Acquisition de contraintes avec des requêtes partielles

Christian Bessiere[1] **Remi Coletta**[1] **Emmanuel Hebrard**[2] **George Katsirelos**[3]
Nadjib Lazaar[1] **Nina Narodytska**[4] **Claude-Guy Quimper**[5] **Toby Walsh**[4]

[1]CNRS, Université de Montpellier, France
[2]LAAS-CNRS, Toulouse, France
[3]INRA Toulouse, France
[4]NICTA, UNSW, Sydney, Australie
[5]Université Laval, Québec, Canada

{bessiere,coletta,lazaar}@lirmm.fr hebrard@laas.fr gkatsi@gmail.com
ninan@cse.unsw.edu.au claude-guy.quimper@ift.ulaval.ca toby.walsh@nicta.com.au

Résumé

Nous apprenons des réseaux de contraintes en utilisant des requêtes partielles. Autrement dit, nous demandons à l'utilisateur de classer une affectation de sous-ensembles de variables comme positive ou négative. Nous fournissons un algorithme qui, étant donné un exemple complet négatif, apprend une contrainte du réseau cible avec un certain nombre de requêtes partielles, logarithmique en taille de l'exemple. Nous présentons une étude théorique sur les bornes inférieures en termes de requêtes pour apprendre certaines classes de réseaux de contraintes et montrons l'optimalité de notre algorithme générique dans certains cas. Enfin, nous évaluons expérimentalement notre algorithme.

Abstract

We learn constraint networks by asking the user partial queries. That is, we ask the user to classify assignments to subsets of the variables as positive or negative. We provide an algorithm that, given a negative example, focuses onto a constraint of the target network in a number of queries logarithmic in the size of the example. We give information theoretic lower bounds for learning some simple classes of constraint networks and show that our generic algorithm is optimal in some cases. Finally we evaluate our algorithm on some benchmarks.

1 Introduction

La programmation par contraintes (PPC) connait un succès croissant et est largement utilisée dans de nombreuses applications industrielles. Toutefois, la première difficulté que rencontre un utilisateur de ce paradigme est la modélisation. Quelle est la traduction fidèle de la spécification du problème en contraintes ? Plusieurs techniques ont été proposées pour répondre à ce besoin de façon automatique. Freuder et Wallace ont proposé dans [7] des stratégies de suggestion (*matchmaker*) pour les applications où les utilisateurs n'ont pas prévu toutes les contraintes à l'avance mais sont néanmoins capables d'en proposer à chaque fois que le système retourne une non-solution. La première version de Conacq (Conacq.1) [3, 4] implémente une acquisition *passive* de contraintes où l'utilisateur fournit un ensemble de solutions/non-solutions (ex. des emplois du temps conçus à la main). Sur la base de ses exemples, Conacq.1 apprend un ensemble de contraintes qui accepte/rejette toutes les solutions/non-solutions fournies au départ. Dans le même cadre passif, une autre approche utilise des solutions et non-solutions et réalise l'acquisition des contraintes grâce à une base de connaissances (incluant des interprétations logiques des solutions/non-solutions) et des techniques de Programmation Logique Inductive [10]. Dans [2], Beldiceanu et Simonis ont proposé ModelSeeker, un système d'acquisition de contraintes passive basé sur le catalogue des contraintes globales, dont la portée des contraintes sont des lignes, des colonnes ou toute autre propriété structurelle que peut capturer ModelSeeker. Seuls les exemples positifs (solutions) sont fournis par l'utilisateur. ModelSeeker implémente également une

technique efficace qui permet de classer les meilleures contraintes candidates sur un ensemble particulier de variables.

La version *active* de CONACQ (CONACQ.2) [5] sollicite l'aide de l'utilisateur à classifier des exemples comme positives/négatives (solutions/non-solutions) avec des requêtes d'appartenance complètes (c.-à-d. affectation complète des variables) [1]. Une acquisition active des contraintes présente plusieurs avantages : i) Le nombre de requêtes nécessaires pour converger à l'ensemble de contraintes cible peut être considérablement réduit. ii) l'utilisateur peut être une machine (ex. système expert, simulateur, etc.). Par exemple, la société NORMIND a récemment recruté un spécialiste en PPC pour transformer un système expert qui détecte les défaillances dans les circuit électroniques d'un avion Airbus en un modèle à contraintes, afin de rendre l'outil plus efficace et facile à maintenir. Un autre exemple est celui d'apprendre les contraintes qui traduisent les actions non atomiques d'un robot (ex. attraper une balle) en posant des requêtes à un simulateur de robot [11]. Le cadre actif d'acquisition de contraintes évoque deux défis connus en apprentissage automatique : i) La qualité et comment générer des requêtes ? ii) Le nombre de requêtes nécessaires pour converger vers l'ensemble des contraintes cible ? Concernant le deuxième point, il a été démontré que le nombre de requêtes complètes nécessaires pour atteindre le réseau de contraintes cible est exponentiel dans le cas général.

Dans cet article, nous proposons QUACQ (pour *QuickAcquisition*), un système actif d'acquisition de contraintes qui demande à l'utilisateur de classifier, non seulement des requêtes complètes, mais aussi des requêtes *partielles*. Étant donné un exemple négatif, QUACQ est capable d'apprendre une contrainte du réseau cible avec un nombre de requêtes (partielles) logarithmiques en taille de l'exemple. Nous présentons également une étude théorique sur les bornes inférieures en termes de requêtes pour apprendre certaines classes de réseaux de contraintes et montrons l'optimalité de notre algorithme générique dans certains cas. Enfin, nous évaluons expérimentalement notre algorithme.

