# Recompression: Technique for Word Equations and Compressed Data

Artur Jeż(✉)

University of Wrocław, Joliot-Curie 15, 50383 Wrocław, Poland
aje@cs.uni.wroc.pl

**Abstract.** In this talk I will present the recompression technique on the running example of word equations. In word equation problem we are given an equation $u = v$, where both $u$ and $v$ are words of letters and variables, and ask for a substitution of variables by words that equalizes the sides of the equation. The recompression technique is based on employing simple compression rules (replacement of two letters $ab$ by a new letter $c$, replacement of maximal repetitions of $a$ by a new letter), and modifying the equations (replacing a variable $X$ by $bX$ or $Xa$) so that those operations are sound and complete. The simple analysis focuses on the size of the instance and not on the combinatorial properties of words that are used. The recompression-based algorithm for word equations runs in nondeterministic linear space.

The approach turned out to be quite robust and can be applied to various generalized, simplified and related problems, in particular, to problems in the area of grammar compressed words. I will comment on some of those applications.

**Keywords:** Algorithms on automata and words · Word equations · Context unification · Equations in groups · Compression · SLPs

## 1 Introduction

### 1.1 Word Equations

The word equation problem, i.e. solving equations in the algebra of words, was first investigated by Markov in the fifties. In this problem we get as an input an equation of the form

$$u = v$$

where $u$ and $v$ are strings of letters (from a fixed alphabet) as well as variables and a *solution* is a substitution of words for variables that turns this formal equation into a true equality of strings of letters (over the same fixed alphabet). It is relatively easy to show a reduction of this problem to the Hilbert's 10-th problem, i.e. the question of solving systems of Diophantine equations. Already then it was generally accepted that Hilbert's 10-th problem is undecidable and Markov wanted to show this by proving the undecidability of word equations.


A. Leporati et al. (Eds.): LATA 2020, LNCS 12038, pp. 44–67, 2020.
https://doi.org/10.1007/978-3-030-40608-0_4

Alas, while Hilbert's 10-th problem is undecidable, the word equation problem is *decidable*, which was shown by Makanin [54]. The termination proof of his algorithm is very complex and yields a relatively weak bound on the computational complexity, thus over the years several improvements and simplifications over the original algorithm were proposed [27,29,43,79]. Simplifications have many potential advantages: it seems natural that simpler algorithm can be generalised or extended more easily (for instance, to the case of equations in groups) than a complex one. Moreover, simpler algorithm should be more effective in practical applications and should have a lower complexity bounds.

*Subcases.* It is easy to show NP-hardness for word equations, so far no better computational complexity lower bound is known. Such hardness stimulated a search for a restricted subclasses of the problem for which efficient (i.e. polynomial) algorithms can be given [4]. One of such subclasses is defined by restricting the amount of different variables that can be used in an equation: it is known that equations with one [11,45] and two [4,10,28] variables can be solved in polynomial time. Already for three variables it is not known, whether they are in NP or not [71] and partial results require nontrivial analysis [71].

*Generalisations.* Since Makanin's original solution much effort was put into extending his algorithm to other structures. Three directions seemed most natural:

- adding constraints to word equations;
- equations in free groups;
- partial commutation;
- equations in terms.

**Constraints.** From the application point of view, it is advantageous to consider word equations that can also use some additional constraints, i.e. we require that the solution for $X$ has some additional properties. This was first done for regular constraints [79], on the other hand, for several types of constraints, for instance length-constraints, it is still open, whether the resulting problem is decidable or not (it becomes undecidable, if we allow counting occurrences of particular letter in the substitutions and arithmetic operations on such counts [3]).

**Free groups.** From the algebraic point of view, the word equation problem is solving equations in a free semigroup. It is natural to try to extend an algorithm from the free semigroup also to the case of free groups and then perhaps even to a larger class of groups (observe, that there are groups and semigroups for which the word problem is undecidable). The first algorithm for the group case was given by Makanin [55,56], his algorithm was not primitively-recursive [44]. Furthermore, Razborov showed that this algorithm can be used to give a description of all solutions of an equation [68] (more readable description of the Razborov's construction is available in [41]). As a final comment, note that such a description was the first step in proving the Tarski's Conjecture for free groups (that the theory of free groups is decidable) [42].

**Partial commutation.** Another natural generalization is to allow partial commutation between the letters, i.e. for each pair of letters we specify, whether $ab = ba$ or not. Such partially commutative words are usually called traces and the corresponding groups are usually known as Right-Angled Artin Groups, RAAGs for short. Decidability for trace equations was shown by Matiyasevich [57] and for RAAGs by Diekert and Muscholl [15]. In both cases, the main step in the proof was a reduction from a partially commutative case to a non-commutative one.

**Terms.** We can view words as very simple terms: each letter is a function symbol of arity 1. In this way word equations are equations over (very simple) terms. It is known, that term unification can be decided in polynomial time, assuming that variables represent closed (full) terms [69]; thus such a problem is unlikely to generalise word equations.

A natural generalisation of term unification and word equations is a *second-order unification*, in which we allow variables to represent functions that take arguments (which need to be closed terms). However, it is known that this problem is undecidable, even in many restricted subcases [16,26,47,49]. *Context unification* [7,8,74] is a natural problem 'in between': we allow variables representing functions, but we insist that they use their argument *exactly once*. It is easy to show that such defined problem generalises word equations, on the other hand, the undecidability proofs for second-order unification do not transfer directly to this model.

Being a natural generalisation is not enough to explain the interest in this problem, more importantly, context unification has natural connections with other, well-studied problems (equality up to constraints [61], linear second-order unification [47,50], one-step term rewriting [62], bounded second order unification [76], ...). Unfortunately, for over two decades the question of decidability of context unification remained open. Despite intensive research, not much is known about the decidability of this problem: only results for some restricted subcases are known: [8,19,47,48,51,75,77,78].

### 1.2 Compression and Word Equations

For more than 20 years since Makanin's original solution there was very small progress in algorithms for word equations: the algorithm was improved in many places, in particular this lead to a better estimation of the running time; however, the main idea (and the general complexity of the proof) was essentially the same.

The breakthrough was done by Plandowski and Rytter [67], who, for the first time, used the compression to solve word equations. They showed, that the shortest solution (of size $N$) of the word equation (of size $n$) has an SLP representation of size $\mathsf{poly}(n, \log N)$; here a *Straight Line Programme* (SLP for short) is simply a context free grammar generating exactly one word. Using the algorithm for testing the equality of two SLPs [63] this easily yields a (non-deterministic) algorithm running in time $\mathsf{poly}(n, \log N)$. Unfortunately, this work did not provide any bound on $N$ and the only known bound (4 times exponential in $n$) came directly from Makanin's algorithm, together those two results

yielded a 3NEXPTIME algorithm. Soon after the bound on the size of the shortest solution was improved to triply exponential [27], which immediately yielded an algorithm from class 2NEXPTIME, however, the same paper [27] improved Makanin's algorithm, so that it workd in EXPSPACE.

Next, Plandowski gave a better (doubly exponential) bound on the size of the shortest solution [64] and thus obtained a NEXPTIME algorithm, in particular, at that time this was the best known algorithm for this problem. The proof was based on novel factorisations of words. By better exploiting the interplay between factorisations and compression, he improved the algorithm so that it worked in PSPACE [65].

It is worth mentioning, that the solution proposed by Plandowski is essentially different than the one given by Makanin. In particular, it allowed generalisations more easily: Diekert, Gutiérrez and Hagenah [13] showed, that Plandowski's algorithm can be extended to the case in which we allow regular constraints in the equation (i.e. we want that the word substituted for $X$ is from a regular language, whose description by a finite automaton is part of the input) and inversion; such an extended algorithm still works in polynomial space. It is easy to show that solving equations in free groups reduces to the above-mentioned problem of word equations with regular constraints and inversion [13] (it is worth mentioning, that in general we do not know whether solving equations in free groups is easier or harder than solving the ones in a free semigroup).

On the other hand, Plandowski showed, that his algorithm can be used to generate a finite representation of all solutions of a word equation [66], which allows solving several decision problems concerning the set of all solutions (finiteness, boundedness, boundedness of the exponent of periodicity etc.). It is not known, whether this algorithm can be generalised so that it generates all solutions also in the case of regular constraints and inversion (or in a free group).

