Processing Compressed Texts: A Tractability Border [⋆]

Yury Lifshits

Steklov Institute of Mathematics at St.Petersburg, Russia,
yura@logic.pdmi.ras.ru

Abstract. What kind of operations can we perform effectively (without full unpacking) with compressed texts? In this paper we consider three fundamental problems: (1) check the equality of two compressed texts, (2) check whether one compressed text is a substring of another compressed text, and (3) compute the number of different symbols (Hamming distance) between two compressed texts of the same length.
We present an algorithm that solves the first problem in $O(n^3)$ time and the second problem in $O(n^2m)$ time. Here n is the size of compressed representation (we consider representations by straight-line programs) of the text and m is the size of compressed representation of the pattern. Next, we prove that the third problem is actually #P-complete. Thus, we indicate a pair of similar problems (equivalence checking, Hamming distance computation) that have radically different complexity on compressed texts. Our algorithmic technique used for problems (1) and (2) helps for computing minimal periods and covers of compressed texts.

1 Introduction

How can one minimize data storage space, without compromising too much on the query processing time? Here we address this problem using data compression perspective. Namely, what kind of problems can be solved in time polynomially depending on the size of *compressed* representation of texts?

Algorithms on compressed texts have applications in various areas of theoretical computer science. They were used for solving word equations in polynomial space [23]; for solving program equivalence within some specific class in polynomial time [14]; for verification of message sequence charts [9]. Fast search in compressed texts is also important for practical problems. Compression for indices of search engines is critical for web, media search, bioinformatics databases. Next, processing compressed objects has close relation to software/hardware verification. Usual verification task is to check some safety property on all possible system states. However, number of such states is so large that can not be verified by brute force approach. The only way is to store and process all states in some implicit (compressed) form.

[⋆] Support by grants INTAS 04-77-7173 and NSh-8464.2006.1 is gratefully acknowledged.

Problem. *Straight-line program* (SLP) is now a widely accepted abstract model of compressed text. Actually it is just a specific class of context-free grammars that generate exactly one string. Rytter showed [24] that resulting encodings of most classical compression methods (LZ-family, RLE, dictionary methods) can be quickly translated to SLP. We give all details on SLPs in Section 2. Recently, an interesting variation of SLP was presented under the name of *collage systems* [13]. For any text problem on SLP-generated strings we ask two following questions: (1) Does a polynomial algorithm exist? (2) If yes, what is exact complexity of the problem? We can think about negative answer to the first question (say, NP-hardness) as an evidence that naive "generate-and-solve" is the best way for that particular problem.

Consider complexity of pattern matching problem on SLP-generated strings (sometimes called fully compressed pattern matching or FCPM). That is, given a SLPs generating a pattern P and a text T answer whether P is a substring of T and provide a succinct description of all occurrences. An important special case is *equivalence problem* for SLP-generated texts. Then we generalize compressed equivalence problem to compressed Hamming distance problem. Namely, given two SLP-generated texts of the same length, compute the number of positions where original texts differ from each other. Equality checking of compressed texts is a natural seems to be a natural problem related to checking changes in backup systems. Fully compressed pattern matching can be used in software verification and media search (audio/video pattern might be quite large and also require compression). Hamming distance is a simple form of approximate matching which is widely used in bioinformatics as well as in media search.

Compressed equality problem was solved for the first time in the paper [22] in 1994 in $O(n^4)$ time. The first solution for fully compressed pattern matching appeared a year later in the paper [12]. Next, a polynomial algorithm for computing combinatorial properties of SLP-generated text was presented in [8]. Finally, in 1997 Miyazaki, Shinohara and Takeda [18] constructed new $O(n^2m^2)$ algorithm for FCPM, where m and n are the sizes of SLPs that generate P and T, correspondingly. In 2000 for one quite special class of SLP the FCPM problem was solved in time $O(mn)$ [10]. Nevertheless, nothing was known about complexity of compressed Hamming distance problem.

