Apprentissage de Contraintes Globales Implicites

Christian Bessière[1] **Rémi Coletta**[1] **Thierry Petit**[2]

[1] LIRMM (CNRS/Université Montpellier II),
161 Rue Ada, 34392 Montpellier Cedex 5, France.
[2] Ecole des Mines de Nantes,
LINA FRE CNRS 2729, 4 rue Alfred Kastler,
44307 Nantes Cedex 3, France.
bessiere@lirmm.fr coletta@lirmm.fr thierry.petit@emn.fr

Résumé

Il peut exister de nombreux modèles à contraintes exprimant un problème donné. Une des barrières s'opposant à une diffusion plus ample de la programmation par contraintes est l'expertise importante requise pour concevoir un modèle permettant une résolution efficace d'un problème. Aussi, l'intérêt de la communauté pour la reformulation automatique des modèles à contraintes est croissant. Cet article présente une approche alternative, consistant à *apprendre* des contraintes implicites d'après des instanciations des variables du problème, solutions et non-solutions. Ces contraintes sont alors ajoutées au modèle afin d'améliorer la résolution. Pour valider l'impact de cette approche, nous proposons un algorithme d'apprentissage de contraintes de cardinalité globale (Gcc). Nous montrons expérimentalement l'intérêt de cet algorithme pour améliorer la qualité des modèles à contraintes.

1 Introduction

La programmation par contraintes est un paradigme qui a prouvé son efficacité pour résoudre des problèmes dans des domaines variés : ordonnancement de tâches, configuration, planification, etc. L'utilisation de ce paradigme nécessite de fortes compétences, et, malheureusement, toutes les équipes de recherche et développement ne disposent pas d'experts en programmation par contraintes. C'est pourquoi l'intérêt de la communauté dans la reformulation automatique de modèles est croissant. Firsch et al. [7] ont proposé une technique de reformulation pure. Ils proposent un système qui traduit une spécification abstraite d'un problème en plusieurs modèles à contraintes différents. Colton et Miguel [5] exploitent un programme de génération de théories pour fournir des concepts et des théorèmes que l'on peut ensuite traduire en *contraintes implicites*. La notion de contrainte implicite est en l'occurrence une notion importante pour concevoir des modèles à contraintes efficaces. Une contrainte implicite n'est pas requise dans la formulation du problème mais elle peut en revanche être très utile pour le résoudre [16]. Par opposition on parle de contrainte *redondante* quand elle n'est pas requise pour la formulation du problème et qu'elle ne modifie pas la quantité de filtrage possible. Un bon exemple d'utilisation de contraintes implicites a été mis en évidence par Régin pour des problèmes d'ordonnancement de championnats sportifs [15].

Dans cet article, nous nous plaçons dans la situation suivante : nous disposons d'un modèle qui exprime correctement le problème, mais le temps de résolution obtenu n'est pas satisfaisant. Notre objectif est alors d'améliorer ce modèle en ajoutant automatiquement des contraintes implicites. Notre idée consiste à définir un algorithme d'apprentissage qui déduit de nouvelles contraintes à partir d'instanciations des variables fournies en donnée. Il ne s'agit donc pas de reformulation au sens habituel du terme. Notamment, notre technique peut permettre d'apprendre des contraintes qui ne sont pas déductibles par une analyse de la structure ou de la sémantique des contraintes. Elles peuvent, au contraire, dépendre fortement des domaines initiaux, et ainsi être différentes si on modifie même très légèrement ces domaines.

On notera que des techniques d'apprentissage ont déjà été appliquées à la programmation par contraintes, dans des contextes différents. Dans [4], un algorithme d'apprentissage est utilisé pour acquérir un modèle initial en inter-agissant avec un utilisateur. Dans [10], les auteurs traitent la tâche spécifique de l'acquisition de propagateurs de contraintes pour améliorer le temps de résolution.

Ici notre but est d'apprendre des contraintes implicites paramétrées. Par exemple, la contrainte `AtLeast`$(N, v, \mathcal{X})$ est satisfaite si et seulement si le nombre d'occurrences de la valeur v dans une instanciation complète de $\mathcal{X}$ est supérieur ou égal à N. N est le paramètre, qui peut être une variable, un intervalle, ou bien une simple valeur entière. Dans notre cadre, un algorithme d'apprentissage visera, par exemple, à apprendre l'intervalle de valeur pour N qui soit le plus restreint possible. Si l'algorithme retourne une borne inférieure pour cet intervalle qui est non nulle, et/ou une borne supérieure strictement inférieure à $|\mathcal{X}|$, alors la contrainte apprise peut être utile pour améliorer le processus de résolution.

En termes d'efficacité, une des facilités principales fournies par la programmation par contraintes est l'utilisation de contraintes globales [2]. Nous proposons dans la section 4 un algorithme d'apprentissage dédié à une contrainte globale très usuelle, la contrainte de cardinalité globale (`Gcc`). Une `Gcc` exprime une propriété très générique : elle contraint le nombre d'occurrences de chaque valeur présente dans les domaines. En d'autres termes, cette contrainte traite globalement une conjonction de `AtLeast` (imposant qu'une valeur v apparaisse au moins un certain nombre de fois) et `AtMost` (imposant qu'une valeur v apparaisse au plus un certain nombre de fois). A cause du concept très général qu'elle représente, cette contrainte est un bon candidat pour être une contrainte implicite utile dans des contextes variés.

