One-Variable Word Equations in Linear Time*

Artur Jeż[1,2,**]

[1] Max Planck Institute für Informatik,
Campus E1 4, DE-66123 Saarbrücken, Germany
[2] Institute of Computer Science, University of Wrocław,
ul. Joliot-Curie 15, PL-50383 Wrocław, Poland
aje@cs.uni.wroc.pl

Abstract. In this paper we consider word equations with one variable (and arbitrary many appearances of it). A recent technique of recompression, which is applicable to general word equations, is shown to be suitable also in this case. While in general case it is non-deterministic, it determinises in case of one variable and the obtained running time is $\mathcal{O}(n)$ (in RAM model).

Keywords: Word equations, string unification, one variable equations.

1 Introduction

Word Equations. The problem of satisfiability of word equations was considered as one of the most intriguing in computer science. The first algorithm for it was given by Makanin [11] and his algorithm was improved several times, however, no essentially different approach was proposed for over two decades.

An alternative algorithm was proposed by Plandowski and Rytter [16], who presented a very simple algorithm with a (nondeterministic) running time polynomial in n and $\log N$, where N is the length of the length-minimal solution. However, at that time the only bound on such length followed from Makanin's work and it was triply exponential in n.

Soon after Plandowski showed, using novel factorisations, that N is at most doubly exponential [14], proving that satisfiability of word equations is in NEXPTIME. Exploiting the interplay between factorisations and compression he improved the algorithm so that it worked in PSPACE [15]. On the other hand, it is only known that the satisfiability of word equations is NP-hard.

One Variable. Constructing a cubic algorithm for the word equations with only one variable (and arbitrarily many appearances of it) is trivial. First nontrivial bound was given by Obono, Goralcik and Maksimenko, who devised an $\mathcal{O}(n \log n)$ algorithm [13]. This was improved by Dąbrowski and Plandowski [2] to $\mathcal{O}(n + \#_X \log n)$, where $\#_X$ is the number of appearances of the variable

* The full version of this paper is available at http://arxiv.org/abs/1302.3481
** This work was supported by Alexander von Humboldt Foundation.

F.V. Fomin et al. (Eds.): ICALP 2013, Part II, LNCS 7966, pp. 324–335, 2013.

in the equation. The latter work assumed that alphabet Σ is finite or that it can be identified with numbers. A general solution was presented by Laine and Plandowski [9], who gave an $\mathcal{O}(n \log \#_X)$ algorithm in a simpler model, in which the only operation on letters is their comparison.

Recompression. Recently, the author proposed a technique of *recompression* based on previous techniques of Mehlhorn et. al [12], Lohrey and Mathissen [10] and Sakamoto [17]. This method was successfully applied to various problems related to grammar-compressed strings [5,3,4]. Unexpectedly, this approach was also applicable to word equations, in which case alternative proofs of many known results were obtained [6].

The technique is based on iterative application of two replacement schemes performed on the text t:

pair compression of ab For two different letters a, b such that substring ab appears in t replace each of ab in t by a fresh letter c.

a's block compression For each maximal block a^ℓ, where a is a letter and $\ell > 1$, that appears in t, replace all a^ℓs in t by a fresh letter a_ℓ.

In one phase, pair compression (block compression) is applied to all pairs (blocks, respectively) that appeared at the beginning of this phase. Ideally, each letter is compressed and so the length of t halves, in a worst-case scenario during one phase t is still shortened by a constant factor.

The surprising property is that such a schema can be efficiently applied even to grammar-compressed data [5,3] or to text given in an implicit way, i.e. as a solution of a word equation [6]. In order to do so, local changes of the variables (or nonterminals) are needed: X is replaced with $a^\ell X$ (or Xa^ℓ), where a^ℓ is a prefix (suffix, respectively) of substitution for X. In this way the solution that substitutes $a^\ell w$ for X is implicitly replaced with one that substitutes w.

Recompression and One-Variable Equations. As the recompression works for general word equations, it can be applied also to restricted subclasses. In the general case it relies on the nondeterminism, however, when restricted to one-variable equations it determinises. A simple implementation has $\mathcal{O}(n+\#_X \log n)$ running time, see Section 3. Adding a few heuristics, data structures and applying a more sophisticated analysis yields an $\mathcal{O}(n)$ running time, see Section 4.