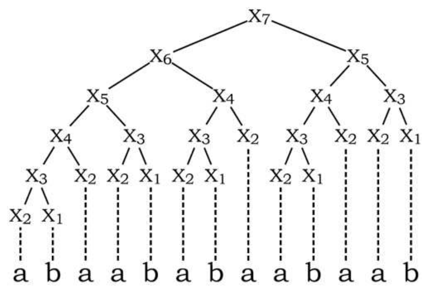
Our results. The key result of the paper is a new $O(n^2m)$ algorithm for pattern matching on SLP-generated texts. As before, m and n are sizes of SLPs generating P and T, correspondingly. This algorithm is not just an improvement over previous ones [8, 10, 12, 18, 22] but is also simpler than they are. Next, we prove $\#\mathsf{P}$-completeness of computing Hamming distance between compressed texts in Section 4. Recall that $\#\mathsf{P}$ is a class of functions, a kind of extension for class of predicates NP. Here for the first time we have closely related problems (equivalence checking, Hamming distance computation) from different sides of the border between efficiently solvable problems on SLP-generated texts and intractable ones. Algorithmic technique from our main FCPM algorithm could be used for computing the shortest period/cover of compressed text. We show this application and state some questions for further research in Section 5.

Historical remarks. Algorithms for finding an explicitly given pattern in a compressed texts were the first results in the field [1, 6]. Next, polynomial algorithms for regular expression matching [19], approximate pattern matching [11] and subsequence matching [5] were constructed for SLP-generated texts. Encouraging experimental results are reported in [20]. Some other problems turned out to be hard. Context-free language membership [17], two-dimensional pattern matching [4], fully compressed subsequence matching [16] are all at least NP-hard. The paper [25] surveys the field of processing compressed texts.

2 Compressed Strings Are Straight-Line Programs

A *Straight-line program* (SLP) is a context-free grammar generating exactly one string. Moreover, we allow only two types of productions: $X_i \to a$ and $X_i \to X_p X_q$ with $i > p, q$. The string represented by a given SLP is a unique text corresponding to the last nonterminal X_m. Although in previous papers X_i denotes only a nonterminal symbol while the corresponding text was denoted by $val(X_i)$ or $eval(X_i)$ we identify this notions and use X_i both as a nonterminal symbol and as the corresponding text. Hopefully, the right meaning is always clear from the context. We say that the size of SLP is equal to the number of productions.

Example. Consider string *abaababaabaab*. It could be generated by the following SLP: $X_7 \to X_6X_5,\ X_6 \to X_5X_4,\ X_5 \to X_4X_3,\ X_4 \to X_3X_2, X_3 \to X_2X_1,$ $X_2 \to a,\ X_1 \to b.$



In fact, the notion of SLP describes only decompression operation. We do not care how such an SLP was obtained. Surprisingly, while the compression methods vary in many practical algorithms of Lempel-Ziv family and run-length encoding, the decompression goes in almost the same way. In 2003 Rytter [24] showed that given any LZ-encoding of string T we could efficiently get an SLP encoding for the same string which is at most $O(\log |T|)$ times longer than the original LZ-encoding. This translation allows us to construct algorithms only in the simplest SLP model. If we get a different encoding, we just translate it to SLP before applying our algorithm. Moreover, if we apply Rytter's translation to LZ77-encoding of a string T, then we get an $O(\log |T|)$-approximation of the *minimal* SLP generating T. The straight-line programs allow the exponential ratio between the size of SLP and the length of original text. For example $X_n \to X_{n-1}X_{n-1}, \ldots X_2 \to X_1X_1, X_1 \to a$ has n rules and generates 2^{n-1}-long text.

We use both $\log |T|$ and n (number of rules in SLP) as parameters of algorithms' complexity. For example, we prefer $O(n \log |T|)$ bound to $O(n^2)$, since in practice the ratio between the size of SLP and the length of the text might be much smaller than exponential.

