Compression of Rewriting Systems for Termination Analysis * †

Alexander Bau[1], **Markus Lohrey**[2], **Eric Nöth**[2], **and Johannes Waldmann**[1]

1 **HTWK Leipzig, Germany**
{abau,waldmann}@imn.htwk-leipzig.de
2 **Universität Leipzig, Germany**
{lohrey,noeth}@informatik.uni-leipzig.de

Abstract

We adapt the TreeRePair tree compression algorithm and use it as an intermediate step in proving termination of term rewriting systems. We introduce a cost function that approximates the size of constraint systems that specify compatibility of matrix interpretations. We show how to integrate the compression algorithm with the Dependency Pairs transformation. Experiments show that compression reduces running times of constraint solvers, and thus improves the power of automated termination provers.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases termination of rewriting, matrix interpretations, constraint solving, tree compression

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.97

1 Introduction

For proving termination of rewrite systems automatically, a standard approach is to use monotone interpretations, e.g., by polynomials [11] or matrices [6] for the function symbols. The conditions of monotonicity and of compatibility with a given rewrite system result in a constraint system for the coefficients of the interpretation. Termination provers then use a constraint solver to obtain an actual interpretation. One way of solving constraint systems is bit-blasting [7, 13, 4]: The unknown numerical coefficients are represented by sequences of boolean unknowns, the constraints are transformed to a formula in propositional logic, and a state-of-the-art SAT solver [5] is applied to find a satisfying assignment, from which the interpretation can be reconstructed.

In the present paper, we describe and investigate a method that allows to obtain small constraint systems for the termination problem of a given rewrite system. From a given rewrite system we compute a *straight line program* that produces all left-hand and right-hand sides of the rewrite system. The elementary operation of this straight line program is *substitution* of terms. Such straight line programs were used for tree compression in [12]. The computed straight line program can be directly seen as a straight line program that computes the coefficients of the linear interpretations for all left-hand and right-hand sides from the unknown coefficients of the function symbols. The elementary operations

* The first author is supported by an European Social Fund grant.
† The second and third author are supported by the DFG research project ALKODA.

of this new straight line program are matrix-matrix and matrix-vector multiplications. We define the cost of the straight line program as the number of matrix-matrix multiplications, which is justified by the fact that matrix-vector multiplications are cheap in comparision to matrix-matrix multiplications.

Hence, our goal is to compute from a given rewrite system a straight line program with small cost. We do this by an adaptation of the TreeRePair algorithm [12] for tree compression. In each iteration, TreeRePair finds a most frequently occurring *digram* (a term pattern consisting of two function symbols) in the input tree and replaces this digram by a fresh symbol. For the purpose of proving termination with matrix interpretations, TreeRePair has been used in [6] (where a naive implementation of TreeRePair has been described independently from [12]). Our new adaptation of TreeRePair chooses in each iteration a digram that reduces the cost (number of matrix multiplications) maximally.

Our cost function (number of matrix multiplications) directly relates to the size of the constraint system (size of the CNF formula). This is a simple monotonic relation. It depends on the number of matrix constraints, the dimensions of the matrices, and the encoding of the operations on the matrix elements. In the present paper, we omit the discussion of elementary operations, and refer to [3] instead. The size of the formula relates, in turn, to the running time of the SAT solver. In general this relation is not monotonic, and it seems impossible to predict the relation exactly, since it depends on the particulars of simplification and resolution strategies embedded in the SAT solver, which is outside the scope of this paper. We therefore just assume that smaller formulas give, in general, shorter running times for solvers, and we test this hypothesis by experiments.

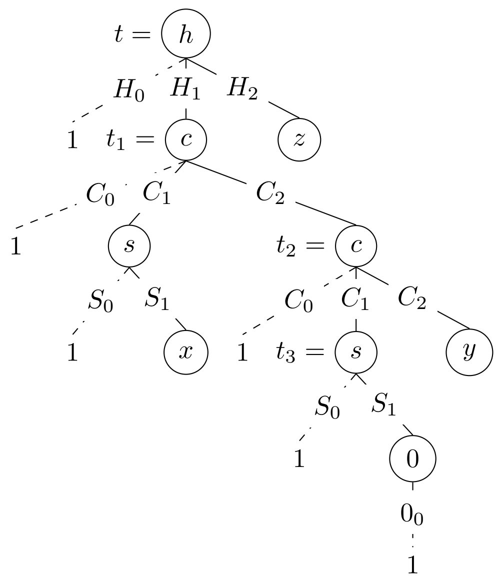
The new contributions of the present paper are:

- We define a cost function for terms (digrams) that reflects the size of constraint systems more accurately, by taking occurrences of variables in subterms into account.
- We present the algorithm MCTreeRePair that finds optimal digrams and discuss its performance.
- We combine our approach with the dependency pairs transformation [1].
- We present an open-source implementation of our algorithm, and we give an experimental evaluation, when applied as part of a realistic termination prover.

2 Terms

We use standard notations for terms and term rewriting systems. Let Σ be a finite *ranked alphabet of symbols* (also called a *signature*), where the *rank* or *arity* of $f \in \Sigma$ is denoted with $\mathrm{rk}(f) \in \mathbb{N}$. Let $\Sigma_l = \{f \in \Sigma \mid \mathrm{rk}(f) = l\}$ for $l \in \mathbb{N}$ and let k be maximal such that $\Sigma_k \neq \emptyset$. A *term* (or *tree*) over Σ is a pair $t = (D, \lambda)$ where D is a finite prefix closed and non-empty subset of $\{1, \dots, k\}^*$ and λ is a function from D to Σ such that for all $p \in D$ and $1 \le d \le k$: $pd \in D$ if and only if $1 \le d \le \mathrm{rk}(\lambda(p))$. Elements of D are also called *positions* or *nodes* of t. Define $|t| = |D|$ (the size of the term t). For $p \in D$ let $t|_p = (\{q \in \{1, \dots, k\}^* \mid pq \in D\}, \lambda')$, where $\lambda'(q) = \lambda(pq)$ for $pq \in D$, be the *subterm* of t rooted at position p. We write $s \trianglelefteq t$ (resp. $s \triangleleft t$) if s is a subterm (resp., a proper subterm) of t. With $\mathrm{Term}(\Sigma)$ we denote the set of all terms over Σ. We use the standard term notation, i.e., if for $t = (D, \lambda)$ we have $\lambda(\varepsilon) = f$, $\mathrm{rk}(f) = n$, and $t_i = t|_i$ for $1 \le i \le n$, then $t = f(t_1, \dots, t_n)$.

Let V be a finite set of *variables* with $\Sigma \cap V = \emptyset$. We define $\mathrm{Term}(\Sigma, V) = \mathrm{Term}(\Sigma \cup V)$, where every variable $x \in V$ gets rank 0, i.e., is treated as a constant. For a term $t \in \mathrm{Term}(\Sigma, V)$ let $\mathrm{Var}(t)$ be the set of variables that occur at least once in t. A *term rewriting system* (TRS) over the signature Σ is a finite set $R \subseteq \mathrm{Term}(\Sigma, V) \times \mathrm{Term}(\Sigma, V)$ such that



coefficients of variables:

$$\begin{aligned}
\mathrm{coeff}_x(t_1) &= C_1 \cdot S_1\\
\mathrm{coeff}_y(t_1) &= C_2 \cdot C_2\\
\mathrm{coeff}_x(t) &= H_1 \cdot \mathrm{coeff}_x(t_1)\\
\mathrm{coeff}_y(t) &= H_1 \cdot \mathrm{coeff}_y(t_1)
\end{aligned}$$

absolute parts:

$$\begin{aligned}
\mathrm{coeff}_1(t_3) &= S_1 \cdot 0_0\\
\mathrm{coeff}_1(t_2) &= C_0 + C_1 \cdot \mathrm{coeff}_1(t_3)\\
\mathrm{coeff}_1(t_1) &= C_0 + C_1 \cdot S_0 + C_2 \cdot \mathrm{coeff}_1(t_2)\\
\mathrm{coeff}_1(t) &= H_0 + H_1 \cdot \mathrm{coeff}_1(t_1)
\end{aligned}$$

Figure 1 Bottom-up computation of the coefficient matrices of $[h(c(s(x), c(s(0), y)), z)]$.

for every rule $(\ell \to r) \in R$ we have $\ell \notin V$ and $\mathrm{Var}(r) \subseteq \mathrm{Var}(\ell)$. The *one-step rewriting relation* $\to_R$ is defined as usual.