The new, simpler algorithm for word equations and demonstration of connections between compression and word equations gave a new hope for solving the context unification problem. The first results were very promising: by using 'tree' equivalents of SLPs [2] computational complexity of some problems related to context unification was established [9,19,48]. Unfortunately, this approach failed to fully generalise Plandowski's algorithm for words: the equivalent of factorisations that were used in the algorithm were not found for trees.

It is worth mentioning, that Rytter and Plandowski's approach, in which we compress a solution using SLPs (or in the non-deterministic case—we guess the compressed representation of the solution) and then perform the computation directly on the SLP-compressed representations using known algorithm that work in polynomial time, turned out to be extremely fruitful in many branches of computer science. The recent survey by Lohrey gives several such successful applications [53].

### 1.3 Recompression

Recompression was developed for a specific problem concerning compressed data (fully compressed membership problem for finite automata [30]) and was later successfully applied to word equations [36] and other problems related to compressed representations. The usual approach for word equations (and compressed data in general) is that one tries to extract information about the combinatorics of the underlying words from the equation (compressed representation) and use this structure to solve the problem at hand. This is somehow natural: if the word can be represented compactly (be it as a solution of a word equation or using some compression mechanism) then it should have a lot of internal structure.

Recompression takes a different approach: our aim is to perform simple compression operations on the solution word of the word equation directly on the compressed representation. We need to modify the equation a bit in order to do that, however, the choice of the compression operation and the analysis focuses on the compressed representation and its properties and (almost) completely ignores the properties of the solution. The idea of performing the compression operation is somehow natural in view of the already mentioned Plandowski and Rytter result [67], that the (length-minimal) solution has a small SLP: since such an SLP exists, we can try to build it bottom-up, i.e. the SLP has a rule $a \to bc$ and so we will replace each $bc$ in the solution by $a$. (There are some complications in case of $b = c$, as then the compression is ambiguous: we solve this by replacing the maximal repetitions of $b$ letter instead of replacing $bb$).

Of course, performing such a compression on the equation might be difficult or even impossible at all and we sometimes need to modify the equation. However, it turns out that a greedy choice suffices to guarantee that the kept equation is of quadratic size. The correctness and size analysis turns out to be surprisingly easy. The method is also very robust, so that it can be applied to various scenarios related to word equations: one variable word equations [35], equations in free groups [14], twisted word equations [12], context unification [31], ... See the following Sections for details of some of those results.

### 1.4 Algorithms for Grammar-Based Compression

Due to ever-increasing amount of data, compression is widely applied in order to decrease the data's size. Still, the stored data is accessed and processed. Decompressing it on each such an occasion basically wastes the gain of reduced storage size. Thus there is a demand for algorithms dealing directly with the compressed data, without explicit decompression. Indeed, efficient algorithms for fundamental text operations (pattern matching, equality testing, etc.) are known for various practically used compression methods (LZ77, LZW, their variants, etc.) [20–25, 63].

Note that above the compression can be seen as a source of problem that we want to overcome. However, as demonstrated by Plandowski and Rytter [67], the compression can also be seen as a solution to some problems, i.e. if we can show that the instance or its solutions is (highly) compressible, then we can

compress it and, using the algorithms mentioned above, perform the computation on the compressed representation. See a recent survey of Lohrey [53], which gives examples of application of this approach in various fields, ranging from group theory, computational topology to program verification.

Compression standards differ in the main idea as well as in details. Thus when devising algorithms for compressed data, quite early one needs to focus on the exact compression method, to which the algorithm is applied. The most practical (and challenging) choice is one of the widely used standards, like LZW or LZ77. However, a different approach is also pursued: for some applications (and most of theory-oriented considerations) it would be useful to *model* one of the practical compression standard by a more mathematically well-founded and 'clean' method. The already mentioned Straight-Line Programs (SLPs), are such a clean formulation for many block compression methods: each LZ77 compressed text can be converted into an equivalent SLP of size $\mathcal{O}(n \log(N/n))$ and in $\mathcal{O}(n \log(N/n))$ time [5,70] (where $N$ is the size of the decompressed text), while each SLP can be converted to an equivalent LZ77-like of $\mathcal{O}(n)$ size in polynomial time. Other reasons of popularity of SLPs is that usually they compress well the input text [46,60] Lastly, a greedy grammar compression can be efficiently implemented and thus can be used as a preprocessing to other compression methods, like those based on Burrows-Wheeler transform [39].

One can treat an SLP as a system of (very simple) word equations, i.e. a production $X \to \alpha$ is rewritten as $X = \alpha$, and so the recompression algorithm generalizes also to such setting. It can be then seen as a variant of *locally consistent parsing* [1,58,72], and indeed those techniques were one of the sources of the recompression approach.

It is no surprise that the highly non-deterministic recompression algorithm determinises when applied to SLPs, what is surprising is that it can be made efficient. In particular, it can be used to checking the equality of two SLPs in roughly quadratic time, which is the fastest known algorithm for this problem [33] (and also for the generalisation of this problem, the fully compressed pattern matching).

The main drawback of grammar compression is that the size of the smallest grammar cannot be even approximated within (small enough) constant factor [5,80]. There are many algorithms that achieve a logarithmic approximation ratio [5,70,73], recompression can also be used to obtain one (in fact: two different). One of those algorithms [32] seems to have a slightly better practical behaviour than the other ones, the second has much simpler analysis than other approximation algorithms [34] (as it is essentially a greedy left-to-right scan).

Just as recompression generalizes from word equations to context unification (i.e. term equations), the approximation algorithm based on recompression for strings can be generalized to trees [38], in which case it produces a so-called tree SLP [2]. This was the first approximation algorithm for this problem.

**Survey's Limitations**

As this is an informal survey presentations, most of the proofs are only sketched or omitted. Due to space constraints, only some applications and results are explained in detail.

## 2 Recompression for Word Equations

We begin with a formal definition of the word equations problem: Consider a finite alphabet $\Sigma$ and set of variables $\mathcal{X}$; during the algorithm $\Sigma$ will be extended by new letters, but it will always remain finite. Word equation is of a form '$u = v$', where $u, v \in (\Sigma \cup \mathcal{X})^*$ and its solution is a homomorphism $S : \Sigma \cup \mathcal{X} \mapsto \Sigma^*$, which is constant on $\Sigma$, that is $S(a) = a$, and satisfies the equation, i.e. words $S(u)$ and $S(v)$ are equal. By $n$ we denote the size of the equation, i.e. $|u| + |v|$. The algorithm requires only small improvements so that it applies also to systems of equations, to streamline the presentation we will not consider this case.

Fix any solution $S$ of the equation $u = v$, without loss of generality we can assume that this is the *shortest solution*, i.e. the one minimising $|S(u)|$; let $N$ denote the *length of the solution*, that is $|S(u)|$. By the earlier work of Plandowski and Rytter [67], we know that $S(u)$ (and also $S(X)$ for each variable $X$) has an SLP (of size $\mathsf{poly}(n, \log N)$), in fact the same conclusion can be to drawn from the later works of Plandowski [64–66]. Regardless of the form of $S$ and SLP, we know, that at least one of the productions in this SLP is of the form $c \to ab$, where $c$ is a nonterminal of the SLP while $a, b \in \Sigma$ are letters. Let us 'reverse' this production, i.e. replace in $S(u)$ all pairs of letters $ab$ by $c$. It is relatively easy to formalise this operation for words, it is not so clear, what should be done in case of equations, so let us inspect the easier fragment first.

**Algorithm 1.** $\mathsf{PairComp}(ab, w)$ Compression of pair $ab$

1: let $c \in \Sigma$ be an unused letter
2: replace all occurrences of $ab$ in $w$ by $c$

Consider an explicitly given word $w$. Performing the '$ab$-pair compression' on it is easy (we replace each pair $ab$ by $c$), as long as $a \neq b$: replacing pairs $aa$ is ambiguous, as such pairs can 'overlap'. Instead, we replace *maximal blocks* of a letter $a$: block $a^\ell$ is *maximal*, when there is no letter $a$ to left nor to the right of it (in particular, there could be no letter at all).

