Réification de Contraintes Tables

Minh Thanh Khong[1]***Christophe Lecoutre**[2] **Yves Deville**[1] **Pierre Schaus**[1]

[1] ICTEAM, Université catholique de Louvain, Belgium
[2] CRIL-CNRS UMR 8188, Université d'Artois, F-62307 Lens, France
{minh.khong,yves.deville,pierre.schaus}@uclouvain.be lecoutre@cril.fr

Résumé

La réification d'une contrainte c consiste à lui associer une variable booléenne b telle que la satisfaction ou non satisfaction de c corresponde à la valeur de vérité de b, ce que l'on peut noter $c^{reif} : c \Leftrightarrow b$. La réification s'avère utile, entre autres, pour combiner logiquement des contraintes et comptabiliser le nombre de contraintes réifiées satisfaites. Étant donné que les contraintes tables jouent un rôle important en programmation par contraintes, dans cet article, nous nous intéressons à leur réification, en proposant notamment un algorithme de filtrage établissant la cohérence d'arc généralisée, sans sur-coût de mémoire. Nous montrons également comment la décomposition de la contrainte `soft-regular` en contraintes tables ternaires réifiées se traduit par une mesure de violation originale.

Abstract

Reifiyng a constraint c consists in associating a Boolean variable b with c such that c is satisfied if and only if b is true, which can be denoted by $c^{reif} : c \Leftrightarrow b$. Reification is useful for logically combining constraints and counting how many reified constraints can be satisfied. Since table constraints play an important role within constraint programming, in this paper, we are interested in their reification. First, we present a filtering algorithm to establish generalized arc consistency on reified table constraints, with no spatial overhead. Next, we show how the decomposition of the soft global constraint `soft-regular` into reified ternary table constraints involves an original violation measure.

1 Introduction

La réification d'une contrainte c est utilisée pour refléter sa valeur de vérité dans une variable booléenne b. Par conséquent, réifier une contrainte c consiste à remplacer c par sa forme réifiée $c^{reif} : c \Leftrightarrow b$, avec maintenant la possibilité que c puisse être violée : b est vrai si et seulement si la contrainte c est satisfaite. Les contraintes réifiées sont utiles pour appliquer des connecteurs logiques entre les contraintes ou exprimer qu'un certain nombre de contraintes doit être satisfait, par exemple, en additionnant les variables (interprétées comme 0-1) associées aux contraintes réifiées [7].

Les contraintes tables, c.-à-d., les contraintes définies en énumérant explicitement les combinaisons acceptées (ou interdites) de valeurs pour les variables de leurs portées, jouent un rôle important en programmation par contraintes. En effet, elles peuvent être considérées comme un service à usage général, offert par les solveurs de contraintes, pour exprimer toute sorte de contraintes, avec la consommation en espace requis comme seule limite à cette approche. Les contraintes tables peuvent être utiles pour combiner efficacement certaines parties de problèmes (par exemple, la combinaison de contraintes fortement liées), et apparaissent naturellement dans de nombreux domaines tels que la configuration de produits et les bases de données. De nombreux algorithmes ont été proposés au fil des ans pour filtrer les contraintes tables [2, 15, 9, 20, 13, 11, 14, 6], ou certaines de leurs formes compactes [12, 4, 10, 18].

Ces dernières années, un certain nombre de travaux ont été proposés pour la réification des contraintes globales [8, 1, 7], ne comprenant toutefois pas les contraintes tables. Dans cet article, nous étudions la réification des contraintes tables, notamment en décrivant un algorithme pour établir la cohérence d'arc généralisée ou *Generalized Arc Consistency* (GAC), suivant la technique de réduction tabulaire simple ou *Simple Tabular reduction* (STR) [20, 13]. Un résultat intéressant de notre travail est que la porte est ouverte à la réification de tous types de contraintes, sous réserve qu'il soit possible en pratique de les reformuler

*Papier doctorant : Minh Thanh Khong[1] est auteur principal.

en contraintes tables.