QUACQ est défini de sorte à pouvoir apprendre un modèle générique. En PPC, les données sont généralement séparées du modèle. Le modèle du Sudoku, par exemple, contient des contraintes génériques comme la permutation des valeurs dans chaque carré. D'autre part, les données définissent les différentes grilles de sudoku avec les cases pré-remplies. Autre exemple, le problème d'emploi du temps, le modèle contient des contraintes génériques comme par exemple « aucun enseignant ne peut donner deux cours dans une même plage horaire ». Par contre, les données permettent de préciser le nombre des salles, la disponibilité d'un enseignant donné pour un créneau donnée, etc. Un des avantages de cette approche est qu'elle permet de réduire l'effort de l'utilisateur. Premièrement, QUACQ converge plus rapidement que les autres systèmes. Deuxièmement, les requêtes partielles sont plus faciles à classifier que les requêtes complètes. Troisièmement, contrairement aux autres approches, QUACQ n'a pas besoin d'exemples positifs pour apprendre l'ensemble des contraintes : un point intéressant lorsque le problème en question n'a jamais été résolu.

2 Contexte

L'apprenant et l'utilisateur doivent partager un même *vocabulaire* pour pouvoir communiquer. Dans le cadre d'acquisition de contraintes, ce vocabulaire est défini par un ensemble (fini) de variables X et leurs domaines $D = \{D(X_i)\}_{X_i \in X}$, où $D(X_i) \subset \mathbb{Z}$ est le domaine fini de la variable X_i. Étant donnée une suite de variables $S \subseteq X$, une *contrainte* représente une paire (c, S) (notée C_S), où c est une relation sur $\mathbb{Z}$ précisant les tuples autorisés pour les variables S. S étant la *portée* de la contrainte C_S. Un *réseau de contraintes* est un ensemble C de contraintes sur le vocabulaire (X, D). Une affectation e_Y sur un sous-ensemble de variables $Y \subseteq X$ est *rejetée* par une contrainte c_S si $S \subseteq Y$ et la projection $e_Y[S]$ de e sur les variables de S n'est pas dans c_S. Une affectation complète sur X est une *solution* de C *ssi* elle n'est rejetée par aucune contrainte dans C. Nous utilisons la notation $sol(C)$ pour l'ensemble des solutions de C et la notation $C[Y]$ pour l'ensemble des contraintes de C dont la portée est incluse dans Y.

En terme d'apprentissage automatique, nous utilisons la notation de *biais de contraintes*, noté B, qui représente un ensemble de contraintes construit à partir d'un langage Γ sur le vocabulaire (X, D). C'est à partir du biais B que l'apprenant construit le réseau de contraintes. Un *concept* est une fonction booléenne sur $D^X = \Pi_{X_i \in X} D(X_i)$. Etant donné un exemple $e \in D^X$, un *concept cible* f_T renvoie 1 pour e *ssi* e est une solution du problème que l'utilisateur a en tête, 0 autrement. Une *requête d'appartenance* $ASK(e)$ est une question de classification posée à l'utilisateur, où e est une affectation complète sur D^X. La réponse à $ASK(e)$ étant *oui ssi* $f_T(e) = 1$.

Pour être en mesure d'utiliser des requêtes *partielles*, nous posons une condition supplémentaire sur les capacités de l'utilisateur. Même si ce dernier n'est pas en mesure d'exprimer les contraintes de son problème, il est capable de décider si une instantiation partielle des variables X viole certaines exigences ou non. Un *réseau cible* est un réseau C_T tel que $sol(C_T) = \{e \in$

$D^X \mid f_T(e) = 1\}$. Une requête *partielle* $ASK(e_Y)$, avec $Y \subseteq X$, est une question de classification posée à l'utilisateur, où e_Y est une affectation *partielle* dans $D^Y = \Pi_{X_i \in Y} D(X_i)$. Un ensemble de contraintes C *accepte* une requête partielle e_Y *ssi* il n'existe aucune contrainte c_S dans C qui rejette $e_Y[S]$. La réponse à $ASK(e_Y)$ étant *oui ssi* C_T accepte e_Y. Pour toute affectation e_Y sur Y, nous notons $\kappa_B(e_Y)$ le sous-ensemble des contraintes de B violées par e_Y. Une affectation e_Y est dite *positive* ou *négative* selon la classification de l'utilisateur.

Nous nous intéressons au problème de *convergence* en acquisition de contraintes. Etant donné un ensemble E d'exemples (partiels) classés par l'utilisateur (en positif/negatif), on dit qu'un réseau de contraintes C est en conformité avec E si C accepte tous les positifs et rejette tous les négatifs dans E. On dit que le processus d'acquisition a *convergé* sur le réseau $C_L \subseteq B$ si C_L est en conformité avec E et pour tout autre réseau $C' \subseteq B$ en conformité avec E, nous avons $sol(C') = sol(C_L)$. S'il n'existe pas de $C_L \subseteq B$ tel que C_L est en conformité avec E, nous disons que nous avons atteint un état *d'effondrement* . Cela se produit lorsque $C_T \not\subseteq B$.

3 Algorithme Générique d'Acquisition de Contraintes

Dans cette section, nous présentons QuAcq, une nouvelle approche d'acquisition active de contraintes. QuAcq prend en entrée un biais B sur un vocabulaire (X, D). Il génère des requêtes (partielles) que pose à l'utilisateur jusqu'à ce qu'il atteigne un état de convergence en retournant un réseau de contraintes C_L équivalent au réseau cible C_T, ou un état d'effondrement. Lorsqu'un utilisateur répond par *oui* à une requête complète, on retire du biais B les contraintes violées par cet exemple. Dans le cas d'un *non*, qa entre dans une boucle (fonctions `FindScope` et `FindC`) qui permet d'ajouter d'une contrainte à C_L.

3.1 Description de QuAcq

qa (algo. 1) initialise le réseau qu'il va apprendre C_L à l'ensemble vide (ligne 1). Si C_L est unsatisfiable (ligne refqa :fail), l'espace des réseaux possibles s'effondre parce qu'il n'existe plus de sous-ensemble du biais B capable de classer correctement les requêtes déjà posées. A la ligne 4, QuAcq calcule une affectation complète e satisfaisant le C_L courant et viole au moins une contrainte du biais B. Si un tel exemple n'existe pas (ligne 5), alors toutes les contraintes qui restent dans B sont des contraintes impliquées par C_L, ce qui nous conduit à un état de convergence. Autrement, QuAcq pose la question $ASK(e)$ à l'utilisateur, qui pourra répondre par *oui* ou *non*.

