Faster Fully Compressed Pattern Matching by Recompression

ARTUR JEŻ, Max Planck Institute für Informatik and Institute of Computer Science, University of Wrocław

In this article, a fully compressed pattern matching problem is studied. The compression is represented by straight-line programs (SLPs)—that is, context-free grammars generating exactly one string; the term *fully* means that both the pattern *and* the text are given in the compressed form. The problem is approached using a recently developed technique of local recompression: the SLPs are refactored so that substrings of the pattern and text are encoded in both SLPs in the same way. To this end, the SLPs are locally decompressed and then recompressed in a uniform way.

This technique yields an $\mathcal{O}((n+m)\log M)$ algorithm for compressed pattern matching, assuming that M fits in $\mathcal{O}(1)$ machine words, where n (m) is the size of the compressed representation of the text (pattern, respectively), and M is the size of the decompressed pattern. If only $m+n$ fits in $\mathcal{O}(1)$ machine words, the running time increases to $\mathcal{O}((n+m)\log M \log(n+m))$. The previous best algorithm due to Lifshits has $\mathcal{O}(n^2 m)$ running time.

Categories and Subject Descriptors: F.2.2 [**Nonnumerical Algorithms and Problems**]: Computations on Discrete Structures; F.4.2 [**Grammars and Other Rewriting Systems**]: Decision Problems

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Pattern matching, compressed pattern matching, algorithms for compressed data, straight-line programs, Lempel-Ziv compression

ACM Reference Format:
Artur Jeż. 2015. Faster fully compressed pattern matching by recompression. ACM Trans. Algor. 11, 3, Article 20 (January 2015), 43 pages.
DOI: http://dx.doi.org/10.1145/2631920

1. INTRODUCTION

Compression and straight-line programs. Due to an ever-increasing amount of data, compression methods are widely applied to decrease the data's size. Still, the stored data is accessed and processed. Decompressing it on each such an occasion basically wastes the gain of reduced storage size. Thus, there is a large demand for algorithms dealing directly with the compressed data, without explicit decompression.

Processing compressed data is not as hopeless as it may seem: it is a popular view that compression basically extracts the hidden structure of the text, and if the compression rate is high, the data has a lot of internal structure. In addition, it is natural to assume that such a structure helps in devising methods dealing directly with the compressed representation. Indeed, efficient algorithms for fundamental text operations (pattern

This research has been supported by National Science Centre (NCN) SONATA 1 grant number 2011/01/D/ST6/07164, 2011–2015.
Author's address: A. Jeż, Max Planck Institute für Informatik, Campus E1 4, DE-66123 Saarbrücken, Germany, and Institute of Computer Science, University of Wrocław, ul. Joliot-Curie 15, 50-383 Wrocław, Poland; email: aje@cs.uni.wroc.pl.

DOI: http://dx.doi.org/10.1145/2631920

matching, equality testing, etc.) are known for various practically used compression methods (LZ77, LZW, their variants, etc.) [Gawrychowski 2011b, 2012a, 2012b, 2013; Gąsieniec et al. 1996a, 1996b; Gąsieniec and Rytter 1999; Hirao et al. 2000; Plandowski 1994].

The compression standards differ in the main idea as well as in details. Thus, when devising algorithms for compressed data, one needs to focus quite early on the exact compression method to which the algorithm is applied. The most practical (and challenging) choice is one of the widely used standards, such as LZW or LZ77. However, a different approach is also pursued: for some applications (and most of theory-oriented considerations), it would be useful to *model* one of the practical compression standards by a more mathematically well-founded and "clean" method. This idea rests at the foundations of the notion of straight-line programs (SLPs), which simply are context-free grammars generating exactly one string. Other reasons for the popularity of SLPs is that usually they compress well the input text [Larsson and Moffat 1999; Nevill-Manning and Witten 1997] and that they are closely related to the LZ77 compression standard: each LZ77 compressed text can be converted into an equivalent SLP of size $\mathcal{O}(n \log(N/n))$ and in $\mathcal{O}(n \log(N/n))$ time [Rytter 2003; Charikar et al. 2005] (where N is the size of the decompressed text), whereas each SLP can be converted to an equivalent LZ77-like of $\mathcal{O}(n)$ size in polynomial time. Finally, a greedy grammar compression can be efficiently implemented and thus can be used as a preprocessing to other compression methods, like those based on Burrows-Wheeler transform [Kärkkäinen et al. 2012].

Problem statement. In this article, we consider the fully compressed membership problem (FCPM), in which we are given a text of length N and pattern of length M, represented by SLPs of size n and m, respectively. We are to answer whether the pattern occurs in the text and give a compact representation of all such occurrences.

Previous and related results. The first algorithmic result dealing with SLPs is for the compressed equality testing—that is, the question of whether two SLPs represent the same text, solved by Plandowski [1994], with $\mathcal{O}(n^4)$ running time. The first solution for FCPM by Karpiński et al. [1995] followed a year later. Next, a polynomial algorithm for computing various combinatorial properties of SLP-compressed texts, particularly pattern matching, was given by Gąsieniec et al. [1996a]; the same authors also presented a faster randomised algorithm for FCPM [Gąsieniec et al. 1996b]. Yet another year later, Miyazaki et al. [1997] constructed an $\mathcal{O}(n^2m^2)$ algorithm for FCPM. A faster $\mathcal{O}(mn)$ algorithm for a special subcase (restricting the form of SLPs) was given by Hirao et al. [2000]. Finally, a state-of-the-art $\mathcal{O}(n^2m)$ algorithm was given by Lifshits [2007]. Note that it beats the $\mathcal{O}(n^4)$ algorithm for equality testing by Plandowski [1994], and no faster algorithm for equality testing was known.

Concerning related problems, fully compressed pattern matching was also considered for LZW compressed strings [Gąsieniec and Rytter 1999], and a linear-time algorithm was developed recently [Gawrychowski 2012b]. Apart from that, there is a large body of work dealing with compressed pattern matching (i.e., when the pattern is given explicitly) for practically used compression standards. We recall those for LZ77 and LZW, as those compression standards are related to SLPs: for LZW a linear-time algorithm was recently given [Gawrychowski 2013], and the case of multiple pattern was also studied [Gawrychowski 2012a], with running time $\mathcal{O}(n \log M + M)$ (alternatively, $\mathcal{O}(n + M^{1+\epsilon})$). For the LZ77-compressed text, for which the problem becomes substantially harder than in the LZW case, an $\mathcal{O}(n \log(N/n) + m)$ algorithm, which is in some sense optimal, was proposed in 2011 [Gawrychowski 2011b].

Equality testing for dynamic strings. Independently of the work on algorithms dealing with SLPs, the data structures for equality testing for dynamic strings were

investigated. In this setting, we have a collection of strings and can add a new string s, which is either a single letter, a substring of string s' from the collection (we are given the first and last position of the substring), or a concatenation of two strings in the collection. The queries to the structure ask for the equality of two strings in the collection. The first and last operation together are enough to straightforwardly simulate SLPs.

The data structure by Mehlhorn et al. [1997], stated in terms of SLPs, yields nearly a cubic algorithm for equality testing of SLPs (as observed by Gawrychowski [2011a]). However, the inside technical details of the construction make extension to pattern matching problematic: although this method can be used to build "canonical" SLPs for the text and the pattern, there is no apparent way to control how these SLPs actually look and how they encode the strings. Moreover, the algorithm essentially uses the fact that the numbers of size N can be manipulated in constant time (in fact, even bit manipulations are needed).

An improved implementation of a similar data structure by Alstrup et al. [2000] solves those problems and allows pattern matching for dynamic strings (as well as improves the running time to a nearly quadratic one). However, the algorithm uses randomised hashing, and derandomisation introduces a logarithmic factor to the running time. Due to the different setting, the transitions to SLPs is possible but not straightforward (especially in the case of pattern matching).

Our results and techniques. We give an $\mathcal{O}((n+m)\log M)$ algorithm for FCPM—that is, a pattern matching problem in which both the text and the pattern are supplied as SLPs. It assumes that numbers of size M can be manipulated in constant time. When this is not allowed and only numbers of $\mathcal{O}(n+m)$ time can be manipulated in constant time, the running time increases to $\mathcal{O}((n+m)\log M\log(n+m))$. Since $M \leq 2^m$, this outperforms, in any case, the previously best $\mathcal{O}(n^2m)$ algorithm by Lifshits [2007] (as well as the earlier algorithms with larger running time). In addition, it outperforms the nearly quadratic randomised algorithm that follows from work of Alstrup et al. [2000] for pattern matching for dynamic strings.

THEOREM 1.1. *Assuming that numbers of size M can be manipulated in constant time, algorithm* FCPM *returns an $\mathcal{O}(n+m)$ representation of all pattern occurrences, where n (m) is the size of the SLP-compressed text (pattern, respectively) and M is the size of the decompressed pattern. It runs in $\mathcal{O}((n+m)\log M)$ time.*

If only numbers of size $n+m$ can be manipulated in constant time, the running time and the representation size increase by a multiplicative $\mathcal{O}(\log(n+m))$ factor.

This representation allows calculation of the number of pattern occurrences and, if N fits in $\mathcal{O}(1)$ codewords, also the position of the first/last pattern. Under the same assumption, the position of an occurrence of an arbitrary rank can be given in $\mathcal{O}(n+m)$ time.

Our approach to the problem is different from all of those previously applied for compressed pattern matching (although it does relate to dynamic string equality testing considered by Mehlhorn et al. [1997] and its generalisation to pattern matching by Alstrup et al. [2000]). We do *not* consider any combinatorial properties of the encoded strings. Instead, we analyse and change the way in which strings are described by the SLPs in the instance. In other words, we focus on the SLPs alone, ignoring any properties of the encoded strings. Roughly speaking, our algorithm aims at having all strings in the instance compressed "in the same way." To achieve this goal, we decompress the SLPs. Since the compressed text can be exponentially long, we do this *locally*: we introduce explicit letters into the right-hand sides of the productions. Then we recompress these explicit strings uniformly: roughly, a fixed pair of letters ab is replaced by a new letter c in both the string and the pattern; such a procedure

is applied for every possible pair of letters. The compression is performed within the rules of the grammar, and often it is necessary to modify the grammar so that this is possible. Since such pieces of text are compressed in the same way, we can "forget" about the original substrings of the input and treat the introduced nonterminals as atomic letters. Such recompression shortens the pattern (and the text) significantly: roughly one "phase" of recompression, in which every pair of letters that was present at the beginning of the phase is compressed, shortens the encoded strings by a constant factor. The compression ends when the pattern is reduced to one letter, in which case the text is a simple SLP-like representation of all pattern occurrences. With some effort, one phase can be implemented in linear time, which yields the promised running time.