Formally, the operations are defined as follows:

- *$ab$ pair compression* For a given word $w$ replace all occurrences of $ab$ in $w$ by a fresh letter $c$.
- *$a$ block compression* For a given word $w$ replace all occurrences of maximal blocks $a^\ell$ for $\ell > 1$ in $w$ by fresh letters $a_\ell$.

We always assume, that in the $ab$-pair compression the letters $a$ and $b$ are different.

Observe, that those operations are indeed 'inverses' of SLP productions: replacing $ab$ with $c$ corresponds to a production $c \to ab$, similarly replacing $a^\ell$ with $a_\ell$ corresponds to a production $a_\ell \to a^\ell$.

**Algorithm 2.** $\mathsf{BlockComp}(a, w)$ Block compression for $a$

1: **for** $\ell > 1$ **do**
2: let $a_\ell \in \Sigma$ be an unused letter
3: replace all maximal blocks $a^\ell$ in $w$ by $a_\ell$

Iterating the pair and blocks compression results in a compression of word $w$, assuming that we treat the introduced symbols as normal letters. There are several possible ways to implement such iteration, different results are obtained by altering the order of the compressions, exact treatment of new letters and so on. Still, essentially each 'reasonable' variant works.

Observe, that if we compress two words, say $w_1$ and $w_2$, in parallel then the resulting words $w_1'$ and $w_2'$ are equal if and only if $w_1$ and $w_2$ are. This justifies the usage of compression operations to both sides of the word equation in parallel, it remains to show, how to do that.

Let us fix a solution $S$, a pair $ab$ (where $a \neq b$); consider how does a particular occurrence of $ab$ got into $S(u)$.

**Definition 1.** *For an equation $u = v$, solution $S$ and pair $ab$ an occurrence of $ab$ in $S(u)$ (or $S(v)$) is*

- explicit*, if it consists solely of letters coming from $u$ (or $v$);*
- implicit*, if it consists solely of letters coming from a substitution $S(X)$ for a fixed occurrence of some variable $X$;*
- crossing*, otherwise.*

*A pair $ab$ is* crossing *(for a solution $S$) if it has at least one crossing occurrence and* non-crossing *(for a solution $S$) otherwise.*

*We similarly define* explicit*,* implicit *and* crossing *occurrences for blocks of letter $a$; $a$ is* crossing*, if at least one of its blocks has a crossing occurrence. (In other words: $aa$ is crossing).*

*Example 1.* Equation

$$aaXbbabababa = XaabbYabX$$

has a unique solution $S(X) = a$, $S(Y) = abab$, under which sides evaluate to

$$aaabbabababa = aaabbabababa.$$

Pair $ba$ is crossing (as the first letter of $S(Y)$ is $a$ and first $Y$ is preceded by a letter $b$, moreover, the last letter of $S(Y)$ is $b$ and the second $Y$ is succeeded by

a letter $a$), pair $ab$ is non-crossing. Letter $b$ is non-crossing, letter $a$ is crossing (as $X$ is preceded by a letter $a$ on the left-hand side of the equation and on the right-hand side of the equation $X$ is succeeded by a letter $a$).

**Algorithm 3.** PairComp($ab$, '$u = v$') Pair compression for $ab$ in an equation $u = v$

1: let $c \in \Sigma$ be a fresh letter
2: replace all occurrences of $ab$ in '$u = v$' by $c$

**Algorithm 4.** BlockComp($a$, '$u = v$') Block compression for a letter $a$ in an equation '$u = v$'

1: **for** $\ell > 1$ **do**
2: let $a_\ell \in \Sigma$ be a fresh letter
3: replace all occurrences of maximal blocks $a^\ell$ in '$u = v$' by $a_\ell$

Fix a pair $ab$ and a solution $S$ of the equation $u = v$. If $ab$ is non-crossing, performing PairComp($ab, S(u)$) is easy: we need to replace every explicit occurrence (which we do directly on the equation) as well as each implicit occurrence, which is done 'implicitly', as the solution is not stored, nor written anywhere. Due to the similarities to PairComp we will simply use the name PairComp($ab$,'$u = v$'), when we make the pair compressions on the equation. The argument above shows, that if the equation had a solution for which $ab$ is non-crossing then also the obtained equation has a solution. The same applies to the block compression, called BlockComp($a$,'$u = v$') for simplicity. On the other hand, if the obtained equation has a solution, then also the original equation had one (this solution is obtained by replacing each letter $c$ by $ab$, the argument for the block compressions the same).

**Lemma 1.** *Let the equation $u = v$ have a solution $S$, such that $ab$ is non-crossing for $S$. Then $u' = v'$ obtained by* PairComp*($ab$,'$u = v$') is satisfiable. If the obtained equation $u' = v'$ is satisfiable, then also the original equation $u = v$ is. The same applies to* BlockComp*($a$,'$u = v$').*

Unfortunately Lemma 1 is not enough to simulate Compression($w$) directly on the equation: In general there is no guarantee that the pair $ab$ (letter $a$) is non-crossing, moreover, we do not know which pairs have only implicit occurrences. It turns out, that the second problem is trivial: if we restrict ourselves to the shortest solutions then every pair that has an implicit occurrence has also a crossing or explicit one, a similar statement holds also for blocks of letters.

**Lemma 2** ([67]). *Let $S$ be a shortest solution of an equation '$u = v$'. Then:*

- *If $ab$ is a substring of $S(u)$, where $a \neq b$, then $a$, $b$ have explicit occurrences in the equation and $ab$ has an explicit or crossing occurrence.*
- *If $a^k$ is a maximal block in $S(u)$ then $a$ has an explicit occurrence in the equation and $a^k$ has an explicit or crossing occurrence.*

The proof is simple: suppose that a pair has only implicit occurrences. Then we could remove them and the obtained solution is shorter, contradicting the assumption. The argument for blocks is a bit more involved, as they can overlap.

Getting back to the crossing pairs (and blocks), if we fix a pair $ab$ (letter $a$), then it is easy to 'uncross' it: by Definition 1 we can conclude that the pair $ab$ is crossing if and only if for some variables $X$ and $Y$ (not necessarily different) one of the following conditions holds (we assume that the solution does not assign an empty word to any variable—otherwise we could simply remove such a variable from the equation):

(CP1) $aX$ occurs in the equation and $S(X)$ begins with $b$;
(CP2) $Yb$ occurs in the equation and $S(Y)$ ends with $a$;
(CP3) $YX$ occurs in the equation, $S(X)$ begins with $b$ and $b$ $S(Y)$ ends with $a$.

In each of these cases the 'uncrossing' is natural: in (1) we 'pop' from $X$ a letter $b$ to the left, in (2) we pop $a$ to the right from $Y$, in (3) we perform both operations. It turns out that in fact we can be even more systematic: we do not have to look at the occurrences of variables, it is enough to consider the first and last letter of $S(X)$ for each variable $X$:

- If $S(X)$ begins with $b$ then we replace $X$ with $bX$ (changing implicitly the solution $S(X) = bw$ to $S'(X) = w$), if in the new solution $S(X) = \epsilon$, i.e. it is empty, then we remove $X$ from the equation;
- if $S(X)$ ends with $a$ then we apply a symmetric procedure.

Such an algorithm is called Pop.

**Algorithm 5.** Pop$(a, b, '$$u = v$$')$

```
1: for X: variable do
2:     if the first letter of S(X) is b then                          ▷ Guess
3:         replace every X w 'u = v' by bX
                          ▷ Implicitly change solution S(X) = bw to S(X) = w
4:         if S(X) = ϵ then                                           ▷ Guess
5:             remove X from u and v
6:     ...                ▷ Perform a symmetric operation for the last letter and a
```

It is easy to see, that for appropriate non-deterministic choices the obtained equation has a solution for which $ab$ is non-crossing: for instance, if $aX$ occurs in the equation and $S(X)$ begins with $b$ then we make the corresponding non-deterministic choices, popping $b$ to the left and obtaining $abX$; a simple proof requires a precise statement of the claim as well as some case analysis.