Nous montrons aussi comment les contraintes tables réifiées peuvent être exploitées dans le cadre de la contrainte `soft-regular`. Les mesures de violation existantes pour `soft-regular` sont basées sur le concept de distance en terme de variables ou d'opérations d'édition [21]. Une mesure de violation naturelle alternative consiste à compter le nombre de fois qu'une transition inexistante dans l'automate de la contrainte doit être utilisée. Cette mesure de violation est directement obtenue après décomposition d'une contrainte `soft-regular` en contraintes tables ternaires réifiées.

Le papier est organisé comme suit. Tout d'abord, la section 2 introduit quelques pré-requis techniques. Ensuite, en section 3, nous présentons un algorithme de filtrage pour les contraintes tables réifiées, et, en section 4, nous montrons comment les contraintes tables réifiées peuvent être utiles pour traiter `soft-regular`. Avant de conclure, quelques résultats expérimentaux préliminaires sont donnés dans la section 5.

2 Pré-requis techniques

2.1 Réseaux de contraintes

Un problème de satisfaction de contraintes (CSP) P, appelé également un réseau de contraintes, est composé d'un ensemble de n variables, $X = \{x_1, \ldots, x_n\}$, et d'un ensemble de e contraintes, $C = \{c_1, \ldots, c_e\}$. Chaque variable x a un domaine associé, noté $dom(x)$, qui contient l'ensemble des valeurs qui peuvent être assignées à x. La taille maximale d'un domaine pour un CSP donné sera notée d. Chaque contrainte c se compose d'un ensemble ordonné de variables, appelé portée (ou scope) de c et noté par $scp(c)$. Chaque contrainte c est définie par une relation, notée $rel(c)$, qui contient les combinaisons autorisées de valeurs pour les variables de $scp(c)$. L'arité d'une contrainte c est la taille de $scp(c)$, et sera généralement désigné par r.

Étant donné une séquence $\langle x_1, \ldots, x_i, \ldots, x_r \rangle$ de r variables, un r-tuple τ pour cette séquence de variables est une séquence de valeurs $\langle a_1, \ldots, a_i, \ldots, a_r \rangle$, où la valeur individuelle a_i est également noté $\tau[x_i]$ ou, lorsqu'il n'y a pas d'ambiguïté, $\tau[i]$. Soit c une contrainte d'arité r, un r-tuple τ est *valide* sur c ssi $\forall x \in scp(c), \tau[x] \in dom(x)$. L'ensemble des tuples valides sur c est $val(c) = \Pi_{x \in scp(c)} dom(x)$. Un tuple τ est *autorisé* par une contrainte c ssi $\tau \in rel(c)$; on dit aussi que c accepte τ. Un *support* sur c est un tuple valide sur c qui est également accepté par c. Une contrainte table positive (resp. négative) est une contrainte dont la sémantique est définie en extension en énumérant l'ensemble des tuples *autorisés* (resp. *interdits*). Cette table (ensemble) est dénoté $table^{init}(c)$.

Une contrainte c est dite *entailed* (resp. *disentailed*) si tous les tuples τ dans $val(c)$ sont acceptés (resp., non acceptés) par c ; en d'autres termes, c est toujours satisfaite (resp. violée).

Une contrainte c est *Generalized Arc-Consistent* (GAC) ssi $\forall x \in scp(c)$, $\forall a \in dom(x)$, il existe un *support* de (x, a) sur c, c.-à-d., un tuple valide τ sur c tel que τ est accepté par c et $\tau[x] = a$.

Une solution pour un CSP est une affectation d'une valeur à chaque variable telle que toutes les contraintes soient satisfaites. Un CSP est dit *cohérent* s'il admet au moins une solution.

2.2 Réification de Contraintes Tables

La réification d'une contrainte c est obtenue en associant une variable booléenne b à c. On obtient alors une contrainte réifiée $c^{reif} : c \Leftrightarrow b$. La sémantique opérationnelle d'un algorithme de filtrage (propagateur) pour la réification d'une telle contrainte est donnée par les règles suivantes :

- si b est à 1, alors c doit être satisfaite,
- si b est à 0, alors c doit être violée,
- si c devient *entailed*, alors b doit être à 1,
- si c devient *disentailed*, alors b doit être à 0.

