STR et compression de contraintes tables

Nebras Gharbi **Fred Hemery** **Christophe Lecoutre** **Olivier Roussel**

CRIL - CNRS UMR 8188,
Université Lille Nord de France, Artois,
rue de l'université, 62307 Lens cedex, France
{gharbi,hemery,lecoutre,roussel}@cril.fr

Résumé

Les contraintes tables sont très étudiées par la communauté de programmation par contraintes. De nombreux algorithmes de filtrage pour ce type de contraintes (et aussi leurs combinaisons, par exemple, voir [6]) ont été proposés ces dernières années. L'un des meilleurs algorithmes, appelé STR2, qui est basé sur la technique dite de réduction tabulaire simple (STR pour Simple Tabular Reduction), maintient dynamiquement la liste des tuples supports de chaque contrainte au cours des processus d'inférence et de recherche. Sur certains problèmes, cet algorithme est toutefois supplanté par une approche consistant à représenter de manière compacte l'ensemble des tuples à l'aide d'un diagramme multivalué (MDD). Dans cet article, nous étudions la possibilité de combiner l'approche STR avec une forme compressée basée sur l'identification de motifs récurrents dans l'ensemble des tuples de chaque contrainte table.

Abstract

Over the recent years, many filtering algorithms have been developed for table constraints. STR2, one of the most efficient algorithms, is based on the technique of simple tabular reduction, meaning that it maintains dynamically the list of supports in each constraint table during inference and search. However, for some specific problems, the approach that consists in representing in a compact way tables by means of multi-valued decision diagrams (MDD) overcomes STR2. In this paper, we study the possibility of combining simple tabular reduction with a compression form of tables based on the detection of recurrent patterns in tuples.

1 Introduction

L'intérêt pour les contraintes tables, qui sont des contraintes définies en extension en listant les tuples autorisés (ou ceux qui sont interdits), est important en programmation par contraintes. En effet, nombre de problèmes impliquent l'utilisation de ce type de contraintes. Dans certains cas, représenter celles-ci n'est malheureusement pas possible car l'espace mémoire requis est trop important. Rappelons que la complexité spatiale pour représenter tous les tuples croît de manière exponentielle avec l'arité des contraintes. En vue de réduire la complexité spatiale, différentes approches ont été proposées dans la littérature. Certaines introduisent des structures de données compactes telles que les tries (arbres de préfixes) [2], les MDDs [1], les tables compressées [3] ou encore les automates déterministes finis (DFA) [7]).

À ce jour, les algorithmes de filtrage les plus efficaces pour les contraintes tables sont ceux basés sur l'utilisation des MDDs, et surtout ceux basés sur la technique STR (Simple Tabular Reduction). En particulier, les variantes STR2 [4] et STR3 [5] de l'algorithme STR1 original [9] ont été montrées particulièrement compétitives sur de nombreuses classes de problèmes. De manière schématique, hormis les problèmes pour lesquels le taux de compression possible avec l'utilisation de MDDs est très élevé, l'approche STR est celle qui est la plus efficace.

Pour étendre la gamme de problèmes pour lesquels STR est appropriée, nous proposons de combiner cette approche avec une technique simple de compression de tuples (différente de celle proposée dans [3] qui remplace des ensembles de tuples par un produit cartésien de sous-domaines). Le principe est d'identifier les motifs (sous-tuples) récurrents au sein des tuples de chaque contrainte, et d'intégrer une indirection au niveau des tuples vers ces motifs. L'algorithme de filtrage STR doit être alors modifié pour prendre en compte ces indirections, un système d'horodatage permettant d'éviter d'effectuer plusieurs fois les tests de validité pour chaque motif.

Dans cet article, nous détaillons notre approche en décrivant l'utilisation d'arbres de préfixes à la lecture des contraintes tables, puis celle de structures annexes pour les motifs et l'horodatage. Nous présentons quelques résultats expérimentaux avant de conclure.

2 Méthode de compression

Une contrainte table est une contrainte qui est définie en extension par un ensemble de tuples. Un tuple représente une combinaison de valeurs pour les variables de la contrainte, autorisée en cas de contrainte positive et réfutée en cas de contrainte négative. Un tuple est dit valide lorsque les valeurs de celui-ci sont présentes dans les domaines courants des variables correspondantes.

Un motif μ est une séquence de valeurs consécutives dans un tuple τ d'une table. On note $|\mu|$ la longueur d'un motif μ, et $\text{nbOcc}(\mu)$ le nombre d'occurrences du motif repérées dans l'ensemble des tuples d'une contrainte donnée.