Définition 1 *Dans une `Gcc(`$\mathcal{X}$`, `T`, lb, ub)`, $\mathcal{X}$ est un ensemble de variables, T un tableau de valeurs, et lb et ub deux tableaux d'entiers positifs en bijection avec les éléments de T, qui représentent les bornes inférieures et supérieures du nombre possible d'occurrences de chaque valeur de T. Etant donnée $v \in T$, lb_v (resp. ub_v) désigne l'entier associé à v dans lb (resp. dans ub). Cette contrainte impose que la valeur v dans T soit affectée dans $\mathcal{X}$ un nombre de fois appartenant à $[lb_v, ub_v]$.*

De nombreuses contributions ont été faites concernant la `Gcc` [14, 12, 11, 13], notamment des algorithmes de filtrage réalisant l'arc-consistance avec une complexité en temps raisonnable. La `Gcc` est disponible dans la plupart des résolveurs de contraintes existants. Etant donné un ensemble de variables, un ensemble de domaines et un ensemble de contraintes, l'algorithme que nous proposons dans cet article (section 4) apprend un ensemble de bornes lb, ub, telles qu'ajouter au modèle une `Gcc` avec ces bornes préserve l'ensemble des solutions du problème traité. Ce travail est le premier exemple d'apprentissage d'une contrainte dépendante d'un ensemble de paramètres. La section 5 montre le gain obtenu en utilisant notre algorithme pour améliorer des modèles à contraintes.

2 Contraintes Implicites Paramétrées

Lorsqu'un problème a été modélisé sous la forme de variables et de contraintes, des propriétés implicites, c'est à dire non explicitement exprimées par des contraintes, peuvent être satisfaites par l'ensemble des solutions. Dans certains cas il peut être intéressant de les exprimer explicitement par de nouvelles contraintes ajoutées au problème. En effet, un des principes de base de la programmation par contraintes consiste à réduire l'espace de recherche en supprimant des valeurs non viables des domaines des variables, et les nouvelles contraintes que l'on exprime peuvent être munies d'algorithmes de filtrage des domaines efficaces, permettant de nouvelles réductions. Dans la littérature, de telles contraintes sont qualifiées de contraintes *implicites* [16, 15]. Dans cet article, nous nous intéressons aux contraintes dépendant d'un ou plusieurs paramètres. Les contraintes paramétrées expriment généralement une règle sur les valeurs affectées aux variables qu'elles impliquent. Le paramètre peut être une valeur, un intervalle, ou une variable. Certaines contraintes sont définies à l'aide d'une collection de paramètres. C'est le cas de la `Gcc`. On peut citer quelques autres exemples de contraintes paramétrées, extraites du catalogue de Beldiceanu [1] : `AtLeast`, `AtMost Change`, `Common`, `Count`, `Gcc`, `Max`, `Min`, `NValue`, etc. Toutes ces contraintes impliquent un ensemble de variables de taille quelconque. Dans le manuel utilisateur de la librairie de programmation par contraintes Ilog Solver 5.0. [8], p. 279, le concept de contrainte implicite est mis en évidence en ajoutant une `Gcc` afin d'améliorer la résolution d'un problème de coloration de graphes.

Considérons un autre exemple. m tâches doivent être ordonnancées. Chacune des tâches a la même durée d et l'ensemble des tâches est partitionné en deux : m_1 tâches requièrent 1 ressource, $m_2 = m - m_1$ tâches requièrent 2 ressources. Le temps est discrétisé par une échelle d'unités commençant au point 0. A un point donné de temps, la quantité maximale de ressource que l'on peut utiliser est bornée par $maxi$. Les tâches doivent commencer en étant séparées les unes des autres par au moins deux unités de temps. En outre, des contraintes de précédence sur le début des tâches sont ajoutées de façon aléatoire. Etant donné un *makespan* (dernier point de l'échelle de temps) égal à $2*(m-1)+d$ (le meilleur réalisable étant données les contraintes sur le début des tâches), la question est de trouver un ordonnancement réalisable, si il existe. Nous avons implémenté en Choco [3] un modèle naïf dans lequel le début de chaque tâche est représenté par une variable, et les ressources sont représentées à chaque point de temps par un tableau de variables, où chaque index correspond à la consommation d'une tâche donnée à ce point de temps. Les contraintes sur la durée et la séparation des tâches sont primitives, et des contraintes de somme sont utilisées pour respecter la borne sur la quantité maximale de ressource. Nous nous plaçons dans le contexte d'un utilisateur non expert. Donc, la stratégie de recherche employée est celle fournie par défaut par Choco (d'abord la variable ayant le plus petit domaine, et d'abord la plus petite valeur du domaine). La recherche est arrêtée si aucun ordonnancement n'est produit après un temps de résolution de 60 secondes. Même avec $m=4$ et $d=4$, et deux contraintes de précédence, ce modèle donne de mauvais résultats (voir la table 1).

m_1/m_2	$maxi$	#nœuds	temps (sec.)
4/0	4	—	>60
4/0	3	42 129	7.4
4/0	2	85	0.12
3/1	4	45	0.030
3/1	3	33	0.041
3/1	2	7 (pas de sol)	0.030

Tab. 1 – Résultats avec un modèle naïf.

Un utilisateur non-expert pourrait être très surpris par ces résultats. Notamment, en contraignant davantage $maxi$, la solution semble plus facile à trouver. L'explication est que les contraintes de ressource propagent davantage dans ce cas là et compensent l'inefficacité de la propagation des contraintes d'ordonnancement des tâches.

A ce stade, il est donc nécessaire d'améliorer le modèle pour le rendre plus robuste du point de vue de l'efficacité de la résolution. Une façon de faire peut être de regarder si les contraintes sur l'ordre des tâches n'induisent pas des limites sur le nombre de tâches à chaque point de temps de l'échelle[1]. La valeur $p=2(m-1)+d$ est exactement la valeur minimale possible pour ordonnancer toutes les tâches si elles sont séparées par au moins deux unités. C'est pourquoi les tâches doivent être placées aussi tôt que possible. C'est à dire une tâche au temps 0, une tâche au temps 2, etc. Il est donc possible d'ajouter une contrainte $\texttt{Gcc}(\mathcal{X}, T, lb, ub)$, où $\mathcal{X}$ est l'ensemble des variables représentant les tâches, T représente les différents points de temps, et lb et ub les bornes sur le nombre d'occurrences dans $\mathcal{X}$ des points de temps, définies de la manière suivante : pour chaque valeur $v \in T$, $ub_v = maxi$; si v est paire et v et plus petite que $2m-1$ alors $lb_v = 1$ (i.e., une tâche doit démarrer ici), sinon $lb_v = 0$. D'autres règles sur les cardinalités pourraient être déduites, par exemple pas de tâche démarrant à des points impairs, (i.e., $ub_v = 0$ si v est impair). Cependant, la `Gcc` additionnelle restreignant seulement les lb_v est suffisante pour obtenir une amélioration significative du processus de résolution (voir la table 2).

m_1/m_2	$maxi$	#nœuds	temps (sec.)
4/0	4	46	0.04
4/0	3	46	0.04
4/0	2	25	0.04
3/1	4	45	0.05
3/1	3	32	0.04
3/1	2	3 (pas de sol)	0.03

Tab. 2 – Résultats avec une `Gcc` ajoutée au modèle.

