Application of Lempel-Ziv Encodings to the Solution of Word Equations

Wojciech Plandowski * **Wojciech Rytter** †

Abstract

One of the most intricate algorithms related to words is Makanin's algorithm solving word equations. The algorithm is very complicated and the complexity of the problem of solving word equations is not well understood. Word equations can be used to define various properties of strings, e.g. general versions of pattern-matching with variables. This paper is devoted to introduce a new approach and to study relations between Lempel-Ziv compressions and word equations. Instead of dealing with very long solutions we propose to deal with their Lempel-Ziv encodings. As our first main result we prove that each minimal solution of a word equation is highly compressible (exponentially compressible for long solutions) in terms of Lempel-Ziv encoding. A simple algorithm for solving word equations is derived. If the length of minimal solution is bounded by a singly exponential function (which is believed to be always true) then LZ encoding of each minimal solution is of a polynomial size (though the solution can be exponentially long) and solvability can be checked in nondeterministic polynomial time. As our second main result we prove that the solvability can be tested in polynomial deterministic time if the lengths of all variables are given in binary. We show also that lexicographically first solution for given lengths of variables is highly compressible in terms of Lempel-Ziv encodings.

1 Introduction

Word equations are used to describe properties and relations of words, e.g. pattern-matching with variables, imprimitiveness, periodicity, conjugation, see [5]. The main algorithm in this area is Makanin's algorithm for solving word equations, see [8]. The time complexity of the algorithm is too high, its most efficient version works in $2^{2^{p(n)}}$ nondeterministic time where $p(n)$ is the maximal index of periodicity of word equations of length n ($p(n)$ is a singly exponential function), see [6]. The descriptional complexity is also too high. As a side effect of our results we present a much simpler algorithm.

It is known that the solvability problem for word equations is NP-hard, even if we consider (short) solutions with the length bounded by a linear function and the right side of equations contains no variables, see [1].

*Turku Centre for Computer Science and Department of Mathematics, Turku University, 20 014, Turku, Finland. On leave from Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02–097 Warszawa, Poland. Email:`wojtekpl@mimuw.edu.pl`

†Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02–097 Warszawa, Poland, and Department of Computer Science, University of Liverpool, UK . Email: `rytter@csc.liv.ac.uk`

The main open problem is to close the gap between NP and $2^{2^{p(d)}}$, and to show the following

Conjecture A: the problem of solving word equations is in NP.

Assume n is the size of the equation and N is the minimal length of the solution (if one exists). It is generally believed that another conjecture is true (at least no counterexample is known):

Conjecture B: N is at most singly exponential w.r.t. n.

Current estimation for the function $\mathcal{N}$ is $2^{2^{2^{p(n)}}}$. We strongly believe that the proper bound is singly exponential. If it is true then our construction would prove that the problem of solvability of word equations is NP-complete.
In this paper we introduce a new approach to deal with word equations: Lempel-Ziv (LZ) encodings of solutions of word equations. Recently many results for several variations of pattern-matching and other combinatorial problems for compressed texts were obtained, see [4, 9, 3]. Many words can be exponentially compressed using LZ-encoding. A motivation to consider compressed solutions follows from the following fact.

Lemma 1
If we have LZ-encoded values of the variables then we can verify the word equation in polynomial time with respect to the size of the equation and the total size of given LZ-encodings.

Proof: We can convert each LZ-encoding to a context-free grammar generating a single word, due to the following claim.

Claim Let $n = |LZ(w)|$. Then we can construct a context-free grammar G of size $O(n^2 \log n)$ which generates w and which is in the Chomsky normal form.

Now we can compute the grammars corresponding to the left and right sides of the equations by concatenating some smaller grammars. The equality of grammars can be checked in polynomial time by the algorithm of [10]. □

Our first result is:

Theorem 2 *[***Main-Result1***]*
Assume N is the size of minimal solution of a word equation of size n. Then each solution of size N can be LZ-compressed to a string of size $O(n^2 \log^2(N)(\log n + \log\log N))$.

As a direct consequence we have:

Corrolary 1. Conjecture B implies conjecture A.

Proof:
If N is exponential then the compressed version of the solution is of a polynomial size. The algorithm below solves the problem in nondeterministic polynomial time. The first step works in nondeterministic polynomial time, the second one works in deterministic polynomial time due to Lemma 1. □

ALGORITHM *Solving_by_LZ-Encoding* ;

guess LZ-encoded solution of size
$O(n^2 \log^2 N(\log n + \log\log N))$;
verify its correctness using the polynomial
time deterministic algorithm from Lemma 1.

Observation. Take $N = 2^{2^{2^{p(n)}}}$. Then the algorithm *Solving_by_LZ-Encoding* is probably the simplest algorithm solving word equations with similar time complexity as the best known (quite complicated) algorithms.

