Acquisition de contraintes par requêtes de généralisation

Christian Bessiere[1] **Remi Coletta**[1] **Abderrazak Daoudi**[1,2]
Nadjib Lazaar[1] **Younes Mechqrane**[1] **El Houssine Bouyakhf**[2]

[1] CNRS, U. Montpellier, France
[2] LIMIARF/FSR, U. Mohammed V Agdal, Rabat, Maroc
{bessiere, coletta, daoudi, lazaar, mechqrane}@lirmm.fr
bouyakhf@fsr.ac.ma

Abstract

Constraint acquisition assists a non-expert user in modeling her problem as a constraint network. In existing constraint acquisition systems the user is only asked to answer very basic questions. The drawback is that when no background knowledge is provided, the user may need to answer a great number of such questions to learn all the constraints. In this paper, we introduce the concept of *generalization query* based on an aggregation of variables into types. We present a constraint generalization algorithm that can be plugged into any constraint acquisition system. We propose several strategies to make our approach more efficient in terms of number of queries. Finally we experimentally compare the recent QUACQ system to an extended version boosted by the use of our generalization functionality. The results show that the extended version dramatically improves the basic QUACQ.

Résumé

L'acquisition de contraintes assiste un novice à modéliser son problème sous forme de réseau de contraintes. Dans les systèmes d'acquisition de contraintes existants, l'utilisateur est seulement demandé à répondre à des questions très simples. L'inconvénient est que lorsqu'aucune connaissance de base n'est fournie, l'utilisateur peut avoir besoin de répondre à un grand nombre de questions pour apprendre toutes les contraintes. Dans cet article, nous introduisons le concept de *requête de généralisation* basé sur une agrégation de variables sous forme de types. Nous présentons un algorithme de généralisation de contraintes qui peut être branché dans n'importe quel système d'acquisition de contraintes. Nous proposons plusieurs stratégies pour rendre notre approche plus efficace en terme de nombre de requêtes. Finalement, nous comparons expérimentalement le récent système QUACQ à une version étendue, renforcée par l'utilisation de notre fonctionnalité de généralisation. Les résultats montrent que la version étendue améliore considérablement la version de base de QUACQ.

1 Introduction

La programmation par contraintes est utilisée pour modéliser et résoudre des problèmes combinatoires dans plusieurs domaines d'application, tels que l'allocation de ressources ou l'ordonnancement. Cependant, la modélisation sous forme de contraintes nécessite une certaine expertise en programmation par contraintes. Cela empêche l'utilisation de cette technologie par un novice ; ce qui a un effet négatif sur l'adoption de la technologie de contraintes par des non-experts.

Plusieurs techniques ont été proposées pour aider l'utilisateur dans la tâche de la modélisation. Dans [10], Freuder et Wallace ont proposé l'agent matchmaker, un processus interactif où l'utilisateur est en mesure de fournir l'une des contraintes de son problème chaque fois que le système propose une mauvaise solution. Dans [12], Lallouet et al. ont proposé un système basé sur la programmation logique inductive qui utilise les connaissances de base sur la structure du problème pour apprendre une représentation du problème classifiant correctement les exemples. Dans [4, 6], Bessiere et al. ont fait l'hypothèse que la seule chose que l'utilisateur est en mesure de fournir est des exemples de solutions et non-solutions de son problème. Sur la base de ces exemples, le système *Conacq.1* apprend un ensemble de contraintes qui classifie correctement tous les exemples donnés jusqu'ici. Ce type d'apprentissage est ap-

pelé *apprentissage passif.* Dans [3], Beldiceanu et Simonis ont proposé *ModelSeeker*, une autre approche d'apprentissage passif. Des exemples positifs sont fournis par l'utilisateur. Le système organise ces exemples sous forme d'une matrice et identifie les contraintes dans le catalogue des contraintes globales ([2]) qui sont satisfaites par des lignes ou toute autre propriété structurelle que peut capturer *ModelSeeker* de tous les exemples.

En revanche, dans un apprentissage actif comme *Conacq.2*, le système propose des exemples à l'utilisateur pour qu'il les classifient comme des solutions ou des non-solutions [7]. Ces questions sont appelées requêtes d'appartenance [1]. CONACQ présente deux défis computationnels. Le premier concerne comment le système génère une requête utile et le deuxième porte sur combien de requêtes sont nécessaires pour le système afin qu'il converge vers l'ensemble de contraintes cibles. Il a été démontré que le nombre de requêtes nécessaire pour converger vers l'ensemble des contraintes cibles peut être exponentiel [8].

QUACQ est un récent système d'apprentissage actif qui est en mesure de demander à l'utilisateur de classifier une requête *partiale* [5]. En utilisant des requêtes partielles et étant donné un exemple négatif, QUACQ est en mesure de trouver une contrainte du problème que l'utilisateur a en tête dans un nombre de requêtes logarithmiques en taille de l'exemple. Cette composante clé de QUACQ permet de converger toujours vers le réseau de contraintes cibles dans un nombre polynomial de requêtes. Cependant, malgré sa bonne borne théorique, il peut s'avérer difficile à mettre en pratique. Par exemple, QUACQ demande à l'utilisateur de classifier plus de 8000 exemples pour apprendre le modèle complet de Sudoku.

