règles et des paramètres qui sont combinés dans un processus de réécriture pour générer un Problème de Satisfaction de Contraintes (CSP) adapté au véritable problème à résoudre. Apprendre une telle spécification à partir de problèmes déjà résolus (données historiques) fournirait un modèle qui pourrait être réutilisé dans un nouveau contexte avec différents paramètres. Par exemple, après la génération du problème d'emploi du temps scolaire, le modèle pourra être construit à partir des données courantes comme le nombre de cours, d'enseignants, les nouvelles salles de cours ...

Dans ce papier, nous présentons une manière d'acquérir une telle spécification 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 appelons *langage de description* et le CSP en sortie est exprimé par des contraintes dans un *langage de contraintes*. La spécification est exprimée par des règles du premier ordre qui associent à un ensemble de prédicats dans le langage de description (corps d'une

règle), un ensemble de contraintes dans le langage de contraintes (tête d'une règle). Nous n'utilisons pas directement un langage de modelage comme Essence ou Zinc pour rester proche d'un système de règles mais les règles que nous apprenons ont un niveau d'abstraction équivalent aux langages de modelage intermédiaire comme Essence', Minizinc ou OPL. En particulier, ils permettent l'utilisation d'arythmétique et de paramètres et ils peuvent être réécrits pour générer un problème contraint de taille différente. Chercher de telles règles est un réel problème pour les techniques d'ILP puisque nos règles présentent des difficultés reconnus dans la communauté ILP telles la recherche aveugle et la traversée de "plateau". Pour dépasser ces difficultés, nous proposons une nouvelle technique basée sur une recherche bidirectionnelle consistant à progressivement réduire les bornes de l'espace de recherche. Pour cela, nous nous basons sur la structure de la saturation d'un exemple pour biaiser l'espace de recherche. La saturation est un objet bien connu en ILP permettant de connaître les différentes caractéristiques d'un exemple en fonction d'une base de connaissance.

Nous organisons ce papier de la manière suivante. Nous introduisons le langage de règles et décrivons comment un modèle peut se réécrire en CSP. Nous présentons ensuite le problème d'apprentissage sous la forme d'un problème d'ILP classique et fournissons les clés pour comprendre et reproduire notre algorithme. Nous présentons divers résultats sur des problèmes classiques de contraintes comme l'emploi du temps, l'ordonnancement d'ateliers et le classique n-reines.

2 Apprentissage de problèmes contraints

2.1 Le langage cible

Soient V un ensemble de variables et $D = (D_X)_{X \in V}$ leurs domaines. Une *contrainte* est une relation c sur un sous-ensemble de variables. Nous notons $var(c)$ les variables sur lesquelles c est définie et par $sol(c) \subseteq D^{var(c)}$ l'ensemble des tuples définissant c. Un CSP est un triplé (V, D, C), où V et D sont définis comme ci-dessus et C est un ensemble de contraintes. Un langage de modelage fournit une manière de spécifier un CSP avec une certaine abstraction. De nombreux langages donnent la possibilité d'utiliser des arguments pour paramétrer leurs modèles [10, 14]. Les paramètres sont alors fournis dans un fichier séparé. Nous présentons dans ce papier un langage de modelage en logique du premier ordre qui retient la notion de paramètres et pose des contraintes pour toutes les substitutions de variables en accord avec les paramètres. Nous ne considérons pas des objets comme les fonctions, souvent utilisées par langage de modelage destiné à un utilisateur humain. Nous appelons notre langage $\mathcal{ML}$.

Une spécification de problème contraint (CPS) dans le langage $\mathcal{ML}$ consiste en un ensemble de règles décrivant quand une contrainte doit être posée dans une instance d'un CSP. Soit T un ensemble de types. Soit $V = (V_t)_{t \in T}$ et $(Const_t)_{t \in T}$ respectivement un ensemble de variables typées et de constantes. Un terme est soit une variable soit une constante. Les prédicats ont aussi des types et sont divisés en deux ensembles disjoints P_D et P_C correspondant respectivement au corps et à la tête de la règle. Les prédicats du corps forment le *langage de description*. Ils sont utilisés pour décrire les exemples et contre-exemple et pour introduire les variables de la règle. Ils ont également une ou plusieurs déclarations de mode : pour chaque argument, on précise s'il s'agit d'une entrée ou d'une sortie du prédicat. Les entrées sont notées $+$ et les sorties $-$. Par exemple, le prédicat *sum(X, Y, Z)* avec la sémantique $X + Y = Z$ pourrait avoir le mode *sum(+,+,-)*. Les prédicats de la tête forment le *langage de contraintes* et sont utilisés pour définir des contraintes qui doivent être satisfaites si le corps est vrai. Ces prédicats sont les précurseurs des contraintes et seront transformés en contraintes pendant la phase de réécriture. Un atome est une expression de la forme $p(t_1, \ldots, t_k)$, où p est un prédicat d'arité k et $t_1, \ldots, t_k$ des termes. Un littéral est un atome ou la négation d'un atome.

