Recompression: a simple and powerful technique for word equations

Artur Jeż, Institute of Computer Science, University of Wrocław, ul. Joliot-Curie 15, 50-383 Wrocław, Poland
aje@cs.uni.wroc.pl

In this paper we present an application of a simple technique of local recompression, previously developed by the author in the context algorithms for compressed strings [Jeż 2014a; Jeż 2015; Jeż 2013], to word equations. The technique is based on local modification of variables (replacing X by aX or Xa) and iterative replacement of pairs of letters occurring 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 a new, independent and self-contained proofs of many known results for word equations. To be more specific, the presented (nondeterministic) algorithm runs in $\mathcal{O}(n \log n)$ space and in time polynomial in n and $\log N$, where n is the size of the input equation and N the size of the length-minimal solution of the word equation. Furthermore, for an $\mathcal{O}(1)$ variables the bound on the space consumption is in fact linear, i.e. $\mathcal{O}(m)$ where m is the size of the space used by the input. This yields that for each k the set of satisfiable word equations with k variables is context-sensitive. The presented algorithm can be easily generalised to a generator of all solutions of the given word equation (without increasing the space usage). Furthermore, a further analysis of the algorithm yields an independent proof of 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 an independent proof of the exponential bound on exponent of periodicity.

Categories and Subject Descriptors: F.4.3 [**Formal Languages**]: Decision problems; F.4.2 [**Grammars and Other Rewriting Systems**]: Decision problems; F.2.2 [**Nonnumerical Algorithms and Problems**]

Additional Key Words and Phrases: Word equations, exponent of periodicity, semantic unification, string unification

1. INTRODUCTION

Word equations

Since the dawn of the computer science, the problem of *word equations* was one of the most intriguing on the intersection between algebra and formal languages: 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, which turns this formal equation into an equality of strings of letters. It is useful to think of a solution S as a homomorphism $S : \Gamma \cup \mathcal{X} \mapsto \Gamma^*$, which is an identity on Γ. In the more general problem of *solving* the equation, we are to give representation of (all or some) solutions of the equation.

The problem of satisfiability of word equations was first fully solved by Makanin [1977]. 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 very high. Over the years the algorithm was gradually improved: by Jaffar [1990] and independently Schulz [1990] to 4-NEXPTIME an by Kościelski and Pacholski [1996] to 3-NEXPTIME, by Diekert to 2-EXPSPACE (unpublished) and by Gutiérrez [1998] to EXPSPACE. It is worth mentioning that for 20 years no essentially different algorithm than MakSAT was proposed. On the other hand, as for today only a simple NP lower bound is known and it is widely believed that 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 estimation the upper bound on *exponent of periodicity* of the solution. Roughly speaking, 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 1996] that exponent of periodicity is at most exponential, this bound is tight.

A major independent step in the field was done by Plandowski and Rytter [1998], who for the first time applied the notion of the *compression* to the solutions of the word equations: they have shown that each length-minimal solution of the word equation is highly compressible, in the sense that using LZ77 compression (a popular practical standard for compression) we can represent each length-minimal solution of word equations (of size N) using an $\mathcal{O}(\log N)$-size encoding. This implies that also LZ77-encoding of values of variables in such a solution has size $\mathcal{O}(\log N)$. Thus, to solve the word equation it is enough to guess the LZ77-encoding of $S(X)$ for each variable X and verify that $S(U) = S(V)$ under this substitution. The latter can be done using known (though recent at that time) polynomial methods for testing the equivalence of two SLPs [Plandowski 1994], which extends to LZ77, as LZ77 encoding can be converted to at most quadratically larger SLP. This yielded a new algorithm for word equations satisfiability, which works in (nondeterministic) polynomial time 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 four times exponential. This gave a 3-NEXPTIME algorithm. In the same year a triply exponential upper bound was given [Gutiérrez 1998], which yielded a 2-NEXPTIME algorithm, but this very paper also gave an EXPSPACE algorithm for word equations.

Later, Plandowski [1999] gave a doubly-exponential upper bound on the size of the minimal solution, which immediately yielded a NEXPTIME algorithm PlaSat2EXP for the problem. This upper bound was obtained by a clever and careful analysis of the minimal solution using so-called $\mathcal{D}$-factorisations, suggested by Mignosi.

Soon after, another algorithm PlaSat, with a PSPACE[1] upper-bound was given by Plandowski [2004]. This algorithm starts with a trivial equation $e = e$ and has a set of operations that can be performed on the equation; so it can be seen as a rewriting system. The set of rewriting rules is quite simple and thus also the algorithm is easy to understand, moreover it is obvious that the rewriting rules are sound (i.e. preserve satisfiability). However the proof of completeness of this rewriting system (i.e. that it properly generates all satisfiable equations) is involved. It was based on usage of exponential expressions, which can be seen as a very simple compression, and on indexed factorisations of words, which extend the already mentioned $\mathcal{D}$-factorisations.

In some sense one can think that this result was obtained in stages, as PlaRytSAT, fuelled with theoretical results on $\mathcal{D}$-factorisations, yielded PlaSat2EXP and this in in turn was upgraded to PlaSat, by exploiting better the interplay between the compression and factorisations.

All mentioned algorithms have a little drawback: while they check satisfiability and can be modified to return *some* solution of the word equation, they do not *solve* it in the sense that they do not provide a representation of all solutions. This was fully resolved by Plandowski [2006], who gave an algorithm PlaSolve, which runs in PSPACE[2] and 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. It is still unknown, whether MakSAT can be used in a similar construction.

[1]The presented algorithm has running time proportional to N, however, it can be extended so that it has the same running-time bounds as the earlier PlaSat2EXP [Plandowski 2012].

[2]PlaSolve is implemented in PSPACE, but the generated representation can be exponential and thus only DEXPTIME running time was claimed in the original paper.

Some research was also done in the restricted variants of word equations, most notably, there are polynomial-time algorithms for equations with only two variables [Ilie and Plandowski 2000; Dąbrowski and Plandowski 2004]. The variant with only one variable has almost-linear running time [Dąbrowski and Plandowski 2011]; the special case of only one variable with $\mathcal{O}(1)$ occurrences in the equation has an optimal linear-time algorithm [Laine and Plandowski 2011], which works in a very simple computational model.

Our contribution

In this paper, we present an application of a simple technique of *local recompression* developed by the author and successfully applied to problems related with compressed data [Jeż 2014a; Jeż 2015; Jeż 2013; Jeż 2014c; Jeż and Lohrey 2014].

Recompression. The idea of the technique is easily explained in terms of solutions of the equations (i.e. words) rather than the equations themselves: consider a solution $S(U) = S(V)$ of the equations $U = V$. In one phase we first list all pairs of different letters ab that occur as substrings in $S(U)$ and $S(V)$. For a fixed pair ab of this kind we greedily replace all occurrences of ab in $S(U)$ and $S(V)$ by a new 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 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 (so in the example, we can first compress ab, obtaining ca and then ba, which has no effect). When all such pairs are replaced, we obtain another equal strings $S'(U')$ and $S'(V')$ (note that the equation $U = V$ may have changed, and the new one is $U' = V'$). Then we iterate the process. In each phase the strings are shortened by a constant factor, and so after $\mathcal{O}(\log N)$ rounds we obtain a pair of trivial (i.e. consisting of a single letter) strings. Now, the original equation is solvable if and only if the obtained letters are the same.

The presented method has many variants, for instance, the pairs that occur seldom are not compressed, pairs that do not overlap are compressed simultaneously etc. However, the respective variants are always based on the general idea and the modifications are introduced to reach some specific goal.

The most problematic part of this idea is that it performs the operation on the solutions, which can be large. If we were to simply guess the solution and then perform the compressions, this would have running time polynomial in N, which is not acceptable. We circumvent the problem, by performing the compression directly on the equation (the *recompression*): the pairs ab occurring in the solution are identified using only the equation and the compression of the solution is done implicitly, by compressing the constants in the equations. However, not all pairs of letters can be compressed in this way, as some of them occur 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* part of the method: 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.

By simple calculations it can be shown that this method:

— transforms solvable equations to solvable equations (for proper nondeterministic choices);
— transforms unsolvable equations to unsolvable equations (for all nondeterministic choices);
— does not introduce new variables;
— in each phase shortens each string (of letters) by a constant factor;
— in one phase introduces only a linear number of new letters to the equation.

In this way, correctness easily follows and both the $\mathcal{O}(\log N \mathsf{poly}(n))$ time and PSPACE bounds hold.

Example 1.1. 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 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 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 occur in the equations, and so we can proceed to the next phase.

The main features of the presented technique is that, at the same time: it is easy to state and apply, its proof of correctness is simple and straightforward, only basic properties of word equations and strings are used in the design, application and analysis. The last property seems to be the most surprising, as in order to apply the technique, no understanding of the word equations and its solutions is actually needed. This is completely different than the approaches based on Makanin's algorithm [Makanin 1977; Jaffar 1990; Schulz 1990; Kościelski and Pacholski 1996; Gutiérrez 1998] and Plandowski's constructions [Plandowski 1999; Plandowski 2004; Plandowski 2006]; however, the PlaRytSAT [Plandowski and Rytter 1998] shared this treat.

Results. 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 $\mathcal{O}(n \log n)$ (bit) space. Furthermore, a more detailed analysis yields that for $\mathcal{O}(1)$ variables the space consumption can be lowered to $\mathcal{O}(m)$, where m is the space (counted in bits) used by the input, thus showing that for each fixed k the set of satisfiable word equations with k variables 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, it supplies an easy proof of doubly-exponential upper bound of Plandowski [1999] on lengths of length-minimal solutions as well as giving a new proof of exponential bound on the exponent of periodicity (though slightly weaker than the one presented by Kościelski and Pacholski [1996]).

The presented method can be easily modified, so that it 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. Then the algorithm for testing satisfiability is used to find out whether there is an edge between two given nodes and what is the substitution labelling it. The whole modification of our algorithm consists of replacing non-deterministic guesses of lengths of strings by guessing the arithmetical relation that these lengths satisfy.

Presentation. We start off with presenting a recompression-based algorithm for word equations, in Section 3. Firstly, we shall describe only its basic properties, which are needed to show that it works in PSPACE and has $\mathcal{O}(\log N \mathsf{poly}(n))$ (nondeterministic) running time. More involved definitions as well as results are given in the following sections. To be more precise, in Section 4 we analyse in more detail the structure of maximal repetitions of one letter in solutions of word equations. This allows reduction of space consumption to $\mathcal{O}(n \log n)$ and is essentially used in following sections. Using these results and a special encoding of letters we show that for $\mathcal{O}(1)$ variables we can lower the space consumption of the algorithm to linear one, hence showing that the word equations with k variables (for a fixed k) are context-sensitive; this is presented in Section 5. Then in Section 6, we recall the classification of solutions, given by Plandowski [2006], and related notions. Using this classification we generalise the main notions and algorithm to a generator of all solutions, see Section 7. Lastly, in Section 8, we show that a more detailed analysis of the algorithm also yields alternative (simple) proofs of exponential bound on the exponent of periodicity and double exponential bound on the size of the length minimal solutions

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. In fact, the only algorithm, with which it can be somehow compared, is the LZ77-based PlaRytSAT [Plandowski and Rytter 1998]. The key difference was that Plandowski and Rytter showed that a length-minimal solution has a short LZ77-representation and then explicitly guessed and verified it. Furthermore, the guessing was in some sense done in top-down fashion. Thus their solution, in some sense, was 'global' (as it guessed the whole solution in one go and did it top-down) and based on solutions' properties (in particular a bound on the size of the length-minimal solution is needed to bound the running time of PlaRytSAT). The novelty and importance of the here proposed method is that it does not use properties of the solutions and that it is very 'local', in the sense that it does not try to build the solution in one go, instead it modifies the equations and variables locally. In particular, in this way we are working with an SLP-encoding of the solution, which is easier in handling than the LZ77-representation.

Lastly the presented algorithm uses only a very limited variant of exponent of periodicity, when the strings in question consist only of repetitions of a single letter. In such a case an exponential bound is easy to obtain. This makes the presented algorithm somehow similar to PlaRytSAT, which does not use at all the bound on exponent of periodicity.

We believe that the presented algorithm is simpler from the previously applied. This is of course a personal feeling, but it is backed up by a smaller memory consumption. This is also backed up by a follow-up work employing this approach as well: in another work of the author, it was shown that the recompression approach in the case of equations with only one variable (and arbitrary many occurrence of it) yields a linear-time algorithm [Jeż 2014d], which is also some argument in favour of this method. Secondly, the recompression approach to word equations generalises to terms, which allowed showing that context-unification (which is a natural problem between word equations and second-order unification) is decidable in PSPACE [Jeż 2014b]; it is worth noting this is the first (and so far only) algorithm for context unification and the decidability status of this problem remained open for two decades. Lastly, a variant of the presented technique was also used to describe a set of all solutions of a word equation in free groups [Diekert et al. 2014], which is also a first result of this kind.

Related techniques. While the presented method of recompression is relatively new, some of its ideas and inspirations go quite back. This technique was developed in order to deal with fully compressed membership problem for NFA and the previous work on this topic by Lohrey and Mathissen [2011] already implemented the idea of replacing strings with fresh letters as well as modifications of the instance so that this is possible and treated maximal blocks of a single letter in a proper way. However, the replacement was not iterated, and the newly introduced blocks 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. [1997], in their work on data structure for equality tests for dynamic strings (cf. also an improved implementation of a similar data structure by Alstrup et al. [2000]). They viewed this process as 'hashing'.

A similar technique, based on replacement of pairs and blocks of the same letter was proposed also by Sakamoto [2005] in the context of constructing a smallest SLP generating a given word. His algorithm was inspired by the RePair algorithm [Larsson and Moffat 1999], which is a practical grammar-based compressor. It possessed the important features of the method: iterated replacement of pairs and blocks, phases (i.e. ignoring letters recently introduced). However, the analysis that stressed the modification of the variables (nonterminals) was not introduced and it was done in a more crude way. Additionally, Sakamoto introduced a special (and involved) pairing technique, which greatly increases the conceptual complexity of his work.

Citing conventions. As this paper aims at being stand alone, many lemmata known from the literature, are supplied with proofs (though sometimes different than the original ones). Thus, in order to distinguish these two types of results, whenever a theorem/lemma has a citation, it means that it was shown before, perhaps in a slightly different variant. Otherwise, the theorem/lemma is new.

2. MAIN NOTIONS AND TECHNIQUES: LOCAL COMPRESSION

Let us formalise the main notions. By Γ we denote the set of letters occurring in the equation $U = V$ or are used for representation of compressed strings (we do not use Σ for this purpose as it is often used for summations). The set $\mathcal{X}$ 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 the input equation, n_v denotes the number of occurrences of variables in the input equation.

A *substitution* is a morphism $S : \mathcal{X} \cup \Gamma \to \Gamma^*$, such that $S(a) = a$ for every $a \in \Gamma$. Each substitution is naturally extended to $(\mathcal{X} \cup \Gamma)^*$. The name represents the intuitive meaning that substitution simply replaces variables by (some) strings. A *solution* of an equation $U = V$ is a substitution S, such that $S(U) = S(V)$. We exclude solutions (and substitutions) that substitute ϵ for X that is present in the equation. This is not restricting, as the general word equations reduce easily to this case: given a word equation it is enough to guess for each variable X whether $S(X) = \epsilon$ or not and remove from the equation the variables for which we guessed that they have ϵ as a solution. On the other hand, by convention, we assume that $S(X) = \epsilon$ for every variable X that is not present in the equation (note that this somehow corresponds to removing the variable from the equation: when we remove X from the equation, we 'assume' that $S(X) = \epsilon$, while when $S(X) = \epsilon$ we can in fact remove X from the equation, without affecting the satisfiability).

Clearly, some solutions are 'smaller' than other and we are naturally interested in the 'smallest': We say that a solution S is *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 $S(V)$, consider the first one:

pair compression of ab. For two different letters ab occurring 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 (consider pair aa and a string aaa) and thus problematic, we need a better notion. For a letter $a \in \Gamma$ we say that a^ℓ is a a's *maximal block* of length ℓ for S, if a^ℓ occurs in $S(U)$ (or $S(V)$) and this occurrence cannot be extended by a nor to the left, neither to the right. We refer to *a's ℓ-block* for shortness. Now, we can introduce the second operation performed on the solutions:

block compression for a. For a letter a occurring in $S(U)$ and each $\ell > 1$ replace all maximal blocks 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 (recall that Makanin showed a doubly exponential bound and this was one of the crucial steps in the analysis of his algorithm).

LEMMA 2.1 (EXPONENT OF PERIODICITY BOUND [KOŚCIELSKI AND PACHOLSKI 1996]). *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 $0 < c < 2$.*

We shall use exponent of periodicity bound only to estimate the lengths of the maximal blocks (i.e. restrict w to single letters in the above definition), and in such a case the proof becomes substantially easier than the general one, see Section 8. Furthermore, an alternative approach, which does not need the exponent of periodicity at all, is also possible, see Section 4 (essentially, we incorporate the proof of Lemma 2.1 into the algorithm).

Fresh letters. As our algorithm runs in PSPACE, it may introduce a large number of 'fresh letters', and so if we insist that each of them is in fact different, this becomes problematic. However, it is enough to assume that a 'fresh letter' does not occur in the equation: after all, even if it occurred in some other iteration, this is completely irrelevant.

Remark 2.2. 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 (to be more specific, of so-called SLP): 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 (an SLP) generating $S(U)$ as a unique word in the language.

Types of pairs and blocks. Both pair compression and block compression (however they are implemented) shorten $S(U)$ (and $S(V)$), which gives the main foundation for this technique. On the other hand, sometimes it is hard to perform these operations: for instance, if we are to compress a pair ab and aX occurs 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 2.3 (*cf. [Jeż 2014a; Jeż 2015]*). 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 occurrence of u is

— *explicit*, if it comes from substring u of U (or V, respectively)
— *implicit*, it it comes from $S(X)$ for an occurrence of a variable X
— *crossing* otherwise.

A string u is *crossing* (with respect to a solution S) if it has a crossing occurrence and *non-crossing* (with respect to a solution S) otherwise.

We say that a pair of ab is a *crossing pair* (with respect to a solution S), if ab has a crossing occurrence. Otherwise, a pair is *non-crossing.* 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*, if there is a maximal block of a which has a crossing occurrence. This is equivalent to a (simpler) condition that aa is a crossing pair.

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

Visible lengths of blocks. We say that a^ℓ is *visible* in S (or ℓ is a *visible length* of a block in S), if there is an occurrence of the a's ℓ-block that is explicit or crossing or it is a prefix or suffix of some $S(X)$; we say that ℓ is a *visible length* for a if there is a visible maximal block a^ℓ.

The following lemma shows that if a pair occurs in the length-minimal solution then it has a crossing or an explicit occurrence; similarly, all lengths of maximal blocks are visible. This means that in order to know what are the pairs and blocks occurring in the length minimal solution, it is enough to know for each variable X, what is the first and last letter of $S(X)$ and what is the length of the a-prefix and b-suffix of $S(X)$.

LEMMA 2.4 (CF. [PLANDOWSKI AND RYTTER 1998, 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 occurrence in U or V and there is a visible occurrence of a^k.*

PROOF. Suppose that ab, where $a \neq b$ has only implicit occurrences. Consider S': $S'(X)$ is $S(X)$ with all abs removed, i.e. replaced with ϵ. Since all occurrences of ab in $S(U)$ and $S(V)$ are implicit, $S'(U)$ ($S'(V)$) is obtained from $S(U)$ ($S(V)$, respectively), by removing all pairs ab. Hence $S'(U) = S'(V)$, i.e. S' is a solution and it is clearly shorter than S, contradiction.

Similar argument shows that if a occurs in $S(U)$ then it has an explicit occurrence in U or V.

To streamline the rest of the presentation and analysis, in the remainder of the proof assume that both U and V begin and end with a letter and not a variable; this is easy to achieve by prepending \$ and appending \$' to both sides of the equation. Alternatively, the cases with variables beginning or ending U or V can be handled in the same way, as the general case.

Consider a maximal a block a^k, for $k > 0$ in $S(U)$ and the letter preceding (succeeding) it, say b and c, respectively; by the assumption that U and V begin and end with a letter, such b and c always exist. Consider the occurrences of ba^kc in $S(U)$ and $S(V)$. Since $b \neq a \neq c$, these occurrences cannot have overlapping a's (though, if $b = c$, these letters can overlap for different occurrences). We want to show that one of these occurrences is crossing or explicit. In such a case the corresponding a^k is visible, which ends the proof.

So suppose that none of these occurrences is crossing nor explicit. Consider S': define $S'(X)$ as $S(X)$ with each ba^kc replaced with bc. This operation is well defined, as the a^k blocks are non-overlapping. As in the case of ab pairs it can be shown that S' is a solution (since all ba^kc are implicit), which contradicts the assumption that S is length-minimal. □

Compression of noncrossing pairs and blocks

Intuitively, when ab is non-crossing, each of its occurrence 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). The appropriate algorithm is given below.

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 occurrence, the a's blocks compression consists simply of replacing explicit a blocks.

In order to show the correctness of those two procedures, we need to first introduce some terminology and notation.

Soundness and completeness. We say that 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; such a procedure *is complete*, if given a satisfiable equation

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

```
1: for each explicit $a$ occurring in $U$ or $V$ do
2:     for each $\ell$ that is a visible length of an $a$ block in $U$ or $V$ do
3:         let $a_\ell \in \Gamma$ be an unused letter
4:         replace every explicit $a$'s maximal $\ell$-block occurring in $U$ or $V$ by $a_\ell$
```

$U = V$ for some nondeterministic choices it returns a satisfiable equation $U' = V'$. Observe, that a composition of sound (complete) procedures is sound (complete, respectively)

A procedure that is complete *implements pair compression of ab* for S, if given an equation $U = V$ with a solution S, for some nondeterministic choices it returns an 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 of a for S.