Pour traiter une contrainte réifiée, nous avons besoin d'un propagateur pour c, un propagateur pour $\neg c$, et nous devons détecter quand c devient entailed ou disentailed. La proposition ci-dessous montre comment on peut déterminer qu'une contrainte table devient *entailed* et *disentailed*.

Proposition 1 *Soit $table(c)$ l'ensemble des supports courants de c, c.-à-d., nous avons $table(c) = table^{init}(c) \cap val(c)$. Nous avons :*

- *c est entailed ssi $|table(c)| = |val(c)|$,*
- *c est disentailed ssi $|table(c)| = 0$.*

Quand une contrainte table devient *entailed*, tous les tuples valides dans c sont des supports de c. L'exemple 1 montre une contrainte qui est *entailed*.

Exemple 1 Étant donné une contrainte table positive c telle que $scp(c) = \{x_1, x_2, x_3\}$ et $table(c)$ défini par :

x_1	x_2	x_3
0	0	0
0	1	0
1	0	0
1	1	0

Si $dom(x_1) = dom(x_2) = \{0, 1\}$ et $dom(x_3) = \{0\}$, alors c est *entailed* parce que $|table(c)| = 4 = |val(c)| = |dom(x_1) \times dom(x_2) \times dom(x_3)|$.

3 Un Algorithme de Filtrage pour GAC

Dans cette section, nous présentons un algorithme de filtrage qui applique GAC sur une contrainte table réifiée donnée. Le principe est de mettre à jour la table de la contrainte réifiée, à chaque appel de l'algorithme de filtrage, de manière à éliminer les tuples qui sont devenus invalides, puis vérifier l'état *entailment* (ou *disentailment*). Pour la gestion de la table, nous utilisons la technique bien connue appelée STR (*Simple Tabular Reduction*) [20, 13]. Ci-dessous, nous présentons les structures de données utilisées par STR, puis nous présentons l'algorithme de filtrage.

3.1 Structures de donnés

La table associée à une contrainte table c est notée $c.table$. Cette table est représentée par une tableau de tuples indexés de 1 à $c.table.length$ qui représente la taille de la table (c.-à-d. le nombre de tuples autorisés). La complexité en espace dans le pire des cas pour cet ensemble est $O(rt)$ où $t = c.table.length$ et r est l'arité de c.

Les algorithmes basés sur STR ont été développés pour le filtrage des contraintes tables. Ils sont parmi les algorithmes les plus efficaces pour les contraintes tables, en particulier parce que leurs structures de données permettent une restauration peu coûteuse lors des retours-arrières. Le principe de STR est de diviser une table en différents ensembles de telle sorte que chaque tuple soit exactement membre d'un ensemble ; ce qui correspond à l'utilisation des *sparse sets* [3, 5]. L'un de ces ensembles contient tous les tuples qui sont couramment valides : les tuples de cet ensemble constituent le contenu de la *table courante*. Pour simplifier, les structures de données liées au backtracking ne sont pas détaillées dans cet article (voir [13]).

Les tableaux suivants donnent accès aux ensembles disjoints de tuples valides et invalides de $c.table$:

- $c.position$ est un tableau de taille $t = c.table.length$ qui fournit un accès indirect aux tuples de $c.table$. À un moment donné, les valeurs de $c.position$ sont une permutation de $\{1, 2, ..., t\}$. Le i^{eme} tuple de c est $c.table[c.position[i]]$. Pour simplifier, ce tuple est noté $\tau(c, i)$.
- $c.limit$ est la position du dernier tuple courant de $c.table$. La table courante de c est exactement composée de $c.limit$ tuples. Les valeurs dans $c.position$ aux indices allant de 1 à $c.limit$ sont les positions des tuples courants de c.