Remark 1.2. Notice that in some sense we build an SLP for both the pattern and string in a bottom-up fashion: pair compression of ab to c is in fact introducing a new nonterminal with a production $c \to ab$. This justifies the name *recompression* being used for the whole process. This is explained in detail later on.

Similar techniques. Although application of the idea of recompression to SLPS is new, related approaches were employed previously: most notably, the idea of replacing short strings by a fresh letter and iterating this procedure was used by Mehlhorn et al. [1997] in their work on data structures for equality testing for dynamic strings, as well as in the later improved implementation by Alstrup et al. [2000]).

In the area of compressed membership problems [Plandowski and Rytter 1999], from which the recompression method emerged, recent work of Lohrey and Mathissen [2011] already implemented the idea of replacing strings with fresh letters, as well as modifications of the instance so that such replacement is possible. However, the replacement was not iterated, and the newly introduced letters were not further compressed.

Additionally, a somehow similar algorithm, which replaces pairs and blocks, was proposed by Sakamoto [2005] in connection with the (approximate) construction of the smallest grammar for the input text. His algorithm was inspired by the RePair algorithm [Larsson and Moffat 1999], which is a practical grammar-based compressor. However, as the text in this case is given explicitly, the analysis is much simpler; in particular, it does not introduce the technique of modification of the grammar according to the applied compressions. Still, the analysis is based on modifying (as a mental experiment) the LZ77 representation.

Other applications of the technique. A variant of the recompression technique has been used to establish the computational complexity of the FCMP for NFAs [Jeż 2014]. This method can also be applied in the area of word equations, yielding simpler proofs and faster algorithms of many classical results in the area, such as the PSPACE algorithm for solving word equations, double exponential bound on the size of the solution, and exponential bound on the exponent of periodicity [Jeż 2013c]. Furthermore, a more tuned algorithm and detailed analysis yields a first linear-time algorithm for word equations with one variable (and arbitrarily, many occurrences of it) [Jeż 2013b]. The method can be straightforwardly applied to obtain a simple algorithm for construction of the (approximation of) smallest grammar generating a given word [Jeż 2013a].

The recompression approach can be also generalised from strings to (ordered and rooted) trees, essentially by reinterpreting the operations for trees and adding one new compression operation. In this way, the first algorithm with a guaranteed approximation ratio for the smallest tree grammar problem was devised [Jeż and Lohrey 2014]. Moreover, the PSPACE algorithm for word equation was extended to context unification, yielding a first decidability proof for this problem [Jeż 2014].

Computational model. Our algorithm uses RadixSort, and we assume that the machine word is of size $\Omega(\log(n+m))$. RadixSort can sort k numbers represented as strings of d-ary digits of lengths $\ell_1, \ell_2, \ldots, \ell_k$ in time $\mathcal{O}(d+\sum_{i=1}^{k} \ell_i)$. In particular, $n+m$ numbers of size $\mathcal{O}((n+m)^c)$ can be sorted in time $\mathcal{O}(c(n+m))$.

We assume that the alphabet of the input is $\{1, 2, \ldots, (n+m)^c\}$ for some constant c. This is not restrictive, as we can sort the letters of the input and replace them with consecutive numbers, starting with 1, in total $\mathcal{O}((n+m)\log(n+m))$ time.

The position of the first occurrence of the pattern in the text might be exponential in n, so we need to make some assumptions to be able to output such a position. Assuming that N fits in a constant amount of codewords, our algorithm can also output the position of the first/last position of the pattern.

We assume that the rules of the grammar are stored as lists so that insertion and deletion of characters can be done in constant time (assuming that a pointer to an element is provided).

Organisation of the article. In Section 2, as a toy example, we show that through using recompression, we can check the equality of two explicit strings. This introduces the first half of the main idea of recompression: iterative replacement of pairs and blocks, as well as some key ideas of the analysis. On the other hand, it completely ignores the (also crucial) way in which the SLP is refactored to match the applied recompression. In Section 3, it is explained how this approach can be extended to pattern matching, and we again consider only the case in which the text and pattern are given explicitly. Although the main idea is relatively easy, the method and the proof involve an exhaustive case inspection.

Next, in Section 4, we show how to perform the equality testing in the case of SLPs. This section introduces the second crucial half of the technique: modification of SLPs in the instance according to the compressions. This section is independent of Section 3 and can be read beforehand. In Section 5, we show how to merge the results of Sections 3 and 4, yielding an algorithm for fully compressed pattern matching, which runs in $\mathcal{O}((n+m)\log M \log(n+m))$.

Finally, in Section 6, we explain how to improve the running time from $\mathcal{O}((n+m)\log M \log(n+m))$ to $\mathcal{O}((n+m)\log M)$ when M fits in $\mathcal{O}(1)$ machine words.

2. TOY EXAMPLE: EQUALITY TESTING

In this section, we introduce the recompression technique and apply it in the trivial case of equality testing of two explicit strings—that is, when their representation is not compressed. This serves as an easy introduction. In Section 3, we take this process a step further by explaining how to perform a pattern matching for explicit strings using recompression. To stress the future connection with the pattern matching, we shall use the letters p (as *pattern*) and t (as *text*) to denote the current two strings for which we test the equality (note that they change during the run of the algorithm). By m and n, we shall denote their initial sizes and by $|p|$ and $|t|$ the current ones.

Earlier work on equality testing. In equality testing, our approach is somehow similar to the one of Mehlhorn et al. [1997] from their work on equality testing for the dynamic strings. In that setting, we are given a set of strings, initially empty, and a set of operations that add new strings to the set. We are to create a data structure that could answer whether two strings in this collection are equal or not.

The method proposed by Mehlhorn et al. [1997] is based on iterative replacement of strings: they defined a schema that replaces a string s with a string s' (where $|s'| \leq c|s|$ for some constant $c < 1$) and iterates the process until a length-1 string is obtained. Most importantly, the replacement is injective—in other words, if $s_1 \neq s_2$, then s_1 and

s_2 are replaced with different strings.[1] In this way, we calculate the unique *signature* of each string, and two strings are equal if and only if their signatures are equal.

The second important property of this schema is that the replacement is *local*: the string is first partitioned into blocks (of constant size), the assignment of the letter to a block depends only on the $\log^* n$ neighbouring letters to the left and right, and each block is replaced (by a single letter) independently.

Recompression. The recompression, as presented in this section, is a variant of this approach in which a different replacement schema is applied. To be specific, our algorithm is based on two types of compressions performed on strings:

Pair compression of ab. For two different letters ab occurring in p or t, replace each of ab in p and t by a *fresh* letter c.

Block compression of a. For each maximal block a^ℓ, with $\ell > 1$, that occurs in p or t, replace all a^ℓ's in p and t by a fresh letter a_ℓ.

By a *fresh letter*, we denote any letter that does not occur in p or t. The *ablock*a^ℓ is *maximal* when it cannot be extended by a letter a to the left or to the right. We adopt the following notational convention throughout the rest of the article: whenever we refer to a letter a_ℓ, it means that the block compression was done for a and a_ℓ is the letter that replaced a^ℓ.

Clearly, both compressions preserve the equality of strings

LEMMA 2.1. *Let p', t' be obtained from p and t by a pair compression (or block compression). Then, $p = t$ if and only if $p' = t'$.*

Using those two operations, we can define the algorithm for testing the equality of two strings.

ALGORITHM 1: SimpleEqualityTesting: outline

```
1: while |p| > 1 and |t| > 1 do
2:     L ← list of letters occurring in t and p
3:     P ← list of pairs occurring in t and p
4:     for each a ∈ L do
5:         compress blocks of a
6:     for each ab ∈ P do
7:         compress pair ab
8: Naively check the equality and output the answer.
```

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

To implement the SimpleEqualityTesting in linear time, we want to use RadixSort on letters. However, it might be that the letters in the current string are from an interval that is much larger than $|p| + |t|$. To exclude this case, at the beginning of each phase we renumber the letters so that they are indeed numbers from an interval of size at most $|p| + |t|$. The following lemma formally states that this can be done in linear time.

LEMMA 2.2. *Without loss of generality, at the beginning of each phase the letters present in p and t form an interval $\{k+1, k+2, \ldots, k+k'\}$ for some k and $k' \leq |p| + |t|$, ensuring this takes at most $\mathcal{O}(|t'| + |p'|)$ time, where $|p'|$ and $|t'|$ are the lengths of the pattern and text at the beginning of the previous phase.*

[1]This is not a information theory problem, as we replace only strings that occur in the instance and moreover can reuse original letters. In other words, the new strings do not encode the original ones but just preserve equality.

In the first phase, we take $|p'| = |p|$ and $|t'| = |t|$.

PROOF. The proof proceeds by an induction on the number of phases.

Consider the first phase. We assumed that the input alphabet consists of letters that can be identified with a subset of $\{1, \ldots, (n+m)^c\}$. Treating them as vectors of length c over $\{0, \ldots, (n+m)-1\}$, we can sort them using RadixSort in $\mathcal{O}(c(n+m))$ time (i.e., linear one). Then we can renumber those letters as $1, 2, \ldots, k$ for some $k \leq n+m$. This takes $\mathcal{O}(n+m) = \mathcal{O}(|p|+|t|)$ time.

For the induction proof, note that we can equivalently show that at the end of the phase, we can ensure that the letters form an interval of numbers, and this takes time $\mathcal{O}(|p'|+|t'|)$, where p and t are the pattern and text at the beginning of the phase.