Cet exemple montre que pour un utilisateur non-expert en programmation par contraintes, même de très petits problèmes peuvent être difficiles à résoudre. Il montre aussi que ce défaut peut être pallié en ajoutant des contraintes implicites, qui rendent le modèle robuste. C'est pourquoi de nouvelles méthodes capables de déduire *automatiquement* de telles contraintes implicites sont les bienvenues.

3 Apprentissage de Contraintes Implicites Paramétrées

La question que l'on se pose est : comment apprendre une contrainte paramétrée ? Le principe de base est que l'on va exploiter l'information fournie par des instances positives et négatives, afin d'apprendre les paramètres. Une instance positive est une instanciation solution, tandis qu'une instance négative est une instanciation qui n'est pas une solution. Par conséquent, si le modèle que l'on utilise pour exprimer le problème n'est pas efficace en terme de résolution, celui-ci ne permettra pas de fournir facilement des instances positives. Cette section est une discussion sur l'ensemble de variables et les contraintes que l'on doit considérer dans le processus d'apprentissage.

[1] Une bonne heuristique de recherche pourrait aussi être utilisée, mais la génération automatique d'heuristiques robustes de recherche sort du cadre de ce papier.

3.1 Découpage du problème

Apprendre une contrainte sur l'ensemble des variables du problème n'est pas nécessairement une bonne idée. Il semble plus pertinent de découper le problème initial en sous-problèmes, chacun d'entre eux relatif à un sous-ensemble de variables et aux contraintes du problème impliquant ces variables. Un tel sous-problème aura plus de chances d'être résolu rapidement, et pourra ainsi produire facilement des instances utiles pour apprendre une contrainte implicite sur le sous-ensemble de variables. Par définition, si une contrainte apprise est valide pour un sous-problème, elle le sera aussi pour le problème complet : les variables et les contraintes du sous-problème sont des sous-ensembles de ceux problème initial. Généralement, plus petits sont les sous-problèmes, plus rapide est leur résolution. La contrepartie est que la contrainte apprise sera moins globale en terme de filtrage. En outre, pour des problèmes dépendant directement d'une certaine taille n, une autre approche pourrait consister à apprendre une contrainte sur un problème avec n petit et en déduire la contrainte implicite correspondante pour une taille n plus grande. Il faudra cependant réaliser cette déduction à la main car à priori il n'existe aucun processus automatique qui puisse garantir la validité de la contrainte ajoutée sur le problème de plus grande taille.

3.2 Relaxation du problème

Il est possible d'apprendre des contraintes implicites sur des relaxations du problème initial. Ce cas est en fait proche du précédent. On pourrait dire que l'on découpe le problème en fonction des contraintes. Il est naturellement aussi possible de considérer des versions relaxées de sous-problèmes. On notera par exemple que la contrainte `Gcc` présentée dans l'exemple d'introduction (section 2) est uniquement relative aux variables représentant les tâches, et n'a pas de rapport avec les contraintes de ressource. Nous verrons dans la section 5 que nous pouvons apprendre une telle contrainte sur un problème où les contraintes de ressource sont ignorées.

3.3 Problèmes d'Optimisation

Dans le contexte d'un algorithme de type Branch and Bound, une façon de procéder consiste à apprendre de nouvelles contraintes implicites à chaque pas de l'optimisation. Elles resteront par définition valides pour de meilleurs valeurs d'objectif (i.e., correspondant au même problème avec une contrainte plus forte sur l'objectif). L'aspect incrémental peut alors être intéressant, car il peut être aisé de trouver une solution avec une valeur d'objectif éloignée de l'optimal. Si la contrainte apprise améliore le modèle d'une façon significative, la prochaine solution, avec une meilleure valeur d'objectif, peut être déduite plus rapidement, et ainsi de suite si on peut à chaque étape déduire une nouvelle contrainte implicite. On peut ainsi observer une coopération intéressante entre le processus d'apprentissage et le processus de résolution.

4 Algorithme d'Apprentissage d'une Contrainte de Cardinalité Globale

Dans cette section nous supposons que l'ensemble des variables, leurs domaines, et l'ensemble des contraintes du modèle initial qui seront utilisés pour apprendre la contrainte `Gcc` sont connus.

Notation 1 *$P = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ désigne le problème utilisé pour apprendre une `Gcc` implicite sur $\mathcal{X}$.*

Dans toute cette section nous supposons également que P admet au moins une solution (le problème n'est pas sur-contraint). Notre but est d'apprendre un ensemble d'intervalles où chaque borne inférieure et supérieure de l'intervalle correspond aux cardinalités maximum et minimum d'une valeur dans n'importe quelle instanciation de $\mathcal{X}$ qui soit une solution. En d'autres termes, le but est d'apprendre la $\texttt{Gcc}(\mathcal{X}, T, lb, ub)$ la plus restreinte possible, où T est l'ensemble des valeurs apparaissant dans les domaines de $\mathcal{X}$ et lb et ub sont les bornes des intervalles appris, telles que l'ensemble des solutions de P soit le même en ajoutant ou sans ajouter cette contrainte au modèle. "Plus restreinte" signifie ici que nous essayons d'apprendre les intervalles les plus réduits possible.