3 Matrix interpretation of terms and the cost of terms

As explained in the introduction we use matrix interpretations for proving termination of term rewriting systems. To define such interpretations, let us fix a semiring S (the ring of matrix coefficients) and a dimension $n \geq 1$ for the further consideration. We want to interpret a term t with m different variables as an m-ary function from S^n to S^n. For a set of variables $U \subseteq V$ we denote with $(S^n)^U$ the set of all mappings from U to S^n, or alternatively, the set of U-indexed tuples over S^n. Moreover, for every symbol $f \in \Sigma_m$ we fix matrices $F_1, \dots, F_m \in S^{n\times n}$ and a vector $F_0 \in S^n$. This allows us to define the linear function $[f] : (S^n)^m \to S^n$ by

$$[f](x_1, \dots, x_m) = F_0 + F_1x_1 + \cdots + F_mx_m, \qquad (1)$$

where $x_1, \dots, x_m \in S^n$. Now let $t \in \mathrm{Term}(\Sigma, V)$ be a term with $U = \mathrm{Var}(t)$. The interpretation $[t] : (S^n)^U \to S^n$ is computed by composing the interpretations for the ranked symbols in the natural way: Let $t = f(t_1, \dots, t_k)$. Then $[t](\overline{x}) = [f]([t_1](\overline{x}_1), \dots, [t_k](\overline{x}_k))$, where $\overline{x} \in (S^n)^U$ and $\overline{x}_i$ is the restriction of $\overline{x}$ to $\mathrm{Var}(t_i) \subseteq U$.

We next define a cost measure for terms that reflects the number of matrix multiplications that have to be done when the coefficients of the linear function that is represented by a term, are computed in a bottom-up manner. Recall that a term $t \in \mathrm{Term}(\Sigma, V)$ with $U = \mathrm{Var}(t)$ represents the linear function $[t] : (S^n)^U \to S^n$, which can be written as an expression $T_0 + T_1x_1 + \cdots + T_kx_k$, where $\mathrm{Var}(t) = \{x_1, \dots, x_k\}$, $T_0 \in S^n$ and $T_1, \dots, T_k \in S^{n\times n}$. We identify $[t]$ with this expression. Moreover, let $\mathrm{coeff}_{x_i}(t) = T_i$. Figure 1 shows the bottom-up calculation of the coefficients of an example term (from Example 2 below). The constant term (F_0 in (1)) is denoted by $\mathrm{coeff}_1(\cdot)$. In total, the computation of $[t]$ needs 4 multiplications of an $(n \times n)$-matrix by an $(n \times n)$-matrix, and 5 multiplications of an $(n \times n)$-matrix by an $(n \times 1)$-matrix (a vector). In the following, we will only count matrix-by-matrix multiplications, and ignore matrix-by-vector multiplications, as well as additions. This is justified by higher asymptotic cost of multiplications (n^3 versus n^2). The matrix dimension in our applications is small (at most 5), see Section 7. Hence, matrix multiplication algorithms with an asymptotic running time better than $O(n^3)$ (e.g., Strassen's algorithm) are not useful in our context.

Let $t = f(t_1, \dots, t_k)$, and assume that all coefficient matrices $\mathrm{coeff}_x(t_i)$ are already known. Then we can compute the coefficient matrix $\mathrm{coeff}_x(t)$ as

$$\mathrm{coeff}_x(t) = \sum_{i=1}^{k} F_i \cdot \mathrm{coeff}_x(t_i),$$

where F_i is from (1), and we set $\mathrm{coeff}_x(t_i) = 0$ if $x \notin \mathrm{Var}(t_i)$. Hence, we have one matrix multiplication for each i with $x \in \mathrm{Var}(t_i)$. But note that the multiplication is trivial if $x = t_i$ (multiplication by the identity matrix, which costs nothing). This motivates the following definition:

▶ **Definition 1.** The *(matrix multiplication) cost* of a term $t = (D, \lambda) \in \mathrm{Term}(\Sigma, V)$ is

$$\mathrm{cost}(t) = \sum_{p \in D \setminus \{\varepsilon\}, \lambda(p) \notin V} |\,\mathrm{Var}(t|_p)|. \qquad (2)$$

The cost of a tuple $(t_1, \dots, t_m)$ of terms is $\sum_{i=1}^{m} \mathrm{cost}(t_i)$.

Note that this definition models a bottom-up evaluation where we do not use any caching, memoization, etc. The following example has been taken from [6].

▶ **Example 2.** Let $\mathrm{rk}(h) = \mathrm{rk}(c) = 2$, $\mathrm{rk}(s) = 1$, and $\mathrm{rk}(0) = 0$. Then we have

$$\mathrm{cost}(h(x, c(y, z))) = 2 \qquad \mathrm{cost}(h(c(s(x), c(s(0), y)), z)) = 4$$
$$\mathrm{cost}(h(c(s(y), x), z)) = 3 \qquad \mathrm{cost}(h(y, c(s(0), c(x, z)))) = 4$$

Figure 1 shows a detailed computation of the coefficients of the interpretation of the term $h(c(s(x), c(s(0), y)), z)$.

4 Term compression with TreeRePair

In this section we describe an adaptation of the RePair compression algorithm for strings, which is called TreeRePair in [12], see also [6]. The idea is to replace frequently occurring tree patterns of size 2 (so called *digrams*) by new symbols (called non-terminals in [12]) and to store in a table the patterns corresponding to the new symbols. Most of the notation from this section will be needed in the next section where we outline an adaptation of TreeRePair with the goal of reducing the matrix multiplication cost of a list of terms.

Let us fix a ranked alphabet Σ.

▶ **Definition 3.** A *digram* over Σ is a triple $d = [f, i, g]$ where $f, g \in \Sigma$ and $1 \leq i \leq \mathrm{rk}(f)$). The *rank* (or arity) of this digram is $\mathrm{rk}(d) = \mathrm{rk}(f) - 1 + \mathrm{rk}(g)$.

We consider a digram d as a new symbol of rank $\mathrm{rk}(d)$.