It was known before that there is a polynomial time deterministic algorithm if the lengths of all variables are given in unary. We strengthen this results allowing binary representations (changing polynomial bounds to exponential). Our second main result is:

Theorem 3 *[***Main-Result 2***]*
Assume the length of all variables are given in binary by a function f. Then we can test solvability in deterministic polynomial time, and produce polynomial-size compression of the lexicographically first solution (if there is any).

Let Σ be an alphabet of constants and Θ be an alphabet of variables. We assume that these alphabets are disjoint. A word equation E is a pair of words $(u, v) \in (\Sigma \cup \Theta)^* \times (\Sigma \cup \Theta)^*$ usually denoted by $u = v$. The *size* of an equation is the sum of lengths of u and v. A *solution* of a word equation $u = v$ is a morphism $h : (\Sigma \cup \Theta)^* \to \Sigma^*$ such that $h(a) = a$, for $a \in \Sigma$, and $h(u) = h(v)$. For example assume we have the equation

$$abx_1x_2x_2x_3x_3x_4x_4x_5 = x_1x_2x_3x_4x_5x_6,$$

and the length of x_i's are consecutive Fibonacci numbers. Then the solution is $h(x_i) = FibWord_i$, where $FibWord_i$ is the i-th Fibbonaci word.

We consider the same version of the LZ algorithm as in [3] (this is called LZ1 in [3]). Intuitively, LZ algorithm compresses the text because it is able to discover some repeated subwords. We consider here the version of LZ algorithm without *self-referencing*. The factorization of w is given by a decomposition: $w = c_1f_1c_2 \ldots f_kc_{k+1}$, where $c_1 = w[1]$ and for each $1 \le i \le k+1$ $c_i \in \Sigma$ and f_i is the longest prefix of $f_ic_{i+1} \ldots f_kc_{k+1}$ which appears in $c_1f_1c_2 \ldots f_{i-1}c_i$. We can identify each f_i with an interval $[p, q]$, such that $f_i = w[p..q]$ and $q \le |c_1f_1c_2 \ldots f_{i-1}c_{i-1}|$. If we drop the assumption related to the last inequality then it occurs a *self-referencing* (f_i is the longest prefix which appears before but not necessarily terminates at a current position). We assume (for simplicity) that this is not the case. We use simple relations between LZ-encodings and context-free grammars.

Example. The LZ-factorization of a word *aababbabbaababbabba#* is given by the sequence:
$c_1\ f_1\ c_2\ f_2\ c_3\ f_3\ c_4\ f_4\ c_5 = a\ a\ b\ ab\ b\ abb\ a\ ababbabba\ \#$ After identifying each subword f_i with its corresponding interval we obtain the LZ encoding of the string. Hence $LZ(aababbabbababbabb\#) = a[1,1]b[2,3]b[4,6]a[2,10]\#$.

As another example we have that the LZ-encoding of $FibWord_n$ is of size $O(n)$.

2 Relations on positions and intervals in the solutions

Let $\mathcal{V}$ be the set of variables. Assume the function $f : \mathcal{V} \to N$ gives the lengths of variables. The function f can be naturally extended to all words over $\mathcal{C} \cup \mathcal{V}$ giving lengths of them under the assumption that the lengths of words which are substituted for variables are defined by f. Let $e : u = v$ be the word equation to consider. Each solution of e in which the lengths of words which are substituted by variables are defined by f is called an f-solution of e. We consider a fixed equation $u = v$ with the lengths of the components of its solution h given by a function f.

We introduce the relation $\mathcal{R}'$ (defined formally below) on positions of the solution, two positions are in this relation iff they correspond to the same symbol in every f-solution ($\mathcal{R}'$ is implied by the structure of equation).

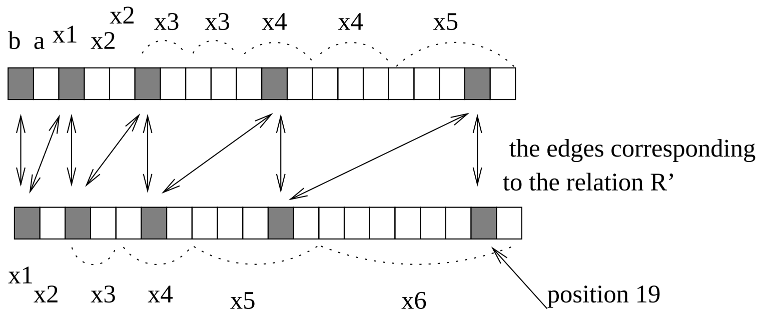

Figure 1: Assume we have equation $abx_1x_2x_2x_3x_3x_4x_4x_5 = x_1x_2x_3x_4x_5x_6$ and the lengths of x_i's are consecutive Fibonacci numbers. Two positions are equivalent (contain always a same symbol) iff they are in the relation $\mathcal{R}$, which is a transitive closure of $\mathcal{R}'$. For example the 19th and the first positions are connected via pairs of positions which are in the relation $\mathcal{R}'$. Hence these positions are equivalent, so the 19th position is in the class corresponding to the constant b.

