# Word Equations in Nondeterministic Linear Space*

**Artur Jeż**

**Institute of Computer Science, University of Wrocław, Wrocław, Poland**
aje@cs.uni.wroc.pl

## Abstract

Satisfiability of word equations is an important problem in the intersection of formal languages and algebra: Given two sequences consisting of letters and variables we are to decide whether there is a substitution for the variables that turns this equation into true equality of strings. The computational complexity of this problem remains unknown, with the best lower and upper bounds being, respectively, NP and PSPACE. Recently, the novel technique of recompression was applied to this problem, simplifying the known proofs and lowering the space complexity to (nondeterministic) $\mathcal{O}(n \log n)$. In this paper we show that satisfiability of word equations is in nondeterministic linear space, thus the language of satisfiable word equations is context-sensitive. We use the known recompression-based algorithm and additionally employ Huffman coding for letters. The proof, however, uses analysis of how the fragments of the equation depend on each other as well as a new strategy for nondeterministic choices of the algorithm, which uses several new ideas to limit the space occupied by the letters.

**1998 ACM Subject Classification** F.4.3 [Formal Languages] Decision Problems, Classes Defined by Grammars or Automata, F.4.2 Grammars and Other Rewriting Systems, F.2.2 Nonnumerical Algorithms and Problems

**Keywords and phrases** Word equations, string unification, context-sensitive languages, space efficient computations, linear space

**Digital Object Identifier** 10.4230/LIPIcs.ICALP.2017.95

## 1 Introduction

Solving *word equations* was an intriguing problem since the dawn of computer science, motivated first by its ties to Hilbert's 10th problem. Initially it was conjectured that this problem is undecidable, which was disproved in a seminal work of Makanin [10]. At first little attention was given to computational complexity of Makanin's algorithm and the problem itself; these questions were reinvestigated in the '90 [6, 18, 9], culminating in the EXPSPACE implementation of Makanin's algorithm by Gutiérrez [5].

The connection to compression was first observed by Plandowski [16], who showed that a length-minimal solution of size $N$ has a compressed representation of size $\mathsf{poly}(n, \log N)$. Plandowski further explored this approach [14] and proposed a PSPACE algorithm [13], which is the best bound up to date; a simpler PSPACE solution also based on compression was proposed by Jeż [8]. On the other hand, this problem is only known to be NP-hard, and it is conjectured that it is in NP.

The importance of these mentioned algorithms lays with the possibility to extend them (in nontrivial ways) to various scenarios: free groups [11, 1, 3], representation of all solutions [15, 8, 17], traces [12, 2], graph groups [4], terms [7] and others.

* Work supported under National Science Centre, Poland project number 2014/15/B/ST6/00615.





While the computational complexity of word equations remains unknown, its exact space complexity is intriguing as well: Makanin's algorithm uses exponential space [5], Plandowski [13] gave no explicit bound on the space usage of his algorithm, a rough estimation is $\mathsf{NSPACE}(n^5)$, the recent solution of Jeż [8] yields $\mathsf{NSPACE}(n \log n)$. Moreover, for $\mathcal{O}(1)$ variables a linear bound on space complexity was shown [8]; recall that languages recognisable in nondeterministic linear space are exactly the context-sensitive languages.

In this paper we show that satisfiability of word equations can be tested in nondeterministic linear space in terms of the number of bits of the input, thus showing that the language of satisfiable word equations is context-sensitive (and by the famous Immerman–Szelepcsényi theorem: the language of unsatisfiable word equations). The employed algorithm is a (variant of) algorithm of Jeż [8], which additionally uses Huffman coding for letters in the equation. On the other hand, the actual proof uses a different encoding of letters, which extends the ideas used in a (much simpler) proof in case of $\mathcal{O}(1)$ variables [8, Section 5]; the other new ingredient is a different strategy of compression: roughly speaking, previously a strategy that minimised the length of the equation was used. Here, a more refined strategy is used: it simultaneously minimises the size of a particular bit encoding, enforces that changes in the equation (during the algorithm) are local, and limits the amount of new letters that are introduced to the equation.

The bound holds when letters and variables in the input are encoded using an arbitrary encoding, in particular, the Huffman coding (so the most efficient one) is allowed.

## 2 The (known) algorithm

We first present a slight variation of the algorithm of Jeż [8] and the notions necessary to understand how it works. The proofs are omitted, yet they should be intuitively clear.

**Notions.** The word equation is a pair $(U, V)$, written as $U = V$, where $U, V \in (\Gamma \cup \mathcal{X})^*$ and $\Gamma$ and $\mathcal{X}$ are disjoint alphabets of *letters* and *variables*, both are collectively called *symbols*. By $n_X$ we denote the number of occurrences of $X$ in the (current) equation; in the algorithm $n_X$ does not change till $X$ is removed from the equation, in which case $n_X$ becomes 0. A *substitution* is a morphism $S : \mathcal{X} \cup \Gamma \to \Gamma'^*$, where $\Gamma' \supseteq \Gamma$ and $S(a) = a$ for every $a \in \Gamma$, a substitution naturally extends to $(\mathcal{X} \cup \Gamma)^*$. A *solution* of an equation $U = V$ is a substitution $S$ such that $S(U) = S(V)$; given a solution $S$ of an equations $U = V$ we call $S(U)$ the *solution word*. We allow the solution to use letters that are not present in the equation, this does not change the satisfiability: all such letters can be changed to a fixed letter from $\Gamma$, and the obtained substitution is still a solution. Yet, the proofs become easier, when we allow the usage of such letters. The alphabet $\Gamma'$ is usually given implicitly: as the set of letters used by the substitution. A *block* is a string $a^\ell$ with $\ell \geq 1$ that cannot be extended to the left nor to the right with $a$.

As we deal with linear-space, the encoding used by the input equation matters. We assume only that the input is given by a fixed (uniquely decodable) coding: each symbol in the input is always given by the same bitstring and given the bitstrings representing the sides of the equation there is only one pair of strings (over $\Gamma \cup \mathcal{X}$) that is encoded in this way. It is folklore that among such codes the Huffman code yields the smallest space consumption (counted in bits) and moreover the Huffman coding can be efficiently computed, also in linear space. As we focus on space counted in bits and use encodings, by $||\alpha||$ we denote the space consumption of the encoding of $\alpha$, the encoding shall be always clear from the context. Furthermore, whenever we talk about space complexity, it is counted in bits.