Dans cet article, nous proposons une nouvelle technique pour rendre l'acquisition de contraintes plus efficace en pratique en utilisant un typage de variables. Dans les problèmes réels, les variables représentent souvent des composantes qui peuvent être classées en différents types. Par exemple, prenons le problème d'emploi du temps scolaire, les variables peuvent représenter des enseignants, des étudiants, des salles, des cours ou des plages horaires. Ces types sont souvent connus par l'utilisateur. Pour prendre en compte les types de variables, nous introduisons la notion de *requête de généralisation* . Nous attendons de l'utilisateur d'être en mesure de décider si une contrainte apprise peut être généralisée à d'autres portées de variables du même type que ceux de la contrainte apprise. Nous introduisons l'algorithme GENACQ qui demande à l'utilisateur de classifier les requêtes de généralisation à chaque fois qu'une contrainte est apprise. Nous proposons plusieurs stratégies et heuristiques pour sélectionner la requête, meilleure candidate, à la généralisation. Nous avons bronché notre fonctionnalité de généralisation dans le système d'acquisition de contraintes QUACQ, pour obtenir l'algorithme G-QUACQ. Nous évaluons expérimentalement le bénéfice de notre technique sur plusieurs jeux de données. Les résultats montrent que G-QUACQ améliore considérablement l'algorithme de base QUACQ en terme de nombre de requêtes.

Le reste de cet article est organisé comme suit. La section 2 donne les définitions nécessaires à la compréhension de la présentation technique. La section 3 décrit l'algorithme de généralisation. Dans la section 4, plusieurs stratégies sont présentées afin de rendre notre approche plus efficace. La section 5 présente les résultats expérimentaux que nous avons obtenus lors de la comparaison de G-QUACQ à QUACQ d'une part et des différentes stratégies dans G-QUACQ d'autre part. La section 6 conclut l'article et donne quelques directions pour les futures travaux.

2 Background

Nous introduisons quelques notions utiles pour la programmation par contraintes et l'apprentissage. La connaissance commune partagée entre un apprenant qui vise à résoudre le problème et l'utilisateur qui connaît le problème est un *vocabulaire*. Ce vocabulaire est représenté par un ensemble (fini) de variables X et de domaines $D = \{D(x_1), \ldots, D(x_n)\}$ sur $\mathbb{Z}$. Une contrainte c représente une relation $rel(c)$ sur un sous-ensemble de variables $var(c) \subseteq X$ (appelé la *portée* de c) qui spécifie quelles assignations de $var(c)$ sont permises. Les problèmes combinatoires sont représentés par des *réseaux de contraintes*. Un réseau de contraintes est un ensemble C de contraintes sur le vocabulaire (X, D). Un exemple e est une assignation (partiel/complète) d'un ensemble de variables $var(e) \subseteq X$. e est rejeté par une contrainte c (i.e., $e \not\models c$) ssi $var(c) \subseteq var(e)$ et la projection $e[var(c)]$ de e sur $var(c)$ n'est pas dans $rl(c)$. Une assignation complète e de X est une solution de C ssi pour tout $c \in C$, c ne rejette pas e. On note par $sol(C)$ l'ensemble des solutions de C.

En plus du vocabulaire, l'apprenant possède un *langage* Γ de relations à partir duquel il peut construire des contraintes sur des ensembles spécifiés de variables. Un *biais de contraintes* est une collection B de contraintes construite à partir du langage Γ sur le vocabulaire (X, D).

$$B = \{c \mid (var(c) \in X) \wedge (\exists r \in \Gamma \; s.t. \; rel(c) = r \cap D^{|var(c)|})\}$$

En terme d'apprentissage automatique, un *concept* est une fonction booléenne sur $D^X = \Pi_{x_i \in X} D(x_i)$, c-à-d, une application qui attribue à chaque exemple $e \in D^X$ une valeur de $\{0, 1\}$. Le *concept cible* est le concept f_T qui renvoie 1 pour e si et seulement si e est une solution du problème que l'utilisateur a dans la tête. Dans un contexte de programmation par contraintes, le concept cible est représenté par *un réseau cible* noté C_T. Une *requête* $Ask(e)$, avec $var(e) \subseteq X$, est une question de classification posée à l'utilisateur, où e est une assignation dans

$D^{var(e)} = \Pi_{X_i \in var(e)} D(X_i)$. Un ensemble de contraintes C *accepte* une assignation e si et seulement si il n'existe aucune contrainte $c \in C$ rejetant e. La réponse à $Ask(e)$ est 'yes' *si et seulement si* C_T accepte e.

Un *type* T_i est un sous-ensemble de variables définit par l'utilisateur comme ayant une propriété commune. Une variable x est de type T_i ssi $x \in T_i$. Une portée $var = (x_1, \ldots, x_k)$ de variables appartient à une séquence $s = (T_1, \ldots, T_k)$ (noté $var \in s$) si est seulement si $x_i \in T_i$ pour tout $i \leq k$. Considérons $s = (T_1, T_2, \ldots, T_k)$ et $s' = (T'_1, T'_2, \ldots, T'_k)$ deux séquences de types. On dit que s' *couvre* s (noté $s \sqsubseteq s'$) ssi $T_i \subseteq T'_i$ pour tout $i = 1..k$. Une relation r *s'applique* sur une séquence de types s si et seulement si $(var, r) \in C_T$ pour tout $var \in s$. Une séquence de types s est *maximale* par rapport à une relation r si et seulement si r s'applique sur s et qu'il n'existe pas un s' qui couvre s sur laquelle r s'applique.