We use the identity $h(u) = h(v)$ in Σ^*, that is identify the corresponding letters on both sides of this identity, to define an equivalence relation $\mathcal{R}$ on positions of $h(u)$. The positions in the equivalence classes are to be filled by the same constant. The constant is uniquely determined if one of the positions in the class corresponds to a constant in an original equation. Otherwise the constant can be chosen arbitrarily. Moreover, the positions in such a class can be filled by any word.

Now, assume that we are given an equation $u_1 \dots u_k = v_1 \dots v_s$ over t variables and a function f such that $f(u) = f(v)$. Denote by $v(j)$ ($u(j)$) the variable or a constant from the left (right) hand of the equation and such that it contains a position j or in case of a constant occurs at position j under the assumption that the lengths of variables are given by the function f. Formally, $v(j) = v_{p+1}$ if $f(v_1 \dots v_p) < j \leq f(v_1 \dots v_{p+1})$. Denote also $l(j) = j - f(v_1 \dots v_p)$ ($r(j) = j - f(u_1 \dots u_p)$) the position in the variable $v(j)$ ($u(j)$) which correspond to j. We define a function $left$: $\{1, \dots, f(u)\} \to N \times (\Theta \cup \Sigma)$ in the following way:

$$left(j) = \begin{cases} (l(j), v(j)) & \text{if } v(j) \text{ is a variable} \\ (j, v(j)) & \text{otherwise.} \end{cases}$$

Similarly, we define the function $right$:

$$right(j) = \begin{cases} (r(j), u(j)) & \text{if } u(j) \text{ is a variable} \\ (j, u(j)) & \text{otherwise.} \end{cases}$$

The relation $\mathcal{R}'$ is defined as follows:

$$i\mathcal{R}'j \text{ iff } left(i) = right(j) \text{ or } left(i) = left(j) \text{ or } \; right(i) = right(j).$$

Finally, an equivalence relation $\mathcal{R}$ on positions $\{1 \dots f(u)\}$ is the transitive and symmetric closure of the relation $\mathcal{R}'$. We say that a position i *belongs* to a variable X if either $left(i) = (j, X)$ or $right(i) = (j, X)$, for some j. Let $\mathcal{C}$ be an equivalence class of the relation $\mathcal{R}$. We say that $\mathcal{C}$ *corresponds* to a constant a if there is a position i in $\mathcal{X}$ such that either $left(i) = (i, a)$ or $right(i) = (i, a)$.
Now the following lemma is obvious.

Lemma 4 *Let $\mathcal{C}$ be an equivalence class of the relation $\mathcal{R}$ connected to an equation $e : u = v$ under the assumption that the lengths of variables are given by the function f. Then the following conditions are satisfied:*
1. If there is a class $\mathcal{C}$ corresponding to no constant then the solution is not of minimal length. The symbols at positions in $\mathcal{C}$ can be filled with a same arbitrary word, in particular by the empty word.
2. For any two positions $i,\ j \in \mathcal{C}$ and an f-solution h of e, $h(u)[i] = h(u)[j]$.
3. If $\mathcal{C}$ corresponds to a constant a and $i \in \mathcal{C}$, then for each f-solution h of e, $h(u)[i] = a$.
4. There is an f-solution of e iff no equivalence class contains positions of different constants of e.
5. A lexicographically first f-solution of e, if exists, can be obtained by filling all positions in all equivalence classes of $\mathcal{R}$ which do not contain a constant by lexicographically first letter of the alphabet.

The relation $\mathcal{R}$ is defined on positions of an f-solution of e. In our considerations we need an extension of this relation to length n segments of an f-solution of e.

$$left_n(j) = \begin{cases} (l(j), v(j)) & v(j) = v(j+n-1) \\ (j, v(j)) & otherwise \end{cases}$$

$$right_n(t) = \begin{cases} (r(j), u(j)) & u(j) = u(j+n-1) \\ (j, a) & otherwise \end{cases}$$

The functions $left_n$ and $right_n$ are used to define length n segments of solutions which have to be equal in each f-solution of e. They are defined by the relation $\mathcal{R}_n$ which is defined as a transitive closure of the following relation $\mathcal{R}'_n$.