Observe that operations from a very general class are sound:

LEMMA 2.5. *The following operations are sound:*

(1) *replacing all occurrences of a variable X with wXv for arbitrary $w, v \in \Gamma^*$;*
(2) *replacing all occurrences of a word $w \in \Gamma^+$ (in U and V) with a fresh letter c;*
(3) *replacing occurrences of a variable X with a word w.*

PROOF. In the first case, if S' is a solution of $U' = V'$ then S defined as $S(X) = wS'(X)v$ and $S(Y) = S'(Y)$ otherwise is a solution of $U = V$.

In the second case, if S' is a solution of $U' = V'$ then S obtained from S' by replacing each c with w is a solution of $U = V$.

Lastly, in the third case, if S' is a solution of $U' = V'$ then we can obtain S from S' by defining the substitution $S(X) = w$ and $S(Y) = S'(Y)$ in other cases. □

Properties of PairCompNCr *and* BlockCompNCr. Now we are ready to show properties of PairCompNCr(a, b) and BlockCompNCr(a).

LEMMA 2.6. PairCompNCr(a, b) *is sound and when ab is a non-crossing pair in an equation $U = V$ (with respect to some solution S) then it is complete and implements the pair compression of ab for S.*

Similarly, BlockCompNCr(a) *is sound and when a has no crossing blocks in $U = V$ (with respect to some solution S) it is complete and implements the block compression of a for S.*

PROOF. From Lemma 2.5 it follows that both PairCompNCr(a, b) and BlockCompNCr(a) are sound.

Suppose that $U = V$ has a solution S such that ab is a noncrossing pair with respect to S. Define S': $S'(X)$ is equal to $S(X)$ with each ab replaced with c (where c is a new letter). Consider $S(U)$ and $S'(U')$. Then $S'(U')$ is obtained from $S(U)$ by replacing each ab: the explicit occurrences of ab are replaced by PairCompNCr(a, b), the implicit ones are replaced by the definition of S' and by the assumption there are no crossing occurrences. The same applies to $S(V)$ and $S'(V')$. Hence $S'(U') = S'(V')$, which concludes the proof in this case.

The proof for the block compression follows in the same way. □

Crossing pairs and blocks compression

The algorithms presented in the previous section cannot be directly applied to crossing pairs or to compression of a's blocks that have crossing occurrences. To circumvent the problem, 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 is enough to change S, so that $S(X) = w$; similar action is applied to variables Y ending with a and with b to the right.

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 popping they can be compressed in parallel. As shown later, for appropriate choice of Γ_ℓ and Γ_r a constant fraction of pairs from $S(U)$ is of the form $\Gamma_\ell\Gamma_r$, see Claim 1.

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

```
1: for $X \in \mathcal{X}$ do
2:     let $b$ be the first letter of $S(X)$                                  ▷ Guess
3:     if $b \in \Gamma_r$ then
4:         replace each $X$ in $U$ and $V$ by $bX$ ▷ Implicitly change $S(X) = bw$ to $S(X) = w$
5:         if $S(X) = \epsilon$ then                                          ▷ Guess
6:             remove $X$ from $U$ and $V$
7:     let $a$ be the ...                     ▷ Perform a symmetric action for the last letter
```

LEMMA 2.7. *The* $\mathsf{Pop}(\Gamma_\ell, \Gamma_r)$ *is sound and complete.*

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

PROOF. From Lemma 2.5 it follows that $\mathsf{Pop}(\Gamma_\ell, \Gamma_r)$ is sound.

Conversely, suppose that $U = V$ has a solution S. Let $\mathsf{Pop}(\Gamma_\ell, \Gamma_r)$ always guess according to S, i.e. in line 2 it guesses b that is indeed the first letter of $S(X)$, and similarly a that is the last letter of $S(X)$, finally it removes X, when $S(X) = \epsilon$. Suppose that $b \in \Gamma_r$ and $a \in \Gamma_\ell$. Define S', which is obtained by removing the leading b and ending a, that is $bS'(X)a = S(X)$ (when $S(X) = a$ then $S'(X) = \epsilon$). It is easy to observe that $S(U) = S'(U')$, similarly $S(V) = S'(V')$, hence S' is a solution of $U' = V'$. Note that we are interested only in non-empty solutions: if the substitution for X is ϵ at any point then we simply remove all occurrences of X from the equation, in which case the solution is turned into a non-empty one.

The cases in which $b \notin \Gamma_r$ or $a \notin \Gamma_\ell$ are done in the same way (for instance, when $b \notin \Gamma_r$ and $a \in \Gamma_\ell$ then $S'(X)a = S(X)$).

It is left to show that in $U' = V'$ each pair $ab \in \Gamma_\ell\Gamma_r$ is noncrossing with respect to such defined S'. Assume for the sake of contradiction that ab is crossing with respect to S' in $U' = V'$. There are three cases

a is to the left of some variable X and the first letter of $S(X)$ is b. Since $a \in \Gamma_\ell$, then Pop did not popped a letter a from X in line 4. Hence the first letter of $S(X)$ and $S'(X)$ are the same. However, as in line 4 the letter was not popped from X and we consider the case in which Pop guessed correctly the first letter, we conclude that the first letter of $S(X)$ is not in Γ_r, while the first letter of $S'(X)$ is, contradiction.

b is to the right of some variable X and the last letter of $S(X)$ is a. This case is symmetric to the previous one.

XY occurs in the equation, $S(X)$ ends with a and $S(Y)$ begins with b. The analysis is similar to the one in the first case.

This ends the case inspection. Hence ab after the loop in line 1 is noncrossing with respect to S'. Note that for appropriate choices, all pairs ab in $\Gamma_\ell\Gamma_r$ become noncrossing. □

Now the presented subprocedures can be merged into one procedure that turns crossing pairs into noncrossing ones and then compresses them, effectively compressing crossing pairs.

LEMMA 2.8. $\mathsf{PairComp}(\Gamma_\ell, \Gamma_r)$ *is sound and complete. To be more precise, for any solution S it implements the pair compression of each pair $ab \in \Gamma_\ell\Gamma_r$.*

Algorithm 4 PairComp(Γ_ℓ, Γ_r) Turning crossing pairs from $\Gamma_\ell\Gamma_r$ into non-crossing ones and compressing them

```
1: run Pop($\Gamma_\ell, \Gamma_r$)
2: for $ab \in \Gamma_\ell\Gamma_r$ do
3:     run PairCompNCr($a, b$)
```

Proof. All subprocedures are sound, and so also PairComp(Γ_ℓ, Γ_r) is.

Concerning completeness and implementing of the pair compression: By Lemma 2.7, for appropriate choices after Pop(Γ_ℓ, Γ_r) the obtained equation $U' = V'$ has a solution S' such that $S(U) = S'(U')$ and each $ab \in \Gamma_\ell\Gamma_r$ is noncrossing with regards to S' Then, by Lemma 2.6 each of PairCompNCr(a, b) implements the pair compression, when ab is noncrossing. As occurrences of different pairs ab and $a'b'$ from $\Gamma_\ell\Gamma_r$ do not overlap, a composition of PairCompNCr(a, b) for each $ab \in \Gamma_\ell\Gamma_r$ implements the pair compression for all $ab \in \Gamma_\ell\Gamma_r$. This concludes the proof. □

The problems with crossing blocks can be solved in a similar fashion: a has a crossing block, if aa is a crossing pair. However, one popping may be not enough: when aX occurs in an equation and $S(X)$ begins with a then after left-popping of a we may end up in the same situation: aX occurs in the equation and $S(X)$ begins with a. 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 $S'(X) = w$, see Algorithm 5.

Algorithm 5 CutPrefSuff Cutting prefixes and suffixes

```
1: for $X \in \mathcal{X}$ do
2:     let $a$, $b$ be the first and last letter of $S(X)$
3:     guess $\ell_X \geq 1$, $r_X \geq 0$   ▷ $S(X) = a^{\ell_X} w b^{r_X}$, where $w$ does not begin with $a$ nor end
   with $b$
4:                                                        ▷ If $S(X) = a_X^{\ell_X}$ then $r_X = 0$
5:     replace each $X$ in $U$ and $V$ by $a^{\ell_X} X b^{r_X}$   ▷ $\ell_X$ and $b_X^{r_X}$ are stored as bitvectors,
6:                                     ▷ implicitly change $S(X) = a_X^{\ell_X} w b_X^{r_X}$ to $S(X) = w$
7:     if $S(X) = \epsilon$ then                                                 ▷ Guess
8:         remove $X$ from $U$ and $V$
```

Lemma 2.9. *CutPrefSuff is sound. It is complete, to be more precise: For a solution S of $U = V$ let for each X the a_X be the first letter of $S(X)$ and $a_X^{\ell_X}$ the a_X suffix of $S(X)$ while b_X the last letter and $b_X^{r_X}$ the b_X suffix. Then when CutPrefSuff pops $a_X^{\ell_X}$ to the left and $b_X^{r_X}$ to the right, the returned equation $U' = V'$ has a solution S' such that $S(U) = S'(U')$ and $U' = V'$ has no crossing blocks with respect to S'.*

Proof. From Lemma 2.5 we obtain that CutPrefSuff is sound.

We present the proof in the case when $S(X) \neq a^{\ell_X}$ for each variable, the argument in the other case is similar.

Suppose that $U = V$ has a solution S. Then let CutPrefSuff guess according to S, i.e. let $\ell_X \geq 1$ and $r_X \geq 1$ be guessed so that $S(X) = a^{\ell_X} w_X b^{r_X}$, where w_X does not begin with a nor end with b. Define $S'(X) = w_X$. It is easy to see that $S(U) = S'(U')$ and $S(V) = S'(V')$, in particular, S' is a solution of $U' = V'$. Furthermore, observe that as the first letter of w_X is not a and the last is not b, there are no crossing blocks in $U' = V'$ with respect to S'. □

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

Algorithm 6 BlockComp Compressing blocks of a

```
1: run CutPrefSuff                          ▷ Removes crossing blocks of $a$
2: for each letter $a \in \Gamma$ do
3:     BlockCompNCr($a$)
```

LEMMA 2.10. BlockComp *is sound. It is complete, to be more precise, let a_X be the first and b_X the last letter of $S(X)$ and ℓ_X the length of the a_X-prefix and r_X of b_X suffix of $S(X)$ (r_X is undefined if $S(X)$ is a block of letters). Then for non-deterministic choices for which the* CutPrefSuff *pops $a_X^{\ell_X}$ to the left and $b_X^{r_X}$ to the right the* BlockComp *implements the blocks compression.*

PROOF. The proof is similar to the proof of Lemma 2.8.

As BlockComp is a composition of sound operations, it is also sound.

So suppose that $U = V$ has a solution S. By Lemma 2.9 after popping the a_X prefix and b_X suffix from each variable, by CutPrefSuff in line 1, the obtained (intermediate) equation $U' = V'$ has a solution S' such that $S(U) = S'(U')$ and $S(V) = S'(V')$ and there are no crossing blocks with respect to S' in $U' = V'$. Then, by Lemma 2.6, each of the BlockCompNCr(a) is sound and implements the a blocks compression. As blocks of different letters are disjoint, this means that the loop in line 2 implements the blocks compression for each letter $a \in \Gamma$. □

3. MAIN ALGORITHM, ITS TIME AND SPACE CONSUMPTION

Now, the algorithm for testing satisfiability of word equations can be conveniently stated.

Algorithm 7 WordEqSat Checking the satisfiability of a word equation

```
1: while $|U| > 1$ or $|V| > 1$ do
2:     BlockComp
3:     $Letters \leftarrow$ the set of letters present in $U$ or $V$
4:     for $i \leftarrow 1 \,..\, 2$ do
                     ▷ One iteration to shorten the solution, one to shorten the equation
5:         guess partition of $Letters$ into $Letters_1$ and $Letters_2$
6:         PairComp($Letters_1, Letters_2$)
7: Solve the problem naively          ▷ With sides of length 1, the problem is trivial
```

We refer to one iteration of the main loop in WordEqSat as one *phase*. Observe that one phase of WordEqSat is executed in (nondeterministic) $\mathsf{poly}(|U| + |V|)$ time.

The somehow counter-intuitive repetition in line 4 has very simple explanation: one of the guessed partition guarantees that the solution's size is reduced by a constant factor, the other guarantees the same for the equation.

The properties of WordEqSat are summarised in the following theorem.

THEOREM 3.1. 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 $\mathcal{O}(n^2)$ space; furthermore, the stored equation has length $\mathcal{O}(n)$.*

The analysis of the space consumption is done in Lemma 3.2, of time consumption in Lemma 3.3 while the correctness is shown in Lemma 3.4. Furthermore, it is shown in the following section that the space consumption can be bounded by $\mathcal{O}(n \log n)$, see Lemma 4.7.

LEMMA 3.2. *For appropriate nondeterministic choices, the equations stored by (successful) computation of* WordEqSat *are of length $\mathcal{O}(n)$, the additional computation performed by* WordEqSat *use $\mathcal{O}(n^2)$ space.*

Furthermore, for appropriate nondeterministic choices, the number of phases is at most $\mathcal{O}(\log n + n_v^{cn_v})$.

PROOF. For the purpose of this proof let a *symbol* be either $a \in \Gamma$, or a^ℓ, where $a \in \Gamma$ and $\ell = \mathcal{O}(2^{cn})$, for constant c from Lemma 2.1. Let us first calculate, how many symbols are introduced into the equation in one round. By "introduce" we do not mean letters that merely replaced pairs or blocks during the compression, but rather letters that were popped into the equation from the variables.

BlockComp is run once and it runs (also once) CutPrefSuff, which introduces two symbols per variable occurrence; PairComp is run 2 times, each time it runs Pop which introduces at most two symbols per variable occurrence. Hence, in one round, at most $6n_v$ letters are introduced into the equation.

On the other hand, the main task of the whole algorithm is compression: it can be shown that for appropriate choices large fraction of letters in $U = V$ are compressed.

CLAIM 1. *Let $U = V$ has a solution S. Consider a phase of* WordEqSat *in which* BlockComp *implements the blocks compression for S, obtaining $U' = V'$ with a corresponding S' the first invocations of* PairComp *implements the pair compression obtaining $U' = V'$ with S' (obtaining $U'' = V''$ with S'') and the second implements the pair compression for $U'' = V''$ with S'' (obtaining $U''' = V'''$ with S'''). Then there are partitions $Letters_1$, $Letters_2$ and $Letters_1'$, $Letters_2'$ such that*

— *$1/6$ of letters in $S(U)$ (rounding down) is compressed in $S'''(U''')$;*
— *at least $(|U| + |V| - 3n_v - 4)/6$ of letters in U or V are compressed in U''' or V'''.*

This can be used to show (inductively) that the length of $U = V$ is at most $79n$: clearly this bound holds for the input instance, which is length n. For the inductive step consider that there are at most $6n_v$ symbols introduced into U' and V' (some of them might be compressed later). On the other hand, by Claim 1, the number of original letters of U and V decreased by at least $(|U| + |V| - 3n_v - 4)/12$. Hence,

$$\begin{aligned} |U'| + |V'| &\leq |U| + |V| - (|U| + |V| - 3n_v - 4)/12 + 6n_v \qquad (1)\\ &\leq \frac{11}{12}(|U| + |V|) + \frac{7}{12}n + 6n\\ &\leq \frac{11}{12} \cdot 79n + \frac{1}{12} \cdot 79n\\ &\leq 79n \ . \end{aligned}$$

Note, that this is the number of symbols, and not letters. However, each symbol representing a^ℓ is compressed into a single letter before the end of the phase, so the given bound holds for the number of letters as well.

Concerning the space consumption, there are three types of symbols in the equation:

— individual letters
— blocks of letters popped from variables
— variables.

Individual letters clearly take at most $\log n$ bits each, so $\mathcal{O}(n \log n)$ bits in total.

By Lemma 2.1 we know that for the length-minimal solution, the blocks of letters a^ℓ popped from a variable have lengths at most exponential in the length of the equation. Since we are interested only in the satisfiability of the equation, we may assume that the considered solution S is indeed length-minimal and so these lengths can be encoded using $\mathcal{O}(n)$ bits, which gives $\mathcal{O}(n^2)$ space consumption for such symbols in total (at any moment we have at most $2n_v \leq 2n$ such letters).

Lastly, the space consumption of variables: the number of variables is at most $n_v \leq n$, (as WordEqSat does not introduce new variable in to the equation) and so they also fit in $\mathcal{O}(n \log n_v)$ bits.

Concerning the number of phases of WordEqSat, observe that calculation similar to the one in (1) shows that if the equation $U = V$ has length larger than $120n_v$, its length drops by a constant factor in a phase. Hence, after at most $\mathcal{O}(\log n)$ phases the length of the equation is reduced to $\mathcal{O}(n_v)$. We can imagine that we restart WordEqSat for this instance. Since the length of the equation will not exceed cn_v for some constant c and the accepting computation clearly does not have loops, we obtain that the number of phases is at most $\mathcal{O}(\log n + (cn_v)^{cn_v}) = \mathcal{O}(\log n + n_v^{c' n_v})$ for some larger constant c'.

It remains to give the proof of Claim 1.

Proof of Claim 1. We first show the first property. Divide $S(U)$ into three-letters segments (ignore the last, partial segment). Consider a random partition of *Letters* into *Letters*$_1$ and *Letters*$_2$, each letters goes into the part of the partition with probability 1/2. Take any segment occurring in $S(U)$, let it be abc. We show that with probability at least 1/2 at least one letter in this segment is compressed.

If any of those letters is equal to its neighbouring letters (perhaps outside this three-letter segment), then it is compressed by Lemma 2.10. So suppose that none of these letters is the same as its neighbouring letters, in particular, they are not compressed by BlockComp. There is a compression inside abc if $ab \in \mathit{Letters}_1\mathit{Letters}_2$ or $bc \in \mathit{Letters}_1\mathit{Letters}_2$. Each of those events has probability 1/4 and they are disjoint, hence the compression occurs with probability 1/2. So regardless of the case, with probability 1/2 at least one of letters in abc is compressed. There are $\lfloor |S(U)|/3 \rfloor$ three-letter segments. The expected number of segments in which at least one letter is compressed is thus at least $\lfloor |S(U)|/6 \rfloor$, so for some partition at least $\lfloor |S(U)|/6 \rfloor$ letters are compressed.

Concerning the second property, observe, that the analysis above applies in the same way, consider any explicit word w' between two variables in U or V (or the explicit word beginning or ending U or V). Then the analysis is the same, except that the number of segments of w' is at least $\lfloor |w'|/3 \rfloor \geq |w'|/3 - 2/3$. Let now $w_1, w_2, \ldots, w_k$ be all such words in $U = V$. Then $\sum_{i=1}^k |w_i| \geq |U| + |V| - n_v$ (as at most n_v symbols in the equation are variables) and $k \leq n_v + 2$ (as at most n_v variables and the '=' sign are the ends of words). So the number of such three letter blocks is at least

$$\begin{aligned}\sum_{i=1}^k \left(\frac{|w_i|}{3} - \frac{2}{3}\right) &= \frac{\sum_{i=1}^k |w_i|}{3} - \frac{2k}{3}\\ &\geq \frac{1}{3}\left((|U| + |V| - n_v) - 2(n_v + 2)\right)\\ &= \frac{|U| + |V| - 3n_v - 4}{3} .\end{aligned}$$

The same expected-value argument yields that at least $(|U| + |V| - 3n_v - 4)/6$ letters are compressed, note that the appropriate partition is guessed as the second partition of *Letters* to *Letters*$_1$ and *Letters*$_2$. This shows the claim. □

With the end of proof of Claim 1, the lemma follows. □

LEMMA 3.3. *Let N be the size of the length-minimal solution. Then for appropriate nondeterministic choices* WordEqSat *accepts after $\mathcal{O}(\log N)$ phases.*

PROOF. The proof follows from the first item in Claim 1. □

LEMMA 3.4. WordEqSat *nondeterministically verifies the satisfiability of a word equation.*

PROOF. Firstly, observe that if $|U| = |V| = 1$ then the satisfiability of word equation is trivial to verify, which is done in last line of WordEqSat.

As WordEqSat is a composition of sound and complete subprocedures, it also is sound and complete. So if the equation is unsatisfiable, 'YES' is never returned, while if 'YES' is returned, the original equation is satisfiable. Finally, Lemma 3.3 shows that for a satisfiable solution, i.e. a one that has a length-minimal solution of length N for some N, WordEqSat accepts the equation after $\mathcal{O}(\log N)$ phases (for appropriate nondeterministic choices). Lastly, since the computation fits in polynomial memory and the accepting computation should not loop, after an exponential number of steps (kept in a counter) we can reject. □

4. MAXIMAL BLOCKS

The quadratic memory consumption of WordEqSat is due to BlockComp. Since we aim at $\mathcal{O}(n \log n)$ memory consumption (counted in bits), we need to improve it. To this end we analyse more carefully the structure and possible lengths of maximal blocks. This analysis allows a different approach to blocks compression: instead of guessing the explicit values of a-prefixes and b-suffixes of variables, we parametrise those values and check which sets of values of those parameters are allowed. To be more precise, the lengths of maximal blocks are expressed in terms of lengths of a-prefixes and b-suffixes while blocks of the same lengths are identified (using non-deterministic guesses) and replaced by the same letter. The verification of feasibility of the guesses boils down to checking the satisfiability of a system of linear Diophantine equations. In particular, the actual lengths of the blocks are not important, it is the satisfiability of the system that matters (in this way we can omit the space consuming guesses of the exact lengths); due to special form of the Diophantine equations, their satisfiability can be checked in linear space.

