Recompression: a simple and powerful technique for word equations*

Artur Jeż

Max Planck Institute für Informatik,
Campus E1 4, DE-66123 Saarbrücken, Germany
Institute of Computer Science, University of Wrocław,
ul. Joliot-Curie 15, PL-50-383 Wrocław, Poland
aje@cs.uni.wroc.pl

Abstract

We present an application of a local recompression technique, previously developed by the author in the context of compressed membership problems and compressed pattern matching, to word equations. The technique is based on local modification of variables (replacing X by aX or Xa) and replacement of pairs of letters appearing in the equation by a 'fresh' letter, which can be seen as a bottom-up *compression* of the solution of the given word equation, to be more specific, building an SLP (Straight-Line Programme) for the solution of the word equation.

Using this technique we give new self-contained proofs of many known results for word equations: the presented nondeterministic algorithm runs in $\mathcal{O}(n \log n)$ space and in time polynomial in $\log N$ and n, where N is the size of the length-minimal solution of the word equation. It can be easily generalised to a generator of all solutions of the word equation. A further analysis of the algorithm yields a doubly exponential upper bound on the size of the length-minimal solution. The presented algorithm does not use exponential bound on the exponent of periodicity. Conversely, the analysis of the algorithm yields a new proof of the exponential bound on exponent of periodicity. For $\mathcal{O}(1)$ variables with arbitrary many appearances it works in linear space.

1998 ACM Subject Classification F.4.3 Formal Languages: Decision problems, F.4.2 Grammars and Other Rewriting Systems, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Word equations, exponent of periodicity, string unification

Digital Object Identifier 10.4230/LIPIcs.STACS.2013.233

1 Introduction

The problem of *word equations* is one of the most intriguing in computer science: given words U and V, consisting of letters (from Γ) and variables (from $\mathcal{X}$) we are to check the *satisfiability*, i.e. decide, whether there is a substitution for variables that turns this formal equation into an equality of strings of letters. It is useful to think of a solution S as a homomorphism $S : \Gamma \cup \mathcal{X} \to \Gamma^*$, which is an identity on Γ. In the more general problem of *solving* the equation, we are to give representation of all solutions of the equation.

The satisfiability problem was solved by Makanin [10] (a more accessible presentation is due to Diekert [2]). The proposed algorithm MakSAT transforms equations and large part of Makanin's work consists of proving that this procedure in fact terminates. While terminating, MakSAT complexity is high and it was gradually improved: by Jaffar and independently Schulz to 4-NEXPTIME [4, 17] Kościelski and Pacholski to 3-NEXPTIME [7], by Diekert to

* This work was partially supported by NCN grant number DEC-2011/01/D/ST6/07164, 2011–2014.


© Artur Jeż;
licensed under Creative Commons License BY-ND
30th Symposium on Theoretical Aspects of Computer Science (STACS'13).
Editors: Natacha Portier and Thomas Wilke; pp. 233–244

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2-EXPSPACE (unpublished, see notes in [2]) and by Gutiérrez to EXPSPACE [3]. On the other hand, only a (simple) NP-hardness is known and it is widely believed that indeed this problem is in NP.

One of the key factors in the proof of termination, as well in later estimations of the complexity of the algorithm, was the upper bound on *exponent of periodicity* of the solution: the exponent of periodicity of a word w is the largest p such that $w = w_1 u^p w_2$ for some $u \neq \epsilon$. The original proof of Makanin gave a doubly exponential bound on the exponent of periodicity of any length-minimal solution of word equations. Later it was shown by Kościelski and Pacholski that it is at most exponential and that this bound is tight [7].

A major independent step in the field was done by Plandowski and Rytter [15], who for the first time applied the *compression* to the solutions of the word equations. They showed that LZ-compressed representation of length-minimal (of size N) is $\mathcal{O}(\mathsf{poly}(\log N, n))$-size. This yielded a new, very simple to both state and analyse, algorithm for word equations satisfiability, which works in (nondeterministic) polynomial in terms of $\log N$ and n. Unfortunately, at that time the only bound on N followed from the original Makanin's algorithm, and it was triply exponential; this gave a 2-NEXPTIME algorithm.

Later, Plandowski gave a doubly-exponential upper bound on the size of the minimal solution [12], which immediately yielded a NEXPTIME algorithm PlaSAT2EXP. This upper bound was obtained by an analysis of the minimal solution using so-called $\mathcal{D}$-factorisations.

Soon after an algorithm PlaSAT with a PSPACE upper-bound was given by Plandowski [13]. This algorithm starts with a trivial equation $e = e$ and has a set of operations that can be performed on the equation. The rewriting rules are simple and the algorithm is easy to understand, moreover it is obvious that they preserve satisfiability. However, the proof of completeness (i.e. that they properly generate all satisfiable equations) is involved. It is based on usage of exponential expressions, which can be seen as a very simple compression, and on indexed factorisations of words, which extend $\mathcal{D}$-factorisations.