La syntaxe de nos règles est la suivante :

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

La figure 2 présente quelques exemples de problèmes spécifiés avec ce langage. Nous avons volontairement enlevé les quantifications universelles de variables pour des raisons de place.

Le premier exemple correspond au problème de coloration de graphe où deux sommets adjacents ne peuvent avoir la même couleur. Le second est un problème d'emploi du temps simplifié où $timetable(L, T, R, S)$ représente un cours L enseigné par un professeur T dans une salle R pendant le créneau S. Deux cours ne peuvent avoir lieu dans la même salle s'ils ont lieu au même moment (première règle) et un enseignant ne peut enseigner deux cours différents au même moment. Le dernier problème est un problème d'organisation d'atelier. Une tâche J de type T doit être faite entre les heures B et E avec la machine M. Les machines peuvent faire tous les type de

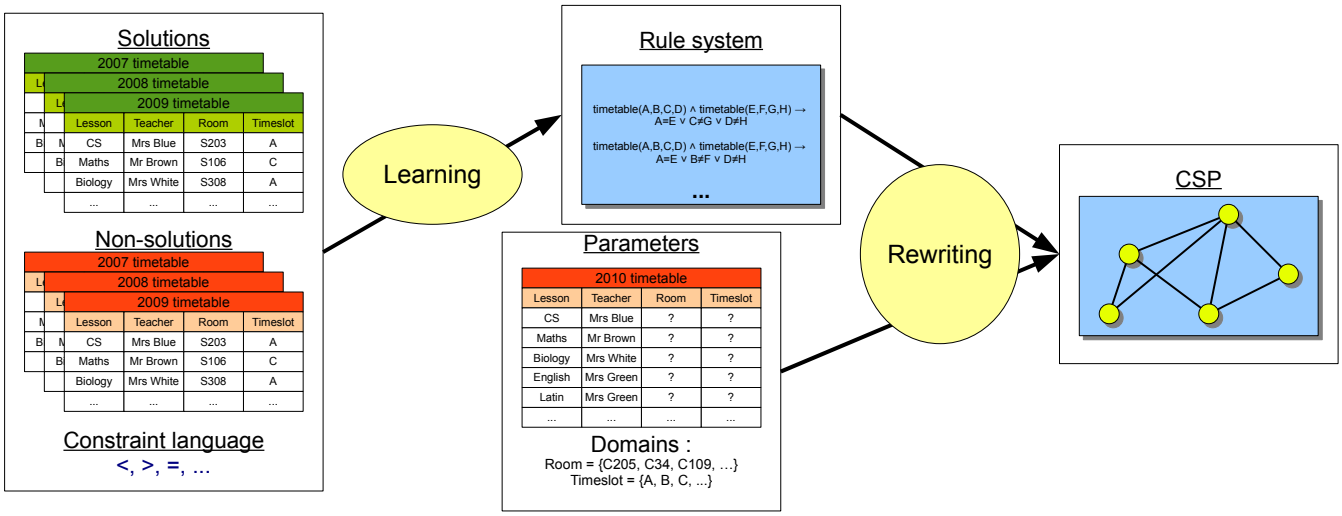

FIGURE 1 – Cadre pour l'apprentissage de problèmes contraints

Graph coloring problem

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

Simplified school timetable

$$timetable(L_1, T_1, R_1, S_1) \wedge timetable(L_2, T_2, R_2, S_2) \rightarrow L_1 = L_2 \ \vee \ R_1 \neq R_2 \ \vee \ S_1 \neq S_2$$
$$\bigwedge$$
$$timetable(L_1, T_1, R_1, S_1) \wedge timetable(L_2, T_2, R_2, S_2) \rightarrow T_1 \neq T_2 \ \vee \ L_1 = L_2 \ \vee \ S_1 \neq S_2$$

Simplified jobshop

$$schedule(J, T, B, E, M) \rightarrow B < E$$
$$\bigwedge$$
$$schedule(J_1, T_1, B_1, E_1, M_1) \wedge schedule(J_2, T_2, B_2, E_2, M_2) \rightarrow J_1 = J_2 \ \vee \ M_1 \neq M_2 \ \vee \ B_1 > E_2 \ \vee \ E_1 < B_2$$
$$\bigwedge$$
$$schedule(J_1, T_1, B_1, E_1, M_1) \wedge schedule(J_2, T_2, B_2, E_2, M_2) \rightarrow J_1 = J_2 \ \vee \ E_1 < B_2 \ \vee \ prev(T_1, T_2)$$

FIGURE 2 – Exemples de CPS

travaux mais certains types doivent être faits avant d'autres (*prev* décrit l'ordre des types de tâches).