3 Algorithme GENACQ

Dans cette section, nous présentons GENACQ, un algorithme d'*acquisition généralisée*. L'idée derrière cet algorithme est, étant donné une contrainte c apprise qui s'applique sur $var(c)$, généraliser cette contrainte sur les séquences de types s qui couvrent $var(c)$ en posant les requêtes de généralisation $AskGen(s, r)$. L'utilisateur répond par 'yes' pour une requête de généralisation $AskGen(s, r)$ si et seulement si pour toute séquence var de variables couvertes par s la relation r s'applique sur var dans le réseau de contraintes cible C_T.

3.1 Description de GENACQ

L'algorithme GENACQ (voir Algorithm 1) prend comme entrée une contrainte c qui a été déjà apprise et un ensemble $NonTarget$ de contraintes qui n'appartient pas au réseau cible. Il utilise aussi la structure de données globale $NegativeQ$, qui est un ensemble de paires (s, r) pour lequel on connaît que r ne s'applique pas sur toutes les séquences de variables qui sont couvertes par s. c et $NonTarget$ peuvent venir de n'importe quel système d'acquisition de contraintes ou comme connaissances préalables. $NegativeQ$ est construite d'une manière incrémentale par chaque appel de GENACQ. GENACQ aussi utilise l'ensemble $Table$ comme structure de données locale. $Table$ va contenir toutes les séquences de types qui sont candidates pour généraliser c.

Dans la ligne 1, GENACQ initialise l'ensemble $Table$ par toutes les séquences s possibles de types qui contiennent $var(c)$. Dans la ligne 2, GENACQ initialise l'ensemble G par l'ensemble vide. G va contenir la sortie de GENACQ, c'est à dire, l'ensemble des séquences maximales de $Table$ sur lesquelles $rel(c)$ s'applique. Le compteur $\#NoAnswers$ compte le nombre de requêtes

Algorithm 1: GENACQ $(c, NonTarget)$

```
1  Table ← {s | var(c) ∈ s} \ {var(c)}
2  G ← ∅
3  #NoAnswers ← 0
4  foreach s ∈ Table do
5      if ∃(s', r) ∈ NegativeQ | rel(c) ⊆ r ∧ s' ⊑ s
       then
6          Table ← Table \ {s}
7      if
       ∃c' ∈ NonTarget | rel(c') = rel(c) ∧ var(c') ∈ s
8      then Table ← Table \ {s}
9  while Table ≠ ∅ ∧ #NoAnswers < cutoffNo do
10     pick s in Table
11     if AskGen(s, rel(c)) = yes then
12         G ← G ∪ {s} \ {s' ∈ G | s' ⊑ s}
13         Table ← Table \ {s' ∈ Table | s' ⊑ s}
14         #NoAnswers ← 0
       else
15         Table ← Table \ {s' ∈ Table | s ⊑ s'}
16         NegativeQ ← NegativeQ ∪ {(s, rel(c))}
17         #NoAnswers + +
18 return G ;
```

consécutives de généralisation qui sont classifiées comme négatives par l'utilisateur. Il est initialisé par zéro (ligne 3). $\#NoAnswers$ n'est pas utilisé dans la version de base de GENACQ mais il sera utilisé dans la version avec cutoffs. (Autrement dit, la version de base utilise $cutoffNo = +\infty$ dans la ligne 9).

La première boucle dans GENACQ (ligne 4) élimine de $Table$ les séquences s pour lesquelles nous connaissons déjà la réponse à la requête $AskGen(s, rel(c))$. Dans les lignes 5-6, GENACQ élimine de $Table$ toutes les séquences s telles qu'une relation r impliquée par $rel(c)$ est déjà connue qu'elle ne s'applique pas sur une séquence s' couverte par s (i.e., (s', r) appartient à $NegativeQ$). C'est sûr de supprimer ces séquences car l'absence de r sur un certaines portées dans s' implique l'absence de $rel(c)$ sur un certaines portées dans s (voir le lemme 1). Dans les lignes 7-8, GENACQ élimine de $Table$ toutes les séquences s telles que $(var, rel(c)) \notin C_T$.

Dans la boucle principale de GENACQ (ligne 9), nous sélectionnons une séquence s de $Table$ à chaque itération et nous posons la requête de généralisation à l'utilisateur (ligne 11). Si l'utilisateur dit 'yes', s est une séquence sur laquelle $rel(c)$ s'applique. Nous mettons s dans G et nous supprimons de G toutes les séquences couvertes par s, afin de garder juste la maximale (ligne 12). Nous supprimons aussi de $Table$ toutes les séquences s' couvertes par s (ligne 13) afin d'éviter de poser des questions redondantes plus tard. Si l'utilisateur dit 'no', nous sup-

primons de $Table$ toutes les séquences s' qui couvrent s (ligne 15) car nous connaissons qu'elles ne sont plus candidates pour la généralisation de $rel(c)$ et nous les mettons dans $NegativeQ$ du fait que la réponse à $(s, rel(c))$ a été 'no'. La boucle termine quand $Table$ est vide et nous renvoyons G (ligne 18).

3.2 Complétude et complexité

Nous analysons la complétude et la complexité de GENACQ en terme de nombre de requêtes de généralisation.