**Lemma 3.** *If the equation '$u = v$' has a solution $S$ then for an appropriate run of* Pop$(a, b, `u = v')$ *(for appropriate non-deterministic choices) the obtained equation $u' = v'$ has a corresponding solution $S'$, i.e. $S(u) = S'(u')$, for which $ab$ is a non-crossing pair. If the obtained equation has a solution then also the original equation had one.*

Thus, we know how to proceed with a crossing $ab$-pair compression: we first turn $ab$ into a non-crossing pair (Pop) and then compress it as a non-crossing pair (PairComp).

We would like to perform similar operations for block compression. For non-crossing blocks we can naturally define a similar algorithm BlockComp$(a, `u = v')$. It remains to show how to 'uncross' a letter $a$. Unfortunately, if $aX$ occurs in the equation and $S(X)$ begins with $a$ then replacing $X$ with $aX$ is not enough, as $S(X)$ may still begin with $a$. In such a case we iterate the procedure until the first letter of $X$ is not $a$ (this includes the case in which we remove the whole variable $X$). Observe, that instead of doing this letter by letter, we can uncross $a$ in one step: it is enough to remove from variable $X$ its whole $a$-prefix and $a$-suffix of $S(X)$ (if $w = a^\ell w' a^r$, where $w'$ does not begin nor end with $a$, $a$-prefix $w$ is $a^\ell$ and $a$-suffix is $a^r$; if $w = a^\ell$ then $a$-suffix and $w'$ are empty). Such an algorithm is called CutPrefSuff.

**Algorithm 6.** CutPrefSuff$(a, `u = v')$ Popping prefixes and suffixes

1: **for** $X$: variable **do**
2: guess the lengths $\ell$, $r$ of $a$-prefix and suffix of $S(X)$ ▷ $S(X) = a^\ell w a^r$
▷ If $S(X) = a^\ell$ then $r = 0$
3: replace occurrences of $X$ in $u$ and $v$ by $a^\ell X a^r$
▷ $a^\ell$, $a^r$ are stored in a compressed way
4: ▷ Implicitly change the solution $S(X) = a^\ell w b^r$ to $S(X) = w$
5: **if** $S(X) = \epsilon$ **then** ▷ Guess
6: remove $X$ from $u$ and $v$

Similarly as in Pop, we can show that after an appropriate run of CutPrefSuff the obtained equation has a (corresponding) solution for which $a$ is non-crossing. Unfortunately, there is another problem: we need to write down the lengths $\ell$ and $r$ of $a$-prefixes and suffixes. We can write them as binary numbers, in which case they use $\mathcal{O}(\log \ell + \log r)$ bits of memory. However in general those still can be arbitrarily large numbers. Fortunately, we can show that in *some* solution those values are at most exponential (and so their description is polynomial-size). This easily follows from the exponential bound on exponent of periodicity [43]. For the moment it is enough that we know that:

**Lemma 4 ([43]).** *In the shortest solution of the equation '$u = v$' each $a$-prefix and $a$-suffix has at most exponential length (in terms of $|u| + |v|$).*

Thus in Pop we can restrict ourselves to $a$-prefixes and suffixes of at most exponential length.

**Lemma 5.** *Let $S$ be a shortest solution of '$u = v$'. For some non-deterministic choices, i.e. after some run of* CutPrefSuff$(a,$'$u = v$'$)$*, the obtained equation '$u' = v'$' has a corresponding solution $S'$, such that $S'(u') = S(u)$, and $a$ is a non-crossing letter for $S'$, moreover, the explicit $a$ blocks in '$u' = v'$' have at most exponential length. If the obtained equation has a solution then also the original equation had one.*

After Pop we can compress $a$-blocks using BlockComp$(a,$'$u = v$'$)$, observe that afterwards long $a$-blocks are replaced with single letters.

We are now ready to simulate Compression directly on the equation. The question is, in which order we should compress pairs and blocks? We make the choice nondeterministically: if there are any non-crossing pairs or letters, we compress them. This is natural, as such compression decreases both the size of the equation and the size of the length-minimal solution of the equation. If all pairs and letters are crossing, we choose greedily, i.e. the one that leads to the smallest equation (in one step). It is easy to show that such a strategy keeps the equation quadratic, more involved strategy, in which we compress many pairs/blocks in parallel, leads to a linear-length equation.

**Algorithm 7.** WordEqSAT Deciding the satisfiability of word equations

```
1: while |u| > 1 or |u| > 1 do
2:     L ← list of letters in u, v
3:     Choose a pair ab ∈ P^2 or a letter a ∈ P                ▷ Guess
4:     if it is crossing then                                  ▷ Guess
5:         uncross it
6:     compress it
7: Solve the problem naively
```

Call one iteration of the main loop a *phase*.

The correctness of the algorithm follows from the earlier discussion on the correctness of BlockComp, CutPrefSuff, PairComp and Pop. In particular, the length of the length-minimal solution drops by at least 1 in each iteration, thus the algorithm terminates.

**Lemma 6.** *Algorithm* WordEqSAT *has $\mathcal{O}(N)$ phases, where $N$ is the length of the shortest solution of the input equation.*

Let us bound the space needed by the algorithm: we claim that for appropriate nondeterministic choices the stored equation has at most $8n^2$ letters (and $n$ variables). To see this, observe first that each Pop introduces at most $2n$ letters, one at each side of the variable. The same applies to CutPrefSuff (formally, CutPrefSuff introduces long blocks but they are immediately replaced with single letters, and so we can think that in fact we introduce only $2n$ letters). By (1)–(3) we know that there are at most $2n$ crossing pairs and crossing letters (as each crossing pair/each crossing letter corresponds to one occurrence of a variable

and one 'side' of such an occurrence). If the equation has $m$ letters (and at most $n$ occurrences of variables) and there is an occurrence of a non-crossing pair or block then we choose it for compression. Otherwise, there are $m$ letters in the equation and each is covered by at least one pair/block, so for one of $2n$ choice at least $\frac{m}{2n}$ letters are covered, so at least $\frac{m}{4n}$ letters are removed by some compression. Thus the new equation has at most

$$\begin{aligned} \underbrace{m}_{\text{previous}} - \underbrace{\frac{m}{4n}}_{\text{removed}} + \underbrace{2n}_{\text{popped}} &= m\left(1 - \frac{1}{4n}\right) + 2n \\ &\leq 8n^2\left(1 - \frac{1}{4n}\right) + 2n \\ &= 8n^2 - 2n + 2n = 8n^2 \end{aligned}$$

letters, where the inequality follows by the inductive assumption that $m \leq 8n^2$. Going for the bit-size, each symbol requires at most logarithmic number of bits, and so

**Lemma 7.** WordEqSAT *runs in* $\mathcal{O}(n^2 \log n)$ *(bit) space.*

With some effort we can make the above if analysis much tighter, see Sect. 4.1.

**Theorem 1 ([36]).** *The recompression based algorithm (nondeterministically) decides word equations problem in* $\mathcal{O}(n \log n)$ *bit-space; moreover, the stored equation has linear length.*

Moreover, with some extra effort one can remove also the logarithmic dependency, and show that satisfiability of word equations is in non-deterministic linear space, i.e. the problem is context sensitive. Surprisingly, it is enough to employ Huffman coding for the equation and run a variant of the algorithm. However, the analysis requires a deeper understanding of how fragments of the equation are changed during the algorithm and how they depend one on another.

**Theorem 2 ([37]).** *A variant of recompression based algorithm which encodes the equation using Huffman coding (nondeterministically) decides word equations problem in* $\mathcal{O}(m)$ *bit-space; where* $m$ *is the bit-size encoding of the input using any prefix-free code.*

Note that we allow some bit-optimization in the size of the input problem.

As a reminder: a PSPACE algorithm for this problem was already known [65]. Its memory consumption is not stated explicitly in that work, however, it is much larger than $\mathcal{O}(n \log n)$: the stored equations are of length $\mathcal{O}(n^3)$ and during the transformations the algorithm uses essentially more memory.