Si la réponse est *oui*, nous pouvons retirer de B l'ensemble $\kappa_B(e)$ des contraintes qui rejettent l'exemple e (ligne6). Dans le cas d'un *non*, cela veut dire que e viole au moins une contrainte du réseau cible C_T. Nous appelons ensuite la fonction `FindScope` pour calculer la portée d'au moins une contrainte violée par e. La fonction `FindC` permet par la suite de déterminer quelle contrainte, avec une telle portée, a été violée par e (ligne 8). Si aucune contrainte n'est retournée par `FindC` (ligne 9), c'est encore une condition suffisante pour atteindre un état d'effondrement (c.-à-d. aucune contrainte de B ne rejette l'exemple négatif e). Autrement, la contrainte retournée par `FindC` est ajoutée au réseau C_L (ligne 10).

Algorithm 1: QuAcq : Acquisition de contraintes avec des requêtes partielles

```
1  C_L ← ∅;
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 B ← B \ κ_B(e) ;
7      else
8          c ← FindC(e, FindScope(e, ∅, X, false));
9          if c = nil then return "collapse";
10         else C_L ← C_L ∪ {c};
```

La fonction récursive `FindScope` prend en paramètres un exemple négatif e, deux ensembles de variables R et Y, et un booléen ask_query. Un invariant de `FindScope` est que e viole au moins une contrainte dont la portée est un sous-ensemble de $R \cup Y$. Lorsque `FindScope` est appelée avec ask_qery = **false**, nous savons déjà si R contient la portée d'une contrainte qui rejette e (ligne 1). Si ask_qery = **true**, nous demandons à l'utilisateur si la projection $e[R]$ est positive ou non (ligne 2). Si *oui*, nous pouvons retirer de B les contraintes qui rejettent $e[R]$, sinon, nous retournons l'ensemble vide (ligne 4). La ligne 5 est atteinte uniquement dans le cas où $e[R]$ ne viole aucune contrainte. Il est évident que $e[R \cup Y]$ viole au moins une contrainte. Par conséquent, et sachant que Y est un singleton, la variable dans Y appartient nécessairement à la portée d'une contrainte violée par $e[R \cup Y]$. La fonction retourne donc Y. Si aucune des conditions de retour n'est satisfaite, l'ensemble Y est divisé en deux (ligne 6) et nous appliquons une technique similaire à QuickXplain [1][9] pour élucider les variables

1. La différence principale est que QuAcq divise un en-

Algorithm 2: Fonction `FindScope` : retourne la portée d'une contrainte dans C_T

```
   function FindScope(in e, R, Y, ask_query) : scope;
   begin
1      if ask_query then
2          if ASK(e[R]) = yes then
3          B ← B \ κ_B(e[R]);
4          else return ∅;
5      if |Y| = 1 then return Y;
6      split Y into < Y_1, Y_2 > such that
       |Y_1| = ⌈|Y|/2⌉ ;
7      S_1 ← FindScope(e, R ∪ Y_1, Y_2, true);
8      S_2 ← FindScope(e, R ∪ S_1, Y_1, (S_1 ≠ ∅));
9      return S_1 ∪ S_2;
```

Algorithm 3: Fonction `FindC` : retourne une contrainte dans C_T avec une portée Y

```
   function FindC(in e, Y) : constraint;
   begin
1      B ← B \ {c_Y | C_L ⊨ c_Y};
2      Δ ← {c_Y ∈ B[Y] | e ⊭ c_Y};
3      if Δ = ∅ then return ∅;
4      while true do
5          choose e' in sol(C_L[Y]) such that
           ∃c, c' ∈ Δ, e' ⊨ c and e' ⊭ c';
6          if e' = nil then
7              if Δ = ∅ then return nil;
8              else pick c in Δ ; return c;
9          if ASK(e') = yes then
10             B ← B \ κ_B(e') ; Δ ← Δ \ κ_B(e');
11         else Δ ← Δ ∩ κ_B(e');
```

d'une contrainte violée par $e[R \cup Y]$ en un nombre logarithmique d'étapes (lignes (lines 7–9).

La fonction `FindC` prend en paramètres e et Y, e étant l'exemple négatif qui a permis à `FindScope` de retourner la portée Y d'une contrainte violée. `FindC` retire de B dans un premier temps toutes les contraintes de portée Y qui sont impliquées par le C_L courant (ligne 1) [2] L'ensemble Δ contient au départ l'ensemble des contraintes violées par e (ligne 2). Si Δ ne contient plus de contraintes avec une portée Y (ligne 3), `FindC` retourne $\varnothing$, ce qui provoquera un effondrement dans QuAcq. A la ligne 5, un exemple e' est généré de manière à avoir dans Δ à la fois des contraintes qui rejettent e' et d'autres qui l'acceptent. Si aucun exemple de ce type n'existe (ligne 6), cela signifie que l'ensemble des contraintes qui restent dans Δ est équivalent à $C_L[Y]$. Dans ce cas, `FindC` retourne une contrainte de Δ, sauf si Δ est vide (lignes 7-8). Si un exemple a été trouvé, on demande à l'utilisateur de le classifier (ligne 9). Si l'exemple est positif, les contraintes violées par cet exemple sont retirées de B et de Δ (ligne 10). Si l'exemple est négatif, on retire de Δ toutes les contraintes qui acceptent cet exemple. nous retirons de Δ toutes les contraintes accepter cet exemple (ligne 11).