All mentioned algorithms check satisfiability and can be modified to return some solution of the word equation, but they do not *solve* it in the sense that they do not provide a representation of all solutions. This was fully resolved by Plandowski [14], who gave an algorithm PlaSolve, which runs in PSPACEand generates a compact representation of all finite solutions of a word equation. This algorithm uses an improved version of PlaSAT, the PlaSATimp, as subprocedure. The representation of the solutions is a directed multigraph, whose nodes are labelled with expressions and edges define substitutions for constants and variables. Such representation reduces many properties of word equations to reachability in graphs (which were exponentially larger), for instance the problem of finiteness of set of solutions is shown to be in PSPACE.

Our contribution

We present an application of a simple technique of *local recompression* developed by the author and successfully applied to problems related with compressed data [5, 6]. The idea of the technique is easily explained in terms of solutions of the equations rather than the equations themselves: consider a solution $S(U) = S(V)$ of the equation $U = V$. In one phase we first list all pairs of different letters ab that appear as substrings in $S(U)$ and $S(V)$. For a fixed pair ab of this kind we greedily replace all appearances of ab in $S(U)$ and $S(V)$ by a 'fresh' letter c. (A slightly more complicated action is performed for pairs aa, for now we ignore this case to streamline the presentation of the main idea). There are possible conflicts between such replacements for different types of pairs (consider string aba, in which we try to replace both pairs ab and ba), we resolve them by introducing some arbitrary order on

types of pairs and performing the replacement for one type of pair at a time, according to the order. When all such pairs are replaced, we obtain another equal pair of strings $S'(U')$ and $S'(V')$ (note that the equation $U = V$ may have changed, say into $U' = V'$). Then we iterate the process. In each phase the strings are shortened by a constant factor, hence after $\mathcal{O}(\log N)$ phases we obtain constant-length strings. The original equation is solvable if and only if the obtained strings are the same.

The problematic part is that the operations are performed on the solutions, which can be large. If we simply guess the solution and then perform the compressions, the running time is polynomial in N. Instead we perform the compression directly on the equation (the *recompression*): the pairs ab appearing in the solution are identified using only the equation and the compression of the solution is two-fold: the pairs ab from U and V are replaced explicitly and the pairs fully within some $S(X)$ are replaced implicitly, by changing S (which is not stored). However, not all pairs of letters can be compressed in this way, as some of them appear on the 'crossing' between a variable and a constant: consider for instance $S(X) = ab$, a string of symbols Xc and a compression of a pair bc. This is resolved by *local decompression*: when trying to compress the pair bc in the example above we first replace X by Xb (implicitly changing $S(X)$ from ab to a), obtaining the string of symbols Xbc, in which the pair bc can be easily compressed.

▶ **Example 1.** Consider an equation $aXca = abYa$ with a solution $S(X) = baba$ and $S(Y) = abac$. In the first phase, the algorithm wants to compress the pairs ab, ca, ac, ba, say in this order. To compress ab, it replaces X with bX, thus changing the substitution into $S(X) = aba$. After compression we obtain equation $a'Xca = a'Ya$. Notice, that this implicitly changed to solution into $S(X) = a'a$ and $S(Y) = a'ac$. To compress ca (into c'), we replace Y by Yc, thus implicitly changing the substitution into $S(Y) = a'a$. Then, we obtain the equation $a'Xc' = a'Yc'$ with a solution $S(X) = a'a$ and $S(Y) = a'a$. The remaining pairs no longer appear in the equations, and so we can proceed to the next phase.

Using the technique of local recompression we give a (nondeterministic) algorithm for testing satisfiability of word equations that works in time $\mathcal{O}(\log N\mathsf{poly}(n))$ and in $\mathcal{O}(n \log n)$ space; PlaSAT stored equation of quadratic length and used cubic space for auxiliary computation. Furthermore, for $\mathcal{O}(1)$ variables a more careful analysis yields that the space consumption (calculated in bits) is $\mathcal{O}(n)$, thus showing that this case is context-sensitive.

The presented algorithm and its analysis are stand-alone, as they do not assume any (non-trivial) properties of the solutions of word equations. To the contrary, the presented algorithm supplies an easy proof of doubly-exponential upper bound of Plandowski [12] on lengths of length-minimal solutions as well as giving a new proof of exponential exponent of periodicity (though slightly weaker than the one presented by Kościelski and Pacholski [7]).

The presented method can be used as a subprocedure in an algorithm generating a representation of all solutions, similarly as PlaSATimp in PlaSolve. The representation provided by our algorithm is similar to representation provided by PlaSolve, i.e. a directed multigraph with edges representing substitutions.