Lemme 1. *Si $AskGen(s, r) = no$ alors pour chaque (s', r') tel que $s \sqsubseteq s'$ et $r' \subseteq r$, nous avons $AskGen(s', r') = no$.*

Preuve. Supposons que $AskGen(s, r) = no$. Par conséquent, il existe une séquence $var \in s$ tel que $(var, r) \notin C_T$. Comme $s \sqsubseteq s'$ nous avons $var \in s'$ et nous connaissons que $(var, r) \notin C_T$. Comme $r' \subseteq r$, nous avons également $(var, r') \notin C_T$. En conséquence, $AskGen(s', r') = no$. □

Lemme 2. *Si $AskGen(s, r) = yes$ alors pour chaque s' tel que $s' \sqsubseteq s$, nous avons $AskGen(s', r) = yes$.*

Preuve. Supposons que $AskGen(s, r) = yes$. Comme $s' \sqsubseteq s$, pour toute $var \in s'$ nous avons $var \in s$ et nous connaissons que $(var, r) \in C_T$. En conséquence, $AskGen(s', r) = yes$. □

Proposition 1 (Complétude)**.** *Lorsqu'il est appelé avec la contrainte c en entrée, l'algorithme GENACQ renvoie toutes les séquences maximales de types couvrant $var(c)$ sur lesquelles la relation $rel(c)$ s'applique.*

Preuve. Toutes les séquences couvrant $var(c)$ sont mises dans $Table$. Une séquences dans $Table$ est soit posée pour la généralisation ou supprimée de $Table$ dans les lignes 6, 8, 13, ou 15. Nous savons par le lemme 1 qu'une séquence supprimée dans la ligne 6, 8, ou 15 conduirait nécessairement à une réponse 'no'. Nous savons du lemme 2 qu'une séquence supprimée dans la ligne 13 est subsumée et moins générale que l'autre que nous venons d'ajouter à G. □

Proposition 2. *Étant donnée une contrainte c apprise et sa $Table$ associée, GENACQ utilise $O(|Table|)$ requêtes de généralisation pour renvoyer toutes les séquences maximales de types couvrant $var(c)$ sur lesquelles la relation $rel(c)$ s'applique.*

Preuve. Pour chaque requête sur $s \in Table$ posée par GENACQ, la taille de $Table$ décroît strictement quelle que soit la réponse. En conséquence, le nombre total de requêtes est majoré par $|Table|$. □

3.3 Exemple illustratif

Prenons le problème de Zèbre pour illustrer notre approche de généralisation. Le problème de Zèbre de Lewis Carroll a une solution unique. Le réseau cible est formalisé en utilisant 25 variables, partitionnées en 5 types de 5 variables chacun. Les 5 types sont *color*, *nationality*, *drink*, *cigaret* et *pet*. Il y a une clique de contraintes $\neq$ sur toutes les paires de variables de même type et 14 contraintes supplémentaires figurant dans la description du problème.

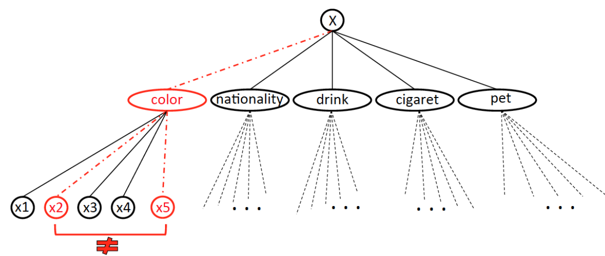

FIGURE 1 – Variables et types pour le problème de Zèbre.

La figure 1 montre les variables du problème de Zèbre et leurs types. Dans cet exemple, la contrainte $x_2 \neq x_5$ a été apprise entre les deux variables de type color x_2 et x_5. Cette contrainte est donnée comme entrée pour l'algorithme GENACQ. GENACQ calcule la $Table$ de toutes les séquences de types qui couvrent la portée (x_2, x_5). $Table = \{(x_2, color), (x_2, X), (color, x_5), (color, color), (color, X), (X, x_5), (X, color), (X, X)\}$. Supposons que nous sélectionnons $s = (X, x_5)$ à la ligne 10 de GENACQ. Selon la réponse de l'utilisateur ('no' dans ce cas), la $Table$ est réduite à $Table = \{(x_2, color), (x_2, X), (color, x_5), (color, color), (color, X)\}$. Comme prochaine itération, nous sélectionnerons $s = (color, color)$. L'utilisateur va répondre par 'yes' car il y a effectivement une clique de $\neq$ sur les variables de type $color$. Par conséquence, $(color, color)$ est ajoutée à G et $Table$ est réduite à $Table = \{(x_2, X), (color, X)\}$. Si nous sélectionnons (x_2, X), l'utilisateur répond par 'no' et nous réduisons $Table$ à l'ensemble vide et nous retournons $G = \{(color, color)\}$, ce qui signifie que la contrainte $x_2 \neq x_5$ peut être généralisée sur toutes les paires de variables dans la séquence $(color, color)$, c'est à dire, $(x_i \neq x_j) \in C_L$ pour tout $(x_i, x_j) \in (color, color)$.

3.4 Utilisation de la généralisation dans QUACQ

GENACQ est une technique générique qui peut être branchée dans n'importe quel système d'acquisition de contraintes. Dans cette section nous présentons G-QUACQ, un algorithme d'acquisition de contraintes obtenu en branchant GENACQ dans QUACQ, le système d'acquisition de contraintes présenté dans [5].

G-QUACQ est présenté dans l'algorithme 2. Comme elles apparaissent dans [5], nous ne donnons pas le code des fonctions `FindScope` et `FindC`. Mais disons quelques mots sur la façon dont elles travaillent. Étant donné les ensembles de variables S_1 et S_2, $\texttt{FindScope}(e, S_1, S_2, false)$ renvoie le sous-ensemble de S_2 qui, conjointement avec S_1, forme la portée d'une contrainte du biais B de contraintes possibles qui rejettent e. Inspiré d'une technique utilisée dans QUICKXPLAIN [11], `FindScope` requiert un nombre de requêtes logarithmiques en $|S_2|$ et linéaire en taille de la portée finale retournée. La fonction `FindC` prend comme paramètre l'exemple négatif e et la portée retournée par `FindScope`. Elle renvoie une contrainte de C_T avec la portée donnée qui rejette e. Pour chaque assignation e, $\kappa_B(e)$ décrit l'ensemble de toutes les contraintes dans B qui rejettent e.