The contents of this section is a simple case of the general approach (of decompositions according to some primitive words) presented in the work of Kościelski and Pacholski [1996].

Arithmetic expressions

We shall now define what is a general form of lengths of maximal blocks in $S(U) = S(V)$. Those lengths are parametrised by the lengths of a-prefixes and b-suffixes of $S(X)$, and so are not simply numbers, but rather expressions involving both numbers and some parameters.

Consider arithmetic expressions using natural constants and variables, such that all expressions are linear in these variables. These expressions are obtained from a word equation $U = V$ in a way described in the following subsection. We say that a set of e_1, e_2, ..., e_m is a *small set of linear Diophantine expressions* (for a word equation $U = V$ with n_v occurrences of variables) if

— the coefficients and constants in each expression are positive natural numbers;
— each variable in the expressions is either x_X or y_X, where X is a variable from $U = V$;
— if X occurs k times in $U = V$ then the sum of coefficients of x_X (y_X) is at most k;
— the sum of values of constants in $\{e_i\}_{i=1}^m$ is at most $|U| + |V| - n_v$.

We say that a system of linear Diophantine equations and inequalities (all inequalities are of the form $x \geq 1$) is a *small linear Diophantine system*, if sides of its equalities form a small set of linear Diophantine expressions and each e_i in this set is used at most two times in this system. As a simple consequence each small system of linear Diophantine equations and inequalities has the following properties:

— each variable in the system is either x_X or y_X, where X is a variable from $U = V$;
— if X occurs k times in $U = V$ then x_X (y_X) has sum of values of its coefficients at most $2k$;
— the sum of values of constants is at most $2(|U| + |V|)$; $(2(|U| + |V| - n_v)$ comes from the equalities while $2n_v$ from the right-hand sides of inequalities).

The size of the small linear Diophantine system is proportional to the size of representation of $U = V$ and furthermore its satisfiability can be (non-deterministically) checked in the same space limits.

LEMMA 4.1. *If the equation $U = V$ is represented using m bits, then the corresponding small linear Diophantine system can be encoded using $\mathcal{O}(m)$ bits, moreover, it can be (non-deterministically) verified in the same space, whether it has a solution in natural numbers.*

PROOF. We encode the equalities in unary: i.e. each constant c is represented as $\underbrace{1 + 1 + \cdots + 1}_{c \text{ times}}$, while each cx is represented as $\underbrace{x + x + \cdots + x}_{c \text{ times}}$. The variables x_X, y_X are encoded in the same way as X in $U = V$, with additional bit to distinguish them. The assumptions on the small system guarantee that:

— The total space used by constants is $2(|U| + |V| - n_v)$, which is at most $2m$.
— The space used by variables in equalities is at most 8 times as much as the space used by variables in $U = V$: a denotation of a variable x_X (y_X) is at most twice as long as the one for variable X and it occurs at most 2 times more in the small linear Diophantine system as X in $U = V$; additionally, we need one bit to distinguish x_X and y_X.
— All inequalities use a variable and one bit to denote 1, so it can be shown (as in the item above) that the space consumption is at most 6 times as much as the space used by variables in $U = V$.

The additional space needed to denote '+' and '=' may increase the space usage only by a constant (note that we might need to change the denotation of other symbols a bit). So indeed the used space is just constantly larger than the space used by $U = V$.

The idea of the verification is that instead of guessing the whole solutions, we guess only the last bits (i.e. the parity of integer variables), verify this guess, simplify the equation and proceed: for each variable x we guess, whether it is even or odd and appropriately replace it with $2x$ or $2x + 1$. Then we verify the guess by checking whether both sides of each equality have the same parity. If so, we divide each side of the inequality by 2 (rounding down) and proceed in the same fashion. Note, that in this way, the coefficients at each variable remain the same over the whole procedure.

There is a little comment concerning the inequalities: all of them were initially $x \geq 1$, and during the algorithms they can be also of the form $x \geq 0$. When rounding down, we need to take care that rounding is done in an appropriate way, for instance $2x \geq 1$ is in fact $2x \geq 2$ and so after dividing and rounding we should end up with $x \geq 1$ again. Observe that this boils down to replacing $x \geq 1$ by $x \geq 0$ if and only if x is replaced by $2x + 1$, otherwise, the inequality remains as it were.

Suppose that the small linear system has a solution $(q_1, q_2, \ldots, q_r)$. We show that for some nondeterministic guesses, the obtained system has a solution $(\lfloor q_1/2 \rfloor, \lfloor q_2/2 \rfloor, \ldots, \lfloor q_r/2 \rfloor)$. Let the algorithm guesses the parity of x according to $(q_1, q_2, \ldots, q_r)$. Then after the loop in line 2 the obtained system has a solution $(\lfloor q_1/2 \rfloor, \lfloor q_2/2 \rfloor, \ldots, \lfloor q_r/2 \rfloor)$. Since each coefficient by the variable is even, the constants at the sides of each equation should be of the same parity, and so the algorithm does not terminate in line 5. Line 7 halves each equation.

Algorithm 8 VerifyDiophantine Checks the satisfiability of a small linear Diophantine system

```
1: while there is a non-zero constant *or* an inequality $x \geq 1$ do
2:     for each variable $x$ do
3:         guess $b_x \in \{0, 1\}$
4:         replace each $x$ with $2x + b_x$
5:     if there is an equation with different parity of constants on the sides then
6:         return Unsatisfiable
7:     divide each equation by 2, rounding down
8:     divide each inequality by 2, round appropriately
9: return Satisfiable                                  ▷ Has a trivial solution $(0, \ldots, 0)$
```

Observe, that an inequality $2x \geq 0$ is equivalent to $x \geq 0$ and inequalities $2x \geq 1$ in fact meant that $2x \geq 2$ and so they are also simply halved.

On the other hand, if $(q_1, q_2, \ldots, q_r)$ is a solution after the changes, then $(2q_1 + b_1, 2q_2 + b_2, \ldots, 2q_r + b_r)$ was the solution of the system at the beginning of the iteration.

Concerning the space usage of VerifyDiophantine, the inequalities are simply stored as a bit for each variable (bit set to i means that $x \geq i$). When the start systems has size $\mathcal{O}(m)$, the intermediate ones have size $\mathcal{O}(m)$ as well (with a larger constant, though): observe that the only new constants (which are also stored in unary) are the 1s from $2x + 1$. Suppose that initially the sum of constants was c and the sum of coefficients at variables m. We show by induction that the sum of constants during the algorithm is at most $\max(c, m)$. This clearly holds in the beginning, let us investigate the changes in one round. The sum of all b_xs introduced is at most m so afterwards the sum of constants is at most $\max(c, m) + m$. Each constant is halved in this round (rounding down), so their sum is at most $(\max(c, m) + m)/2 \leq \max(c, m)$ at the end of the round, as claimed. □

Constructing a small Diophantine system from a word equation

Lemma 2.4 suggests that the crucial to consider, when dealing with maximal blocks, are the lengths of the a prefixes and b-suffixes of $S(X)$ for different variables X. We now show that this intuition can be formally stated and later show how to use this formulation in a more efficient implementation of BlockComp.

In order to perform the blocks compression in BlockComp, we first guess the first (a_X) and last (b_X) letter of $S(X)$ for each X, then the length of the a_X-prefix (ℓ_X) and b_X-suffix (r_X) of $S(X)$, pop the a_X-prefix and b_X suffix from X and finally compress the maximal explicit blocks. We now defer the guess of ℓ_X and r_X for as long as possible, in fact, we shall not guess them at all. Intuitively, we treat the lengths of the a_X-prefix and b_X-suffix of $S(X)$ as *parameters* (or variables ranging over positive natural numbers) and to stress this we denote them by x_X and y_X. We can pop prefixes in this way, by replacing X with $a_X^{x_X} X b_X^{y_X}$ and even calculate the lengths e_1, e_2, … of explicit maximal blocks E_1, E_2, … in $S(U)$ and $S(V)$: these are arithmetic expressions using constants and parameters $\{x_X, y_X\}_{X \in \mathcal{X}}$. In order to compress such maximal blocks, we guess which of them are equal, and write the corresponding linear Diophantine equations. If this system is feasible then we replace the maximal blocks: E_i and E_j are replaced with the same letter if and only if

— they are blocks of the same letter a *and*
— e_i and e_j are declared to be equal.

.

This approach still requires that we know what is the first and last letter of each variable. A *prefix-suffix structure* for an equation $U = V$ tells for each X occurring in the equation what is its first (by convention: a_X) and last (by convention: b_X) letter and whether X is a block of one letter, i.e. whether $S(X) \in a_X^+$.

Given a prefix-suffix structure by x_X and y_X we denote the parameters (or variables) that denote the lengths of the a_X-prefix and b_X-suffix of X (if X is a block of letters then by convention the y_X is not used). Given a prefix-suffix structure we can identify the visible maximal blocks and describe their lengths (in terms of $\{x_X, y_X\}_{X\in\mathcal{X}}$), they are simply arithmetic expressions in $\{x_X, y_X\}_{X\in\mathcal{X}}$. To distinguish such blocks from 'real' blocks, we call them *parametrised visible maximal blocks* and denote by $\mathcal{E}_1, \ldots, \mathcal{E}_m$, while the usual blocks by $E_1, \ldots, E_m$.

LEMMA 4.2. *Consider a prefix-suffix structure for $U = V$. Let $\mathcal{E}_1, \mathcal{E}_2, \ldots$ be parametrised visible maximal blocks of $U = V$ for this structure and $e_1, e_2, \ldots$ be their lengths expressed in terms of $\{x_X, y_X\}_{X\in\mathcal{X}}$ and constants. Then $e_1, e_2, \ldots$ are a small set of linear Diophantine expressions in $\{x_X, y_X\}_{X\in\mathcal{X}}$.*

PROOF. Consider a parametrised visible $\mathcal{E}_i$:

— $\mathcal{E}_i$ may begin with either an explicit a or a maximal a-suffix of some $S(X)$ (which may be whole $S(X)$);
— 'in the middle' it may contain either explicit as or $S(X) \in a^+$;
— it ends with an explicit a or a maximal a-prefix of some $S(X)$ (which may be whole $S(X)$).

Thus, whenever $\mathcal{E}_i$ is visible, e_i is a linear combination of x_X, y_X (where $X \in \mathcal{X}$) and natural numbers.

We show that the terms $e_1, \ldots, e_k$ are a small set of linear Diophantine expressions. For the purpose of the proof, denote by n_X the number of times variable X is used in the equation $U = V$. We bound the number of times x_X and y_X occur in expressions $e_1, \ldots, e_k$ and the size of additive constants used in $e_1, \ldots, e_k$:

— each x_X (y_X) occurs at most n_X times, as for a fixed occurrence of variable X there is at most one parametrised maximal block $\mathcal{E}_i$ that spans over the prefix (suffix, respectively) of this occurrence;
— the total size of used constants is $|U| + |V| - n_v$: for a fixed explicit occurrence of a letter a, there are is exactly one parametrised maximal block $\mathcal{E}_i$ that spans over it. $\square$

Now let us explain the relation between solutions and prefix-suffix structures. A solution S and prefix-suffix structure are *coherent* if $S(X)$ indeed begins and ends with a_X and b_X and $S(X)$ is a block of letters if and only if the prefix-suffix structure says so. Furthermore, there is also a relation between lengths of maximal visible blocks of S and parametrised visible blocks: intuitively, when ℓ_X and r_X are the lengths of the a_X-prefix and b_X-suffix of $S(X)$ then the length of the i-th visible maximal block is e_i with $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ substituted for $\{x_X, y_X\}_{X\in\mathcal{X}}$. To make this more formal and shorter, in the following, for an arithmetic expression e in variables $\{x_X, y_X\}_{X\in\mathcal{X}}$, we use $e[\{\ell_X, r_X\}_{X\in\mathcal{X}}]$ to denote the value of e when $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ is substituted for $\{x_X, y_X\}_{X\in\mathcal{X}}$.

LEMMA 4.3. *Given a coherent prefix-suffix structure and a substitution S, let $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_k$ be the parametrised visible maximal blocks for this structure, and $e_1, e_2, \ldots, e_k$ their lengths, while $E_1, E_2, \ldots, E_{k'}$ be the lengths of the visible maximal blocks for S. Then $k' = k$ and for each i the $\mathcal{E}_i$ and E_i are blocks of the same letter and $|E_i| = e_i[\{\ell_X, r_X\}_{X\in\mathcal{X}}]$, where ℓ_X and r_X are the lengths of the a_X prefix and b_X suffix of $S(X)$.*

PROOF. Since the first and last letters of $S(X)$ are the same as in the prefix-suffix structure and $S(X)$ is a block of letters if and only if prefix-suffix structure says so, the $\mathcal{E}_i$ and E_i consists of the corresponding explicit letters and prefixes/suffixes of variables. In particular, the number of parametrised blocks and blocks is the same, for each i the $\mathcal{E}_i$ and

E_i are blocks of the same letter and lastly, the length of E_i corresponds to the length of $\mathcal{E}_i$ in which the values of parameters x_X and y_X are replaced by the actual lengths of prefixes and suffixes of $S(X)$; this shows that $|E_i| = e_i[\{\ell_X, r_X\}_{X\in\mathcal{X}}]$. □

So far we do not know, which parametrised blocks represent blocks of the same length. To identify such blocks, we write a (small) linear Diophantine system that bounds the e_1, …, e_k together: we guess the partition of $e_1, e_2, \ldots, e_k$, elements of one partition should correspond to parametrised blocks of the same length (which in particular means that we assume that if e_i and e_j are in one part then E_i and E_j are parametrised blocks of the same letter). Then for each part $\{e_{i_1}, e_{i_2}, \ldots, e_{i_m}\}$ of the partition we write equations equalising their lengths:

$$e_{i_1} = e_{i_2},\ e_{i_2} = e_{i_3},\ \ldots,\ e_{i_{m-1}} = e_{i_m} \ . \tag{2}$$

We also add the inequalities $x_X \geq 1$ (and $y_X \geq 1$) for every variable used in the equalities (intuitively, since we claim that $S(X)$ begins or ends with a block of letters of length x_X or y_X, we want those blocks to consist of at least one letter). If for some variable X the $S(X)$ is a block of letters, we use only x_X in the equations, y_X is not used in the constructed system. Thus we have obtained a linear Diophantine system in x_X and y_X. This is formalised in WordtoDioph.

Algorithm 9 WordtoDioph Creates a system of equations for a prefix-suffix structure

```
Require: prefix-suffix structure
 1: for $X \in \mathcal{X}$ do
 2:     if $X$ represents a block of letters then        ▷ According to the prefix-suffix structure
 3:         let $a_X$ be the first letter of $X$         ▷ According to the prefix-suffix structure
 4:         introduce parameter $x_X$                     ▷ $S(X) = a_X^{x_X}$
 5:         add inequality $x_X \geq 1$ to $D$            ▷ $S(X)$ is non-trivial
 6:     else
 7:         let $a_X$ and $b_X$ be the first and last letter of $X$
                                                          ▷ According to the prefix-suffix structure
 8:         introduce parameters $x_X$ and $y_X$
                                        ▷ Lengths of the of $a_X$-prefix and $b_X$-suffix of $S(X)$
 9:         add inequalities $x_X \geq 1$ and $y_X \geq 1$ to $D$
                                                  ▷ The leading and ending blocks are non-trivial
10: let $\{\mathcal{E}_1, \ldots, \mathcal{E}_k\}$ be the parametrised visible maximal blocks (read from left to right)
11: for each $\mathcal{E}_i$ do
12:     let $e_i \leftarrow |\mathcal{E}_i|$              ▷ Arithmetic expression in $\{x_X, y_X\}_{X\in\mathcal{X}}$
13: partition $\{\mathcal{E}_1, \ldots, \mathcal{E}_k\}$, each part has only $a$-blocks for some $a$    ▷ Guess
14: for each part $\{\mathcal{E}_{i_1}, \ldots, \mathcal{E}_{i_{k_p}}\}$ do
15:     for each $\mathcal{E}_{i_j} \in \{\mathcal{E}_{i_1}, \ldots, \mathcal{E}_{i_{k_p}}\}$ do
16:         add an equation $e_{i_j} = e_{i_{j+1}}$ to $D$      ▷ Ignore the meaningless last equation
17: return the partition, arithmetic expressions $e_1, \ldots, e_k$ and $D$.
```

Lemma 4.4. *The system of linear Diophantine equations and inequalities returned by* WordtoDioph *is small.*

Proof. The system of Diophantine linear equations is small if its sides form a small set of linear expressions and each such an expression is used at most twice. The sides are of this form by Lemma 4.2 and each expression is used at most twice by the construction. □

It remains to link the constructed system to some solution of the word equation: we say that S and a system D constructed by WordtoDioph are *coherent* (or simply, that D is S-coherent), if S is coherent with the prefix-suffix structure used by WordtoDioph to generate D and the partition of parametrised visible maximal blocks $\{\mathcal{E}_1, \ldots, \mathcal{E}_k\}$ in line 13 is done as in $S(U) = S(V)$, i.e. $\mathcal{E}_i$ and $\mathcal{E}_j$ go into the same part if and only if the corresponding maximal blocks E_i and E_j of $S(U) = S(V)$ are equal.

LEMMA 4.5. *For a solution S of a word equation $U = V$ there is a unique S-coherent system D. When ℓ_X and r_X are the lengths of the a_X-prefix and b_X-suffix of $S(X)$ the $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ is a solution of D.*

PROOF. Concerning the existence and uniqueness: in WordtoDioph we simply make all the nondeterministic choices according to S.

To see that $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ is a solution of D observe that an equation $e_i = e_j$ is added only when $|E_i| = |E_j|$. From Lemma 4.3 we know that $e_i[\{\ell_X, r_X\}_{X\in\mathcal{X}}] = |E_i|$ and $e_j[\{\ell_X, r_X\}_{X\in\mathcal{X}}] = |E_j|$, hence $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ satisfies this equation and as $e_i = e_j$ was chosen arbitrarily, we obtain that it satisfies D. Note that all the inequalities are trivially satisfied. □

Improving BlockComp

We make the next step in the outlined strategy: after guessing a small system of Diophantine equations, we verify its satisfiability and use it to perform the block compression. To be more precise: the BlockCompImp firstly guesses the prefix-suffix structure, then uses WordtoDioph to generate a system of linear equations out of $U = V$, then it verifies its satisfiability. Then it pops the prefixes and suffixes out of each variable, however, it does not guess the exact lengths, but rather uses the prefix-suffix structure, i.e. it pops a^{x_X} to the left of X and b^{y_X} to the right (of course, no popping to the right is done when X is removed after the initial pop, i.e. the prefix-suffix structure declares that $S(X)$ is a block of letters). Then we replace blocks of the same letter whose lengths are equalised in the system of the linear Diophantine equations by a fresh letter.

Note that in this way in the word equation (temporarily) we have symbols a^x, where x is a variable with a value in natural numbers. We are not going to give any semantics for that, as this is not needed, but still we would like to consider maximal blocks of letters: the a^x can be a part of a maximal a-block, moreover, we assume that $x > 0$, i.e. if the letter to the left of a^x is $b \neq a$ and the same letter is to the right, those bs are in different blocks. We use the name *parametrised explicit maximal blocks* with an obvious meaning.

As a first step, we begin with describing the improved version of CutPrefSuff, the CutPrefSuffImp.

Algorithm 10 CutPrefSuffImp Cutting prefixes and suffixes, parametrised version

```
Require: prefix-suffix structure
1: for $X \in \mathcal{X}$ do
2:     let $a_X$, $b_X$ be the first and last letter of $S(X)$  ▷ Given by the prefix-suffix structure
3:     if $X$ is a block of letters then  ▷ According to the prefix-suffix structure
4:         replace each $X$ in $U$ and $V$ by $a^{x_X}$
5:     else
6:         replace each $X$ in $U$ and $V$ by $a^{x_X} X b^{y_X}$  ▷ $x_X$, $y_X$ are variables
7:         if $S(X) = \epsilon$ then  ▷ Guess
8:             remove $X$ from $U$ and $V$
```

We are not going to state the exact properties of CutPrefSuffImp, we shall give them collectively for BlockCompImp, the improved version of BlockComp. For now we only note that during CutPrefSuffImp the visible parametrised blocks are changed into explicit ones.

Lemma 4.6. *Let $\mathcal{E}_1$, $\mathcal{E}_2$, ..., $\mathcal{E}_m$ be the parametrised visible maximal blocks for a prefix-suffix structure. Then after* CutPrefSuffImp *these are exactly the parametrised explicit maximal blocks.*

Proof. The proof is obvious: whenever a prefix (suffix) of an occurrence of X took part in some parametrised visible maximal block, we popped this prefix (suffix) from X and so now it is part of a corresponding parametrised explicit maximal block. □

Now we are ready to describe the improved version of BlockComp as well as its properties. The procedure first guesses the prefix-suffix structure of the solution and then nondeterministically constructs the coherent Diophantine system. Then it verifies the satisfiability of this system, rejecting if the verification is negative. Then it pops the (parametrised) prefix and suffix from each variable (according to the prefix-suffix structure) and identifies the maximal (parametrised) blocks and partitions them according to the system of Diophantine equations. For each part it replaces all (parametrised) maximal blocks from it by a fresh letter.