Comparison with previous approaches to word equations

The presented method and the obtained algorithm is independent from all previously known algorithms for word equations, i.e. from original MakSAT and its variants, from PlaRytSAT (and its variant PlaSAT2EXP), from PSPACE algorithm PlaSAT as well as its modification PlaSATimp. It can be somehow compared with the LZ-based PlaRytSAT [15]. The key difference is that Plandowski and Rytter showed that a length-minimal solution has a short

LZ-representation and then explicitly guessed and verified it. Thus their solution is 'global' and based on solutions' properties. The novelty of the here proposed method is that it does not use properties of the solutions and that it is very 'local', as it does not try to build the solution in one go and modifies the equations and variables locally.

The presented algorithm uses a limited variant of exponent of periodicity, in which the strings in question are repetitions of the same letter. In such a case the bound on such restricted exponent of periodicity easily reduces to a bound on minimal solutions of a system of linear Diophantine equations, which are well known. This again makes the presented algorithm somehow similar to PlaRytSAT, which does not use the exponent of periodicity bound.

Related techniques

While the presented method of recompression is relatively new, some of its ideas and inspirations go quite back. It was developed in order to deal with fully compressed membership problem for NFA and the previous work on this topic by Mathissen and Lohrey [9] already implemented the idea of replacing strings with fresh letters as well as modifications of the instance so that this is possible. Furthermore they treated maximal blocks of a single letter in a proper way. However, the replacement was not iterated, and the newly introduced letters could not be further compressed.

The idea of replacing short strings by a fresh letter and iterating this procedure was used by Mehlhorn et. al [11] in their work on data structure for equality tests for dynamic strings (cf. also an improved implementation of a similar data structure by Alstrup, Brodal and Rauhe [1]). In particular their method can be straightforwardly applied to equality testing for SLPs, yielding a nearly quadratic algorithm (as observed by Gawrychowski). However, the inside technical details of the construction make it problematic to extend: while this method can be used to build 'canonical' SLPs for the text and the pattern, there is no apparent way to control how these SLPs look like and how do they encode the strings.

A similar technique, based on replacement of pairs and blocks of the same letter was proposed by Sakamoto [16] in the context of constructing a smallest SLP generating a given word. His algorithm was inspired by a practical grammar-based compression algorithm RePair [8]. It possessed the important features of the method: iterated replacement, and ignoring letters recently introduced. However, the analysis that stressed the modification of the variables (nonterminals, in there considered case of grammars) was not introduced and it was done in a different way.

2 Main notions and techniques: local compression

By Γ we denote the set of letters appearing in the equation $U = V$ or are used for representation of compressed strings, $\mathcal{X}$ denotes a set of variables. The equation is written as $U = V$, where $U, V \in (\Gamma \cup \mathcal{X})^*$. By $|U|$, $|V|$ we denote the length of U and V, n denotes the length of the input equation, n_v is the number of appearances of variables in the input.

A *substitution* is a morphism $S : \mathcal{X} \cup \Gamma \to \Gamma^*$, such that $S(a) = a$ for every $a \in \Gamma$, substitution is naturally extended to $(\mathcal{X} \cup \Gamma)^*$. A *solution* of an equation $U = V$ is a substitution S, such that $S(U) = S(V)$; a solution S is a *length-minimal*, if for every solution S' it holds that $|S(U)| \leq |S'(U)|$.

Operations

In essence, the presented technique is based on performing two operations on $S(U)$ and $S(V)$, the first one is *pair compression of ab*: For two different letters ab appearing in $S(U)$ replace each of ab in $S(U)$ and $S(V)$ by a fresh letter c.

The compression of pair aa is ambiguous, we compress maximal blocks instead: For a letter $a \in \Gamma$ we say that a^ℓ is a a's *maximal block* of length ℓ (for a solution S), if a^ℓ appears in $S(U)$ and this appearance cannot be extended by a letter a to the right, neither to the left. We refer to *a's ℓ-block* for shortness. The second operation is *block compression for a*: For a letter a appearing in $S(U)$ and for each maximal block a^ℓ replace all a^ℓs in $S(U)$ and $S(V)$ by a fresh letter a_ℓ.

The lengths of the maximal blocks can be upper bounded using the well-known exponential bound on exponent of periodicity:

▶ **Lemma 2** (Exponent of periodicity bound [7]). *If solution S is length-minimal and w^ℓ for $w \neq \epsilon$ is a substring of $S(U)$, then $\ell \leq 2^{cn}$ for some constant $0 < c < 2$.*

▶ Remark. WordEqSat introduces new letters to the instance, replacing pairs of letters or maximal blocks of one letter. We insist that these new symbols are called and treated as letters. On the other hand, we can think of them as non-terminals of a context-free grammar: if c replaced ab, then this corresponds to a production $c \to ab$, similarly, $a_\ell \to a^\ell$. In this way we can think that WordEqSat builds a context-free grammar generating $S(U)$ as a unique word in the language.