Outline of the Algorithm. We present an algorithm for one-variable equation based on the recompression; it also provides a compact description of all solutions of such an equation. Intuitively: when pair compression is applied, say ab is replaced by c (assuming it *can* be applied), then there is a one-to-one correspondence of the solutions before and after the compression, this correspondence is simply exchange of all abs by cs and vice-versa. The same applies to the block compression. On the other hand, the modification of X can lead to loss of solutions (for technical reasons we do not consider the solution ϵ): when X is to

be replaced with $a^\ell X$ then each solution of the form $a^\ell w$ has a corresponding solution w, but solution a^ℓ is lost in the process. So before the replacement, it is tested whether a^ℓ is a solution and if so, it is reported. The testing is performed by on-the-fly evaluation of both sides under substitution $X = a^\ell$ and comparing the obtained strings letter by letter until a mismatch is found or both strings end.

It is easy to implement the recompression so that one phase takes linear time. The cost is distributed to explicit words between the variables, each such w is charged $\mathcal{O}(|w|)$. If such w is long enough, its length decreases by a constant factor in one phase, see Lemma 8. Thus, such cost is charged to the lost length and sums to $\mathcal{O}(n)$ in total. However, this is not true when w is short (in particular, of constant length). In this case we use the fact that there are $\mathcal{O}(\log n)$ phases and in each phase such cost is at most $\mathcal{O}(\#_X)$ (i.e. proportional to the number of explicit words in total).

Using the following heuristics as well as more involved analysis the running time can be lowered to $\mathcal{O}(n)$ (see Section 4 for some details):

- We save space used for problematic 'short' words between the variables (and thus time needed to compress them in a phase): instead of storing multiple copies of the same short string we store it once and have pointers to it in the equation. Additionally we prove that those short words are substrings of 'long' words, which allows a bound on the sum of their lengths.
- when we compare $Xw_1Xw_2 \dots w_mX$ from one side of the equation with its copy appearing on the other side, we make such a comparison in $\mathcal{O}(1)$ time (using suffix arrays);
- the $(Xu)^m$ and $(Xu')^{m'}$ (under substitution for X) are compared in $\mathcal{O}(|u| + |u'|)$ time instead of naive $\mathcal{O}(m \cdot |u| + m' \cdot |u'|)$, using simple facts from combinatorics on words.

Furthermore a more insightful analysis shows that problematic 'short' words in the equation invalidate several candidate solutions. This allows a tighter estimation of the time spent on testing the solutions.

Model. To perform the recompression efficiently, an algorithm for grouping pairs is needed. When we identify the symbols in Σ with consecutive numbers, this is done using RadixSort in linear time[1]. Thus, all (efficient) applications of recompression technique make such an assumption. On the other hand, the second of the mentioned heuristics craves checking substring equality in $\mathcal{O}(1)$, to this end a suffix array [7] with a structure for answering *longest common prefix query* (lcp) [8] is employed on which we use range minimum queries [1]. The last structure needs the flexibility of the RAM model to run in $\mathcal{O}(1)$ time per query.

[1] The RadixSort runs in time linear in number of numbers plus the universe size. Since we introduce numbers in each phase, it might be that the latter is much larger than the equation length. However, after each phase in linear time we can replace the letters appearing in the equation so that they constitute an interval of numbers, which yields that the RadixSort has indeed linear running time.