Algorithm 11 BlockCompImp

```
1: guess the prefix-suffix structure
2: run WordtoDioph
3: run VerifyDiophantine on $D$          ▷ Check if the guessed choices can be fulfilled
4: run CutPrefSuffImp                    ▷ There are no crossing blocks
5: let $\{\mathcal{E}_1, \ldots, \mathcal{E}_k\}$ be the explicit parametrised maximal blocks
         ▷ Those are exactly the parametrised visible maximal blocks from WordtoDioph
6: for each part $\{\mathcal{E}_{i_1}, \ldots, \mathcal{E}_{i_{k_p}}\}$ of the partition returned by WordtoDioph do
7:     let $a_{e_{i_1}} \in \Gamma$ be an unused letter
8:     for each $\mathcal{E}_{i_j}$ do
9:         replace every $\mathcal{E}_{i_j}$ by $a_{e_{i_1}}$
```

Lemma 4.7 (cf. Lemma 2.10). BlockCompImp *is sound. For a solution S of $U = V$ and the nondeterministic choices that lead to a creation of an S-coherent system by* WordtoDioph *the* BlockCompImp *implements the blocks compression; to be more precise, the obtained word equation $U' = V'$ is identical (up to renaming the letters) to the equation obtained by* BlockComp *when it implements the block compression for S. In particular,* BlockCompImp *is complete.*

BlockCompImp *uses a constant time more memory than the equation $U = V$, in particular, the additional memory usage of* WordEqSat *when using* BlockCompImp *is linear.*

Proof. Suppose that BlockCompImp applied on $U = V$ created a linear Diophantine system D that has a solution $\{\ell_X, r_X\}_{X \in \mathcal{X}}$. Then we can think of BlockCompImp as if it replaced each X with $a^{\ell_X} X b^{r_X}$ and then replaced some blocks of the same letter and the same length with fresh letters. Thus by Lemma 2.5 it is sound.

Concerning completeness and the implementation of the block compression, we use the fact that BlockComp has both those properties (Lemma 2.10). Suppose that $U = V$ has a solution S and consider the satisfiable instance $U' = V'$ obtained by BlockComp out of $U = V$ that has a solution S' such that $S'(U')$ is obtained from $S(U)$ by compressing blocks of letters (by Lemma 2.10 we know that for some non-deterministic choices indeed BlockComp returns such an equation). We show that the run of BlockCompImp in which

WordtoDioph returns the S-coherent system D returns $U' = V'$ (up to renaming letters), which will end the proof. In the following, let ℓ_X and r_X be the length of the prefix and suffix popped from X by CutPrefSuff, by Lemma 2.10 we know that we can restrict ourselves to the case when ℓ_X is the length of the a_X prefix of $S(X)$ and r_X of the b_X suffix of $S(X)$.

Concerning the corresponding BlockCompImp, consider the non-deterministic choices for which the WordtoDioph returns a small Diophantine system that is S-coherent: by Lemma 4.5 such a system exists and it is satisfiable. Let $\mathcal{E}_1$, $\mathcal{E}_2$, …, $\mathcal{E}_k$ be the consecutive parametrised visible maximal blocks in $U = V$ and E_1, E_2, …, $E_{k'}$ be the visible maximal blocks in $U = V$ for S. By Lemma 4.5 the $\mathcal{E}_i$ and E_i are blocks of the same letter and $k = k'$. Consider, what happens with the former blocks when we apply CutPrefSuffImp: they become the parametrised explicit maximal blocks, see Lemma 4.6. Similarly, the E_1, E_2, …, E_k become explicit blocks when CutPrefSuff is applied on them, as we pop the a_X-prefix and b_X-suffix from each variable. Now, $\mathcal{E}_i$ and $\mathcal{E}_j$ are replaced with the same letter by BlockCompImp if and only if e_i and e_j are equalised in D (note that not necessarily $e_i = e_j$ is in D, but it contains equation that imply this, i.e. a sequence of equations $e_i = e_{i_1}$, $e_{i_1} = e_{i_2}$, …, $e_{i_m} = e_j$). By definition of the S-coherent system this happens if and only if $|E_i| = |E_j|$. Hence $\mathcal{E}_i$ and $\mathcal{E}_j$ are replaced with the same letter by BlockCompImp if and only if E_i and E_j are by BlockComp. Which ends the proof for the second claim.

Concerning the memory consumption, observe that by Lemma 4.1, the linear Diophantine system D can be encoded using only a constant more bits than the word equation and the same space can be used to verify the satisfiability of the system. All other operations can be easily implemented in the same memory bounds. ◻

Similar solutions

Thanks to Lemma 4.5 we know that each solution S has a corresponding system of Diophantine equations (the S-coherent one) and that the lengths $\{\ell_X, r_X\}_{X \in \mathcal{X}}$ of the a-prefixes and b-suffixes of S are a solution of the S-coherent system. Still, there are two questions: on one hand for a given system D we know nothing about letters in $S(X)$ that are not in the a_X-prefix nor in the b_X-suffix of $S(X)$. Moreover, other solutions of D should also induce a solution of a word equation. In this section we investigate the relations between all such induce solutions of the word equation. Intuitively, different solutions S' of $U = V$ induced in this way differ from S in lengths of maximal blocks in $S'(U)$.

We say that two words w and w' are *similar*, if $w = E_1 E_2 \dots E_k$ and $w' = E'_1 E'_2 \dots E'_k$, where for each i the E_i and E'_i are non-empty blocks of the same letter, i.e. for some a we have $E_i, E'_i \in a^+$, and they are maximal blocks in w and w', respectively, i.e. E_{i-1} and E_{i+1} as well as E'_{i-1} and E'_{i+1} are blocks of some other letters. Two substitutions S and S' are *similar*, if for every variable X the $S(X)$ and $S'(X)$ are similar. Note that from the definition it follows that if S and S' are similar than they have the same coherent prefix-suffix structure.

If S and S' are similar then also $S(U)$ and $S'(U)$ are.

LEMMA 4.8. *Let S and S' be similar solutions of a word equation $U = V$. Then $S(U)$ and $S'(U)$ are similar.*

Consider representation of $S(U)$ and $S'(U)$ as concatenation of maximal blocks E_1, E_2, …, E_k, and E'_1, E'_2, …, $E'_{k'}$ respectively. Then for each i the E_i is a crossing (visible) block if and only if E'_i is.

PROOF. Concerning the first claim: since S and S' are similar, for each variable the $S(X)$ and $S'(X)$ can be represented as $F_1 \dots F_m$ and $F'_1 \dots F'_m$, where each F and F' are maximal blocks of letters and F_i and F'_i are blocks of the same letter. Now each E_i and E'_i consist of corresponding explicit letters as well as corresponding blocks, in particular, E_i includes some F_j from $S(X)$ if and only if E'_i includes some F'_j from $S'(X)$.

Concerning the second claim, we show it for the visible case, the proof is the same for the crossing blocks. By symmetry it is enough to show that when E_i is visible (crossing) then also E'_i is. We use the same observation, as before: note that E_i is visible, when it contains an explicit letter or a leading (or ending) block F_j of letters from some $S(X)$. But then the same happens for E'_i and F'_j. $\square$

Now, given a solution S of a word equation $U = V$ and its S-coherent system D of Diophantine equations we shall define a class of solutions of $U = V$, all such solutions will be similar. Each such a solution S' is uniquely defined by one solution $\{\ell'_X, r'_X\}_{X\in\mathcal{X}}$ of D, to stress it we denote it by $S[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$. When $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ are the lengths of a_X-prefixes and b_X-suffixes of $S(X)$ (for each X), the construction shall guarantee that $S[\{\ell_X, r_X\}_{X\in\mathcal{X}}] = S$.

Consider a variable X and its representation as maximal blocks $F_1F_2 \dots F_k$ of $S(X)$. Since $S[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$ is to be similar with S, $S[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}](X)$ is defined as $F'_1F'_2 \dots F'_k$, where F_i and F'_i are blocks of the same letter. It is left to define the lengths of F'_1, F'_2, ..., F'_k with respect to $\{\ell'_X, r'_X\}_{X\in\mathcal{X}}$. Let e_1, e_2, ..., e_i be the length of the parametrised visible maximal blocks of the prefix-suffix structure that is coherent with S.

Consider any solution $\{\ell'_X, r'_X\}_{X\in\mathcal{X}}$ of D and blocks F_i and F'_i in $S(X)$ and $S[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}](X)$, respectively. There are three cases:

(L 1) F_i is a prefix or a suffix of $S(X)$ (and so also F'_i is for $S'(X)$). Then the length of F_i is ℓ_X when it is a prefix (or r_X when it is a suffix) and we set the length of F'_i to ℓ'_X (or r_X, respectively).

(L 2) F_i is not the prefix nor the suffix but it has visible length (so F'_i is also not a prefix nor a suffix and has visible length, by Lemma 4.8). Let $E_{i'}$ be a visible block (in $S(U)$ or $S(V)$) such that $|E_{i'}| = |F_i|$, by definition of a visible length such a block exists. By Lemma 4.5 we know that $|E_{i'}| = e_{i'}[\{\ell_X, r_X\}_{X\in\mathcal{X}}]$ and so we set $|F'_i|$ to $e_i[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$.

(L 3) F_i is not the prefix nor the suffix and does not have a visible length (so the same applies to E'_i). In this case we simply give F'_i the same length as F_i.

It remains to check the validity of the construction.

Lemma 4.9. *Given a solution S of a word equation $U = V$ and the S-coherent Diophantine system D, for each solution $\{\ell'_X, r'_X\}_{X\in\mathcal{X}}$ the corresponding $S' = S[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$ is a solution of $U = V$ that is similar to S.*

Furthermore, for any variable X we can give an arithmetic expression e_X in variables $\{x_X, y_X\}_{X\in\mathcal{X}}$ such that $|S'(X)| = e_X[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$ and e_X depends on x_X and y_X (if the latter exists).

Proof. It follows straight from the definition of S' that S and S' are similar.

Let $E_1, \dots, E_k$ be a representation of $S(U)$ as a concatenation of maximal blocks and $E'_1, \dots, E'_k$ a representation of $S'(U)$. Since by Lemma 4.8 the $S'(U)$ and $S(U)$ are similar, to show that S' is a solution of $U = V$ it is enough to show that $|E_i| = |E_j|$ then also $|E'_i| = |E'_j|$, and the rest follows by a simple induction.

Consider a maximal block E_i. There are three cases:

visible. It is visible. Then by Lemma 4.8 also E'_i is visible. Furthermore, by Lemma 4.3 the length of E'_i is $e_i[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$.

invisible with visible length. It is invisible but has a visible length, so also E'_i is invisible, by Lemma 4.8. Then E'_i is a block in some $S'(X)$ that is not a prefix not a suffix of $S'(X)$. Then by (L 2) its length is $e_{i'}[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$, where $E_{i'}$ is a visible block such that $|E_{i'}| = |E_i|$. In particular, in the previous case it was shown that $|E'_{i'}| = e_{i'}[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$ and so $|E'_i| = |E'_{i'}| = e_{i'}[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$.

invisible length. It has an invisible length, in particular, it is invisible. Then by Lemma 4.8 also E'_i is invisible and so by (L 3) it has length $|E_i|$

Now, consider some E'_i and E'_j that are blocks of the same letter and such that $|E_i| = |E_j|$. There are two possibilities: $|E_i|$ is a visible length or it is an invisible length. If it is an invisible length, then it was shown already that $|E'_i| = |E_i|$ and $|E'_j| = |E_j|$ and hence $|E'_i| = |E'_j|$ as claimed. If it is a visible length, then it was already shown that $|E'_i| = e_{i'}[\{\ell'_X, r'_X\}_{X \in \mathcal{X}}]$ and $|E'_j| = e_{j'}[\{\ell'_X, r'_X\}_{X \in \mathcal{X}}]$, where $e_{i'}$ and $e_{j'}$ are such that $|E_i| = |E_{i'}|$ and $|E_j| = |E_{j'}|$ (note that it might be that $i = i'$ or that $i \neq i'$ and similarly for j and j'). Then $|E_{i'}| = |E_{j'}|$ and so the equality $e_{i'} = e_{j'}$ follows from the system D (i.e. there is a sequence of equations $e_{i'} = e_{i_1}, e_{i_1} = e_{i_2}, \ldots, e_{i_p} = e_{j'}$). As $\{\ell'_X, r'_X\}_{X \in \mathcal{X}}$ is a solution of D, we conclude that $e_{i'}[\{\ell'_X, r'_X\}_{X \in \mathcal{X}}] = e_{j'}[\{\ell'_X, r'_X\}_{X \in \mathcal{X}}]$, and so $|E'_i| = |E'_j|$.

It is left to show the second claim, concerning the existence of an arithmetic expression for e_X. This is obvious by the definition of S': let $S(X) = F_1F_2 \ldots F_m$, where each F_i is a maximal block. Then F_1 has length ℓ_X, F_m length r_X (so we add x_X and y_X to e_X), when F_i has invisible length then F'_i has length $|F_i|$ (so we add a constant $|F_i|$ to e_X) and if it has a visible length, then the length is expressed as some $e_i[\{\ell'_X, r'_X\}_{X \in \mathcal{X}}]$ (so we add e_i to e_X). In the end, e_X is the sum of all such arithmetic expressions for $|F_1|, |F_2|, \ldots, |F_m|$. □

5. LINEAR SPACE FOR $\mathcal{O}(1)$ VARIABLES

Idea

As already shown, the length of the word equation kept by WordEqSat is linear, see Lemma 3.2 and the additional space consumption of WordEqSat is proportional to the storage size of the current equation, see Lemma 4.7. However, the letters in this equation can be all different, even if the input equation is over two letters. Hence the upper bound on the space usage that we can give is (nondeterministic) $\mathcal{O}(n \log n)$ bits. We would like to improve the space consumption to linear; to be more precise, we would like the space consumption to be $\mathcal{O}(m)$ bits, where the input equation used m bits in a natural encoding.[3] We fail in a general case, such a bound is shown only for $\mathcal{O}(1)$ variables (although it holds for arbitrary many occurrences of these variables in the equation, i.e. n_v is not bounded and the alphabet size is arbitrary).

The main obstacle is the encoding of letters introduced by WordEqSat. We show that when we look at the computation of WordEqSat that do not remove the variables from the equation, the space consumption can be limited to $\mathcal{O}(m)$, where m is the storage size (calculated in bits) of the equation at the beginning of the stage. Then for $k = \mathcal{O}(1)$ variables we can consider k stages of WordEqSat, a stage ends when a variable is removed from the equation. In this way the space consumption will be estimated by $c^k m$ bits, which is linear for a constant k.

Encoding of letters. Consider string of explicit letters between two consecutive variables X and Y in $U = V$, together with the variables. During WordEqSat the XwY will be changed to $Xw^{(1)}Y$, $Xw^{(2)}Y$, ... Observe, that 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). It is easy to encode $v^{(i)}$ using only a constant time more bits than w: 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

[3]The proofs given in this section work assuming that each occurrence of a letter (variable) in the input is always given using the same bit representation, however, it is *not* assumed that all letters and/or variables have the representations of the same length, in particular the presented method works also when the input equation is compressed using Huffman coding.

a tree of b as a right subtree; using any usual encoding the size of such representation is only constant times larger than the original text w.

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 exactly as it was in the input equation while the following numbers are encoded in binary. Note that the same code '$(XwY)1$' is (usually) used in each phase, but it denotes different letters in the respective phases.

In this way different occurrences of the same letter a may get different codes: in such case we collect the codes for a and add the information that they all represent the same letter.

Compressing all pairs. This approach raises a new concern: it might be that the length $|x^{(i)}| + |y^{(i)}|$ is non-constant: WordEqSat only guarantees that the length of the whole $|U| + |V|$ is $\mathcal{O}(n)$, but some fragments (i.e. explicit words between variables) may become large. However, for $\mathcal{O}(1)$ variables this can be solved easily, as we can enforce that in one phase *each* pair of consecutive letters is compressed: firstly, a simple preprocessing (to be precise, $\mathsf{Pop}(\Gamma, \Gamma)$) ensures that there are only $\mathcal{O}(k)$ crossing pairs, where k is a number of variables. Then non-crossing pairs are compressed separately (not causing any increase of size of the kept word equations) and each of the crossing pair ab is compressed using $\mathsf{PairComp}(\{a\}, \{b\})$.

Algorithm 12 LinWordEqSat Checking the satisfiability of a word equation in linear space for $\mathcal{O}(1)$ variables

```
1: while $|U| > 1$ or $|V| > 1$ do
2:     BlockCompImp                                   ▷ Block compression
3:     Pop($\Gamma, \Gamma$)                          ▷ The number of crossing pairs is $\mathcal{O}(k)$
4:     $P \leftarrow$ list of non-crossing pairs     ▷ Guess
5:     $P' \leftarrow$ list of crossing pairs        ▷ Guess, at most $2k$ pairs
6:     for $ab \in P$ do
7:         run PairCompNCr($a, b$)
8:     for $ab \in P'$ do                             ▷ $P' \leq 2k$
9:         PairComp($\{a\}, \{b\}$)
10: Solve the problem naively                       ▷ With sides of length 1, the problem is trivial
```

The properties of LinWordEqSat are summarised in the below theorem.

THEOREM 5.1. LinWordEqSat *is sound and complete. For k variables, it runs in (nondeterministic) space of $\mathcal{O}(mk^{ck})$ bits, for some constant c, where m is the space consumption (measured in bits) of the input word equation.*

For the input equation $U = V$ define consecutive *stages*: a stage ends immediately when one variable is removed from the kept equation. Then the next stage starts instantly afterwards. In this way there are at most $k + 1$ stages.

We begin with showing the correctness of LinWordEqSat, the proof is a slight modification of the proof of correctness of WordEqSat, see Lemma 3.4.

LEMMA 5.2. LinWordEqSat *is sound and complete. The kept equation has length $\mathcal{O}(kn)$ in one stage, where equation at the beginning of the stage has length n.*

PROOF. All subprocedures in LinWordEqSat are known to be sound and complete, note that a proper guess of noncrossing pairs in line 4 is needed, as $\mathsf{PairCompNCr}(a, b)$ is complete only for a noncrossing pair ab. Observe that if $a'b'$ is another noncrossing pair to be compressed, then after $\mathsf{PairCompNCr}(a, b)$, when S' is a solution of $U' = V'$ which implements the pair compression for ab, the pair $a'b'$ is noncrossing with respect to S', as none of the first/last letter of any $S(X)$ can be b'/a'. So also LinWordEqSat is sound and complete.

Concerning the space consumption: since we try to compress each crossing pair, a stronger version of Claim 1 can be shown:

Claim 2 (cf. Claim 1). *Let $U = V$ has a solution S. For appropriate choices, the equation $U' = V'$ obtained at the end one stage of* LinWordEqSat *has a solution S' such that*

— *for each pair of two consecutive letters in U (or V), one of these letters is compressed in U' (or V', respectively);*
— *for each pair of two consecutive letters in $S(U)$, one of these letters is compressed in $S'(U')$.*

Proof. Consider any two consecutive letters ab. If $a = b$ then they are compressed by BlockCompImp. If they are not and one of the letters is compressed in BlockCompImp then we are done. Otherwise, ab will be either in P or in P' and we try to compress it. We fail only if one of those two letters was already compressed. □ .

To show the bound on the length of the kept equation, we first estimate that the number of crossing pairs is indeed $\mathcal{O}(k)$. Observe that after $\text{Pop}(\Gamma, \Gamma)$ in line 3 each occurrence of a variable X is preceded (succeeded) by the same letter, say a_X (b_X, respectively). When b (a) is the first (last, respectively) letter of $S(X)$, the X brings only two crossing pairs $a_X b$ and $b_X a$. As there are k different variables, there are at most $2k$ different crossing pairs.

Using a similar argument as in Lemma 3.2, it can be shown that the length of the kept equation is $\mathcal{O}(nk)$, as Pop is run $k+1$ times and BlockCompImp once in one stage and each such run introduces at most $\mathcal{O}(n)$ letters. □

Occurrences of letters

We distinguish two types of occurrences of explicit letters in $U = V$ in one stage: *inner* and *outer* occurrences; note, that the same letter a may have at the same time both an inner and an outer occurrence. Each explicit letter at the beginning of the stage is inner, each letter popped from a variable is outer. When we compress two (or more) inner letters, the result is an inner letter; otherwise the letter that replaced some string is outer. Observe that this implies that each substring w between two variables, say X and Y, can be partitioned into $w = xvy$, where x, y consist solely of outer letters and v consists solely of inner letters (each of x, v, y may be empty). As already noted, the same letter may be encoded in several different ways, this is not a problem, we separately keep a list of different representations of the same letter. Note that this increases the space consumption by a constant.

Inner letters. The inner letters are encoded as follows: when compressing two (or more) inner letters represented as a and b we represent them as (a, b), where '(', ',' and ')' are some appropriately coded symbols; we can think of this as a flattened tree. Note, that in this way when a string of input symbols is compressed into string w', then w' uses only constant time more bits than w.

Lemma 5.3. *The space used for encoding of the inner letters is $\mathcal{O}(m)$, where m is the space (in bits) used for the encoding of the equation at the beginning of the stage.*

The proof is obvious from the above definition.

Outer letters. The outer letters are encoded in a different way: note that if XwY has two different occurrences in $U = V$ then in both of them the outer letters (and inner ones) will be equal in one stage, and so can be encoded using the same symbols.

Lemma 5.4. *Let XwY has two different occurrences in $U = V$ at the beginning of the stage. Then in this stage both those occurrences are represented using the same strings.*

Proof. Observe that the letters popped from a variable depend only on the variable and not on the surrounding letters. Then the string w between those two variables is transformed exactly in the same way in both occurrences. □

We want to encode the outer letters occurring in the string representing XwY as $(XwY)\#(\textit{letter number})$, where '$(XwY)$' is encoded as in the equation at the beginning of the stage and the following 'letter number' is encoded in binary. Lemma 5.4 guarantees that such representations used for different occurrences of XwY are the same, however, we still do not know, how many different such letters are needed. The following lemma shows that $|x|$ and $|y|$ are linear in k, which guarantees that numbers used to denote 'letter number' are also linear in k.

