A Compression Method for STR

Nebras Gharbi (student)
Fred Hemery, Christophe Lecoutre, and Olivier Roussel (supervisors)

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

Abstract. Over the recent years, many filtering algorithms have been developed for table constraints. Simple Tabular Reduction (STR) is an effective approach to filter table constraints. 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 tables in a compact way by means of multi-valued decision diagrams (MDD) overcomes STR. In this paper, we study the possibility of combining simple tabular reduction with tables compression based on the detection of recurrent patterns in tuples.

1 Introduction

Table constraints, which are defined in extension by listing allowed tuples (or those that are disallowed), are important for modelling many problems in constraint programming. In some cases, representing such constraints is unfortunately not possible due to the memory space required. In fact, the spatial complexity to represent all tuples grows exponentially with the arity of the constraints. In order to reduce the space complexity, different approaches have been proposed. Some of them are compact data structures based approaches, such as tries [2], MDDs [1], compressed tables [3] or the deterministic finite automata (DFA [7]). Other recent developments have been presented in [6, 9].

The most effective filtering algorithms for table constraints are those based on MDDs, and those based on STR. Most notably, the variants STR2 [4] and STR3 [5] of the algorithm STR1 [10] are competitive on many classes of problems. Except for problems where the compression with the use of MDDS is very effective, the STR approach is more effective. We propose an approach combining STR with a form of compression which is different from the ones described in [3] and [11] where a Cartesian product representation is used for compression. The idea is to identify the recurrent patterns (sub-tuples) in the tuples of each constraint, and replace theirs occurrences by references to a patterns table. The filtering algorithm STR must be modified to take into consideration these patterns appearing at different positions, thus a time stamp system is used to avoid repeated validity tests.

In this paper, we describe our approach, giving some details about the frequent patterns detection, and the used data structures. We present some experimental results before concluding.

2 Compression method

Definition 1. *A table constraint is a constraint which lists explicitly a set of tuples. A tuple τ represents a combination of values for the variables in the constraint scope. These tuples are allowed in case of a positive constraint and disallowed in case of a negative one. A tuple is said valid when its values are present in the current domains of the corresponding variables.*

Definition 2. *A pattern μ is a sequence of consecutive values in a tuple τ of a table constraint. We note $|\mu|$ the length of a pattern μ, and $nbOcc(\mu)$ the number of occurrences of the pattern all over the tuples of a given constraint.*

In order to reduce the spatial complexity of every table constraint, we detect the most frequent patterns and replace every occurrence of a pattern by a unique symbol. Therefore, the longer the patterns and the more frequent they are, the smaller the table constraint representation will be. It is important to note that we consider that the patterns extracted in our approach are independent from their position in the tuple. According to that, a pattern does not correspond necessarily to the assignment of the same values to the same variables, but rather the same sequence of values. This choice was made to hopefully maximize the frequency of possible patterns, and, thus, to obtain a better compression.

To identify the relevant patterns, we first create a trie from the various tuples of a given table constraint. A trie contains all the existing sequences of values and their number of occurrences. To guarantee a certain level of compression efficiency, the minimal length of patterns is fixed in our approach to 3, and the maximal length to the constraint arity minus 1. Secondly, it is necessary to identify the most relevant patterns for the compression process. To do that, we introduce the notion of **score** of a pattern μ as follows:

$$score(\mu) = |\mu| \times nbOcc(\mu)$$

A selection threshold is fixed, and only the patterns having a score that is greater than this threshold are retained in the compression algorithm and stored in the patterns table. For efficiency reasons, the total number of retained patterns is limited by a second parameter in order to control the compression time.

After detecting the most relevant patterns, iterating over the constraint tuples is necessary to detect such patterns and establish references towards the patterns table. If in a tuple, several patterns overlap, the compression algorithm chooses first and foremost the pattern having the best score.

We give now an illustration of the whole process from patterns detection to table compression. Table 1 represents a positive table constraint of arity 5, involving variables $x_1, x_2, \ldots, x_5$. We can notice that several patterns are repeated

among tuples such as *cbc*, *aab* and *abb* as patterns of length 3 and *aabb*, *acbc* and *cbca* as patterns of length 4. Through visiting all tuples of our constraint, we can build the trie illustrated in Figure 1, where for simplicity reasons, the number of occurrences is only given for leaves. Every leaf identifies a path μ (going from the root to the leaf) associated with the nbOcc(μ) counter.

	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. Table constraint C_{x_1,x_2,x_3,x_4,x_5}

The compression algorithm allows us to have a compressed version of the table constraint; its logical representation is given in Figure 2(a) whereas the patterns table is given in Figure 2(b). Tuples τ_1, τ_3, τ_4, τ_5 of the compressed table respectively reference the pattern μ_3 at positions 1, 2, 3 and 2, and tuples τ_2, τ_6 respectively reference the pattern μ_2 at positions 1 and 2. This mechanism of compression can entail an important reduction of the space occupied by the table. This is illustrated by the physical view given in Figure 2(c).