Types of pairs and blocks

Both pair compression and block compression shorten $S(U)$ (and $S(V)$). On the other hand, sometimes it is hard to perform these operations: for instance, if we are to compress a pair ab and aX appears in U, moreover, $S(X)$ begins with b, then the compression is problematic, as we need to somehow modify $S(X)$. The following definition allows distinguishing between pairs (blocks) that are easy to compress and those that are not.

▶ **Definition 3** (cf. [5, 6]). Given an equation $U = V$ and a substitution S and a substring $u \in \Gamma^+$ of $S(U)$ (or $S(V)$) we say that this appearance of u is *explicit*, if it comes from substring u of U (or V, respectively); *implicit*, if it comes (wholly) from $S(X)$ for some variable X; *crossing* otherwise. A string u is *crossing* (with respect to S) if it has a crossing appearance and non-crossing otherwise.

We say that a pair of ab is a *crossing pair* (with respect to S), if ab has a crossing appearance. Otherwise, a pair is *non-crossing* (with respect to S). Unless explicitly stated, we consider crossing/non-crossing pairs ab in which $a \neq b$. Similarly, a letter $a \in \Gamma$ has a *crossing block* (with respect to S), if there is a block of a which has a crossing appearance.

Compression of noncrossing pairs is easy, so is block compression when a has no crossing block. In other cases, the compression seems difficult.

We say that a^ℓ is *visible* in S, if there is an appearance of the a's ℓ-block that is explicit or crossing or it is a prefix or suffix of some $S(X)$.

▶ **Lemma 4** (cf. [15, Lemma 6]). *Let S be a length-minimal solution of $U = V$.*

- *If ab is a substring of $S(U)$, where $a \neq b$, then ab is an explicit pair or a crossing pair.*
- *If a^k is a maximal block in $S(U)$ then a has an explicit appearance in U or V and there is a visible appearance of a^k.*

Compression of noncrossing pairs and blocks

Intuitively, when ab is non-crossing, each of its appearance in $S(U)$ is either explicit or implicit. Thus, to perform the pair compression of ab on $S(U)$ it is enough to separately replace each explicit pair ab in U and change each ab in $S(X)$ for each variable X. The latter is of course done implicitly (as $S(X)$ is not written down anywhere).

Algorithm 1 PairCompNCr(a, b) Pair compression for a non-crossing pair

1: let $c \in \Gamma$ be an unused letter
2: replace each explicit ab in U and V by c

Similarly when none block of a has a crossing appearance, the a's blocks compression consists simply of replacing explicit a blocks.

Algorithm 2 BlockCompNCr(a) Block compression for a letter a with no crossing block

1: **for** each explicit a's ℓ-block appearing in U or V **do**
2: let $a_\ell \in \Gamma$ be an unused letter
3: replace every explicit a's ℓ-block appearing in U or V by a_ℓ

Preserving satisfiability and unsatisfiability

We say that a nondeterministic procedure *preserves unsatisfiability*, when given a unsatisfiable word equation $U = V$ it cannot transform it to a satisfiable one, regardless of the nondeterministic choices; such a procedure *preserves satisfiability*, if given a satisfiable equation $U = V$ for some nondeterministic choices it returns a satisfiable equation $U' = V'$.

A procedure that preserves satisfiability *implements pair compression of a pair ab* for a solution S of an equation $U = V$ for some nondeterministic choices it returns equation $U' = V'$ with a solution S', such that $S'(U')$ is obtained from $S(U)$ by replacing each ab by c; similarly we say that a procedure implements blocks compression for a.

▶ **Lemma 5.** PairCompNCr(a, b) *preserves the unsatisfiability; for each solution S, if ab is a non-crossing pair for S in an equation $U = V$ then it preserves satisfiability and implements the pair compression of ab, .*

BlockCompNCr(a) *preserves unsatisfiability; for each solution S, if a has no crossing blocks in $U = V$ (for S) it preserves satisfiability and implements a's block compression.*

Crossing pairs and blocks compression

The presented algorithms cannot be directly applied to crossing pairs or to compression of a's blocks that have crossing appearances. To fix this, we modify the instance: if a pair ab is crossing because there is a variable X such that $S(X) = bw$ for some word w and a is to the left of X, it is enough to *left-pop* b from $S(X)$: we replace each X with bX and implicitly change S, so that $S(X) = w$; similar action is applied to variables Y ending with a and with b to the right (*right-popping* a from $S(X)$). Afterwards, ab is non-crossing with respect to S.