3.2 Exemple Illustratif

Nous illustrons le fonctionnement de QuAcq sur un exemple simple. Considérons l'ensemble des variables $X_1, \ldots, X_5$ avec les domaines $\{1..5\}$, un langage $\Gamma = \{=, \neq\}$, un biais $B = \{=_{ij}, \neq_{ij} | \ i, j \in 1..5, i < j\}$, et un réseau de contraintes cible $C_T = \{=_{15}, \neq_{34}\}$. Supposons que le premier exemple généré à la ligne 4 dans QuAcq est $e_1 = (1, 1, 1, 1, 1)$. La trace d'exécution de `FindScope`$(e_1, \varnothing, X_1 \ldots X_5, \textbf{false})$ est présenté dans le tableau ci-dessous. Chaque ligne correspond à un appel à `FindScope`. Les requêtes portent toujours sur un sous-ensemble de variables de R. '$\times$' dans la colonne ASK signifie que l'appel précédent retourné un $\varnothing$, donc la question est ignorée. Les requêtes des lignes 1 et 2.1 dans le tableau ont permis à la fonction `FindScope` de retirer les contraintes $\neq_{12}, \neq_{13}, \neq_{23}$ et $\neq_{14}, \neq_{24}$ de B. Une fois la portée (X_3, X_4) est retourné, `FindC` exige a besoin d'un seul exemple pour retourner $\neq_{34}$ et retirer$=_{34}$ de B. Supposons maintenant que l'exemple suivant généré par QuAcq soit $e_2 = (1, 2, 3, 4, 5)$. `FindScope` retournera la portée (X_1, X_5) et `FindC` retournera par la suite $=_{15}$ le même traitement effectué sur e_1. Les contraintes $=_{12}, =_{13}, =_{14}, =_{23}, =_{24}$ sont retirées de B suite à une requête partielle positive sur $X_1, \ldots, X_4$ et $\neq_{15}$ par `FindC`. Ensuite, des exemples comme $e_3 = (1, 1, 1, 2, 1)$ et $e_4 = (3, 2, 2, 3, 3)$, positifs, permettront de retirer les contraintes $\neq_{25}, \neq_{35}, =_{45}$ et $=_{25}, =_{35}, \neq_{45}$ de B et atteindre la convergence.

call	R	Y	ASK	*return*
0	$\varnothing$	X_1, X_2, X_3, X_4, X_5	$\times$	X_3, X_4
1	X_1, X_2, X_3	X_4, X_5	yes	X_4
1.1	X_1, X_2, X_3, X_4	X_5	no	$\varnothing$
1.2	X_1, X_2, X_3	X_4	$\times$	X_4
2	X_4	X_1, X_2, X_3	yes	X_3
2.1	X_4, X_1, X_2	X_3	yes	X_3
2.2	X_4, X_3	X_1, X_2	no	$\varnothing$

semble de variables alors que QuickXplain divise un ensemble de contraintes

2. Cette étape pouvait être effectuée dans QuAcq, juste après la ligne 10, le coût de calcul étant réduit nous avons préféré l'effectuer dans `FindC`.

3.3 Complexité

Nous analysons la complexité de QuAcq en termes de nombre de requêtes. Les requêtes sont posées à l'utilisateur dans les lignes 6 de QuAcq, 2 de `FindScope` et 9 de `FindC`.

Proposition 1. *Etant donnés un biais B construit à partir d'un langage Γ, un réseau cible C_T et une portée Y, `FindC` a besoin de $O(|\Gamma|)$ requêtes pour retourner une contrainte c_Y de C_T si cette dernière existe.*

Démonstration. A chaque fois que `FindC` génère une requête, quelle que soit la réponse de l'utilisateur, la taille de l'ensemble Δ diminue strictement. Ainsi, le nombre total de requêtes posées dans `FindC` est majorée par $|\Delta|$, qui lui-même, est majorée par le nombre de contraintes du langage Γ d'arité $|Y|$ (voir ligne 2 de `FindC`).

□

Proposition 2. *Étant donnés un biais B, un réseau cible C_T et un exemple $e \in D^X \setminus sol(C_T)$, `FindScope` a besoin de $O(|S| \cdot \log|X|)$ requêtes pour retourner la portée S d'une des contraintes de C_T violée par e.*

Démonstration. `FindScope` est un algorithme récursif qui génère, au plus, une requête par appel (ligne 2). Par conséquent, le nombre de requêtes est majoré par le nombre de n ?uds de l'arbre des appels récursifs à `FindScope`. Nous allons montrer qu'une feuille de l'arbre est soit sur une branche qui conduit à une variable de la portée S qui sera retournée, ou elle représente le n ?ud fils d'une telle branche. Quand une branche ne conduit pas à une variable de la portée S, cette branche nous conduit nécessairement vers des feuilles qui correspondent à des appels de `FindScope` qui retournent $\varnothing$. La seule façon pour qu'un appel de `FindScope` au niveau des feuilles retourne l'ensemble vide est d'avoir reçu la réponse *non* à sa requête (ligne 4). Soit R_{fils}, Y_{fils} les valeurs des paramètres R et Y pour un appel feuille avec la réponse *non*, et R_{parent}, Y_{parent} les valeurs des paramètres R et Y de l'appel parent dans l'arborescence des appels récursifs. A partir de la réponse *non* sur la requête $ASK(e[R_{fils}])$, on peut déduire que $S \subseteq R_{fils}$ mais $S \nsubseteq R_{parent}$ parce que l'appel au niveau parent à reçu un *oui*. Considérons d'abord le cas où la feuille représente le fils gauche du n ?ud parent. Par construction, $R_{parent} \subsetneq R_{fils} \subsetneq R_{parent} \cup Y_{parent}$. En conséquence, Y_{parent} intersecte S, et le n ?ud parent est sur une branche qui conduit à une variable dans S. Considérons maintenant le cas où la feuille représente le fils droit du n ?ud parent. Comme nous sommes sur une feuille, si la variable ask_query est à faux, on sort nécessairement de `FindScope` par la ligne 5, ce qui signifie que ce n ?ud est la fin d'une branche qui nous conduit à une variable dans S. Ainsi, nous sommes sûrs d'avoir ask_query à vrai, ce qui signifie que le fils gauche du n ?ud parent a retourné un ensemble non vide et que le n ?ud parent est sur une branche qui mène vers une variable dans S.

Nous avons prouvé que chaque feuille est soit sur une branche qui mène vers une variable dans S, soit elle représente un fils d'un n ?ud qui mène vers une variable dans S. Ainsi, le nombre de n ?uds dans l'arbre est au plus deux fois le nombre de n ?uds dans les branches qui conduisent à une variable de S. Les branches peuvent être, au plus, de longueur $\log|X|$. Par conséquent, le nombre total de requêtes générées par `FindScope` est au plus $2 \cdot |S| \cdot \log|X|$, qui est dans $O(|S| \cdot \log|X|)$.