## 3 Extensions of the Algorithm for Word Equations

### 3.1 $\mathcal{O}(n \log n)$ Space

In order to improve the space consumption from quadratic to $\mathcal{O}(n \log n)$ we want to perform several compressions in parallel. To make it more precise, observe that

- All block compressions (also for different letters) can be performed in parallel, as such blocks do not overlap. Moreover, uncrossing different letters can also be done in parallel: if $a$ is the first letter of $S(X)$ and $b$ the last, then we pop from $X$ the $a$-prefix and $b$-suffix.
- If $\Sigma_\ell$ and $\Sigma_r$ are disjoint, then the pair compressions for $ab$ with $a \in \Sigma_\ell$ and $b \in \Sigma_r$ can be done in parallel. Similarly as in the previous case, uncrossing can be done in parallel, by popping first letter if it is from $\Sigma_r$ and last if it is from $\Sigma_\ell$.
- We do not compress all pairs, only those from $\mathcal{O}(1)$ partitions $\Sigma_\ell$, $\Sigma_r$ that cover 'many' occurrences of pairs in the equation and in the solution.

The crucial things is the choice of partitions. It turns out that choosing a random partition reduces the length of the solution by a constant fraction: consider two consecutive letters $ab$ in $S(X)$. If $a = b$ then they will be compressed as part of the maximal block. If $a \neq b$ then there is $1/4$ chance that $ab \in \Sigma_\ell\Sigma_r$. Thus, in expectation, the length of the word shortens by one fourth of its length.

A similar argument also shows that the number of letters in the equation remains linear, when a random partition is chosen. Thus, the equation will be of linear size (though each letter may need $\mathcal{O}(\log n)$ bits for the encoding).

### 3.2 Equations with Regular Constraints and Inversion; Equations in Free Groups

As already mentioned, it is natural and important to extend the word equations by regular constraints and inversion, in particular this leads to an algorithm for equations in free groups [13] (the reduction between those two problems is fully syntactical and does not depend on the particular algorithm for solving word equations). Note that it is not known, whether the algorithm generating a representation of all solutions can be also extended by regular constraints and inversion. Thus the only previously known algorithm for representation of all solutions of an equation in a free group was due to Razborov [68], and it was based on Makanin's algorithm for word equations in free groups.

Adding the regular constraints to the recompression based algorithm WordEqSAT is fairly standard: We can encode all constraints using one nondeterministic finite automaton (the constraints for particular variables differ only in the set of accepting states). For each letter $c$ we store its *transition function*, i.e. a function $f_c : Q \to 2^Q$, which says that the automaton in state $q$ after reading a letter $c$ reaches a state in $f_c(q)$. This function naturally extends to words: it still defines which states can be reached from $q$ after reading $w$. Formally $f_{wa} = (f_w \circ f_a)(q) = \{p \mid \exists q' \in f_w(q) \text{ i } p \in f_a(q')\}$ for a letter $a$. If we introduce a new letter $c$ (which replaces a word $w$) then we naturally define the transition function $f_c \leftarrow f_w$. We can express the regular constraints in terms of this function: saying that $S(X)$ is accepted by an automaton means that $f_{S(X)}(q_0)$ is one of the accepting states. So it is enough to guess the value of $f_{S(X)}$ which satisfies this condition; in this way we can talk about the value $f_X$ for a variable $X$. Popping letters from a variable means that we need to adjust

the transition function, i.e. when we replace $X$ by $aX$ then $f_X = f_a \circ f_{X'}$, we similarly define $f_X$ when we pop letters to the right.

More problems are caused by the *inversion*: intuitively it corresponds to taking the inverse element in the group and on the semigroup level we this is simulated by requiring that $\overline{\overline{a}} = a$ for each letter $a$ and $\overline{a_1 a_2 \dots a_m} = \overline{a_m} \dots \overline{a_2 a_1}$. This has an impact on the compression: when we compress a pair $ab$ to $c$, then we should also replace $\overline{ab} = \overline{b}\overline{a}$ by a letter $\overline{c}$. At the first sight this looks easy, but becomes problematic, when those two pairs are not disjoint, i.e. when $\overline{a} = a$ (or $\overline{b} = b$); in general we cannot exclude such a case and if it happens, in a sequence $ba\overline{b}$ during the pair compression for $ba$ we want to simultaneously replace $ba$ and $a\overline{b}$, which is not possible. Instead, we replace maximal fragments that can be fully covered with pairs $ab$ or $\overline{b}\overline{a}$, in this case this: the whole triple $ba\overline{b}$. In the worst case (when $a = \overline{a}$ and $b = \overline{b}$) we need to replace whole sequences of the form $(ab)^n$, which is a common generalisation of both pairs and blocks compression.

**Theorem 3** ([6,14]). *A recompression based algorithm generates in polynomial space the description of all solutions of a word equation in free semigroups with inversion and regular constraints.*

### 3.3 Context Unification

Recall that the context unification is a generalisation of word equations to the case of terms (Fig. 2). What type of equations we would like to consider? Clearly we consider terms over a fixed signature (which is usually part of the input), and allow occurrences of constants and variables. If we allow only that the variables represent full terms, then the satisfiability of such equations is decidable in polynomial time [69] and so probably does not generalise the word equations (which are NP-hard). This is also easy to observe when we look closer at a word equation: the words represented by the variables can be concatenated at both ends, i.e. they represent terms with a missing argument.

We arrive at a conclusion that our generalisation should use variables *with arguments*, i.e. the (second-order) variables take an argument that is a full term and can use it, perhaps several times. Such a definition leads to a *second-order unification*, which is known to be undecidable even in very restricted subcases [16, 26,47,49].

Thus we would like to have a subclass of second order unification that still generalises word equations. In order to do that we put additional restriction on the solutions: each argument can be used by the term *exactly once*. Observe that this still generalises the word equations: using the argument exactly once naturally corresponds to concatenation (Fig. 1).

Formally, in the context unification problem [7,8,74], we consider an equation $u = v$ in which we use term variables (representing closed terms), which we denote by letters $x, y$, as well as context variables (representing terms with one 'hole' for the argument, they are usually called *contexts*), which we denote by letters $X, Y$. Syntactically, $u$ and $v$ are terms that use letters from signature $\Sigma$

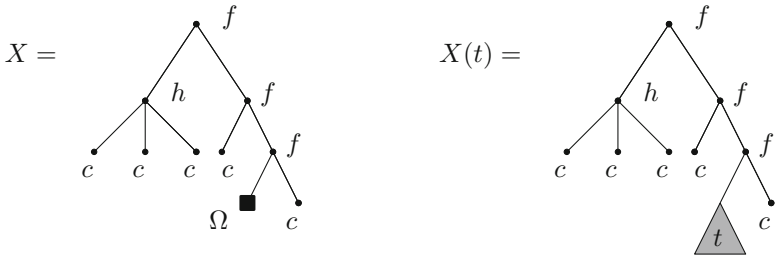


**Fig. 1.** A context and the same context applied on an argument.

(which is part of the input), term variables and context variables, the former are treated as symbols of arity 0, while the latter as symbols of arity 1. A *substitution* $S$ assigns to each variable a closed term over $\Sigma$ and to each context variable it assigns a *context*, i.e. a term over $\Sigma \cup \{\Omega\}$ in which the special symbol $\Omega$ has arity 0 and is used exactly once. (Intuitively it corresponds to a place in which we later substitute the argument). $S$ is extended to $u$, $v$ in a natural way, note that for a context variable $X$ the term $S(X(t))$ is obtained by replacing in $S(X)$ the unique symbol $\Omega$ by $S(t)$. A *solution* is a substitution satisfying $S(u) = S(v)$.

*Example 2.* Consider a signature $\{f, c, c'\}$, where $f$ has arity 2 while $c, c'$ have arity 0 and consider an equation $X(c) = Y(c')$, where $X$ and $Y$ are context variables. The equation has a solution $S(X) = f(\Omega, c'), S(Y) = f(c, \Omega)$ and then $S(X(c)) = f(c, c') = S(Y(c'))$.

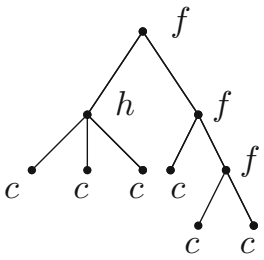


**Fig. 2.** Term $f(h(c, c, c), f(c, f(c, c)))$ viewed as a tree, $f$ is of arity 2, $h$: 3 and $c$: 0.

We try to apply the main idea of the recompression also in the case of terms: we iterate local compression operations and we guarantee that the word (term) equation is polynomial size. Since several term problems were solved using compression-based methods [9,17–19,48], there is a reasonable hope that our approach may succeed.