Suppose that at the beginning of the phase the letters formed an interval $[k+1\,..\,k+k']$. Each new letter, introduced in place of a compressed pair or block, is assigned a consecutive value, starting from $k+k'+1$, and so after the phase the letters occurring in p and t are either within $[k+1\,..\,k+k']$ (the old letters) or within an interval $[k+k'+1\,..\,k+k'']$ (the new letters), for some $k' \leq k'' \leq k'+|p|+|t|$ (the second inequality follows from the fact that the introduction of a new letter shortens p or t by at least one letter). We now want to renumber the letters so that the ones actually used in p and t form an interval of numbers (which is then of size at most $|p|+|t|$). We go through p and t, and for each letter a we increase the counter $count[a]$ by 1. We then go through $count[k+1\,..\,k+k']$ and assign consecutive numbers, starting from $k+k''+1$, to letters with nonzero count. As each such number occurs in either p or t, there are at most $|p|+|t|$ of them, so the obtained interval $[k+k'+1\,..\,k+k'+k'']$ is of size at most $|p|+|t|$. Last, we replace each letter a in p and t with the corresponding new letter.

Concerning the running time, by induction assumption $k' \leq |p'|+|t'|$, and k'' by definition is at most $k'+|p'|+|t'|$. Hence, all operations take time $\mathcal{O}(|p'|+|t'|)$, as claimed. □

The crucial property of SimpleEqualityTesting is that in each phase the lengths of p and t shorten by a constant factor.

LEMMA 2.3. *Let $|p|, |t| > 1$, and consider any two consecutive letters in one of them at the beginning of the phase. Then at least one of those letters is compressed until the end of the phase. In particular, the strings p' and t' obtained after one phase have lengths at most $\frac{2|p|+1}{3}$ and $\frac{2|t|+1}{3}$, respectively.*

PROOF. Fix the two consecutive letters of p at the beginning of the phase, as in the statement of the first claim of the lemma (the proof for t is the same). Let those letters be a and b. If $a = b$, then they are compressed during the blocks compression. Suppose that $a \neq b$. Then ab is listed in P, and we try to compress this occurrence of ab during the pair compressions. This fails if and only if one of letters from this occurrence was already compressed when we considered ab during the pair compression, which shows the first claim.

Let m_u and m_c denote the number of uncompressed and compressed letters of the pattern (in this phase). Of course, $m_u + m_c = |p|$. Consider an uncompressed letter that is not the last letter of p. The two letters to the right are compressed together: by the already proved first claim, the left one is compressed and clearly is not compressed with the uncompressed letter. Thus, with each uncompressed letter (except perhaps the last letter of p), we can associate two compressed letters to its right. Therefore, $m_u \leq 1 + \frac{|p|-1}{3} = \frac{|p|+2}{3}$ (the "+1" is for the last letter of p, which may be also uncompressed). Then, also $m_c \geq \frac{2|p|-2}{3}$. The length of the new pattern p' is equal to $|p|$ minus the number of removed letters, which is at least $m_c/2$. As $m_u + m_c = |p|$, we obtain that

after the compression

$$\begin{aligned}|p'| &\leq |p| - \frac{m_c}{2}\\ &\leq |p| - \frac{1}{2}\cdot\frac{2|p|-2}{3}\\ &= \frac{2|p|+1}{3}\ .\end{aligned}$$

The calculations for t are the same. □

With proper implementation, one phase takes $\mathcal{O}(|p| + |t|)$ time, assuming that the alphabet Σ used in p and t can be identified with numbers an interval of size $|p| + |t|$. By Lemma 2.2, this can be ensured.

LEMMA 2.4. *Assuming that at the beginning of a phase the alphabet Σ used in p and t can be identified with an interval of size at most $|p| + |t|$, one phase of* SimpleEqualityTesting *can be implemented in $\mathcal{O}(|p| + |t|)$ time.*

PROOF. We go through the p and t. Whenever we spot a pair ab (of different letters), we create a record (a, b, p), where p is the pointer to this occurrence of ab (i.e., this is a pointer to the first letter in the pair). Similarly, when we spot a maximal block a^ℓ (where $\ell > 1$), we put a record (a, ℓ, p), where p is again a link to this maximal block (say, to the first letter of the block). Clearly, this takes linear time.

We sort the triples for blocks using RadixSort (we ignore the third coordinate). Since the letters form an interval of size at most $|p| + |t|$ and blocks have length at most $|p| + |t|$, this can be done in $\mathcal{O}(|p| + |t|)$ time. Then we go through the sorted list and replace a^ℓ with a_ℓ for $\ell > 1$. Since all occurrences of a^ℓ are consecutive on the sorted list, this can be done in time $\mathcal{O}(1)$ per processed letter. Hence, the total running time is linear.

Similarly, we sort the triples of pairs. For each ab on the list, we replace all of its occurrences by a fresh letter. Note that as the list is sorted, before considering a pair $a'b'$ we either replaced all or none occurrences of a different pair ab (depending on whether ab is earlier or later in the list). Hence, this effectively implements iterated pair compression.

It might be that the occurrence of a pair ab that we want to replace is no longer there, as one of its letters (or both) were already compressed as part of occurrences of different pairs. This situation is easy to identify: when we consider a link to an occurrence of ab, we verify whether the letter at this position is indeed a and the one to the right is b. If not, then we do nothing; if so, then we replace them appropriately. □

Now, we can state the theorem describing all of the important properties of SimpleEqualityTesting.

THEOREM 2.5. SimpleEqualityTesting *runs in $\mathcal{O}(n + m)$, where m and n are the initial lengths of p and t, and tests the equality of p and t.*

PROOF. By iterative application of Lemma 2.1, each compression that is performed by SimpleEqualityTesting preserves the equality of strings, so SimpleEqualityTesting returns a proper answer. Concerning the running time, the assumptions of Lemma 2.4 are met due to Lemma 2.2, and so each phase of SimpleEqualityTesting takes time linear in the current length of p and t. And as $|p|$ and $|t|$ shorten by a constant factor in each phase (see Lemma 2.3), this takes a total of $\mathcal{O}(n + m)$ time. □

Building of a grammar. As noted in the Introduction, SimpleEqualityTesting basically generates a context-free grammar, whose some nonterminals generate p and t

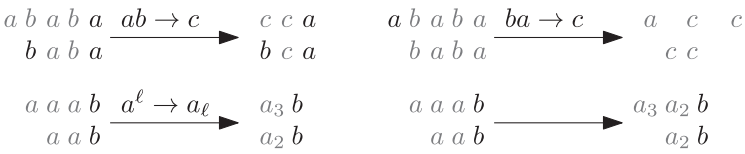

Fig. 1. Two potential problems for pattern matching based on recompression. The grey letters are to be replaced. On the left-hand side, there are occurrences of the pattern in the text, and on the right-hand side, they are lost. On the right, we present a variant that fixes the beginning in those cases.

(additionally, this context-free grammar is an SLP, which is formally defined in the Introduction as well). To be more precise, each replacement of ab by c corresponds to an introduction of a new nonterminal c with a production $c \to ab$ and replacement of each ab with c, which generates the same string as ab does. Similarly, the replacement of a^k with a_k corresponds to an introduction of a new nonterminal a_k with a rule $a_k \to a^k$.

Weight. When we think of introduced letters as nonterminals of an SLP, it is useful to store the lengths of the derived string, which alternatively can be viewed as the length of the substring of the input represented by the letter. This is formalised using *weight* of letters, which is extended to strings in a natural way. Every letter a in the input grammar has $\mathrm{w}(a) = 1$, whereas when a new letter a replaces the string w, we set $\mathrm{w}(a) = \mathrm{w}(w)$. When N (i.e., the size of the text) fits in a constant amount of code words, the weight of each letter can be calculated in constant time, so we can store the weights of the letters on the fly in a table. If this is not the case, we use a notion as a tool in the analysis.

3. TOY EXAMPLE: PATTERN MATCHING

Potential problems. The approach used in the previous section basically applies to the pattern matching as well but with one exception: we have to treat the "ends" of the pattern in a careful way. Consider $t = ababa$ and $p = baba$ (see Figure 1). Then compression of ab into c results in $t' = cca$ and $p' = bca$, which no longer occurs in t'. The other problem occurs during the block compression (which is also illustrated in Figure 1): consider $p = aab$ and $t = aaab$. After the block compression, the pattern is replaced with $p' = a_2b$ and the text with $t' = a_3b$.

In general, the problems arise because the compression in t is done partially on the p occurrence and partially outside it, so it cannot be reflected in the compression of p itself. We say that the compression *spoils pattern's beginning* (*end*) when such partial compression occurs on the pattern occurrence beginning (end, respectively). In other words, when a, b are the first and last letters of p, then we cannot perform a pair compression for ca or bc (for any letter c) or the a or b block compression. If there is no spoiling, the compression preserves the occurrences of the pattern in the text.

LEMMA 3.1. *If the pair compression (block compression) does not spoil the end or the beginning, then there is a one-to-one correspondence between occurrences of the (old) pattern in the (old) text before the compression step and occurrences of the (new) pattern in the (new) text after the compression step.*

Solving the problem. In the first example illustrated in Figure 1 (i.e., for $t = ababa$ and $p = baba$), spoiling of the pattern's beginning can be circumvented by enforcing a compression of the pair ba in the first place: when the two first letters of the pattern are replaced by a fresh letter c, then the beginning of the pattern no longer can be spoiled in this phase (as c will not be compressed in this phase). We say that a pattern's beginning (end) is *fixed* by a pair or block compression if after this compression a first (last, respectively) letter of the pattern is a fresh letter, so it is not in L and no pair

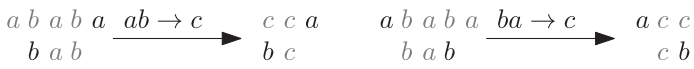

Fig. 2. Neither ab compression nor ba compression preserves the occurrences of the pattern.

containing it is in P. Our goal is to fix both the beginning and end without spoiling any of them. Then, by Lemma 3.1, the following pair and block compressions performed by SimplePatternMatching will preserve the occurrences of the pattern in the text.