▶ **Definition 4.** To the digram $d = [f, i, g]$ with $\mathrm{rk}(d) = n$ and $\mathrm{rk}(g) = l$ we associate the rewrite rule

$$\mathsf{rule}(d) = (d(x_1, \dots x_n) \to f(x_1, \dots x_{i-1}, g(x_i, \dots, x_{i+l-1}), x_{i+l}, \dots x_n)).$$

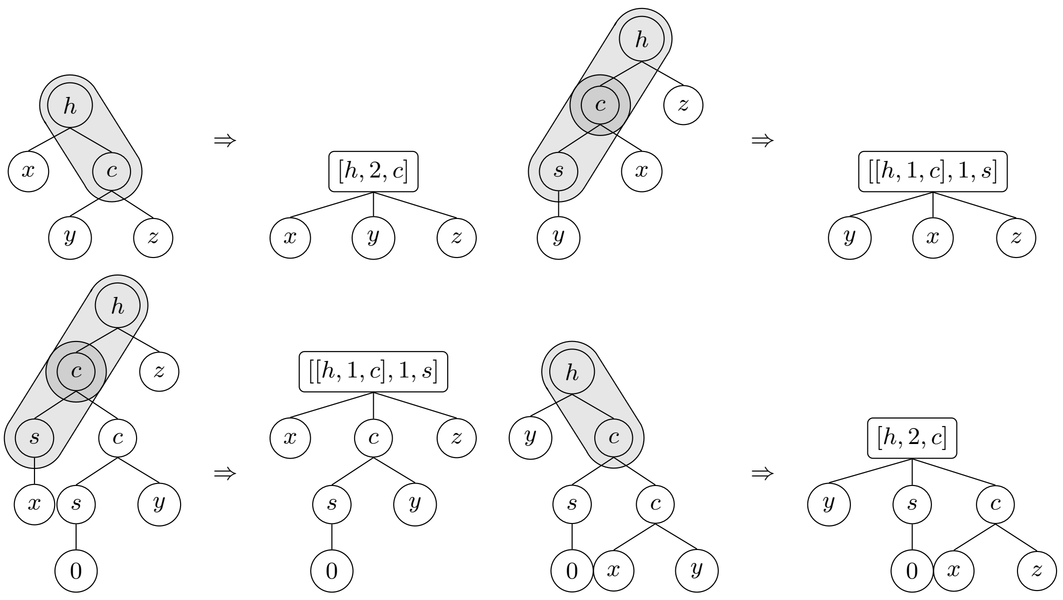


Figure 2 The replaced digrams from Example 7.

With $\mathsf{rule}(d)^{-1}$ we denote the reverse rule

$$f(x_1, \ldots x_{i-1}, g(x_i, \ldots, x_{i+l-1}), x_{i+l}, \ldots x_n)) \to d(x_1, \ldots x_n).$$

The right-hand side of the rule $\mathsf{rule}(d)$ can be seen as the tree pattern represented by the digram d.

▶ **Definition 5.** A *compressed term list* is a list $(t_1, \ldots, t_m \mid d_1, \ldots, d_n)$, where

- for each $1 \le i \le n$, d_i is a digram over the signature $\Sigma \cup \{d_1, \ldots, d_{i-1}\}$, and
- for each $1 \le i \le m$, $t_i \in \mathrm{Term}(\Sigma \cup \{d_1, \ldots, d_n\}, V)$.

The idea is that a compressed term list $(t_1, \ldots, t_m \mid d_1, \ldots, d_n)$ represents the term list $(s_1, \ldots, s_m)$ that is obtained by replacing nodes labeled with digram symbols by the corresponding tree patterns (of size 2). This motivates the following definition:

▶ **Definition 6.** The *expansion* of a compressed term list $(t_1, \ldots, t_m \mid d_1, \ldots, d_n)$ is the list $(s_1, \ldots, s_m)$ where s_i is the unique normal form of t_i w.r.t. to the (confluent and terminating) term rewriting system $\{\mathsf{rule}(d_1), \ldots, \mathsf{rule}(d_n)\}$.

▶ **Example 7.** Let $\mathrm{rk}(h) = \mathrm{rk}(c) = 2$, $\mathrm{rk}(s) = 1$, $\mathrm{rk}(0) = 0$, and consider the following compressed term list:

$$([h, 2, c](x, y, z),\ [[h, 1, c], 1, s](y, x, z),\ [[h, 1, c], 1, s](x, c(s(0), y), z),\ [h, 2, c](y, s(0), c(x, z)) \mid [h, 1, c],\ [h, 2, c],\ [[h, 1, c], 1, s])$$

The expansion of this list is the following term list consisting of the terms from Example 2:

$$(h(x, c(y, z)),\ h(c(s(y), x), z),\ h(c(s(x), c(s(0), y)), z),\ h(y, c(s(0), c(x, z)))). \tag{3}$$

Figure 2 shows the replaced digrams from the above list.

In our applications, a term list will be a list of all left-hand and right-hand sides of a TRS. If term (list) compression is the main objective, then the goal would be to compute a small compressed term list whose expansion is the input term list. For this let us define the size of a compressed term list $(t_1, \dots, t_m \mid d_1, \dots, d_n)$ as $\sum_{i=1}^{m} |t_i| + n$. This definition is justified by the fact that a digram can be stored by two symbols (either symbols from the initial signature or references to previously defined digrams) and an integer, which needs constant space in a uniform cost model. In Example 7 the compressed term list has size 25, whereas the expanded term list has size 28.

From the results of [2] it follows immediately that it is NP-complete to check whether for a given term list $\bar{t} = (t_1, \dots, t_m)$ and a given number k there exists a compressed term list of size at most k whose expansion is $\bar{t}$. This holds even for the special case that all symbols in $\bar{t}$ are unary and $m = 1$ (in this case $\bar{t} = t_1$ is basically a string and a compressed term list is a context-free grammar in Chomsky normal form that only produces t_1).

In [12], the algorithm TreeRePair for producing a small compressed term list was presented. Let us briefly explain the idea, for which we need a few definitions. Fix a digram $d = [f, i, g]$ and a term list $\bar{t} = (t_1, \dots, t_m)$, where $t_j = (D_j, \lambda_j)$. An *occurrence* of d in $\bar{t}$ is a pair (j, p) where $1 \le j \le m$, $p \in D_j$, $\lambda_j(p) = f$, and $\lambda_j(pi) = g$. A set Occ of occurrences of d in $\bar{t}$ is *non-overlapping* if for every $(j, p) \in \mathsf{Occ}$ we have $(j, pi) \notin \mathsf{Occ}$. Clearly, if $f \neq g$, then every set of occurrences is non-overlapping. If Occ is non-overlapping then we can apply in each term t_j simultaneously at all positions p with $(j, p) \in \mathsf{Occ}$ the rewrite rule $\mathsf{rule}(d)^{-1}$. Let t'_j be the resulting term. We write $(t_1, \dots, t_m) \to_{\mathsf{Occ}} (t'_1, \dots, t'_m)$.

Let $\mathsf{max}(d, \bar{t})$ be the maximal size among all non-overlapping sets of occurrences of d in $\bar{t}$. We can easily determine a non-overlapping set $\mathsf{maxOcc}(d, \bar{t})$ of occurrences of d in $\bar{t}$ such that $|\mathsf{maxOcc}(d, \bar{t})| = \mathsf{max}(d, \bar{t})$: If $f \neq g$ then we can set $\mathsf{maxOcc}(d, \bar{t})$ to the set of all occurrences of d in $\bar{t}$. On the other hand, if $f = g$, then we obtain $\mathsf{maxOcc}(d, \bar{t})$ as follows. For every maximal chain of positions $p, pi, pii, \dots, pi^k \in D_j$ in a term $t_j = (D_j, \lambda_j)$ from our list such that $\lambda_j(pi^\ell) = f$ for all $0 \le \ell \le k$ we do the following: If k is odd, then we add the pairs $(j, p), (j, pi^2), \dots, (j, pi^{k-1})$ to $\mathsf{maxOcc}(d, \bar{t})$. If k is even, then we add the pairs $(j, p), (j, pi^2), \dots, (j, pi^{k-2})$ to $\mathsf{maxOcc}(d, \bar{t})$. In the case that k is even we could have also put the pairs $(j, pi), (j, pi^3), \dots, (j, pi^{k-1})$ into $\mathsf{maxOcc}(d, \bar{t})$; this choice would also lead to an occurrence set of size $\mathsf{max}(d, \bar{t})$. We prefer $(j, p), (j, pi^2), \dots, (j, pi^{k-2})$ since this is the better choice for our adaptation MCTreeRePair of TreeRePair described in Section 5. A high-level description of the TreeRePair algorithm looks as follows:

```
input: a term list $\bar{t} = (t_1, \dots, t_m)$
$\bar{d} := \varepsilon$ (a list of digrams)
while there exists a digram $d$ with $\mathsf{max}(d, \bar{t}) > 1$ do
    let $d$ be a digram with $\mathsf{max}(d, \bar{t}) \ge \mathsf{max}(d', \bar{t})$ for all digrams $d'$
    let $\bar{u}$ such that $\bar{t} \to_{\mathsf{maxOcc}(d, \bar{t})} \bar{u}$
    $\bar{t} := \bar{u}$; $\bar{d} := (\bar{d}, d)$
endwhile
output: $(\bar{t} \mid \bar{d})$
```