Lemma 5.5. *In one stage, at the beginning of the phase, the maximal substring of outer letters has length $\mathcal{O}(k)$. Furthermore, the space used for the encoding of outer letters in a stage is $\mathcal{O}(km)$, where m is the size of the representation of the equation at the beginning of the stage.*

Proof. As there are $k + 1$ applications of Pop, the length of such block increases by at most $2k + 2$ (it may be expanded from both ends if v is empty). On the other hand, by Claim 2 each substring of length 4 is replaced by a substring of length 3 or less in one phase of LinWordEqSat. This applies to the substrings of outer letters, and similarly as in (1) from Claim 1 it can be shown that these substrings have length $\mathcal{O}(k)$.
As only $\mathcal{O}(k)$ number of different letters per XwY is encoded as outer letters, and each occurrence of a letter encoded as $(XwY)i$ can be charged to an occurrence of XwY at the beginning of the stage, so the space consumption can be bounded as an $\mathcal{O}(k)$ times the consumption at the beginning of the stage. □

Now the proof of Theorem 5.1 follows easily.

Proof of Theorem 5.1. Since the number of different variables is k, there are at most k stages. Note that during one stage the space consumption increases at most ck times, where c does not depend on k, nor n, see Lemma 5.3 and 5.5. Thus, the total space consumption is at most $(kc)^k$ times greater than the one of the input equation.
The correctness follows from Lemma 5.2. □

6. SOLUTIONS OTHER THAN LENGTH MINIMAL

In the next section we give an algorithm generating a (finite) representation of all solutions of a word equation. However, so far we have considered mainly the length minimal solutions, and clearly there are other ones. In this section we recall the classification of solutions, taken from work of Plandowski [2006]. The main result of this classification is the identification of the *minimal solutions*, which have all properties of the length-minimal solutions that we use, except the exponential bound on the exponent of periodicity; however BlockCompImp eliminates the need for this bound, which suggest that this bound is not essential, at least for checking the validity of a word equation. The solutions (substitutions) are classified not by their length, instead we consider whether one solution can be obtained from another using homomorphisms. If so then the former solution is clearly 'more complicated' than the latter.

We first extend the notion of the solution, so that it can include letters that do not occur in the equation: By Γ' we denote the letters that can occur in the solution, even though they do not occur in the equation; formally Γ' is an arbitrary set such that $\Gamma' \cap \Gamma = \emptyset$ (and of course $\Gamma' \cap \mathcal{X} = \emptyset$). Then *substitution* is a morphism $S : \mathcal{X} \cup \Gamma \mapsto (\Gamma \cup \Gamma')^+$ that satisfies the previous assumption that $S(a) = a$ for every $a \in \Gamma$; a notion of the solution generalises to this setting. We call Γ' *free letters* of the solution.

We use the name *operator* to denote functions transforming substitutions. A special class of operators is particularly important for us: given a morphism $\phi : \Gamma \cup \Gamma' \mapsto (\Gamma \cup \Gamma')^+$ by Φ (so capitalised ϕ) we denote a corresponding morphism that acts on substitutions, changing $S(X)$ by ϕ, to be precise $\Phi[S](X) = \phi(S(X))$ and $\Phi[S](a) = a$ for $a \in \Gamma \cup \Gamma'$. For composition of operators we use the usual symbol $\circ$, however, when indexed composition is used, we denote it by $\prod$, for lack of a better symbol.

Definition 6.1 (*cf. [Plandowski 2006]*). A solution $S : \mathcal{X} \cup \Gamma \mapsto (\Gamma \cup \Gamma')^+$ of an equation $U = V$ is a *unifier* (with *free letters* Γ'), when $S(U)$ contains at least one letter from Γ'. S' is an *instance* of a unifier solution S, if $S' = \Phi[S]$ for some non-erasing non-permutating[4] morphism $\phi : (\Gamma \cup \Gamma') \mapsto (\Gamma \cup \Gamma')^+$ that is constant on Γ. A solution S is *minimal*, if it is not a unifier solution, nor an instance of a unifier solution; it is a *minimal unifier* if it is a unifier solution and it is not an instance of another unifier solution.

The assumption that the instance of a unifier solution is obtained by a non-erasing morphism is technical, but it ensures easier and cleaner classification of minimal solutions. We forbid the homomorphism to be a permutation, as we do not want that a solution as its own instance. It is easy to observe that as $\Gamma' \cap \Gamma = \emptyset$, every instance S' of a unifier solution S is a solution (perhaps a unifier one). Note that in general a satisfiable word equation may have no minimal solutions or no minimal unifier solutions.

Example 6.2. Consider an equation $aXb = Y$. Then each $S(X) = w$ and $S(Y) = awb$ is a solution. Then $S(X) = w \in \Gamma$ is length-minimal; when w contains a free letter, then S is a unifier solution, when $w \in \Gamma'$ then this is a minimal unifier solution. There are no minimal solutions.

Consider an equation $aX = Xa$, then each $S(X) = a^n$ is a minimal solution, $S(X) = a$ is a length-minimal one; there are no unifier solutions.

Consider an equation $aXYX^3 = XYaY^2$. Since $S(aXY)$ and $S(XYa)$ have always the same length, this is equivalent to a system of equations $aXY = XYa$ and $X^3 = Y^2$. The former has solutions $X = a^n, Y = a^m$ and the latter ensures that $3n = 2m$. All such solutions are minimal and $S(X) = a^2$, $S(Y) = a^3$ is length-minimal. There are no other solutions, in particular, no unifier solutions.

Typical operators

While in the definition of minimal solutions the operator Φ corresponding to a morphism ϕ is arbitrary, in the proofs we usually see morphisms that are related to pair compression and blocks compression. By $h_{c \to ab}$ denote the morphism which replaces c by ab and is constant on all other letters, the $h_{c \to ab}^{-1}$ is the corresponding inverse morphism (note, that when $a \neq b$ the inverse is well-defined); by bl_a denote the morphism which, for each ℓ, replaces a_ℓ by a^ℓ. Since a block of a can have various partition into subblocks of as, bl_a^{-1} is not well defined. For the purpose of this paper, we specify its action as follows: bl_a^{-1} replaces each a's maximal ℓ-block by a letter a_ℓ. The $H_{c \to ab}$ and Bl_a denote the corresponding operators, $H_{c \to ab}^{-1}$ the inverse operator and Bl_a^{-1} the inverse mapping.

Properties of minimal solutions

As already noted, the minimal solutions inherit most of the crucial properties of length-minimal solutions. In particular, a variant of Lemma 2.4 holds for them.

[4]A morphism ϕ is *non-erasing* if $\phi(a) \neq \epsilon$ for every letter a and it is *non-permutating* if ϕ is not a permutation on its domain.

LEMMA 6.3 (CF. [PLANDOWSKI AND RYTTER 1998, LEMMA 6], CF. LEMMA 2.4). *Let S be a 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 occurrence in U or V and there is a visible occurrence of a^k.*

PROOF. The first claim, which regards a pair ab, is shown using the following fact (we do not assume that S is minimal, as we reuse Claim 3 later on in this more general setting):

CLAIM 3. *If ab, where $a \neq b$, is not an explicit nor a crossing pair for a solution S for $U = V$, then $S' = H^{-1}_{c\to ab}[S]$ for a free letter $c \in \Gamma'$ is a unifier solution of $U = V$. In particular, $S = H_{c\to ab}[S']$ is an instance of S' and so it is not minimal.*

PROOF. Consider $S' = H^{-1}_{c\to ab}[S]$. Since ab is not an explicit nor a crossing pair, each occurrence of ab in $S(U)$ (and $S(V)$) comes from $S(X)$ for some variable X. Thus $S'(U)$ is obtained from $S(U)$ be replacing each ab by c. The same applies to $S(V)$ and $S'(V)$ as well, consequently S' is a solution of $U = V$. Formally:

$$S'(U) = (H^{-1}_{c\to ab}[S])(U) = h^{-1}_{c\to ab}(S(U)) = h^{-1}_{c\to ab}(S(V)) = (H^{-1}_{c\to ab}[S])(V) = S'(V).$$

Since c is a free letter, S' is a unifier solution. Furthermore, as c does not occur in $S(X)$ for any X, then $(H_{c\to ab} \circ H^{-1}_{c\to ab})[S] = S$: indeed, the $H^{-1}_{c\to ab}$ replaces each ab by c in every $S'(X)$, while $H_{c\to ab}$ replaces each c by ab in every $S(X)$. Hence, $S = H_{c\to ab}[S']$ and as $h_{c\to ab}$ is non-erasing, non-permutating and constant on Γ, we conclude that S is an instance of S', which contradicts the assumption that S is minimal. □

Now the first claim of the lemma follows by a contraposition of Claim 3.

Consider now the second claim, which regards the maximal blocks of a. Observe that if a occurs in $S(U)$ and it does not occur in U, nor V, then it is a letter from Γ' and so, by definition, S is a unifier solution and thus cannot be a minimal solution, hence a occurs in U or in V.

To streamline the presentation and analysis, in the remainder of the proof assume that both U and V begin and end with a letter and not a variable; this is easy to achieve by prepending \$ and appending \$′ to both sides of the equation. Alternatively, the cases with variables beginning or ending U or V can be handled in the same way, as the general case.

Consider a maximal a block a^ℓ, for $\ell > 0$ in $S(U)$ and the letter preceding (succeeding) it, say b and c, respectively; by the assumption that U and V begin and end with a letter, such b and c always exist. Consider the occurrences of $ba^\ell c$ in $S(U)$ and $S(V)$. Since $b \neq a \neq c$, these occurrences cannot have overlapping a's (though, if $b = c$, these letters can overlap for different occurrences). Suppose that none of these occurrences is crossing or explicit. Then for each of such occurrences there is a variable Y such that $ba^\ell c$ is wholly contained within some occurrence of $S(Y)$. Change the solution S into S', by replacing each $ba^\ell c$ in each $S(Y)$ by bvc for a free letter v; since a^ℓ in various occurrences of $ba^\ell c$ do not overlap, such replacement is well-defined. Then S' is still a solution, in fact, a unifier solution. Furthermore, S is its instance, contradiction. Hence, there is an explicit or a crossing occurrence (with respect to S) of $ba^\ell c$. Then this occurrence restricted to a^ℓ satisfies the claim of the lemma. □

Minimal unifier solutions

It is already known from the work of Plandowski [2006, Lemma 1] that finding minimal unifier solutions reduces to finding minimal solutions. This is a consequence of the following lemma, which is strengthening of Lemma 6.3 for minimal unifier solutions.

Lemma 6.4 (cf. [Plandowski 2006, Lemma 1]). *If S is a minimal unifier solution with a free letter $v \in \Gamma'$, then for some variables X and Y it holds that v is the first letter of $S(X)$ and v is the last letter of $S(Y)$.*

Proof. The proof is similar to the proof of Lemma 6.3. Suppose that v is not a last letter for any $S(X)$. Consider the right-most occurrence of v in $S(U)$, and let a be the letter directly to the right of this v's occurrences; such a letter exists as v has no occurrence in the equation and is not a last letter in any $S(X)$. The pair va is non-crossing for S (by the assumption) and so by Claim 3 we obtain that $S = H_{b \to va}[S']$ for some fresh letter b and a unifier solution S'. To conclude that S is not minimal, it is left to show that $h_{b \to va}$ is non-erasing (obvious), non-permutating (also true, as $h_{b \to va}(b) = va \notin \Gamma \cup \Gamma'$) and constant on Γ (true, as $b \notin \Gamma$).

Symmetric argument can be given, when v is not a first of some $S(X)$. □

Intuitively Lemma 6.4 yields that search for minimal unifier solutions reduces to looking for minimal solutions: it is enough to 'left-pop' a letter from each variable, in this way the free letters are introduced into the equation and become standard letters. With appropriate nondeterministic guesses, the unifier solutions of $U = V$ will correspond to non-unifier solutions of $U' = V'$. The precise statement needs some additional definitions, which are introduced in the next section; thus the formal statement is deferred to the following section, see Lemma 7.12.

7. REPRESENTATION OF ALL SOLUTIONS

The first PSPACE algorithm for verifying the satisfiability of word equations, PlaSat, was extended by its author to PlaSolve, which returns a finite, graph-like, representation of all finite solutions of a given word equation [Plandowski 2006]. This extension is done in two stages: firstly the original PlaSat is modified into another algorithm PlaSatImp, which also only verifies satisfiability of word equations; then PlaSolve uses PlaSatImp as a subprocedure in generation of a graph representation of *all* finite solutions of a word equation. The modification into PlaSatImp is nontrivial, and its correctness required a separate, involved proof. In this section we show that WordEqSat that uses BlockComplmp instead of BlockComp also can be used to generate a (similar) representation of all solutions of a given word equation.

Representing all solutions. We want to use WordEqSat, which still only verifies satisfiability, as a subprocedure for an algorithm generating a (finite) description of all finite solutions. The approach is simple and in fact is similar to the earlier approach used by Plandowski [2006]: the representation is modelled by a graph, with nodes labelled with equations $U = V$ that are considered by WordEqSat and edges representing transformation performed by WordEqSat. To be precise, if an equation $U = V$ is transformed into $U' = V'$ by WordEqSat (for some nondeterministic choices) we put an edge between nodes labelled by these two equations and label it with an operator that transforms solutions of $U' = V'$ into solutions of $U = V$; furthermore, each solution of $U = V$ can be represented in this way (perhaps by transformation of a solution of some other equation $U'' = V''$, which is obtained from $U = V$ for different non-deterministic choices); note that we do not guarantee that there is a unique way to represent S in such a way. Also, nodes with trivial equations (i.e. $|U| = |V| = 1$) have only one, easy to define, minimal solution (or no solution at all). Concerning the space consumption, since WordEqSat runs in PSPACE, such generation of labelled vertices and edges can also be performed in PSPACE.

Theorem 7.1 (cf. [Plandowski 2006]). *The graph representation of all minimal solutions and minimal unifier solutions of an equation $U = V$ can be constructed in* PSPACE. *The size of the constructed graph is at most exponential.*

As already noted, representing all minimal unifier solution can be reduced to representing all minimal solutions, which will be formally stated in Lemma 7.12. Thus, in the following, we focus on the representation of minimal solutions of a given word equation. We begin with the description of operators that are used to transform the solutions.

Transforming solutions and inverse operators

So far we only know that WordEqSat is sound and complete, however, we do not really know what happens with particular solutions: we do not know how to obtain the solution of the original equation $U = V$ from the transformed equation $U' = V'$, even worse it might be that many of them are somehow lost in the translation. To describe the correspondence of solutions, we strengthen the notions of soundness and completeness (so that they resemble more the notions of implementing the pair compression and block compression).

Given a (nondeterministic) procedure transforming the equation $U = V$ we say that this procedure *transforms the minimal solutions*, if based on the nondeterministic choices and the input equation we can define a family of operators $\mathcal{H}$ such that

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

Note that both $U' = V'$ and $\mathcal{H}$ depend on the nondeterministic choices, so it might be that for different choices we can transform $U = V$ to $U' = V'$ (with $\mathcal{H}'$) and to $U'' = V''$ (with a family $\mathcal{H}''$).

We also say that the equation $U = V$ with its solution S are *transformed into* $U' = V'$ with S' and that $\mathcal{H}$ is the *corresponding family of inverse operators*. In many cases, $\mathcal{H}$ consists of a single operator H, in such case we call it the *corresponding inverse operator*, furthermore, in some cases H does not depend on $U = V$, nor on the nondeterministic choices.

In some cases for an equation $U = V$ and its solution S we explicitly tell, for which nondeterministic choices it is transformed to some other equation $U' = V'$ with a solution S' (intuitively: for the choices that implement the pair compression for S or the block compression).

Our main goal is to show that subprocedures of WordEqSat transform the minimal solutions and to give the appropriate family of operators.

Inverse operators occurring in WordEqSat

We describe the family of inverse operators corresponding to various subprocedures of WordEqSat.

Pair compression. Define an operator $\mathsf{Prepend}_{w,X}$ for a string $w \in (\Gamma \cup \Gamma')^*$, which prepends w to substitution for X and leaves other variables untouched, formally:

$$\mathsf{Prepend}_{w,X}[S](X) = wS(X) \quad \text{and} \quad \mathsf{Prepend}_{w,X}[S](Y) = S(Y), \text{ for } Y \neq X.$$

Define $\mathsf{Append}_{w,X}$ similarly, by appending w to $S(X)$:

$$\mathsf{Append}_{w,X}[S](X) = S(X)w \quad \text{and} \quad \mathsf{Append}_{w,X}[S](Y) = S(Y), \text{ for } Y \neq X.$$

LEMMA 7.2 (CF. LEMMA 2.7). $\mathsf{Pop}(\Gamma_\ell, \Gamma_r)$ *transforms the minimal solutions.*
Suppose it left-popped $b_X \in \Gamma_r \cup \{\epsilon\}$ and right-popped $a_X \in \Gamma_\ell \cup \{\epsilon\}$ from X, then the corresponding inverse operator is:

$$H = \prod_{X \in \mathcal{X}} \mathsf{Append}_{a_X,X} \circ \mathsf{Prepend}_{b_X,X} \ .$$

PROOF. Fix the nondeterministic choices and let H be as defined in the lemma statement. Observe that from the proof of Lemma 2.7 it follows that if S is a solution of $U = V$ (note that we do not need to assume here that S is minimal) then $U' = V'$ has a solution S' such that $S(U) = S'(U')$. Since for each variable X we replaced X with $a_X X b_X$ (or $a_X b_X$, when X was removed from the equation), this means that $S = H[S']$.

On the other hand, when S' is a solution of $U' = V'$ then $S = H[S']$ satisfies $S(U) = S'(U')$: we replaced X with $a_X X b_X$ (or $a_X b_X$, when X was removed from the equation) in $U = V$ and $S(X)$ is obtained exactly by prepending a_X and appending b_X to $S'(X)$. Similarly $S(V) = S'(V')$, which makes S a solution of $U = V$.

It is left to show that if $U = V$ with minimal solution S is transformed to $U' = V'$ with S' then also S' is a minimal solution. Suppose that S' is not minimal, i.e. it is either a unifier solution or an instance of a unifier solution.

it is a unifier solution. Then S' has a free letter. As H only prepends and appends letters, also $S = H[S']$ has a free letter, which makes it a unifier solution,contradicting its minimality.

it is an instance of a unifier solution. Then $S' = \Phi[S'']$ for some unifier solution S'' and non-erasing, non-permutating morphism ϕ which is constant on Γ. Observe that $S = H[\Phi[S'']]$. We claim that $S = \Phi[H[S'']]$: indeed, this follows from the fact that H only prepends and appends letters from Γ, which are not affected by ϕ. Since $H[S'']$ has a free letter, this makes S an instance of a $H[S'']$, contradiction with the minimality of S. □

We now investigate the inverse operator associated with PairComp:

LEMMA 7.3 (CF. LEMMA 2.8). PairComp(Γ_ℓ, Γ_r) *transforms the minimal solutions. To be more precise, for the nondeterministic choices that implement the pair compression for solution S of $U = V$ obtaining $U' = V'$ with a corresponding solution S', the $U = V$ with S is transformed to $U' = V'$ with S'.*

Let H be the inverse operator of the Pop *applied in* PairComp*, furthermore let* PairComp *replaced pairs $ab \in \Gamma_\ell\Gamma_r$ with $c^{(ab)}$. Then the corresponding inverse operator is:*

$$H \circ \prod_{ab \in \Gamma_\ell \Gamma_r} H_{c^{(ab)} \to ab} \quad . \tag{3}$$

Note that as c^{ab} is not in $\Gamma_\ell \cup \Gamma_r$ then the order of applying the $H_{c^{(ab)} \to ab}$ does not matter.

PROOF. Observe that by Lemma 7.2 the application of Pop transforms the minimal solutions; furthermore, the inverse operator for it is well-described. So it is left to consider the compression of pairs performed by PairComp. Such a compression is a composition of many PairCompNCr (and as the pairs are non-overlapping, the order of those compression does not matter), so it is enough to show that when PairCompNCr(a, b) transforms $U = V$ with a minimal S to $U' = V'$ with S' then S' is minimal and $H_{c \to ab}$ is the corresponding inverse operator.

Suppose that S is a solution of $U = V$. Observe that for appropriate non-deterministic choices (done in Pop) the pair ab is noncrossing, see Lemma 2.7. Moreover, the compressions of a pair $a'b'$ cannot make ab crossing, as $a'b'$ do not overlap with ab, the letter that replaced $a'b'$ is not a and not b and lastly no letters are popped from the variables during the compression of non-crossing pairs. Then by Lemma 2.6, PairCompNCr(a, b) implements the pair compression, i.e. the returned equation $U' = V'$ has a solution S' such that $S(U) = h_{c \to ab}(S'(U'))$. Since PairCompNCr$(a, b)$ does not modify variables, this means that $S = H_{c \to ab}[S']$, as claimed.