3.2 STR-Reif

Nous décrivons maintenant STR-Reif, un algorithme pour imposer GAC sur une contrainte table réifiée $c^{reif} : c \Leftrightarrow b$. Cet algorithme est une modification de STR2 [13] appliquée aux contraintes tables réifiées. A noter qu'il peut être utilisé pour une contrainte table positive réifiée ($c.positive = true$) mais aussi pour une contrainte table réifiée négative ($c.positive = false$).

Pour une vérification rapide de la validité des tuples, nous réutilisons l'ensemble S^{val} de STR2 [13] : il contient les variables dont le domaine a été réduit depuis l'invocation précédente de STR-Reif. A la fin de l'invocation de STR-Reif, pour chaque variable $x \in scp(c)$, chaque tuple τ tel que $\tau[x] \notin dom(x)$ a été éliminé de la table courante de c. Au prochain appel, s'il n'y a pas eu de retour-arrière et si $dom(x)$ n'a pas changé, alors $\tau[x] \in dom(x)$ pour chaque tuple courant τ de c. Pour cette raison, lors de la vérification de la validité des tuples, seules les variables présentes dans S^{val} sont vérifiées par l'algorithme 2. A noter que nous utilisions $c.lastSize[x]$ (taille de $dom(x)$ la dernière fois que STR-Reif a été appelé) pour construire S^{val} (lignes 6-10 de l'algorithme 1).

En lignes 1-4, STR-Reif vérifie tout d'abord si b est assignée. Si elle est assignée à 1 (resp, 0), nous avons besoin de poster le propagateur pour assurer c (resp, $\neg c$). Dans ce cas, il n'y a plus d'appel au propagateur de c^{reif}. Il existe de nombreux algorithmes pour cette tâche, par exemple, STR2 [13] pour les contraintes tables positives et STR-Ni [16] pour les contraintes tables négatives. Lorsque $dom(b) = \{0, 1\}$, nous devons vérifier l'entailment ou le disentailment de c (lignes 20-26), après avoir mis à jour la table courante (lignes 11-19).

Proposition 2 *STR-Reif établit GAC sur toute contrainte table réifiée.*

Preuve. Lorsque b est assignée, STR-Reif assure GAC sur la contrainte c si b est 1 ou $\neg c$ si b est 0 (en supposant que les propagateurs pour c et $\neg c$ appliquent GAC). Autrement, STR-Reif garantit de filtrer $dom(b)$ lorsque cela est possible, selon la proposition 1.

Proposition 3 *Pour une contrainte table positive, la complexité temporelle dans le pire de cas de STR-Reif est $O(r't' + r''d)$ où $r' = |S^{val}|$ désigne le nombre de variables pour lesquelles les opérations de validité doivent être effectuées, t' désigne la taille de la table courante de c et r'' le nombre de variables avec des domaines non singletons.*

Preuve. Effectuer le teste de validité se fait en $O(r')$ comme on peut le voir dans l'algorithme 2, puisque

Algorithm 1: STR-Reif($c^{reif}(c, b)$)

```
   // On teste d'abord si b est mis à 0 ou 1
 1 if dom(b)={1} then
 2 |  post(c)
 3 else if dom(b)={0} then
 4 |  post(¬c)
 5 else // dom(b) = {0,1}
 6 |  S^val ← ∅
 7 |  foreach x ∈ scp(c) : |dom(x)| ≠ c.lastSize[x]
   |  do
 8 |  |  S^val ← S^val ∪ {x}
 9 |  |  c.lastSize[x] ← |dom(x)|
10 |  end
11 |  i ← 1
12 |  while i ≤ c.limit do
13 |  |  if c.isValidTuple(S^val, τ(c,i)) then
14 |  |  |  i ← i + 1
15 |  |  else
16 |  |  |  c.swapTuple(i, c.limit)
17 |  |  |  c.limit ← c.limit − 1
18 |  |  end
19 |  end
   |  // Tester entailment ou disentailment
20 |  if c.limit = 0 then
21 |  |  if c.positive then dom(b) ← {0}
22 |  |  else dom(b) ← {1}
23 |  else if c.limit = |val(c)| then
24 |  |  if c.positive then dom(b) ← {1}
25 |  |  else dom(b) ← {0}
26 |  end
27 end
```