Notice that at the same time, the same compression can fix the beginning and spoil the end: for instance, in the example depicted in Figure 2 for $t = ababa$ and $p = bab$, compressing ba into c fixes the beginning and spoils the end, whereas compression of ab into c spoils the beginning and fixes the end. This example demonstrates that the case in which the first and last letter of the pattern are the same is more problematic than the case in which they are different.

In general, fixing the beginning and end without prior spoiling of them requires some work. We first describe the idea behind the particular actions, then show that they indeed fix the beginning and end without spoiling them, and finally show that we can guarantee that the lengths of p and t shorten by a constant factor in a phase in this case as well.

ALGORITHM 2: SimplePatternMatching: outline

```
1:  while |p| > 1 do
2:      L ← list of letters in p, t
3:      P ← list of pairs in p, t
4:      if p[1] ≠ p[|p|] then                ▷ The first and last letter of the pattern are different
5:          FixEndsDifferent(p[1], p[|p|])
6:      else                  ▷ p[1] = p[m]  ▷ The first and last letter of the pattern are the same
7:          FixEndsSame(p[1])
8:      for a ∈ L do
9:          compress blocks of a in p and t
10:     for ab ∈ P do
11:         compress pair ab in p and t
12: verify, whether p[1] occurs in t
```

There are four main subcases to consider when trying to fix the beginning. They depend on the following:

—whether the first and last letter of the pattern are the same or not, and
—whether the first and second letter of the pattern are the same or not (i.e., whether p begins with a pair or a block).

We consider them in the order of increasing difficulty.

The first and last letter of the pattern are different. Suppose that the first and last letter of the pattern are different. Furthermore, if the first two letters of the pattern are ab for $a \neq b$, then we can fix the beginning by compressing the pair ab before any other pairs (or blocks) are compressed. This will fix the beginning and not spoil the end (since the last letter is not a).

This cannot be applied when $a = b$ or, in other words, p has a leading ℓ-block of letters a for some $\ell > 1$. The problem is that each m-block for $m \geq \ell$ can begin an occurrence of the pattern in the text. The idea of the solution is to replace the leading ℓ-block of p with a_ℓ but then treat a_ℓ in both p and t as a marker of a (potential) beginning of the

pattern, meaning that each block a^m for $m > \ell$ should be replaced with a string ending with a_ℓ. To be more precise:

—for $m < \ell$, each m-block is replaced by a fresh letter a_m;
—each ℓ-block is replaced with a_ℓ; and
—for $m > \ell$, each m-block is replaced by a *pair* of letters $a_m a_\ell$, where a_m is a fresh letter.

This modifies the block compression; however, there is no reason why we needed to replace a^m by exactly one letter in the block compression. Rwo letters are fine as long as

—the replacement function is injective;
—they are shorter than the replaced text; and
—the introduced substring does not occur in p and t.

We shall not formalise this intuition; instead, the proofs will simply show that there is a one-to-one correspondence between occurrences of the pattern before and after such a modified block compression.

For instance, in the example considered in Figure 1 from the original $t = aaab$ and $p = aab$, we obtain $t' = a_3 a_2 b$ and $p' = a_2 b$; clearly, p' has an occurrence in t'. In this way, we fixed the pattern beginning (without spoiling the end).

Note that if t ends with an a block of length greater than ℓ, then we introduce ending a_ℓ in t. This a_ℓ cannot be used by any pattern occurrence, so we remove it from t.

Now it is left to fix the pattern's end, which is done in a similar way. There are two details:

—It might be that the end was already fixed during the fixing of the beginning (which happens when the first two letters ab of p are the same as the last two letters). In such a case we do nothing, as the end was already fixed.
—We may need to compress a pair including a letter introduced during the fixing of the beginning: it might be that the second to last letter of p, say b', was introduced during the fixing of the beginning. However, there is no additional difficulty in this (note that in SimpleEqualityTesting, we never compressed letters that were compressed earlier in the phase, yet no part of the analysis assumed this).

ALGORITHM 3: FixEndsDifferent(a, a') : the $a \neq a'$ are the first and last letter of p

Require: $a \neq a'$
1: $b \leftarrow p[2]$
2: **if** $a \neq b$ **then** ▷ Compress the leading pair ab
3: compress ab in t and p
4: **else** ▷ $a = b$: compress the a blocks
5: let $\ell \leftarrow$ length of the p's a-prefix
6: **for** $1 < m \leq \ell$ **do**
7: replace each maximal block a^m in p, t by a_m
8: **for** $m > \ell$ **do**
9: replace each maximal block a^m in p, t by $a_m a_\ell$
10: **if** t ends with a_ℓ **then**
11: remove this a_ℓ ▷ Cannot be used by pattern occurrence
▷ Symmetric for the ending letter

The first and last letter of the pattern are the same. The described approach does not work when the first and last letter of the pattern are the same. As a workaround, we alter the pattern so that the first and last letter are in fact different and then apply

the previous approach: let a^ℓ and a^r be the a-prefix and suffix of p (it may be that $\ell = 1$ or $r = 1$). Then we replace them with a_L and a_R, which are different, even if $\ell = r$. The a_L and a_R are "markers," and their occurrences in t denote that a pattern occurrence can begin or end here (so they have a role similar to a_ℓ in FixEndsDifferent). Now we make a block compression for a in which we additionally put those markers: a^m, for $m \geq \ell, r$, is replaced by $a_R a_m a_L$. This reflects the fact that a^m can *both* begin and end the pattern occurrence—the former consumes ending a_L and the latter the leading a_R. The exact replacement of a^m for $m \leq \max(\ell, r)$ depends on whether $\ell < r$, $\ell = r$, or $\ell > r$; for instance, when $\ell = r$:

—for $m < \ell$, we replace m-blocks with a_m; and
—for $m = \ell$, we replace ℓ-blocks with $a_R a_L$.

The other replacement schemes are similar (see Lemma 3.2 and FixEndsSame for details).

Note that in this way, it is possible that t begins with a_R or ends with a_L, none of which can be used by a pattern occurrence. For simplicity, we remove such occurrences of a_R and a_L.

For $\ell = r = 1$, this actually enlarges p and t, and for $\ell = r = 2$ not always shortens them. To fix this, we make an additional round of pair replacement immediately after the blocks replacement: we make the compression of pairs of the form $\{a_L b \mid b \in \Sigma \setminus \{a_L\}\}$ (note that those pairs cannot overlap, so all of them can be replaced in parallel), followed by compression of pairs $\{ba_R \mid b \in \Sigma \setminus \{a_R\}\}$. The latter compression allows compression of the letters introduced in this phase—that is, $a_L b a_R$ is first compressed into ca_R and then into c'. It can be routinely checked that this schema shortens both p and t: as an example, consider bab', which is first replaced with $ba_R a_L b'$, then by $ba_R c$, and finally with $c'c$, which is shorter than bab'; other cases are analysed similarly (see Lemma 3.5 for details). When a block compression and pair compression is applied afterward, Lemma 2.3 still holds, although for a smaller constant (i.e., pattern and text shorten by a constant factor in each phase).

Concerning the occurrences of the pattern, when pattern is reduced to single letter, say c, each occurrence of c in t corresponds to one occurrence of a pattern in the original text. There is also a special case: when pattern is reduced to a^ℓ, then after the blocks compression, we list all occurrences of letters a_m for $m \geq \ell$ within t, each of them represents $m - \ell + 1$ occurrences of the original pattern in the original text.

Analysis

LEMMA 3.2. *When the first and last letter of the pattern are different, in $\mathcal{O}(|p| + |t|)$ time* FixEndsSame *fixes both the beginning and end without prior spoiling of them.*

There is a one-to-one correspondence between the pattern occurrences in the new text and old pattern occurrences in the old text.

PROOF. The manner of performing the appropriate operations has been described; now we will analyse their properties and implementations.

Fixing the beginning. Let b be the second letter of p. First, suppose that $b \neq a$. We (naively) perform the compression of the pair ab by reading both p and t from the left to the right and changing each ab into c as we go. Clearly, the running time is $\mathcal{O}(|p| + |t|)$. Note that the beginning and end were not spoiled in the process, and so there is a one-to-one correspondence of new and old pattern occurrences (see Lemma 3.1).

Now, suppose that $b = a$, and let a^ℓ be the a-prefix of p. Then, we perform the compression of blocks for the letter a as in SimpleEqualityTesting, with the additional

replacement of a^m with $a_m a_\ell$ for $m > \ell$. Identifying blocks a^m for $m > \ell$ is done on the way, as we sort according to length of blocks anyway.

We show that no pattern occurrence was lost, neither has any new pattern occurrence been introduced. Therefore, let $t = w_1 a^m w_2 w_3$ and $p = a^\ell w_2$, where w_2 does not begin or end with a; first, consider the case in which $m > \ell$. Observe that as the first and last letter of the pattern are different, we know that $w_2 \neq \epsilon$. Let w_i be replaced by w'_i; note that the two occurrences of w_2 in p and t are replaced with the same string (as they do not begin or end with a). Then, the new text is $t' = w'_1 a_m a_\ell w'_2 w'_3$, and the new pattern is $p' = a_\ell w'_2$; thus, there is a pattern occurrence in the new text. The case in which $m = \ell$ is shown in the same way.

Conversely, let $w'_1 a_\ell w'_2 w'_3$ be the new text and $a_\ell w'_2$ the new pattern. The pattern was obtained from $a^\ell w_2$ for some w_2. Furthermore, $w'_1 a^\ell$ was obtained from some $w_1 a^m$ for $m \geq \ell$ (this is the only way to obtain a_ℓ), and the only way to obtain w'_2 is from the same w_2. Hence, no new pattern occurrence has been introduced.

This fixes the pattern beginning, and since the last letter of p is not a, it did not spoil the pattern end.