Suppose that $U = V$ with a minimal solution S is transformed into $U' = V'$ with a solution S', which is not minimal. There are two cases: either S' is a unifier solution, or it is an instance of the unifier solution; we consider only the latter, the former is shown using a similar argument. Then on one hand $S = H_{c\to ab}[S']$ and on the other $S' = \Phi[S'']$ for morphism $\phi : \Gamma \cup \Gamma' \mapsto (\Gamma \cup \Gamma')^+$ which is non-erasing, non-permutating and constant on Γ. Then $S = (H_{c\to ab} \circ \Phi)[S'']$ and it is left to show that $H_{c\to ab} \circ \Phi$ corresponds to some non-erasing non-permutating morphism ϕ'. Define $\phi'(x) = (h_{c\to ab} \circ \phi)(x)$. It is easy to observe that ϕ' is non-erasing and constant on Γ: indeed, both ϕ and $h_{c\to ab}$ are non-erasing and constant on Γ, so their composition is as well. Lastly, it is non-permutating: observe that c is not in the image of ϕ', as it is not in the image of $h_{c\to ab}$, so ϕ' cannot be a permutation.

It is left to show that when S' is a solution of $U' = V'$ returned by $\mathsf{PairCompNCr}(a, b)$ then $S = H_{c\to ab}[S']$ is a solution of $U = V$. Note that $S(U) = h_{c\to ab}(S'(U'))$, as each explicit c in U' was obtained by replacing ab by $\mathsf{PairCompNCr}(a, b)$, while each implicit c in $S'(U')$ was replaced by ab by $H_{c\to ab}$. In the same way $S(V) = h_{c\to ab}(S'(V'))$, which shows that S is a solution of $U = V$. □

Block compression. For block compression BlockCompImp, the family of inverse operators is quite complicated. Instead of introducing it and proving its properties in one go, we choose to make some intermediate steps, which hopefully make a smoother presentation. We begin with describing the family of inverse operators for CutPrefSuff and then give the one for BlockComp; in each of those cases the corresponding family consists of a single operator, similarly as in the case of Pop and PairComp. From Lemma 4.7 we know that a run of BlockCompImp represents several runs of BlockComp, and so in some sense it is a 'parametrised' BlockComp. This approach extends to inverse operators: The family of inverse operators for BlockCompImp represents (in a parametrised way) several inverse operators for different runs of BlockComp. The family is defined using the solutions of the system D created by BlockCompImp, with a single inverse operator corresponding to a solution of the Diophantine system D.

For CutPrefSuff the analysis is exactly the same as for Pop: the inverse operator appends and prepends the letters that were popped by CutPrefSuff.

LEMMA 7.4. CutPrefSuff *transforms the minimal solutions. The corresponding inverse operator is*

$$\prod_{X \in \mathcal{X}} \mathsf{Prepend}_{a_X^{\ell_X}, X} \circ \mathsf{Append}_{b_X^{r_X}, X} \ ,$$

where $a_X^{\ell_X}$ ($b_X^{r_X}$) is the prefix (suffix, respectively) popped from $S(X)$.

PROOF. The proof is similar to the proof of Lemma 7.2 and it is thus omitted. □

This allows stating the result for BlockComp: intuitively it first replaces each a_ℓ with appropriate a^ℓ and then appends and prepends the prefixes and suffixes popped by CutPrefSuff, in a similar way as the inverse operator for PairComp, see Lemma 7.3.

Note that the inverse operator needs to supply the information, which letter is a_ℓ and the value of ℓ (since a_ℓ is just a naming convention that makes the read-up of the paper more accessible, the algorithm does not know which letter 'is' a_ℓ and what 'is' the value of ℓ). Also, at the first glance it seems that the inverse operator could replace an arbitrary number of different letters a_ℓ. Still, as we are interested only in transforming the minimal solutions, this is not the case: by Lemma 6.3 in minimal solutions if a^ℓ occurs in $S(U)$ then a^ℓ is a visible maximal block. Hence, the corresponding inverse operator does not need to introduce blocks of other form.

LEMMA 7.5. BlockComp *transforms the minimal solutions. To be more precise, for the nondeterministic choices that implement the blocks compression for solution S of $U = V$ obtaining $U' = V'$ with a corresponding solution S', the $U = V$ with S is transformed to $U' = V'$ with S'.*

Let H be the inverse operator of the CutPrefSuff*, then the corresponding inverse operator for* BlockComp *is*

$$H \circ \prod_{a \in \Gamma} Bl_a \enspace ,$$

where bl_a replaces a_ℓ with a^ℓ; if a^ℓ replaces a_ℓ then a^ℓ is a visible maximal block in $U = V$ for S. Without loss of generality we may assume that H appends $a_X^{\ell_X}$ and prepends $b_X^{r_X}$, where a_X and b_X are the first and last letter of $S(X)$ and ℓ_X and r_X are the lengths of the a_X prefix and b_X suffix of $S(X)$.

At the first glance, the condition that the inverse operator for BlockComp replaces a_ℓ by a^ℓ only when a^ℓ is a visible block in $U = V$ for S seems bad, as we promised that the inverse operators *do not* depend on the particular solutions, but rather on the equation and the non-deterministic choices (here: of BlockComp). However, after a second thought, there is nothing bad with this: observe that the lengths of the visible blocks are implicitly defined in the nondeterministic choices of CutPrefSuff, as it pops prefixes of length ℓ_X and suffixes of length r_X from variable X and the lengths of the visible blocks linearly depend on $\{\ell_X, r_X\}_{X \in \mathcal{X}}$, see Lemma 4.3. This observation is not formalised: in any case Lemma 7.5 is used only as an intermediate step in description of the inverse operators for BlockCompImp. And in case of BlockCompImp the given inverse operator shall not depend on the solution S (though this formulation of Lemma 7.5 is helpful in the proof).

Proof. By Lemma 7.4 the CutPrefSuff transforms minimal solutions. Furthermore, for appropriate choices, there are no crossing blocks in the obtained $U' = V'$ with respect to S', see Lemma 2.9.

Observe that afterwards BlockComp is a composition of BlockCompNCr(a) for all letters a. The rest of the proof is similar to the one in Lemma 7.3. Furthermore, by Lemma 2.10 we know that in the run of BlockComp that implements the blocks compression the CutPrefSuff pops exactly the a_X-prefix and b_X-suffix from each variable X, where $S(X)$ begins with a_X and ends with b_X, so we can also use those choices when transforming the solution S.

Concerning the restriction of the replaced letters a_ℓ, we show that when a_ℓ is morphed to a^ℓ that is not a maximal block in $S(U)$ then we can in fact remove this part of morphism from the operator and the obtained operator is still the inverse one. When S is a minimal solution, then whenever a^ℓ is a maximal block, it also has a visible occurrence in $U = V$ for S, see Lemma 6.3. As the solution S' of $U' = V'$ corresponds to the implementation of block compression, the maximal blocks in $S(U)$ and $S(V)$ are obtained by replacing single letters in $S'(U')$ and $S'(V')$ by blocks. Hence, suppose that the inverse operator morphs a letter b_k to b^k and b^k is not a visible maximal block in $U = V$ for S. Hence b^k is not a maximal block in $S(U)$. In particular, b_k does not occur in $S'(U')$, so we can remove this morphed pair from the inverse operator. □

Now we are ready to define the family of operators for BlockCompImp. Intuitively, it will encode several inverse operators associated with different nondeterministic choices of BlockComp in a compact way: instead of making explicit guesses about the lengths of the prefixes and suffixes of $S(X)$, it will parametrise them using variables x_X and y_X. On the other hand $S'(U')$ contains letters, which represent blocks of letters and lengths of those blocks linearly depend on x_X and y_X. The coherence of all these lengths is guaranteed by an appropriate system of Diophantine equations (exactly as in the case of BlockCompImp).

Recall that for an arithmetic expression e_i in variables $\{x_X, y_X\}_{X \in \mathcal{X}}$ the $e_i[\{\ell_X, r_X\}_{X \in \mathcal{X}}]$ denotes the value of e_i when $\{\ell_X, r_X\}_{X \in \mathcal{X}}$ are substituted for $\{x_X, y_X\}_{X \in \mathcal{X}}$.

Suppose that BlockCompImp constructs a linear Diophantine system D in variables $\{x_X, y_X\}_{X \in \mathcal{X}}$, and popped a prefix $a_X^{x_X}$ of a_X and suffix $b_X^{y_X}$ of b_X from X. (Note that D depends on the nondeterministic choices of BlockCompImp.) Then we define a (finite or infinite) family of inverse operators: let $\{e_i\}_{i=1}^m$ be the lengths of parametrised visible maximal blocks for the coherent prefix-suffix structure (in variables $\{x_X, y_X\}_{X\in\mathcal{X}}$). By the definition the sides of equations in D are those expressions. Then, for a solution $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ of D the following operator is in a family of operators $\mathcal{H}_D$:

$$\left(\prod_{X\in\mathcal{X}} \mathsf{Prepend}_{a_X^{\ell_X},X} \circ \mathsf{Append}_{b_X^{r_X},X}\right) \circ \prod_{a\in\Gamma} Bl_a \ , \tag{4}$$

where bl_a replaces the letter a_{e_i} by $a^{e_i[\{\ell_X,r_X\}_{X\in\mathcal{X}}]}$, and no other letters are replaced. Note that the operator needs to explicitly point the letters that it treats as a_{e_i} as well as the arithmetic expressions e_i (so that it can calculate $e_i[\{\ell_X, r_X\}_{X\in\mathcal{X}}]$): the index e_i in a_{e_i} is just a notation convention to make the read-up of the paper easier, the actual letter does not carry any information about a nor e_i.

LEMMA 7.6. *Let* BlockCompImp *return a satisfiable linear Diophantine system D. Then* BlockCompImp *transforms the solutions and $\mathcal{H}_D$ is the corresponding family of inverse operators.*

PROOF. Consider an equation $U = V$ and its solution S. We first want to show that for some choices of BlockCompImp the obtained equation $U' = V'$ has a solution S' such that $S = H[S']$ for some $H \in \mathcal{H}_D$, where $\mathcal{H}_D$ is the corresponding family of inverse operators, defined in (4). To this end we use the fact that BlockComp transforms the minimal solution and that its corresponding inverse operator is known, see Lemma 7.5.

By Lemma 7.5 BlockComp implements the blocks compression, consider the corresponding run for S and $U = V$ (obtaining $U' = V'$ with a corresponding S'). By the same lemma we know that for those very choices BlockComp transforms the $U = V$ with S to $U' = V'$ with S'. Consider on the other hand the run of BlockCompImp in which WordtoDioph returns a Diophantine system D that is S-coherent. Then by Lemma 4.7 this run leads to the same instance $U' = V'$ (up to renaming of letters). So it is left to show that the appropriate inverse operator is in $\mathcal{H}_D$.

The inverse operator H for BlockComp first replaces letters a_ℓ with a^ℓ, where a^ℓ is a visible maximal block in $U = V$ for S, and then appends $a_X^{\ell_X}$ and prepends $b_X^{r_X}$ to each variable X, where ℓ_X and r_X are the lengths of the a_X-prefix and b_X-suffix of $S(X)$. Observe that by Lemma 4.5 the $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ is a solution of D. Note that H is in $\mathcal{H}_D$: when a^ℓ is the maximal block E_i then by Lemma 4.3 the corresponding parametrised maximal block has length e_i such that $|E_i| = e_i[\{\ell_X, r_X\}]$ and the inverse operator corresponding to $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ replaces a_{e_i} with $a^{e_i[\{\ell_X,r_X\}]} = a^{|E_i|} = a^\ell$ and then prepends $a_X^{\ell_X}$ and appends $b_X^{r_X}$ to substitution for X.

We now show that if an equation $U = V$ is transformed by BlockCompImp into $U' = V'$ which has a solution S' and $H \in \mathcal{H}_D$ then $S = H[S']$ is a solution of $U = V$. Let us first recall, how $U' = V'$ is obtained from $U = V$ and how H looks like.

By definition, BlockCompImp first guesses the prefix-suffix structure for $U = V$ (i.e. what is the first and last letter of $S(X)$ and whether $S(X)$ is a block of letters) pops the $a_X^{\ell_X}$ prefix and $b_X^{r_X}$-suffix from X for each $X \in \mathcal{X}$, guesses system D coherent with the prefix-suffix structure (whose sides are lengths of parametrised explicit maximal blocks) and replaces blocks whose lengths are equalised in the system by a single letter a_{e_i} (where e_i is one of the lengths of the equalised blocks). Then H corresponds to a solution $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ of D: it first replaces a_{e_i} by $a^{e_i[\{\ell_X,r_x\}_{X\in\mathcal{X}}]}$ and then prepends $a_X^{\ell_X}$ and appends $b_X^{r_X}$ to $S(X)$ for each $X \in \mathcal{X}$.

To show that $S = H[S']$ is a solution of $U = V$ we show that $S(U)$ is obtained from $S'(U')$ by replacing each a_{e_i} with $a^{e_i[\{\ell_X,r_x\}_{X\in\mathcal{X}}]}$; the same will hold for $S(V)$ and in this way S is shown to be a solution of $U = V$. To this end we define an intermediate substitution S_1 and an equation $U_1 = V_1$: the S_1 is obtained from S' similarly as S, but without the appending and prepending the letters, just by replacing a_{e_i} with $a^{e_i[\{\ell_X, r_x\}_{X \in \mathcal{X}}]}$. Similarly, define U_1 and V_1 by replacing those letters in U' and V'. Then clearly $S_1(U_1)$ is $S'(U')$ with each a_{e_i} replaced with $a^{e_i[\{\ell_X, r_x\}_{X \in \mathcal{X}}]}$.

Since appending $a_X^{\ell_X}$ and prepending $b_X^{r_X}$ to $S_1(X)$ for each $X \in \mathcal{X}$ turns S_1 to S, by reversing the procedure we obtain that popping $a_X^{\ell_X}$ to the left and $b_X^{r_X}$ to the right from each $X \in \mathcal{X}$ turns S to S_1. To finish the proof we show that when we pop $a_X^{\ell_X}$ to the left and $b_X^{r_X}$ to the right, we turn $U = V$ to $U_1 = V_1$. To this end we just need to show that the consecutive maximal blocks of letters in $U = V$ after the popping are the same as in $U_1 = V_1$. Since we pop exactly the prefixes and suffixes, the former are exactly the visible maximal blocks in $U = V$ for S, which by Lemma 4.3 have lengths $e_i[\{\ell_X, r_X\}]$. On the other hand, the maximal blocks in $U_1 = V_1$ are obtained by substitutions for single letters in $U' = V'$ and are of lengths $e_i[\{\ell_X, r_X\}]$, which ends the proof. □

Representation of a single minimal solution

We now show that each *minimal* equation can be obtained by retracing the steps of some successful run of WordEqSat. It turns out that during this retracing we can restrict ourselves to equations that are short and inverse operators that morph, prepend and append only letters actually present in the equation. We define these notions formally, they are used again the definition of the graph representation of all solutions.

Definition 7.7. We say that a word equation $U = V$ is *proper* if $U, V \in (\Gamma \cup \mathcal{X})^*$, in total U and V have at most n_v occurrences of variables (where n_v is the number of occurrences of variables in the input equation) and $|U| + |V| \leq cn$ (for an appropriately chosen in advance constant c). An equation is *trivial* if both its sides have length at most 1.

A family $\mathcal{H}$ of inverse operators (corresponding to the transformation of $U = V$ to $U' = V'$) is *proper* if it is one of the families defined in (3) or in (4). Furthermore $H \in \mathcal{H}$ morphs only letters present in $U' = V'$ and appends/prepends only letters that occur in $U' = V'$ or images of such letters by the morphing in H.

The intuition is as follows: the first type of edges, labelled with (3), corresponds to PairComp in WordEqSat. The second, labelled with (4), corresponds to BlockCompImp.

Note that the assumption that proper families of operators may only append, prepend and morph letters that are present in the equation is a restriction on the potential form of inverse operators from families (3) and (4) and it is shown in Lemma 7.8 below that indeed such restricted families are enough to describe all minimal solutions.

On the other hand such a restriction makes it easier to describe such families: proper families of inverse operators need to specify, which letters are replaced, but in both cases they list at most cn letters for replacement (as this is the size of the equation). Furthermore, the family of inverse operators (4) needs also to specify the expression e_i for each of the letters it intends to replace. Since there are at most cn such letters, there are also at most cn such expressions. So the whole description size is polynomial.

LEMMA 7.8. *If $U_0 = V_0$ is of size n and has a minimal solution S_0 then there exists a sequence of proper equations $U_0 = V_0$, $U_1 = V_1$, ..., $U_m = V_m$ such that*

— *$V_m = U_m$ is trivial and $m = \mathcal{O}(\log |S_0(U_0)|)$;*
— *for appropriate nondeterministic choices a subprocedure (*BlockCompImp *or* PairComp*) of* WordEqSat *transforms an equation $U_{i-1} = V_{i-1}$ with a minimal solution S_{i-1} to $U_i = V_i$ with a minimal solution S_i;*
— *the corresponding inverse operator is from a proper family.*

PROOF. Lemma 7.3 and Lemma 7.6 guarantee that PairComp and BlockCompImp transform minimal solutions, so there exist a sequence $U_0 = V_0$, $U_1 = V_1$, ..., $U_m = V_m$ together with minimal solutions S_0, S_1,..., S_m such that

— $U_{i-1} = V_{i-1}$ is transformed to $U_i = V_i$ by some subprocedure (BlockCompImp or PairComp) of WordEqSat, where $\mathcal{H}_i$ is the corresponding family of inverse operators;
— $S_{i-1} = H_i[S_i]$ for some $H_i \in \mathcal{H}_i$;
— $U_m = V_m$ is trivial.

What is not known is whether:

(Q 1) each $U_i = V_i$ is proper?
(Q 2) $m = \mathcal{O}(\log(|S_0(U_0)|))$?
(Q 3) each $\mathcal{H}_i$ is proper?

As soon as we settle (Q 1)–(Q 3), the proof is complete.

The (Q 3) is easy: since S_i is minimal, by definition it assigns only letters from $U_i = V_i$, so there is no reason for H_i to morph any other letters (as they are simply not in $S_i(U_i)$); also, by Lemma 7.2 the inverse operator for Pop prepends and appends letters that were popped from variables, i.e. either they are present in $U_{i+1} = V_{i+1}$ or some letters in $U_{i+1} = V_{i+1}$ replaced pairs that includes such a letter. a similar argument holds for the inverse operator for BlockCompImp, see Lemma 7.6. This shows that indeed each $\mathcal{H}_i$ is proper and so establishes (Q 3).

Concerning (Q 1) note that we do not introduce any new occurrences of variables, so we just need to bound the lengths of equations $U_0 = V_0$, $U_1 = V_1$, ..., $U_m = V_m$; such a bound was already given in Lemma 3.2, though it was not guaranteed there that the appropriate minimal solution is transformed.

Similarly, for (Q 2) the length of the successful computation is given in Lemma 3.3, but again nothing is known about transforming minimal solutions.

Both proofs of Lemma 3.2 and Lemma 3.3 rely on Claim 1, the following stronger version of Claim 1 gives all the needed properties:

CLAIM 4. *Let $U = V$ have a minimal solution S. For appropriate choices, during one phase it is transformed to an equation $U' = V'$ with a minimal solution S' such that*

— *$1/6$ of letters in $S(U)$ (rounding down) is compressed in $S'(U')$.*
— *at least $(|U| + |V| - 3n_v - 4)/6$ of letters in U or V are compressed in U' or V';*

Now, the proof for (Q 1) follows in the same way as in Lemma 3.2: we choose the sequence of equations $U_0 = V_0$, $U_1 = V_1$, ..., $U_m = V_m$ guaranteed to exist by Claim 4 . As in Lemma 3.2 it can be shown that each such an equation has size at most $79n$ and the intermediate equations have length at most $85n$. Similarly, the same run guarantees that the size of the corresponding solution S_i shrinks by a constant factor at the beginning of each phase (so every three equations).

It is left to show Claim 4. To this end note that Claim 1 shows that the two shortening properties for a solution S hold for the non-deterministic choices that implement the block compression and pair compression for this solution (for appropriate partition of letters). But Lemma 7.6 shows that for a solution S the non-deterministic choices in BlockComp that implement the block compression also transform the minimal solution; similar claim holds for PairComp by Lemma 7.3. Which ends the proof. □

Representation of all minimal solutions

As already said, the set of minimal solutions will be represented by a directed graph. Intuitively, the graph $\mathcal{G}$ represents all paths from Lemma 7.8.

Definition 7.9. Directed graph $\mathcal{G}$ for a satisfiable input equation $U_0 = V_0$ has

— nodes labelled with satisfiable proper equations;
— edges of $\mathcal{G}$ are labelled with a family of proper operators;
— an edge from $U = V$ to $U' = V'$ is labelled with $\mathcal{H}$ if and only if for some nondeterministic choices WordEqSat (that uses BlockCompImp instead of BlockComp) transforms $U = V$ into $U' = V'$ and $\mathcal{H}$ is the corresponding family of inverse operators;
— each node is reachable from node labelled with $U_0 = V_0$.

We say that S for $U = V$ is *obtained by a path* to $U' = V'$ with S' if S is obtained by applying a composition of inverse operators on the path from $U = V$ to $U' = V'$ applied to S'.

We need to show that on one hand $\mathcal{G}$ can be constructed in PSPACE, and on the other, that it describes all minimal solutions of a word equation. We begin with the latter.

LEMMA 7.10. *Let S be a minimal solution of an equation $U = V$ (of size n) that is a node in $\mathcal{G}$. Then there is a path in $\mathcal{G}$ starting in $U = V$ and ending in a trivial equation $U' = V'$ with a minimal solution S' such that S can be obtained from S' by this path.*

Moreover, each S obtained in this way is a solution of $U = V$.

If the equation $U = V$ is trivial (i.e. $|U|, |V| \leq 1$) then there is at most one minimal solution, which is easy to describe.