seules les variables présentes dans S^{val} sont considérées. Les boucles en lignes 7-10 et 12-19 de l'algorithme 1 sont $O(r), O(r't')$, respectivement. Lorsque la table est positive, STR2 peut être sollicité (lorsque b est à 1), et sa complexité est en $O(r't' + r''d)$. Donc, la complexité temporelle dans le pire de cas de l'algorithme est $O(r't' + r''d)$.

Notez que la complexité en espace dans le pire de cas de STR-Reif est le même que STR2, qui est, $O(n + rt)$ [13].

4 Décomposition de `soft-regular`

Dans cette section, nous considérons la décomposition de la contrainte `soft-regular` en contraintes ternaires positives réifiées, et introduisons dans ce contexte une nouvelle mesure de violation.

Algorithm 2: isValidTuple(S^{val} : variables, τ : tuple) : Boolean

```
1 foreach variable x ∈ S^val do
2 |  if τ[x] ∉ dom(x) then
3 |  |  return false
4 |  end
5 end
6 return true
```

4.1 Contrainte `regular`

La contrainte globale `regular` a été introduite dans [19, 21]. Elle est définie sur une séquence de variables et indique qu'une séquence de valeurs prises par ces variables doivent appartenir à un langage régulier spécifié. Cette contrainte peut être appliquée, par exemple, pour des problèmes de *rostering* ou *car sequencing*.

Avant d'introduire la contrainte `regular`, nous rappelons la définition d'un automate fini déterministe (DFA).

Un DFA est décrit par un 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ où Q est un ensemble d'états, Σ un alphabet, $\delta : Q \times \Sigma \to Q$ une fonction partielle de transition, $q_0 \in Q$ l'état initial, et $F \subseteq Q$ l'ensemble des états finaux (ou acceptants). Etant donné un mot (une séquence de symboles) en entrée, l'automate démarre dans son état initial q_0 et traite les symboles les uns après les autres, appliquant la fonction de transition δ à chaque étape afin de mettre à jour l'état courant. Le mot est accepté ssi le dernier état atteint appartient à l'ensemble des états finaux F. Les mots acceptés par M sont dits appartenir au langage définit par M, noté $L(M)$. Les langages reconnus par des DFAs sont précisément les langages réguliers.

Par exemple, avec le DFA M représenté par la figure 1, on peut observer que le mot *aaabbaa* est accepté par $L(M)$ tandis que le mot *caab* ne l'est pas.

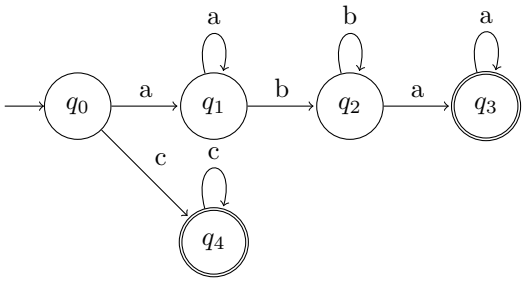

FIGURE 1 – Un DFA avec l'état initial q_0 et les états finals q_3 et q_4.

Définition 1 (Contrainte `regular`) *Soit* $M = (Q, \Sigma, \delta, q_0, F)$ *un DFA et* $X = \langle x_1, \ldots, x_r \rangle$ *une*

séquence de variables. La contrainte $\texttt{regular}(X, M)$ *accepte tous les tuples dans* $\{\langle d_1, \ldots, d_r\rangle \mid d_i \in dom(x_i), \forall i \in 1..r \wedge d_1 \ldots d_r \in L(M)\}$.

Exemple 2 Considérons une séquence de 4 variables $X = \langle x_1, x_2, x_3, x_4\rangle$ avec $dom(x_i) = \{a, b, c\}, \forall i \in 1..4$ et l'automate M représenté par la figure 1. Les tuples (c, c, c, c) et (a, a, b, a) sont acceptés par la contrainte $\texttt{regular}(X, M)$ tandis que (c, a, a, b) et (c, c, a, c) ne le sont pas.