Fixing the end. We want to apply the same procedure at the end of the p. However, there can be some perturbation, as fixing the beginning might have influenced the end. Let $b'a'$ be the two last letters of p:

—The last letter could have been compressed already, which happens only when $b'a' = ab$. In this case, we got lucky and make no additional compression, as the end of the pattern has been already fixed.
—The second to last letter (b') of p was compressed (not with the last letter) into the letter c (either due to pair compression or block compression). In this case, we make the compression of the pair ca' even though c is a fresh letter. Note that as c is the first letter of this pair, this will not spoil the beginning of the pattern.

The rest of the cases, as well as the analysis of the preceding exceptions, are the same as those in the case of fixing the beginning. □

Now we consider the more involved case in which the first and last letter of the pattern are the same.

Remark 3.3. Observe that the treatment in FixEndsSame is not symmetrical for beginning a^ℓ and ending a^r. This is because we want the weights of letters, particularly a_L and a_R, to be natural numbers, and we are interested in positions of beginnings of the pattern (and not its ends). The approach in FixEndsSame allows a simple weight assignment in which $\mathrm{w}(a_L) = L\,\mathrm{w}(a)$ and $\mathrm{w}(a_R) = 0$.

LEMMA 3.4. *When the first and last letter of the pattern are equal, in $\mathcal{O}(|p| + |t|)$ time* FixEndsSame *fixes both the beginning and end without prior spoiling of them.*

There is a one-to-one correspondence between the pattern occurrences in the new text and old pattern occurrences in the old text.

PROOF. Let the first (and last) letter of the pattern be a. There is a simple special case when $p \in a^*$. Then it is enough to perform a usual compression of *ablocksandmarktheletters* a_m for $m \geq \ell$; each such letter corresponds to $m - \ell + 1$ occurrences of the pattern. To this end, we perform the a-blocks compression (for blocks of a only), as in SimpleEqualityTesting. This includes the sorting of blocks according to their length. Hence, blocks of length at least ℓ can be identified and marked in time $\mathcal{O}(|p| + |t|)$.

Now consider the case in which the pattern has some letter other than a—that is, $p = a^\ell u a^r$ where $u \neq \epsilon$ and u does not begin or end with a. The main

ALGORITHM 4: FixEndsSame(a)

Require: first and last letter of p is a

let $\ell \leftarrow$ the length of p's a-prefix, $r \leftarrow$ the length of a-suffix
replace the leading a^ℓ and ending a^r in p by a_L and a_R
for $1 < m < \min(\ell, r)$ **do**
    replace each maximal a^m in p, t by a_m
for $m > \max(\ell, r)$ **do**
    replace each maximal a^m in p, t by $a_R a_m a_L$
▷ The following actions depend on the relation between ℓ and r.
if $\ell = r$ **then**
    replace each maximal a^ℓ in p, t by $a_R a_L$
if $\ell < r$ **then**
    replace each maximal a^ℓ in p, t by a_L
    for $r > m > \ell$ **do**
        replace each maximal a^m in p, t by $a_m a_L$
    replace each maximal a^r in p, t by $a_R a_r a_L$
if $\ell > r$ **then**
    for $r \leq m < \ell$ **do**
        replace each maximal a^m in p, t by $a_R a_m$
    replace each maximal a^ℓ in p, t by $a_R a_L$
▷ End of replacement
if t ends with a_L **then**
    remove this a_L
if t begins with a_R **then**
    remove this a_R
compress all pairs of the form $a_L b$ with $b \in \Sigma \setminus \{a_L\}$
compress all pairs of the form $b a_R$ with $b \in \Sigma \setminus \{a_R\}$
if $1 = r < \ell$ **then**
    compress all pairs of the form $a_1 b$ with $b \in \Sigma \setminus \{a_1\}$

principle of the replacement was already discussed: first, a tuned version of the *a-blockscompressionisperformed*, *whichintroducesmarkers* a_L and a_R denoting the pattern beginning and end, respectively; then a compression of the pairs of the form $\{a_L b \mid b \in \Sigma \setminus \{a_L\}\}$; and finally $\{b a_R \mid b \in \Sigma \setminus \{a_R\}\}$ is performed.

Although the block compression scheme was already given for $r = \ell$, the ones for $\ell > r$ and $r < \ell$ were not, so we begin with their precise description (see Algorithm 3).

The replacement of blocks for $\ell < r$ is as follows:

—for $m < \ell$, maximal blocks a^m are replaced with a_m;
—for $m = \ell$, maximal blocks a^ℓ are replaced with a_L;
—for $\ell < m < r$, maximal blocks a^m are replaced with $a_m a_L$; and
—for $m \geq r$, maximal blocks a^m are replaced with $a_R a_m a_L$.

As in the case of the normal block compression, for $m = 1$ we identify a_1 with a (and do not make any replacement) and further in the phase allow the compression of pairs including a.

The compression of blocks for $r < \ell$ is similar:

—for $m < r$, blocks a^m are replaced with a_m;
—for $r \leq m < \ell$, blocks a^m are replaced with $a_R a_m$;
—for $m = \ell$, blocks a^ℓ are replaced with $a_R a_L$; and
—for $m > \ell$, blocks a^m are replaced with $a_R a_m a_L$.

Again, we identify a_1 with a (and do not make any replacement) and allow further compression of pairs including a.

After the block compression, regardless of the actual scheme, we compress pairs of the form $\{a_L b \mid b \in \Sigma \setminus \{a_L\}\}$ and then $\{ba_R \mid b \in \Sigma \setminus \{a_R\}\}$. Since in one such group pairs do not overlap, this can be done easily in linear time using RadixSort, as in the case of pair compression in SimpleEqualityTesting. (When compressing the second group of pairs, we allow compression of letters introduced in the compression in the first group.)

Finally, there is a special case: when $1 = r < \ell$, the compression of the pairs $\{a_1 b \mid b \in \Sigma \setminus a_1\}$ is also performed (this is needed, because so far a single a is replaced with $a_R a_1$ and a_R is compressed with the letter to the left). The running time is again linear. When t after the block compression begins (ends) with a_R (a_L, respectively), we remove it from t, as this letter cannot be used by any pattern occurrence anyway.

Clearly, both the beginning and end were fixed during the block compression, and we still need to guarantee that pattern occurrences were not lost or gained in the process. The argument is similar to that in Lemma 3.2. Therefore, let $p = a^\ell w_2 a^r$; where $w_2 \neq \epsilon$ and it does not begin or end with a, we can make this assumption as $p \notin a^*$. Let $t = w_1 a^m w_2 a^n w_3$, where $m \geq \ell$ and $n \geq r$. There are several cases: we focus on one, and the others are shown in the same way. Suppose that $m > \ell > r$ and $\ell > n > r$. Let w_i be replaced by w_i'; note that as w_2 does not begin or end with a, indeed both occurrences in p and t are replaced with the same string. Then, $p' = a_L w_2' a_R$, whereas $t' = w_1' a_R a_m a_L w_2' a_R a_n w_3'$, so there is an occurrence of the pattern. The other cases are shown similarly.

In the other direction, suppose that $p' = a_L w_2' a_R$ occurs in $t' = w_1' a_L w_2' a_R w_3'$. Observe that w_2' in both was obtained from the same w_2; furthermore, the only way to obtain a_L (a_R) in t' is from a^m (a^n, respectively) for some $m \geq \ell$ (some $n \geq r$, respectively). Thus, $p = a^\ell w_2 a^r$ appeared in $t = w_1 a^m w_2 a^n w_3$.

Thus, it is left to show that the following pair compressions do not spoil the beginning or end of the (new) pattern. Consider the first compression of the pairs of the form $\{a_L b \mid b \in \Sigma \setminus \{a_L\}\}$: is it possible that it spoils the end? Spoiling of the end happens when the last letter of the pattern (i.e., a_R) is compressed with a letter to its right. Hence, $a_L = a_R$, which is not possible, as a_L and a_R are different symbols. Therefore, consider the second compression phase in which pairs of the form $\{ba_R \mid b \in \Sigma \setminus a_R\}$ are compressed. Suppose that the beginning was spoiled in the process. Let b be the letter compressed with the leading a_L in the pattern (by the assumption that $p \notin a^*$, such b exists), and let c be the fresh letter that replaced $a_L b$. Then, the beginning is spoiled when the pair of the form xc is compressed, but this implies $a_R = c$, which is not possible. Last, consider the special case (i.e., $r = 1 < \ell$) in which pairs of the form $a_1 x$ additionally were compressed. This cannot spoil the end, as the last letter of p is not a_1. Suppose that this spoils the beginning. We already know that $a_L b$ was replaced with c. As already shown, c could not be compressed with the letter to the left; however, it is possible that it was compressed with the letter to the right and replaced with c'. Still, the only possible way to spoil the beginning is to compress $a_1 c$ or $a_1 c'$, depending on the case. In both cases, this implies that before the first compression phase, there was a substring $a_R a_1 a_L b$ in p, which contradicts our replacement scheme. □

Shortening

Now, when the whole replacement scheme is defined, it is time to show that fixing preserves the main property of original SimpleEqualityTesting: that in each round the lengths of p and t are reduced by a constant factor (see Lemma 2.3). Roughly, our replacement schema took care of that: for instance, even though we replaced a single a with $a_R a_L$, we made sure that a_R is compressed with a previous letter and a_L is merged with a following letter. Effectively, we replaced three letters with two. This is slightly

weaker than replacing two letters with one but still shortens the string by a constant factor. The other cases are analysed similarly. The following lemma takes care of the details.

LEMMA 3.5. *When* $|p|, |t| > 1$, *then one phase of* SimplePatternMatching *shortens those lengths by a constant factor.*

PROOF. We group the compressed substrings into *fragments*; one fragment shall intuitively correspond to a (short) substring that was compressed into a shorter one. Letters that were not altered are not assigned to fragments. Formally, we show that there is a grouping of letters in p and t into fragments (in the beginning of the phase) such that

—(Fra 1) there are no consecutive letters not assigned to fragments; and
—(Fra 2) each fragment w is of length at least two, and until the end of the phase it is replaced with a substring of length at most $\min(|w| - 1, 3)$.