En contraste avec les langages de modelage classiques, la présence de disjonction rend difficile la compréhension pour un utilisateur humain. Cependant, la majorité de ces disjonctions sera supprimée lors de la compilation du CSP.

2.2 Instanciation d'un CPS

Prenons comme exemple le problème d'emploi du temps. Nous supposons que suite à la phase d'apprentissage l'utilisateur a obtenu un CPS dans le langage décrit précédemment. Pour obtenir un modèle de CSP, il doit fournir une table partiellement remplie représentant le prédicat *timetable* (voir fig. 1) et les domaines liés à son problème actuel qui sont, dans l'exemple, les salles et les créneaux. Ces tables partielles, qu'on appellera par la suite *extension partielle*, permettent de fixer les paramètres du CPS.

L'objectif de l'utilisateur est alors d'obtenir un CSP pouvant compléter cette extension partielle. Dans la figure 1, il s'agit de déterminer les salles et les créneaux où auront lieux les cours sachant que les professeurs ont déjà été choisis.

La traduction d'un CPS en CSP se décompose en deux phases. La première va compléter les extensions partielles avec des variables de CSP en leur fixant le domaine correspondant. La seconde va, pour chaque règle du CPS, chercher les substitutions possibles du corps et produire les contraintes correspondantes grâce à la tête. L'algorithme est présenté dans la figure 2.2 et est expliqué dans la suite. Il prend en paramètre un modèle CS , correspondant au CPS décrivant son problème, et les données actuelles de son problèmes : un ensemble d'extension partielle et les domaines pour chaque type.

Une extension partielle d'un prédicat p est un couple (p, E), où E est un ensemble de tuples $\langle x_1, x_2, \ldots, x_k\rangle$ définissant ce prédicat. x_i est soit une constante soit ? signifiant que l'utilisateur ne connaît pas la valeur pouvant être associée au reste du tuple. Dans la suite, on notera *ext* l'ensemble des extensions partielles fournies par l'utilisateur. La première phase (ligne 3) consiste à compléter *ext* en remplaçant les ? par des variables de CSP avec le bon domaine (donné par le prédicat).

La seconde phase consiste à produire à partir des règles du CPS les contraintes propre à l'instance de l'utilisateur. Pour cela, nous générons toutes les substitutions du corps G possible tel qu'il soit satisfait (ligne 8). Étant donné une substitution σ, le corps est satisfait si chaque atome $p(t_1, \ldots, t_k)$ de G est satisfait avec σ. L'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 intentionnellement, si $\sigma(p(t_1, \ldots, t_k))$ est valide en respect de la définition. Un problème se pose dans le cas d'un p intentionnel. Que se passe-t-il quand parmi les entrées de p, il y a une variable de CSP ? Dans ce cas, nous générons des variables de CSP auxiliaires en sorties. Par exemple, considérons l'atome suivant $sum(X, Y, Z)$ et la substitution $\{X/2, Y/v_1, Z/?\}$ avec pour domaine de la variable v_1 : [2..6]. Dans ce cas, la substitution est complétée avec Z/v_2 avec comme domaine pour v_2 : $[-\infty.. + \infty]$. Une fois ces substitutions calculées, l'algorithme va substituer la tête de la règle pour produire les contraintes (ligne 21). Les contraintes produites sont alors des disjonctions de contraintes. Cependant, si on prend l'exemple des emplois du temps, les variables d'un certain nombre de contraintes seront déjà fixées. Par exemple, prenons 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. La contrainte produite par cette substitution est alors $Mrs\ Green \neq Mrs\ Green \vee Latin = English \vee v_2 \neq v_4$. Elle peut être immédiatement simplifiée en $v_2 \neq v_4$. Ce n'est cependant pas le cas à chaque fois ; si cette substitution est appliquée à la première règle, il restera une disjonction.

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

```
1.  //Complete the partial extension
2.  // with CSP variables with domains
3.  ext ← COMPLETE(ext, domains)
4.  //Initialize the variables set
5.  //with these in ext
6.  vars ← GETVAR(ext)
7.  constraints ← ∅
8.  for each G → C ∈ CS
9.  // Generate all substitution of the body
10.    subst ← GENERATEALLSUBST(G, ext)
11.    for each σ ∈ subst
12. //If there are atoms with no matching
13. // in ext,it adds aux. variables
14. // and the constraint
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. //It adds the constraint
20. //corresponding to the head
21.       constraint.add(σ(C))
22. //Finally, it returns the CSP
23. return CSP(vars, domains, constraints)
```

FIGURE 3 – Traduction d'un CPS en CSP

2.2.1 Tâche d'apprentissage

Dans cette partie, nous décrivons la tâche d'apprentissage consistant à induire à partir de solutions et non-solutions d'un problème, un CPS. Le choix de notre langage était motivé par l'utilisation des techniques de programmation logique inductive (ILP). Nous présentons donc cette tâche comme un problème d'apprentissage relationnel classique.