□

Theorem 1. *Etant donnés un biais B construit à partir d'un langage Γ de contraintes d'arité bornée, et un réseau cible C_T, QuAcq a besoin de $O(|C_T| \cdot (\log|X| + |\Gamma|))$ requêtes pour apprendre un réseau équivalent à C_T ou à s'effondrer, et de $O(|B|)$ requêtes pour prouver la convergence.*

Démonstration. A chaque fois qu'on qu'un exemple est classé négatif à la ligne 6 dans QuAcq, la portée d'une contrainte c_S dans C_T est retournée avec au plus $|S| \cdot \log|X|$ requêtes (proposition 2). Sachant que le langage Γ ne contient que des contraintes d'arité bornée, $|S|$ est également bornée et c_S est retournée avec $O(|\Gamma|)$ requêtes ou on atteint un état d'effondrement (proposition 1). Par conséquent, le nombre de requêtes nécessaires pour apprendre C_T ou de s'effondrer est en $O(|C_T| \cdot (\log|X| + |\Gamma|))$. La convergence est atteinte une fois B est réduit à vide grâce aux exemples classés positive à la ligne 6 dans QuAcq. Chacun de ces exemples conduit nécessairement au retrait d'au moins une contrainte de B suite à la façon dont on génère les exemples à la ligne 4. Cela donne un total de $O(|B|)$.

□

4 Langages Simples

Nous considérons des langages simples dans le but d'obtenir une estimation théorique de l'efficacité de QuAcq. Pour chacun de ces langages, nous analysons le nombre de requêtes nécessaires pour apprendre un réseau dans ce langage. Dans certains cas, nous montrons que QuAcq apprend les problèmes dans le langage donné avec un nombre de requêtes asymptotiquement optimal. Cependant, pour certains langages, un nombre de requêtes plus important est nécessaire dans le pire des cas. Dans notre analyse, nous considérons

le cas où la solution de C_L qui maximise le nombre de contraintes violées dans le biais B est choisie lorsqu'un exemple complet doit être généré (ligne 4 de QuAcq).

4.1 Langages pour lesquels QuAcq est optimal

Theorem 2. QuAcq *apprend un réseau booléen dans le langage* $\{=, \neq\}$ *avec un nombre de requêtes asymptotiquement optimal.*

Démonstration. (Esquisse.) Tout d'abord, nous donnons une borne inférieure sur le nombre de requêtes nécessaires pour apprendre un réseau dans ce langage. Prenons la restriction aux seules égalités. Dans une instance de ce langage, toutes les variables d'un composant connexe doivent être égales. L'ensemble de ces instances est donc isomorphique à l'ensemble des partitions de n objets, dont la taille est donnée par le *nombre de Bell* :

$$C(n+1) = \begin{cases} 1 & \text{si } n = 0 \\ \sum_{i=1}^{n} \binom{n}{i} C(n-i) & \text{si } n > 0 \end{cases} \quad (1)$$

Par un argument de la théorie de l'information, on peut donc déduire qu'au moins $\log C(n)$ requêtes sont nécessaires pour apprendre un tel problème. Celà implique une borne inférieure de $\Omega(n \log n)$, puisque $\log C(n) \in \Omega(n \log n)$ (voir [6] pour une preuve). Le langage $\{=, \neq\}$ est plus riche, et donc requiert au moins autant de requêtes.

Ensuite, nous considérons la requête soumise à l'utilisateur dans la ligne 6 de QuAcq, et nous faisons le compte des réponses *oui* et *non*. L'observation clé est qu'une instance de ce langage contient au plus $O(n)$ contraintes non redondantes. Pour chaque réponse *non* en ligne 6 de QuAcq, une nouvelle contrainte va être ajoutée à C_L. Seules les contraintes non redondantes sont découvertes de cette manière puisque la requête doit satisfaire C_L. Il s'en suit qu'au plus $O(n)$ telles requêtes reçoivent une réponse *non*, chacune entrainant $O(\log n)$ requêtes supplémentaires au travers de la procédure `FindScope`.

Maintenant nous pouvons borner le nombre de réponses *oui* à la ligne 6 de QuAcq. La même observation sur la structure de ce langage est à nouveau utile. Dans la version complete de la preuve, nous montrons que la requête qui maximise le nombre de contraintes violées dans le biais B tout en satisfaisant les contraintes de C_L viole au moins $\lceil |B|/2 \rceil$ contraintes de B. Donc, chaque requête dont la réponse est *oui* réduit de moitié le nombre de contraintes dans B. Il s'en suit que la requête soumise à la ligne 6 de QuAcq ne peut recevoir plus de $O(\log n)$ réponse *oui*. Le nombre total de requêtes est donc borné par $O(n \log n)$. □

Le même argument s'applique aux sous-langages $\{=\}$ et $\{\neq\}$ sur des domaines booléens. De plus, celà est toujours vrai pour $\{=\}$ sur des domaines arbitraires.

Corollary 1. QuAcq *apprend un réseau sur des domaines non bornés dans le langage* $\{=\}$ *avec un nombre de requêtes asymptotiquement optimal.*

4.2 Langages pour lesquels QuAcq est non-optimal

Tout d'abord, nous montrons qu'un réseau de contraintes booléennes dans le langage $\{<\}$ peut être appris avec $O(n)$ requêtes. Ensuite, nous montrons que QuAcq a besoin de $\Omega(n \log n)$ requêtes.