4.1 Principe Général

Notation 2 *e désigne une instanciation de $\mathcal{X}$. $\texttt{occ}(v, e)$ désigne le nombre de fois où la valeur v apparaît dans e.*

Etant donnée une instanciation e de $\mathcal{X}$, la `Gcc` implicite que nous cherchons est satisfaite par e si et seulement si pour toute valeur $v \in T$ on a $lb_v \leq \texttt{occ}(v, e) \leq ub_v$. Au début de la phase d'apprentissage on sait seulement que $0 \leq lb_v \leq |\mathcal{X}|$, $0 \leq ub_v \leq |\mathcal{X}|$, et $lb_v \leq ub_v$. Le but est alors d'exploiter au mieux l'information fournie par les bornes afin de réduire le plus possible ces intervalles.

Soit e_j^- une instance négative. La première idée est de tester pour chaque $v \in e_j^-$ si $\texttt{occ}(v, e_j^-)$ est une cause de l'échec. En gros, si P n'a pas de solution quand le nombre maximal d'occurrences de v est $\texttt{occ}(v, e_j^-)$ alors on peut affecter lb_v à $\texttt{occ}(v, e_j^-) + 1$.

En effet, il a été appris qu'au moins une occurrence supplémentaire de v est requise. Et de façon symétrique, pour ub_v : si P n'a pas de solution en imposant que le nombre maximum d'occurrences de v soit $\texttt{occ}(v, e_j^-)$ alors on peut mettre à jour ub_v en lui affectant la valeur $\texttt{occ}(v, e_j^-) - 1$. Cependant, en pratique, un tel schéma d'apprentissage est trop coûteux. Pour tester si il existe des solutions avec une certaine limite sur le nombre d'occurrences d'une valeur, un problème doit être résolu. On souhaite éviter de faire cela systématiquement. Il est possible d'éviter de nombreuses résolutions en exploitant l'information produite d'après :

- les instances positives,
- le nombre d'occurrences des valeurs dans les instances négatives telles que les problèmes augmentés avec une limite sur ces occurrences ont des solutions.

Dans cette optique il est nécessaire de stocker pour chaque valeur v les intervalles courants possibles pour lb_v et pour ub_v.

Notation 3 *$min(lb_v)$ désigne la valeur minimale possible pour lb_v et $max(lb_v)$ sa valeur maximale possible. On définit de façon similaire $min(ub_v)$ et $max(ub_v)$.*

Le but sera pour chaque v d'augmenter successivement $min(lb_v)$ et de diminuer $max(ub_v)$, et ensuite de propager le résultat afin de mettre à jour lb et ub. Les deux autres quantités $max(lb_v)$ et $min(ub_v)$ seront utiles pour améliorer l'efficacité de la technique d'apprentissage.

4.2 Réduction des bornes d'après les instances

Considérons un exemple dans lequel on souhaite apprendre une $\texttt{Gcc}(\mathcal{X}, T, lb, ub)$ où $\mathcal{X} = \{x_1, x_2, x_3\}$ et $D(x_i) = \{a, b, c\}$, $\forall i \in \{1, 2, 3\}$. Considérons la valeur a. Initialement, lb_a peut être n'importe quel entier dans $\{0, 1, 2, 3\}$, car $\mathcal{X}$ contient trois variables. Similairement $ub_a \in \{0, 1, 2, 3\}$.

Soit $e_1^+ = [(x_1, a), (x_2, b), (x_3, a)]$ une instance positive. ub_a, la borne supérieure sur le nombre de a, ne peut pas être inférieure à $2 = \texttt{occ}(a, e_1^+)$ car e_1^+ est positive. Symétriquement, lb_a ne peut pas être plus grande que 2. En revanche une instance positive n'apporte aucune information concernant la valeur maximale de ub_a et la valeur minimale de lb_a.

Propriété 1 *Soit e_j^+ une instance positive. On a :*

$$max(lb_v) \leq \texttt{occ}(v, e_j^+)$$
$$min(ub_v) \geq \texttt{occ}(v, e_j^+)$$

Proof : e_j^+ est positive On ne peut pas restreindre les cardinalités des valeurs davantage que leurs occurrences dans e_j^+. □

Grâce à e_1^+ et la propriété 1, on peut réduire les possibilités pour lb et ub (voir table 3). Soit à présent

lb_a	ub_a	lb_b	ub_b	lb_c	ub_c
0	–	0	–	**0**	**0**
1	–	**1**	**1**	–	1
2	**2**	–	2	–	2
–	3	–	3	–	3

TAB. 3 – Déductions sur lb et ub obtenues à partir de l'instance $e_1^+ = [(x_1, a), (x_2, b), (x_3, a)]$.

l'instance négative $e_2^- = [(x_1, a), (x_2, b), (x_3, c)]$. On a $|T| * 2 = 6$ causes possibles pour sa réfutation : soit il y a trop de variables instanciées avec la valeur a, soit pas assez, et similairement pour b et c. L'information supplémentaire que l'on a est les valeurs possibles des bornes supérieures et inférieures des cardinalités déjà apprises à l'aide de e_1^+ (voir la table 3). Par exemple, considérons la valeur a. Il n'y a pas trop d'occurrences de a dans e_2^- car $min(ub_a)$ est égal à 2, et donc plus grand que $\texttt{occ}(a, e_2^-)$. Autrement dit, on sait que 2 n'est pas trop et par conséquent 1 ne peut pas être trop. Il n'y a peut être pas assez d'occurrences de a car lb_a pourrait finalement être une valeur strictement supérieure que $\texttt{occ}(a, e_2^-) = 1$, comme le montre la table 3.

Corollaire 1 *Soit e_j^- une instance négative.*
Si $min(ub_v) \geq \texttt{occ}(v, e_j^-)$ alors il n'y a pas trop de v dans e_j^-, sinon il y en a peut être trop.
Si $max(lb_v) \leq \texttt{occ}(v, e_j^-)$ alors il y a suffisamment de v dans e_j^-, sinon il y en a peut être pas assez.