The somewhat cryptic (Fra 2) is a short way of saying that we disallow fragments of length 1, fragments of length 2 are replaced with a single letter, fragments of length 3 are replaced with at most two letters, and fragments of length 4 or more are replaced with at most three letters. In this way, a similar reasoning as in Lemma 2.3 yields that one phase of SimplePatternMatching shortens both p and t by a constant factor (although smaller than in case of SimpleEqualityTesting).

So it is left to show that fixing, followed by block compression and pair compression, allows grouping into fragments satisfying (Fra 1–Fra 2). The analysis splits, depending on whether the first and last letter of p are different or not.

We first investigate the simpler case, in which the first and last letter of the pattern are different. A fragment is created for each substring that is substituted with a smaller one (i.e., a block or a compressed pair). Then (Fra 1) is equivalent to saying that there are no two consecutive letters in p or t that were both not compressed. A reasoning identical to the one in Lemma 2.3 shows that this is the case.

Concerning (Fra 2), we focus on the changes performed during the fixing of the beginning; the analysis is similar for the fixing of the end. First observe that indeed each fragment is of length at least 2. Moreover, when the first and second letter of p are different, each fragment is replaced with a single letter, so (Fra 2) holds. When the first and second letter of p are identical, then this analysis does not apply to a^m that are replaced with $a_m a_\ell$. But this happens for $m > \ell > 1$ (i.e., $m \geq 3$). So in this case, (Fra 2) holds as well.

So we now move to the more difficult case, when the first and last letter of the pattern are the same (say it is a letter a). In general, the proof follows a similar idea, but we need to accommodate the special actions that were performed during the fixing of the beginning and end.

Except for blocks of a (and perhaps letters neighbouring them), all fragments are defined for the input strings in the same way—that is, each pair or block compressed after FixEndsSame forms a fragment. As in the preceding case, it can be showed that for them (Fra 2) holds. Therefore, it is left to define the fragments for blocks of a and letters neighbouring them.

To define fragments, let us first consider a more coarser notion, called *clusters*. Clusters are defined using the smallest transitive and symmetric relation satisfying the following conditions:

—Two neighbouring a's are in the same cluster.
—If this a will be replaced with a string containing a_R, then the letter to its left is in the same cluster as this a as well.

—If this a will be replaced with a string containing a_L, then the letter to its right is in the same cluster as well.
—If $1 = r < \ell$, then each a is in the same cluster with the letter to its right.

As an example, if $m > \ell, r$, then all letters in ba^mca^nd are in the same cluster. Note that if $r > 1$, then a maximal block consisting of a single letter a does not end up in a cluster, and if $r = 1$, then it is in a cluster. This is a small technicality that shall be addressed at the appropriate moment.

From the definition, it follows that the maximal cluster if of the form

$$x^{(0)}a^{m_1}x^{(1)}a^{m_2}x^{(2)}\cdots x^{(k-1)}a^{m_k}x^{(k)}$$

for some $k \geq 1$, where $x^{(1)}, x^{(2)}, \ldots, x^{(k-1)}$ are single letters and $x^{(0)}$ and $x^{(k)}$ are single letters or ϵ. Then define the first fragment as $x^{(0)}a^{m_1}x^{(1)}$ and each other as $a^{m_i}x^{(i)}$. It remains to be seen that such defined fragments satisfy (Fra 1–Fra 2).

To see that (Fra 1) holds, first observe that if a letter $b \neq a$ is compressed during FixEndsSame, then it is in a cluster: it could be compressed with a_R (to its right) or a_L (to its left) or a_1 (to its left), but from the definition of the cluster, it is included in the cluster. Then, considering that any two consecutive letters unassigned to fragments were not altered during FixEndsSame, the same analysis as in Lemma 2.3 shows that at least one of them was compressed during the rest of SimplePatternMatching, so it was assigned to a fragment, which is a contradiction.

We now show that (Fra 2) holds for the fragments defined using clusters (other fragments were already considered). If during FixEndsSame a block a^{m_i} introduces a letter a_L to the right, then $x^{(i)} \neq \epsilon$—that is, it is a letter, and this a_L is compressed with $x^{(i)}$ (the only exception: a^{m_i} ends t, in which case this a_L is removed from t). Similarly, if it introduces a_R to the left, it is compressed with $x^{(i-1)}$ (or removed). Then, a^{m_i} is replaced with a single letter or ϵ (and in the special case $m_i = r = 1 < \ell$, it is replaced with a_1, which is then compressed with $x^{(i)}$). Hence, the resulting string is

$$y^{(0)}a_{m_1}y^{(1)}a_{m_2}y^{(2)}\cdots y^{(k+1)}a_{m_k}y^{(k)},$$

where $|y^{(i)}| = |x^{(i)}|$ for each i and each a_{m_i} is either a letter or ϵ. We define the results of compressing the fragments as $y^{(0)}a_{m_1}y^{(1)}$ for the first fragment and $a_{m_i}y^{(i)}$ for each other fragment. We check (Fra 2) for them. Observe that as $|y_{m_i}|, |a_{m_i}| \leq 1$, each of the compressed fragments have length at most 3. It is left to show that the length of each fragment decreases by at least 1. Since $|y^{(i)}| = |x^{(i)}|$, it is enough to show that $|a_{m_i}| \leq |a^{m_i}| - 1 = m_i - 1$. As $|a_{m_i}| \leq 1$, this is true for $m_i \geq 2$, so we are left with $m_i = 1$—that is, a single a that is a maximal block. We need to show that $a_{m_i} = \epsilon$. The further proof depends on the relation between ℓ, r and 1:

—If $\ell, r > 1$, then a is not going to be replaced during FixEndsSame, so this case is nonexistent.
—If $1 = \ell < r$, then a is replaced with a_L, which is merged with $x^{(i)}$, so $a_{m_i} = \epsilon$.
—If $1 = \ell = r$, then a is replaced with a_Ra_L, which are merged with $x^{(i-1)}$ and $x^{(i)}$, so $a_{m_i} = \epsilon$.
—If $1 = r < \ell$, then a is replaced with a_Ra_1, which are merged with $x^{(i-1)}$ and $x^{(i)}$, so $a_{m_i} = \epsilon$. □

Concerning other operations, they are implemented in the same way as in the case of SimpleEqualityTesting, so in particular, the pair compression and block compression run in $\mathcal{O}(|p|+|t|)$ (see Lemma 2.4). Furthermore, since the beginning and end are fixed, those operations do not spoil pattern occurrences (see Lemma 3.1).

As a result, we are able to show that SimplePatternMatching runs in linear time and preserves the occurrences of the pattern, which follows from Lemmas 3.2 and 3.4.

THEOREM 3.6. SimplePatternMatching *runs in $\mathcal{O}(n+m)$ time and correctly reports all occurrences of a pattern in a text, where n and m are the original lengths of t and p.*

The running time is clear: each phase takes linear time, and the length of text and pattern are shortened by a constant factor in a phase.

Building of a grammar and weights of letters revisited. Note that the more sophisticated replacement rules in the fixing of beginning and end endanger our view of recompression as creation of a context-free grammar for p and t. Still, this can be fixed easily.

For the fixing of the beginning when the first and last letter are different, there are symmetric actions performed at the beginning and at the end, so we focus only on the former. The problematic part is the replacement of a^m for $m > \ell$ with $a_m a_\ell$ (as in all other cases, the replacement is the same as in the case of equality testing). Then we simply declare that a_ℓ replaced a^ℓ (note that this is consistent with the fact that a^ℓ is replaced with a_ℓ) and a_m replaced $a^{m-\ell}$. Since $m > \ell$, this is well defined. In addition, the weights are defined as $\mathrm{w}(a_\ell) = \ell \cdot \mathrm{w}(a)$ and $\mathrm{w}(a_m) = (m-\ell) \cdot \mathrm{w}(a)$ for $m > \ell$.

When the first and last letter are the same, the situation is a bit more complicated. For the block replacement, similarly we declare that a_L replaces a^ℓ (and so $\mathrm{w}(a_L) = \ell \cdot \mathrm{w}(a)$), a_m the a^m for $m < \ell$ (so $\mathrm{w}(a_m) = m \cdot \mathrm{w}(a)$), and $a^{m-\ell}$ for $m > \ell$, which implies $\mathrm{w}(a_m) = (m-\ell) \cdot \mathrm{w}(a)$. Finally, to be consistent, we need to define $a_R \to \epsilon$ and $\mathrm{w}(a_R) = 0$. It can be verified by case inspection that in this way all blocks are replaced properly and the weight is preserved, except the ending block for p (for which the a^r is replaced with a_R). Although this somehow falsifies our informal claim that we create an SLP for p, this is not a problem, as the occurrences of the pattern are preserved and the weight of p decreases. (We can think that we shortened the pattern by those ending a^r letters, but the occurrences were preserved.) The removal of letters a_L from the end of text and a_R from the beginning can be viewed in the same way.

The a_R generating ϵ (and having weight 0) is a bit disturbing, but note that we enforce the compression of pairs of the form $\{ba_R \mid b \in \Sigma \setminus a_R\}$ (and if a_R is the first letter of t, then we remove it). In this way, all a_R are removed from the instance. Furthermore, when ba_R is replaced with b', we can declare that the rule for b' is $b' \to \alpha$, where b has a rule $b \to \alpha$ (and we can forget about the letter a_R altogether). In this way, no productions have ϵ at their right-hand sides, and no letters have weight 0.

4. EQUALITY TESTING FOR STRAIGHT-LINE PROGRAMS

In this section, we extend the SimpleEqualityTesting to the setting in which both p and t are given using SLPs. In particular, we introduce and describe the second important property of the recompression: local modifications of the instance so that pair and block compressions can be performed on the compressed representation directly.

4.1. Straight-Line Programmes

Formally, an SLP is a context-free grammar G over the alphabet Σ with a set of nonterminals $\{X_1, \ldots, X_k\}$, generating a one-word language. We denote the string defined by nonterminal X by $\mathrm{val}(X)$, like *value*. The *size* of the SLP is the sum of lengths of the right-hand sides of the rules. Without loss of generality, we may assume that in each rule there are at most two nonterminals (and arbitrary many letters), as each SLP can

be transformed to such a form and its size at most doubles during such a transformation. More formally, we assume that the SLP satisfies the following conditions:

—each X_i has exactly one production, which has at most two nonterminals; (1a)

—if X_j occurs in the rule for X_i, then $j < i$; and (1b)

—if $\mathrm{val}(X_i) = \epsilon$, then X_i is not on the right-hand side of any production. (1c)

We refer to these conditions collectively as (1). Note that (1) does not exclude the case when $X_i \to \epsilon$, and allowing such a possibility streamlines the analysis.