Afin de réduire la complexité spatiale de la représentation des tuples de chaque contrainte, nous allons repérer les motifs les plus fréquents et remplacer chaque occurrence de motif par un symbole unique. De ce fait, la taille utilisée pour représenter la table sera d'autant plus faible que la longueur des motifs sera grande et que leur nombre d'occurrences sera important.

Il est important de noter que nous considérons que les motifs extraits dans le cadre de notre approche sont indépendants de leur position initiale dans le tuple. En conséquence, un motif ne correspond pas obligatoirement à l'affectation des mêmes valeurs aux mêmes variables mais plutôt la même suite d'affectation à une séquence de variables consécutives. Ce choix a été fait dans l'espoir d'obtenir des motifs les plus fréquents possibles, et donc une meilleure compression.

Pour identifier les motifs pertinents, nous allons dans un premier temps créer une forêt d'arbres de préfixes à partir des différents tuples d'une contrainte table donnée. Un arbre enregistre toutes les séquences existantes de valeurs de longueur donnée et leur nombre d'occurrences. Pour garantir un certain niveau d'efficacité de compression, la longueur minimale des séquences est fixée dans notre approche à 3, et la longueur maximale à l'arité de la contrainte moins 1.

Dans un second temps, il nous faut identifier les motifs les plus efficaces pour le processus de compression. Pour cela nous allons introduire la notion de **score** d'un motif μ comme suit :

$$\text{score}(\mu) = |\mu| \times \text{nbOcc}(\mu)$$

Un seuil de sélection est fixé, seuls les motifs dont le score est supérieur au seuil de sélection sont retenus dans l'algorithme de compression et stockés dans la table des motifs. Pour des raisons d'efficacité, le nombre total de motifs retenus est borné par un second paramètre afin de contrôler le temps de compression.

Le processus de compression utilise donc les motifs dont le score est supérieur au seuil de sélection. Un parcours de la table est effectué pour détecter la présence des motifs retenus et établir une référence vers la table de motifs. Si dans un tuple, plusieurs motifs se recouvrent, l'algorithme de compression choisit en priorité le motif ayant le meilleur score. Après compression, la table contient des tuples de longueurs différentes composés de valeurs et de références vers la table des motifs.

Nous donnons maintenant une illustration de ce processus. La table 1 représente une contrainte table positive d'arité 5, portant sur les variables x_1, x_2, ..., x_5. Nous pouvons remarquer que plusieurs motifs se répètent au sein des tuples parmi lesquels nous pouvons citer *cbc*, *aab* et *abb* comme des motifs de longueur 3 et *aabb*, *acbc* et *cbca* comme des motifs de longueur 4. À la lecture de notre contrainte table nous pouvons construire les arbres de préfixes schématisés à la figure 1 correspondant aux motifs possibles de tailles 3 (minimum) et 4 (arité de la contrainte moins 1). Chaque feuille identifie un chemin μ (allant de la racine à la feuille) auquel il est possible d'associer (au niveau de la feuille) le compteur $\text{nbOcc}(\mu)$ (ceci n'est pas montré sur la figure).

	x_1	x_2	x_3	x_4	x_5
τ_1	(c,	b,	c,	a,	c)
τ_2	(a,	a,	b,	b,	a)
τ_3	(a,	c,	b,	c,	a)
τ_4	(b,	a,	c,	b,	c)
τ_5	(b,	a,	a,	b,	b)
τ_6	(a,	c,	b,	c,	b)
τ_7	(a,	c,	a,	c,	a)

TABLE 1 – Contrainte en extension C_{x_1,x_2,x_3,x_4,x_5}

L'application de l'algorithme de compression nous permet d'avoir une version compressée de la contrainte table ; sa représentation logique est donnée par la figure 2(a) tandis que la table des motifs est donnée par la figure 2(b). Les tuples τ_1, τ_3, τ_4, τ_5 de la table compressée font référence au motif μ_3 aux positions respectives 1, 2, 3, et 2, et les tuples τ_2, τ_6 font référence au motif μ_2 aux positions respectives 1 et 2. Ce mécanisme de compression peut entraîner une réduction importante de l'espace mémoire occupé par la contrainte table de départ. Ceci est illustré par la vue physique donnée par la figure 2(c).