**Nondeterministic Linear Space.** We recall some basic facts about the nondeterministic space-bounded computation. A nondeterministic procedure *is sound*, when given a unsatisfiable word equation $U = V$ it cannot transform it to a satisfiable one, regardless of the nondeterministic choices; a procedure is *complete*, if given a satisfiable equation $U = V$ for some nondeterministic choices it returns a satisfiable equation $U' = V'$. A composition of sound (complete) procedures is sound (complete, respectively). It is enough that we show linear-space bound for one particular computation: as the bound is known, we limit the space available to the algorithm and reject the computations exceeding it.

**The algorithm.** We use (a variant of) recompression algorithm [8], which conceptually applies the following two operations on $S(U)$ and $S(V)$: given a string $w$ and alphabet $\Gamma$

- the $\Gamma$ block compression of $w$ is a string $w'$ obtained by replacing every block $a^\ell$, where $a \in \Gamma$ and $\ell \geq 2$, with a fresh letter $a_\ell$;
- the $(\Gamma_\ell, \Gamma_r)$ pair compression of $w$, where $\Gamma_\ell, \Gamma_r$ is a partition of $\Gamma$, is a string $w'$ obtained by replacing every occurrence of a pair $ab \in \Gamma_\ell\Gamma_r$ with a fresh letter $c_{ab}$.

A *fresh* letter means that it is not currently used in the equation, nor in $\Gamma$, yet each occurrence of a fixed $ab$ is replaced with the same letter. The $a_\ell$ and $c_{ab}$ are just notation conventions, the actual letters in $w'$ do not store the information how they were obtained. For shortness, we call $\Gamma$ block compression the $\Gamma$ compression or block compression, when $\Gamma$ is clear from the context; similar convention applies to $(\Gamma_\ell, \Gamma_r)$ pair compression, called $(\Gamma_\ell, \Gamma_r)$ compression or pair compression, when $(\Gamma_\ell, \Gamma_r)$ is clear from the context. We say that a pair $ab \in \Gamma_\ell\Gamma_r$ is *covered* by a partition $\Gamma_\ell, \Gamma_r$.

The intuition is that the algorithm aims at performing those compression operations on the solution word and to this end it modifies the equation a bit and then performs the compression operations on $U$ and $V$ (and conceptually also on the solution, i.e. on $S(X)$ for each variable $X$). Below we describe, how it is performed on the equation.

BlockComp: For the equation $U = V$ and the alphabet $\Gamma$ of letters in this equation for each variable $X$ we first guess the first and last letter of $S(X)$ as well as the lengths $\ell, r$ of the longest prefix consisting only of $a$, called $a$-prefix, and $b$-suffix (defined similarly) of $S(X)$. Then we replace $X$ with $a^\ell X b^r$ (or $a^\ell b^r$ or $a^\ell$ when $S(X) = a^\ell b^r$ or $S(X) = a^\ell$); this operation is called *popping $a$-prefix and $b$-suffix*. Then we perform the $\Gamma$-block compression on the equation (this is well defined, as we can treat variables as symbols from outside $\Gamma$).

PairComp: For the alphabet $\Gamma$, which will always be the alphabet of letters in the equation right before the block compression we partition $\Gamma$ into $\Gamma_\ell$ and $\Gamma_r$ (in a way described in Section 3.2) and then for each variable $X$ guess whether $S(X)$ begins with a letter $b \in \Gamma_r$ and if so, replace $X$ with $bX$ or $b$, when $S(X) = b$, and then do a symmetric action for the last letter and $\Gamma_\ell$; this operation is later referred to as *popping letters*. Then we perform the $(\Gamma_\ell, \Gamma_r)$ compression on the equation.

LinWordEq works in *phases*, until an equation with both sides of length 1 is obtained: in a single phase it establishes the alphabet $\Gamma$ of letters in the equation, performs the $\Gamma$ compression and then repeats: guess the partition of $\Gamma$ to $\Gamma_\ell$ and $\Gamma_r$ and perform the $(\Gamma_\ell, \Gamma_r)$ compression, until each pair $ab \in \Gamma^2$ was covered by some partition.

**Correctness.** Given a solution $S$ we say that some nondeterministic choices *correspond to* $S$, if they are done as if LinWordEq knew $S$. For instance, it guesses correctly the first letter of $S(X)$ or whether $S(X) = \epsilon$. (The choice of a partition does not fall under this category.)

▶ **Lemma 1** ([8, Lemma 2.8 and Lemma 2.10])**.** BlockComp *is sound and complete; to be more precise, for any solution $S$ of an equation $U = V$ for the nondeterministic choices*

*corresponding to $S$ the returned equation $U' = V'$ has a solution $S'$ such that $S'(U')$ is the $\Gamma$ compression of $S(U)$ and $S'(X)$ is obtained from $S(X)$ by removing the $a$-prefix and $b$-suffix, where $a$ is the first letter of $S(X)$ and $b$ the last, and then performing the $\Gamma$ compression.*

*When $\Gamma_\ell$ and $\Gamma_r$ are disjoint, the* PairComp$(\Gamma_\ell, \Gamma_r)$ *is sound and complete; to be more precise, for any solution $S$ of an equation $U = V$ for the nondeterministic choices corresponding to $S$ the returned equation $U' = V'$ has a solution $S'$ such that $S'(U')$ is the $(\Gamma_\ell, \Gamma_r)$ compression of $S(U)$ and $S'(X)$ is obtained from $S(X)$ by removing the first letter of $S(X)$, if it is in $\Gamma_r$, and the last, if it is in $\Gamma_\ell$, and then performing the $(\Gamma_\ell, \Gamma_r)$ compression.*

The solution $S'$ from Lemma 1 is called a *solution corresponding to $S$* after $(\Gamma_\ell, \Gamma_r)$ compression ($\Gamma$ compression, respectively); we also talk about a *solution corresponding to $S$*, when the compression operation is clear from the context and extend this notion to a solution corresponding to $S$ after a phase. What is important later on is how $S'$ is obtained from $S$: it is modified as if the subprocedures knew first/last letter of $S(X)$ and popped appropriate letters from the variables and then compressed pairs/blocks in substitution for variables.

Lemma 1 yields the soundness and completeness of LinWordEq, for the termination we observe that iterating the compression operations shortens the string by a constant fraction, thus the length of a solution word shortens by a constant fraction in each phase.