Le but est donc d'apprendre la définition d'un CPS, noté CS qui discriminera correctement les solutions E^+ des non-solutions E^-. On dira que CS couvre un exemple e en respect avec une base de connaissance $\mathcal{B}$, noté $(e \cup \mathcal{B}) \models CS$, si pour toute substitution σ des variables de CS vers les constants de e, $\sigma(CS)$ est vrai. Un exemple est rejeté s'il n'est pas couvert. Une définition est satisfaisante si elle couvre tous les exemples de E^+ et rejette ceux de E^-. Pour résumer, nous pouvons formaliser notre problème d'apprentissage de la manière suivante : étant donnés deux ensembles E^+ et E^-, et une base de connaissance $\mathcal{B}$, trouver une définition pour CS telle que :

- $\forall e^+ \in E^+ : (e^+ \cup \mathcal{B}) \models CS$
- $\forall e^- \in E^- : (e^- \cup \mathcal{B}) \not\models CS$

Le cadre traditionnel d'apprentissage en ILP consiste à apprendre une définition composée de clauses. Un exemple est alors couvert s'il existe une substitution couvrant au moins une règle. Cette définition correspond à des formes normales disjonctives (DNF). Or, un CPS est en forme normale conjonctive. En passant au concept négatif, c'est à dire en cherchant la négation de CS rejetant les exemples de E^+ et couvrant ceux de E^- (autrement dit, en inversant les ensembles positifs et négatifs), on se ramène à l'apprentissage d'une définition en DNF et ainsi, pouvoir utilisé les techniques de l'état de l'art. Par manque de place, nous ne détaillerons pas cette transformation dans ce papier. Le lecteur intéressé pourra consulter [13]. Ainsi, dans la suite de ce papier, nous chercherons à apprendre des définitions pour la négation du problème cible. Le CPS pourra être obtenu en prenant la négation de la définition apprise.

Nous nous intéressons dans ce papier à la famille d'algorithme *separate-and-conquer*[11]. Ces algorithmes cherchent à apprendre un ensemble de règles jusqu'à ce que tous les exemples soient discriminés. Dans la suite, nous ne nous intéressons plus qu'à l'apprentissage d'une règle de la définition. Les caractéristiques principales d'un système de recherche d'une règle sont données par la définition d'un espace de recherche, un opérateur de raffinement et une heuristique capable de choisir, à chaque étape, le meilleur candidat parmi les raffinements possibles. Dans ce contexte, l'opérateur de raffinement joue un rôle essentiel et les approches sont usuellement séparées en deux catégories : d'une part, les recherches *top-down* qui commencent avec une clause générale et qui construisent des spécialisations et de l'autre, les recherches *bottom-up* qui commencent avec une clause spécifique et puis la généralisent. Dans la plupart de ces cas, l'opérateur de raffinement permet d'organiser l'espace de recherche comme un treillis, commençant soit par une clause générale nommée *top* ($\top$) soit par une spécifique nommée *bottom* ($\bot$).

3 Saturation Minimale

Nous avons observé que la majorité des systèmes d'ILP classiques échouait sur l'apprentissage de nos CPS ou finissait par trouver une solution après une recherche exhaustive. Ces échecs peuvent être expliqués par les difficultés contenues dans les problèmes considérés qui sont bien connus dans la communauté ILP. De récents travaux sur la transition de phase en ILP[17] et sur la recherche aveugle ou la traversée de "plateau"[1] montrent de réelles difficultés sur certaines classes de problèmes. C'est notamment le cas pour l'apprentissage de CPS. Dans la suite de notre article, nous proposons une recherche bidirectionnelle où l'idée principale est de réduire progressivement l'espace de recherche. À chaque étape, notre espace de recherche est défini par un couple d'hypothèses $(H_{\top_i}, H_{\bot_i})$ limitant l'espace. Notre opérateur de raffinement produit de nouvelles limites $(H_{\top_{i+1}}, H_{\bot_{i+1}})$ où $H_{\top_{i+1}}$ est plus spécifique que $H_{\top_i}$ et $H_{\bot_{i+1}}$ est plus générale que $H_{\bot_i}$. La recherche s'arrête quand $H_{\top_i} = H_{\bot_i}$ qui correspond à la règle apprise.