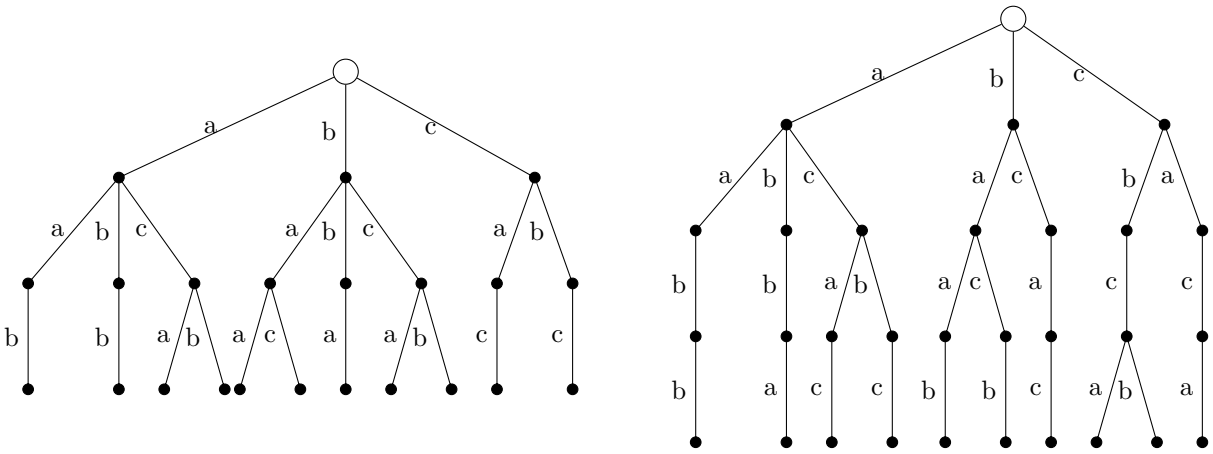

FIGURE 1 – Arbres de préfixes de longueur 3 et 4 construits à partir de la contrainte de la table 1

	x_1	x_2	x_3	x_4	x_5
τ_1	μ_3			a	c
τ_2	μ_2			b	a
τ_3	a	μ_3			a
τ_4	b	a	μ_3		
τ_5	b	μ_2			b
τ_6	a	μ_3			b
τ_7	a	c	a	c	a

(a) Table compressée (vue logique)

μ_1	...
μ_2	a,a,b
μ_3	c,b,c
...	...

(b) Table des motifs

τ_1	μ_3	a	c		
τ_2	μ_2	b	a		
τ_3	a	μ_3	a		
τ_4	b	a	μ_3		
τ_5	b	μ_2	b		
τ_6	a	μ_3	b		
τ_7	a	c	a	c	a

(c) Table compressée (vue physique)

FIGURE 2 – Compression de la table

3 Algorithme de filtrage

Lors du filtrage d'une contrainte table, il est nécessaire de vérifier la validité des tuples, ce qui implique de vérifier la validité des motifs. Quand un motif apparaît plusieurs fois dans la table à partir de la même position, nous souhaitons n'effectuer le test de validité du motif qu'une seule fois, ce qui permet de traduire la compression spatiale en une réduction du temps de filtrage.

Pour ce faire, nous utilisons un compteur *time* qui est incrémenté à chaque filtrage de la contrainte table et un tableau $stamps[\mu_i, j]$ associé à la contrainte. Pour un motif μ_i qui s'applique à partir d'une position j, $stamps[\mu_i, j]$ donne le résultat du dernier test de validité de (μ_i, j) (champ $stamps[\mu_i, j].valid$) ainsi que la valeur $stamps[\mu_i, j].time$ du compteur *time* lors de ce test. Chaque fois que la validité de (μ_i, j) doit être testée, nous vérifions d'abord si $stamps[\mu_i, j].time$ est égal à la valeur courante de *time*. Si c'est le cas, la validité a déjà été testée dans l'opération de filtrage courante et donc $stamps[\mu_i, j].valid$ fournit directement la réponse, ce qui évite des calculs inutiles. Sinon, il faut effectivement tester la validité de (μ_i, j) et sauvegarder le résultat dans $stamps[\mu_i, j]$.

	position		
	1	2	3
μ_2	$0, ?$	$0, ?$	$0, ?$
μ_3	$0, ?$	$0, ?$	$0, ?$

(a) Initialisation des champs $(time, valid)$

	position		
	1	2	3
μ_2	$1, F$	$1, V$	$0, ?$
μ_3	$1, F$	$1, V$	$1, F$

(b) À la fin de $time = 1$

FIGURE 3 – Évolution de la structure *stamps*

La figure 3 présente un exemple d'évolution de la structure *stamps*. Au départ, tous les éléments sont initialisés à $time = 0$ et le champ *valid* reste non assigné (voir figure 3(a)). Lors du premier filtrage de la table, le compteur global *time* passe à 1. Pour déterminer si le tuple τ_3 (par exemple) est valide, il faut s'assurer que μ_3 à la position $j = 2$ est valide. Comme $stamps[\mu_3, 2].time$ n'est pas égal à la valeur courante de *time*, nous savons que ce test n'a pas déjà été fait. Nous vérifions donc si c, b et c sont toujours présents dans les domaines de x_2, x_3 et x_4 respectivement. Nous supposons ici que le résultat est positif. Nous stockons donc $(1, Vrai)$ dans $stamps[\mu_3, 2]$. Lorsque nous testons ensuite la validité du tuple τ_5, on s'aperçoit immédiatement que la validité de μ_3 a été testée lors du filtrage courant et il suffit de prendre le résultat sauvegardé dans $stamps[\mu_3, 2].valid$. En supposant que μ_2 et μ_3 sont tous les deux invalides en position 1, valides en position 2 et que μ_3 est non valide en position 3,