Pair and block compression easily generalise to sequences of letters of arity 1 (we can think of them as words), unfortunately, there is no guarantee that a term has even one such letter. Intuitively, we rather expect that it has mostly leaves and symbols of larger arity. This leads us to another local compression operation: *leaf compression*. Consider a node labelled with $f$ and its $i$-th child that is a leaf. We want to compress $f$ with this child, leaving other children (and their subtrees) unchanged. Formally, given $f$ of arity at least 1, position $1 \leq i \leq \mathrm{ar}(f)$ and a letter $c$ of arity 0 the $\mathsf{LeafComp}(f, i, c, t)$ operation (*leaf compression*) replaces in term $t$ nodes labelled with $f$ and subterms $t_1, \ldots, t_{i-1}, c, t_{i+1}, \ldots, t_{\mathrm{ar}(f)}$ (where $c$ and position $i$ are fixed, while other terms $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_{\mathrm{ar}(f)}$—varying) by a term labelled with $f'$ and subterms $t'_1, \ldots, t'_{i-1}, t'_{i+1}, \ldots, t'_{\mathrm{ar}(f)}$ that are obtained by applying recursively $\mathsf{LeafComp}$ to terms $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_{\mathrm{ar}(f)}$; in other words, we first change the label from $f$ to $f'$ and then remove the $i$-th child, which has a label $c$ and we apply such a compression to all occurrences of $f$ and $c$ in parallel.

The notion of crossing pair generalizes to this case in a natural way and the uncrossing replaces a term variable with a constant or replaces $X(t)$ with $X(f(x_1, \ldots, x_i, t, x_{i+1}, \ldots, x_\ell))$. Note that this introduces new variables.

Now the whole algorithm looks similar as in the case of word equations, we simply use additional compression operation. However, the analysis is much more involved, as the new uncrossing introduces fresh term variables. However, their number at any point can be linearly bounded and the polynomial upper-bound follows.

**Theorem 4 ([31]).** *Recompression based algorithm solves context unification in nondeterministic polynomial space.*

## 4 Recompression and Compressed Data

The recompression technique is (partially) inspired by methods coming from the algorithm's design [1,58]. In this section we show that it is able to contribute back to algorithmics: some algorithmic questions for compressed data can be solved using a recompression technique. The obtained solutions are as good and sometimes better than the known ones, which is surprising taking into the account the robustness of the method.

### 4.1 Straight Line Programs and Recompression

Recall that the *Straight Line Programme* (*SLP*) was defined as a context-free grammar whose each nonterminal generates exactly one word. We employ the following naming conventions for SLPs: its nonterminals are ordered (without loss of generality: $X_1, X_2, \ldots, X_m$), each nonterminal has exactly one production and if $X_j$ occurs in the production for $X_i$ then $j < i$; we will use symbols $\mathcal{A}$, $\mathcal{B}$, etc. to denote an SLP. The unique word generated by a nonterminal $X_i$ is denoted by $\mathrm{val}(X_i)$, while the whole SLP $\mathcal{A}$ defines a word $\mathrm{val}(\mathcal{A}) = \mathrm{val}(X_m)$.

We can treat SLP as a system of word equations (in variables $X_1, \ldots, X_m$): production $X_i \to \alpha_i$ corresponds to an equation $X_i = \alpha_i$; observe that such an equality is meaningful as $\mathrm{val}(X) = \mathrm{val}(\alpha)$ (where val is naturally extended to strings of letters and nonterminals), moreover, this is the unique solution of this equation. Thus the recompression technique can be applied to SLPs as well (so far we used recompression only to one equation but it easily generalises also to a system of equations).

However, there are two issues that need to be solved: non-determinism and efficiency: the recompression for word equations is highly non-deterministic while algorithms for SLPs should, if possible, be deterministic and we usually want them to be efficient, i.e. we want as small polynomial degree as possible.

Let us inspect the source of non-determinism of recompression-based approach, it is needed to:

1. establish, whether $\mathrm{val}(X_i) = \epsilon$;
2. establish the first (and last) letter of $\mathrm{val}(X_i)$;
3. establish the length of $a$-prefix and suffix of $\mathrm{val}(X_i)$;
4. the choice of the partition to compress.

The first three question ask about some basic properties of the solution and can be easily answered in case of SLPs: assuming that we already know the answers for $X_j$ for $j < i$: let $X_i \to \alpha_i$, then we first remove from $\alpha_i$ all nonterminals $X_j$, for which $\mathrm{val}(X_j) = \epsilon$, and then

1. $\mathrm{val}(X_i) = \epsilon$ if and only if $\alpha_i = \epsilon$;
2. the first letter of $\mathrm{val}(X_i)$ is the first letter of $\alpha_i$ or the first letter of $\mathrm{val}(X_j)$, if the first symbol of $\alpha_i$ is $X_j$;
3. the length of the $a$-prefix depends only on the letters $a$ in $\alpha_i$ and the lengths of $a$-prefixes in nonterminals in $\alpha_i$.

All those conditions can be verified in linear time. The last question is of different nature. However, the argument used to show that a good choice of a partition exists actually shown that in expectation the choice is a good one and this approach can be easily derandomised using conditional expectation approach. In particular, this subprocedure can be implemented in linear time.

Concerning the running time, the generalisations of Pop, PairComp, CutPrefSuff and BlockComp can be implemented in linear time, thus the recompression for SLPs runs in polynomial (in SLP's size) time, so polynomial in total.

**Lemma 8.** *The recompression for SLPs runs in $\mathcal{O}(n \log N) \leq \mathcal{O}(n^2)$ time, where $n$ is the size of the input SLP and $N$ is the length of the defined word.*

### 4.2 SLP Equality and Fully Compressed Pattern Matching

One of the first (and most important) problems considered for SLPs is the equality testing, i.e. for two SLPs we want to decide if they define the same word. The first polynomial algorithm for this problem was given in 1994 by Plandowski [63],

to be more precise, his algorithm run in $\mathcal{O}(n^4)$ time. Afterwards research was mostly focused on the more general problem of *fully compressed pattern matching*: for given SLPs $\mathcal{A}$ and $\mathcal{B}$ we want to decide, whether $\mathrm{val}(\mathcal{A})$ occurs in $\mathrm{val}(\mathcal{B})$ (as a subword). The first solution to this problem was given by Karpiński et al. [40] in 1995. Gasieniec et al. [21] gave a faster randomised algorithm. In 1997 Miyazaki et al. [59] constructed an $\mathcal{O}(n^4)$ algorithm. Finally, Lifshits gave an $\mathcal{O}(n^3)$ algorithm for this problem [52]. All of the mentioned papers were based on the same original idea as Plandowski's algorithm.

Recompression can be naturally applied to equality testing of SLPs: given two SLPs $\mathcal{A}$ and $\mathcal{B}$ we add an equation $X_{m_\mathcal{A}} = Y_{m_\mathcal{A}}$ and ask about the satisfiability of the whole system. As already observed, the recompression based algorithm will work in polynomial time. It turns out that the proper implementation (using many nontrivial algorithmic techniques) runs in time $\mathcal{O}(n \log N)$, where $N = |\mathrm{val}(\mathcal{A})| = |\mathrm{val}(\mathcal{B})|$ (if $|\mathrm{val}(\mathcal{A})| \neq |\mathrm{val}(\mathcal{B})|$ then clearly $\mathcal{A}$ and $\mathcal{B}$ are not equal) and $n$ the sum of sizes of SLPs $\mathcal{A}$ and $\mathcal{B}$. In order to obtain such a running time, we need several optimisations.