Theorem 3. *Les réseaux de contraintes booléennes dans le langage* $\{<\}$ *peuvent être appris en* $O(n)$ *requêtes.*

Démonstration. Notez que, pour décrire un tel problème, les variables peuvent être partitionnées en trois ensembles. i) Les variables qui doivent prendre la valeur 0 (c.-à-d. la partie gauche d'une contrainte $<$), ii) les variables qui doivent prendre la valeur 1 (c.-à-d. la partie droite d'une contrainte $<$), iii) et les variables libres (c.-à-d. les variables qui n'apparaissent dans aucune contrainte). Dans une première étape, on partitionne les variables en trois ensembles, G, D, I initialement vide et correspondant respectivement à *Gauche*, *Droite* et *Inconnu*. Durant cette étape, nous avons trois invariants :

1. Il n'existe pas de $x, y \in I$ tel que $x < y$ appartient au réseau cible C_T
2. $x \in G$ *ssi* il existe $y \in I$ et $x < y$ est dans le réseau cible C_T
3. $x \in D$ *ssi* il existe $y \in I$ et $y < x$ dans le réseau cible C_T

Nous passons par toutes les variables du problème, une à la fois. Soit x la dernière variable tirée. Nous posons une requête à l'utilisateur où x, ainsi que toutes les variables dans I sont mises à 0, et toutes les variables de D sont mises à 1 (variables dans G restent sans affectation). Si la réponse est *oui*, alors il n'y a pas de contrainte entre x et une variable $y \in I$, donc x est mise dans I sans toucher aux invariants. Autrement, x est soit impliquée dans une contrainte $y < x$ ou $x < y$ avec $y \in I$. Afin de décider quelle valeur prendre pour x, une deuxième requête est générée où la valeur de x passe à 1 et toutes les valeurs des autres variables restent inchangées. Si la réponse à cette requête est *oui*, alors la première hypothèse est vraie et nous mettons x dans D, autrement, nous mettons x dans G. Là encore, les invariants sont vérifiés.

A la fin de cette étape, on peut dire que les variables dans I n'ont pas de contraintes entre eux. Cependant, elles peuvent être impliquées dans des contraintes avec des variables de G ou de D. Dans la deuxième étape, nous passons en revue les variables $x \in I$, et nous générons une requête où toutes les variables de G sont mises à 0, toutes les variables de D sont mises à 1 et x est mise à 0. Si la réponse est *non*, on peut dire qu'il existe une contrainte $y < x$ avec $y \in G$ et donc x est ajoutée à D (et retirée de I). Sinon, nous posons la même requête, mais avec la valeur de x qui passe à 1. Si la réponse est *non*, il existe alors un $y \in D$ tel que $x < y$ appartient au réseau cible, donc x est ajoutée à D (et retirée de I). Pour le dernier cas où la réponse est *oui* pour les deux requêtes, on peut conclure que x n'appartient à aucune contrainte du réseau cible. Lorsque chaque variable a été examinée de cette manière, les variables restantes dans I ne sont pas impliquées dans les contraintes du réseau cible.

□

Theorem 4. QuAcq *n'apprend pas les réseaux de contraintes booléennes dans le langage* $\{<\}$ *avec un nombre minimal de requêtes.*

Démonstration. D'après le théorème 3, ces réseaux peuvent être appris en $O(n)$ requêtes. Ce type de réseaux peut contenir jusqu'à $n-1$ contraintes non redondantes. QuAcq apprend une contrainte à la fois, où chaque appel à `FindScope` prend $\Omega(\log n)$ requêtes. Par conséquent, QuAcq a besoin de $\Omega(n \log n)$ requêtes. □

5 Evaluation Expérimentale

Pour évaluer la faisabilité de notre approche, nous avons produit des résultats expérimentaux avec QuAcq sur une machine Xeon E5462@2.80GHz Intel avec 16 Go de RAM, et sur plusieurs problèmes.

Random. Nous avons généré des réseaux de contraintes binaires de façon aléatoire, avec 50 variables, des domaines de taille 10, et m contraintes binaires. Les contraintes binaires sont générées à partir du langage $\Gamma = \{\geq, \leq, <, >, \neq, =\}$. QuAcq prend en entrée un biais B contenant le graphe complet de 7350 contraintes binaires de Γ. Les résultats présentent la moyenne de 100 exécutions de QuAcq avec deux densités différentes : $m = 12$ (sous-contraint) et $m = 122$ (transition de phase).

Règles de Golomb. (prob006 in [8]) Le réseau cible est formulé avec m variables correspondant aux m marques de la règle, et des contraintes d'arité variable. Nous présentons les résultats sur des règles de taille 8. QuAcq est initialisé avec un biais de 770 contraintes basé dans le langage $\Gamma = \{|x_i - x_j| \neq |x_k - x_l|, |x_i - x_j| = |x_k - x_l|, x_i < x_j, x_i \geq x_j\}$, comprenant des contraintes binaires, ternaires [3] et quaternaires.

Problème du Zèbre. Le problème du Zèbre de Lewis Carroll a une solution unique. Le réseau cible est formulé avec 25 variables, des domaines de 5 valeurs, 5 cliques de contraintes $\neq$ et 14 contraintes supplémentaires figurant dans la description du problème. Pour tester QuAcq sur ce problème, nous l'avons initialisé avec un biais B de 4450 contraintes unaires et binaires d'un langage Γ de 24 contraintes (contraintes arithmétiques et des contraintes de distance).

Sudoku. Le réseau cible du problème de Sudoku est formulé avec 81 variables, des domaines de taille 9 et 810 contraintes $\neq$ sur les lignes, les colonnes et les carrés. Nous avons initialisé QuAcq avec un biais B de 6480 contraintes binaires du langage $\Gamma = \{=, \neq\}$.

Pour chaque problème, nous rapportons la taille $|C_L|$ du réseau appris (qui peut être plus petit que le réseau cible en raison des contraintes redondantes), le nombre total de requêtes $\#q$, le nombre de requêtes d'appartenance (complètes) $\#q_c$ (c.-à-d. les requêtes de la ligne 6 de QuAcq), la taille moyenne des requêtes $\bar{q}$, et le temps moyen nécessaire pour générer une requête (en secondes).

5.1 QuAcq et convergence

Pour assurer une convergence rapide, nous avons besoin de requête classée positive et qui réduit au maximum le biais B. Le meilleur moyen d'avoir de telles requêtes est de générer un exemple à la ligne 4 de QuAcq qui maximise les contraintes violées dans B. Nous avons implémenté l'heuristique *max* pour générer une solution du C_L courant qui viole un nombre maximum de contraintes de B. Toutefois, cette heuristique peut être couteuse puisqu'elle résout un problème d'optimisation. Nous avons ensuite ajouté des bornes de 1 et 10 secondes pour l'heuristique *max*, pour obtenir respectivement deux variantes *max-1* et *max-10*. Nous avons également implémenté une heuristique moins couteuse que nous appelons *sol*. Elle résout simplement le C_L courant et s'arrête à la première solution qui viole au moins une contrainte de B.