▶ **Lemma 2.** *Let $w$ be a string over an alphabet $\Gamma$ and $w'$ a string obtained from $w$ by a $\Gamma$ compression followed by a sequence of $(\Gamma_\ell, \Gamma_r)$ compressions (where $\Gamma_\ell, \Gamma_r$ is a partition of $\Gamma$) such that each pair $ab \in \Gamma^2$ is covered by some partition. Then $|w'| \leq \frac{2|w|+1}{3}$.*

▶ **Theorem 3.** LinWordEq *is sound, complete and terminates (for appropriate nondeterministic choices) for satisfiable equations. It runs in linear (bit) space.*

In the following, we will also need one more technical property of block compression.

▶ **Lemma 4.** *Consider a solution $S$ during a phase with nondeterministic choices corresponding to $S$ and the corresponding solution $S'$ of $U' = V'$ after the block compression. Then $S'(U')$ has no two consecutive letters $aa \in \Gamma$.*

This is true after block compression and afterwards no letters from $\Gamma$ are introduced.

**Compressing blocks in small space.** Storing, even in a concise way, the lengths of popped prefixes and suffixes in $\Gamma$ compression makes attaining the linear space difficult. This was already observed [8] and a linear-space implementation of BlockComp was given [8]. It performs a different set of operations, yet the effect is the same as for BlockComp. Instead of explicitly naming the lengths of blocks, we treat them as integer parameters; then we declare, which maximal blocks are of the same length (those lengths depend linearly on the parameters); verifying the validity of such a guess is done by writing a system of (linear) Diophantine equations that formalise those equalities and checking its satisfiability. This procedure is described in detail in [8, Section 4]. In the end, it can be implemented in linear bitspace.

▶ **Lemma 5** ([8, Lemma 4.7])**.** BlockComp *can be implemented in space linear in the bit-size of the equation*

**Huffman coding.** At each step of the algorithm we encode letters (though not variables) in the equation using Huffman coding. This may mean that when going from $U = V$ to $U' = V'$ the encoding of letters changes and in fact using the former encoding in the latter equation

may lead to super-linear space (imagine that we pop from each variable a letter that has a very long code). Using standard methods changing the encoding during a transition from $U = V$ to $U' = V'$ can be done in $\mathcal{O}(||U = V|| + ||U' = V'||)$ bit-space.

▶ **Lemma 6.** *Given a string (encoded using some uniquely decodable code), its Huffman coding can be computed in linear bitspace.*

*Each subprocedure of* LinWordEq *that transforms an equation $U = V$ to $U' = V'$ can be implemented in bit-space $\mathcal{O}(||U = V||_1 + ||U' = V'||_2)$, where $||\cdot||_1$ and $||\cdot||_2$ are the Huffman codings for letters in $U = V$ and $U' = V'$, respectively.*

## 3 Space consumption

In order to bound the space consumption, we will use bit-encoding of letters that depends on the current equation. We use the term 'encoding' even though it may assign different codes to different occurrences of the same letter, but two different letters never have the same code. Since we are interested in linear space only, we do not care about the multiplicative $\mathcal{O}(1)$ factors in the space consumption and can assume that our code is prefix-free, say by terminating each encoding with a special symbol \$. We show that such an encoding uses linear space, which also shows that the Huffman encoding of the letters in the equation uses linear space, as Huffman encoding uses the smallest space among the prefix codes.

The idea of our 'encoding' is: for each letter in the current equation we establish an interval $I$ of indices in the original equation (viewed as as string $U_0V_0[1\,..\,|U_0V_0|]$) on which it 'depends' (this has to be formalised) and encode this letter as $U_0V_0[I]\#i$, when it is $i$th in the sequence of letters assigned $I$ and $U_0V_0[I]$ is the original equation restricted to indices in $I$ The dependency is formalised in Section 3.1, while Section 3.2 first gives the high-level intuition and then upper-bound on the used space.

For technical reasons we insert into the equation ending markers at the beginning and end of $U$ and $V$, i.e. write them as $@U@, @V@$ for some special symbol @. Those markers are ignored by the algorithm, yet they are needed for the encoding.

### 3.1 Dependency intervals

First, we need some notation. The input equation is denoted by $U_0 = V_0$, the $U = V$ and $U' = V'$ are used for the current equation and equation after performing some operation. We treat the input equation as a single string $U_0V_0$ and consider its *indices*, i.e. numbers from 1 to $|U_0V_0|$, denoted by letters $i, i', j$ and intervals of such indices, denoted by letter $I, I'$ or $[i\,..\,j]$. The $U_0V_0[I]$ and $U_0V_0[i\,..\,j]$ denotes the substring of $U_0V_0$ restricted to indices in $[I]$ or in $[i\,..\,j]$. We use a partial order $\leq$ on intervals: $[i\,..\,j] \leq [i'\,..\,j']$ if $i \leq i'$ and $j \leq j'$.

In the current equation, i.e. the one stored by LinWordEq, we do not consider indices but rather *positions* and denote them by letters $p, q$. We do not think of them as numbers but rather as pointers: when $U = V$ is transformed by some operation to $U' = V'$ but the letter/variable at position $p$ was not affected by this transformation, we still say that this letter/variable is at position $p$. On the other hand, the affected letters are on positions that were not present in $U = V$. In the same spirit we denote by $p$ the positions in $U = V$ and the corresponding position in $S(U) = S(V)$. We still use the left-to-right ordering on positions, use $p - 1$ and $p + 1$ to denote the previous and next position; we also consider intervals of positions, yet they are used rarely so that they are not confused with intervals of indices, on which we focus mostly. Given an equation $U = V$ and an interval of positions $P$ by $UV[P]$

we denote the string of letters and variables at positions in $P$, again, this notation is used rarely. In the input equation the index and position is the same.

With each position $p$ in the (current) equations (including the endmarkers) we associate *dependency interval* $\mathrm{dep}(p)$, called *depint*; if the depint is a single index $\{i\}$, we denote it $i$. The idea is that the letter at position $p$ is uniquely determined by $U_0V_0[\mathrm{dep}(I)]$ (and the nondeterministic choices of the algorithm), note that it may include both variables and letters. We use the notions of $\subseteq$ and $\supseteq$ for the depints with a usual meaning; we take unions of the them, denoted by $\cup$, but only when the result is an interval. We say that $I$ and $I'$ are similar, denoted as $I \sim I$, if $U_0V_0[I] = U_0V_0[I']$. Given an interval $I$ of indices in $U_0V_0$ by $\mathrm{Pos}(I)$ we denote positions in the current equation whose depint is $I$, i.e. $\mathrm{Pos}(I) = \{p \mid \mathrm{dep}(p) = I\}$. In the analysis it is also convenient to look at positions whose depint is a superset of $I$: $\mathrm{Pos}_{\supseteq}(I) = \{p \mid \mathrm{dep}(p) \supseteq I\}$, this is usually used for $I = \{i\}$

We shall ensure the following properties:

**(I1)** Given a depint $I$, the $\mathrm{Pos}(I)$ is an interval of positions, similarly $\mathrm{Pos}_{\supseteq}(I)$.

**(I2)** Given depints $I, I'$ such that $\mathrm{Pos}(I) \neq \emptyset \neq \mathrm{Pos}(I')$ either $I \leq I'$ or $I \geq I'$.