This idea can be employed much more efficiently: consider a partition of Γ into Γ_ℓ and Γ_r. The 'left-popping' from each variable a letter from Γ_r and 'right-popping' a letter from Γ_ℓ guarantees that each pair $ab \in \Gamma_\ell\Gamma_r$ is non-crossing. Since pairs from $\Gamma_\ell\Gamma_r$ do not overlap, after the preprocessing they can be compressed in parallel.

Algorithm 3 $\mathsf{Pop}(\Gamma_\ell, \Gamma_r)$

for $X \in \mathcal{X}$ **do**
  let b be the first letter of $S(X)$ ▷ Guess
  if $b \in \Gamma_r$ **then**
    replace each X in U and V by bX ▷ Implicitly change $S(X) = bw$ to $S(X) = w$
    if $S(X) = \epsilon$ **then** ▷ Guess
      remove X from the equation
▷ Perform a symmetric action for the last letter

▶ **Lemma 6.** $\mathsf{Pop}(\Gamma_\ell, \Gamma_r)$ *preserves satisfiability and unsatisfiability.*

Furthermore, if S is a solution of $U = V$ then for all nondeterministic choices the obtained $U' = V'$ has a solution S' such that $S'(U') = S(U)$ and for some nondeterministic choices each pair ab from $\Gamma_\ell\Gamma_r$ is non-crossing (with regard to S').

Algorithm 4 $\mathsf{PairComp}(\Gamma_\ell, \Gamma_r)$

run $\mathsf{Pop}(\Gamma_\ell, \Gamma_r)$
for $ab \in \Gamma_\ell\Gamma_r$ **do**
  run $\mathsf{PairCompNCr}(a, b)$

▶ **Lemma 7.** $\mathsf{PairComp}(\Gamma_\ell, \Gamma_r)$ *preserves satisfiability and unsatisfiability and for each solution it implements the pair compression of each pair $ab \in \Gamma_\ell\Gamma_r$.*

The problems with crossing blocks can be solved in a similar fashion: a has a crossing block, if aa is a crossing pair. So we 'left-pop' a from X until the first letter of $S(X)$ is different than a, we do the same with the ending letter b. This can be alternatively seen as removing the whole a-prefix (b-suffix, respectively) from X: suppose that $S(X) = a^\ell w b^r$, where w does not start with a nor end with b. Then we replace each X by $a^\ell X b^r$ implicitly changing the solution to $S(X) = w$. This eliminates crossing blocks with respect to S.

Algorithm 5 $\mathsf{CutPrefSuff}$ Cutting prefixes and suffixes

for $X \in \mathcal{X}$ **do**
  let a, b be the first and last letter of $S(X)$
  guess $\ell_X \geq 1$, $r_X \geq 0$ ▷ $S(X) = a^{\ell_X} w b^{r_X}$, w does not begin with a nor end with b
  replace each X in U and V by $a^{\ell_X} X b^{r_X}$ ▷ a^{ℓ_X}, b^{r_X} is stored in a compressed form
▷ implicitly change $S(X) = a^{\ell_X} w b^{r_X}$ to $S(X) = w$
  if $S(X) = \epsilon$ **then** ▷ Guess
    remove X from the equation

▶ **Lemma 8.** $\mathsf{CutPrefSuff}$ *preserves unsatisfiability and satisfiability. For a solution S of $U = V$ and appropriate nondeterministic choices it returns an equation $U' = V'$ that has a solution S' such that $S(U) = S'(U')$ and $U' = V'$ has no crossing blocks with respect to S'.*

The procedure $\mathsf{CutPrefSuff}$ allows defining a procedure $\mathsf{BlockComp}$ that compresses maximal blocks of all letters, regardless of whether they have crossing blocks or not.

Algorithm 6 BlockComp

```
1: run CutPrefSuff
2: for each letter a ∈ Γ do
3:     BlockCompNCr(a)
```

▶ **Lemma 9.** BlockComp *preserves unsatisfiability and satisfiability and implements the block compression (for all letters a).*

3 Main algorithm, its time and space consumption, solutions' size

Now, the algorithm for testing satisfiability of word equations can be conveniently stated. We refer to one iteration of the main loop in WordEqSat as one *phase*.

Algorithm 7 WordEqSat Checking the satisfiability of a word equation

```
1: while |U| > 1 or |V| > 1 do
2:     run BlockComp
3:     L ← the set of letters present in U or V
4:     for i ← 1..2 do                    ▷ One to compress the equation, one the solution
5:         guess partition of L into L1 and L2, run PairComp(L1, L2)
6: Solve the problem naively              ▷ With sides of length 1, the problem is trivial
```

The somehow peculiar double iteration of line 5 is for technical reasons: in one of the iterations we make sure that the solution is compressed, in the other that the equation representation is compressed, see Lemma 11.