Ce corollaire montre que bien que e_j^- soit une instance négative, il peut être très utile de prendre en compte toutes les instances positives précédentes dans le processus d'apprentissage. A cause des instances positives et de leur effet sur les minimums et maximums des bornes inférieures et des bornes supérieures, seulement un sous-ensemble de valeurs de e_j^- peuvent effectivement être la cause de la réfutation de e_j^-. Si ce sous-ensemble est vide alors rien ne peut être appris de cette instance. La table 4 montre l'application du corollaire 1 à e_2^-. Elle fournit les informations heuristiques pour savoir ce qui doit être testé afin de déterminer une raison possible de réfutation de e_2^-. Deux causes sont possibles mais aucune n'est garantie comme étant valide. Une dernière étape est donc nécessaire. Si toutes les causes de réfutation sont rejetées, cela signifiera simplement qu'il n'existe pas une $\texttt{Gcc}$ plus restreinte à apprendre à l'aide de cette instance. Dans le cas

Valeur	trop ?	pas assez ?
$\texttt{occ}(a, e_2^-) = 1$	Non	Peut être
$\texttt{occ}(b, e_2^-) = 1$	Non	Non
$\texttt{occ}(c, e_2^-) = 1$	Peut être	Non

TAB. 4 – Causes possibles de réfutation de $e_2^- = [(x_1, a), (x_2, b), (x_3, c)]$.

contraire on pourra mettre à jour les paramètres de la `Gcc`.

L'idée consiste à résoudre un CSP issu du CSP initial, dans lequel on ajoute de nouvelles contraintes de cardinalité pour tester chacune des causes possibles de réfutation. Par exemple, pour déterminer si il est possible qu'il n'y ait pas assez de a dans e_2^- (cf. table 4), on ajoute une contrainte imposant d'avoir *au plus* un a dans une solution. Si le CSP obtenu est insatisfiable, on sait que "pas assez de a" est une cause valide pour la réfutation de e_2^-. On doit avoir *au moins* deux a dans une solution de ce problème. Dans le cas contraire on sait qu'il est possible d'avoir un unique a (ou aucun) dans une instanciation solution.

Propriété 2 *Soit* $\texttt{occ}(v, e_j^-)$ *une cause possible de réfutation d'une instance négative* e_j^-*. Soit* Q *le problème "augmenté" obtenu en ajoutant la contrainte* $\texttt{AtMost(occ}(v, e_j^-), v, \mathcal{X}\texttt{)}$ *à* P*.*
Si Q *n'a pas de solution,* $min(lb_v) \geq \texttt{occ}(v, e_j^-) + 1$*, sinon* $max(lb_v) \leq \texttt{occ}(v, e_j^-)$*.*

Preuve : P a une ou plusieurs solutions. Si Q n'a pas de solution, cela signifie que toutes les solutions de P violent la contrainte $\texttt{AtMost(occ}(v, e_j^-), v, \mathcal{X}\texttt{)}$ ajoutée dans Q. Ainsi n'importe quelle solution de P a au moins $\texttt{occ}(v, e_j^-) + 1$ occurrences de v. Dans ce cas $min(lb_v)$ devrait être supérieur ou égal à $\texttt{occ}(v, e_j^-) + 1$. Si Q a une solution alors il n'existe pas de contrainte implicite dans P qui impose que le nombre d'occurrences de v soit plus grand que $\texttt{occ}(v, e_j^-)$. lb_v ne peut pas être supérieure ou égale à $\texttt{occ}(v, e_j^-)$. □

Propriété 3 *Soit* $\texttt{occ}(v, e_j^-)$ *une cause possible de réfutation d'une instance négative* e_j^-*. Soit* Q *le problème "augmenté" obtenu en ajoutant la contrainte* $\texttt{AtMost(occ}(v, e_j^-), v, \mathcal{X}\texttt{)}$ *à* P*.*
Si Q *n'a pas de solution,* $max(ub_v) \leq \texttt{occ}(v, e_j^-) - 1$*, sinon* $min(ub_v) \geq \texttt{occ}(v, e_j^-)$*.*

Preuve : Symétrique à la propriété 2. □

Ainsi, on imposera successivement des contraintes exprimant chaque cause possible de réfutation. Dans notre exemple, supposons que le problème a toujours des solutions avec la contrainte supplémentaire `AtMost(1,a,X)`. On sait alors que lb_a ne peut pas être supérieure ou égale à 2. On peut mettre à jour les valeurs possibles de lb_a (voir la table 5). On doit alors tester aussi un CSP augmenté avec la contrainte `AtLeast(1,c,X)`. (voir la table 4). Supposons qu'il n'y ait alors pas de solution. Cela signifie qu'une occurrence de c est trop, et on peut mettre à jour les valeurs possibles de ub_c comme le montre la table 5. On voit en comparant la table 5 à la table 3 que le fait

lb_a	ub_a	lb_b	ub_b	lb_c	ub_c
0	–	0	–	0	**0**
1	–	1	1	–	–
–	2	–	2	–	–
–	3	–	3	–	–

TAB. 5 – Déductions sur lb et ub obtenues à partir de l'instance $e_2^- = [(x_1, a), (x_2, b), (x_3, c)]$.

de traiter e_2^- a permis une restriction des paramètres de la `Gcc` que l'on veut apprendre. Ce processus peut être stoppé après n'importe quel nombre d'instances fournies, grâce à la propriété suivante :

Propriété 4 *Soit* E *un ensemble d'instanciations de* $\mathcal{X}$ *utilisées pour apprendre les bornes possibles des cardinalités d'une* `Gcc` *sur* $\mathcal{X}$*. Soit* T *l'ensemble des valeurs. La contrainte* $\texttt{Gcc}(\mathcal{X}, T, lb, ub)$ *où pour chaque* $v \in T$ *chaque* lb_v *est fixée à* $min(lb_v)$ *et chaque* ub_v *est affecté à* $max(ub_v)$ *est une contrainte implicite pour* P*.*

Preuve : Si P est utilisé pour déterminer pour chaque $e \in E$ si e est une solution ou non, alors cette propriété est vérifiée d'après les propriétés 1, 2 et 3. □

Quel que soit l'ordre choisi pour prendre en compte les instances dans E, la même `Gcc` sera apprise. En effet, le même ensemble de causes valides de réfutation sera identifié à partir des instances négatives. Cependant, le nombre de problèmes à résoudre pour obtenir le résultat dépend de l'ordre des instances.