Algorithm 2: G-QUACQ

```
1  C_L ← ∅, NonTarget ← ∅;
2  while true do
3      if sol(C_L) = ∅ then return "collapse"
4      choose e in D^X accepted by C_L and rejected by B
5      if e = nil then return "convergence on C_L"
6      if Ask(e) = yes then
7          B ← B \ κ_B(e)
8          NonTarget ← NonTarget ∪ κ_B(e)
9      else
10         c ← FindC(e, FindScope(e, ∅, X, false))
11         if c = nil then return "collapse"
12         else
13             G ← GENACQ(c, NonTarget)
14             foreach s ∈ G do
15                 C_L ← C_L ∪ {(var, rel(c)) | var ∈ s}
```

G-QUACQ a une structure très similaire à QUACQ. Il initialise l'ensemble $NonTarget$ et le réseau C_L, et va apprendre à partir de l'ensemble vide (ligne 1). Si C_L est insatisfiable (ligne 3), l'espace des réseaux possibles s'effondre parce qu'il n'existe aucun sous-ensemble du biais B donné qui est en mesure de classifier correctement les exemples déjà posés à l'utilisateur. Dans la ligne 4, QUACQ calcule une assignation complète e qui satisfait C_L mais qui viole au moins une contrainte de B. Si un tel exemple n'existe pas (ligne 5), alors toutes les contraintes dans B sont impliquées par C_L, et l'algorithme converge. Si l'algorithme ne converge pas, nous proposons l'exemple e à l'utilisateur qui va répondre par yes ou no (ligne 6). Si la réponse est yes, nous supprimons de B l'ensemble $\kappa_B(e)$ de toutes les contraintes dans B qui rejettent e (ligne 7) et nous ajoutons à l'ensemble $NonTarget$ toutes ces contraintes écartées afin de les utiliser dans GENACQ (ligne 8). Si la réponse est no, nous sommes sûr que e viole au moins une contrainte du réseau cible C_T. Nous appelons alors la fonction `FindScope` pour découvrir la portée de l'une de ces contraintes violées. `FindC` permet de sélectionner laquelle parmi les portées données est violée par e (ligne 10). Si aucune contrainte n'est retournée (ligne 11), c'est encore une condition de s'effondrer vu que nous n'avons pas trouvé dans B une contrainte qui rejette l'un des exemples négatifs. Autrement, nous savons que la contrainte c retournée par `FindC` appartient au réseau cible C_T. C'est ici que l'algorithme diffère de QUACQ vu que nous faisons appel à GENACQ pour retrouver toutes les séquences maximales de types couvrant $var(c)$ sur lesquelles $rel(c)$ s'applique. Elles sont retournées dans G (ligne 13). Ensuite, pour toute séquence de variables var appartenant à l'une de ces séquences dans G, nous ajoutons la contrainte $(var, rel(c))$ au réseau appris C_L (ligne 14).

4 Stratégies

GENACQ apprend les séquences maximales de types sur lesquelles une contrainte peut être généralisée. L'ordre dans lequel les séquences ont été sélectionné de $Table$ dans la ligne 10 de l'algorithme 1 n'est pas spécifié par l'algorithme. Comme illustré sur l'exemple suivant, les différents ordres peuvent entraîner plus ou moins rapidement aux bonnes séquences (maximales) sur lesquelles une relation r s'applique. Revenons à notre exemple sur le problème de Zèbre (Section 3.3). Dans la manière par laquelle nous développons l'exemple, nous avons besoin seulement de 3 requêtes de généralisation pour vider l'ensemble $Table$ et converger sur la séquence maximale $(color, color)$ sur laquelle $\neq$ s'applique :

1. $AskGen((X, x_5), \neq) = no$
2. $AskGen((color, color), \neq) = yes$
3. $AskGen((x_2, X), \neq) = no$

Utilisant un autre ordre, GENACQ a besoin de 8 requêtes de généralisation :

1. $AskGen((X, X), \neq) = no$
2. $AskGen((X, color), \neq) = no$
3. $AskGen((color, X), \neq) = no$
4. $AskGen((X, x_5), \neq) = no$
5. $AskGen((x_2, X), \neq) = no$
6. $AskGen((x_2, color), \neq) = yes$
7. $AskGen((color, x_5), \neq) = yes$
8. $AskGen((color, color), \neq) = yes$

Si l'on veut réduire le nombre de requêtes de généralisation, on peut se demander quelle stratégie utiliser. Dans cette section, nous présentons deux techniques. La première idée est de sélectionner les séquences dans l'ensemble $Table$ en suivant un ordre donné par une heuristique qui permet de minimiser le nombre de requêtes de généralisation. La deuxième idée est d'utiliser un cuttof sur le nombre successive de requêtes négatives auquel nous acceptons de faire face, conduisant à une stratégie de généra-

lisation non complète : la sortie de GENACQ ne sera plus garantie d'être les séquences *maximales*.

4.1 Heuristiques de sélection de requêtes

Nous proposons certaines heuristiques de sélection de requêtes basées sur les variables/contraintes impliquées dans les requêtes. Commençons par l'heuristique la plus intuitive, celle basée sur le nombre de variables impliquées dans la séquence s.