SLPs for p and t. For our purposes, it is more convenient to treat the two SLPs for p and t as a single context-free grammar G. By m, n, we denote the initial sizes of the SLPs for p and t, respectively. For simplicity, we assume that the set of nonterminals is $\{X_1, \ldots, X_{n+m}\}$, the text being given by X_{n+m} and the pattern by X_m. If the input SLP has fewer nonterminals, we can always add some dummy ones that are not used in any production. Additionally, we assume that X_m is not referenced by any other nonterminal (this simplifies the analysis). The size of G kept by the algorithm will be small: $\mathcal{O}((n+m)\log(n+m))$ (see Lemma 4.10). Recall as well that M and N are the initial lengths of the pattern and text, respectively.

Letters. During our algorithm, the alphabet Σ is increased many times, and whenever this happens, the new letter is assigned number $|\Sigma|+1$. The $|\Sigma|$ does not become large in this way: it remains of size $\mathcal{O}((n+m)\log(n+m)\log M)$ (see Lemma 4.10). Furthermore, observe that Lemma 2.2 generalises easily to SLPs; thus, without loss of generality, we may assume that in each phase Σ consists of consecutive natural numbers.

Let $X_i \to \alpha_i$ be a production in G, then a substring $u \in \Sigma^+$ of α_i occurs *explicitly* in the rule; this notion is introduced to distinguish them from the substrings of $\mathrm{val}(X_i)$.

(Non)crossing occurrences. The outline of the algorithm matches SimpleEqualityTesting; the crucial difference is the way in which we perform the compression of pairs and blocks when p and t are given as SLPs. Before we investigate this, we need to understand when the compression (of pairs and blocks) is easy to perform and when it is hard.

Suppose that we are to compress a pair ab. If b is a first letter of some $\mathrm{val}(X_i)$ and aX_i occurs explicitly in the grammar, then the compression seems hard, as it requires modification of G. Similar problems occur when: a is the last letter of some $\mathrm{val}(X_j)$ and $X_j b$ occurs in G; or $X_i X_j$ occurs in G and $\mathrm{val}(X_i)$ ends with a while $\mathrm{val}(X_j)$ begins with b. On the other hand, if none of mentioned situations occur, then replacing all explicit ab's in G does the job. This intuition is formalised in the following definition.

Definition 4.1 (*(Non)crossing Pairs*). Consider a pair ab and its fixed occurrence in $\mathrm{val}(X_i)$, where the rule for X_i is $X_i \to uX_j vX_k w$ (or $X_i \to uX_j v$ or $X_i \to u$). We say that this occurrence is

explicit ($for X_i$) if this ab comes from u, v, or w;
imlicit ($for X_i$) if this occurrence comes from $\mathrm{val}(X_j)$ or $\mathrm{val}(X_k)$; and
crossing ($for X_i$). otherwise.

A pair ab is *crossing* if it has a crossing occurrence for any X_i; it is *noncrossing* otherwise.

Unless explicitly written, we use the notions of crossing and noncrossing pairs only to pairs of *different* letters. Note that if ab occurs implicitly in some X_i, then it has an explicit or crossing occurrence in some X_j for $j < i$.

It can be easily shown that a pair ab is crossing if and only if one of the following conditions hold:

(CP 1) aX_i occurs in one of the rules and $\mathrm{val}(X_i)$ begins with b;
(CP 2) X_ib occurs in one of the rules and $\mathrm{val}(X_j)$ ends with a;
(CP 3) X_iX_j occurs in one of the rules, $\mathrm{val}(X_i)$ ends with a, and $\mathrm{val}(X_j)$ begins with b.

Therefore, when ab is crossing, in some sense it "crosses" between nonterminal and a neighbouring letter (or a nonterminal).

The notions of (non)crossing pairs is usually not applied to pairs of the form aa; instead, for a letter $a \in \Sigma$, we consider its maximal blocks as defined in earlier sections.

Definition 4.2 (*(Non)crossing Blocks*). Consider a letter a and an occurrence of a^ℓ in $\mathrm{val}(X_i)$, where the rule for X_i is $X_i \to uX_jvX_kw$ (or $X_i \to uX_jv$ or $X_i \to u$). We say that this occurrence is

explicit (*for* X_i) if this a^ℓ comes from u, v, or w;
implicit (*for* X_i) if this occurrence comes from $\mathrm{val}(X_j)$ or $\mathrm{val}(X_k)$; and
crossing (*for* X_i) otherwise.

A letter a *has a crossing block* if some a^ℓ has a crossing occurrence in some X_i; a *has no crossing blocks* otherwise.

It can be easily shown that a has a crossing block if and only if aa is a crossing pair.

Note that when a has crossing blocks, it might be that some blocks of a are part of explicit and crossing occurrences at the same time. However, when a has no crossing blocks, then a maximal explicit block of a is not part of a larger crossing block.

The crossing pairs and letters with crossing blocks are intuitively hard to compress, whereas noncrossing pairs and letters without crossing blocks are easy to compress. The good news is that we can give a bound on the number of crossing pairs and blocks in terms of n, m alone—that is, this bound is independent of the size of the grammar. This implies that even when we enlarge G during the algorithm, the bound on the number of crossing pairs is still the same. Note that the lemma allows a slightly more general form of the grammar, in which blocks a^ℓ are represented using a single symbol. Such a form occurs as an intermediate product of our algorithm, so we need to deal with it as well.

Lemma 4.3. *Consider a grammar satisfying* (1) *in which blocks of a letter additionally can be represented as a single symbol. Let the number of nonterminals of this grammar be* $n + m$. *Then there are at most* $2(n + m)$ *different letters with crossing blocks and at most* $4(n + m)$ *different crossing pairs and at most* $|G|$ *noncrossing pairs. For a letter* a, *there are at most* $|G| + 4(n + m)$ *different lengths of* a*'s maximal blocks in* p *and* t.

Proof. Observe that if a has a crossing block, then for some X_i the first or last letter of $\mathrm{val}(X_i)$ is a. Since there are $n + m$ nonterminals, there are at most $2(m + n)$ letters with crossing blocks.

If ab is a crossing pair, then it can be associated with an occurrence of some X_i in the grammar, as in (CP 1)–(CP 3). Only two different crossing pairs can be associated with a single occurrence of X_i, as the total number of occurrences of variables in the right-hand sides is at most $2(n + m)$; it follows that there are at most $4(n + m)$ occurrences of a crossing pair, so there are at most $4(n + m)$ different crossing pairs.

If ab is a noncrossing pair, then ab occurs explicitly in some of the rules of the grammar, and there are at most $|G|$ such substrings (note that when a^ℓ is represented by one symbol, it still contributes to pairs in the same way as a single a: if we replace a^ℓ with a single a, exactly the same noncrossing pairs occur in the rule).

The argument for maximal blocks of a is a little more involved. First, consider maximal blocks that have an explicit occurrence in the rules of G; for simplicity, let the nonterminals also count for ending maximal blocks and similarly for the ends of rules. Then, each letter (or block of letters that are represented as one symbol) is assigned to at most one maximal block and so there are at most $|G|$ such blocks, so at most $|G|$ different lengths. Assign other blocks to nonterminals: a crossing block a^ℓ is assigned to X_i with a rule $X_i \to uX_jvX_kw$ (the analysis for $X_i \to uX_jv$ is similar, and no block is assigned to X_i when its rule is $X_i \to u$) if a maximal block a^ℓ occurs in $\mathrm{val}(X_i)$, but it does not in $\mathrm{val}(X_j)$ or in $\mathrm{val}(X_k)$—that is, it has a crossing occurrence for X_i. This implies that this maximal block contains the first or last letter of $\mathrm{val}(X_j)$ or $\mathrm{val}(X_k)$, so there are at most four blocks assigned to this rule, which yields the desired bound of $4(n+m)$ on the number of such blocks. □

4.2. The Algorithm

When the notions of crossing and noncrossing pairs (blocks) are known, we can give some more detail of EqualityTesting. Similarly to SimpleEqualityTesting, it performs the compression in phases until one of p, t has only one letter; however, for running time reasons, it is important to distinguish between compression of noncrossing pairs and crossing ones (this is not crucial for blocks, as shown later).

ALGORITHM 5: EqualityTesting: outline

```
1: while |p|, |t| > 1 do
2:     P ← list of pairs
3:     L ← list of letters
4:     for each a ∈ L do
5:         compress blocks of a
6:     P' ← crossing pairs out of P
7:     P ← noncrossing pairs out of P
8:     for each ab ∈ P do
9:         compress pair ab
10:    for each ab ∈ P' do
11:        compress pair ab
12: Output the answer.
```

As in the case of SimpleEqualityTesting, the length of p and t shorten by a constant factor in a phase, so there are $\mathcal{O}(\log(\min(M, N)))$ many phases.

LEMMA 4.4. EqualityTesting *has* $\mathcal{O}(\min(\log M, \log N))$ *phases.*

The proof is the same as in the case of Lemma 2.3, as EqualityTesting and SimpleEqualityTesting transform p and t in the same way.

4.2.1. Compression of Noncrossing Pairs. We begin with describing the compression of a noncrossing pair ab, as it is the easiest to explain. Intuitively, when ab is a noncrossing pair, each of its occurrences in G are implicit or explicit; thus, it should be enough to compress each explicit occurrence of ab.

ALGORITHM 6: PairCompNcr(ab, c): compression of a noncrossing pair ab

```
1: for i ← 1..m+n do
2:     replace every explicit ab in the rule for X_i by c
```

As in case of SimpleEqualityTesting, the compression of all noncrossing pairs can be performed in parallel in linear time, using RadixSort to group the occurrences.