On peut conclure que les propriétés 1, 2, 3 et le corollaire 1 fournissent une technique pour apprendre des contraintes `Gcc` implicites. Plus grand sera l'ensemble d'instances, plus restreinte sera la contrainte apprise, mais naturellement cela engendrera un coût en temps supérieur pour la phase d'apprentissage.

4.3 Accélération du Processus d'Apprentissage

La sous-section précédente propose une technique permettant d'apprendre les bornes des cardinalités de valeurs pour un problème donné. Pour apprendre ces bornes il est nécessaire de résoudre des sous-problèmes. Dans le but de rendre le processus d'apprentissage aussi rapide que possible, cette section présente des techniques simples améliorant le processus d'apprentissage.

Utilisation de la `Gcc` pour tester les causes de réfutation : Nous avons vu dans la sous-section 4.2 que chaque instance négative requiert de résoudre le problème P sur lequel on apprend la contrainte, en lui ajoutant une contrainte supplémentaire. Sachant que la propriété 4 garantit qu'à n'importe quelle étape de la phase d'apprentissage la contrainte `Gcc` est une contrainte implicite pour P, il est intéressant d'utiliser cette `Gcc` quand on teste une instance négative, à la place de simples contraintes ajoutées de type `AtLeast` ou `AtMost`. Ces contraintes peuvent être incluses dans la `Gcc` courante, ce qui offrira au moment du test une globalité de filtrage supplémentaire.

Limite de temps : il est possible d'éviter de tomber dans des cas pathologiques où la résolution d'un sous-problème particulier est trop longue en imposant une limite de temps sur chaque résolution. Si le résolveur de contraintes n'a pas déterminé si il existe une solution pour le problème augmenté considéré avant la limite, la recherche s'arrête et, simplement, rien n'est mis à jour dans la `Gcc`. On note qu'avec une limite de temps, la garantie d'apprendre la même contrainte en prenant en compte le même ensemble d'instances, mais dans des ordres différents, est perdue.

Application d'une consistance locale : au lieu de lancer une résolution complète à chaque test, jusqu'à obtenir soit une solution soit une preuve d'inconsistance, il est possible d'effectuer une "forme relâchée de résolution", en appliquant une consistance locale. De cette manière, si un échec est détecté (i.e., un domaine vidé), nous sommes sûrs que la cause de réfutation testée est valide. En revanche rien ne peut être déduit si il n'y a pas d'échec. Il s'agit à présent de déterminer le type de consistance locale le mieux adapté à notre problématique d'apprentissage.

La première idée est d'utiliser l'arc-consistance (AC). Malheureusement, en pratique, l'AC n'est généralement pas assez forte pour détecter une inconsistance. Dans notre contexte il est donc nécessaire d'étudier des consistances plus fortes. On est alors confronté au fait que les résolveurs de contraintes actuels ne permettent pas, en général, d'appliquer de façon générique une consistance plus forte que l'AC. Aussi, nous proposons de réaliser une consistance de singleton (SAC), que nous pouvons mettre en œuvre en appliquant l'AC sur des sous-problèmes particuliers (pour plus de détails sur la SAC nous invitons le lecteur à lire l'article de Debruyne et Bessière [6]).

Définition 2 (SAC) *Un réseau de contraintes* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *est Singleton Arc Consistant ssi :* $\forall X_i \in \mathcal{X}, \forall v_i \in D(X_i)$*, l'AC n'aboutit pas sur une inconsistance sur* $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{X_i = v_i\})$.

On peut facilement définir la k-singleton arc-consistance (k-SAC), qui est une consistance locale plus forte que la *SAC*.

Définition 3 (k-SAC) *Un réseau de contraintes* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *est k-SAC ssi :* $\forall \{X_{i_1}, \ldots, X_{i_k}\} \subseteq \mathcal{X}, \forall v_{i_1} \in D(X_{i_1}), \ldots, v_{i_k} \in D(X_{i_k})$*, l'AC n'aboutit pas sur une inconsistance sur* $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{X_{i_1} = v_{i_1}, \ldots, X_{i_k} = v_{i_k}\})$.

Enfin, nous introduisons une relaxation de la k-SAC qui sera particulièrement adaptée aux tests des causes de réfutation lors de l'apprentissage des bornes des cardinalités d'une `Gcc`.

Définition 4 (k-SAC(v)) *Un réseau de contraintes* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *est k-SAC(v) ssi :* $\forall X_i \in \mathcal{X}, v \in D(X_i), \exists \{X_{j_1} \cdots X_{j_{k-1}}\} \subseteq \mathcal{X}$ *tel que l'AC n'aboutit pas sur une inconsistance sur* $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{(X_{j_1} = v), \cdots, (X_{j_{k-1}} = v), (X_i = v)\}$

5 Résultats expérimentaux

5.1 Problème de Satisfaction

Cette section présente une série d'expérimentations de notre algorithme sur le problème décrit dans la section 2. Pour tous ces tests, l'algorithme d'apprentissage a été lancé sur un problème relâché de ses contraintes de ressource. La simplicité de ce problème relâché nous a permis d'avoir des temps d'apprentissage de seulement quelques milisecondes pour m=4 et m=5. Parmi les optimisations de la section 4.3 nous avons implémenté l'utilisation de la `Gcc` pour tester les causes de réfutation, et une procédure de limitation du temps de résolution des sous-problèmes. 15 instances on été utilisées pour apprendre chaque `Gcc`, et la limite de temps a été fixée à 0.4 secondes par instance. La table 6 montre les résultats obtenus quand le modèle décrit en section 2 est augmenté de la contrainte `Gcc` implicite apprise. Concernant la colonne de temps, la première donnée est le temps d'apprentissage et la deuxième est le temps de résolution du problème augmenté de la `Gcc`.

On voit que les résultats sont similaires à ceux de la table 2. Cela prouve que l'algorithme d'apprentissage nous fournit la robustesse requise pour la résolution de ce problème.