In [12] the user of TreeRePair may specify a parameter r with the following meaning: Only digrams d with $\mathrm{rk}(d) \le r$ will be considered in each iteration of the algorithm. This has two advantages:

- For the test data in [12] (large XML tree structures) it leads to a better compression ratio, see [12] for an explanation of this possibly surprising fact. The optimal value turned out to be $r = 4$.

- Bounding the maximal rank of digrams improves the running time of TreeRePair drastically.

Let us briefly explain the second point: A naive implementation of the above algorithm, where we count digram frequencies in each iteration of the while-loop from scratch needs quadratic time. The implementation from [12] avoids this; thereby some occurrences of self-overlapping digrams of the form $[f, i, f]$ get lost (as explained at the end of this section). But the resulting negative effect on the compression ratio turned out to be negligible. We recall some implementation details from [12], since our implementation of MCTreeRePair uses the same principles.

The input terms from $\bar{t}$ are represented as pointer structures, where every node stores a pointer to its parent node and a list of pointers to its children. An occurrence (j, p) of d in $\bar{t}$ is simply represented by a pointer to node p of t_j. Initially, for every digram d all occurrences from the set $\mathsf{maxOcc}(d, \bar{t})$ are inserted into a doubly linked list (one for each digram) and the size of $\mathsf{maxOcc}(d, \bar{t})$ (which is $\mathsf{max}(d, \bar{t})$) is counted. This can be done in a single pass over the input term list $\bar{t}$. If the total number of nodes in the input term list $\bar{t}$ is n then clearly at most n digram replacements can be done in total. Each time an occurrence (j, p) of the digram $d = [f, i, g]$ is replaced, the following steps are done:

- Delete the node pi of t_j, set the parent pointer for every children pik $(1 \leq k \leq \mathrm{rk}(g))$ of pi to p, insert the list of child pointers of node pi into the list of child pointers of node p, and change the label of p to d.
- Decrement the count-value for digrams d' that overlap the replaced occurrence of d at position p, and remove the overlapping occurrences of d' from the current list of d'-occurrences.
- Increment the count-value for those digrams d' that are introduced by the replacement step (digrams of the form $[h, l, d]$ or $[d, l, h]$), and insert new occurrences of d' into the list of d'-occurrences.

Assume that $\mathrm{rk}(f) = m$ and $\mathrm{rk}(g) = n$. Then at most $m + n$ digram occurrences overlap the replaced occurrence of d at position p. The rank of d is $m + n - 1$. Hence, if we only replace digrams of rank at most r, then at most $r + 1$ digram occurrences overlap the replaced occurrence of d at position p. Hence, per digram replacement only a constant number of updates is necessary.

A problem arises with self-overlapping digrams of the form $[f, i, f]$. Consider for instance the term $f(a, f(b, f(c, d)))$. The occurrence of $[f, 2, f]$ at the root position ε would belong to the set $\mathsf{maxOcc}([f, 2, f], \bar{t})$, whereas the second occurrence at position 2 would not. Now assume that we replace the digram $[f, 1, a]$ (by adding further terms to the term list, $[f, 1, a]$ might become the most frequent digram). We obtain the term $A_1(f(b, f(c, d)))$. If we would compute the set $\mathsf{maxOcc}([f, 2, f], \bar{t})$ from scratch, the occurrence of $[f, 2, f]$ at position 1 in the term $A_1(f(b, f(c, d)))$ would be inserted into the set $\mathsf{maxOcc}([f, 2, f], \bar{t})$. But in the implementation from [12] described above this occurrence is lost.

5 Minimizing the matrix multiplication cost using TreeRePair

In this section we present our adaptation of TreeRePair with the goal of reducing the matrix multiplication cost of a given term list. We call our algorithm MCTreeRePair ("MC" for "matrix cost"). The point is that a compressed term list may have smaller cost than its expansion. First, we have to define the cost of a digram:

▶ **Definition 8.** The *cost* of digram $d = [f, i, g]$ is $\mathrm{cost}(d) = \mathrm{rk}(g)$.

Note that $\mathrm{cost}(d) = \mathrm{rk}(g)$ is exactly the number of matrix multiplications needed to compute the coefficients for the linear function that is represented by d.

▶ **Definition 9.** The *(matrix multiplication) cost* of a compressed term list $(t_1, \ldots, t_m \mid d_1, \ldots, d_n)$ is

$$\mathrm{cost}(t_1, \ldots, t_m \mid d_1, \ldots, d_n) = \sum_{i=1}^{m} \mathrm{cost}(t_i) + \sum_{i=1}^{n} \mathrm{cost}(d_i). \tag{4}$$

If $(s_1, \ldots, s_m)$ is the expansion of $(t_1, \ldots, t_m \mid d_1, \ldots, d_n)$ then we can compute the coefficients for the linear functions $[s_1], \ldots, [s_m]$ with $\mathrm{cost}(t_1, \ldots, t_m \mid d_1, \ldots, d_n)$ many matrix multiplications.

▶ **Example 10.** Let us compute the cost of the compressed term list from Example 7. We have:

$$\begin{aligned} \mathrm{cost}([h,2,c](x,y,z)) &= 0 & \mathrm{cost}([h,1,c]) &= 2 \\ \mathrm{cost}([[h,1,c],1,s](y,x,z)) &= 0 & \mathrm{cost}([h,2,c]) &= 2 \\ \mathrm{cost}([[h,1,c],1,s](x,c(s(0),y),z)) &= 1 & \mathrm{cost}([[h,1,c],1,s]) &= 1 \\ \mathrm{cost}([h,2,c](y,s(0),c(x,z))) &= 2 \end{aligned}$$

Hence, the total cost is 8. In contrast, the cost of the expanded term list in (3) is 13.

▶ **Definition 11.** The *savings* of a non-overlapping set of occurrences Occ of a digram $d = [f, i, g]$ in a term list $\bar{t} = (t_1, \ldots, t_m)$, briefly $\mathsf{save}(\mathsf{Occ}, \bar{t})$, is defined as follows, where $t_j = (D_j, \lambda_j)$:

$$\mathsf{save}(\mathsf{Occ}, \bar{t}) = -\mathrm{cost}(d) + \sum_{(j,p)\in\mathsf{Occ}} |\operatorname{Var}(t_j|_{pi})|. \tag{5}$$

In other words: We add to the negative cost of d for each $(j,p) \in \mathsf{Occ}$ the number of different variables below the i-th child of node p (the lower digram node).