**(I3)** For depints $I \sim I'$ it holds that $U_0V_0[I] = U_0V_0[I']$.

**Assigning depints to letters.** When $X$ at position $p$ pops a letter into position $p'$ then $\mathrm{dep}(p') \leftarrow \mathrm{dep}(p)$ (which is the position of this occurrence of $X$ in the input equation). Whenever we perform the $(\Gamma_\ell, \Gamma_r)$ compression then in parallel for each position $p$ such that $UV[p] \in \Gamma_\ell$ we assign $\mathrm{dep}(p) \leftarrow \mathrm{dep}(p) \cup \mathrm{dep}(p+1)$ ($p+1$ may be a a position variable or an endmarker). Then we perform a symmetric action for positions whose letters are in $\Gamma_r$ (so for $p-1$).

For $\Gamma$ compression, we perform in parallel the following operation for each block (perhaps of length 1) of a letter in $\Gamma$: given a maximal block $a^\ell$ at positions $p, p+1, \ldots, p+\ell-1$ we set the depints of those positions to $\bigcup_{i=-1}^{\ell} \mathrm{dep}(p+i)$ (note that $p-1$ and $p+\ell$ are included).

In the following we mostly focus on $\mathrm{Pos}_{\supseteq}(i)$. As this is an interval of positions, we visualize that $\mathrm{Pos}_{\supseteq}(I)$ extends to the neighbouring positions. Thus we will refer to operations of changing the depints before the block compression and pair compression as *extending of* $\mathrm{Pos}_{\supseteq}(I)$ to new positions; those positions *get their depints extended*. Note that this notion does not apply to the case when we pop letters from variables.

Depints defined in this way satisfy the conditions (I1–I3).

▶ **Lemma 7.** *(I1–I3) hold during* LinWordEq.

**Proof.** We first show (I1) for $\mathrm{Pos}_{\supseteq}(i)$. The proof is by induction; this is true at the beginning. If we make a union of depints, a position adjacent to a position in $\mathrm{Pos}_{\supseteq}(i)$ symbol can become part of $\mathrm{Pos}_{\supseteq}(i)$ (this can be iterated when the depints are changed before the blocks compression), which is fine. During the compression, we compress symbols on positions with the same depints, so this is fine. When we pop a letter from variable at position $p$ to position $p'$ then $\mathrm{dep}(p') = \mathrm{dep}(p) \in \mathrm{Pos}_{\supseteq}(i)$ and by inductive assumption $\mathrm{Pos}_{\supseteq}(i)$ was an interval, which shows the claim.

We now show by induction that $i \leq i'$ implies $\mathrm{Pos}_{\supseteq}(i) \leq \mathrm{Pos}_{\supseteq}(i')$. Clearly this holds at the beginning, as then $\mathrm{Pos}_{\supseteq}(i) = \mathrm{Pos}(i) = \{i\}$ and $\mathrm{Pos}_{\supseteq}(i') = \mathrm{Pos}(i') = \{i'\}$. Consider the moment, in which the condition $\mathrm{Pos}_{\supseteq}(i) \leq \mathrm{Pos}_{\supseteq}(i')$ is first violated, by symmetry it is enough to consider the case in which the first position in $\mathrm{Pos}_{\supseteq}(i')$ is smaller than the first in $\mathrm{Pos}_{\supseteq}(i)$. If this position was just popped then it cannot be popped to the right, as the position of popping variable is in $\mathrm{Pos}_{\supseteq}(i')$. So it was popped to the left. But then the variable that popped it was on position in $\mathrm{Pos}_{\supseteq}(i')$ and by induction assumption $\mathrm{Pos}_{\supseteq}(i') \geq \mathrm{Pos}_{\supseteq}(i)$,

so it had a position from $\mathrm{Pos}_{\supseteq}(i)$ to its left, contradiction. The other option is that this happened when a depint of a position was changed so that it got into $\mathrm{Pos}_{\supseteq}(i')$. But then the position to its right was in $\mathrm{Pos}_{\supseteq}(i')$ and by induction assumption either this position was in $\mathrm{Pos}_{\supseteq}(i)$ or some position to the left of it was; in both cases the position also got into $\mathrm{Pos}_{\supseteq}(i)$.

Now (I1) for $\mathrm{Pos}_{\supseteq}([i\,..\,j])$ for an arbitrary depint $[i\,..\,j]$ follows: $\mathrm{Pos}_{\supseteq}([i\,..\,j]) = \bigcap_{k=i}^{j} \mathrm{Pos}_{\supseteq}(k)$ and as each $\mathrm{Pos}_{\supseteq}(k)$ is an interval, also $\mathrm{Pos}_{\supseteq}([i\,..\,j])$ is.

For the purpose of the proof, define $\mathrm{Pos}_{\subseteq}(I) = \{p \mid \mathrm{dep}(p) \subseteq I\}$ (a dual notion to $\mathrm{Pos}_{\supseteq}(I)$).

▶ **Claim 8.** *$\mathrm{Pos}_{\subseteq}(I)$ consists of consecutive positions. Given two similar depints $I \sim I'$ $UV[\mathrm{Pos}_{\subseteq}(I)]$ and $UV[\mathrm{Pos}_{\subseteq}(I')]$ are equal as sequences of symbols and the corresponding positions in them have similar depints.*

The proof follows by a simple, yet tedious induction and it is omitted.

Claim 8 is stronger than (I3) and so it implies it. Concerning (I1): $\mathrm{Pos}(I) = \mathrm{Pos}_{\supseteq}(I) \cap \mathrm{Pos}_{\subseteq}(I)$; as both are intervals, also $\mathrm{Pos}(I)$ is.