4.2 Décomposition en contraintes tables réifiées

Nous proposons maintenant de décomposer la contrainte `soft-regular` en contraintes ternaires positives réifiées. Cela implique une nouvelle mesure de violation, appelé "dfa", qui est définie comme le nombre de fois où on doit utiliser une transition inexistante dans l'automate de la contrainte. Plus formellement, nous définissons l'ensemble (théorique) des états de longueur $r + 1$ pour un DFA $M = (Q, \Sigma, \delta, q_0, F)$ comme :

$$\mathcal{S}^r_M = \{\langle s_0, \ldots, s_r\rangle \in Q^{r+1} | s_0 \in \{q_0\} \wedge s_r \in F\}$$

Le coût d'un r-tuple τ par rapport à un DFA $M = (Q, \Sigma, \delta, q_0, F)$ et une séquence des états $S = \langle s_0, \ldots, s_r\rangle \in \mathcal{S}^r_M$ est défini comme :

$$\text{dfa-cost}_M(\tau, S) = \#\{i \in 1..r | (s_{i-1}, \tau[i], s_i) \notin \delta\}$$

Le coût d'un r-tuple τ par rapport à un DFA M est :

$$\text{dfa-cost}_M(\tau) = \min_{S \in \mathcal{S}^r_M} \text{dfa-cost}_M(\tau, S)$$

Nous pouvons appliquer cette définition à une contrainte $\texttt{regular}(X, M)$, avec $X = \langle x_1, \ldots, x_r\rangle$, en considérant que le coût de la violation d'un r-tuple τ sur X est $\text{dfa-cost}_M(\tau)$ donné par la définition ci-dessus.

Exemple 3 Lorsque cette nouvelle mesure est utilisée pour la contrainte `regular` de l'exemple 2, nous obtenons $\text{dfa-cost}_M(c, a, a, b) = 2$, en considérant par exemple la séquence des états suivante $\langle q_0, q_4, q_1, q_1, q_3\rangle$.

La contrainte `soft-regular` peut maintenant être définie en considérant la mesure de violation "dfa". Précisément, nous pouvons transformer une contrainte $\texttt{regular}(X, M)$ en une contrainte $\texttt{soft-regular}^{dfa}(X, M, z)$ où z est la variable de coût dont la valeur est définie par la mesure de violation "dfa". Nous pouvons calculer le coût de violation en décomposant la contrainte comme suit. D'abord, nous introduisons $r + 1$ nouvelles variables y_i $(i \in 0..r)$ tel que $dom(y_0) = \{q_0\}$, $dom(y_i) = Q, \forall i \in 1..r - 1$ et $dom(y_r) = F$. Nous introduisons également r variables booléennes z_i, $i \in 1..r$, pour l'objectif de réification. Ensuite, nous introduisons r contraintes tables réifiées $c_i^{reif} : c_i \Leftrightarrow z_i$ où c_i est une contrainte table ternaire positive classique telle que $scp(c_i) = \{y_{i-1}, x_i, y_i\}$ et $rel(c_i) = \delta$ pour $i = 1..r$. Enfin, nous ajoutons une contrainte linéaire $z = \sum_{i \in 1..r}(1 - z_i)$.

Notez que $rel(c_i)$ correspond aux transitions valides dans M, ce qui signifie que $z_i = 1$ ssi $(y_{i-1} x_i y_i) \in \delta$. Par cette propriété, pour tout tuple τ sur X,

$$\text{dfa-cost}_M(\tau) = \min z = \min \sum_{i=1..r} (1 - z_i)$$

Nous devons alors minimiser la valeur de z pour obtenir le coût de violation pour la contrainte `soft-regular`.