3 A New Algorithm for Fully Compressed Pattern Matching

Decision version of the fully compressed pattern matching problem (FCPM) is as follows:

INPUT: Two straight-line programs generating P and T
OUTPUT: Yes/No (whether P is a substring in T?)

Other variations are: to find the first occurrence, to count all occurrences, to check whether there is an occurrence from the given position and to compute a "compressed" representation of all occurrences. Our plan is to solve the last one, that is, to compute an auxiliary data structure that contains all necessary information for effective answering to the other questions.

We use a computational assumption which was implicitly used in all previous algorithms. In analysis of our algorithm we count arithmetical operations on positions in *original texts* as unit operations. In fact, text positions are integers with at most $\log |T|$ bits in binary form. Hence, in terms of bit operations the algorithm's complexity is larger than our $O(n^2m)$ estimate up to some $\log |T|$-dependent factor.

Explanation of the algorithm goes in three steps. We introduce a special data structure (*AP-table*) and show how to solve pattern matching problem using this table in Subsection 3.1. Then we show how to compute AP-table using *local search* procedure in Subsection 3.2. Finally, we present an algorithm for local search in Subsection 3.3.

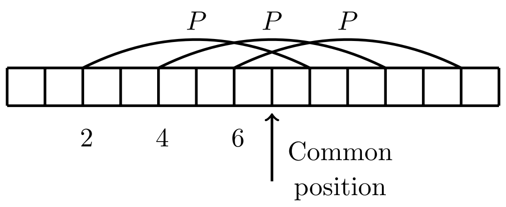
3.1 Pattern Matching via Table of Arithmetical Progressions

We need some notation and terminology. We call a *position* in the text a point between two consequent letters. Hence, the text $a_1 \dots a_n$ has positions $0, \dots, n$ where first is in front of the first letter and the last one after the last letter. We say that some substring *touches* a given position if this position is either inside or on the border of that substring. We use the term *occurrence* both for a corresponding substring and for its starting position. Again, we hope that the right meaning is always clear from the context.

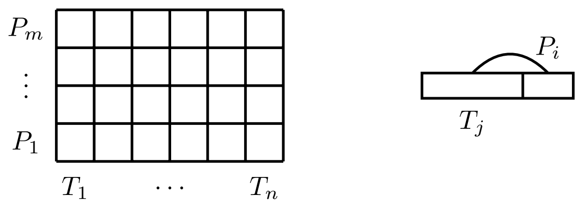
Let $P_1, \dots, P_m$ and $T_1, \dots, T_n$ be nonterminal symbols of SLPs generating P and T. For each of these texts we define a special *cut* position. It is a starting position for one-letter texts and merging position for $X_i = X_r X_s$. In the example above, the cut position for the intermediate text X_6 is between 5th and 6th letters: $abaab|aba$, since X_6 is obtained by concatenating $X_5 = abaab$ and $X_4 = aba$.

Our algorithm is based on the following theoretical fact (it was already used in [18], a similar technique was used also in [2]):

Lemma 1 (Basic Lemma). *All occurrences of P in T touching any given position form a single arithmetical progression (ar.pr.)*



The *AP-table* (table of arithmetical progressions) is defined as follows. For every $1 \leq i \leq m, 1 \leq j \leq n$ the value $AP[i,j]$ is a code of ar.pr. of occurrences of P_i in T_j that touch the cut of T_j. Note that any ar.pr. could be encoded by three integers: first position, difference, number of elements. If $|T_j| < |P_i|$ we define $AP[i,j] = \varnothing_1$, and if text is large enough but there are no occurrences we define $AP[i,j] = \varnothing_2$



Claim 1: Using AP-table (actually, only top row is necessary) one can solve decision, count and checking versions of FCPM in time $O(n)$.
Claim 2: One can compute the whole AP-table by dynamic programming method in time $O(n^2m)$.