By the following lemma, $\mathsf{save}(\mathsf{Occ}, \bar{t})$ is exactly the cost-reduction obtained by replacing all digram occurrences from Occ.

▶ **Lemma 12.** *Let $(t_1, \ldots, t_m \mid d_1, \ldots, d_n)$ be a compressed term list, let $d = [f, i, g]$ be a digram, and let Occ be a non-overlapping set of occurrences of d in $(t_1, \ldots, t_m)$. Let $(t_1, \ldots, t_m) \to_{\mathsf{Occ}} (t'_1, \ldots, t'_m)$. Then we have*

$$\mathrm{cost}(t'_1, \ldots, t'_m \mid d_1, \ldots, d_n, d) = \mathrm{cost}(t_1, \ldots, t_m \mid d_1, \ldots, d_n) - \mathsf{save}(\mathsf{Occ}, [t_1, \ldots, t_m]). \tag{6}$$

Proof. Let $\mathsf{Occ}_j = \{p \mid (j,p) \in \mathsf{Occ}\}$. Using (4) and (5), it follows that (6) is equivalent to

$$\sum_{j=1}^{m} \mathrm{cost}(t_j) = \sum_{j=1}^{m} \mathrm{cost}(t'_j) + \sum_{(j,p)\in\mathsf{Occ}} |\operatorname{Var}(t_j|_{pi})|.$$

This follows from

$$\mathrm{cost}(t_j) = \mathrm{cost}(t'_j) + \sum_{p\in\mathsf{Occ}_j} |\operatorname{Var}(t_j|_{pi})|$$

for all $1 \le j \le m$. But this is a consequence of (2). Applying rule $\mathsf{rule}(d)^{-1}$ at all positions $p \in \mathsf{Occ}_j$ in t_j means that we remove all nodes pi with $p \in \mathsf{Occ}_j$ from t_j. Moreover, for all other nodes of t_j the number of different variables below the node does not change. Also note that for all $p \in \mathsf{Occ}_j$, node pi of t_j is not labeled with a variable. ◀

By Lemma 12, in order to reduce the cost of a (compressed) term list maximally, we have to find a non-overlapping set of occurrences (of some digram d) with maximal savings.

▶ **Definition 13.** For the digram $d = [f, i, g]$ and the term list $\overline{t} = (t_1, \dots, t_m)$ let $\mathsf{maxsave}(d, \overline{t})$ be the maximum of $\mathsf{save}(\mathsf{Occ}, \overline{t})$, where we maximize over all non-overlapping sets of occurrences Occ of d in $\overline{t}$.

Recall the definition of the non-overlapping occurrence set $\mathsf{maxOcc}(d, \overline{t})$ from Section 4.

▶ **Lemma 14.** *We have* $\mathsf{save}(\mathsf{maxOcc}(d, \overline{t}), \overline{t}) = \mathsf{maxsave}(d, \overline{t})$.

Proof. Let $d = [f, i, g]$. The case $f \neq g$ is clear, since then $\mathsf{maxOcc}(d, \overline{t})$ is the set of all occurrences of d in $\overline{t}$. Now assume that $f = g$. Recall that we obtain $\mathsf{maxOcc}(d, \overline{t})$ by considering all maximal chains of positions $p, pi, pii, \dots, pi^k \in D_j$ in a term $t_j = (D_j, \lambda_j)$ from our list such that $\lambda_j(pi^\ell) = f$ for all $0 \le \ell \le k$. If k is odd, then we put the occurrences $(j, p), (j, pi^2), \dots, (j, pi^{k-1})$ into $\mathsf{maxOcc}(d, \overline{t})$. If k is even, then we put the occurrences $(j, p), (j, pi^2), \dots, (j, pi^{k-2})$ into $\mathsf{maxOcc}(d, \overline{t})$. Note that in case k is even, the set of occurrences $\{(j, pi), (j, pi^3), \dots, (j, pi^{k-1})\}$ has the same size as the chosen set of occurrences $\{(j, p), (j, pi^2), \dots, (j, pi^{k-2})\}$. But the latter gives a larger (or the same) savings according to (5), since $\mathrm{Var}(t_j|_{pi^{\ell+1}}) \subseteq \mathrm{Var}(t_j|_{pi^\ell})$ and thus $|\mathrm{Var}(t_j|_{pi^{\ell+1}})| \le |\mathrm{Var}(t_j|_{pi^\ell})|$ for all $0 \le \ell \le k-1$. ◀

On a high level, MCTreeRePair works as follows:

```
input: a term list $\overline{t} = (t_1, \dots, t_m)$
$\overline{d} := \varepsilon$ (a list of digrams)
while there exists a digram $d$ with $\mathsf{maxsave}(d, \overline{t}) > 0$ do
    let $d$ be a digram with $\mathsf{maxsave}(d, \overline{t}) \ge \mathsf{maxsave}(d', \overline{t})$ for all digrams $d'$
    let $\overline{u}$ such that $\overline{t} \to_{\mathsf{maxOcc}(d, \overline{t})} \overline{u}$
    $\overline{t} := \overline{u}$; $\overline{d} := (\overline{d}, d)$
endwhile
output: $(\overline{t} \mid \overline{d})$
```

Here is a complete example run of MCTreeRePair.

▶ **Example 15.** Let $\mathrm{rk}(h) = \mathrm{rk}(c) = 2$, $\mathrm{rk}(s) = 1$ and $\mathrm{rk}(0) = 0$. Consider the following term rewriting system (which consists of the terms from Example 2):

$$h(x, c(y, z)) \to h(c(s(y), x), z), \qquad h(c(s(x), c(s(0), y)), z) \to h(y, c(s(0), c(x, z)))$$

The matrix multiplication cost is 13, see Example 2. The $\mathsf{maxsave}$-values of the digrams in this system are (we omit the second parameter in $\mathsf{maxsave}$ for the term list):

$$\mathsf{maxsave}(h, 1, c) = 2, \quad \mathsf{maxsave}(c, 1, s) = 1, \quad \mathsf{maxsave}(s, 1, 0) = 0$$
$$\mathsf{maxsave}(h, 2, c) = 2, \quad \mathsf{maxsave}(c, 2, c) = 1,$$

Let us decide to replace the digram $d := (h, 1, c)$ (we could also choose $(h, 2, c)$). We obtain the following system:

$$h(x, c(y, z)) \to d(s(y), x, z), \qquad d(s(x), c(s(0), y), z) \to h(y, c(s(0), c(x, z)))$$

The new $\mathsf{maxsave}$-values are:

$$\mathsf{maxsave}(h, 2, c) = 2, \quad \mathsf{maxsave}(c, 2, c) = 0, \quad \mathsf{maxsave}(c, 1, s) = -1$$
$$\mathsf{maxsave}(d, 1, s) = 1, \quad \mathsf{maxsave}(d, 2, c) = -1,$$

So, we next replace the digram $e := (h, 2, c)$ and obtain the system:

$$e(x, y, z) \to d(s(y), x, z), \qquad d(s(x), c(s(0), y), z) \to e(y, s(0), c(x, z))$$

At this point, $f := (d, 1, s)$ is the only diagram with a strictly positive $\mathsf{maxsave}$-value, namely 1. Hence, we replace this digram and get the final compressed system

$$e(x, y, z) \to f(y, x, z), \qquad f(x, c(s(0), y), z) \to e(y, s(0), c(x, z)).$$

Our implementation of MCTreeRePair follows the same strategy as TreeRePair described in the previous section. In a first step we compute for each node p of a tree t_j in the input term list $\bar{t}$ the number $|\operatorname{Var}(t_j|_p)|$ of different variables in the subtree rooted at p. These numbers are necessary to compute the savings of a set of digram occurrences according to (5). Then, as for standard TreeRePair, we insert for every digram d all occurrences from the set $\mathsf{maxOcc}(d, \bar{t})$ into a doubly linked list (one for each digram). Thereby, we also compute the savings $\mathsf{maxsave}(d, \bar{t}) = \mathsf{save}(\mathsf{maxOcc}(d, \bar{t}), \bar{t})$ according to (5). Note that when we replace a certain occurrence (j, p) of the digram $d = [f, i, g]$ in the tree t_j, and the resulting tree is t'_j (i.e., we apply the inverse rule $\mathsf{rule}(d)^{-1}$ at position p in t_j), then we do not have to recompute the numbers $|\operatorname{Var}(t'_j|_q)|$ (for all nodes q of t'_j): In our implementation of the digram replacement, we remove the node pi from the pointer structure representing t_j, The new parent node of pik ($1 \le k \le \operatorname{rk}(g)$) becomes node p. Thereby, the number of different variables below a certain node does not change.