▶ **Theorem 10.** WordEqSat *nondeterministically verifies the satisfiability of word equations. It can verify an existence of a length-minimal solution of length N in $\mathcal{O}(\mathsf{poly}(n)\log N)$ time and $\mathcal{O}(n \log n)$ space. For appropriate choices, the stored equation has length $\mathcal{O}(n)$.*

The correctness of WordEqSat follows from the fact that its subprocedures preserve satisfiability and unsatisfiability, which was shown in the previous section.

The lemma below formally states the idea that the solutions are compressed and it is a technical foundation for proofs of space and time consumption bounds.

▶ **Lemma 11.** *Let $U = V$ has a solution S. For appropriate nondeterministic choices the equation $U' = V'$ obtained at the end of the phase has a solution S' such that i) at least $1/6$ of letters in U or V are compressed in U' or V'; ii) at least $1/6$ of letters in $S(U)$ are compressed in $S'(U')$.*

Proof. We show the claim for $S(U)$ and then comment how the proof applies to U. Divide $S(U)$ into three-letter segments. We prove that for some partition into L_1 and L_2, in at least halve of these segments one of the letters in them is compressed, which shows the claim. Consider any such segment, let it be abc. If any of those letters is the same as its neighbouring letters, then by Lemma 9 for some nondeterministic choices BlockComp implements the compression of blocks, and so this letter is compressed and we are done.

So suppose that none of these letters is the same as its neighbouring letters, in particular, they are not compressed by BlockComp. Consider a random partition of L into L_1 and L_2, each letter goes to one part with probability $1/2$. There is a compression inside abc if $ab \in L_1L_2$ or $bc \in L_1L_2$. Each of those events has probability $1/4$ and they are disjoint,

hence the compression appears with probability at least $1/2$. So regardless of the case, with probability $1/2$ at least one of letters in abc is compressed. There are $|S(U)|/3$ three-letter segments. The expected number of segments in which at least one letter is compressed is thus at least $|S(U)|/6$, so for some partition at least $|S(U)|/6$ letters are compressed.

Concerning U, the analysis is similar: we consider segments in U and V instead of $S(U)$ and $S(V)$. This corresponds to the second partition of L to L_1 and L_2. ◀

Lemma 11 is enough to show the bound on used memory: on one hand the new letters are introduced to the equations by Pop and CutPrefSuff, and their total number is $\mathcal{O}(n_v)$ per phase. This increases the lengths of U and V by $\mathcal{O}(n_v)$. On the other hand U and V are shortened by a constant factor; this yields a linear bound on $|U'|$ and $|V'|$. Moreover, Lemma 11 yields that for some choices there are $\mathcal{O}(\log N)$ phases.

4 Other results

4.1 Theoretical properties

Using the approach of recompression we give (alternative and often simpler) proofs of a doubly-exponential bound on the size of the length-minimal solution and an exponential bound on the periodicity bound.

4.1.1 Double exponential bound on minimal solutions

The running time of WordEqSat is polynomial in n and $\log N$ and it is easy to also *lower-bound* it in terms of $\log N$. On the other hand the length of the stored equations is $\mathcal{O}(n)$, which yields that there are exponentially (in n) many different configurations. Comparing those two bounds yields a doubly exponential bound on N.

4.1.2 Exponential bound on exponent of periodicity

For a word w the *exponent of periodicity* $\mathrm{per}(w)$ is the maximal k such that u^k is a substring of w, for some $u \in \Gamma^+$; Γ-*exponent of periodicity* $\mathrm{per}_\Gamma(w)$ restricts the choice of u to Γ. This notion is naturally transferred to equations: For an equation $U = V$, define the exponent of periodicity as $\max_S [\mathrm{per}(S(U))]$, where the maximum is taken over all length-minimal solutions S of $U = V$; define the Γ-*exponent of periodicity* of $U = V$ in a similar way.

An exponential upper bound on Γ-exponent of periodicity of an equation is easy to obtain. Fix a solution S, for each variable X define ℓ_X (r_X): the length of maximal prefix (suffix, respectively) of $S(X)$ that is a block of the same letter. From Lemma 4 it follows that each length of maximal block can be expressed in terms of $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ and constants. We define a different solution S' which is obtained by altering $\{\ell_X, r_X\}_{X\in\mathcal{X}}$. To guarantee that such a S' is indeed a solution we require that if two maximal blocks in $S(U)$ are of the same length, they have the same length in $S'(U)$ as well. Such a condition boils down to an equality of two expressions using $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ and constants. When we treat $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ as variables, we obtain a system of linear Diophantine equations. Each solution of this system defines a solution of $U = V$, in particular, length-minimal solutions come from minimal (in an appropriate sense) solutions of such Diophantine systems, which are known to be at most exponential.