Proof of Claim 1. We get the answer for decision FCPM by the following rule: P occurs in T iff there is j such that $AP[m,j]$ is nonempty. Checking and counting are slightly more tricky. Recursive algorithm for checking: test whether the candidate occurrence touches the cut in the current text. If yes, use AP-table and check the membership in the corresponding ar.pr., otherwise call recursively this procedure either for the left or for the right part. We can inductively count the number of P-occurrences in all $T_1, \ldots, T_n$. To start, we just get the cardinality of the corresponding ar.pr. from AP-table. Inductive step: add results for the left part, for the right part and cardinality of central ar.pr. without "just-touching" occurrences.

3.2 Computing AP-table

Sketch of the algorithm for computing AP-table:

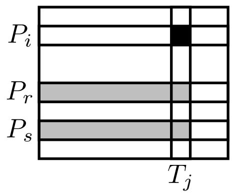
```
1. Preprocessing: compute lengths and cut positions for all intermediate texts;
2. Compute all rows and columns of AP-table that correspond to one-letter texts;
3. From the smallest pattern to the largest one, from the smallest text
   to the largest one consequently compute AP[i,j]:
   (a) Compute occurrences of the larger part of $P_i$ in $T_j$ around the cut of $T_j$;
   (b) Compute occurrences of the smaller part of $P_i$ in $T_j$ that start
       at ending positions of the larger part occurrences;
   (c) Intersect occurrences of smaller and larger part of $P_i$ and merge
       all results to a single ar.pr.
```

Step 1: preprocessing. At the very beginning we inductively compute arrays of lengths, cut positions, first letter, last letter of the texts $P_1, \ldots, P_m$, $T_1, \ldots, T_n$ in time $O(n+m)$.

Step 2: computing AP-table elements in one-letter rows/columns. Case of $|P_i| = 1$: compare it with T_j if $|T_j| = 1$ or compare it with the last letter of the left part and the first one of the right part (we get this letters from precomputation stage). The resulting ar.pr. has at most two elements. Hence, just O(1) time used for computing every cell in the table. Case of $|T_j| = 1$: if $|P_i| > 1$ return $\varnothing_1$, else compare letters. Also $O(1)$ time is enough for every element.

Step 3: general routine for computing next element of AP-table. After one-letter rows and columns we fill AP-table in lexicographic order of pairs (i, j). Let $P_i = P_r P_s$. We use already obtained elements $AP[r, 1], \ldots, AP[r, j]$ and $AP[s, 1], \ldots, AP[s, j]$ for computing $AP[i, j]$. In other words, we only need information about occurrences of left/right part of the current pattern in the current and all previous texts.

Order of computation:
"grey" values are used for computing "black" one
(we assume $P_i = P_r P_s$)

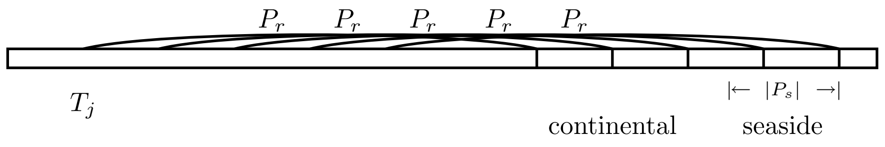


Let the cut position in T_j be γ (we get it from the preprocessing stage), and without loss of generality let $|P_r| \geq |P_s|$. The *intersection method* is (1) to compute all occurrences of P_r "around" cut of T_j, (2) to compute all occurrences of P_s "around" cut of T_j, and (3) shift the latter by $|P_r|$ and intersect.

Miyazaki et al. [18] construct a $O(mn)$ realization of intersection method. We use a different (more accurate) way and new technical tricks that require only $O(n)$ time for computing $AP[i, j]$. We use the same first step as in intersection method. But on the second one we look only for P_s *occurrences that* *start from P_r endings.* We design a special auxiliary *local search* procedure that extracts useful information from already computed part of AP-table. Procedure $LocalSearch(i, j, [\alpha, \beta])$ returns occurrences of P_i in T_j inside the interval $[\alpha, \beta]$. Important properties: (1) Local search uses values AP[i,k] for $1 \leq k \leq j$, (2) It works properly only when $|\beta - \alpha| \leq 3|P_i|$, (3) It works in time $O(j)$, (4) The output of local search is a pair of ar.pr., all occurrences inside each ar.pr. have a common position, and all elements of the second are to the right of all elements of the first. We now show how to compute a new element using 5 local search calls.