Exemple 4 Reconsidérons la contrainte de l'exemple 2. Pour transformer $\texttt{regular}(X, M)$ en $\texttt{soft-regular}^{dfa}(X, M, z)$ avec $X = \langle x_1, x_2, x_3, x_4\rangle$, nous ajoutons les variables y_0, y_1, y_2, y_3 et y_4 ainsi que les variables booléennes z_1, z_2, z_3 et z_4.

Les contraintes tables ternaires réifiées ont la forme $c_i^{reif} : c_i \Leftrightarrow z_i$ avec $rel(c_i)$ définie comme suit (voir les transitions de l'automate en figure 1) :

y_{i-1}	x_i	y_i
q_0	c	q_4
q_4	c	q_4
q_0	a	q_1
q_1	a	q_1
q_1	b	q_2
q_2	b	q_2
q_2	a	q_3
q_3	a	q_3

Le coût de violation "dfa" d'un tuple correspond à la valeur minimale de $(1 - z_1) + (1 - z_2) + (1 - z_3) + (1 - z_4)$, où les valeurs des variables z_i ont été calculées à partir des différentes contraintes tables réifiées. Nous obtenons $z = 2$ pour $\langle c, a, a, b\rangle$ et $z = 1$ pour $\langle c, c, a, c\rangle$.

5 Résultats expérimentaux

Nous avons implémenté STR-Reif dans OscaR [17], un solveur écrit en Scala, et effectué quelques tests sur un Mac OS X 10.10.5 avec a 2.70GHz Intel Core i5 et 16GB de mémoire. Pour tester notre algorithme, nous avons généré des instances du problème académique d'alignement photographique. Dans ce problème, des personnes alignées doivent être prises en photo, et certaines d'entre elles ont des préférences sur les personnes voisines. En fait, pour les instances incohérentes de ce problème, nous avons utilisé des

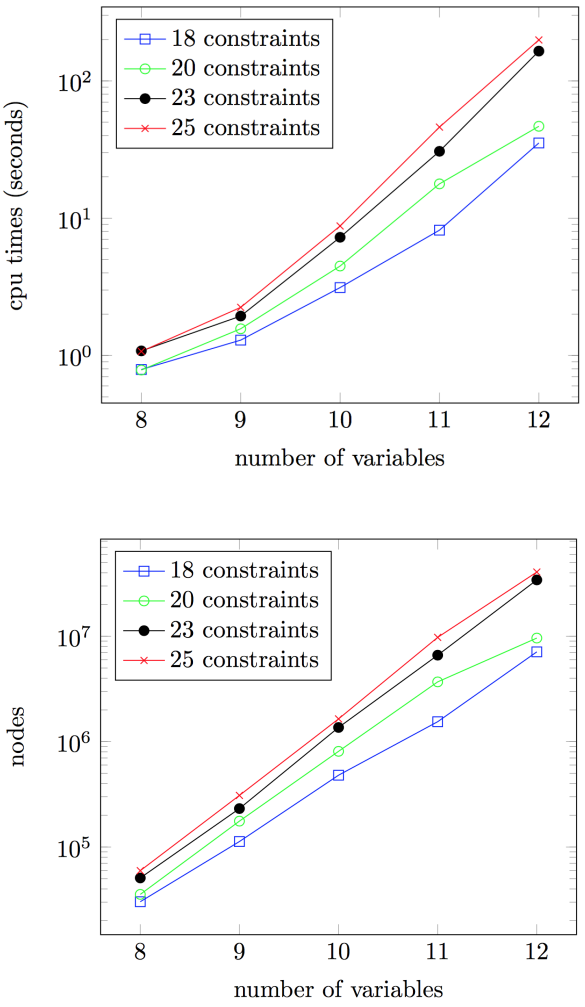

FIGURE 2 – Le temps moyen CPU et le nombre de noeuds visités pour les instances du problème d'alignement photographique

contraintes tables réifiées de manière à maximiser le nombre de contraintes satisfaites (problème MaxCSP).

Pour générer une série d'instances, nous avons utilisé les paramètres d'entrée suivants :

— n, le nombre de variables (c.-à-d. le nombre de personnes),

— e, le nombre de contraintes sur les préférences (c.-à-d. le nombre de préférences entre les personnes).