- **max_VAR :** Cette heuristique sélectionne une séquence s impliquant un nombre maximal de variables, c'est à dire, maximisant $|\bigcup_{T \in s} T|$. L'intuition derrière cette heuristique est que, si l'utilisateur répond par 'yes', un grand nombre de contraintes sera inféré.
- **min_VAR :** Cette heuristique sélectionne une séquence s impliquant un nombre minimal de variables, c'est à dire, minimisant $|\bigcup_{T \in s} T|$. L'intuition derrière cette heuristique est que, nous augmentons la chance d'avoir une réponse 'yes', et, si l'utilisateur répond par 'no', un grand nombre de séquences est supprimé de $Table$.

Nous pouvons avoir des heuristiques similaires pour le nombre de contraintes possibles impliqué dans la séquence s.

- **max_CST :** Cette heuristique sélectionne la séquence s maximisant le nombre de contraintes possibles (var, r) dans le biais, telle que var est dans s et r est la relation que nous cherchons à généraliser. L'intuition derrière cette heuristique est que, si l'utilisateur dit 'yes', la généralisation sera maximale selon le nombre de contraintes.
- **min_CST :** Cette heuristique sélectionne une séquence s minimisant le nombre de contraintes possibles (var, r) dans le biais, telle que var est dans s et r est la relation que nous cherchons à généraliser. L'intuition derrière cette heuristique est de minimiser la chance de recevoir une réponse 'no'.

Comme base de comparaison, nous définissons un sélecteur aléatoire.

- **random :** Il choisit au hasard une séquence s dans $Table$.

4.2 Utilisation des Cutoffs

L'idée ici est de quitter GENACQ avant d'avoir prouvé la maximalité des séquences retournées. Nous mettons un seuil $cutoffNo$ sur le nombre de réponses consécutives négatives afin d'éviter l'utilisation des requêtes pour vérifier des séquences peu prometteuses. L'espoir est que GENACQ renvoie les séquences quasi-maximales de types en dépit de ne pas pouvoir prouver la maximalité. Cette stratégie de cuttoff est mise en œuvre en mettant la variable $cutoffNo$ à une valeur prédéfinie. Dans les lignes 14 et 17 de GENACQ, un compteur de réponses négatives consécutives est respectivement initialisé et incrémenté selon la réponse de l'utilisateur. Dans la ligne 9, ce compteur est comparé à $cutoffNo$ afin de décider de quitter ou non.

5 Expérimentations

Nous avons fait quelques expérimentations pour évaluer l'impact de l'utilisation de notre fonctionnalité de généralisation GENACQ dans le système d'acquisition de contraintes QUACQ. Nous avons implémenté GENACQ et nous l'avons branché dans QUACQ, pour obtenir la version G-QUACQ. Nous présentons tout d'abord les jeux de données que nous avons utilisés pour nos expérimentations. Ensuite, nous présentons les résultats de plusieurs expérimentations. La première compare la performance de G-QUACQ à QUACQ de base. La deuxième présente les expérimentations évaluant les différentes stratégies que nous avons proposées (heuristiques de sélection de la requête et cuttoffs) sur G-QUACQ. La troisième évalue la performance de notre approche de généralisation quand notre connaissance sur les types de variables est incomplète.

5.1 Jeux de données

Problème de Zèbre. Comme présenté dans la section 3.3, Le problème de Zèbre de Lewis Carroll est formulé en utilisant 5 types de 5 variables chacun, avec 5 cliques de contraintes $\neq$ et 14 contraintes supplémentaires données dans la description du problème. Nous avons alimenté QUACQ et G-QUACQ par un biais B de 4450 contraintes unaires et binaires prises à partir d'un langage de 24 contraintes arithmétiques et contraintes de distance.
Sudoku. Le modèle de Sudoku est exprimé en utilisant 81 variables de domaines de taille 9, et 810 contraintes binaires $\neq$ sur les lignes, les colonnes et les carrés. Dans ce problème les types sont les 9 lignes, les 9 colonnes et les 9 carrés de 9 variables chacun. Nous avons alimenté QUACQ et G-QUACQ par un biais B de 6480 contraintes binaires prises à partir du langage $\Gamma = \{=, \neq\}$.
Carré latin. Le problème de Carré latin se compose d'une table de taille $n \times n$ dans laquelle chaque élément se produit une fois dans chaque ligne et chaque colonne. Pour ce problème, nous utilisons 25 variables de domaines de taille 5 et de 100 contraintes binaires $\neq$ sur les lignes et les colonnes. Les lignes et les colonnes sont les types de variables (10 types). Nous avons alimenté QUACQ et G-QUACQ par un biais B de contraintes basé sur le langage $\Gamma = \{=, \neq\}$.
Problème d'affectation de fréquences radio. Le problème du RLFAP consiste à fournir des canaux de communication à partir de ressources spectrales limitées [9]. Ici, nous construisons une version simplifiée du RLFAP qui consiste à distribuer toutes les fréquences disponibles sur les stations de base du réseau. Le modèle de contrainte

TABLE 1 – QUACQ vs G-QUACQ.

	QUACQ	G-QUACQ +random			
	$\#Ask$	$\#Ask$	$\#AskGen$	$\#q_P$	$\#q_N$
Zèbre	638	257	67	10	57
Sudoku	8645	260	166	42	124
Carré latin	1129	117	60	16	44
RLFAP	1653	151	37	16	21
Purdey	173	82	31	5	26

à 25 variables avec des domaines de tailles 25 et 125 contraintes binaires. Nous avons cinq stations de cinq terminaux (émetteurs/récepteurs), ce qui forment cinq types. Nous avons alimenté QUACQ et G-QUACQ par un biais B de 1800 contraintes prises à partir d'un langage de 6 contraintes arithmétiques et de distance.