2 Preliminaries

One-Variable Equations. Consider a word equation $\mathcal{A} = \mathcal{B}$ over one variable X, by $|\mathcal{A}| + |\mathcal{B}|$ we denote its length and n is the initial length of the equation. Without loss of generality one of $\mathcal{A}$ and $\mathcal{B}$ begins with a variable and the other with a letter [2]: If they both begin with the same symbol (be it letter or nonterminal), we can remove this symbol from them, without affecting the set of solutions; if they begin with different letters, this equation clearly has no solution. The same applies to the last symbols of $\mathcal{A}$ and $\mathcal{B}$. Thus, in the following we assume that the equation is of the form

$$A_0 X A_1 \dots A_{n_\mathcal{A}-1} X A_{n_\mathcal{A}} = X B_1 \dots B_{n_\mathcal{B}-1} X B_{n_\mathcal{B}} \ , \tag{1}$$

where $A_i, B_j \in \Sigma^*$ are called (*explicit*) *words*, $n_\mathcal{A}$ ($n_\mathcal{B}$) denotes the number of appearances of X in $\mathcal{A}$ ($\mathcal{B}$, respectively). A_0 (*first word*) is nonempty and exactly one of $A_{n_\mathcal{A}}$, $B_{n_\mathcal{B}}$ (*last word*) is nonempty. If this condition is violated for any reason, we greedily repair by cutting letters from appropriate strings.

A *substitution* S assigns a string to X, we extend S to $(X \cup \Sigma)^*$ with an obvious meaning. A *solution* is a substitution such that $S(\mathcal{A}) = S(\mathcal{B})$. For an equation $\mathcal{A} = \mathcal{B}$ we are looking for a description of all its solutions. We disregard the empty solution $S(X) = \epsilon$ and always assume that $S(X) \neq \epsilon$. In such a case by (1) we can determine the first (last) letter of $S(X)$ in $\mathcal{O}(1)$ time.

Lemma 1. *Let a be the first letter of A_0. If $A_0 \in a^+$ then $S(X) \in a^*$ for each solution S of $\mathcal{A} = \mathcal{B}$, all such solutions can be calculated and reported in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time. If $A_0 \notin a^*$ then there is at most one solution $S(X) \in a^+$, the length of such a solution can be returned in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time. For $S(X) \notin a^+$ the lengths of the a-prefixes of $S(X)$ and A_0 are the same.*

A symmetric version of Lemma 1 holds for the suffix of $S(X)$. By $\mathsf{SimpleSolution}(a)$ we denote a procedure that for $A_0 \notin a^*$ returns the unique ℓ such that $S(X) = a^\ell$ is a solution (or nothing, if there is no such solution).

Representation of Solutions. Consider any solution S of $\mathcal{A} = \mathcal{B}$. If $|S(X)| \leq |A_0|$ then $S(X)$ is a prefix of A_0. When $|S(X)| > |A_0|$ then $S(\mathcal{A})$ begins with $A_0 S(X)$ while $S(\mathcal{B})$ begins with $S(X)$ and thus $S(X)$ has a period A_0. Hence $S(X) = A_0^k A$, where A is a prefix of A_0 and $k > 0$. In both cases $S(X)$ is uniquely determined by $|S(X)|$, so it is enough to describe such lengths.

Each letter in the current instance of our algorithm represents some (compressed) string of the input equation, we store its *weight* which is the length of such a string. Furthermore, when we replace X with $a^\ell X$ (or $X a^\ell$) we keep track of the weight of a^ℓ. In this way, for each solution of the current equation we know what is the length of the corresponding solution of the original equation and this identifies it uniquely.

Preserving Solutions. All subprocedures of the algorithm should preserve solutions, i.e. there should be a one-to-one correspondence between solutions

before and after the application of the subprocedure. However, as they replace X with $a^\ell X$ (or Xb^r), some solutions are lost in the process and so they should be reported. We formalise these notions.

We say that a subprocedure *preserves solutions* when given an equation $\mathcal{A} = \mathcal{B}$ it returns $\mathcal{A}' = \mathcal{B}'$ such that for some strings u and v

- some solutions of $\mathcal{A} = \mathcal{B}$ are reported by the subprocedure,
- S is an unreported solution of $\mathcal{A} = \mathcal{B}$ if and only if there is a solution S' of $\mathcal{A}' = \mathcal{B}'$ such that $S(X) = uS'(X)v \neq uv$.