**Theorem 5 ([33]).** *The recompression based algorithm for equality testing for SLPs runs in $\mathcal{O}(n \log N)$ time, where $n$ is the sum of SLPs' sizes while $N$ the size of the defined (decompressed) words.*

In order to use the recompression technique for the fully compressed pattern matching problem, we need some essential modifications: consider *ba*-pair compression on a pattern *ab* and text *bab*. We obtain the same pattern *ab* and text *cb*, loosing the only occurrence of the pattern in the text. This happens because the compression (on the text) is done partially on the pattern occurrence and partially outside it. To remedy this, we perform the compression operations in a particular order, which takes into the account what are the first and last letters of pattern and text. (In the considered example, we make the *ab*-pair compression first and this preserves the occurrences of the pattern.) Similar approach works also for block compression.

**Theorem 6 ([33]).** *The recompression based algorithm for fully compressed pattern matching runs in $\mathcal{O}(n \log M)$ time, where $n$ is the sum of SLPs' sizes while $M$ the length of the (uncompressed) pattern.*

## References

1. Alstrup, S., Brodal, G.S., Rauhe, T.: Pattern matching in dynamic texts. In: Shmoys, D.B. (ed.) SODA, pp. 819–828. ACM/SIAM (2000). https://doi.org/10.1145/338219.338645, http://dl.acm.org/citation.cfm?id=338219.338645
2. Busatto, G., Lohrey, M., Maneth, S.: Efficient memory representation of XML document trees. Inf. Syst. **33**(4–5), 456–474 (2008)
3. Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. Math. Log. Q. **34**(4), 337–342 (1988). https://doi.org/10.1002/malq.19880340410

4. Charatonik, W., Pacholski, L.: Word equations with two variables. In: IWWERT, pp. 43–56 (1991). https://doi.org/10.1007/3-540-56730-5_30
5. Charikar, M., et al.: The smallest grammar problem. IEEE Trans. Inf. Theory **51**(7), 2554–2576 (2005). https://doi.org/10.1109/TIT.2005.850116
6. Ciobanu, L., Diekert, V., Elder, M.: Solution sets for equations over free groups are EDT0L languages. IJAC **26**(5), 843–886 (2016). https://doi.org/10.1142/S0218196716500363
7. Comon, H.: Completion of rewrite systems with membership constraints. Part I: deduction rules. J. Symb. Comput. **25**(4), 397–419 (1998). https://doi.org/10.1006/jsco.1997.0185
8. Comon, H.: Completion of rewrite systems with membership constraints. Part II: constraint solving. J. Symb. Comput. **25**(4), 421–453 (1998). https://doi.org/10.1006/jsco.1997.0186
9. Creus, C., Gascón, A., Godoy, G.: One-context unification with STG-compressed terms is in NP. In: Tiwari, A. (ed.) 23rd International Conference on Rewriting Techniques and Applications (RTA 2012). LIPIcs, vol. 15, pp. 149–164. Schloss Dagstuhl – Leibniz Zentrum fuer Informatik, Dagstuhl, Germany (2012). https://doi.org/10.4230/LIPIcs.RTA.2012.149, http://drops.dagstuhl.de/opus/volltexte/2012/3490
10. Dąbrowski, R., Plandowski, W.: Solving two-variable word equations. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 408–419. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27836-8_36
11. Dąbrowski, R., Plandowski, W.: On word equations in one variable. Algorithmica **60**(4), 819–828 (2011). https://doi.org/10.1007/s00453-009-9375-3
12. Diekert, V., Elder, M.: Solutions of twisted word equations, EDT0L languages, and context-free groups. In: Chatzigiannakis, I., Indyk, P., Kuhn, F., Muscholl, A. (eds.) ICALP. LIPIcs, vol. 80, pp. 96:1–96:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.ICALP.2017.96
13. Diekert, V., Gutiérrez, C., Hagenah, C.: The existential theory of equations with rational constraints in free groups is PSPACE-complete. Inf. Comput. **202**(2), 105–140 (2005). http://dx.doi.org/10.1016/j.ic.2005.04.002
14. Diekert, V., Jeż, A., Plandowski, W.: Finding all solutions of equations in free groups and monoids with involution. Inf. Comput. **251**, 263–286 (2016). https://doi.org/10.1016/j.ic.2016.09.009
15. Diekert, V., Muscholl, A.: Solvability of equations in free partially commutative groups is decidable. Int. J. Algebr. Comput. **16**, 1047–1070 (2006). https://doi.org/10.1142/S0218196706003372. Conference version in Proceedings of ICALP 2001, pp. 543-554, LNCS 2076
16. Farmer, W.M.: Simple second-order languages for which unification is undecidable. Theor. Comput. Sci. **87**(1), 25–41 (1991). https://doi.org/10.1016/S0304-3975(06)80003-4
17. Gascón, A., Godoy, G., Schmidt-Schauß, M.: Context matching for compressed terms. In: Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24–27 June 2008, Pittsburgh, PA, USA, pp. 93–102. IEEE Computer Society (2008). https://doi.org/10.1109/LICS.2008.17
18. Gascón, A., Godoy, G., Schmidt-Schauß, M.: Unification and matching on compressed terms. ACM Trans. Comput. Log. **12**(4), 26 (2011). https://doi.org/10.1145/1970398.1970402

19. Gascón, A., Godoy, G., Schmidt-Schauß, M., Tiwari, A.: Context unification with one context variable. J. Symb. Comput. **45**(2), 173–193 (2010). https://doi.org/10.1016/j.jsc.2008.10.005
20. Gasieniec, L., Karpiński, M., Plandowski, W., Rytter, W.: Efficient algorithms for Lempel-Ziv encoding. In: SWAT, pp. 392–403 (1996). https://doi.org/10.1007/3-540-61422-2_148
21. Gasieniec, L., Karpiński, M., Plandowski, W., Rytter, W.: Randomized efficient algorithms for compressed strings: the finger-print approach. In: CPM, pp. 39–49 (1996). https://doi.org/10.1007/3-540-61258-0_3
22. Gasieniec, L., Rytter, W.: Almost optimal fully LZW-compressed pattern matching. In: DCC, pp. 316–325. IEEE Computer Society (1999)
23. Gawrychowski, P.: Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 421–432. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23719-5_36
24. Gawrychowski, P.: Tying up the loose ends in fully LZW-compressed pattern matching. In: Dürr, C., Wilke, T. (eds.) STACS. LIPIcs, vol. 14, pp. 624–635. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2012). https://doi.org/10.4230/LIPIcs.STACS.2012.624
25. Gawrychowski, P.: Optimal pattern matching in LZW compressed strings. ACM Trans. Algorithms **9**(3), 25 (2013). https://doi.org/10.1145/2483699.2483705
26. Goldfarb, W.D.: The undecidability of the second-order unification problem. Theor. Comput. Sci. **13**, 225–230 (1981). https://doi.org/10.1016/0304-3975(81)90040-2
27. Gutiérrez, C.: Satisfiability of word equations with constants is in exponential space. In: FOCS, pp. 112–119 (1998). https://doi.org/10.1109/SFCS.1998.743434
28. Ilie, L., Plandowski, W.: Two-variable word equations. ITA **34**(6), 467–501 (2000). https://doi.org/10.1051/ita:2000126
29. Jaffar, J.: Minimal and complete word unification. J. ACM **37**(1), 47–85 (1990)
30. Jeż, A.: The complexity of compressed membership problems for finite automata. Theory Comput. Syst. **55**, 685–718 (2014). https://doi.org/10.1007/s00224-013-9443-6
31. Jeż, A.: Context unification is in PSPACE. In: Koutsoupias, E., Esparza, J., Fraigniaud, P. (eds.) ICALP. LNCS, vol. 8573, pp. 244–255. Springer (2014). https://doi.org/10.1007/978-3-662-43951-7_21, full version at http://arxiv.org/abs/1310.4367
32. Jeż, A.: Approximation of grammar-based compression via recompression. Theor. Comput. Sci. **592**, 115–134 (2015). https://doi.org/10.1016/j.tcs.2015.05.027
33. Jeż, A.: Faster fully compressed pattern matching by recompression. ACM Trans. Algorithms **11**(3), 20:1–20:43 (2015). https://doi.org/10.1145/2631920
34. Jeż, A.: A really simple approximation of smallest grammar. Theor. Comput. Sci. **616**, 141–150 (2016). https://doi.org/10.1016/j.tcs.2015.12.032
35. Jeż, A.: One-variable word equations in linear time. Algorithmica **74**, 1–48 (2016). https://doi.org/10.1007/s00453-014-9931-3
36. Jeż, A.: Recompression: a simple and powerful technique for word equations. J. ACM **63**(1), 4:1–4:51 (2016). https://doi.org/10.1145/2743014
37. Jeż, A.: Word equations in nondeterministic linear space. In: Chatzigiannakis, I., Indyk, P., Kuhn, F., Muscholl, A. (eds.) ICALP. LIPIcs, vol. 80, pp. 95:1–95:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.ICALP.2017.95
38. Jeż, A., Lohrey, M.: Approximation of smallest linear tree grammar. Inf. Comput. **251**, 215–251 (2016). https://doi.org/10.1016/j.ic.2016.09.007