Concerning (I2), we show a stronger statement: given positions $p, p+1$ it holds that $\mathrm{dep}(p) \leq \mathrm{dep}(p{+}1)$. Let $i, i'$ be the leftmost indices in $\mathrm{dep}(p), \mathrm{dep}(p{+}1)$, respectively. Assume for the sake of contradiction that $i > i'$. We already showed that then $\mathrm{Pos}_{\supseteq}(i) \geq \mathrm{Pos}_{\supseteq}(i')$. So if $p+1 \in \mathrm{Pos}_{\supseteq}(i') \leq \mathrm{Pos}_{\supseteq}(i) \ni p$ then also $p \in \mathrm{Pos}_{\supseteq}(i')$, i.e. $i' \in \mathrm{dep}(p)$. As $i' < i$ then the leftmost index in $\mathrm{dep}(p')$ cannot be $i$. The proof for rightmost index is similar. ◀

**Encoding of letters.** Letters in $\mathrm{Pos}(I)$ are encoded as $U_0V_0[I]\#1$, $U_0V_0[I]\#2$, etc. Note, that there is no a priori bound on the size of such numbers. Furthermore, if $I' \sim I$ then encoding $I\#i$ and $I'\#i$ is the same (these are the same symbols by (I3).

## 3.2 Pair compression strategy

We assume that LinWordEq makes the nondeterministic choices according to the solution, thus the space consumption of a run depends only on the choices of the partitions during pair compression, called *a strategy*. We describe a linear-space strategy.

**Idea.** Imagine we ensured that during one phase each variable popped $\mathcal{O}(1)$ letters and each $\mathrm{Pos}_{\supseteq}(i)$ expanded by $\mathcal{O}(1)$ letters. Then $|\mathrm{Pos}_{\supseteq}(i)| = \mathcal{O}(1)$: we introduced $\mathcal{O}(1)$ positions to $\mathrm{Pos}(i)$, say at most $k$, and by Lemma 2 among positions in $\mathrm{Pos}_{\supseteq}(i)$ at the beginning of the phase there were at least $2/3$ took part in compression, so their number dropped by $1/3$; thus $|\mathrm{Pos}_{\supseteq}(i)| \leq 3k$. As a result, $|\mathrm{Pos}(I)| \leq 3k$ for each depint $I$: as $\mathrm{Pos}(I) \subseteq \mathrm{Pos}_{\supseteq}(i)$ for $i \in I$. This would yield that the whole bit-space used for the encoding is linear: each number $m$ used in $U_0V_0[I]\#m$ is at most $3k = \mathcal{O}(1)$, so they increase the size by at most a constant fraction. On the other hand, the depints consume:

$$\sum_{I:\mathrm{depint}} ||U_0V_0[I]|| \cdot |\mathrm{Pos}(I)| = \sum_{i:\mathrm{index}} ||U_0V_0[i]|| \cdot |\mathrm{Pos}_{\supseteq}(i)|$$

(a simple proof is given later) and the right hand side is linear in terms of the input equation: $|\mathrm{Pos}_{\supseteq}(i)| = \mathcal{O}(1)$ and $\sum_{i:\mathrm{index}} ||U_0V_0[i]||$ is the the bit-size of the input equation.

It remains to ensure that $\mathrm{Pos}_{\supseteq}(i)$ do not extend too much and variables do not pop too much letters. Given a phase, we call a letter *new*, if it was introduced during this phase. New letters cannot be popped nor can $\mathrm{Pos}_{\supseteq}(i)$ be extended to positions with new letters. Thus they are used to prevent extending $\mathrm{Pos}_{\supseteq}(i)$ and popping: it is enough to ensure that

the first/last letter of a variable is new and that a letter on the position to the left/right of $\mathrm{Pos}_{\supseteq}(I)$ is new.

Unfortunately, we cannot ensure this for all variables $\mathrm{Pos}_{\supseteq}(i)$. We can make this true *in expectation*: given a random partition there is a $1/4$ probability that a fixed pair is compressed (and the resulting letter is new). This requires formalisation and calculations.

**Strategy.** Given a solution $S$ of an equation we say that a variable $X$ is *left blocked* if $S(X)$ has at most one letter or the first or second letter in $S(X)$ is new, otherwise a variable is *left unblocked*; define *right blocked* and *right unblocked* variables similarly. An index $i$ is *left blocked* if in $S(U)$ (or $S(V)$, respectively) there is at most position to the left of $\mathrm{Pos}_{\supseteq}(i)$ or one of the letters on the positions one and two to the left of $\mathrm{Pos}_{\supseteq}(i)$ is new, otherwise $i$ is *left unblocked*; define *right blocked* and *right unblocked* indices similarly.

▶ **Lemma 9.** *Consider a solution $S = S_0$ and consecutive solutions $S_1, S_2, \ldots$ corresponding to it during a phase. If a variable $X$ becomes left (right) blocked for some $S_k$, then it is left (right, respectively) blocked for each $S_\ell$ for $\ell \geq k$ and it pops to the left (right, respectively) at most 1 letter after it became left (right, respectively) blocked. If an index $i$ becomes left (right) blocked for some $S_k$ then it is left (right, respectively) blocked for each $S_\ell$ for $\ell \geq k$ and at most one letter to the left (right, respectively) will have its depint extended by $i$ after $i$ became left (right, respectively) blocked.*

The proof follows by a simple case inspection and it is omitted.

The strategy iterates steps 1, 2, 3 and 4. In a step $i$ it chooses a partition so that the corresponding $i$-th sum below decreases by $1/2$, unless this sum is already 0:

$$\sum_{X\in\mathcal{X} \text{ left unblocked}} n_X \cdot ||X|| + \sum_{X\in\mathcal{X} \text{ right unblocked}} n_X \cdot ||X|| \tag{1}$$

$$\sum_{i\text{: left unblocked index}} ||U_0V_0[i]|| + \sum_{i\text{: right unblocked index}} ||U_0V_0[i]|| \tag{2}$$

$$\sum_{X\in\mathcal{X} \text{ left unblocked}} n_X + \sum_{X\in\mathcal{X} \text{ right unblocked}} n_X \tag{3}$$

$$\sum_{i\text{: left unblocked index}} 1 + \sum_{i\text{: right unblocked index}} 1 \tag{4}$$

The idea of the steps is: (1) upper-bounds the increase of bit-size of depints in the equation after popping letters. So by iteratively halving it we ensure that total encoding increase caused by popping letters is small. Similarly, (2) upper-bounds the increase due to expansion of indices to new depints. The following (3) is connected (in a more complex way) to an increase, after popping, of number of bits used for numbers in the encoding. Similarly (4) to an increase after the extension of depints.

▶ **Lemma 10.** *During the pair compression* LinWordEq *can always choose a partition that at least halves the value of a chosen non-zero sum among* (1)–(4).