$$\begin{aligned} i\mathcal{R}'_n j \iff & left_n(i) = left_n(j) \text{ or } \; left_n(i) = right_n(j) \text{ or} \\ & right_n(i) = left_n(j) \text{ or } \; right_n(i) = right_n(j). \end{aligned}$$

Lemma 5 *Let h be an f-solution of a word equation $e : u = v$ and let $\mathcal{E}$ be an equivalence class of $\mathcal{R}_n$. If $i,\ j \in \mathcal{E}$ then $h(u)[i..i+n-1] = h(u)[j..j+n-1]$.*

3 Minimal solutions are highly LZ-compressible

Assume $h(u) = h(v) = \mathcal{T}$ is a solution of a given word equation E. A **cut** in $\mathcal{T}$ is a border of a variable or a constant in $\mathcal{T}$. There is a linear number of such cuts and they will be denoted by small Greek letters.

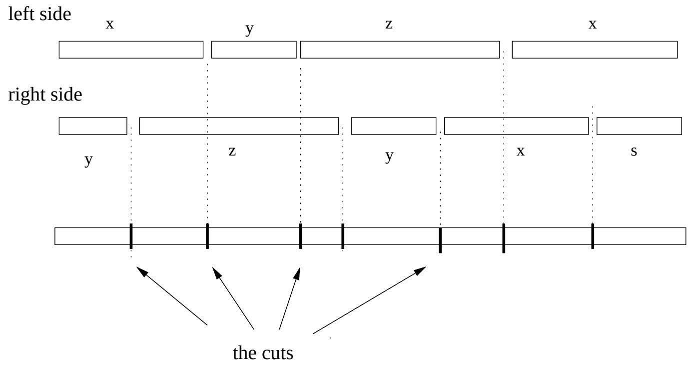

Figure 2: The cuts for the equation $xyzx = yzyxs$ with fixed length of variables, corresponding to the figure.

We say that a subword w of $\mathcal{T}$ **overlaps** a cut γ iff an occurrence of w extends to the left and right of γ or γ is a border of an occurrence.

Lemma 6 *[**key lemma**]*
Assume $\mathcal{T}$ is the minimal length solution of the equation E. Then each subword of $\mathcal{T}$ has an occurrence which overlaps at least one cut in $\mathcal{T}$.

Proof: Assume that both sides of the equations are equal $\mathcal{T}$, where $\mathcal{T}$ is the minimal length solution of the equation E. Assume also that a subword $w = \mathcal{T}[i, j]$ of size t of $\mathcal{T}$ has no occurrence which overlaps at least one cut in $\mathcal{T}$. This implies that it never happens $i \ \mathcal{R}_t \ p$, for an interval $[p, q]$ overlapping a cut. It is easy to see that in this situation no position inside $[i, j]$ is in the relation $\mathcal{R}$ with any constant (since each constant is a neighbor of a cut, by definition). Hence in the equivalence class $\mathcal{C}$ corresponding to some position in $[i, j]$ there is no constant. Due to Lemma 4 we can delete all symbols on positions belonging to $\mathcal{C}$. In this way a new shorter solution is produced, which contradicts minimality of the initial solution $\mathcal{T}$. This completes the proof. $\square$

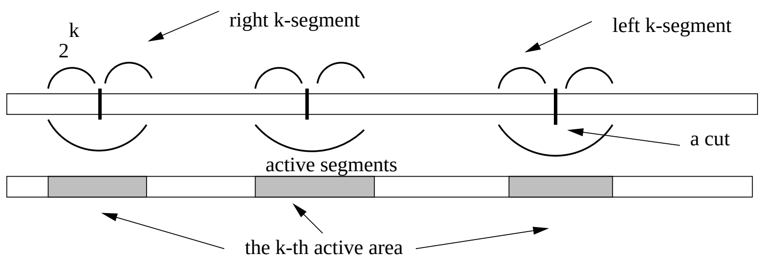

Figure 3: Active segments and the k-th active area.

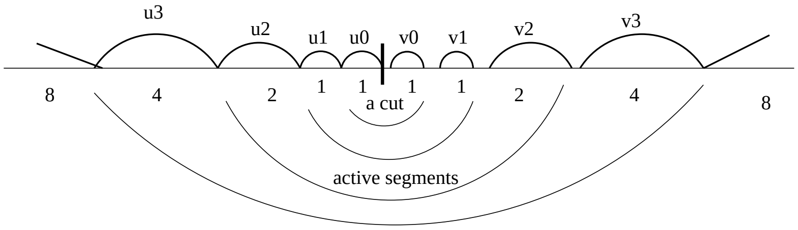

Figure 4: The structure of active segments for the same cut.