Purdey. Comme le Zèbre, ce problème a une seule solution. Quatre familles ont été arrêté par le magasin général de Purdey, chacun pour acheter un article différent et payer différemment. Sous un ensemble de contraintes supplémentaires figurant dans la description, le problème est de savoir comment nous pouvons associer chaque famille a l'article qu'elle a acheté et comment elle a payé pour cela. Le réseau cible de Purdey a 12 variables avec les domaines de tailles 4 et 30 contraintes binaires. Ici, nous avons trois types de variables qui sont *family*, *bought* et *paid*, chacun d'eux contient quatre variables.

5.2 Résultats

Pour toutes nos expérimentations, nous rapportons, le nombre total $\#Ask$ de requêtes posées par QUACQ de base, le nombre total $\#AskGen$ de requêtes de généralisation et le nombre $\#q_N$ et $\#q_P$ de requêtes de généralisation successivement négatives et positives, où $\#AskGen = \#q_P + \#q_N$.

Notre première expérimentation compare QUACQ et G-QUACQ dans sa version de référence, G-QUACQ +rand, sur notre jeux de données. Le tableau 1 rapporte les résultats. Nous observons que le nombre de requêtes posées par G-QUACQ est considérablement réduit par rapport à QUACQ. Cela est particulièrement vrai sur des problèmes avec de nombreux types impliquant de nombreuses variables, tel que Sudoku ou Carré latin. G-QUACQ acquiert le Sudoku en 260 requêtes standards plus 166 requêtes de généralisation, alors que QUACQ l'acquiert en 8645 requêtes standards.

Concentrons-nous maintenant sur le comportement de nos différentes heuristiques dans G-QUACQ. Le tableau 2(haut) rapporte les résultats obtenus avec G-QUACQ en utilisant min_VAR, min_CST, max_VAR, et max_CST pour acquérir le modèle du Sudoku. (Les autres problèmes montrent des tendances similaires.) Les

TABLE 2 – G-QUACQ avec les heuristiques et la strategie cutoff sur le Sudoku.

	cutoff	$\#Ask$	$\#AskGen$	$\#q_P$	$\#q_N$
random	$+\infty$	260	166	42	124
min_VAR			90	21	69
min_CST			132	63	69
max_VAR			263	63	200
max_CST			247	21	226
min_VAR	3	260	75	21	54
	2		57	21	36
	1		**39**	21	18
min_CST	3	626	238	112	126
	2	679	231	132	99
	1	837	213	153	60

résultats montrent clairement que max_VAR, et max_CST sont de très mauvaises heuristiques. Elles sont pires que random. En revanche, min_VAR et min_CST surpassent significativement random. Elles nécessitent respectivement 90 et 132 requêtes de généralisation au lieu de 166 pour random. Notez qu'elles demandent tous le même nombre de requêtes standards (260) car elles ont toutes trouvé les mêmes ensembles maximaux de séquences pour chaque contrainte apprise.

En bas du tableau 2, nous comparons le comportement de nos deux meilleures heuristiques (min_VAR et min_CST) lorsqu'elles sont combinées à la stratégie de cutoff. Nous avons essayé toutes les valeurs de cutoff de 1 à 3. Une première observation est que min_VAR reste la meilleure quelque soit la valeur de cuttof. Fait intéressant, même avec un cutoff égale à 1, min_VAR nécessite le même nombre de requêtes standards que les versions de G-QUACQ sans cuttof. Cela signifie que l'utilisation de min_VAR comme heuristique de sélection dans $Table$, G-QUACQ est en mesure de retourner les séquences maximales même si elle est arrêtée après la première réponse négative de généralisation. Nous observons également que le nombre de requêtes de généralisation avec min_VAR diminue lorsque le cuttof devient plus petit (90 à 39 quand cuttof passe de $+\infty$ à 1). En regardant les deux dernières colonnes, nous voyons que c'est le nombre $\#q_N$ de réponses négatives qui diminue. La bonne performance de min_VAR + cutoff=1 peut donc être expliquée par le fait que min_VAR sélectionne en premier les requêtes qui couvrent un nombre minimal de variables ; ce qui augmente les chances d'avoir une réponse 'yes'. Finalement, nous observons que l'heuristique min_CST n'a pas les mêmes bonnes caractéristiques que min_VAR. Plus le cutoff est faible, plus le nombre de requêtes standards nécessaires devient important, pas de compensation pour l'économie de nombre de requêtes de généralisation (de 260 à 837 requêtes standards pour min_CST quand cutoff va de

TABLE 3 – G-QUACQ avec `random`, `min_VAR`, et cutoff=1 sur Zèbre, Carré latin, RLFAP, and Purdey.

	$\#Ask$	$\#AskGen$	$\#q_P$	$\#q_N$
Zèbre				
Random	257	67	10	57
`min_VAR`		48	5	43
`min_VAR` +cutoff=1		**23**	5	18
Carré latin				
Random	117	60	16	44
`min_VAR`		34	10	24
`min_VAR` +cutoff=1		**20**	10	10
RLFAP				
Random	151	37	16	21
`min_VAR`		41	14	27
`min_VAR` +cutoff=1		**22**	14	8
Purdey				
Random	82	31	5	26
`min_VAR`		24	3	21
`min_VAR` +cutoff=1		**12**	3	9

TABLE 4 – G-QUACQ lorsque le pourcentage de types fourni augmente.