Step 3a: finding occurrences of bigger part of the pattern. We apply local search for finding all occurrences of P_r in the interval $[\gamma - |P_i|, \gamma + |P_r|]$. Its length is $|P_i| + |P_r| \leq 3|P_r|$. As an answer we get two ar.pr. of all potential starts of P_r occurrences that touch the cut. Unfortunately, we are not able to do the same for P_s, since the length of interesting interval is not necessarily constant in terms of $|P_s|$. So we are going to find only occurrences of P_s that start from endings of two arithmetical progressions of P_r occurrences.

Step 3b: finding occurrences of smaller part of the pattern. We process each ar.pr. separately. We call an ending *continental* if it is at least $|P_s|$ far from the last ending in progression, otherwise we call it *seaside*.



Since we have an ar.pr. of P_r occurrences that have common position (property 4 of local search), all substrings of length $|P_s|$ starting from continental endings are identical. Hence, we need to check all seaside endings and *only one* continental position. For checking the seaside region we just apply local search for $|P_s|$-neighborhood of last endpoint and intersect the answer with ar.pr. of seaside ending positions. Intersecting two ar.pr. could be done in time $O(\log |T|)$. Indeed, the difference of resulting ar.pr. is equal to the least common multiple of differences of initial progressions. To find the first common point we should solve an equation of type $ax \equiv b \pmod{c}$. This kind of equations can be solved by technique similar to Euclid algorithm.

For checking continental region we apply local search for $|P_s|$ substring starting from the first continental ending.

Step 3c: simplifying answer to a single progression. Complete answer consists of all continental endings/or none of them, plus some sub-progression of seaside endings, plus something similar for the second ar.pr. Since all of these four parts are ar.pr. going one after another, we could simplify the answer to one ar.pr. (it must be one ar.pr. by Basic Lemma) in time $O(1)$.

Complexity analysis. We use one local search call for P_r, four local search calls for P_s, and twice compute the intersection of arithmetical progressions. Hence, we perform 7 steps of $O(n)$ complexity for computing a new element.

3.3 Realization of Local Search

Local search finds all P_i occurrences in the substring $T_j[\alpha, \beta]$. On the first step we run recursive **crawling procedure** with main parameters (i, j, α, β). After halting of all computation branches we get a sorted list of progressions representing P_i occurrences within $[\alpha, \beta]$ interval in T_j. Then we run **merging procedure** that simplifies all progressions to two ones.

Crawling procedure. Here we have three main parameters $(i, j, [\alpha, \beta])$ and two auxiliary ones: *global shift* and *pointer to output list*. Initially we start with main parameters of local search, zero shift and a pointer to empty list. In every call we take the ar.pr. of occurrences of P_i in T_j that touch the cut, leave only occurrences within the interval, and output this truncated ar.pr. (adding global shift) to the list (using current pointer). After that, we check whether the intersection of the interval $[\alpha, \beta]$ with left/right part of T_j is at least $|P_i|$ long. If so, we recursively call crawling procedure with the same i, the index of left/right part of T_j, and with this intersection interval. We also update global shift parameter for the right part and use new pointers to places just before/after inserted element.

Consider the set of all intervals we work with during the crawling procedure. Note that, by construction, any pair of them are either disjoint or embedded. Moreover, since the initial interval is at most $3|P_i|$, there are no four pairwise disjoint intervals in this set. If we consider a sequence of embedded intervals, then all intervals correspond to their own intermediate text from $T_1, \ldots, T_j$. Therefore, there were at most $3j$ recursive calls in crawling procedure and it works in time $O(j)$. At the end we get a sorted list of at most $3n$ arithmetical progressions. By "sorted" we mean that the last element of k-th progression is less than or equal to the first one of $k+1$-th progression. It follows from construction of crawling procedure, that output progressions could have only first/last elements in common.