For $k = 0, 1, \ldots \log|\mathcal{T}|$ and each cut γ in $\mathcal{T}$ denote by $l_k(\gamma)$ and $r_k(\gamma)$ the subwords of length 2^k whose right (left) border is the cut γ. Denote also $segment_k(\gamma)$ to be the concatenation of $l_k(\gamma)$ and $r_k(\gamma)$. We say that $l_k(\gamma)$ and $r_k(\gamma)$ are respectively, left and right **characteristic words** of rank k and words $segment_k(\gamma)$ are **active segments**. The union of all active segments of rank k is denoted by **Active-Area**(k).

Theorem 7 *[**Main-Result 1**]*
Assume N is the size of minimal solution of a word equation of size n. Then each solution of size N can be LZ-compressed to a string of size $O(n^2 \log^2(N)(\log n + \log\log N))$.

Proof: For a given cut γ consider consecutive words $u_0(\gamma), u_1(\gamma), u_2(\gamma), \ldots$ whose lengths are 1, 1, 2, 4, ..., and which are on the left of γ. Similarly we define words $v_0(\gamma), v_1(\gamma), \ldots$ to the right of γ, see Figure 4. The sequences of these words continue maximally to the left (right) without hitting another cut. Then for $k \geq 0$

$$segment_{k+1}(\gamma) = u_{k+1}\ segment_k(\gamma)\ v_{k+1}$$

Claim 1. $\mathcal{T}$ is covered by a concatenation of a linear number of active segments.

It is easy to see that due to Lemma 6 we have.

Claim 2. Each of the words u_{k+1} and v_{k+1} is contained in $segment_k(\beta)$, $segment_k(\alpha)$ for some cuts α, β.

We can write now:

$$segment_{k+1}(\gamma) = segment_k(\beta)[i..j]\ segment_k(\gamma)\ segment_k(\alpha)[p..q]$$

for some cuts α, β and intervals $[i..j]$, $[p..q]$.

In this way we have recurrences describing larger active segments in terms of smaller active segments (as their subwords). We start with active segments of a constant size.

Claim 3. Assume we have a set of m recurrences describing consecutive words in terms of previously defined words, as concatenations of finite number of subwords of these earlier words. Assume we start with words of constant size. Then the last described word has an LZ-encoding of size $O(m^2 \log m)$. This small LZ-encoding can be computed in deterministic polynomial time w.r.t. m if the recurrences are given.

Sketch of the proof of the Claim. Assume the words computed by recurrences are $z_1, z_2, \ldots, z_m$. Then we can create one long word $z = z_1 \cdot z_2 \cdot \ldots z_m$ which has the LZ-encoding of size $O(m)$ given by recurrences. We can transform this encoding to a context-free grammar of size $O(m^2 \log m))$ generating z as a single word, we refer to the claim in the proof of Theorem 11 in [4]. Next we construct a grammar of size $O(m^2 \log m)$ for z_m as a segment of z. Next we can transform this grammar to a LZ-encoding of similar size.

In our case we have $m = O(n \log N)$ as a bound for the number of possible $\log N$ segments for n cuts together with n subwords of segments, needed in Claim 1. Hence the resulting encoding is $O(m^2 \log m)$ which is $O(n^2 \log^2(N)(\log n + \log\log N))$. $\square$

4 Polynomial time algorithm for a given vector of lengths

We use again the idea of geometrically decreasing neighborhoods (active areas) of the cuts, which are the most essential in the solution. Let us fix the length of the variables and $h(u) = h(v) = \mathcal{T}$. We want to represent the relation between positions of $\mathcal{T}$ (implied by the equation) restricted to the k-th active areas, starting from large k and eventually finishing with $k = 0$, which gives the relation of a polynomial size which can be used directly to check solvability. So we compute consecutive structures like *shortcuts* in a graph corresponding to the relation on positions (identifying symbols on certain pairs of positions). The crucial point is to represent succinctly exponential sets of occurrences, this is possible due to the following fact.

Lemma 8
The set of occurrences of a word w inside a word v which is exactly twice longer than w forms a single arithmetic progression.

Denote by $\mathcal{S}_k$ the relation $\mathcal{R}_{2^k}$ restricted to positions and intervals which are within the active area of rank k.
Our main data structure at the k-th iteration is the k-th *overlap structure* $\mathcal{OS}_k$, which is a collection:

$$\{Overlaps_k(w, \beta) \ : \beta \in CUTS(\mathcal{T})\}$$

where w is a characteristic word of rank k.
The sets in $\mathcal{OS}_k$ consist of overlaps of characteristic words against the cuts in $\mathcal{T}$. We consider only overlaps which fit inside $segments_k(\gamma)$, and which form arithmetic progressions and are implied by the structure of the equation, the relation $\mathcal{R}_{2^k}$. The overlap structure $Overlaps$ has three features

- for each cut β and a characteristic word w of rank k the set $\{Overlaps_k(w, \beta) \ : \beta \in CUTS(\mathcal{T})\}$ forms single arithmetic progression,
- in each f-solution of the equation the words of length 2^k which start at positions in $Overlaps_k(w, \beta)$ are equal to w,
- the sum $Overlaps_k(w, \beta)$ which is taken over all cut points β is a union of some equivalence classes in $\mathcal{S}_k$.