	% of types	$\#Ask$	$\#AskGen$
Zèbre	0	638	0
	20	619	12
	40	529	20
	60	417	27
	80	332	40
	100	257	48
Sudoku 9×9	0	8645	0
	33	3583	232
	66	610	60
	100	260	39
Carré latin	0	1129	0
	50	469	49
	100	117	20
Purdey	0	173	0
	33	111	8
	66	100	10
	100	82	12

$+\infty$ à 1). Cela signifie qu'avec `min_CST`, lorsque cutoff est trop petit, GENACQ ne retourne pas les séquences maximales de types où la contrainte apprise s'applique.

Dans le tableau 3, nous rapportons la performance de G-QUACQ avec `random`, avec `min_VAR` et avec `min_VAR` +cutoff=1 pour tous les autres problèmes. Nous voyons que `min_VAR` et cutoff=1 améliore significativement les performances de G-QUACQ pour tous les problèmes.

De ces expériences, nous voyons que G-QUACQ avec `min_VAR` +cutoff=1 conduit à des économies considérables en nombre de requêtes par rapport à QUACQ : 257+23 au lieu de 638 pour Zèbre, 260+39 au lieu de 8645 pour Sudoku, 117+20 au lieu de 1129 pour Carré latin, 151+22 au lieu de 1653 pour RLFAP et 82+12 au lieu de 173 pour Purdey.

Dans notre dernière expérimentation, nous montrons l'effet sur la performance de G-QUACQ lors d'un manque de connaissances sur certains types de variables. Nous avons repris nos 5 jeux de données dans lesquels nous avons fait varier le pourcentage de types connus par l'algorithme. Cela simule une situation dans laquelle l'utilisateur ne sait pas que certaines variables sont du même type. Par exemple, dans le Sudoku, l'utilisateur n'a pas pu remarqué que les variables sont regroupées en colonnes. La Figure 2 présente le nombre de requêtes standards et requêtes de généralisation posées par G-QUACQ avec `min_VAR` +cuttof=1 pour apprendre le modèle de RLFAP lorsqu'il est alimenté par une connaissance de plus en plus précise des types. Nous observons que dès qu'un petit pourcentage de types est connu (20%), G-QUACQ réduit considérablement le nombre de requêtes. Le tableau 4 donne la même information pour tous les autres problèmes.

6 Conclusion

Nous avons proposé une nouvelle technique pour rendre l'acquisition de contraintes plus efficace en utilisant des informations sur les types de composantes des variables représentants le problème. Nous avons introduit les requêtes de généralisation, un nouveau type de requête qui demande à l'utilisateur de généraliser une contrainte à d'autres portées de variables du même type où cette contrainte éventuellement s'applique. Notre nouvelle technique, GENACQ, peut être appelée à généraliser chaque

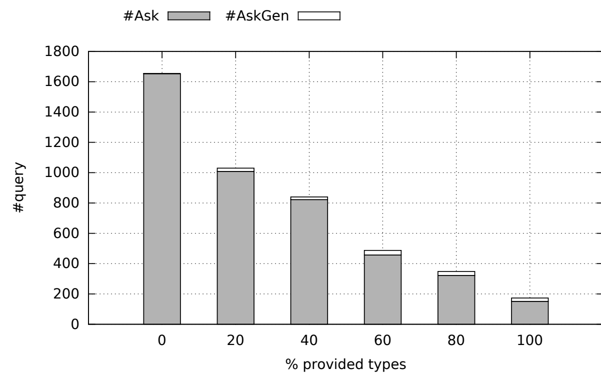

FIGURE 2 – G-QUACQ sur RLFAP lorsque le pourcentage de types fourni augmente.

nouvelle contrainte apprise par un système d'acquisition de contraintes. Nous avons proposé plusieurs heuristiques et stratégies pour sélectionner la requête, meilleure candidate à la généralisation. Nous avons branché GENACQ dans le système d'acquisition de contraintes QUACQ, pour obtenir l'algorithme G-QUACQ. Nous avons évalué expérimentalement le bénéfice de notre approche sur plusieurs jeux de données, avec et sans connaissance complète sur les types de variables. Les résultats montrent que G-QUACQ améliore considérablement l'algorithme de base QUACQ en terme de nombre de requêtes.

Références

[1] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4) :319–342, 1987.

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

[3] Nicolas Beldiceanu and Helmut Simonis. A model seeker : Extracting global constraint models from positive examples. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2012.

[4] Christian Bessiere, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2004.

[5] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. IJCAI/AAAI, 2013.

[6] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In 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*, pages 23–34. Springer, 2005.

[7] Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 50–55, 2007.

[8] Christian Bessiere and Frédéric Koriche. Non learnability of constraint networks with membership queries. Technical report, Coconut, Montpellier, France, February, 2012.

[9] Bertrand Cabon, Simon de Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio link frequency assignment. *Constraints*, 4(1) :79–89, 1999.

[10] Eugene C. Freuder and Richard J. Wallace. Suggestion strategies for constraint-based matchmaker agents. *International Journal on Artificial Intelligence Tools*, 11(1) :3–18, 2002.

[11] Ulrich Junker. Quickxplain : Preferred explanations and relaxations for over-constrained problems. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 167–172. AAAI Press / The MIT Press, 2004.

[12] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 45–52. IEEE Computer Society, 2010.