By $\mathrm{PC}_{ab\to c}(w)$ we denote the string obtained from w by replacing each ab by c (we assume that $a \neq b$, so this is well-defined), this corresponds to pair compression. We say that a subprocedure *properly implements pair compression* for ab, if it satisfies the conditions for preserving solutions above, but with $\mathrm{PC}_{ab\to c}(S(X)) = uS'(X)v$ replacing $S(X) = uS'(X)v$. Similarly, by $\mathrm{BC}_a(w)$ we denote a string with maximal blocks a^ℓ replaced by a_ℓ (for each $\ell > 1$) and we say that a subprocedure *properly implements blocks compression* for a letter a.

Given an equation $\mathcal{A} = \mathcal{B}$, its solution S and a pair $ab \in \Sigma^2$ appearing in $S(\mathcal{A})$ (or $S(\mathcal{B})$) we say that this appearance is *explicit*, if it comes from substring ab of $\mathcal{A}$ (or $\mathcal{B}$, respectively); *implicit*, if it comes (wholly) from $S(X)$; *crossing* otherwise. A pair is *crossing* if it has a crossing appearance and *noncrossing* otherwise. A similar notion applies to maximal blocks of as, in which case we say that a *has a crossing block* or it *has no crossing blocks*. Alternatively, a pair ab is crossing if b is the first letter of $S(X)$ and aX appears in the equation or a is the last letter of $S(X)$ and Xb appears in the equation or a is the last and b the first letter of $S(X)$ and XX appears in the equation.

Unless explicitly stated, we consider crossing/noncrossing pairs ab in which $a \neq b$. As the first (last) letter of $S(X)$ is the same for each S, the definition of the crossing pair *does not depend on the solution*; the same applies to crossing blocks.

When a pair ab is noncrossing, its compression is easy, as it is enough to replace each explicit ab with a fresh letter c, we refer to this procedure as $\mathsf{PairCompNCr}(a, b)$. Similarly, when no block of a has a crossing appearance, the a's blocks compression consists simply of replacing explicit a blocks, we call this procedure $\mathsf{BlockCompNCr}(a)$.

Lemma 2. *If ab is a noncrossing pair then* $\mathsf{PairCompNCr}(a, b)$ *properly implements pair compression for ab. If a has no crossing blocks, then* $\mathsf{BlockCompNCr}(a)$ *properly implements the block compression for a.*

The main idea of the recompression method is the way it deals with the crossing pairs: imagine that ab is a crossing pair, this is because $S(X) = bw$ and aX appears in $\mathcal{A} = \mathcal{B}$ or $S(X) = wa$ and Xb appears in it (the remaining case, in which $S(X) = awb$ and XX appears in the equation is treated in the same way). The cases are symmetric, so we deal only with the first one. To 'uncross' ab in this case it is enough to 'left-pop' b from X: replace each X in the equation with bX and implicitly change the solution to $S(X) = w$.

Algorithm 1. Pop(a, b)

```
1: if b is the first letter of S(X) then
2:     if SimpleSolution(b) returns 1 then          ▷ S(X) = b is a solution
3:         report solution S(X) = b
4:     replace each X in A = B by bX    ▷ Implicitly change S(X) = bw to S(X) = w
5:                                          ▷ perform symmetric actions for a
```

Lemma 3. Pop(a, b) *preserves solutions. After it the pair ab is noncrossing.*

The presented procedures are merged into PairComp(a, b) that turns crossing pairs into noncrossing ones and then compresses them.

Lemma 4. PairComp(a, b) *properly implements the pair compression of ab.*

The number of noncrossing pairs can be large, however, applying Pop(a, b), where b, a are the first and last letters of the $S(X)$ reduces their number to 2.

Lemma 5. *After* Pop(a, b)*, where b, a are the first and last letters of the* $S(X)$*, the solutions are preserved and there are at most two crossing pairs.*

The problems with crossing blocks are solved in a similar fashion: a has a crossing block, if and only if aa is a crossing pair. So we 'left-pop' a from X until the first letter of $S(X)$ is different than a, we do the same with the ending letter b. This effectively removes 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 $S(X) = w$. The corresponding procedure is called CutPrefSuff.

Lemma 6. CutPrefSuff *preserves solutions and after its application there are no crossing blocks of letters.*

BlockComp(a) compresses all blocks of a, regardless of whether it is crossing or not, by first applying CutPrefSuff and then BlockCompNCr(a).