**Proof.** Consider (1) and take a random partition, in the sense that each letter $a \in \Gamma$ goes to the $\Gamma_\ell$ with probability $1/2$ and to $\Gamma_r$ with probability $1/2$. Let us fix a variable $X$ and its side, say left. What happens with $n_X \cdot ||X||$ in (1) in the sum corresponding to left unblocked variables? If $X$ is left blocked then, by Lemma 9, it will stay left blocked and so the contribution is and will be 0. If it is left unblocked, then its two first letters $a, b$ are not new, so they are in $\Gamma$. If $S(X)$ has only those two letters, then with probability $1/2$ the $a$

will be in $\Gamma_r$ and it will be popped and $X$ will become left blocked (as $S(X)$ has only one letter), the same analysis applies, when the third leftmost letter is new. The remaining case is that the three leftmost letters in $S(X)$ are not new, let them be $a, b, c \in \Gamma$. By Lemma 4 $a \neq b \neq c$. With probability 1/4 $ab \in \Gamma_\ell\Gamma_r$ and with probability 1/4 $bc \in \Gamma_\ell\Gamma_r$. Those events are disjoint (as in one $b \in \Gamma_r$ and in the other $b \in \Gamma_\ell$) and so their union happens with probability 1/2. In both cases $X$ will become left blocked, as a new letter is its first or second in $S(X)$. In all uninvestigated cases the contribution of $n_X \cdot ||X||$ cannot raise, which shows the claim in this case. The case of (3) is shown in the same way.

For (2), the analysis for an index $i$ that is left unblocked is similar, but this time we consider the positions to the left of $\mathrm{Pos}_{\supseteq}(i)$ and $\mathrm{Pos}_{\supseteq}(i)$ can extend to them (instead of letters being popped from variables in case of (1)) and some of them may be compressed to one. Note that if there are no letters to the left/right then this index is blocked from this side. The case of (4) is shown in the same way. ◀

**Space consumption.** We now give the linear space bound on the size of equation. This formalises the intuition from the beginning of Section 3.2. As a first step, we show an upper-bound on the encoding size of the equation; define

$$H_d(U,V) = \sum_{i:\text{index}} ||U_0V_0[i]|| \cdot |\mathrm{Pos}_{\supseteq}(i)|, \quad H_n(U,V) = \sum_{i:\text{index}} 2|\mathrm{Pos}_{\supseteq}(i)| \cdot \log(|\mathrm{Pos}_{\supseteq}(i)|+1) \ ,$$

and $H(U,V) = H_d(U,V) + H_n(U,V)$. $H_d$ corresponds to the size of $U_0V_0[I]$ in the encoding and $H_d$: of the numbers in the encoding.

▶ **Lemma 11.** *Given the equation $(U,V)$ it holds that $||(U,V)|| \leq H_d(U,V) + H_n(U,V)$.*

The proof follows by simple symbolic transformation of the definitions.

Instead of showing a linear bound on $||(U,V)||$ we give a linear bound on $H(U,V)$. Recall that $(U_0,V_0)$ denotes the input equation.

▶ **Lemma 12.** *Consider an equation $U = V$, its solution $S$, a phase of* LinWordEq *which makes the nondeterministic choices according to $S$ and partitions according to the strategy. Let the returned equation be $(U',V')$. Then $H(U',V') \leq \frac{5}{6}H(U,V) + \alpha||(U_0,V_0)||$ and in a phase $H$ on intermediate equations is at most $\beta H(U,V) + \gamma||(U_0,V_0)||$ for some constants $\alpha, \beta, \gamma$.*

**Proof.** We separately estimate the $H_d$ and $H_n$. Concerning $H_d$, let us first estimate $||U_0V_0[\mathrm{dep}(p)]||$ summed over positions $p$ of letters popped into the equation during a phase (note, this does not include the size of numbers used in the encoding). For each variable we pop perhaps several letters to the left and right before block compression, but those letters are immediately replaced with single letters, so we count each as 1; also, when this side of a variable becomes blocked, it can pop at most one letter. Otherwise, a side of a variable pops at most 1 letter per pair compression, in which it is unblocked from this side. Note that the depint is the same as for variable, so the encoding size is $||X||$. So in total the bit-size of popped letters is at most:

$$\underbrace{\sum_{X\in\mathcal{X}} 2n_X \cdot ||X||}_{\text{block compression}} + \underbrace{\sum_{X\in\mathcal{X}} 2n_X \cdot ||X||}_{\text{after } X \text{ becomes blocked}} + \\ + \sum_{P:\text{ partition}} \left( \sum_{\substack{X\in\mathcal{X} \\ \text{left unblocked in } P}} n_X \cdot ||X|| + \sum_{\substack{X\in\mathcal{X} \\ \text{right unblocked in } P}} n_X \cdot ||X|| \right). \quad (5)$$

Observe that the third sum (the one summed over all partitions) at the beginning of the phase is equal to $\sum_X 2n_X \cdot ||X||$, as no side of the variable is blocked, and by the strategy point (1) its value at least halves every 4th pair compression (and it cannot increase, as by Lemma 9 no side of the variable can cease to be blocked). Thus (5) is at most

$$4\sum_X n_X \cdot ||X|| + 8\sum_X n_X \cdot ||X|| \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) = 20\sum_X n_X \cdot ||X|| \leq 20||(U_0, V_0)|| \ .$$

We now similarly estimate how many positions got into $\mathrm{Pos}_{\supseteq}(i)$ due to expansion of $\mathrm{Pos}_{\supseteq}(i)$: $\mathrm{Pos}_{\supseteq}(i)$ can expand to two letters during the block compression (to be more precise: to positions that are inside a block and to the positions to the let/right ones, but positions in a block are replaced with a single letter and one of them was in $\mathrm{Pos}_{\supseteq}(i)$) to one position at each side after $i$ becomes blocked and by one position for each partition $P$ in which this side of $i$ is not blocked. So the increase in the bit-size is

$$\sum_{i\colon \text{index}} 4||U_0V_0[i]|| + \sum_{P\colon \text{partition}} \Bigg( \sum_{\substack{i\colon \text{index} \\ \text{left unblocked in } P}} ||U_0V_0[i]|| + \sum_{\substack{i\colon \text{index} \\ \text{right unblocked in } P}} ||U_0V_0[i]|| \Bigg) \tag{6}$$

and as in (5) similarly at the beginning of the phase the second sum (so the one summed by partitions) is $\sum_{i\colon \text{index}} 2||U_0V_0[i]|| = 2||(U_0, V_0)||$ and it at least halves every 4th partition, by strategy point (2). Thus similar calculations show that (6) is at most $20||(U_0, V_0)||$.