Merging procedure. We go through the resulting list of progressions. Namely, we compare the distance between the last element of current progression and the first element of the next progression with the differences of these two progressions. If all three numbers are equal we merge the next progression with the current one. Otherwise we just announce a new progression. Applying Basic Lemma to $\delta_1 = \lfloor \frac{2\alpha+\beta}{3} \rfloor$ and $\delta_2 = \lfloor \frac{\alpha+2\beta}{3} \rfloor$ positions, we see that all occurrences of P_i in $[\alpha, \beta]$ interval form at most two (one after another) arithmetical progressions. Namely, those who touch δ_1 and those who don't touch but touch δ_2. Here we use that $\beta - \alpha \leq 3|P_i|$, and therefore any occurrence of P_i touches either δ_1 or δ_2. Hence, our merging procedure starts a new progression at most once.

3.4 Discussion on the Algorithm

Here we point out two possible improvements of the algorithm. Consider in details the "new element routine". Note that local search uses only $O(h)$ time, where h is the height of the SLP generating T, while intersection of arithmetical progressions uses even $O(\log |T|)$. Hence, if it is possible to "balance" any SLP up to $O(\log |T|)$ height, then the bound for working time of our algorithm becomes $O(nm \log |T|)$.

It is interesting to consider more rules for generating texts, since collage systems [13] and LZ77 [26] use concatenations and *truncations*. Indeed, as Rytter [24] showed, we could leave only concatenations expanding the archive just by factor $O(\log |T|)$. However, we hope that the presented technique works directly for the system of truncation/concatenation rules. We also claim that AP-table might be translated to a polynomial-sized SLP generating all occurrences of P in T.

4 Hardness result

Hamming distance (denoted as $HD(S,T)$) between two strings of the same length is the number of characters which differ. *Compressed Hamming distance problem* (counting version): given two straight-line programs generating texts of the same length, compute Hamming distance between them.

A function belongs to class #P if there exists a nondeterministic Turing machine M such that the function value corresponding to input x is equal to the number of accepting branches of $M(x)$. In other words, there exists a polynomially-computable function $G(x,y)$ such that $f(x) = \#\{y | G(x,y) = \text{"yes"}\}$. A function f has a [1]-Turing reduction to a function g, if there exist polynomially-computable functions E and D such that $f(x) = D(g(E(x)))$. We call a function to be #P-complete (under [1]-Turing reductions), if it belongs to the class #P and every other function from this class has a [1]-Turing reduction to it.

Theorem 1. *Compressed Hamming distance problem is #P-complete.*

Proof. Membership in #P. We can use a one-position-comparison as G function: $G(T,S;y) = \text{"yes"}$, if $T_y \neq S_y$. Then number of y giving answer "yes" is exactly equal to Hamming distance. Function G is polynomially computable. Indeed, knowing lengths of all intermediate texts we can walk through SLP decompression tree "from the top to the bottom" and compute value of T_y in linear time.

#P-hardness. It is enough to show a reduction from another complete problem. Recall the well-known #P-complete problem *subset sum* [7]: given integers $w_1, \ldots, w_n, t$ in binary form, compute the number of sequences $x_1, \ldots, x_n \in \{0,1\}$ such that $\sum_{i=1}^{n} x_i \cdot w_i = t$. In other words, how many subsets of $W = \{w_1, \ldots, w_n\}$ have the sum of elements equal to t? We now construct a [1]-Turing reduction from subset sum to compressed Hamming distance. Let us fix input values for subset sum. We are going to construct two straight-line programs such that Hamming distance between texts generated by them can help to solve subset sum problem.