Les tables 7 et 8 montrent les résultats obtenus avec un nombre de contraintes de précédence aléatoires égal à 4, et $m = d = 5$. La recherche d'une solution du problème complet est stoppée lorsque le temps dépasse 60 secondes.

m_1/m_2	$maxi$	#nœuds	temps (sec.)
4/0	4	47	0.297 + 0.026
4/0	3	47	0.279 + 0.018
4/0	2	23	0.265 + 0.015
3/1	4	43	0.312 + 0.031
3/1	3	30	0.286 + 0.019
3/1	2	3 (pas de sol.)	0.279 + 0.013

TAB. 6 – Problèmes avec $m = 4$ en utilisant une contrainte `Gcc` implicite apprise.

m_1/m_2	$maxi$	#nœuds	temps (sec.)
5/0	5	–	> 60
5/0	4	–	> 60
5/0	3	–	> 60
5/0	2	53 250 (pas de sol.)	19
3/2	5	–	> 60
3/2	4	–	> 60
3/2	3	1 790 (pas de sol.)	1.2
1/4	5	11 065	3.9
1/4	4	7 985 (pas de sol.)	2.9

TAB. 7 – Problèmes avec $m = 5$ sans utilisation d'une `Gcc` implicite.

m_1/m_2	$maxi$	#nœuds	temps (sec.)
5/0	5	74	0.729 + 0.030
5/0	4	74	0.730 + 0.025
5/0	3	63	0.726 + 0.024
5/0	2	361 (pas de sol.)	0.721 + 0.243
3/2	5	63	0.734 + 0.025
3/2	4	92	0.717 + 0.059
3/2	3	123 (pas de sol.)	0.738 + 0.122
1/4	5	39	0.740 + 0.023
1/4	4	154 (pas de sol.)	0.718 + 0.133

TAB. 8 – Problèmes avec $m = 5$ en utilisant la contrainte implicite `Gcc` apprise.

Nous avons été capables de résoudre tous les problèmes avec $m = 6$ en quelques secondes (incluant le temps d'apprentissage) alors que la plupart d'entre eux n'était pas résolvables sans contrainte implicite, même après plusieurs minutes.

Notre objectif avec ce premier test était de tester un modèle naïf, avec une stratégie de recherche basique, et d'expérimenter combien notre technique pouvait améliorer sa robustesse. Naturellement ce benchmark est adapté à l'utilisation de `Gcc` implicites, notamment car le makespan est fixé à son minimum réalisable. Mais notre but ici est de fournir un système permettant à un utilisateur ayant l'intuition "il y a peut être des `Gcc` implicites à apprendre" de trouver automatiquement ces contraintes implicites. L'effort relatif est bien plus faible que celui d'étudier chaque combinaison des paramètres m et d, et les contraintes de précédence, pour essayer de déduire à la main ces contraintes de cardinalité en fonction de critères structurels ou sémantiques.

5.2 Problème d'Optimisation

Dans le cadre du cursus de l'IUT d'Informatique de Montpellier, les étudiants doivent réaliser un projet individuel. Chaque étudiant choisit son projet d'après un ensemble de sujets initialement proposés par les enseignants. Plus précisément, chaque étudiant doit fournir une liste $(p_1, \ldots, p_m)$ des m projets qu'il préfère. Leurs préférences doivent être totalement ordonnées : un projet p_i est strictement préféré à un projet p_{i+1}. Malheureusement, en pratique, les étudiants effectuent des choix souvent très similaires, et il arrive qu'il n'existe aucune solution telle que chacun des étudiants obtienne un projet qu'il a choisi[2].

Ainsi, l'objectif de ce problème est d'assigner des projets aux étudiants, de telle sorte que deux étudiants distincts aient deux projets distincts, tout en maximisant autant que possible la satisfaction des étudiants. Un étudiant est très satisfait si il obtient son premier choix, moins si il obtient son second choix, et ainsi de suite jusqu'au point où il ne peut obtenir aucun projet de sa liste, ce qui correspond à la satisfaction la plus faible possible. Jusqu'à l'année dernière, les enseignants effectuaient ces affectations à la main, ce qui pouvait prendre un temps très conséquent pour finalement obtenir une solution loin d'être optimale. Il y a deux ans, un des projets proposés était de résoudre ce problème d'affectation à l'aide de la programmation par contraintes. Les étudiants qui ont travaillé sur ce projet ont fourni le modèle suivant : chaque

[2]Dans ce cas, les étudiants restants réalisent de nouveaux choix parmi les projets non déjà assignés, et une nouvelle affectation est calculée.

étudiant est représenté par une variable dont le domaine est constitué par l'ensemble des choix effectués par l'étudiant, plus une valeur "joker" 0, signifiant "aucun projet n'est affecté lors de cette recherche". Des contraintes sont posées entre chaque paire (X_i, X_j) d'étudiants, exprimant "$(X_i \neq X_j) \vee (X_i = X_j = 0)$". Des variables de préférence mesurent la satisfaction de chaque étudiant. Elles sont liées aux variables représentant les étudiants par des contraintes définies en extension. $\{(p_1, m), \ldots, (p_m, 1), (0, 0)\}$. L'objectif consiste à minimiser la somme des variables de préférence. Par la suite, le modèle que nous venons de présenter sera référencé comme le *modèle initial*.

Le résolveur de contraintes est d'abord lancé sur le modèle initial pour une courte période de temps (1 sec.) afin de fournir une première solution S_0, et une borne supérieure sur la qualité des solutions. Avant de démarrer la phase d'apprentissage, une contrainte acceptant uniquement les instanciations avec une valeur d'objectif au moins aussi bonne que celle de S_0 est posée. L'apprentissage est alors réalisé sur un ensemble d'instances E, contenant S_0 (marquée comme positive) et une ensemble d'instances aléatoires (généralement négatives). Nous n'avons pas imposé de limite de temps, en revanche la consistance *k-SAC(v)*(décrite en section 4.3) a été utilisée à la place de résolutions complètes. A la fin de la phase d'apprentissage, la contrainte apprise est ajoutée au problème et le solveur est lancé.