Notre algorithme est basé sur la structure de la saturation, une opération usuelle en ILP. La saturation est un ensemble de littéraux clos apportant un maximum d'information sur un exemple. Elle est organisée en plusieurs couches ordonnées. Un littéraux appartient à une couche k si les termes nécessaires à son introduction ont été introduits par des littéraux des couches précédentes. Nous commençons par formaliser cet objet avant de présenter notre algorithme de recherche de règle.

3.1 Saturation

Avant de définir la saturation, nous introduisons quelques notations : étant donné un littéral l, nous notons $input(l)$ et $output(l)$ les ensembles des termes associés respectivement aux entrées et aux sorties du littéral l. Nous utilisons la même notation pour une formule f en notant $input(f) = \bigcup_{l \in f} input(l)$

et $output(f) = \bigcup_{l \in f} output(l)$. De plus, l'ensemble des termes apparaissant dans un littéral l est noté $terms(l)$. $vars(f)$ est une opération qui consiste à substituer toutes les constantes d'une formules f par des variables telles que les constantes sont remplacées par des variables distinctes. Comme indiqué précédemment, la saturation d'un exemple est un ensemble de littéraux qui peut être structuré en niveaux : nous notons $layer(l)$ le niveau dans lequel apparaît le littéral k [1]. L'ensemble des littéraux appartenant au niveau k est noté $litsOfLayer(k)$ et est défini par : $litsOfLayer(k) = \{l \mid layer(l) = k\}$. Enfin, étant donnés un ensemble d'exemples positifs et un ensemble d'exemples négatifs, nous notons $p(f)$ et $n(f)$ le nombre d'exemples respectivement **p**ositifs et **n**égatifs couverts par la formule f.

Saturation d'un exemple Dans cette partie, nous présentons une formalisation de l'opérateur de saturation, consistant à collecter tous les littéraux clos qui décrivent un exemple ou qui sont liés à sa description. Cette construction sera illustrée sur l'exemple suivant, inspiré du classique exemple des trains de Michalski :

Exemple

Dans cet exemple, chaque train est composé d'un certain nombre de wagons, chaque wagon est décrit par son nombre de roues et sa taille. De plus chaque wagon transporte un certain nombre d'objets de différentes formes géométriques. La situation suivante décrit deux trains correspondant respectivement à un exemple positif et un exemple négatif du même concept cible :

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

$t_2^- : \{has\_car(t_2^-, c_1), has\_car(t_2^-, c_2), has\_car(t_2^-, c_3),$
$wheels(c_1, 1), wheels(c_2, 4), wheels(c_3, 3),$
$long(c_1), long(c_2), short(c_3), load(c_1, circle, 4),$
$load(c_1, rectangle, 2), load(c_2, rectangle, 5)$
$load(c_2, circle, 6), load(c_3, circle, 2)\}$

Nous considérons ici que les prédicats utilisés sont associés aux modes suivants : $has\_car(+,-)$, $shape(+,-)$, $wheels(+,-)$ et $load(+,-,-)$. Si on suppose que la définition cible caractérise les trains ayant au moins deux wagons transportant des objets de même forme et que l'un des deux a à la fois plus de roues et plus d'objets de la forme commune, alors une définition possible sous forme de clause est :

$goodtrain(T) : -has\_car(T, C_1), has\_car(T, C_2),$
$C_1 \neq C_2, wheels(C_1, W_1), wheels(C_2, W_2),$
$W_1 < W_2, load(C_1, O, L_1), load(C_2, O, L_2),$
$L_1 < L_2$

La saturation d'un exemple est obtenue en ajoutant autant de littéraux clos que possible à partir de la base de connaissance, en imposant que tous les littéraux soient connectés à l'exemple. La saturation est construite à partir des informations sur les domaines et sur les modes des prédicats. Il s'agit d'un ensemble de littéraux qui est structuré en niveaux : un littéral l appartient au niveau k si ses termes correspondant à des positions d'entrée ($input(l)$) apparaissent dans des littéraux de niveaux inférieurs et que au moins l'un de ces littéraux appartient au niveau $k-1$. Cette opération peut produire un ensemble infini, par conséquent elle est paramétrée par une profondeur maximale, noté i, correspondant au niveau maximum considéré.

Pour définir la saturation, nous caractérisons les différents niveaux qui la composent : le niveau k de saturation, noté $sat_k(S_l)$, est construit récursivement à partir de l'ensemble S_l de tous les littéraux des niveaux inférieurs à k :

$$sat_k(S_l) = \{l \mid input(l) \subseteq terms S_l \\ \wedge\ input(l) \cap output(litsOfLayer(k-1)) \neq \emptyset\}$$

Nous pouvons maintenant définir l'ensemble de tous les littéraux de niveau supérieur ou égal à k (mais bornés par i) à partir de l'ensemble S_l des littéraux introduits dans les niveaux précédents. Nous notons $sat(S_l, k, i)$ cet ensemble, il est défini par :

$$sat(S_l, k, i) = sat_k(S_l) \cup sat(S_l, k+1, i) \quad k \leq i$$
$$sat(S_l, k, i) = \emptyset \quad k > i$$

Finalement, la saturation de l'exemple e pour le concept cible p, avec une profondeur maximum i peut être écrite :

$$sat(p(e), i) = sat(\{p(e)\}, 1, i)$$

Exemple (cont.)