Our idea is the following. Let $s = w_1 + \cdots + w_n$. We construct two texts of length $(s+1)2^n$, describing them as a sequence of 2^n blocks of size $s+1$. The first text T is an encoding of t. All its blocks are the same. All symbols except one are "0", the only "1" is located at the $t+1$-th place. Blocks of the second text S correspond to all possible subsets of W. In every such block the only "1" is placed exactly after the place equal to the sum of elements of the corresponding subset. In a formal way, we can describe T and S by the following formulas:

$$T = (0^t 10^{s-t})^{2^n}, \quad S = \prod_{x=0}^{2^n-1} (0^{\bar{x}\cdot\bar{w}} 10^{s-\bar{x}\cdot\bar{w}}).$$

Here $\bar{x} \cdot \bar{w} = \sum x_i w_i$ and $\prod$ denotes concatenation.

The string S (let us call it *Lohrey string*) was used for the first time in the Markus Lohrey's paper [17], later it was reused in [16]. Lohrey proved in [17] that knowing input values for subset sum one can construct polynomial-size SLP that generates S and T in polynomial time. Notice that $HD(T, S)$ is exactly two times the number of subsets of W with elements' sum nonequal to t. Therefore, the subset sum answer can be computed as $2^n - \frac{1}{2}HD(T, S)$.

It turns out that #P-complete problems could be divided in subclasses that are not Karp-reducible to each other. E.g. recently [21] a new class TotP was presented. A function belongs to TotP, if there exists a nondeterministic machine M such that $f(x)$ is equal to the number of *all* branches of $M(x)$ minus one. Many problems with polynomially-easy "yes/no" version and #P-complete counting version belong to TotP.

We can show that compressed Hamming distance belongs to TotP. Indeed, we can test equality of substrings and split computation every time when *both* the left half and the right half of a substring in the first text are not equal to the corresponding left/right halves of the same interval in the second text. We should also add a dummy computing branch to every pair of nonequal compressed texts.

5 Consequences and Open Problems

A *period* of string T is a string W (and also an integer $|W|$) such that T is a prefix of W^k for some integer k. A *cover* (notion originated from [3]) of a string T is a string C such that any character in T is covered by some occurrence of C in T. Problem of *compressed periods/covers:* given a compressed string T, find the length of minimal period/cover and compute a "compressed" representation of all periods/covers.

Theorem 2. *Assume that AP-table can be computed in time $O((n+m)^k)$. Then compressed periods problem can be solved in time $O(n^k \log |T|)$, while compressed cover problem can be solved in time $O(n^k \log^2 |T|)$.*

Corollary. Our $O(n^2 m)$ algorithm for AP-table provides $O(n^3 \log |T|)$ and $O(n^3 \log^2 |T|)$ complexity bounds for compressed periods and compressed covers, correspondingly.

For complete proof of Theorem 2 we refer to technical report version of this paper [15]. The compressed periods problem was introduced in 1996 in the extended abstract [8]. Unfortunately, the full version of the algorithm given in [8] (it works in $O(n^5 \log^3 |T|)$ time) was never published.

We conclude with some problems and questions for further research:

1. To speed up the presented $O(n^2 m)$ algorithm for fully compressed pattern matching. *Conjecture:* improvement to $O(nm \log |T|)$ is possible. More precisely, we believe that every element in AP-table can be computed in $O(\log |T|)$ time.
2. Is it possible to speed up computing of edit distance (Levenshtein distance) in the case when one text is highly compressible? Formally, is it possible to compute the edit distance in $O(nm)$ time, where n is the length of T_1, and m is the size of SLP generating T_2? This result leads to speedup of edit distance computing in case of "superlogarithmic" compression ratio. Recall that the classical algorithm has $O(\frac{n^2}{\log n})$ complexity.
3. Consider two SLP-generated texts. Is it possible to compute the length of the longest common substring for them in polynomial (from SLPs' size) time?

Compressed suffix tree. Does there exist a data structure for text representation such that (1) it allows pattern matching in time *linear to the pattern's length*, and (2) for some reasonable family of "regular" texts this structure requires less storing space than original text itself?