One might try to reduce the computation of the $\mathsf{maxsave}(d, \bar{t})$-values to ordinary digram counting (as done in TreeRePair) as follows: Let t be a term, let u be a subterm of t and let x be a variable. We say that a subterm $s \trianglelefteq t$ is a *maximal non-x subterm of* t if x does not occur in s, but x occurs in every subterm u with $s \triangleleft u$. We denote by t_x the term in which all maximal non-x subterms have been replaced by some dummy symbol N.

▶ **Example 16.** Let $t = f(g(x), h(x, s(y)))$. Then $t_x = f(g(x), h(x, N), t_y = f(N, h(N, s(y))$.

For a term t, let $\{x_1, \dots, x_n\}$ be the set of variables occurring in t. Then one can check that $\mathsf{maxsave}(d, t)$ equals the number of non-overlapping occurrences of d in the term list $(t_{x_1}, \dots, t_{x_n})$ minus the cost of d. However this can lead to a quadratic blowup of the input terms. To see that, consider the term

$$t_n = \underbrace{f(\dots(f}_{n \text{ times}}\, g\, \underbrace{(x_1, \dots, x_n)}_{n \text{ times}})).$$

where f has rank 1 and g has rank n. This tree has size $2n+1$, but the term list $(t_{x_1}, \dots, t_{x_n})$ has total size $n(2n + 1)$.

6 Compression and the dependency pairs transformation

The dependency pairs transformation [1] converts the full termination problem of R over Σ into the top termination problem of $\operatorname{DP}(R)$ relative to R, over signature $\Sigma \cup \Sigma^{\#}$, where

$$\operatorname{DP}(R) := \{l^{\#} \to s^{\#} \mid (l \to r) \in R, s = (D, \lambda) \trianglelefteq r, \lambda(\varepsilon) \in \operatorname{defined}(R), s \not\triangleleft l\},$$

where $\operatorname{defined}(R)$ is the set of all symbols that occur in root positions of left-hand sides of rules of R, and the operation of *marking* the top symbol is $f(t_1, \dots, t_n)^{\#} = f^{\#}(t_1, \dots, t_n)$.

▶ **Example 17.** For (the string rewrite system) $R = \{a^2b^2 \to b^3a^3\}$, we obtain $\operatorname{defined}(R) = \{a\}$ and $\operatorname{DP}(R) = \{a^{\#}ab^2 \to a^{\#}a^2, a^{\#}ab^2 \to a^{\#}a, a^{\#}ab^2 \to a^{\#}\}$.

When using monotone matrix interpretations to prove relative top termination, one uses two-sorted interpretations (called *weakly monotone algebras* [6]). An m-ary marked symbol $f^\#$ is interpreted by a linear function $[f^\#] : (S^n)^m \to S$, whereas an m-ary unmarked symbol f is interpreted by a linear function $[f] : (S^n)^m \to S^n$.

We now discuss how two-sortedness affects compression. We observe that the cost model that just counts matrix multiplications is wrong for computing interpretations for $\mathrm{DP}(R)$, and consequently MCTreeRePair produces inefficient results. This is shown in the following example.

▶ **Example 18.** Let $f^\#$ be a marked binary symbol and g, h be unmarked unary symbols. The interpretation of $f^\#$ has coefficient matrices $F_1^\#, F_2^\#$ (of dimension $1 \times n$), and the interpretation of g (resp., h) has a coefficient matrix G_1 (resp., H_1) of dimension $n \times n$. Consider the computation of the coefficient for variable x in the term $t = f^\#(g(h(x)), g(h(x)))$. It is tempting to first replace the two occurrences of the digram $e = (g, 1, h)$. So we compute $E_1 = G_1 H_1$ (a product of two $(n \times n)$-matrices) with n^3 elementary multiplications. Then we compute the coefficient of x in t as $F_1^\# E_1 + F_2^\# E_1$, which needs another $2n^2$ elementary multiplications (this can be reduced to n^2 multiplications by computing $(F_1^\# + F_2^\#)E_1$). Hence, in total we need $n^3 + 2n^2$ (or $n^3 + n^2$ if we use the alternative) multiplications.

But there is a better way: Compute $(F_1^\# G_1)H_1$, that is, first multiply $F_1^\#$ by G_1 and then the result by H_1, and similarly $(F_2^\# G_1)H_1$, which needs in total only $4n^2$ multiplications. In terms of digrams, this means that we replace the following digrams (which occur only once) in this order: $c := (f^\#, 1, g)$, $d := (c, 1, h)$, $e := (d, 2, g)$, $f := (e, 2, h)$.

The example shows that computation of the interpretation of marked terms is best done "from the top". In this way we can avoid multiplications of two $(n \times n)$-matrices. We draw the conclusion that we should compress only R, because that is where the expensive multiplications take place.

We now describe how we obtain compression for $\mathrm{DP}(R)$ as a side effect. The reason is that all terms in $\mathrm{DP}(R)$ have shape $f^\#(t_1, \ldots, t_n)$ where each t_i is a subterm of some term in R. This implies that we can extract a compressed version of t_i.

We compress R over Σ, obtaining a compressed system R_c over the extended alphabet Σ_c consisting of Σ and digrams. We then compute the compressed version of $\mathrm{DP}(R)$ by applying a modified operation DP_c on the compressed system R_c. This operation $\mathrm{DP}_c(R_c)$ has two ingredients:

- computation of the set of all subterms (in compressed form) of a compressed term, and
- marking of the top symbol of a compressed term.

In both cases the output term(s) should be compressed, and be obtained without completely unpacking the input term. These operations can be realized in a straightforward manner by expanding digrams as needed, at the current position. We make no attempts at constructing fresh digrams.

▶ **Example 19.** Let f be a unary and $t = f^8(x)$. Consider the compressed term $t_c = d_3(x)$ with digrams $d_3 = (d_2, 1, d_2), d_2 = (d_1, 1, d_1), d_1 = (f, 1, f)$. The proper subterms of t in compressed form are $f d_1 d_2(x)$, $d_1 d_2(x)$, $f d_2(x)$, $d_2(x)$, $f d_1(x)$, $d_1(x)$, $f(x)$, x, and $d_3(x)^\#$ is $f^\# f d_1 d_2(x)$.