Nous considérons que nous disposons d'une connaissance du domaine décrivant les prédicats $\neq$ et $<$. Le prédicat $\neq$ permet de comparer d'une part les wagons et d'autre part les formes géométriques, ce prédicat a pour mode $\neq (+,+)$. Le prédicat $<$ permet de comparer les nombres de roues ou les nombres d'objets, il a pour mode $<(+,+)$. La saturation de l'exemple t_1^+ avec une profondeur maximale $i = 3$ est organisée en niveaux de la manière suivante :

1. La notion de niveau est relative à un exemple et sa saturation. Puisque la construction d'une clause est réalisée à l'aide d'un exemple graine, dans la suite, pour chaque notion qui est liée à un exemple ou sa saturation, il s'agira implicitement de l'exemple graine ou de sa saturation afin de ne pas alourdir les notations.

Niveau	*Littéraux*
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	$2 < 3, 2 < 4, triangle \neq circle,$ $3 < 5, 3 < 10, 2 < 5, 2 < 10, 5 < 10$

3.2 Apprentissage de règles par saturation minimale

Dans cette section, nous présentons notre méthode d'apprentissage. À chaque étape de la recherche, la clause en construction est représentée par un ensemble de littéraux. L'espace de recherche représentant l'ensemble des clauses qui peuvent être apprises est organisé en un treillis borné par deux clauses notées $\top$ et $\bot$, et muni d'une relation d'ordre partiel permettant de comparer la généralité entre 2 clauses. L'ordre partiel considéré ici est l'inclusion, une clause s_1 est plus générale qu'une clause s_2 si s_1 est un sous-ensemble de s_2. La recherche est guidée par un exemple positif appelé "graine" et noté s dans la suite ; en notant p le prédicat cible, nous définissons $\top = \{p(s)\}$ et $\bot = sat(\top, i)$ où i est le niveau maximum de saturation. Cet espace de recherche est usuellement utilisé dans les algorithmes de l'état de l'art. En supposant que cette clause $\bot$ rejette tous les exemples négatifs (elle couvre au moins un exemple positif, la graine), l'espace de recherche contient ainsi au moins une règle discriminante, rejetant les exemples négatifs. Notre objectif est de découvrir une règle qui soit meilleure que $\bot$, dans le sens où elle couvre un maximum d'exemples positifs, tout en rejetant les exemples négatifs.

Notre algorithme d'apprentissage est basé sur une recherche bi-directionnelle qui raffine progressivement les hypothèses $H_\top$ et $H_\bot$ en conservant à chaque étape la relation $H_\bot = sat(H_\top, i)$.

Cette section est organisée en deux parties : nous présentons d'abord notre opérateur de raffinement puis l'algorithme d'apprentissage, basé sur une itération d'étapes de raffinement.

Opérateur de raffinement Cet opérateur recherche, dans une couche k, un ensemble de littéraux à ajouter à la clause en construction, $H_\top$, telle que la clause saturée correspondant $H_\bot$ discrimine les exemples positifs des négatifs. Pour déterminer cet ensemble de littéraux, nous recherchons un ensemble minimal par une recherche en largeur.

Afin de formaliser l'opérateur de raffinement, nous commençons par définir l'ensemble des couples candidats $(H'_\top, H'_\bot)$ obtenus par raffinements à partir de l'hypothèse en cours de construction $(H_\top, H_\bot)$. Cet ensemble de candidats est noté $candidates(H_\top, H_\bot, k, i)$, où k représente la couche à l'intérieur de laquelle le raffinement est recherché et i correspond à la profondeur maximale :

$$\begin{aligned} candidates(H_\top, H_\bot, k, i) = \\ \{(H'_\top, H'_\bot) | H'_\top = H_\top \cup S_k \\ \wedge\ S_k \subseteq litsOfLayer(k) \\ \wedge\ H'_\bot = sat(H'_\top, k, i) \wedge n(H'_\bot) = 0\} \end{aligned}$$

Cet ensemble est constitué de couples $(H'_\top, H'_\bot)$ où $H'_\top$ est plus spécifique que $H_\top$ et $H'_\bot$ est plus général que $H_\bot$ et pour lesquels $H'_\bot$ rejette tous les exemples négatifs.