Notre première expérimentation était de comparer *max-1* et *max-10* sur de gros problèmes. On observe que les performances lorsqu'on utilise *max-1* n'est pas aussi pire en nombre de requêtes que *max-10*. Par exemple, sur rand_122, on a $\#q = 1074$ pour *max-1* et $\#q = 1005$ pour *max-10*. Le temps moyen pour générer une requête étant 0.14 secondes pour *max-1* et 0,86 pour *max-10* avec, respectivement, une borne de

3. Les contraintes ternaires sont obtenues lorsque $i = k$ ou $j = l$ dans $|x_i - x_j| \neq |x_k - x_l|$

TABLE 1 – Résultats de QUACQ (convergence).

		$\|C_L\|$	$\#q$	$\#q_c$	$\bar{q}$	temps
rand_12	*max-1*	12	196	34	24.04	0.23
	sol	12	286	133	33.22	0.09
rand_122	*max-1*	86	1074	94	13.90	0.14
	sol	83	1062	120	15.64	0.06
Golomb	*max-1*	91	488	101	5.12	0.32
	sol	138	709	153	5.31	0.25
Zèbre	*max-1*	60	638	64	8.22	0.15
	sol	60	634	63	8.20	0.02
Sudoku	*max-1*	810	8645	821	20.58	0.16
	sol	810	9593	815	20.84	0.06

1 et 10 secondes. Nous avons donc décidé de ne pas présenter les résultats avec *max-10*.

Le tableau 1 présente les résultats obtenus avec QUACQ pour apprendre un réseau de contraintes jusqu'à convergence en utilisant les heuristiques *max-1* et *sol* . Une première observation est que l'heuristique *max-1* nécessite généralement moins de requêtes que *sol* pour atteindre la convergence. Cela est particulièrement vrai pour rand_12, qui est très creux, et Golomb, qui contient de nombreuses contraintes redondantes. Si on regarde de plus près, cette différence s'explique principalement par le fait que *max-1* nécessite beaucoup moins de requêtes complètes positives que *sol* pour réduire B à vide et prouver la convergence (rand_12 : 22 requêtes complètes positives pour *max-1* et 121 pour *sol*). Mais en général, *sol* n'est pas aussi mauvais que nous pouvions espérer. La raison en est que, exceptés les réseaux très creux, le nombre de contraintes de B violées «par chance» avec *sol* est assez grand. La deuxième observation est que lorsque le réseau contient un grand nombre de contraintes redondantes, *max-1* converge sur un réseau plus petit que *sol* . Nous avons observé cela sur Golomb, et d'autres problèmes non rapportés ici. La troisième observation est que la taille moyenne des requêtes est toujours significativement plus petite que le nombre de variables du problème. Une dernière observation est que *sol* est très rapide en termes de génération de requêtes. Il est donc envisageable d'utiliser QUACQ sur des problèmes réels de taille importante.

Notre deuxième expérimentation consiste à évaluer l'effet de la taille du biais sur le nombre de requêtes. Sur le problème du zèbre, nous avons initialisé QUACQ avec des biais de différentes tailles et stocké le nombre de requêtes pour chaque exécution. La figure 1 montre que lorsque $|B|$ augmente, le nombre de requêtes suit une échelle logarithmique. Un résultat prometteur car cela signifie que l'acquisition de contraintes avec des biais expressifs (de grande taille) passe à l'échelle.

QUACQ a deux principaux avantages comparant à CONACQ. Le premier est la taille moyenne des requêtes $\bar{q}$ avec les requêtes partielles, qui sont probablement plus facile à classifier pour l'utilisateur. Le second avantage est le temps que nécessite QUACQ pour générer des requêtes. CONACQ.2 a besoin de trouver des exemples qui violent exactement une contrainte du biais pour atteindre la convergence (ce qui peut être coûteux à calculer). D'autre part, QUACQ peut utiliser des heuristiques pas très couteuses comme *max-1* et *sol* pour générer des requêtes.

5.2 QuAcq as a solver

CONACQ.2 et MODELSEEKER ont besoin d'exemples positifs pour apprendre un réseau de contraintes. Par contre, QUACQ peut apprendre sans exemple positif complet. Cette propriété peut être très utile lorsque le problème n'a pas été préalablement résolu. Nous avons simplement besoin de sortir de QUACQ dès qu'un exemple complet est classé par l'utilisateur comme positif. Nous avons évalué cette fonctionnalité en résolvant une séquence de 5 instances de Sudoku, des grilles avec des cases pré-remplies. Pour chaque grille, on sort de QUACQ lorsque la solution est trouvée. Comme le but n'est pas d'atteindre la convergence, nous avons remplacé l'heuristique *max-1* par un *min-1*, une heuristique qui tente de satisfaire autant que possible les contraintes de B, avec une borne d'une seconde. Chaque exécution prend en entrée le C_L et le B résultants de l'exécution précédente, vu que le réseau partiellement appris d'une exécution donnée reste valide comme point de départ d'une résolution d'une autre grille. Le nombre de requêtes nécessaires pour résoudre chacune des 5 grilles est, respectivement, 3859, 1521, 687, 135 et 34. La taille de C_L après chaque exécution est, respectivement, 340, 482, 547, 558, et 561. Nous notons, que pour la première grille, où QUACQ commence avec un biais complet, nous trouvons la solution en seulement 44% de requêtes nécessaires pour que QUACQ converge (voir le tableau 1). Après cela, chaque exécution de QUACQ sur les grilles qui restent, a besoin de moins de requêtes pour trouver une solution vu que C_L se rapproche de plus en plus du réseau cible.