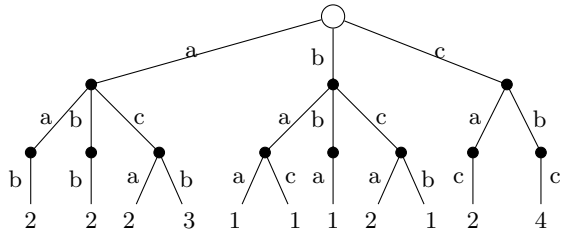
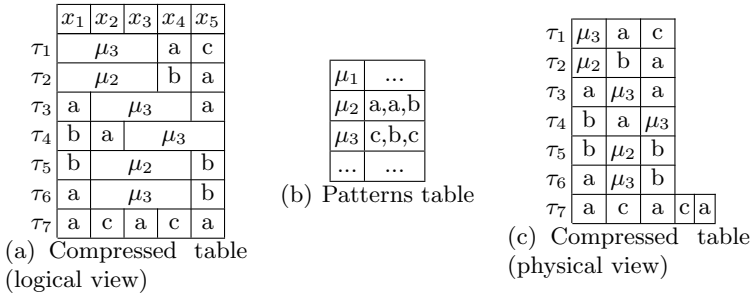


Fig. 1. The trie for patterns of size 3 built from the constraint given in Table 1

2.1 Filtering algorithm

During the filtering of a table constraint, it is necessary to check the validity of tuples, which implies to verify the validity of the patterns. When a pattern



(a) Compressed table (logical view)

(b) Patterns table

(c) Compressed table (physical view)

Fig. 2. The process of compression

appears several times in the table at the same position, we wish to test the validity of the pattern only once, which allows us to speed up the filtering.

To do this, we use a counter *time* which is incremented every time we filter the table constraint and a set of timestamps $stamps[\mu_i, j]$. For a pattern μ_i which applies from a position j, $stamps[\mu_i, j]$ gives the result of the last test of validity (μ_i, j) (field stamps $[\mu_i, j].valid$) as well as the value stamps $[\mu_i, j].time$ of the counter *time* when this test was made. Every time the validity of (μ_i, j) must be tested, we verify first if stamps $[\mu_i, j].time$ is equal to the current value of *time*. If it is the case, the validity was already tested in the current step of filtering and, thus, $stamps[\mu_i, j].valid$ supplies directly the answer, which avoids useless calculations. Otherwise, it is necessary to test the validity of (μ_i, j) and to save the result in $stamps[\mu_i, j]$.

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

(a) Initialisation ($time = 0$)

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

(b) At the end of $time = 1$

Fig. 3. Evolution of the *stamps* structure

Figure 3 presents an example of the evolution of the structure *stamps*. First, all elements are initialized at $time = 0$ and the field *valid* remains unassigned (see Figure 3(a)). During the first filtering of the table, the global counter *time* is set to 1. To determine whether tuple τ_3 (for example) is valid, it is necessary to ensure that μ_3 at position $j = 2$ is valid. As stamps$[\mu_3, 2]$.time is not equal to the current value of *time*, we know that this test was not already made in the current filtering step. Thus, we verify whether c, b and c are still present in the domains of x_2, x_3 and x_4 respectively. We suppose here that the result is

positive. Hence, we store $(1, T)$ in stamps$[\mu_3, 2]$. Later, when we test the validity of tuple τ_5, we notice immediately (our assumption) that the validity of μ_3 was tested during the current filtering and it is enough to use the result saved in $stamps[\mu_3, 2].valid$. Assuming that μ_2 and μ_3 are both invalid at position 1, valid at position 2 and that μ_3 is not valid at position 3, we obtain at the end of the filtering the result presented in Figure 3(b).

For a constraint of arity r and a pattern μ, there at most $r - |\mu| + 1$ possible pairs of (μ, j) (because 1≤ j≤ r-$|\mu|$+ 1). So, the structure *stamps* has a size $O(m.r)$ where m is the number of detected patterns.

3 Experiments

In order to show the practical interest of our approach (STR^c), we compared the behavior of the algorithms STR1 [10], STR2 [4], STR3 [5] and STR^c when they are integrated into the search algorithm MAC (which maintains the property of generalized arc consistency during search). To validate our approach, we made some tests on instances of two distinct problems: the first one corresponds to the series mdd introduced in [1] and the second one is nonograms [8]. The experimental results on representative instances are given in table 2; the heuristic dom/ddeg is used to ensure the same search path.

On such instances, STR^c allows a spatial reduction of at least 50 % compared to STR1 and STR2, and a speed-up factor up to 4 compared to STR3. Resolution times are given in a "build+search" form where *build* indicates the time to build the instance and *search* the necessary time to find a solution. When we consider the time of search, it seems that STR^c competes with STR1, but stay, however, supplanted by STR2. It is likely that a better tuning of the parameters used for STR^c (in our experiment, 500 patterns allowed with a selection threshold equal to 50) would allow us to better control the compression time.

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. Memory space and CPU time for some instances resolution with MAC

4 Conclusion and Future work

In this paper, we tried to combine both techniques of simple tabular reduction and compression. Identifying recurring patterns in the set of tuples present in different tables, allowed us to reduce memory space and also the CPU time by avoiding redundant validity tests. From our preliminary tests, the STR^c algorithm we propose seems to be competitive with STR1 (in the search step) but supplanted by STR2. We are considering various optimizations of this new algorithm: a better tuning of parameters is necessary to identify the frequent patterns.

In the future, we shall study other forms of compression, always in conjunction with STR algorithms.

Acknowledgments

This work has been supported by both CNRS and OSEO within the ISI project 'Pajero'.

References

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. J.-B. Mairy, P. Hentenryck, and Y. Deville. An optimal filtering algorithm for table constraints. In *Proceedings of CP'12*, pages 496–511. 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-C. Régin. Improving the expressiveness of table constraints. In *Proceedings of the workshop ModRef'11 held with CP'11*, 2011.
10. J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.
11. W. Xia and R. Yap. Optimizing STR algorithms with tuple compression. In *Proceedings of CP'13*, 2013. To appear.