Pour choisir un raffinement parmi l'ensemble des candidats, nous sélectionnons celui pour lequel la taille de S_k est minimum et s'il existe plusieurs raffinements possibles, nous choisissons celui qui couvre le plus grand nombre d'exemples positifs. Le nombre de candidats est donc exponentiel avec la taille de $litsOfLayer(k)$. Afin de ne pas les générer tous, nous recherchons le plus petit par une recherche en largeur, l'algorithme de raffinement peut se résumer ainsi ;

Algorithm : REFINE$(H_\top, H_\bot, k, i)$

```
1.  //To search the smallest
2.  //subset of litsOfLayer(k)
3.  for j from 0 to |litsOfLayer(k)|
4.    Scandidates = ∅
5.  //to enumerate subset
6.  // of layer k of size j
7.    for each subset S_k of litsOfLayer(k)
8.     such that |S| = j
9.       H'_⊤ = H_⊤ ∪ S_k
10.        H'_⊥ = sat(H'_⊤, k, i)
11.       if n(H'_⊥) = 0 then
12.         Scandidates.add((H'_⊤, H'_⊥))
13. //if there exist at less one candidate,
14. //the algorithm returns one covering
15. //the maximum of positive examples
16.   if Scandidates ≠ ∅ then
17.     return argmax_{(H'_⊤,H'_⊥)∈Scandidates} p(H'_⊥)
18. //if none candidate is found,
19. //it returns the same pair
20. return (H_⊤, H_⊥)
```

Exemple (cont.)

Pour illustrer notre opérateur de raffinement, supposons que l'hypothèse en cours de construction soit donnée par le couple $(H_\top, H_\bot) = (vars(\{goodtrain(t_1^+)\}), vars(sat(\{goodtrain(t_1^+)\}, 3))$

et que le raffinement soit produit par $\text{REFINE}(H_\top, H_\bot, 1, 3)$.

L'algorithme commence par rechercher un ensemble de taille 1 dans le premier niveau. Tous les raffinements conduisent à un même couple hypothèse (a renommage de variables près) :

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

où $H'_\top$ *apparaît en gras et* $H'_\bot$ *contient tous les littéraux. Ce couple n'est pas acceptable puisque* $H'_\bot$ *couvre l'exemple négatif* t_2^-.

Si nous considérons maintenant les sous-ensembles de taille 2, nous obtenons les couples suivants (à renommage de variables près) :

Niveau	*Littéraux*
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	$VW2 < VW3, VL3 < VL5$

Niveau	*Littéraux*
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 définition rejette tous les exemples négatifs, par conséquent c'est le couple correspondant qui sera retourné par l'algorithme de raffinement.

Apprentissage d'une règle Une règle intéressante est une règle qui rejette tous les exemples négatifs et qui couvre un maximum d'exemples positifs. Dans notre recherche bi-directionnelle, nous construisons par raffinements successifs des couples d'hypothèses $(H_\top, H_\bot)$, qui convergent vers un couple vérifiant $H_\top = H_\bot$. Afin de l'obtenir, le raffinement est opéré niveau par niveau, en commençant par le niveau $k = 1$ et en finissant par $k = i$ où i est le niveau maximum. Nous initialisons cette recherche par un couple correspondant à un exemple graine et à sa saturation, où toutes les constantes sont remplacées par des variables. L'algorithme suivant décrit la recherche d'une règle définissant un concept cible p, où l'espace de recherche est construit à partir d'un exemple graine s.

Algorithm : $\text{LEARNRULE}(p, s, i)$

```
1. H⊤ = {p(s)}
2. H⊥ = vars(sat(p(s), i))
3. for each layer k from 1 to i
4.     (H⊤, H⊥) = REFINE(H⊤, H⊥, k, i)
5. return H⊤
```

4 Expériences

Pour illustrer l'efficacité de notre approche, nous avons produit des jeux de données sur des problèmes classiques. Pour cela, nous avons généré, pour les problèmes donnés en exemple à la section 2.1, un ensemble de solutions et de non-solutions. Pour produire un exemple positif, nous avons choisi des tailles aléatoires pour les différents ensembles (ex. pour la coloration de graphe, les nombres de sommets et de couleurs), puis résolu le CSP correspondant avec une heuristique aléatoire. Pour un exemple négatif, nous avons procédé d'une manière équivalente à la différence que les contraintes sont relâchées et que l'on s'assure qu'il y est au moins une contrainte non-satisfaite.