On the other hand, the number of positions in $\mathrm{Pos}_{\supseteq}(i)$ drops till the end of the phase by at least $\frac{|\mathrm{Pos}_{\supseteq}(i)|}{3} - 1$ due to compression:

1. If $U_0V_0[i]$ is a letter, then $\mathrm{Pos}_{\supseteq}(i)$ are all positions of letters and Lemma 2 yields that $\mathrm{Pos}_{\supseteq}(i)$ looses at least $\frac{|\mathrm{Pos}_{\supseteq}(i)|-1}{3}$ positions.
2. If $U_0V_0[i]$ is an ending marker, then the marker itself is unchanged and the remaining positions in $\mathrm{Pos}_{\supseteq}(i)$ are letter-positions and Lemma 2 applies to them, so $\mathrm{Pos}_{\supseteq}(i)$ looses at least $\frac{|\mathrm{Pos}_{\supseteq}(i)|-2}{3} < \frac{|\mathrm{Pos}_{\supseteq}(i)|}{3} - 1$ positions.
3. If $U_0V_0[i]$ is a variable then $\mathrm{Pos}_{\supseteq}(i)$ includes the position of a variable and Lemma 2 applies to strings of letters to the left and right, say of length $\ell, r$, where $\ell + r = |\mathrm{Pos}_{\supseteq}(i)| - 1$.

Then due to compressions $\mathrm{Pos}_{\supseteq}(i)$ looses at least $\frac{\ell-1}{3} + \frac{r-1}{3} = \frac{|\mathrm{Pos}_{\supseteq}(i)|}{3} - 1$ positions.
Thus:

$$\begin{aligned} H_d(U', V') \leq\ & \underbrace{40||(U_0, V_0)||}_{\text{new positions in depints}} + \underbrace{\sum_{i\colon \text{index}} ||U_0V_0[i]|| \cdot \left(\frac{2}{3}|\mathrm{Pos}_{\supseteq}(i)| + 1\right)}_{\text{old positions lost}} \\ &= 40||(U_0, V_0)|| + \sum_{i\colon \text{index}} \frac{2}{3}||U_0V_0[i]|| \cdot |\mathrm{Pos}_{\supseteq}(i)| + \sum_{i\colon \text{index}} ||U_0V_0[i]|| \\ &= 41||(U_0, V_0)|| + \frac{2}{3}H_d(U, V) \ . \end{aligned}$$

We also estimate the maximal value of $H_d$ during the phase, as for intermediate equations we cannot guarantee that the compression reduced the length of all letters. We already showed that in a phase we increase $H_d$ by $40||(U_0, V_0)||$. This yields a bound of $H_d(U, V) + 40||(U_0, V_0)||$, which shows the part of the claim of Lemma for $H_d$.

Concerning $H_n$, for an index $i$ let $k_i, o_i, e_i$ denote, respectively: $|\mathrm{Pos}_{\supseteq}(i)|$ at the beginning of the phase, number of positions of letters popped from a variable with depint $i$ and number of positions to whose depint $i$ extended. First we estimate $\sum_{i\colon \text{index}} h(o_i)$ and $\sum_{i\colon \text{index}} h(e_i)$

and then use those estimations to calculate the bound on $H_n(U', V')$. We first inspect the case of $o_i$; let $P_1, P_2, \ldots$ denote the consecutive partitions in phase. We show that

$$\sum_{i\text{: index}} h(o_i) \leq \sum_{X\in\mathcal{X}} 25n_X + \sum_{m\geq 1} m \cdot \Big( \sum_{\substack{X\in\mathcal{X} \\ \text{left unblocked in } P_m}} n_X + \sum_{\substack{X\in\mathcal{X} \\ \text{right unblocked in } P_m}} n_X \Big). \quad (7)$$

The inequality follows as: if (one occurrence of) $X$ popped $o_X$ letters, then it was not blocked on left/right side for $o_1/o_2$ partitions, where $o_1 + o_2 \geq o_X - 4$ (note that one sequence can be popped to the left and right during block compression but it is immediately replaced with a single letter, so we treat them as one letter, also one letter can be popped to the left/right after $X$ became blocked). Then in right hand side of (7) the contribution from (one occurrence of) $X$ is at least

$$25 + \frac{o_1(o_1+1) + o_2(o_2+1)}{2} \geq \frac{(o_X-4)^2}{4} + \frac{o_X-4}{2} + 25 \geq o_X \log(o_X+1) \ ,$$

where the first inequality follows as $o_1 + o_2 \geq o_X - 4$ and the second can be checked by simple numerical calculation. Lastly, in (7) each $o_i$ is equal to an appropriate $o_X$.

The sum in braces on the right hand side of (7) initially is at most $2|U_0V_0| \leq 2||(U_0, V_0)||$ and by strategy choice (3) it is at least halved every 4th step. So this sum is at most:

$$\sum_{i\geq 0} \underbrace{(16i+10)}_{4 \text{ consecutive steps}} \cdot \underbrace{2||(U_0,V_0)||}_{\text{initial size}} \cdot (1/2)^i = 104||(U_0, V_0)||$$

and consequently

$$\sum_{i\text{: index}} h(o_i) \leq 129||(U_0, V_0)|| \ . \quad (8)$$

The analysis for $e_i$ is similar: for a single index $i$ the estimation of the number of position by which $\mathrm{Pos}_{\supseteq}(i)$ extends is the same as the estimation of number of letters popped from an occurrence of a variable, thus

$$\sum_{i\text{: index}} h(e_i) \leq 129||(U_0, V_0)|| \ . \quad (9)$$

We now estimate, how many positions were lost due to compression, recall that $k_i$ is the size of $\mathrm{Pos}_{\supseteq}(i)$ at the beginning of the phase. Using the same analysis as in the case of $H_d$, from Lemma 2 it follows that at least $\frac{k_i}{3} - 1$ positions were lost in the phase due to compression . Thus

$$H_n(U', V') \leq \sum_{i\text{: index}} h\left(\frac{2}{3}k_i + 1 + o_i + e_i\right). \quad (10)$$

Consider two subcases: if $\frac{2}{3}k_i + 1 + o_i + e_i \leq \frac{5}{6}k_i$, then the summand can be estimated as $h(\frac{5}{6}k_i) \leq \frac{5}{6}h(k_i)$ and we can upper bound the sum over those cases by $\frac{5}{6}\sum_{i\text{: index}} h(k_i)$. If $\frac{2}{3}k_i + 1 + o_i + e_i > \frac{5}{6}k_i$ then $1 + o_i + e_i > \frac{1}{6}k_i$ and so $\frac{2}{3}k_i + 1 + o_i + e_i < 5(1 + o_i + e_i)$. Thus (10) is upper-bounded by:

$$H_n(U', V') \leq \frac{5}{6} \sum_{i\text{: index}} h(k_i) + \sum_{i\text{: index}} h(5(1 + o_i + e_i)) \ .$$

Using simple properties of $h$ as well as (8)–(9) we upper-bound the right hand side by $\frac{5}{6}H_n(U, V) + 15540||(U_0, V_0)||$.