The second and the third conditions gives us the following property of the set $\mathcal{OS}_0$ which deals with one-letter subwords of each f-solution.
Lemma 9 *The equation has an f-solution iff for each characteristic word w of rank 0 there is no set in $\bigcup_{\beta \in CUTS(\mathcal{T})} Overlaps_0(w, \beta)$ in $\mathcal{OS}_0$ which contains two different constants of the equation.*
If $\mathcal{OS}_0$ is given then solvability can be tested in polynomial time.
Proof: The implication from left to right is a consequence of the second property of sets $Overlaps_0$. If the equation has no f-solution then by Lemma 4 one of the equivalence

classes of $\mathcal{R}$ contains a positions of different constants of the equation. Since $\mathcal{S}_0$ is a restriction of $\mathcal{R}$ to positions of constants and starting and ending symbols of variables, by second and the third property of $Overlaps_0$ the result follows. $\square$

A *package* is a set of starting positions of some occurrences of some word w inside words which are twice longer than w. It is always an arithmetic progression and is stored as a pair (b, e) where b is the first number in the progression, e the last one. Since the distance between consecutive numbers in the progression will be the same for all packages it will be stored in one global variable *per*, which is a period of w. Each set $Overlaps_k(w, \beta)$ is represented as a package.

The algorithm works on graphs $G_k(w)$ where w is a characteristic word of rank k which is by definition of length 2^k. The vertices of the graph are the characteristic words of rank $k+1$ represented by two numbers: starting and ending positions of these words in an f-solution. There is an edge $u \to v$ labeled β in $G_k(w)$ if the set $Overlaps_{k+1}(u, \beta)$ is not empty and v is one of the words $l_{k+1}(\beta)$ or $r_{k+1}(\beta)$.

Each vertex v keeps a package $package(v)$ of occurrences of w in v. Initially, $package(v)$ is empty for all vertices except the vertex $v(w)$ which is $l_{k+1}(\beta)$ if $w = l_k(\beta)$) or $r_{k+1}(\beta)$ if $w = r_k(\beta)$. The set $package(v(w))$ consists of one position which is the occurrence of w as the word $l_k(\beta)$ or $r_k(\beta)$ in $v(w)$. At the end the sets $package(v)$ contain all occurrences of w in v which can be deduced from the initial distribution of $package(v)$ and how the packages can move using the set $\mathcal{OS}_{k+1}$ of overlaps of characteristic words of rank $k+1$.

> **Algorithm** *Solvability_For_Given_Lengths*
> **for** k:=$\log T$ **downto** 0 **do**
>   {*invariant* $\mathcal{OS}_{k+1}$ is known}
>   **for each** characteristic word w of rank k **do**
>   *Close_Graph*($G_k(w)$,$v(w)$)
>   compute $\mathcal{OS}_k$ on the basis of the closed graphs $G_k(w)$
>   {*invariant* $\mathcal{OS}_k$ is computed}
> test solvability using $\mathcal{OS}_0$ and Lemma 8

Due to the fact that we operate on packages the set $package(v)$ may contain additional occurrences of w which cannot be deduced in a direct way from $\mathcal{OS}_k$, i.e. by simply moving the information on occurrences along the edges of G_k. Since the resulting set is to be a single progression we use operation $Join$ for merging several packages of occurrences of w inside the same word into the smallest package containing all input packages. The legality of this operation is justified by the following fact.

Fact 10 *Let p_1, p_2 be two packages of occurrences of a word w inside a twice longer word v. Then $Join(p_1, p_2)$ is also a package of occurrences of w in v.*

Example 11 The operation $Join$ of joining packages can result in changing the distance between consecutive numbers in the input progressions *per* if the numbers in progressions do not synchronize as in the following case $Join(\{1,3\},\{6,8\}) = \{1,2,3,4,5,6,7,8\}$.

To formalize the above we define the closure of a graph $G_k(w)$ as the smallest set of packages containing initial distribution of the packages and such that each edge $v \to u$ of the graph is closed, i.e. transferring a package $package(v)$ from a vertex v to u produces a package which is a subset of $package(u)$ (no new packets are produced). Transferring a package along the edge $v \to w$ labeled β consists in putting $package(v)$ in all occurrences of v in $Overlaps_{k+1}(v, \beta)$, joining them into one package, extracting those which drops into u and joining them with $packages(u)$.