Pour évaluer notre méthode, nous nous sommes comparés aux seuls systèmes d'ILP classiques ayant réussi à trouver une définition. Seulement Propal (la version décrite dans [2], plus rapide que celle de [3]) et Aleph[19] ont réussi. Aleph est un système offrant de nombreuses possibilités de configurations. La première que nous avons considérée, que nous appellerons Aleph1, correspond à un parcours en largeur du treillis limité à l'exploration de 200 000 nœuds de recherche visités et à une liste ouverte infinie. La seconde, appelée Aleph2, diffère seulement sur la stratégie de recherche utilisée où nous avons considéré la recherche heuristique, où nous avons testé chaque heuristique implémenté dans Aleph avec des résultats similaires. Nous avons implémenté un prototype pour notre algorithme et la figure 4 présente les résultats obtenus avec ces systèmes. Pour calculer la précision du CPS appris (troisième colonne), nous avons généré de nouveaux exemples et calculé le rapport d'exemples correctement discriminés (couverts pour les exemples positifs, rejetés pour les négatifs) sur le nombre total d'exemples. Plus le concept cible est compliqué, plus Propal et Aleph2 trouvent des CPS incorrectes. Pour les n-reines, nous avons stoppé Propal après 10 heures de calcul. Cela illustre la difficulté que rencontre les approches top-down quand la recherche est aveugle. Aleph1 réussit sur tous les jeux de donnés. Cependant, il nécessite un important temps de calcul pour les jeux de données compliqués. Notre méthode réussi

sur tous les jeux de données : elle trouve des CPS précis en peu de temps. Même si le sens est identique, les règles apprises ne sont cependant pas toujours exactement celles attendus. Par exemple, pour les n-reines, la contrainte apprise pour la diagonale est :

$$position(Q1, X1, Y1) \wedge\ position(Q2, X2, Y2)$$
$$\wedge\ ecart(Y1, Y2, V1) \wedge\ ecart(Y2, Y2, V2)$$
$$\rightarrow\ Q1 = Q2 \vee\ V2 \neq V1$$

Notre prototype et nos jeux de données peuvent obtenus en contactant par e-mail les auteurs.

5 Conclusion

Nos travaux cherchent à éviter les limitations rencontrées avec CONACQ [4]. À notre connaissance, CONACQ et notre système sont les seules études concernant l'acquisition de CSP. Dans [7], Bessière et al. propose une méthode pour automatiquement générer des points de vue à partir d'exemples. Cependant, aucune méthode n'est donnée pour générer les contraintes sur ces variables. Alors que l'acquisition n'est pas très étudiée, plusieurs travaux existent sur la découverte de contraintes impliquées ou rendondantes [8, 6]. Dans ce cas, la tâche d'apprentissage consiste à apprendre seulement à partir d'exemple positifs puisque la discrimination des solutions et non-solutions est déjà assurée par le modèle. Ces approches sont complémentaires à la notre et les CPS, et les CSP générés à partir d'eux, gagneraient à être reformulés pour être plus efficaces.

Dans ce papier, nous avons présenté un cadre pour obtenir automatiquement un modèle abstrait de CSP. Notre approche, basée sur la programmation logique inductive, nécessite des exemples que l'utilisateur considère comme des solutions et non-solutions de problèmes liés. Ensuite, il obtient une spécification (un CPS) qui peut être traduit par la suite en CSP en ajoutant les données de son véritable problème. Notre apport principal concerne la tâche d'apprentissage. Même si nos CPS sont écrits en logique du premier ordre, les systèmes classiques échouent lors de l'apprentissage. Les problèmes de recherche aveugle et de transition de phase rendent notre classe de problèmes difficiles à résoudre. Nous avons donc développé un nouvel algorithme de recherche bidirectionnel basé sur le raffinement progressif des bornes de l'espace de recherche. Les résultats sont très encourageants et ouvrent la perspective d'enrichir le langage pour permettre d'exprimer de nouveaux problèmes.

Références

[1] 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.

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

[3] 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.

[4] 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.

[5] Christian Bessière, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In Veloso [20], pages 50–55.

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

[7] 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.

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

[9] Remi Coletta, Christian Bessière, Barry O'Sullivan, Eugene C. Freuder, Sarah O'Connell, and Joël Quinqueton. Semi-automatic modeling by constraint acquisition. In Francesca Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 812–816. Springer, 2003.

[10] 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 [20], pages 80–87.

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

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

	Propal			Our algorithm		
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	4583.84	61.67%

FIGURE 4 – Expérience sur des problèmes classiques

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

[14] 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.

[15] 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.

[16] 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.

[17] Alessandro Serra, Attilio Giordana, and Lorenza Saitta. Learning on the phase transition edge. In *IJCAI*, pages 921–926, 2001.

[18] Barbara M. Smith. Modelling. In T. Walsh F. Rossi, P. van Beek, editor, *Handbook of Constraint Programming*, chapter 11, pages 377–406. Elsevier, 2006.

[19] Ashwin Srinivasan. *A learning engine for proposing hypotheses (Aleph).*

[20] Manuela M. Veloso, editor. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007.