▶ **Example 20** (Example 17 continued)**.** For $R = \{a^2 b^2 \to b^3 a^3\}$ compression produces $R_c = \{de \to ebda\}$ with digrams $d = (a, 1, a), e = (b, 1, b)$. We obtain $\mathrm{DP}_c(R_c) = \{a^\# ae \to a^\# a^2, a^\# ae \to a^\# a, a^\# ae \to a^\#\}$. Note that the right-hand side $a^\# a^2$ of the first rule gets $a^\# a$ from marking the expansion of the digram d, plus an adjacent a.

To compute efficiently the interpretations of marked terms, like $f^\# f d_1 d_2(x)$ from Example 19, we work from the top, i.e., left-to-right. This can be modelled by the introduction of digrams $e_1 = (f^\#, 1, f)$, $e_2 = (e_1, 1, d_1)$, $e_3 = (e_2, 1, d_2)$. The interpretations of these digrams are linear functions from S^n to S. For the computation of the coefficients of these linear functions, we have to multiply a $(1 \times n)$-matrix with a $(n \times n)$-matrix (but never two $(n \times n)$-matrices). We therefore compress $\mathrm{DP}_c(R_c)$ by repeatedly replacing digrams that occur at the root of some term from $\mathrm{DP}_c(R_c)$. The algorithm stops when all children of all top symbols are variables.

We summarize the DP-MCTreeRePair algorithm:

```
input: a rewriting System $R$ over $\Sigma$
$R_c := \mathrm{MCTreeRePair}(R)$
$D_c := \mathrm{DP}_c(R_c)$
while there exists a digram $d$ that occurs at the top of some rule in $D_c$
    $D_c :=$ replace each occurence of $d$ in lhs and rhs of $D_c$
endwhile
output: $(D_c, R_c)$ such that $\mathrm{expand}(D_c) = \mathrm{DP}(R)$ and $\mathrm{expand}(R_c) = R$.
```

▶ **Example 21** (Example 20 continued). In $\mathrm{DP}_c(R_c)$ we introduce digrams $d_1 = (a^\#, 1, a), d_2 = (d_1, 1, e), d_3 = (d_1, 1, a)$, and obtain $\{d_2 \to d_3, d_2 \to d_1, d_2 \to a^\#\}$.

We now compare the number of matrix and vector operations for different compression methods applied with the dependency pairs transformation. We compare to a "naive" compression method as well.

▶ **Example 22.** For the symbolic evaluation of an n-dimensional matrix interpretation for the rewriting system from Example 2, Table 1 contains in column (p, q, r) the number of multiplications of a $(p \times q)$-matrix by a $(q \times r)$-matrix.

Table 1 Number of matrix multiplications for the rewriting system from Example 2.

method	$(1, n, 1)$	$(1, n, n)$	$(n, n, 1)$	(n, n, n)
uncompressed $(\mathrm{DP}(R) \cup R)$	4	8	20	18
$\mathrm{MCTreeRePair}(\mathrm{DP}(R) \cup R)$	4	8	13	12
$\mathrm{DP\text{-}MCTreeRePair}(R)$	9	11	9	8

By applying algorithm DP-MCTreeRePair, the number of matrix-by-matrix multiplications is lowest—in fact it is equal to the number of matrix-by-matrix multiplications of $\mathrm{MCTreeRePair}(R)$.

7 Experiments

We implemented a version of MCTreeRePair as described in Sections 5 and 6, and we evaluated our implementation in two settings:

- We evaluated how compression reduces the size of constraint systems for rewrite systems from the Termination Problems Data Base (more precisely, the SRS/TRS standard/relative subsets of TPDB version 8), which consists of 3027 files.
- We determined the influence of compression on the power of an actual termination prover.

The source code is available from `https://github.com/jwaldmann/matchbox`. The complete experimental data (log files) is available from `http://www.informatik.uni-leipzig.de/~noeth/`

To measure the "compressibility" of TPDB problems, we used the matrix multiplication cost from (2) as well as the actual size of the resulting SAT constraint system with fixed parameters for the matrix dimension and the bit width of matrix entries. We compared these measures for the settings with and without compression, for both the original systems and for their DP-transformed versions. Table 2 shows the results. Column "cost" shows the accumulated costs of all terms from the corpus. Column "CNF-size" shows the accumulated number of variables and clauses that are being generated by the "bit blasting" translation from the (arctic) integer constraint problem. For "no compression" and "compression" we use (3×3)-matrices with 3-bit entries, for "DP" and "DP and compression" we use (3×3)-matrices with 5-bit entries. Note that we obtain an overall compression ratio of about 3 for both the matrix multiplication cost and the actual CNF size (number of clauses). For DP, these ratios are 3.44 and 1.71, respectively. We conclude that our cost model gives, on average, a very good approximation of the real cost.

Table 2 Total cost and CNF-size with and without compression, for 3027 systems from TPDB.

method	cost	CNF-size (variables, clauses)
no compression	$1.61 \cdot 10^6$	$4.04 \cdot 10^8, 3.23 \cdot 10^9$
compression	$5.18 \cdot 10^5$	$1.30 \cdot 10^8, 1.04 \cdot 10^9$
dependency pairs (DP)	$1.51 \cdot 10^6$	$1.92 \cdot 10^9, 6.22 \cdot 10^9$
DP and compression	$4.39 \cdot 10^5$	$1.11 \cdot 10^9, 3.63 \cdot 10^9$

For estimating the effect of compression on the performance of a termination prover, we used a restricted version of *matchbox*. It optionally applies the dependency pairs transformation and then repeats the following steps until there are no more strict rules:

- If the system is linear, remove rules by additive weights (linear polynomials of slope 1 with absolute coefficients computed by the GLPK solver for linear inequalities).
- For increasing matrix dimensions, try to remove rules by natural matrix interpretations for original systems [6] (solved by binary bit-blasting) and arctic matrix interpretations for DP-transformed systems [10] (solved by unary bit-blasting [3]). In both cases, MINISAT [5] is used as the backend solver.

We apply the "cheap" method (additive weights) first so that the remaining constraint systems are non-trivial. We isolate the effect of compression by using matrix interpretations as the only non-cheap method.

Our experiment then consists of a comparison between the performances of an implementation with and without compression. The following parameters are fixed at the beginning: the boolean encoding of numbers (in particular, their bit width), the matrix dimensions that are being used, the compiler settings, runtime settings, and resources of the execution platform (timeout, memory size, cores). We choose "sensible" values for these parameters, but make no particular attempt to optimize them.

Our implementation exploits parallelism: We search for matrix interpretations in dimensions $1, 2, \ldots, D$ in parallel (for some parameter D that is fixed in advance), i.e., we generate constraint systems $C_1, C_2, \ldots, C_D$ and submit each of them to a separate instance of the SAT solver. As soon as one C_i is solved, we stop the other computations, remove some rules from the input problem (according to the interpretation that was obtained as the solution of C_i), and start afresh. In this way, we actually measure the time that the constraint solver needs in the positive case(s) only. Compare this with a sequential implementation, where we would have to wait for C_i to be recognized as unsolvable, before attempting to solve C_{i+1}. In this case, the total time would include several unsuccessful attempts as well. But in reality (of proving termination automatically), we are not interested in unsolvable C_i, because they do not contain information on the termination problem. (We cannot distinguish between unsatisfiability due to non-termination, or due to insufficient bit width.)

Table 3 shows the results of our experiments. Column "# yes instances" shows the number of rewrite systems for which termination is successfully proven within 1 minute (the time out). Column "average time yes" is the average time needed to prove termination overall yes instances. It shall be noted that the number of "# yes instances" includes the number of systems for which termination could be proven by "cheap" methods, as described above (there were 50 such cases without using the dependency pairs method, and 250 with it). As can be seen, the number of systems which can be proven to be terminating increases by about 7% (3,5% for DP) when using MCTreeRepair-compression. We conclude that compression of rewriting systems using MCTreeRePair does improve the power of a termination prover that uses a constraint solver to find interpretations.