To show a similar bound in the case of exponent of periodicity, we investigate, how it can be changed by subprocedures of WordEqSat. On one hand, if the exponent of periodicity is equal to Γ-exponent of periodicity then it is at most exponential. We show that when it is

not then each subprocedure of WordEqSat can modify it at most by a constant. Hence, the upper bound on the exponent of periodicity is at most the sum of number of subprocedures and the Γ-exponent of periodicity, which are both at most exponential.

4.2 Linear space for $\mathcal{O}(1)$ variables

The length of the word equation kept by WordEqSat is linear. However, the letters in this equation can be all different, even if the input equation is over two letters. Hence the (nondeterministic) space usage is $\mathcal{O}(n \log n)$ bits. For $\mathcal{O}(1)$ variables (and unbounded number of appearances) we improve the bit consumption to only constant larger than the input.

The main obstacle is the encoding of letters introduced by WordEqSat, we give a more suitable one. Consider string of explicit letters between two consecutive variables X and Y in $U = V$. During WordEqSat the XwY will be changed to $Xw^{(1)}Y$, $Xw^{(2)}Y$, ... Each $w^{(i)}$ can be partitioned into 3 substrings $x^{(i)}v^{(i)}y^{(i)}$, where the letters in $v^{(i)}$ represent solely the letters from w, while each letter in $x^{(i)}$ ($y^{(i)}$) represent also some letter popped at some point from X (Y, respectively). We encode $v^{(i)}$ using only a constant time more bits than w: roughly, we represent letters as trees and when merging a and b into c, the tree of c has the tree of a as a left subtree and a tree of b as a right subtree.

On the other hand, the letters in $x^{(i)}$ and $y^{(i)}$ depend solely on XwY, so we simply encode them as $(XwY)1$, $(XwY)2$, ..., $(XwY)(|x^{(i)}| + |y^{(i)}|)$, where (XwY) is encoded as the corresponding fragment of the input and the numbers are encoded in binary. Note, that in this way different appearances of the same letter a may get the same code: in such case we collect the codes for a and add the information that they all represent the same letter.

It might be that $|x^{(i)}| + |y^{(i)}|$ is non-constant: WordEqSat guarantees that the length of the whole $|U| + |V|$ is $\mathcal{O}(n)$, but some fragments may become large. However, for $\mathcal{O}(1)$ variables we can enforce that in one phase WordEqSat compresses *each* pair of consecutive letters. In this way a stronger variant of Lemma 11 yields that each of $|x^{(i)}| + |y^{(i)}|$ is $\mathcal{O}(1)$. Let LinWordEqSat denote the such modified WordEqSat.

▶ **Theorem 12.** LinWordEqSat *preserves unsatisfiability and satisfiability. For k variables, it runs in (nondeterministic) $\mathcal{O}(mk^{ck})$ space, for some constant c, where m is the size of the input measured in bits.*

4.3 Representation of all solutions

Plandowski [14] gave an algorithm that generated a finite, graph-like representation of all solutions of a word equations. It is based on the idea that PlaSAT not only preserves satisfiability and unsatisfiability, but it in some sense operates on the solutions: when it transforms $U = V$ to $U' = V'$ then solutions of $U' = V'$ correspond to solutions of $U = V$. Moreover, each solution of $U = V$ can be represented in this way for some $U' = V'$. Hence, all solutions can be represented as a graph as follows: nodes are labelled with equations of the form $U = V$ and a directed edge leads from $U = V$ to $U' = V'$ if for some nondeterministic choices the former equation is transformed into the latter by PlaSAT. Furthermore, the edge describes, how the solutions of $U' = V'$ can be changed into the solutions of $U = V$. In this process PlaSAT is used to establish the existence of the edge and the operations that are associated with this edge. Since WordEqSat runs in PSPACE, such generation of labelled vertices and edges can also be performed in PSPACE, furthermore the description of a vertex (edge) is of polynomial size, so iteration over all vertices (edges, respectively) can be implemented in PSPACE. We describe how to employ WordEqSat in a similar procedure.

▶ **Theorem 13** (cf. [14]). *The graph representation of all solutions of an equation $U = V$ can be constructed in* PSPACE. *The size of the constructed graph is at most exponential.*

Transforming the solutions

As a first step we define precisely what does it mean that a subprocedure transforms the solutions. We use a notion of an *operator*, which is simply a function taking and returning a substitution. Then given a (nondeterministic) procedure transforming the equation $U = V$ we say that this procedure *transforms the solutions*, if based on the nondeterministic choices and the input equation we can define a family of operators $\mathcal{H}$ such that

- for any solution S of $U = V$ there are some nondeterministic choices that lead to an equation $U' = V'$ such that $S = H[S']$ for some solution S' of the equation $U' = V'$ and some operator $H \in \mathcal{H}$;
- for every solution S' of the obtained equation $U' = V'$ and for every operator $H \in \mathcal{H}$ the $H[S']$ is a solution of $U = V$.