Sachant que la taille réelle du problème est trop grande pour que celui-ci soit résolu plusieurs fois, nous avons d'abord réalisé des tests sur des problèmes réduits issus des données réelles. Ces problèmes préservent les ratio "nombre de projets/nombre d'étudiants" et la répartition particulière des choix. Le gain obtenu sur le problème complèt est donné à la fin de la section.

La table 9 montre le temps moyen et le nombre de nœuds du modèle initial en fonction du nombre d'étudiants n.

n	#nœuds	temps (sec.)
10	110	0.2
15	2 648	7.6
20	137 982	183.2

TAB. 9 – Modèle initial.

La table 10 montre les résultats obtenus avec les mêmes données que celles de la table 9 en apprenant une `Gcc` sur les variables représentant les étudiants, en fonction du nombre d'étudiants n impliqués dans le problème et le nombre d'instances $|E|$ utilisées pendant la phase d'apprentissage. Le temps d'apprentissage inclut le temps de génération de l'instance positive initiale.

n	$\|E\|$	#nœuds	temps (sec.)
10	10	12	0.2 + 0.0
15	10	57	2.6 + 1.1
15	20	50	3.8 + 1.1
15	30	50	5.1 + 0.9
15	40	50	5.9 + 0.9
20	10	4 801	5.7 + 14.9
20	20	2 800	6.8 + 13.0
20	30	2 643	7.9 + 8.4
20	40	1 998	9.1 + 8.4

TAB. 10 – Modèle reformulé.

La première observation que l'on peut faire est que, grâce à la relaxation à une consistance locale plutôt que des résolutions complètes, le temps d'apprentissage est très raisonnable (9.1 sec. dans le pire des cas). La deuxième est que plus nous utilisons d'instances, plus la contrainte apprise est précise, et plus compact est l'arbre de recherche. Il semble donc sur ce problème qu'il soit toujours préférable d'améliorer le modèle avant de résoudre le problème. En effet, la somme du temps d'apprentissage et du temps de résolution est inférieure au temps requis pour résoudre directement le modèle initial. Pour $n = 10$, la phase d'apprentissage n'est pas lancée car la solution optimale a été trouvée pendant la recherche de la première solution initiale. Le gain est de quelques secondes pour $n = 15$, puis il atteint un facteur 10 lorsque $n = 20$.

Sur le problème réel ($n = 60$), cette approche a permis de résoudre le problème en 43 minutes (dont 90 secondes d'apprentissage, réalisé sur 50 instances) alors que la résolution du modèle initial a été stoppée après 12 heures. Cette méthode ne nécessite pas de réglage particulier car le temps de la phase d'apprentissage est contrôlé, et elle semble robuste au nombre d'instances sélectionnées pour l'apprentissage.

6 Perspectives

L'aide à la modélisation paraît devenir un thème de recherche important en programmation par contraintes, notamment dans le but de favoriser sa diffusion. Actuellement, le nombre de contributions sur ce thème est encore relativement limité. Notre première perspective sera de généraliser les aspects algorithmiques de notre étude à d'autres contraintes à paramètres. Une autre perspective consiste à étudier les améliorations réalisables dans le domaine de la robustesse des stratégies de recherche par défaut des résolveurs de contraintes, notamment par des processus prenant en compte le modèle.

7 Conclusion

Cet article présente un nouveau paradigme d'apprentissage de contraintes à partir de solutions et non-solutions de sous-problèmes. Le but est d'améliorer automatiquement des modèles à contraintes. Afin de valider l'impact de notre approche nous avons conçu et implémenté un algorithme pour apprendre des contraintes de cardinalité globale, ainsi que des principes pour améliorer cet algorithme selon le type de problème pris en compte. Ces travaux ont été validés par une série d'expérimentations.

Références

[1] N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. *Proceedings CP*, pages 52–66, 2000.

[2] C. Bessière and P. Van Hentenryck. To be or not to be... a global constraint. *Proceedings CP*, pages 789–794, 2003.

[3] Choco. A Java library for constraint satisfaction problems, constraint programming and explanation-based constraint solving. *URL : http ://sourceforge.net/projects/choco*, 2005.

[4] R. Coletta, C. Bessiere, B. O'Sullivan, E.C. Freuder, S. O'Connell, and J. Quinqueton. Semi-automatic modeling by constraint acquisition. *Proceedings CP*, pages 812–816, 2003.

[5] S. Colton and I. Miguel. Constraint generation via automated theory formation. *Proceedings CP*, pages 575–579, 2001.

[6] R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14 :205–230, 2001.

[7] A. M. Firsch, C. Jefferson, B. Martinez Hernandez, and Ian Miguel. The rules of modelling : towards automatic generation of constraint programs. *2004 Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2004.

[8] Ilog. Solver 5.0 User's Manual. *URL : http ://www.ilog.fr*, 2000.

[9] N. Jussien. The versatility of using explanations within constraint programming. *Habilitation thesis of Université de Nantes*, 18 September 2003. also available as RR-03-04 research report at École des Mines de Nantes.

[10] A. Legtchenko, A. Lallouet, and A. Ed-Dbali. Intermediate consistencies by delaying expensive propagators. *Procedings FLAIRS*, 2004.

[11] C.-G. Quimper. Enforcing domain consistency on the extended global cardinality constraint is np-hard. Technical Report CS-2003-39, School of Computer Science, University of Waterloo, 2003.

[12] C.-G. Quimper, P. Van Beek, A. Lopez-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. *Proceedings CP*, pages 600–614, 2003.

[13] C.-G. Quimper, A. López-Ortiz, P. van Beek, and A. Golynski. Improved algorithms for the *global cardinality* constraint. *Proceedings CP*, 3258 :542–556, 2004.

[14] J-C. Régin. Generalized arc consistency for global cardinality constraint. *Proceedings AAAI*, pages 209–215, 1996.

[15] J-C. Régin. Minimization of the number of breaks in sports scheduling problems using constraints programming. *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, 57 :115–130, 2001.

[16] B. Smith, K. Stergiou, and T. Walsh. Modelling the golomb ruler problem. In J.C. Régin and W. Nuijten, editors, *Proceedings IJCAI'99 workshop on non-binary constraints*, Stockholm, Sweden, 1999.