Table 3 also shows our results when "naive" compression based on TreeRePair (as outlined in Section 4) instead of MCTreeRePair is used in the termination prover. Surprisingly, the number of systems for which termination can be proven is less than without any compression. Here is a possible explanation: When computing the interpretation of a term t without variables bottom-up, only cheap matrix-by-vector multiplications are needed; expensive matrix-by-matrix multiplications do not occur. But compression based on ordinary TreeRePair only tries to reduce the size of t and therefore may introduce diagrams which lead to expensive matrix-by-matrix multiplications when evaluating the digrams. On the other hand, MCTreeRePair will not introduce any diagrams in t: Every digramm occurrence $d = [f, i, g]$ in t, where g has at least arity 1 yields the negative contribution $-\operatorname{cost}(d) = -\operatorname{rk}(g)$ to the savings according to (5).

Table 3 Influence of compression on the matchbox termination prover.

method	average time yes	# yes instances
no compression	11.9	584
compression with MCTreeRePair	12.2	628
naive compression with TreeRePair	11.9	571
dependency pairs (DP)	1.85	681
DP and compression	4.10	709

All values in Table 3 were obtained for an unlimited maximal rank for diagrams. We also experimented with bounded maximal ranks, and it turned out that the optimal value (w.r.t. resulting number of termination proofs for Matchbox using DP-MCTreeRePair) seems to be $r = 4$ (whis is also the optimal value for XML-compression based on TreeRePair in [12]): The number of proofs is slightly larger than with unbounded rank, and we have no explanation at the moment.

8 Discussion and summary

Does compression really preserve semantics? For any given interpretation of function symbols, the interpretation of a compressed term is equivalent to the interpretation of the original term. The underlying reason is that matrix multiplication is associative: digrams correspond to sub-multiplications.

When solving matrix constraints by bit-blasting, the range for matrix elements is a finite set, prescribed by the bit width of the encoding. This implies that arithmetical operations may overflow (For natural numbers, addition and multiplication may overflow; for arctic numbers, multiplication may), so they are partial functions. These partial functions are no longer associative: For instance, consider the integer product abc for three bit integers a, b and c. For $a = 7$, $b = 7$, $c = 0$ the product $a(bc)$ is representable, while the product $(ab)c$ is not. Now, (ab) could be a digram that occurs during compression, while (bc) could correspond to an evaluation of the uncompressed term. Then the constraint system generated from the compressed terms may be unsatisfiable, while the original system is satisfiable. Take the bit width w as a parameter. It can be shown that the original system O and the compressed system C are equivalent in the sense that for each satisfying assignment s of $O(w)$ there is some $w' \geq w$ such that a padded version s' of s satisfies $C(w')$.

Does compression work with more advanced termination methods? The basic dependency pairs method has many refinements [9, 8], which we ignore here since they appear orthogonal to the topic of compression. For instance, by using (estimated) dependency graphs, one obtains termination subproblems that refer to subsets of $\mathrm{DP}(R)$. The "usable rules" method creates subproblems that contain subsets of R. In both cases, compression can be obtained by the methods shown in Section 6.

Is the data base sufficient? We were running experiments on problems contained in the Termination Problems Data Base (TPDB). It may be argued that most of the problems in TPDB are small, and do not need compression. Our experiments show that even for small problems, compression may help. For instance, consider problem TRS/Gebhardt_06/02.trs. We apply the DP transformation, a simplex solver, and arctic matrix interpretations. The nontrivial part is to find a matrix interpretation of dimension 4. We use 5 bit arctic unary numbers. Without compression, we get 16 multiplications of (4×4)-matrices, resulting in a CNF with 45423 variables and 146770 clauses, which is solved in 18 seconds. With compression, we get only 7 matrix multiplications, the CNF has 22303 variables and 71970 clauses, and is solved in 10 seconds. Another point is that TPDB problems might not be typical "real life" termination problems. It appears that most of the TPDB problems are hand-crafted: They are taken from publications, where they serve to illustrate certain isolated points. So, they tend to be small but hard (and trivial only when one applies a specific, advanced method). Application problems, on the other hand, may be large but "easy", and appear hard only because of their size, and compression can reduce size.

Extensions. The method given in the paper counts matrix-by-matrix multiplications only. Our experiments confirm that this is a reasonable simplification. For still better compression, we additionally need to take into account the cost of vector-by-matrix multiplications at the top of DP-transformed rules, and also the matrix-by-vector multiplications for evaluating absolute parts. This implies extensions in the definition of the savings of a digram, and in the algorithm to incrementally update the savings information. In full generality, this includes the "matrix chain multiplication" optimization problem—and goes beyond it, since it is not just about parenthesizing matrix chains, but also about re-using subexpressions. We leave that as possible direction for future work.

Conclusion. We presented the MCTreeRePair algorithm, which reduces the size of constraint systems that determine matrix interpretations for automatically proving termination of rewriting. MCTreeRePair is based on the tree compression algorithm TreeRePair. To obtain a good compression for these constraint systems, we enriched TreeRePair with a cost function that is sensitive to the number of variables in subtrees. We showed that this additional information can be handled without too much extra work. We also showed that the dependency pairs transformation can be applied to compressed systems directly. We tested our implementation for problems from the Termination Problems Data Base. MCTreeRePair reduces the sizes of the resulting constraint systems by factor of approx. 1/3. We also provided experimental evidence showing that smaller constraint systems tend to be solved faster by a state-of-the art solver. To conclude, compression seems to be a useful addition to termination provers that use interpretation methods.

References

1 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

2 M. Charikar, E. Lehman, A. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf.Theory*, 51(7):2554–2576, 2005.

3 Michael Codish, Yoav Fekete, Carsten Fuhs, Jürgen Giesl, and Johannes Waldmann. Exotic semiring constraints. In *Proc. SMT 2012*, pages 87–96. `http://smt2012.loria.fr/SMT2012.pdf`, 2012.

4 Michael Codish, Igor Gonopolskiy, Amir M. Ben-Amram, Carsten Fuhs, and Jürgen Giesl. SAT-based termination analysis using monotonicity constraints over the integers. *TPLP*, 11(4-5):503–520, 2011.

5 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. SAT 2005*, LNCS 3569, pages 61–75. Springer, 2005.

6 Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.

7 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT 2007*, LNCS 4501, pages 340–354. Springer, 2007.

8 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *J. Autom. Reasoning*, 37(3):155–203, 2006.

9 Nao Hirokawa and Aart Middeldorp. Dependency pairs revisited. In *Proc. RTA 2004*, LNCS 3091, pages 249–268. Springer, 2004.

10 Adam Koprowski and Johannes Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybern.*, 19(2):357–392, 2009.

11 Dallas S Lankford. On proving term rewriting systems are noetherian. Memo MTP-3, Mathematics Department, Louisiana Tech. University, Ruston, LA, 1979.

12 Markus Lohrey, Sebastian Maneth, and Roy Mennicke. Tree structure compression with RePair. In *Proc. DCC 2011*, pages 353–362. IEEE Press, 2011.

13 Peter Schneider-Kamp, Carsten Fuhs, René Thiemann, Jürgen Giesl, Elena Annov, Michael Codish, Aart Middeldorp, and Harald Zankl. Implementing RPO and POLO using SAT. In *Deduction and Decision Procedures*, volume 07401 of *Dagstuhl Seminar Proceedings.* IBFI, Schloss Dagstuhl, Germany, 2007.