Application to verification. Algorithm for symbolic verification is one of the major results in model checking. It uses OBDD (ordered binary decision diagrams) representations for sets of states and transitions. It is easy to show that every OBDD representation can be translated to the SLP-representation of the same size. On the other hand there are sets for which SLP representation is logarithmically smaller. In order to replace OBDD representations by SLPs we have to answer the following question. Given two SLP-represented sets A and B, how to compute a close-to-minimal SLP representing $A \cap B$ and a close-to-minimal SLP representing $A \cup B$?

References

1. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. In *SODA'94*, 1994.
2. A. Amir, G. M. Landau, M. Lewenstein, and D. Sokol. Dynamic text and static pattern matching. In *WADS'03*, LNCS 2748, pages 340–352. Springer-Verlag, 2003.
3. A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Inf. Process. Lett.*, 39(1):17–20, 1991.
4. P. Berman, M. Karpinski, L. L. Larmore, W. Plandowski, and W. Rytter. On the complexity of pattern matching for highly compressed two-dimensional texts. *Journal of Computer and Systems Science*, 65(2):332–350, 2002.
5. P. Cegielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich. Window subsequence problems for compressed texts. In *CSR'06*, LNCS 3967. Springer-Verlag, 2006.

6. M. Farach and M. Thorup. String matching in lempel-ziv compressed strings. In *STOC '95*, pages 703–712. ACM Press, 1995.
7. M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness*. Freeman, 1979.
8. L. Gąsieniec, M. Karpinski, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encoding (extended abstract). In *SWAT'96*, LNCS 1097, pages 392–403. Springer-Verlag, 1996.
9. B. Genest and A. Muscholl. Pattern matching and membership for hierarchical message sequence charts. In *LATIN'02*, LNCS 2286, pages 326–340. Springer-Verlag, 2002.
10. M. Hirao, A. Shinohara, M. Takeda, and S. Arikawa. Fully compressed pattern matching algorithm for balanced straight-line programs. In *SPIRE'00*, pages 132–138. IEEE Computer Society, 2000.
11. J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *CPM'00*, LNCS 1848, pages 195–209. Springer-Verlag, 2000.
12. M. Karpinski, W. Rytter, and A. Shinohara. Pattern-matching for strings with short descriptions. In *CPM'95*, LNCS 937, pages 205–214. Springer-Verlag, 1995.
13. T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theor. Comput. Sci.*, 298(1):253–272, 2003.
14. S. Lasota and W. Rytter. Faster algorithm for bisimulation equivalence of normed context-free processes. In *MFCS'06*, LNCS 4162, pages 646–657. Springer-Verlag, 2006.
15. Y. Lifshits. Algorithmic properties of compressed texts. Technical Report PDMI, 23/2006, 2006.
16. Y. Lifshits and M. Lohrey. Quering and embedding compressed texts. In *MFCS'06*, LNCS 4162, pages 681–692. Springer-Verlag, 2006.
17. M. Lohrey. Word problems on compressed word. In *ICALP'04*, LNCS 3142, pages 906–918. Springer-Verlag, 2004.
18. M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight line programs. In *CPM'97*, LNCS 1264, pages 1–11. Springer-Verlag, 1997.
19. G. Navarro. Regular expression searching on compressed text. *J. of Discrete Algorithms*, 1(5-6):423–443, 2003.
20. G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *CPM'99*, LNCS 1645, pages 14–36. Springer-Verlag, 1999.
21. A. Pagourtzis and S. Zachos. The complexity of counting functions with easy decision version. In *MFCS'06*, LNCS 4162, pages 741–752. Springer-Verlag, 2006.
22. W. Plandowski. Testing equivalence of morphisms on context-free languages. In *ESA'94*, LNCS 855, pages 460–470. Springer-Verlag, 1994.
23. W. Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004.
24. W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1–3):211–222, 2003.
25. W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In *ICALP'04*, LNCS 3142, pages 15–27. Springer-Verlag, 2004.
26. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.