Intuitively, in this way all solutions of an equation $U = V$ can be represented using the solutions of the possible outcome equations. We say that that $\mathcal{H}$ is the *corresponding family of inverse operators*. In many cases, $\mathcal{H} = \{H\}$, in such case we call H the *corresponding inverse operator*.

The lemmata showing that subprocedures of WordEqSat preserve satisfiability can be strengthened to show that they in fact transform the solutions and the appropriate family of inverse operators can be given.

Representation of solutions

The set of solutions will be represented by a directed graph $\mathcal{G}$: the vertices of $\mathcal{G}$ are labelled with equations $U = V$ that appear during the run of WordEqSat; furthermore, we require that the length of $U = V$ is at most cn and it has at most n_v appearances of variables, for some constant c. There is an edge from $U = V$ to $U' = V'$ if and only if for some nondeterministic choices WordEqSat transforms $U = V$ to $U' = V'$; such an edge is labelled with the corresponding family of inverse operators, which can be in one of the following forms: H replaces c in each $S(X)$ by ab; H appends (prepends) a^{ℓ_X} to $S(X)$; for each k replace each a_k in all $S(X)$ by a^k. Note that in the last type of operators, the a_k is just a naming convention, a priori we do not know, which letter is going to be replaced. So, the operators in $\mathcal{H}$ need to specify, which letters in $U' = V'$ are being replaced with a blocks and the lengths of the respective blocks. As such lengths are potentially unbounded, $\mathcal{H}$ can be infinite. However, using approach similar to the one presented in Section 4.1.2, the respective lengths can be represented compactly by a system of linear Diophantine equations, and so the description remains polynomial.

In order to generate the graph representation we need to be able to decide whether: a given equation $U = V$ labels a node in $\mathcal{G}$; given two equations $U = V$ and $U' = V'$ and $\mathcal{H}$ whether there is an edge from $U = V$ to $U' = V'$ labelled with $\mathcal{H}$. WordEqSat can be naturally used to answer such queries. Thus the construction of $\mathcal{G}$ is easy: We iterate over all equations $U = V$ of length at most cn, for a fixed $U = V$ we check whether it labels a node in $\mathcal{G}$. If so, we output it. We then perform a similar iteration for edges: for each pair of equations $U = V$ and $U' = V'$ and family of inverse operators $\mathcal{H}$ we verify, whether there is an edge from $U = V$ to $U' = V'$ labelled with $\mathcal{H}$. If so, we output the appropriate edge.

Acknowledgements I would like to thank P. Gawrychowski for initiating my interest in compressed membership problems, which eventually led to this work and for pointing to relevant literature [9, 11]; J. Karhumäki, for his question, whether the techniques of local recompression can be applied to the word equations; W. Plandowski for his comments and suggestions on the manuscript and questions concerning the space consumption that led to linear space algorithm for $\mathcal{O}(1)$ variables; anonymous referees, who pointed out several shortcomings and mistakes.

References

1 Stephen Alstrup, Gerth Stolting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *SODA*, pages 819–828, 2000.
2 Volker Diekert. Makanin's algorithm. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, chapter 12, pages 342–390. Cambridge University Press, 2002.
3 Claudio Gutiérrez. Satisfiability of word equations with constants is in exponential space. In *FOCS*, pages 112–119. IEEE Computer Society, 1998.
4 Joxan Jaffar. Minimal and complete word unification. *J. ACM*, 37(1):47–85, 1990.
5 Artur Jeż. Compressed membership for NFA (DFA) with compressed labels is in NP (P). In Christoph Dürr and Thomas Wilke, editors, *STACS*, volume 14 of *LIPIcs*, pages 136–147. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
6 Artur Jeż. Faster fully compressed pattern matching by recompression. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *ICALP*, volume 7391 of *LNCS*, pages 533–544. Springer, 2012.
7 Antoni Kościelski and Leszek Pacholski. Complexity of Makanin's algorithm. *J. ACM*, 43(4):670–684, 1996.
8 N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Data Compression Conference*, pages 296–305. IEEE Computer Society, 1999.
9 Markus Lohrey and Christian Mathissen. Compressed membership in automata with compressed labels. In Alexander S. Kulikov and Nikolay K. Vereshchagin, editors, *CSR*, volume 6651 of *LNCS*, pages 275–288. Springer, 2011.
10 G. S. Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 2(103):147–236, 1977. (in Russian).
11 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.
12 Wojciech Plandowski. Satisfiability of word equations with constants is in NEXPTIME. In *STOC*, pages 721–725, 1999.
13 Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004.
14 Wojciech Plandowski. An efficient algorithm for solving word equations. In Jon M. Kleinberg, editor, *STOC*, pages 467–476. ACM, 2006.
15 Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of words equations. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *LNCS*, pages 731–742. Springer, 1998.
16 Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005.
17 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.