We should estimate the maximal $H_n$ value during the phase, as inside a phase we cannot guarantee that letters get compressed, i.e. estimate $\sum_{i:\text{ index}} h(k_i + o_i + e_i)$. Using similar calculation as in the case of (10) and properties of $h$ we obtain:

$$\sum_{i:\text{ index}} h(k_i + o_i + e_i) \leq 8H_n(U,V) + 2064||(U_0,V_0)|| \ ,$$

which shows the claim of the Lemma in the case of $H_n$ and so also in case of $H$. ◀

## 3.3 Proof of Theorem 3

By Lemma 1 all our subprocedures are sound, so we never accept an unsatisfiable equation.

We now give analyse the nondeterministic choices that yield termination, completeness and linear space consumption. Consider an equation $U = V$ at the beginning of the phase, let $\Gamma$ be the set of letters in this equation. If it has a solution $S'$, then it also has a solution $S$ over $\Gamma$ such that $|S(X)| = |S'(X)|$ for each variable: we can replace letters outside $\Gamma$ with a fixed letter from $\Gamma$. During the phase we will make nondeterministic choices according to this $S$.

Let the equation obtained at the end of the phase be $U' = V'$ and $S'$ be the corresponding solution. Then $|S'(U')| \leq \frac{2|S(U)|+1}{3}$ by Lemma 2 and we begin the next phase with $S'$. Hence we terminate after $\mathcal{O}(\log N)$ phases, where $N$ is the length of some solution of the input equation.

Let the run also choose the partitions according to the strategy. We show by induction that for an equation $(U,V)$ at the beginning of a phase $H(U,V) \leq \delta||(U_0,V_0)||$, where $\delta$ is a constant. Initially $H_n(U_0,V_0) = ||(U_0,V_0)||$ and $H_d(U_0,V_0) = 2||(U_0,V_0)||$, as for each index $\text{dep}(i) = \{i\}$; hence the claim holds. By Lemma 12 the inequality at the end of each phase holds for $\delta = 6\alpha$ for $\alpha$ from Lemma 12. For intermediate equations $H(U,V) \leq (6\alpha\gamma + \beta)||(U_0,V_0)||$, by Lemma 12, where $\beta, \gamma$ are the constants from Lemma 12.

To upper-bound the space consumption, we also estimate other stored information: we also store the alphabet from the beginning of the phase (this is linear in the size of the equation at the beginning of the phase) and the mapping of this alphabet to the current symbols (linear in the equation at the beginning of the phase plus the size of the current equation). The terminating condition that some pair of letters in $\Gamma^2$ was not covered is guessed nondeterministically, we do not store $\Gamma^2$. The pair compression and block compression can be performed in linear space, see Lemma 6. Note that this includes the change of Huffman coding.

## References


1 Volker Diekert, Claudio Gutiérrez, and Christian Hagenah. The existential theory of equations with rational constraints in free groups is PSPACE-complete. *Inf. Comput.*, 202(2):105–140, 2005.

2 Volker Diekert, Artur Jeż, and Manfred Kufleitner. Solutions of word equations over partially commutative structures. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *ICALP*, volume 55 of *LIPIcs*, pages 127:1–127:14. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.127.

3 Volker Diekert, Artur Jeż, and Wojciech Plandowski. Finding all solutions of equations in free groups and monoids with involution. *Inf. Comput.*, 251:263–286, 2016. doi:10.1016/j.ic.2016.09.009.

4 Volker Diekert and Markus Lohrey. Word equations over graph products. *International Journal of Algebra and Computation*, 18(3):493–533, 2008.

5 Claudio Gutiérrez. Satisfiability of word equations with constants is in exponential space. In *FOCS*, pages 112–119. IEEE Computer Society, 1998. doi:10.1109/SFCS.1998.743434.

6 Joxan Jaffar. Minimal and complete word unification. *J. ACM*, 37(1):47–85, 1990.

7 Artur Jeż. Context unification is in PSPACE. In Elias Koutsoupias, Javier Esparza, and Pierre Fraigniaud, editors, *ICALP*, volume 8573 of *LNCS*, pages 244–255. Springer, 2014. doi:10.1007/978-3-662-43951-7_21.

8 Artur Jeż. Recompression: a simple and powerful technique for word equations. *J. ACM*, 63(1):4:1–4:51, Mar 2016. doi:10.1145/2743014.

9 Antoni Kościelski and Leszek Pacholski. Complexity of Makanin's algorithm. *J. ACM*, 43(4):670–684, 1996.

10 Gennadiĭ Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 2(103):147–236, 1977. (in Russian).

11 Gennadiĭ Makanin. Equations in a free group. *Izv. Akad. Nauk SSR*, Ser. Math. 46:1199–1273, 1983. English transl. in Math. USSR Izv. 21 (1983).

12 Yuri Matiyasevich. Some decision problems for traces. In Sergej Adian and Anil Nerode, editors, *LFCS*, volume 1234 of *LNCS*, pages 248–257. Springer, 1997. Invited lecture.

13 Wojciech Plandowski. Satisfiability of word equations with constants is in NEXPTIME. In *STOC*, pages 721–725. ACM, 1999. doi:10.1145/301250.301443.

14 Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004. doi:10.1145/990308.990312.

15 Wojciech Plandowski. An efficient algorithm for solving word equations. In Jon M. Kleinberg, editor, *STOC*, pages 467–476. ACM, 2006. doi:10.1145/1132516.1132584.

16 Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of word equations. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *LNCS*, pages 731–742. Springer, 1998. doi:10.1007/BFb0055097.

17 Alexander A. Razborov. *On Systems of Equations in Free Groups*. PhD thesis, Steklov Institute of Mathematics, 1987. In Russian.

18 Klaus U. Schulz. Makanin's algorithm for word equations – two improvements and a generalization. In Klaus U. Schulz, editor, *IWWERT*, volume 572 of *LNCS*, pages 85–150. Springer, 1990. doi:10.1007/3-540-55124-7_4.