Lemma 7. BlockComp(a) *properly implements the block compression for a before its application.*

3 Main Algorithm

The following algorithm OneVar is basically a simplification of the general algorithm for testing the satisfiability of word equations [6].

Algorithm 2. OneVar reports all solutions of a given word equation

```
1: while |A_0| > 1 do
2:     Letters ← letters in A = B
3:     run CutPrefSuff                                   ▷ There are now crossing blocks
4:     for a ∈ Letters do          ▷ Compressing blocks, time O(|A| + |B|) in total
5:         run BlockComp(a)
6:     Pop(a, b), where a is the first and b the last letter of S(X)
7:                                              ▷ Now there are only two crossing pairs
8:     Crossing ← list of crossing pairs, Non-Crossing ← list of noncrossing pairs
9:     for each ab ∈ Non-Crossing do   ▷ Compress noncrossing pairs, O(|A| + |B|)
10:        PairCompNCr(a, b)
11:    for ab ∈ Crossing do          ▷ Compress the 2 crossing pairs, O(|A| + |B|)
12:        PairComp(a, b)
13: TestSolution                                          ▷ Test solutions from a*
```

We call one iteration of the main loop a *phase*.

Theorem 1. OneVar *runs in time* $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}| + (n_{\mathcal{A}} + n_{\mathcal{B}}) \log(|\mathcal{A}| + |\mathcal{B}|))$ *and correctly reports all solutions of a word equation* $\mathcal{A} = \mathcal{B}$.

The most important property of OneVar is that the explicit strings between the variables shorten (assuming they are long enough): We say that a word A_i (B_j) is *short* if it consists of at most $C = 100$ letters and *long* otherwise.

Lemma 8. *If* A_i (B_j) *is long then its length is reduced by* $1/4$ *in this phase; if it is short then after the phase it still is.*

If the first word is short then its length is shortened by at least 1 *in a phase.*

It is relatively easy to estimate the running time of one phase.

Lemma 9. *One phase of* OneVar *can be performed in* $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ *time.*

The cost of one phase is charged towards the words $A_0, \ldots, A_{n_{\mathcal{A}}}, B_1, \ldots, B_{n_{\mathcal{B}}}$ proportionally to their lengths. Since the lengths of the long words drop by a constant factor in each phase, in total such cost is $\mathcal{O}(n)$. For short words the cost is $\mathcal{O}(1)$ per phase and there are $\mathcal{O}(\log n)$ phases by Lemma 8.

4 Heuristics and Better Analysis

The main obstacle in the linear running time is the necessity of dealing with short words, as the time spent on processing them is difficult to charge. The improvement to linear running time is done by four major modifications:

several equations We store a system of several equations and look for a solution of such a system. This allows removal of some words from the equations.

small solutions We identify a class of particularly simple solutions, called *small*, and show that a solution is reported within $\mathcal{O}(1)$ phases from the moment when it became small. In several cases of the analysis we show that the solutions involved are small and so it is easier to charge the time spent on testing them.

storage All words are represented by a structure of size proportional to the size of the long words. In this way the storage space (and so also time used for compression) decreases by a constant factor in each phase.

testing The testing procedure is modified, so that the time it spends on the short words is reduced. We also improve the rough estimate that SimpleSolution takes time proportional to $|\mathcal{A}| + |\mathcal{B}|$ to an estimation that counts for each word whether it was included in the test or not.

Several Equations. We store several equations and look for substitutions that simultaneously satisfy all of them. Hence we have a collection $\mathcal{A}_i = \mathcal{B}_i$ of equations, for $i = 1, \ldots, m$, each of them is of the form (1). This system is obtained by replacing one equation $\mathcal{A}'_i\mathcal{A}''_i = \mathcal{B}'_i\mathcal{B}''_i$ with equivalent two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$.

Each of the equations $\mathcal{A}_i = \mathcal{B}_i$ in the system specifies the first and last letter of the solution, length of the a-prefix and suffix etc., exactly in the same way as it does for a single equation. However, it is enough to use only one of them, say $\mathcal{A}_1 = \mathcal{B}_1$, as if there is any conflict then there is no solution at all. The consistency is not checked, simply when we find out about inconsistency, we terminate immediately. We say that A_i (B_j) is first or last if it is in any of the stored equations.