6 Conclusion

Nous avons proposé QUACQ, un algorithme qui apprend un réseau de contraintes en demandant à l'utilisateur de classifier des affectations partielles en tant que positive ou négative. Chaque fois que l'algorithme reçoit un exemple négatif, l'algorithme dé-

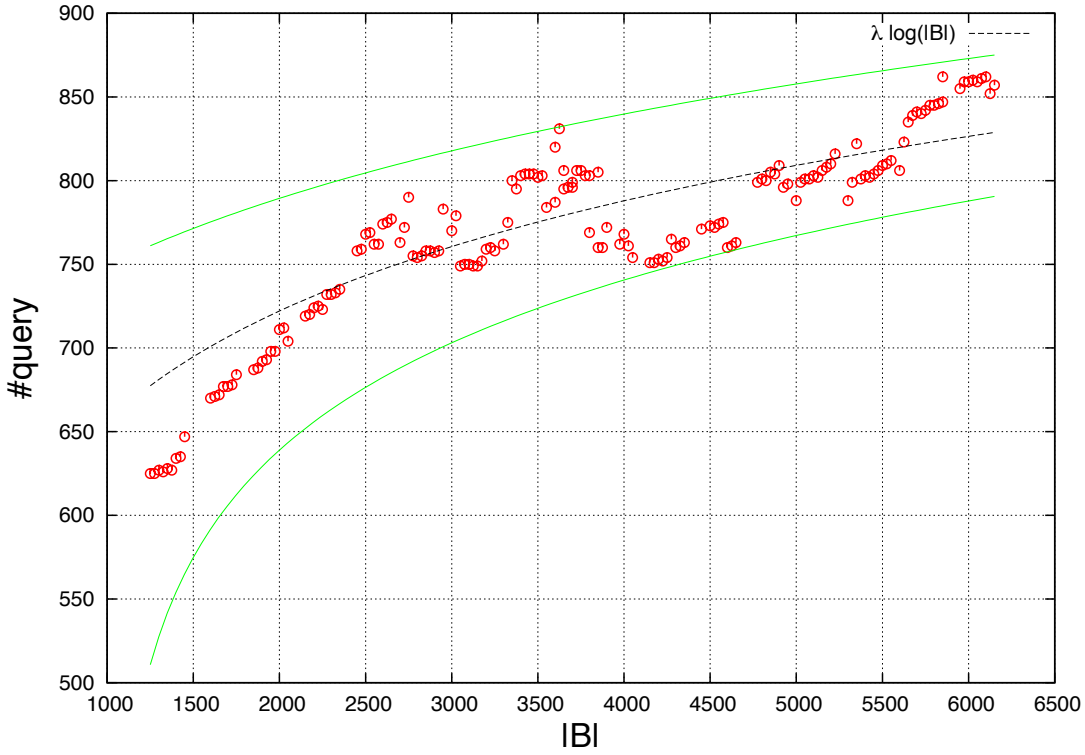

FIGURE 1 – QUACQ avec différentes tailles de biais sur le problème du zèbre

couvre une contrainte au problème en effectuant un nombre logarithmique de requêtes. Nous avons démontré que QUACQ est optimal pour certains langages de contraintes. Notre approche comporte quelques avantages par rapport aux travaux existants. Premièrement, elle converge toujours sur le réseau de contraintes cible en un nombre polynomial de requêtes. Deuxièmement, les requêtes sont souvent plus courtes que les requêtes d'appartenance et sont plus faciles à répondre par un utilisateur. Troisièmement, contrairement aux autres techniques, l'utilisateur n'a pas à fournir des exemples positifs pour converger. Cette dernière propriété peut être très utile lorsque le problème n'a pas été préalablement résolu. Nos expérimentations démontrent que générer de bonnes requêtes par QUACQ n'est pas difficile d'un point de vue computationnel et que lorsque le biais croît, le nombre de requêtes augmente de façon logarithmique. Ces résultats sont prometteurs pour l'utilisation de QUACQ sur des problèmes réels. Cepencdant, les problèmes avec des réseaux de contraintes denses (tel le Sudoku) requièrent un trop grand nombre de requêtes pour être répondu par un humain. Il serait intéressant d'utiliser MODELSEEKER pour apprendre rapidement des contraintes globales et d'utiliser QUACQ pour finaliser le modèle.

Références

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

[2] N. Beldiceanu and H. Simonis. A model seeker : Extracting global constraint models from positive examples. In *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'12), LNCS 7514, Springer–Verlag*, pages 141–157, Quebec City, Canada, 2012.

[3] C. Bessiere, R. Coletta, E. Freuder, and B. O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'04), LNCS 3258, Springer–Verlag*, pages 123–137, Toronto, Canada, 2004.

[4] C. Bessiere, R. Coletta, F. Koriche, and B. O'Sullivan. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In *Proceedings of the European Conference on Machine Learning (ECML'05), LNAI 3720, Springer–Verlag*, pages 23–34, Porto, Portugal, 2005.

[5] C. Bessiere, R. Coletta, B O'Sullivan, and M. Paulin. Query-driven constraint acquisition. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 44–49, Hyderabad, India, 2007.

[6] N.G. De Bruijn. *Asymptotic Methods in Analysis*. Dover Books on Mathematics. Dover Publications, 1970.

[7] E.C. Freuder and R.J. Wallace. Suggestion strategies for constraint-based matchmaker agents. In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP'98), LNCS 1520, Springer–Verlag*, pages 192–204, Pisa, Italy, 1998.

[8] I.P. Gent and T. Walsh. Csplib : a benchmark library for constraints. http ://www.csplib.org/, 1999.

[9] U. Junker. Quickxplain : Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172, San Jose CA, 2004.

[10] A. Lallouet, M. Lopez, L. Martin, and C. Vrain. On learning constraint problems. In *Proceedings of the 22nd IEEE International Conference on Tools for Artificial Intelligence (IEEE-ICTAI'10)*, pages 45–52, Arras, France, 2010.

[11] M. Paulin, C. Bessiere, and J. Sallantin. Automatic design of robot behaviors through constraint network acquisition. In *Proceedings of the 20th IEEE International Conference on Tools for Artificial Intelligence (IEEE-ICTAI'08)*, pages 275–282, Dayton OH, 2008.