PROOF. Concerning the third claim, observe that describing all minimal solution of a satisfiable equation $U = V$ such that $|U| = |V| = 1$ is easy:

— if a variable X does not occur in the equation, then $S(X) = \epsilon$ for each minimal solution X, so it is enough to consider variables that occur in $U = V$;
— if $U, V \in \Gamma$ then either there is one minimal solution ($S(X) = \epsilon$ for each $X \in \mathcal{X}$), when $U = V$, or none solution, when $U \neq V$;
— if one of the sides is a letter and the other a variable, then there is only one solution;
— if both sides consist of variables, then there is no minimal solution (as each solution is an instance of a unifier solution that assigns v to variables on both sides).

If $U = \epsilon$ or $V = \epsilon$, then a similar analysis shows that if $U = V = \epsilon$ then the unique minimal solution assigns ϵ to each variable and otherwise there is no minimal solution.

For the second claim, observe that it summarizes the properties of subprocedures of WordEqSat, presented in Lemma 7.3 and 7.6, which claim that PairComp and BlockCompImp transform minimal solutions and the definition of the graph $\mathcal{G}$.

For the first claim observe that this is just a reformulation of Lemma 7.8. □

Constructing the representation of all minimal solutions. We show that it is possible to generate $\mathcal{G}$ within PSPACE. In order to do so we should be able to decide in PSPACE whether:

node label check. a given proper equation $U = V$ labels a node in $\mathcal{G}$.
edge label check. given two satisfiable proper equations $U = V$ and $U' = V'$ and a proper family $\mathcal{H}$ of operators there is an edge from $U = V$ to $U' = V'$ labelled with $\mathcal{H}$.

When PSPACE procedures for these two tasks are known, constructing $\mathcal{G}$ is easy:

— we iterate over all proper equations (they have length at most cn), for a fixed equation $U = V$ we check whether $U = V$ labels a node in $\mathcal{G}$. If so, we output it.
— we iterate over all pairs of proper equations $U = V$ and $U' = V'$ and every proper family of operators $\mathcal{H}$ labelling this edge; note that both (3) and (4) can be described using $\mathcal{O}(n)$ symbols. For a fixed triple we verify, whether there is an edge from $U = V$ to $U' = V'$ labelled with $\mathcal{H}$. If so, we output triple $(U = V, \mathcal{H}, U' = V')$.

Clearly this procedure uses only polynomial space and properly generates $\mathcal{G}$.

It is thus left to show that node and edge label checks can be performed in PSPACE.

Lemma 7.11. *Node and label checks can be performed in* PSPACE.

Proof. Consider first node label check. It is trivial to verify, whether $U = V$ has length at most cn and at most n_v variables' occurrences. Using WordEqSat we can verify in PSPACE, whether $U = V$ is satisfiable. Also, in NPSPACE we can verify, whether WordEqSat transforms $U_0 = V_0$ to $U = V$: we begin with $U_0 = V_0$ and transform it using WordEqSat until $U = V$ is obtained. As by Lemma 3.2 WordEqSat uses $\mathcal{O}(n \log n)$ space, this is doable in NPSPACE. As NPSPACE=PSPACE, we are done.

In a similar way we can show that edge label check can be performed in NPSPACE. Firstly using node label check we verify, whether both $U = V$ and $U' = V'$ label nodes of $\mathcal{G}$. The label $\mathcal{H}$ uniquely identifies the subprocedure of WordEqSat that should be applied to $U = V$ in order to obtain $U' = V'$; we thus take $U = V$ and apply this subprocedure and check whether we obtain $U' = V'$. As WordEqSat uses $\mathcal{O}(n \log n)$ space, by Lemma 3.2 and Lemma 4.7, then this can be tested in NPSPACE and consequently edge label check can be executed in PSPACE $\square$

Minimal unifier solutions. The presented procedures are enough to construct a finite representation of all minimal solutions. It is left to formalise the transformation of minimal unifier solutions to minimal solutions. Since by Lemma 6.4 we know that S is a minimal unifier solution with a free letter a then this free letter is a first letter of some $S(X)$. So when we left-pop the first letter of each variable, we introduce all free letters used by S into the equation, making them usual letters and turning the minimal unifier solution S into a minimal solution S' of the new equation.

Lemma 7.12. *Let S be a minimal unifier solution of $U = V$. Then for some nondeterministic choices* $\mathsf{Pop}(\Gamma', \emptyset)$ *returns an equation $U' = V'$ such that $S = \Phi[S']$ for some minimal solution S' of $U' = V'$, where*

$$\Phi = \prod_{X \in \mathcal{X}} \mathsf{Prepend}_{a_X, X} \enspace ,$$

where a_X is the symbol left-popped from a variable X, i.e. $a_X \in \Gamma' \cup \epsilon$.

Proof. We know from Lemma 7.2 that Pop transforms minimal solutions and the given operator Φ is the corresponding inverse operator. In fact, the proof in the direction we use does not assume that S is minimal, it can be an arbitrary solution. Hence for a minimal unifier solution S and appropriate guesses the $U = V$ with S is transformed by Pop into $U' = V'$ with S', such that $S = \Phi[S']$. To be more precise, the guesses are consistent with S, in the sense that Pop left-pops a letter a_X if and only if the first letter of $S(X)$ is a_X and $a_X \in \Gamma'$. So it is left to show that if S is a minimal unifier solution, then S' is a minimal solution.

We first show that $S'(U')$ has no free letters. By Lemma 6.4, if a is a free letter in S, then a is the first letter of some $S(X)$. And we fixed the nondeterministic choices for which Pop left-pops a from X, and so a occurs as an explicit letter in $U' = V'$. As we choose a arbitrarily, all free letters of $S(U)$ occur in $U' = V'$. As $S'(U') = S(U)$, by Lemma 2.7, the S' has no free letters and so it is a non-unifier solution.

Suppose that S' is not minimal, as the case that it is a unifier solution is already excluded, this happens only in the case when S is an instance of some other unifier solution, i.e. there is a unifier solution S'' such that $S' = \Phi'[S'']$ for some non-erasing, non-permutating morphism $\phi' : (\Gamma \cup \Gamma') \cup \Gamma'' \mapsto (\Gamma \cup \Gamma')^+$ which is constant on $\Gamma \cup \Gamma'$ (the set Γ'' is a new set of free letters, such that $\Gamma'' \cap (\Gamma' \cup \Gamma) = \emptyset$). We show that this contradicts the assumption that S is a minimal unifier solution. Since Φ prepends letters from $\Gamma \cup \Gamma'$, which are not affected by ϕ', it can be concluded that $S = \Phi'[\Phi[S']]$:

$$\begin{aligned} S &= \Phi[S'] && \text{by definition} \\ &= \Phi[\Phi'[S'']] && \text{by a contrario assumption on } S' \\ &= \Phi'[\Phi[S'']] && \text{as } \phi' \text{ does not affect letters added by } \Phi \ . \end{aligned}$$

It is left to show that $\Phi[S'']$ is a unifier solution: but S'' contains letters from Γ'' and Φ only prepends letters to $S''(X)$, hence $\Phi[S'']$ is a unifier solution. As we already know that ϕ' is non-erasing and non-permutating, and constant on $\Gamma \cup \Gamma'$, we conclude that S is an instance of $\Phi[S'']$, which contradicts the assumption that it is minimal. □

Now we are ready to give the proof of the representation Theorem 7.1 on generating the finite representation of minimal unifier solutions of a word equation.

Proof of Theorem 7.1. Consider first a graph representation of minimal solutions of an equation. Each node has at most polynomial description, so does the edges. By Lemma 7.11 it can be checked in polynomial space, whether a node is present in the graph and whether an edge (labelled) joins two nodes. Since the description of a node (edge) has polynomial size, there are at most exponentially many nodes (edges, respectively).

In order to generate a graph representation of all minimal and unifier-minimal solutions, we use the approach presented in Lemma 7.12. Given an equation $U = V$ we iterate over equations $U' = V'$ of length at most $n+n_v$ and using at most n_v free letters from Γ', whether $\mathsf{Pop}(\Gamma', \emptyset)$ can transform $U = V$ into $U' = V'$. If so, output the node labelled with $U' = V'$ and make a graph representation of all its minimal solutions. The label on the edge from $U = V$ to $U' = V'$ is the inverse operator returned by Pop, see Lemma 7.12. Clearly, this procedure still runs in PSPACE, and so also the generated graph has exponential size. □ .

8. OTHER THEORETICAL PROPERTIES

In this section, we give (alternative) proofs of two known theoretical properties of word equations, using the approach of recompression: an exponential bound on the periodicity bound and the doubly-exponential bound on the size of the length-minimal solution.

Exponential bound on exponent of periodicity

As already described in the introduction, exponential bound on exponent of periodicity, shown by Kościelski and Pacholski [1996], is one of the most often used results on words equations. Their proof follows by first considering so-called P-presentations of a string; roughly, given a string w and a primitive word P, a P-presentation is a canonical factorisation of w into powers of P and other strings. Then each power of P is associated with a number and treating such numbers as variables leads to a system of satisfiable Diophantine equations. Solutions of this system induced solutions of the word equation. In particular, length-minimal solution corresponded to minimal (in some sense) solution of the Diophantine equation. This is similar to results presented in Section 4, but considering P-presentations instead of letters makes the argument much more involved.

Using known results on minimal solutions of Diophantine equations and some simple calculus, an exponential upper-bound on the exponent of periodicity was shown. The last step of this proof, i.e. the estimation of the minimal solution, was relatively easy, while both the P-presentations and reduction from P-presentations to a system of equations were involved.

We now show that using local recompression one can obtain exponential upper bound on exponent of periodicity relatively easy.

Exponent of periodicity. We begin with a bit more detailed definition of the exponent of periodicity.

Definition 8.1. 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. The notion of exponent of periodicity is naturally transferred from strings to equations: For an equation $U = V$, define the exponent of periodicity as

$$\mathrm{per}(U = V) = \max_S \left[\mathrm{per}(S(U))\right] \ ,$$

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.

We show that an exponential upper bound on Γ-exponent of periodicity is easy and natural to obtain, one can think of it as a restriction of Kościelski and Pacholski [1996] original proof to its last part, i.e. to estimation of the minimal solution of a system of Diophantine equations. Then we show that the compression applied in WordEqSat basically preserves the exponent of periodicity, in particular it reduces the calculation of upper bound on $\mathrm{per}(U = V)$ to calculation of upper bound on $\mathrm{per}_\Gamma(U = V)$.

Minimal solutions of linear Diophantine systems. Consider a system of m linear Diophantine equations in r variables $x_1, \ldots, x_r$, written as

$$\sum_{j=1}^{r} n_{i,j} x_j = n_i \qquad \text{for } i = 1, \ldots, m \tag{5a}$$

together with inequalities guaranteeing that each x_i is positive

$$x_j \geq 1 \qquad \text{for } j = 1, \ldots, r \ . \tag{5b}$$

In the following, we are interested only in *natural* solutions, i.e. the ones in which each component is a natural number; observe that inequality (5b) guarantees that each of the component is greater than zero. We introduce a partial ordering on such solutions:

$$(q_1, \ldots, q_r) \geq (q'_1, \ldots, q'_r) \quad \text{if and only if} \quad q_j \geq q'_j \text{ for each } j = 1, \ldots, r.$$

A solution $(q_1, \ldots, q_r)$ is a *minimal* if it satisfies (5) and there is no solution smaller than it. (Note, that there may be incomparable minimal solutions.)

It is known, that each component of the minimal solution is at most exponential:

LEMMA 8.2 (CF. [KOŚCIELSKI AND PACHOLSKI 1996, COROLLARY 4.4]). *For a system of linear Diophantine equations* (5) *let* $w = r + \sum_{i=1}^{m} |n_i|$ *and* $c = \sum_{i=1}^{m} \sum_{j=1}^{r} |n_{i,j}|$. *If* $(q_1, \ldots, q_r)$ *is its minimal solution, then* $q_j \leq (w + r)e^{c/e}$.

The proof is a slight extension of the original proof of Kościelski and Pacholski, which takes in to the account also the inequalities. For completeness, we recall its proof, as given in [Kościelski and Pacholski 1996].

PROOF PROOF, CF. [KOŚCIELSKI AND PACHOLSKI 1996]. The proof follows by estimation based on work of von zur Gathen and Sieveking [1978] and independently by Lambert [1987]

CLAIM 5 ([VON ZUR GATHEN AND SIEVEKING 1978]; [LAMBERT 1987]). *Consider a (vector) equations and inequalities* $Ax = B$, $Cx \geq D$ *with integer entries in* A, B, C *and* D. *Let* M *be the upper bound on the absolute values of the determinants of square submatrices of the matrix* $\begin{pmatrix} A \\ C \end{pmatrix}$, r *be the number of variables and* w *the sum of absolute values of elements in* B *and* D. *Then for each minimal natural solution* $(q_1, \ldots, q_r)$ *of* (5), *for each* $1 \leq i \leq r$ *we have* $q_i \leq (w + r)M$. $\square$

So it remains to estimate M from Claim 5. Observe that as the matrix C in our case is an identity, it is enough to consider the bound on the values of determinants of square submatrices of $(n_{i,j})$, which was done by Kościelski and Pacholski [1996], the rest of the proof is a simple recollection of their argument.

Recall the Hadamard inequality: for any matrix $N = (n_{i,j})_{i,j=1}^k$ we have

$$\det^2(N) \leq \prod_{j=1}^{k} \sum_{i=1}^{k} n_{i,j}^2 \ .$$

Therefore

$$\begin{aligned}
\det(N) &\leq \left(\prod_{j=1}^{k} \sum_{i=1}^{k} n_{i,j}^2 \right)^{1/2} && \text{Hadamard inequality}\\
&\leq \left(\prod_{j=1}^{k} \left(\sum_{i=1}^{k} |n_{i,j}| \right)^2 \right)^{1/2} && \text{trivial}\\
&= \prod_{j=1}^{k} \sum_{i=1}^{k} |n_{i,j}| && \text{simplification}\\
&\leq \left(\frac{\sum_{j=1}^{k} \left(\sum_{i=1}^{k} |n_{i,j}| \right)}{k} \right)^k && \text{inequality between means}\\
&\leq \left(\frac{c}{k} \right)^k && \text{by definition } \sum_{j=1}^{k} \sum_{i=1}^{k} |n_{i,j}| = c\\
&\leq e^{c/e} && \text{calculus: sup at } k = c/e.
\end{aligned}$$

Taking N to be any submatrix of $(n_{i,j})$ yields that $M \leq e^{c/e}$ and consequently $q_i \leq (w + r)e^{c/e}$, as claimed. □

Now from Lemma 8.2 it can be easily concluded that

LEMMA 8.3. *In each minimal solution of the small system of linear Diophantine equations for word equation $U = V$ all coordinates are $\mathcal{O}((|U| + |V|)e^{2n_v/e})$.*

PROOF. Recall that by the definition of the small system of linear Diophantine equations (for a word equation $U = V$), this system has

— at most twice as many variables as $U = V$, (so $r \leq 2n_v$ in terms of Lemma 8.2);
— the sum of coefficients at variables (so c in the terms of Lemma 8.2) is $2n_v$;
— the sum of values of constants of the equalities and inequalities (so w in the terms of Lemma 8.2) is $2(|U|+|V|+n_v)$ (i.e. $2(|U|+|V|)$ for equations and $2n_v$ for the inequalities).

Hence from Lemma 8.2 it follows that each coordinate of a minimal solution of a small system of linear Diophantine equations is at most

$$2(|U| + |V| + n_v)e^{2n_v/e} = \mathcal{O}((|U| + |V|)e^{2n_v/e}) \ ,$$

as claimed □

From Lemma 8.3 we can infer the upper-bound on the Γ-exponent of periodicity of the length-minimal solution of the word equation.

LEMMA 8.4 (CF. [KOŚCIELSKI AND PACHOLSKI 1996], CF. LEMMA 2.1). *Consider a solution S of a word equation $U = V$, the S-coherent Diophantine system D and its solution $\{\ell_X, r_X\}_{X \in \mathcal{X}}$ and the corresponding induced solutions $S[\{\ell_X, r_X\}_{X \in \mathcal{X}}]$. For a length-minimal S' among them the Γ-exponent of periodicity of $S'(U)$ is $\mathcal{O}(n_v(|U| + |V|e^{2n_v/e}))$, while $\mathrm{per}_\Gamma(S'(X))$ for any variable X is $\mathcal{O}((|U| + |V|)e^{2n_v/e})$.*

PROOF. By Lemma 4.9 all solutions $S[\{\ell_X, r_X\}_{X \in \mathcal{X}}]$ are similar. Let, as in the statement, S' be a length minimal among them, let it correspond to a solution $\{\ell'_X, r'_X\}_{X \in \mathcal{X}}$ of D. Then by definition ℓ'_X, (r'_X) are the lengths of the a_X-prefix (b_X-suffix) of $S'(X)$. We show that $\{\ell'_X, r'_X\}_{X \in \mathcal{X}}$ is a minimal solution of D: suppose for the sake of contradiction that it is not. Then there is a solution $\{\ell''_X, r''_X\}_{X \in \mathcal{X}}$ of D, such that

$$\ell''_X \leq \ell'_X \quad \text{and} \quad r''_X \leq r'_X \quad \text{for each } X \in \mathcal{X} \tag{6}$$

and at least one of those inequalities is strict, without loss of generality let $\ell''_Y < \ell'_Y$. By Lemma 4.9 for each variable X there is an arithmetic expression e_X such that $|S'(X)| = e_X[\{\ell'_X, r'_X\}_{X \in \mathcal{X}}]$ and $|S''(X)| = e_X[\{\ell''_X, r''_X\}_{X \in \mathcal{X}}]$. By (6) we obtain that $|S'(X)| \geq |S''(X)|$ for each variable (note that by the definition each coefficient in the arithmetic expression is non-negative). Furthermore, Lemma 4.9 also guarantees that each e_X depends on x_X and y_X (if y_X is used at all), hence by the choice of Y also $|S'(Y)| > |S''(Y)|$ and so S' is not length-minimal, contradiction.

Then by the minimality of $\{\ell'_X, r'_X\}_{X \in \mathcal{X}}$ we obtain that each ℓ'_X and r'_X is $\mathcal{O}((|U| + |V|)e^{2n_v/e})$, by Lemma 8.3. As the maximal a block is a concatenation of explicit letters from the equation and a_X-prefixes and b_X-suffixes of $S(X)$ for various X, its length is at most

$$\left(\sum_{X \in \mathcal{X}} \ell_X + r_X\right) + (|U| + |V| - n_v) = \mathcal{O}(n_v(|U| + |V|)e^{2n_v/e}) \ ,$$

which ends the proof. □

As a short corollary we obtain:

THEOREM 8.5 (CF. [KOŚCIELSKI AND PACHOLSKI 1996], CF. LEMMA 2.1). *The Γ-exponent of periodicity of a word equation $U = V$ with n_v occurrences of variables is $\mathcal{O}(n_v(|U| + |V|e^{2n_v/e}))$.*

General exponent of periodicity. So far we have only shown that Γ-exponent of periodicity is at most exponential. However judging by the work of Kościelski and Pacholski [1996], the difficulty is elsewhere, in the case of exponent of periodicity for longer words. We show that this is not the case: in the following lemma we show that employing the recompression technique we obtain an exponential bound on the exponent of periodicity as a corollary of a similar bound for Γ-exponent of periodicity. Unfortunately, our result is weaker than the one obtained by Kościelski and Pacholski, as they in fact had a 2^{cn} bound, for appropriate c.

LEMMA 8.6. *Let $U = V$ with a solution S be transformed by some subprocedure of* WordEqSat, *i.e.* PairComp *or* BlockComp *(or* BlockCompImp*) into $U' = V'$ with S'. Then $\mathrm{per}(S'(U')) \leq \mathrm{per}(S(U))$. Furthermore*

$$\mathrm{per}(S(U)) = \mathrm{per}_\Gamma(S(U)),\ \textit{or} \tag{per 1}$$

$$\mathrm{per}(S'(U')) \geq \mathrm{per}(S(U)) - 1 \ . \tag{per 2}$$

PROOF. Recall that by Lemmata 2.7 and 2.9 for Pop and CutPrefSuff it holds that $S(U) = S'(U')$ and so the claim trivially holds, as $\mathrm{per}(S'(U')) = \mathrm{per}(S(U))$. So it is enough to show the claim for PairComp and BlockComp restricted to compression (the analysis for BlockCompImp is the same).

We first show that $\text{per}(S(U)) \geq \text{per}(S'(U'))$ for PairComp. By Lemma 7.3 the corresponding inverse operator (when we restrict ourselves to compression) is $\prod_{ab\in\Gamma_\ell\Gamma_r} H_{c^{(ab)}\to ab}$, that is, each $c^{(ab)}$ is replaced with appropriate c in each substitution for X. Hence $S(U) = \prod_{ab\in\Gamma_\ell\Gamma_r} h_{c^{(ab)}\to ab}(S'(U'))$. Let w^k be a substring of $S'(U')$, then $\prod_{ab\in\Gamma_\ell\Gamma_r} h_{c^{(ab)}\to ab}(w^k)$ replaces each such $c^{(ab)}$ independently and so $(\prod_{ab\in\Gamma_\ell\Gamma_r} h_{c^{(ab)}\to ab}(w))^k$ is a substring of $S(U)$, hence $\text{per}(S'(U')) \leq \text{per}(S(U))$.

Similarly, BlockComp is a composition of CutPrefSuff, which preserves the exponent of periodicity. Hence it is enough to consider the inverse operator for BlockComp restricted to the compression. By Lemma 7.5 it is $\prod_{a\in\Gamma} Bl_a$, where bl_a replaces each a_ℓ with a^ℓ for some letters a_ℓ. Hence $S(U) = \prod_{a\in\Gamma} bl_a(S'(U'))$. Consider any w^k that is a substring of $S'(U')$. Then $\prod_{a\in\Gamma} bl_a(S'(w^k)) = (\prod_{a\in\Gamma} bl_a(S'(w)))^k$ is a substring of $S(U)$. Thus $\text{per}(S'(U')) \leq \text{per}(S(U))$.