All operations on a single equation from previous sections (popping letters, cutting prefixes/suffixes, pair/block compression, etc.) generalise to a system of equations and they preserve their properties and running times, with the length of a single equation $|\mathcal{A}| + |\mathcal{B}|$ replaced by a sum of lengths of all equations $\sum_{i=1}^{m} |\mathcal{A}_i| + |\mathcal{B}_i|$.

Small Solutions. We say that a word w represented as $w = w_1 w_2^{\ell} w_3$ (where ℓ is arbitrary) is *almost periodic*, with *period size* $|w_2|$ and *side size* $|w_1 w_3|$ (note that several such representations may exist, we use this notion for a particular representation that is clear from the context). A substitution S is *small*, if $S(X) = (w)^k v$, where w, v are almost periodic, with period size at most C and side size at most $6C$.

Lemma 10. *Suppose that S is a small solution. There is a constant c such that within c phases the corresponding solution is reported by* OneVar.

Storing. While the long words are stored exactly as they used to, the short words are stored more efficiently: we keep a table of short words and equations point to the table of short words instead of storing them. We say that such

a representation is *succinct* and its size is the sum of lengths of words stored in it. Note that we do *not* include the size of the equation.

The correctness of such an approach is guaranteed by the fact that equality of two explicit words is not changed by OneVar, which is shown by a simple induction.

Lemma 11. *Consider any words A and B in the input equation. Suppose that during* OneVar *they were transformed to $A' = B'$, none of which is a first nor last word. Then $A = B$ if and only if $A' = B'$.*

Hence, to perform the compression it is enough to read the succinct representation without looking at the whole equation. In particular, the compression (both pair and block) can be performed in time proportional to the size of the succinct representation.

Lemma 12. *The compression in one phase of* OneVar *can be performed in time linear in size of the succinct representation.*

Ideally, we want to show that the succinct representation has size proportional to the length of long words. In this way its size would decrease by a constant factor in each phase and thus be $\mathcal{O}(n)$ in total. In reality, we are quite close to this: the words stored in the tables are of two types: normal and overdue. The *normal* words are substrings of the long words or A_0^2 and consequently the sum of their sizes is proportional to the size of the long words. A word becomes *overdue* if at the beginning of the phase it is not a substring of a long word or A_0^2. It might be that it becomes a substring of such a word later, it does not stop to be an overdue word in such a case. The new overdue words can be identified in linear time using standard operations on a suffix array for a concatenation of long and short strings appearing in the equations.

Lemma 13. *In time proportional to the sum of sizes of the long words plus the number of overdue words we can identify the new overdue words.*

The overdue words can be removed from the equations in $\mathcal{O}(1)$ phases after becoming overdue. This is shown by a serious of lemmata.

We say that for a substitution S the word A is *arranged against itself* if each A in $S(\mathcal{A})$ coming from explicit $A_i = A$ corresponds to $B_j = A$ at the same positions in $S(\mathcal{B})$ (and symmetrically, for the sides of the equation exchanged).

Lemma 14. *Consider a word A in a phase in which it becomes overdue and a solution S. Then either S is small or A is arranged against itself.*

The proof is rather easy: we consider the $A_i = A$ that is not arranged against some $B_j = A$ in $S(\mathcal{A}) = S(\mathcal{B})$. Since by definition it also cannot be arranged against a subword of a long word, case inspection gives that one of the $S(X)$ preceding or succeeding A_i overlaps with some other $S(X)$, yielding that $S(X)$ is periodical. Furthermore, this period has length at most $|A_i| \leq C$, hence $S(X)$ is small.

Due to Lemmata 10 and 14 the overdue words can be removed in $\mathcal{O}(1)$ phases after their introduction: suppose that A becomes an overdue word in phase ℓ. Any solution, in which an overdue word A is not arranged against another copy of A is small and so it is reported after $\mathcal{O}(1)$ phases. Then an equation $\mathcal{A}'_i X A X \mathcal{A}''_i = \mathcal{B}'_i X A X \mathcal{B}''_i$, where $\mathcal{A}'_i$ and $\mathcal{B}'_i$ do not have A as a word, is equivalent to two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$ and this procedure can be applied recursively to $\mathcal{A}''_i = \mathcal{B}''_i$. This removes all copies of A from the system.