39. Kärkkäinen, J., Mikkola, P., Kempa, D.: Grammar precompression speeds up Burrows–Wheeler compression. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 330–335. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34109-0_34
40. Karpinski, M., Rytter, W., Shinohara, A.: Pattern-matching for strings with short descriptions. In: Galil, Z., Ukkonen, E. (eds.) CPM 1995. LNCS, vol. 937, pp. 205–214. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60044-2_44
41. Kharlampovich, O., Myasnikov, A.: Irreducible affine varieties over a free group. II: systems in triangular quasi-quadratic form and description of residually free groups. J. Algebra **200**, 517–570 (1998)
42. Kharlampovich, O., Myasnikov, A.: Elementary theory of free non-abelian groups. J. Algebra **302**, 451–552 (2006)
43. Kościelski, A., Pacholski, L.: Complexity of Makanin's algorithm. J. ACM **43**(4), 670–684 (1996). https://doi.org/10.1145/234533.234543
44. Kościelski, A., Pacholski, L.: Makanin's algorithm is not primitive recursive. Theor. Comput. Sci. **191**(1–2), 145–156 (1998). https://doi.org/10.1016/S0304-3975(96)00321-0
45. Laine, M., Plandowski, W.: Word equations with one unknown. Int. J. Found. Comput. Sci. **22**(2), 345–375 (2011). https://doi.org/10.1142/S0129054111008088
46. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. In: Data Compression Conference, pp. 296–305 (1999). https://doi.org/10.1109/DCC.1999.755679
47. Levy, J.: Linear second-order unification. In: Ganzinger, H. (ed.) RTA 1996. LNCS, vol. 1103, pp. 332–346. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61464-8_63
48. Levy, J., Schmidt-Schauß, M., Villaret, M.: On the complexity of bounded second-order unification and stratified context unification. Log. J. IGPL **19**(6), 763–789 (2011). https://doi.org/10.1093/jigpal/jzq010
49. Levy, J., Veanes, M.: On the undecidability of second-order unification. Inf. Comput. **159**(1–2), 125–150 (2000). https://doi.org/10.1006/inco.2000.2877
50. Levy, J., Villaret, M.: Linear second-order unification and context unification with tree-regular constraints. In: Bachmair, L. (ed.) RTA 2000. LNCS, vol. 1833, pp. 156–171. Springer, Heidelberg (2000). https://doi.org/10.1007/10721975_11
51. Levy, J., Villaret, M.: Currying second-order unification problems. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, pp. 326–339. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45610-4_23
52. Lifshits, Y.: Processing compressed texts: a tractability border. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 228–240. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73437-6_24
53. Lohrey, M.: Algorithmics on SLP-compressed strings: a survey. Groups Complex. Cryptol. **4**(2), 241–299 (2012)
54. Makanin, G.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik **2**(103), 147–236 (1977). (in Russian)
55. Makanin, G.: Equations in a free group. Izv. Akad. Nauk SSR Ser. Math. **46**, 1199–1273 (1983). English translation in Math. USSR Izv. 21 (1983)
56. Makanin, G.: Decidability of the universal and positive theories of a free group. Izv. Akad. Nauk SSSR Ser. Mat. 48, 735–749 (1984). in Russian. English translation. In: Math. USSR Izvestija **25**(75–88) (1985)
57. Matiyasevich, Y.: Some decision problems for traces. In: Adian, S., Nerode, A. (eds.) LFCS 1997. LNCS, vol. 1234, pp. 248–257. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63045-7_25

58. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica **17**(2), 183–198 (1997). https://doi.org/10.1007/BF02522825
59. Miyazaki, M., Shinohara, A., Takeda, M.: An improved pattern matching algorithm for strings in terms of straight-line programs. In: Apostolico, A., Hein, J. (eds.) CPM 1997. LNCS, vol. 1264, pp. 1–11. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63220-4_45
60. Nevill-Manning, C.G., Witten, I.H.: Identifying hierarchical structure in sequences: a linear-time algorithm. J. Artif. Intell. Res. (JAIR) **7**, 67–82 (1997). https://doi.org/10.1613/jair.374
61. Niehren, J., Pinkal, M., Ruhrberg, P.: On equality up-to constraints over finite trees, context unification, and one-step rewriting. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 34–48. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63104-6_4
62. Niehren, J., Pinkal, M., Ruhrberg, P.: A uniform approach to under specification and parallelism. In: Cohen, P.R., Wahlster, W. (eds.) ACL, pp. 410–417. Morgan Kaufmann Publishers/ACL (1997). https://doi.org/10.3115/979617.979670
63. Plandowski, W.: Testing equivalence of morphisms on context-free languages. In: van Leeuwen, J. (ed.) ESA 1994. LNCS, vol. 855, pp. 460–470. Springer, Heidelberg (1994). https://doi.org/10.1007/BFb0049431
64. Plandowski, W.: Satisfiability of word equations with constants is in NEXPTIME. In: STOC, pp. 721–725. ACM (1999). https://doi.org/10.1145/301250.301443
65. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM **51**(3), 483–496 (2004). https://doi.org/10.1145/990308.990312
66. Plandowski, W.: An efficient algorithm for solving word equations. In: Kleinberg, J.M. (ed.) STOC, pp. 467–476. ACM (2006). https://doi.org/10.1145/1132516.1132584
67. Plandowski, W., Rytter, W.: Application of Lempel-Ziv encodings to the solution of word equations. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 731–742. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055097
68. Razborov, A.A.: On systems of equations in free groups. Ph.D. thesis, Steklov Institute of Mathematics (1987). (in Russian)
69. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
70. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. **302**(1–3), 211–222 (2003). https://doi.org/10.1016/S0304-3975(02)00777-6
71. Saarela, A.: On the complexity of Hmelevskii's theorem and satisfiability of three unknown equations. In: Diekert, V., Nowotka, D. (eds.) DLT 2009. LNCS, vol. 5583, pp. 443–453. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02737-6_36
72. Sahinalp, S.C., Vishkin, U.: Symmetry breaking for suffix tree construction. In: Leighton, F.T., Goodrich, M.T. (eds.) SODA, pp. 300–309. ACM (1994). https://doi.org/10.1145/195058.195164
73. Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression. J. Discrete Algorithms **3**(2–4), 416–430 (2005). https://doi.org/10.1016/j.jda.2004.08.016
74. Schmidt-Schauß, M.: Unification of stratified second-order terms (1994). Internal Report 12/94, Johann-Wolfgang-Goethe-Universität

75. Schmidt-Schauß, M.: A decision algorithm for stratified context unification. J. Log. Comput. **12**(6), 929–953 (2002). https://doi.org/10.1093/logcom/12.6.929
76. Schmidt-Schauß, M.: Decidability of bounded second order unification. Inf. Comput. **188**(2), 143–178 (2004). https://doi.org/10.1016/j.ic.2003.08.002
77. Schmidt-Schauß, M., Schulz, K.U.: On the exponent of periodicity of minimal solutions of context equations. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 61–75. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0052361
78. Schmidt-Schauß, M., Schulz, K.U.: Solvability of context equations with two context variables is decidable. J. Symb. Comput. **33**(1), 77–122 (2002). https://doi.org/10.1006/jsco.2001.0438
79. Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization. In: Schulz, K.U. (ed.) IWWERT 1990. LNCS, vol. 572, pp. 85–150. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55124-7_4
80. Storer, J.A., Szymanski, T.G.: The macro model for data compression. In: STOC, pp. 30–39 (1978)