nous obtenons à la fin du filtrage le résultat présenté en figure 3(b).

Pour une contrainte d'arité r et un motif μ, il n'y a que $r - |\mu| + 1$ couples (μ, j) possibles (car $1 \leq j \leq r - |\mu| + 1$). La structure *stamps* a donc une taille en $O(m.r)$ où m est le nombre de motifs identifiés.

4 Résultats expérimentaux

Pour montrer le potentiel de notre approche (STR^c), nous avons comparé le comportement des algorithmes STR1, STR2, STR3 et STR^c lorsqu'ils sont intégrés à l'algorithme de recherche MAC (qui maintient la propriété de cohérence d'arc généralisée lors d'une recherche arborescente). Nous avons effectué quelques tests sur des instances de deux problèmes distincts : le premier correspond aux séries mdd introduites dans [1] et le second à la construction de nonogrammes [8]. Les résultats figurent en table 2 ; l'heuristique dom/ddeg est utilisée pour garantir le même parcours d'arbre (dom/wdeg est plus versatile).

L'algorithme STR^c permet une économie spatiale d'au moins 50% par rapport à STR1 et STR2, et jusqu'à un facteur 4 par rapport à STR3. Les temps de résolution sont donnés sous la forme build+search où build indique le temps pour construire l'instance et search le temps nécessaire pour trouver une solution. Lorsqu'on considère le temps de recherche, il apparaît que STR^c rivalise avec STR1, mais reste toutefois supplanté par STR2. Il est vraisemblable qu'un meilleur réglage des paramètres utilisés pour STR^c (dans notre expérimentation, 500 motifs autorisés avec un seuil de sélection égal à 50) permettrait de mieux contrôler le temps nécessaire à la construction.

Instance		STR1	STR2	STR3	STR^c
mdd-25-7-23	mem	147M	147M	384M	102M
	CPU	2.3+26.3	2.3+13.7	2.7+50.0	11.4+30.6
mdd-23-15-1	mem	223M	238M	597M	127M
	CPU	4.0+31.1	3.9+10.8	4.8+220	37.6+33.7
non-gp-65	mem	33M	33M	76M	21M
	CPU	0.5+41.1	0.6+12.0	0.7+9.1	7.3+37.5
non-gp-130	mem	221M	230M	663M	129M
	CPU	4.1+0.5	4.1+0.3	5.3+0.6	120+0.5

TABLE 2 – Espace mémoire et temps CPU pour résoudre quelques instances avec MAC.

5 Conclusion

Dans cet article, nous avons cherché à combiner l'approche STR avec une compression originale des contraintes tables. En identifiant les motifs récurrents dans l'ensemble de tuples présents dans les différentes tables, il est possible d'économiser l'espace mémoire (à l'aide d'un système d'indirection) et également le temps CPU en évitant les tests de validité redondants sur les motifs enregistrés. L'algorithme STR^c que nous proposons semble, d'après nos tests préliminaires, rivaliser avec STR1 (au niveau de la recherche) mais supplanté par STR2. Nous envisageons d'étudier diverses optimisations de ce nouvel algorithme, ainsi qu'un réglage plus fin des paramètres nécessaires à l'identification des motifs. Nous projetons également, dans un futur proche, d'étudier d'autres modèles de compression, toujours en conjonction avec l'approche STR.

Remerciements

Ce travail bénéficie du soutien du CNRS et d'OSEO dans le cadre du projet ISI Pajero.

Références

[1] 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.

[2] 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.

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

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

[5] C. Lecoutre, C. Likitvivatanavong, and R. Yap. A path-optimal GAC algorithm for table constraints. In *Proceedings of ECAI'12*, pages 510–515, 2012.

[6] A. Paparrizou and K. Stergiou. An efficient higher-order consistency algorithm for table constraints. In *Proceedings of AAAI'12*, pages 335–541, 2012.

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

[8] G. Pesant, C.-G. Quimper, and A. Zanarini. Counting-based search : Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 43 :173–210, 2012.

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