Lemma 15. *Consider the set of overdue words introduced in phase ℓ. Then in phase $\ell + c$ (for some constant c) we can remove all words A from equations. The obtained set of equations has the same set of solutions. The time spend on removal of overdue words, over the whole run of* OneVar*, is $\mathcal{O}(n)$.*

This allows to bound the time spent on compression.

Lemma 16. *The running time of* OneVar*, except for time used to test the solutions, is $\mathcal{O}(n)$.*

Testing. SimpleSolution checks whether S is a solution by comparing $S(\mathcal{A}_i)$ and $S(\mathcal{B}_i)$ letter by letter, replacing X with a^ℓ on the fly. We say that in such a case a letter b in $S(\mathcal{A}_i)$ is *tested against* the corresponding letter in $S(\mathcal{B}_i)$.

Suppose that for a substitution S a letter from A_i is tested against a letter from $S(XB_j)$ (there is some asymmetry regarding A_is and B_js in the definition, this is a technical detail without an importance). We say that this test is:

protected if at least one of A_i, A_{i+1}, B_j, B_{j+1} is long

failed if A_i, A_{i+1}, B_j and B_{j+1} are short and a mismathch for S is found till the end of A_{i+1} or B_{j+1};

aligned if $A_i = B_j$ and $A_{i+1} = B_{j+1}$, all of them are short and the first letter of A_i is tested against the first letter of B_j;

misaligned if all of A_i, A_{i+1}, B_j, B_{j+1} are short, $A_{i+1} \neq A_i$ or $B_{j+1} \neq B_j$ and this is not an aligned test;

periodical if $A_{i+1} = A_i$, $B_{j+1} = B_j$, all of them are short and this is not an aligned test.

It is easy to show by case inspection that each test is of one of those type. We calculate the cost of each type of tests separately. For failed tests note that there are constantly many of them in each of the $\mathcal{O}(\log n)$ phases.

Lemma 17. *The number of all failed tests is $\mathcal{O}(\log n)$.*

For protected tests, we charge the cost of the protected test to the long word and only $\mathcal{O}(|A|)$ such tests can be charged to one long word A in a phase. On the other hand, each long word is shortened by a constant factor in a phase and so this cost can be charged to those removed letters and thus the total cost of those tests (over the whole run of OneVar) is $\mathcal{O}(n)$.

Lemma 18. *In one phase the number of protected tests is proportional to the length of long words. Thus there are $\mathcal{O}(n)$ such tests in total.*

In case of the misaligned tests, consider the phase in which the last of A_{i+1}, A_i, B_{j+1}, B_j becomes short. We show that the corresponding solution S' is small in this phase and so by Lemma 10 it is reported within $\mathcal{O}(1)$ following phases. The proof is quite technical, it follows a general idea of Lemma 14: we show that $S(X)$ overlaps with itself and so it has a period. A closer inspection proves that this period is almost periodical.

The cost of the misaligned test is charged to the last word among A_i, A_{i+1}, B_j, B_{j+1} that became short, say, B_j and only $\mathcal{O}(1)$ such tests are charged to this B_j (over the whole run of OneVar). Hence there are $\mathcal{O}(n)$ misaligned tests.

Lemma 19. *There are $\mathcal{O}(n)$ misaligned tests during the whole run of* OneVar.

Consider the maximal set of consecutive aligned tests, they correspond to comparison of $A_iXA_{i+1}\dots A_{i+k}X$ and $B_jXB_{j+1}\dots B_{j+k}X$, where $A_{i+\ell} = B_{j+\ell}$ for $\ell = 0, \dots, k$. Then the next test is either misaligned, protected or failed, so if the cost of all those aligned tests can be bounded by $\mathcal{O}(1)$, they can be associated with the succeeding test. Note that instead of performing the aligned tests (by comparing letters), it is enough to identify the maximal (syntactically) equal substrings of the equation. From Lemma 11 it follows that this corresponds to the (syntactical) equality of substrings in the original equation. We identify such substrings in $\mathcal{O}(1)$ per substring using a suffix array constructed for the input equation.