Dans ce modèle, les variables ont un domaine $1..n$, et il y a une contrainte *alldifferent* pour indiquer que les gens se tiennent à des positions différentes. Nous avons également e contraintes sur les préférences représentées comme des contraintes tables binaires réifiées.

La figure 2 montre le temps moyen cpu (en secondes) et le nombre de noeuds visités, calculés à partir de 10 instances, pour chaque classe de $< n, e >$. On peut observer que l'algorithme monte à l'échelle assez bien (c.-à-d. de manière quasi-linéaire) quand le nombre de variables n ou le nombre de contraintes e augmente. Toutefois, cette expérimentation est tout à fait préliminaire, et nous projetons de faire des expériences plus poussées dans le futur.

6 Conclusion

Dans ce papier, nous avons présenté un algorithme qui permet d'imposer la cohérence d'arc généralisée sur les contraintes tables réifiées sans exiger d'espace supplémentaire : nous utilisons uniquement les tables initiales des contraintes. Nous avons également montré comment il était possible de réifier la version *soft* de la contrainte `regular` en appliquant une simple décomposition en contraintes tables ternaires réifiées, et en adoptant une nouvelle mesure de violation. Comme toute contrainte peut être représentée (en théorie) avec une table, la contrainte table réifiée proposée offre une approche générale pour la réification.

Remerciements Le premier auteur est financé comme Aspirant du FRIA-FNRS belge.

Références

[1] N. Beldiceanu, M. Carlsson, P. Flener, and J. Pearson. On the reification of global constraints. *Constraints*, 18(1) :1–6, 2013.

[2] C. Bessiere and J. Régin. Arc consistency for general constraint networks : preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, 1997.

[3] P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4) :59–69, 1993.

[4] K. Cheng and R. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2) :265–304, 2010.

[5] V. Le clement de saint marcq, P. Schauss, C. Solnon, and C. Lecoutre. Sparse-sets for domain implementation. In *Proceedings of TRICS'13*, 2013.

[6] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table : Efficiently filtering table constraints with reversible sparse bit-sets. *https ://arxiv.org/abs/1604.06641*.

[7] F. Fages and S. Soliman. Reifying global constraints. Technical Report RR-8084, HAL, 2012.

[8] T. Feydy, Z. Somogyi, and P. Stuckey. Half reification and flattening. In *Proceedings of CP'11*, pages 286–301, 2011.

[9] I.P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, pages 191–197, 2007.

[10] N. Gharbi, F. Hemery, C. Lecoutre, and O. Roussel. Sliced table constraints : Combining compression and tabular reduction. In *Proceedings of CPAIOR'14*, pages 120–135, 2014.

[11] P. Van Hentenryck J.-B. Mairy and Y. Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints*, 19(1) :77–120, 2014.

[12] G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, pages 379–393, 2007.

[13] C. Lecoutre. STR2 : Optimized simple tabular reduction for table constraints. *Constraints*, 16(4) :341–371, 2011.

[14] C. Lecoutre, C. Likitvivatanavong, and R. Yap. STR3 : A path-optimal filtering algorithm for table constraints. *Artificial Intelligence*, 220 :1–27, 2015.

[15] O. Lhomme and J.-C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.

[16] H. Li, Y. Liang, J. Guo, and Z. Li. Making simple tabular reduction works on negative table constraints. In *Proceedings of AAAI'13*, pages 1629–1630, 2013.

[17] OscaR Team. OscaR : Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[18] G. Perez and J.-C. Régin. Improving GAC-4 for Table and MDD constraints. In *Proceedings of CP'14*, pages 606–621, 2014.

[19] G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'04*, pages 482–495, 2004.

[20] J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177 :3639–3678, 2007.

[21] W. van Hoeve, G. Pesant, and L.-M. Rousseau. On global warming : Flow-based soft global constraints. *Journal of Heuristics*, 12(4-5) :347–373, 2006.