We move to the second claim of the lemma, i.e. we are going to show that (per 1) or (per 2) holds. Let $m = \text{per}(S(U))$. If there is $a \in \Gamma$ such that a^m is a substring of $S(U)$, then (per 1) holds. So assume that w^m is a substring of $S(U)$, for some $w \notin \Gamma \cup \{\epsilon\}$. Moreover, we can assume that $w \neq a^k$ for every a and k, as this clearly reduces to the case of $w = a$.

The idea of the rest of the argument is simple: when $\text{per}(S(U)) = m$ then there is a substring w^m in $S(U)$. In the ideal case each cop of w in w^m is compressed in the same way and so in $S'(U')$ we have a corresponding substring w'^m. However, it can be that compressions occur in between different w's in w^m. However, we can choose a substring w_1^{m-1} within w^m such that each copy of w_1 is compressed in the same way. The choice of w_1 depends on whether the first and last letter of w are going to be compressed or not.

Consider first $\text{PairComp}(\mathit{Letters}_1, \mathit{Letters}_2)$, and let $w = bua$, recall that by the assumption $|w| > 1$ and so $|u| \geq 0$, i.e. it can be that $u = \epsilon$. How does the image of w^m looks like in $S'(U')$? This depends on whether $a \in \mathit{Letters}_1$ and whether $b \in \mathit{Letters}_2$, in total there are four cases. From Lemma 7.3 we know that for $h^{-1} = \prod_{a'\in \mathit{Letters}_\ell, b'\in \mathit{Letters}_r} h^{-1}_{c\to a'b'}$ we have $S'(U') = h^{-1}(S(U)))$.

$b \notin \mathit{Letters}_2$. The further analysis depends on whether $a \in \mathit{Letters}_1$ or not

- $a \notin \mathit{Letters}_1$. Consider any $w = bua$ in w^m. Observe that by case assumptions, the first letter of w is never compressed with letter to the left and the last letter is never compressed with the letter to the right. So in this case w^m after compression will be represented as $(h^{-1}(w))^m$, and so $(h^{-1}(w))^m$ is a substring of $h^{-1}(S(U))$; thus, $\text{per}(S'(U')) \geq \text{per}(S(U))$.
- $a \in \mathit{Letters}_1$. Consider $w^m = (bua)^m$. As in the previous case, the leading b is never compressed with the previous letter In this case it might be that the last letter of w^m is compressed with the following letter, however, each other last a in bua is not (as the following letter is $b \notin \mathit{Letters}_2$). Hence the compression of the prefix w^{m-1} results in $(h^{-1}(w))^{m-1}$ and so $\text{per}(S'(U')) \geq \text{per}(S(U)) - 1$.

$b \in \mathit{Letters}_2$. Similarly, the further analysis depends on whether $a \in \mathit{Letters}_1$ or not

- $a \notin \mathit{Letters}_1$. The case is symmetric to the subcase above, in which $a \in \mathit{Letters}_1$ and $b \notin \mathit{Letters}_2$, in particular in a similar way we show that $\text{per}(S'(U')) \geq \text{per}(S(U)) - 1$.
- $a \in \mathit{Letters}_1$. Represent w^m as $b(uab)^{m-1}ua$. Observe that each ab in $(uab)^{m-1}$ is compressed and replaced with a new letter c. Furthermore, the first letter in each u in $(uab)^{m-1}$ is not compressed with the letter to the left, as this is in each case $b \in \mathit{Letters}_2$. Hence, $(uab)^{m-1}$ is compressed into $(h^{-1}(uab))^{m-1}$ and so $\text{per}(S'(U')) \geq \text{per}(S(U)) - 1$.

The analysis for BlockComp (and similarly BlockCompImp) is even simpler: let $w = b^\ell u a^r$, where u does not begin with b and does not end with a. By Lemma 7.5 we know that for $bl^{-1} = \prod_{a \in \Gamma} bl_a^{-1}$ we have $S'(U') = bl^{-1}(S(U))$ Then

$$w^m = (b^\ell u a^r)^m = b^\ell (u a^r b^\ell)^{m-1} u a^r .$$

As by the assumption u does not begin with b nor end with a, hence each copy of $ua^r b^\ell$ in $S(U)$ is compressed in the same way and so $S'(U')$ contains $bl^{-1}((ua^r b^\ell)^{m-1}) = (bl^{-1}(ua^r b^\ell))^{m-1}$, and so $\text{per}(S'(U')) \geq m-1$, as claimed. □

As a promised corollary we obtain the exponential bound on the exponent of periodicity.

Theorem 8.7 (cf. [Kościelski and Pacholski 1996], cf. Lemma 2.1). *The exponent of periodicity of equation of a length-minimal solution S is single exponential in $|U| + |V|$.*

Proof. Denote $U = V$ and some its length-minimal solution S by $U_1 = V_1$ and S_1. Let $U_1 = V_1$, $U_2 = V_2$, …, $U_m = V_m$ be all equations generated during the run of WordEqSat, in this order, let S_1 be transformed to S_2, …, S_m during this run, and let ϕ_2, …, ϕ_m be the corresponding inverse operators. We claim that if S_1 is length-minimal then for each i we have that

$$\text{per}_\Gamma(S_i(U_i)) = \mathcal{O}(n_v(|U_i| + |V_i|)e^{2n_v/e}) \ . \tag{7}$$

Suppose that this is not the case. Consider the S_i-coherent system of Diophantine equations, let S_i correspond to $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ and consider some minimal solution $\{\ell'_X, r'_X\}_{X\in\mathcal{X}}$ that is not larger than $\{\ell_X, r_X\}_{X\in\mathcal{X}}$. Then by Lemma 4.9 the $S'_i = S_i[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$ is also a solution, which is similar to S_i. As those solutions are similar, S'_i can be obtained by deleting some letters from S_i. Then $(\phi_2 \circ \phi_3 \circ \cdots \circ \phi_i)[S'_i]$ is also a solution of $U_1 = V_1$ which is shorter than $S = (\phi_2 \circ \phi_3 \circ \cdots \circ \phi_i)[S_i]$, contradiction.

None of the equations $U_1 = V_1$, $U_2 = V_2$, …, $U_m = V_m$ is repeated, and as each of them is of length at most $c'n$ (see Lemma 3.2), thus $m \leq (c'n)^{c'n+1} \leq n^{cn}$, for some constants c and c'. By Lemma 8.6 it holds that $\text{per}(S_i(U_i)) = \text{per}_\Gamma(S_i(U_i))$ or $\text{per}(S_i(U_i)) \leq \text{per}(S_{i+1}(U_{i+1}))+1$. Observe that for $S_m(U_m)$ we have $\text{per}(S_m(U_m)) = 1$: since $|U_m|, |V_m| \leq 1$ we have two cases:

— if any of U_m or V_m is ϵ then $S_m(U_m) = \epsilon$ and $\text{per}(S_m(U_m)) = 0$;
— if one of U_m or V_m is a letter, say a, then $S_m(U_m) = a$ and clearly $\text{per}(S_m(U_m)) = 1$;
— if both U_m and V_m are variables, say U_m is X and V_m is Y (we do not assume that $X \neq Y$). Suppose that $S_m(X) = S_m(Y)$ is longer than one letter, say it is aw. Consider S'_m, where $S'_m(X) = S'_m(Y) = w$ and it is equal to S_m otherwise. Then $S'_1 = (\phi_2 \circ \phi_3 \circ \cdots \circ \phi_m)[S'_m]$ is also a solution of $U_1 = V_1$ and $S'_1(U_1)$ is shorter than $S_1(U_1)$, as $S_1 = (\phi_2 \circ \phi_3 \circ \cdots \circ \phi_m)[S_m]$. This contradicts the assumption that S_1 is length minimal. Hence $S_m(X) = S_m(Y)$ has only one letter and so $\text{per}(S_m(U_m)) = 1$.

Let i be the smallest index among $1, 2, \ldots, m$ such that $\text{per}_\Gamma(S_i(U_i)) = \text{per}(S_i(U_i))$. Note that such an i exists, as m satisfies this condition. Recall that by (7)

$$\text{per}_\Gamma(S_i(U_i)) \leq c' n_v(|U_i| + |V_i|)e^{2n_v/e}$$

for some constant c'. By (per 2) for $j = 1, 2, \ldots, m-1$ we have $\text{per}(S_j(U_j)) \leq \text{per}(S_{j+1}(U_{j+1})) + 1$ we conclude that

$$\begin{aligned}\text{per}(S_1(U_1)) &\leq (i-1) + c' n_v(|U_i| + |V_i|)e^{2n_v/e} \\ &= \mathcal{O}(n^{cn} + n_v n e^{2n_v/e}) \\ &= \mathcal{O}(n^{cn}) \ ,\end{aligned}$$

for some constant c, in particular it is single exponential in $n = |U_1| + |V_1|$. Since this holds for an arbitrary length-minimal solution S_1 of $U_1 = V_1$, this yields the claim. $\square$

Double exponential bound on minimal solutions

It was shown by Plandowski [1999] that the size of the length minimal solution of word equation is always doubly exponential. This result was achieved by careful and clever analysis of factorisations of minimal solutions; the proof is basically independent from the analysis of the PlaSat, though uses similar types of factorisations of words (and in fact the doubly-exponential bound can be inferred from PlaSat after some simple modifications [Plandowski 2012]).

Since we know that on one hand the running time of WordEqSat is polynomial in n and $\log N$ (see Theorem 3.1) on the other the space consumption is $\mathcal{O}(n \log n)$ (see Lemma 4.7), the doubly exponential upper bound on $\log N$ seems natural. However, both presented bounds are upper bounds and so cannot be directly compared. To compare them we want to show that the running time is in fact also *lower-bounded* in terms of $\log N$.

LEMMA 8.8. *Let N be the size of the length-minimal solution of a word equation of size n. Then the number of phases of* WordEqSat *is $\Omega(\log N/\mathsf{poly}(n))$ for every accepting run, regardless of the nondeterministic choices.*

PROOF. Suppose that the equation $U = V$ is transformed into an unsatisfiable equation $U' = V'$; then we are done, as it will never be turned into a satisfiable instance. So in the following we consider only the case, in which each of the equations is satisfiable.

The solution S' of $U' = V'$ is obtained from S of $U = V$ by two separate compression sub-phases: in the first, some maximal blocks of letters may be compressed into one letter, in the second, some pairs ab, for $a \neq b$ are replaced by a fresh letter (see Lemmata 7.5 and 7.3). In the following we shall compare the lengths of the length-minimal solutions before and after one such compression subphase, i.e. estimate N/N', where N and N' are the lengths of the length-minimal solutions before and after the subphase, respectively.

We begin with the second phase, as it is easier to analyse. Notice, that if c is introduced as a letter for a pair ab then c is not compressed in the rest of this subphase, hence at most two letters are compressed into one and those new letters are not further compressed. Let S' be a length minimal solution of $U' = V'$, i.e. of size N'. Take the solution S that is transformed into S' by pair compression (we know that it exists, by Lemma 7.3). Then $S(U)$ is obtained from $S'(U')$ by replacing each such c by the corresponding ab, in particular, $S(U)$ is at most twice as long as $S'(U')$. Also, $S(U)$ is not longer than the length-minimal solution of $U = V$, i.e. of length at most N. Thus

$$\begin{aligned}\frac{N}{N'} &= \frac{N}{|S'(U')|}\\ &\leq \frac{|S(U)|}{|S'(U')|}\\ &\leq 2 \ .\end{aligned}$$

Hence, the second compression subphase shortens the shortest solution by a factor of at most 2.

Let us return to the first sub-phase, the idea of the analysis is similar as in the first case. Consider again any length-minimal solution S' of $U' = V'$, let its length be N'. Take any solution S that is transformed into S' by BlockCompImp. Consider the S-coherent system of Diophantine equations D and the solutions $S[\{\ell_X, r_X\}_{X\in\mathcal{X}}]$ induced by different solutions of D, see Lemma 4.9. Take the length-minimal among them, let it be S_1. Then its Γ-exponent of periodicity is $\mathcal{O}((|U| + |V|)e^{2n_v/e})$ by Lemma 8.4. Now, note that as S and S_1 are similar, the application of block compression to $S(U)$ and $S_1(U)$ results in a string of the same length: similar solutions have the same number of maximal blocks and each of those blocks is replaced with a single letter. As the former is $S'(U')$, we get that $S'(U')$ is $\mathcal{O}((|U| + |V|)e^{2n_v/e})$ times shorter than $S_1(U)$. Consequently

$$\begin{aligned}\frac{|S_1(U)|}{N'} &= \frac{|S_1(U)|}{|S'(U')|}\\ &\leq cne^{2n_v/e} \ .\end{aligned}$$

Since $|S_1(U)| \geq N$, where N is the length of the length-minimal solution of $U = V$, we obtain that

$$\begin{aligned}\frac{N}{N'} &\leq \frac{|S_1(U)|}{|S'(U')|}\\ &\leq cn_v e^{2n_v/e} \ .\end{aligned}$$

Taking into the account the factor 2 in the second sub-phase we obtain the upper bound

$$2cne^{2n_v/e}$$

on the proportion between length minimal solutions in the consecutive phases.

So let $N = N_1, N_2, \ldots, N_m$ be the lengths of length-minimal solutions in consecutive phases, where m is the last phase. Then $N_i/N_{i+1} \leq 2cne^{2n_v/e}$ and $N_m \leq 1$, hence

$$N \leq (2cne^{2n_v/e})^m$$

and so

$$m \geq \frac{\log N}{\mathsf{poly}(n)} \ ,$$

as claimed. □

COROLLARY 8.9 (CF. [PLANDOWSKI 1999]). *The size of the length-minimal solution of a word equation of size n is at most $2^{q(n)\cdot n_v^{cn_v}}$ for some polynomial q and constant c.*

PROOF. By Lemma 3.2 the equation stored by WordEqSat has at most $cn_v^{cn_v} \log n$ many phases. On the other hand, by Lemma 8.8, there are at least $c'(\log N)/p(n)$ phases, for some constant c' and polynomial p. Thus,

$$c'(\log N)/p(n) \leq cn_v^{cn_v} \log n \ ,$$

which yields the claim. □

Acknowledgements

I would like to thank A. Okhotin for his remarks about ingenuity of Plandowski's result, which somehow stayed in my memory; P. Gawrychowski for initiating my interest in compressed membership problems and compressed pattern matching, exploring which eventually led to this work as well as for pointing to relevant literature [Lohrey and Mathissen 2011; Mehlhorn et al. 1997]; J. Karhumäki, for his explicit question, whether the techniques of local recompression can be applied to the word equations; last not least, W. Plandowski for his numerous comments and suggestions on the manuscript as well as questions concerning the exact space consumption that eventually led to results in Section 5.

This work was partially supported by NCN grant number 2011/01/D/ST6/07164, 2011–2015.

REFERENCES

Stephen Alstrup, Gerth Stolting Brodal, and Theis Rauhe. 2000. Pattern Matching in Dynamic Texts. In *SODA*. ACM/SIAM, 819–828. DOI:http://dx.doi.org/10.1145/338219.338645

Volker Diekert, Artur Jeż, and Wojciech Plandowski. 2014. Finding All Solutions of Equations in Free Groups and Monoids with Involution. In *CSR (LNCS)*, Edward A. Hirsch, Sergei O. Kuznetsov, Jean-Éric Pin, and Nikolay K. Vereshchagin (Eds.), Vol. 8476. Springer, 1–15. DOI:http://dx.doi.org/10.1007/978-3-319-06686-8_1

Robert Dąbrowski and Wojciech Plandowski. 2004. Solving Two-Variable Word Equations. In *ICALP (LNCS)*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.), Vol. 3142. Springer, 408–419. DOI:http://dx.doi.org/10.1007/978-3-540-27836-8_36

Robert Dąbrowski and Wojciech Plandowski. 2011. On Word Equations in One Variable. *Algorithmica* 60, 4 (2011), 819–828. DOI:http://dx.doi.org/10.1007/s00453-009-9375-3

Claudio Gutiérrez. 1998. Satisfiability of Word Equations with Constants is in Exponential Space. In *FOCS*. IEEE Computer Society, 112–119. DOI:http://dx.doi.org/10.1109/SFCS.1998.743434

Lucian Ilie and Wojciech Plandowski. 2000. Two-variable word equations. *ITA* 34, 6 (2000), 467–501. DOI:http://dx.doi.org/10.1051/ita:2000126

Joxan Jaffar. 1990. Minimal and Complete Word Unification. *J. ACM* 37, 1 (1990), 47–85.

Artur Jeż. 2013. Approximation of grammar-based compression via recompression. In *CPM (LNCS)*, Johannes Fischer and Peter Sanders (Eds.), Vol. 7922. Springer, 165–176. DOI:http://dx.doi.org/10.1007/978-3-642-38905-4_17 full version at http://arxiv.org/abs/1301.5842.

Artur Jeż. 2014a. The Complexity of Compressed Membership Problems for Finite Automata. *Theory of Computing Systems* 55 (2014), 685–718. Issue 4. DOI:http://dx.doi.org/10.1007/s00224-013-9443-6

Artur Jeż. 2014b. Context unification is in PSPACE. In *ICALP (LNCS)*, Elias Koutsoupias, Javier Esparza, and Pierre Fraigniaud (Eds.), Vol. 8573. Springer, 244–255. DOI:http://dx.doi.org/10.1007/978-3-662-43951-7_21 full version at http://arxiv.org/abs/1310.4367.

Artur Jeż. 2014c. A *really* simple approximation of smallest grammar. In *CPM (LNCS)*, Sergei Kuznetsov, Alexander Kulikov, and Pavel Pevzner (Eds.), Vol. 8486. Springer, 182–191. full version at http://arxiv.org/abs/1310.4367.

Artur Jeż. 2014d. One-Variable Word Equations in Linear Time. *Algorithmica* (2014). DOI:http://dx.doi.org/10.1007/s00453-014-9931-3 accepted and available online.

Artur Jeż. 2015. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms* 11, 3 (Jan. 2015), 20:1–20:43. DOI:http://dx.doi.org/10.1145/2631920

Artur Jeż and Markus Lohrey. 2014. Approximation of smallest linear tree grammar. In *STACS 2014 (LIPIcs)*, Ernst W. Mayr and Natacha Portier (Eds.), Vol. 25. Schloss Dagstuhl — Leibniz-Zentrum fuer Informatik, 445–457.

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

Markku Laine and Wojciech Plandowski. 2011. Word Equations with One Unknown. *Int. J. Found. Comput. Sci.* 22, 2 (2011), 345–375. DOI:http://dx.doi.org/10.1142/S0129054111008088

J. L. Lambert. 1987. Une borne pour les générateurs des solutions entières positives d'une équation diophantienne linéaire. *Compte-rendu de L'Académie des Sciences de Paris* 305, 1 (1987), 39–40.

N. Jesper Larsson and Alistair Moffat. 1999. Offline Dictionary-Based Compression. In *Data Compression Conference*. IEEE Computer Society, 296–305. DOI:http://dx.doi.org/10.1109/DCC.1999.755679

Markus Lohrey and Christian Mathissen. 2011. Compressed Membership in Automata with Compressed Labels. In *CSR (LNCS)*, Alexander S. Kulikov and Nikolay K. Vereshchagin (Eds.), Vol. 6651. Springer, 275–288. DOI:http://dx.doi.org/10.1007/978-3-642-20712-9_21

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

Kurt Mehlhorn, R. Sundar, and Christian Uhrig. 1997. Maintaining Dynamic Sequences under Equality Tests in Polylogarithmic Time. *Algorithmica* 17, 2 (1997), 183–198. DOI:http://dx.doi.org/10.1007/BF02522825

Wojciech Plandowski. 1994. Testing Equivalence of Morphisms on Context-Free Languages. In *ESA (LNCS)*, Jan van Leeuwen (Ed.), Vol. 855. Springer, 460–470. DOI:http://dx.doi.org/10.1007/BFb0049431

Wojciech Plandowski. 1999. Satisfiability of Word Equations with Constants is in NEXPTIME. In *STOC*. ACM, 721–725. DOI:http://dx.doi.org/10.1145/301250.301443

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

Wojciech Plandowski. 2006. An efficient algorithm for solving word equations. In *STOC*, Jon M. Kleinberg (Ed.). ACM, 467–476. DOI:http://dx.doi.org/10.1145/1132516.1132584

Wojciech Plandowski. 2012. personal communication. (2012).

Wojciech Plandowski and Wojciech Rytter. 1998. Application of Lempel-Ziv Encodings to the Solution of Word Equations. In *ICALP (LNCS)*, Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel (Eds.), Vol. 1443. Springer, 731–742. DOI:http://dx.doi.org/10.1007/BFb0055097

Hiroshi Sakamoto. 2005. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms* 3, 2-4 (2005), 416–430. DOI:http://dx.doi.org/10.1016/j.jda.2004.08.016

Klaus U. Schulz. 1990. Makanin's Algorithm for Word Equations—Two Improvements and a Generalization. In *IWWERT (LNCS)*, Klaus U. Schulz (Ed.), Vol. 572. Springer, 85–150. DOI:http://dx.doi.org/10.1007/3-540-55124-7_4

Joachim von zur Gathen and Malte Sieveking. 1978. A bound on solutions of linear integer equations and inequalities. *Proceedings of AMS* 72, 1 (1978), 155–158.