Lemma 12 *Given closed graph $G_k(w)$, for each characteristic word w of rank k, the set $\mathcal{OS}_k$ can be computed in polynomial time.*

The algorithm for constructing an f-solution is more complicated. In this algorithm we have to compute a grammar which represents all characteristic words. A production for $l_{k+1}(\beta)$, which is now treated as a nonterminal in the created grammar, is of the form $l_{k+1}(\beta) \to l'_k(\beta) l_k(\beta)$ and the production for $r_{k+1}(\beta)$ is $r_{k+1}(\beta) \to r_k(\beta) r'_k(\beta)$ where r' and l' are the halves of r_{k+1} and l_{k+1}. The productions for the words r' and l' are built on the basis of some of the occurrences of these words over some cut γ of the equation. We compute it using the same technique as for finding the occurrences of words l_k and r_k on the cuts. If such an occurrence does not exist the word l'_k and r'_k can be replaced with the word a^{2^k} which can be represented by a grammar of size $O(k)$. Otherwise the production for a word $r'_k(\beta)$ is of the form $r'_k(\beta) \to Suffix(t, l_k(\gamma)) Pref(s, r_k(\gamma))$ where $Suffix(t, u)$ ($Pref(t, u)$) is a suffix (prefix) of length t of u. The application of the operations of $Suffix$ and $Pref$ generalizes context-free productions, and the whole construction is equivalent via a polynomial time/size transformation to a standard context-free grammar.

Theorem 13 *[Main-Result 2]*
Assume the length of all variables are given in binary by a function f. Then we can test solvability in polynomial time, and produce polynomial-size compression of the lexicographically first solution (if there is any).

5 Computing $Close\_Graph(G_k(w), v(w))$

Assume w and k are fixed. The operation $Close\_Graph(G_k(w), v(w))$ consists essentially in computing occurrences of w inside characteristic words of rank $k+1$. These occurrences are treated as packets. Initially we have one occurrence (initial packet) which is an occurrence of w in $v(w)$, then due to the overlaps of words of rank $k+1$ implied by the overlap structure OS_{k+1} the packets move and replicate. Eventually we have packages (changing sets of known occurrences) which are arithmetical progressions with difference *per* which is the currently known period (not necessarily smallest) of w. Our algorithm works under the assumption that no operation $Join$ which is used in transferring packages changes the period *per*. If during any of the phases of the algorithm this number changes the algorithm starts from the beginning but with the new value of *per*. Since this number after a change is a divisor of the previous one, the number of changes of the period is at most $\log N$.

The crucial property we use in our algorithm is that if some cycle of G' is closed then if new occurrences comes from another part of the graph, then they cannot give new occurrences by transferring them several times through the cycle. Their influence on the packages in the cycle ends before the cycle is fully traversed.

We start by describing how to close a cycle. This will be the one of the basic operations in other parts of the algorithm. To see the route of a package starting from a vertex of a cycle we build a tower of words corresponding to consecutive vertices of the cycle, see Fig. 5. In each word in the tower, the position of the package is shown.

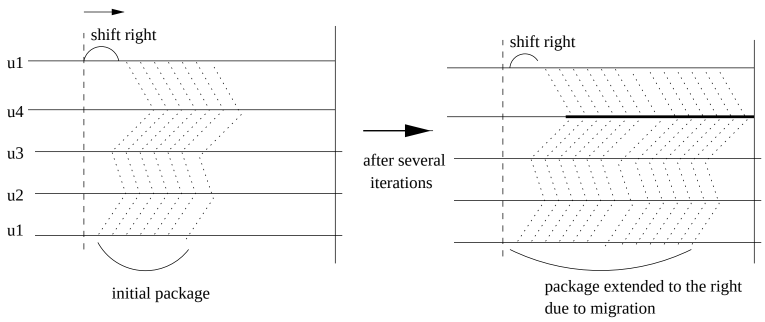

Figure 5: Closure of a cycle $\mathcal{C} = (u_1, u_2, u_3, u_4, u_1)$ in $G_k(w)$.

Now we describe the algorithm which computes the closure of a meta-cycle. A *meta-cycle* is a graph $(B, t \to s)$ which is composed of an edge $t \to s$ and an acyclic graph B such that each node of B belongs to some path from s (source) to t (target). In the following algorithm we extend the operation transfer to an acyclic graphs with one source and one target treating the graph as one meta-edge. The extended operation scans the vertices in topological order. Computing $package(v)$ for a next vertex v consists in transferring the packets from its predecessors along the edges ending at v.