Lemma 20. *The total cost of aligned tests is $\mathcal{O}(n)$.*

For the periodical tests we apply a similar charging strategy. Suppose that we are to test the equality of (suffix of) $S((A_iX)^\ell)$ and (prefix of) $S(X(B_jX)^k)$. Firstly, it is easy to show that the next test is either misaligned, protected or failed. Secondly, if $|A_i| = |B_j|$ then the test for $A_{i+\ell'}$ and $B_{j+\ell}$ for $0 < \ell' \leq \ell$ is the same as for A_i and B_j and so they can be all skipped. If $|A_i| > |B_j|$ then the common part of $S((A_iX)^\ell)$ and $S(X(B_jX)^k)$ have periods $|S(A_iX)|$ and $|S(B_jX)|$ and consequently has a period $|A_i| - |B_j| \leq C$. So to test the equality of $S((A_iX)^\ell)$ and (prefix of) $S(X(B_jX)^k)$ it is enough to test first common $|A_i| - |B_j|$ letters and check whether both $S(A_iX)$ and $S(B_jX)$ have period $|A_i| - |B_j|$.

Lemma 21. *Performing all periodical tests takes in total $\mathcal{O}(n)$ time*

This yields that the total time of testing is linear.

Lemma 22. *The time spent on testing solutions during* OneVar *is $\mathcal{O}(n)$.*

Acknowledgements I would like to thank P. Gawrychowski for initiating my interest in compressed membership problems and compressed pattern matching, exploring which led to this work and for pointing to relevant literature [10,12]; J. Karhumäki, for his explicit question, whether the techniques of local recompression can be applied to the word equations; W. Plandowski for his numerous comments and suggestions on the recompression applied to word equations.

References

1. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. 22(2), 221–242 (1993)
2. Dąbrowski, R., Plandowski, W.: On word equations in one variable. Algorithmica 60(4), 819–828 (2011)
3. Jeż, A.: Faster fully compressed pattern matching by recompression. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 533–544. Springer, Heidelberg (2012)
4. Jeż, A.: Approximation of grammar-based compression via recompression. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 165–176. Springer, Heidelberg (2013)
5. Jeż, A.: The complexity of compressed membership problems for finite automata. Theory of Computing Systems, 1–34 (2013), http://dx.doi.org/10.1007/s00224-013-9443-6
6. Jeż, A.: Recompression: a simple and powerful technique for word equations. In: Portier, N., Wilke, T. (eds.) STACS. LIPIcs, vol. 20, pp. 233–244. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl (2013), http://drops.dagstuhl.de/opus/volltexte/2013/3937
7. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)
8. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
9. Laine, M., Plandowski, W.: Word equations with one unknown. Int. J. Found. Comput. Sci. 22(2), 345–375 (2011)
10. Lohrey, M., Mathissen, C.: Compressed membership in automata with compressed labels. In: Kulikov, A., Vereshchagin, N. (eds.) CSR 2011. LNCS, vol. 6651, pp. 275–288. Springer, Heidelberg (2011)
11. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik 2(103), 147–236 (1977) (in Russian)
12. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica 17(2), 183–198 (1997)
13. Obono, S.E., Goralcik, P., Maksimenko, M.N.: Efficient solving of the word equations in one variable. In: Privara, I., Ružička, P., Rovan, B. (eds.) MFCS 1994. LNCS, vol. 841, pp. 336–341. Springer, Heidelberg (1994)
14. Plandowski, W.: Satisfiability of word equations with constants is in NEXPTIME. In: STOC, pp. 721–725 (1999)
15. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM 51(3), 483–496 (2004)
16. Plandowski, W., Rytter, W.: Application of Lempel-Ziv encodings to the solution of word equations. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 731–742. Springer, Heidelberg (1998)
17. Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression. J. Discrete Algorithms 3(2-4), 416–430 (2005)