A *boundary packet* in a package is the leftmost and the rightmost number in a package. The algorithm uses the property that a meta-cycle contains a nonclosed cycle iff a boundary packet of $package(s)$ after transferring the package inside a graph B remains boundary and its position is to the left of its old position if it was the leftmost packet and its position is to the right of its old position if it was the rightmost packet.

Algorithm *Close_Meta-cycle*$(B,t \to s))$
oldpckg:=$package(s)$;
transfer $package(t)$ from t to s along $t \to s$;
while $package(s) \neq oldpckg$ **do**
    oldpckg:=$package(s)$;
    transfer $package(s)$ from s to t inside the graph B;
    if one of boundary packets in *oldpckg* is boundary in $package(s)$
    then close a cycle which contains the route of the boundary packet (*)

Lemma 14
1. The operation (*) is performed at most $2 \cdot mv$ times where mv is the number of vertices in the meta-cycle.
2. If each cycle of the meta-cycle is closed then the while loop runs at most once.

Theorem 15 *The algorithm* Close_Graph *works in polynomial time.*

Sketch of the proof. Let n be the number of vertices of G. An acyclic version of the graph G is a graph $Acyclic(G)$ which represents all paths of length n of G. The graph consists of n layers of vertices, each layer represents copies of the vertices of G. All edges join consecutive layers of $Acyclic(G)$. If there is an edge $v \to w$ in G labeled β then there is an edge labeled β between the copies of v and w in all consecutive layers of

Acyclic(G). There are also special edges which are not labeled and they go between copies of the same vertex in consecutive layers. The operation transfer of packages along these edges just copies the packages. It is not difficult to prove that transferring the packages in *Acyclic*(G) simulates transferring the packages n times in G using simultaneously all edges in particular each package *package*(v) travels on all paths starting from v of length n. The restriction of *Acyclic*(G) to all vertices reachable from a copy of v from the first layer of *Acyclic*(G) is called *Path*(v). Similarly a graph *Simple_Paths*(u,v) is created from *Acyclic*(G) by removing all vertices which do not belong to a path from a copy of v in the first layer and some copy of u and by joining all copies of u into one vertex. The graph *Simple_Paths*(u,v) is acyclic and has one source u and target node v. Transferring a package from u to v in *Simple_Paths*(u,v) transferring a package between u and v along all paths of length at most n in G in particular transferring packages along all simple paths (which do not contain a cycle) of G.

Algorithm *Close_Graph(G,source)*
G':=the vertex *source*;
T:=active edges going in G from *source*;
while $T \neq \emptyset$ **do**
 {*invariant* graph G' is closed}
 take an edge $u \to v$ from T and put it into G';
 construct the graph SP=*Simple_Paths*(v,u);
 if SP is not empty **then** Close_Meta-cycle(SP,$u \to v$);
 construct the graph *Paths*(v);
 transfer *package*(u) inside acyclic graph *Paths*(v);
 find new edges in G which started to be active and put them into T

References

[1] Angluin D., Finding patterns common to a set of strings, *J.C.S.S.*, **21(1)**, 46-62, 1980.
[2] Choffrut, C., and Karhumäki, J., Combinatorics of words, *in* G.Rozenberg and A.Salomaa (eds), *Handbook of Formal Languages*, Springer, 1997.
[3] Farah, M., Thorup M., String matching in Lempel-Ziv compressed strings, *STOC'95*, 703-712, 1995.
[4] L. Gąsieniec, M. Karpiński, W. Plandowski and W. Rytter, *Randomized Efficient Algorithms for Compressed Strings: the finger-print approach.* in proceedings of the CPM'96, LNCS 1075, 39-49, 1996.
[5] Karhumäki J., Mignosi F., Plandowski W., The expressibility of languages and relations by word equations, *in ICALP'97*, LNCS 1256, 98-109, 1997.
[6] Koscielski, A., and Pacholski, L., Complexity of Makanin's algorithm, *J. ACM* **43**(4), 670-684, 1996.
[7] A. Lempel, J. Ziv, On the complexity of finite sequences, *IEEE Trans. on Inf. Theory*, 22, 75-81, 1976.
[8] Makanin, G.S., The problem of solvability of equations in a free semigroup, *Mat. Sb.*, Vol. 103,(145), 147-233, 1977. English transl. in *Math. U.S.S.R.* Sb. Vol 32, 1977.
[9] Miyazaki M., Shinohara A., Takeda M., An improved pattern matching algorithm for strings in terms of straight-line programs, *in CPM'97*, LNCS 1264, 1-11, 1997.
[10] W. Plandowski, Testing equlity of morphisms on context-free languages, in ESA'94