To simplify the notation, we use $PC_{ab\to c}(w)$ to denote w with each ab replaced by c. Moreover, we say that a procedure *implements the pair compression* for ab if after its application the obtained p' and t' satisfy $p' = PC_{ab\to c}(p)$ and $t' = PC_{ab\to c}(t)$.

LEMMA 4.5. *When ab is noncrossing,* PairCompNcr *properly implements the pair compression.*

PROOF. To distinguish between the nonterminals before and after the compression of ab, we use "primed" nonterminals (i.e., X'_i, for the nonterminals after this compression) and "unprimed" (i.e., X_i) for the ones before. We show by induction on i that

$$\operatorname{val}(X'_i) = PC_{ab\to c}(\operatorname{val}(X_i)); \tag{2}$$

note that $PC_{ab\to c}$ is well defined, as we assumed that $a \neq b$.

Indeed, (2) holds when the production for X_i has no nonterminal on the right-hand side, as in this case each pair ab on right-hand side of the production for X_i was replaced by c and so $\operatorname{val}(X'_i) = PC_{ab\to c}(\operatorname{val}(X_i))$.

When $X_i \to uX_jvX_kw$, then

$$\begin{aligned}\operatorname{val}(X_i) &= u\operatorname{val}(X_j)v\operatorname{val}(X_k)w \quad \text{and}\\ \operatorname{val}(X'_i) &= PC_{ab\to c}(u)\operatorname{val}(X'_j)PC_{ab\to c}(v)\operatorname{val}(X'_k)PC_{ab\to c}(w)\\ &= PC_{ab\to c}(u)PC_{ab\to c}(\operatorname{val}(X_j))PC_{ab\to c}(v)PC_{ab\to c}(\operatorname{val}(X_k))PC_{ab\to c}(w),\end{aligned}$$

with the last equality following by the induction assumption. Notice that since ab is a noncrossing pair, all occurrences of ab in $\operatorname{val}(X_i)$ are contained in u, v, w, $\operatorname{val}(X_j)$ or $\operatorname{val}(X_k)$, as otherwise ab is a crossing pair, which contradicts the assumption. Thus,

$$PC_{ab\to c}(\operatorname{val}(X_i)) = PC_{ab\to c}(u)PC_{ab\to c}(\operatorname{val}(X_j))PC_{ab\to c}(v)PC_{ab\to c}(\operatorname{val}(X_k))PC_{ab\to c}(w),$$

which shows that $PC_{ab\to c}(\operatorname{val}(X_i)) = \operatorname{val}(X'_i)$. □

As in the case of SimpleEqualityTesting, the pair compression of all noncrossing pairs can be effectively implemented, with the help of RadixSort for grouping of the occurrences.

LEMMA 4.6. *The noncrossing pairs compression can be performed in $\mathcal{O}(|G|)$ time.*

PROOF. We go through the list productions of G. Whenever we spot an explicit pair ab, we put $(a, b, 1, p)$ in the list of pairs' occurrences, where 1 indicates that this occurrence is noncrossing and p is the pointer to the occurrence in G.

It is easy to list the crossing pairs. We begin by calculating for each nonterminal X_i the first and last letter of $\operatorname{val}(X_i)$, which is done in a bottom-up fashion. Then for each nonterminal X_i occurring in the right-hand side of some rule, we list the crossing pairs that it creates, as listed in (CP 1)–(CP 3). To be more precise, when a is the first and b the last letter of $\operatorname{val}(X_i)$, we look at the letters preceding and succeeding this occurrence of X_i (or the last and first letters of nonterminals preceding or succeeding X_i); let them be a' and b'. We then create tuples $(a', a, 0, p)$ and $(b, b', 0, p)$: the flag 0 indicates that these pairs are crossing and the pointer p is not important, as it is not going to be used for anything.

Then we sort all of these tuples lexicographically, using RadixSort in $\mathcal{O}(|G|)$ time: by Lemma 4.10, the size of Σ is polynomial in $n + m$, so RadixSort sorts the tuples in $\mathcal{O}(|G| + n + m) = \mathcal{O}(|G|)$ time. Thus, for each pair, we obtain a list of its occurrences. Moreover, we can establish in $\mathcal{O}(|G|)$ time which pairs are crossing and which are noncrossing: since $0 < 1$, the first occurrence of ab on the list will have 0 on the third coordinate of the tuple if and only if the pair ab is crossing.

For a fixed noncrossing pair ab, the compression is performed as in the case of SimpleEqualityTesting(see Lemma 2.4). We go through the associated list and use pointers to localise and replace all occurrences of ab. As in SimpleEqualityTesting, if any of the letters of this occurrence were already compressed, we do nothing. Such the situation can be easily verified, we simply look at whether the pointed letter is a and the one to the right in the rule is b, which is done by comparing the occurrence of the pair with the letters in the tuple representing the pair. For a crossing pair, we do nothing.

The correctness follows in the same way as in Lemma 2.4; it only remains to estimate the running time. Since rules of G are organised as lists, the pointers can be manipulated in constant time, so the whole procedure takes $\mathcal{O}(|G|)$ time. □

4.2.2. Compression of Crossing Pairs. We intend to reduce the case of crossing pairs to the case of noncrossing ones—that is, given a crossing pair, we want to "uncross" it and then compress using the procedure for compression of noncrossing pairs (i.e., PairCompNcr). Note that we *do not* uncross all pairs in one go; instead, we fix the pair, make it uncrossing, and then compress it.

Let ab be a crossing pair. Suppose that this is because of (CP 1)—that is, aX_i occurs in some rule and $\mathrm{val}(X_i) = bw$ for some nonterminal X_i. To remedy this, we "left-pop" the leading b from X_i: we modify G so that $\mathrm{val}(X_i) = w$ and replace each X_i with bX_i in the rules' bodies. We apply this procedure to each nonterminal in an increasing order.

It turns out that the condition that X_i is to the right of a is not needed; we left-pop b whenever $\mathrm{val}(X_i)$ begins with b. A symmetrical procedure is applied for a letter a and nonterminals X_i such that $\mathrm{val}(X_i) = w'a$. It can be easily shown that after left-popping b and right-popping a, the pair ab is no longer crossing; therefore, it can be compressed using an approach similar to the one in Algorithm 6.

Uncrossing a pair ab works for a fixed pair ab, so it has to be applied to each crossing pair separately. It would be good to uncross several pairs at the same time. In general, it seems difficult (or even impossible) to uncross an arbitrary set of pairs at the same time. Still, parallel uncrossing can be done for group of pairs of a specific form: when we partition the alphabet Σ to Σ_ℓ and Σ_r, pairs from $\Sigma_\ell\Sigma_r$ can be uncrossed in parallel, as occurrences of pairs from $\Sigma_\ell\Sigma_r$ cannot overlap, moreover, we pop to the left only letters from Σ_ℓ and to the right only letters from Σ_r, so for each letter we want to perform at most one pop. Furthermore, using a general construction (based on binary expansion of numbers), we can find $\mathcal{O}(\log(n+m))$ partitions such that each of $4(n+m)$ crossing pairs is included in at least one of those partitions.

As a last remark, note that letters should not be popped from X_m and X_{n+m}. On one hand, those nonterminals are not used in the rules, so they cannot be used to create a crossing pair; on the other hand, since they define p and t, we should not apply popping to them, as this would change the text or pattern. (This is the place in which we use the assumption that X_m does not occur in the rules' bodies.)

LEMMA 4.7. *If Σ_ℓ and Σ_r are disjoint, after* Pop(Σ_ℓ, Σ_r) *no pair in $\Sigma_\ell\Sigma_r$ is crossing. Furthermore,* $\mathrm{val}(X_m)$ *and* $\mathrm{val}(X_{n+m})$ *have not changed.*
Pop *runs in time $\mathcal{O}(n+m)$ and introduces at most $4(n+m)$ letters to G.*

PROOF. Let β_i be the string popped from α_i to the left—that is, a letter from Σ_r or ϵ and similarly γ_i the string popped to the right. As before, by X_i, α_i, and so forth, we denote the nonterminals, rules, and the like in the input and X'_i, α'_i, and the like the ones in the output. We use the term i-th rule (and i-th string) to denote the contemporary value of the rule for X_i and the derived string. As a first step of the proof, we show the following by a simple induction on i:

(1) When we have processed the i-th rule, for $j > i$ the j-th string is equal to $\mathrm{val}(X_i)$, i.e. as in the input.

ALGORITHM 7: Pop(Σ_ℓ, Σ_r): popping letters from Σ_ℓ and Σ_r

```
1:  for i ← 1..n+m, except m and m+n do
2:      let X_i → α_i and b the first letter of α_i
3:      if b ∈ Σ_r then                                   ▷ Left-popping
4:          remove leading b from α_i
5:          replace X_i in G's rules by bX_i
6:          if α_i = ϵ then
7:              remove X_i from rules of G                ▷ X_i is empty
8:          else
9:              let a be the last letter of α_i
10:             if a ∈ Σ_ℓ then                           ▷ Right-popping
11:                 remove ending a from α_i
12:                 replace X_i in G's rules by X_i a
13:                 if α_i = ϵ then
14:                     remove X_i from rules of G        ▷ X_i is empty
```

(2) $\beta_i \neq \epsilon$ if and only if $\mathrm{val}(X_i)$ begins with a letter from Σ_r; additionally, this letter is β_i.
(3) $\gamma_i \neq \epsilon$ if and only if $\mathrm{val}(X_i)$ ends with a letter from Σ_ℓ; additionally, this letter is γ_i.
(4) $\mathrm{val}(X_i) = \beta_i \,\mathrm{val}(X'_i)\gamma_i$.

To see (1), observe that if we remove β_i from the front of i-th rule, we also replace X_i with $\beta_i X_i$; the same applies to popping letters to the right. In addition, X_i is removed from the rules if and only if i-th string is ϵ. To see (2), note that by induction assumption on (1) before considering i-th rule, the i-th string is $\mathrm{val}(X_i)$. Thus, if the i-th rule begins with a letter, it is the first letter of $\mathrm{val}(X_i)$ and we are done (as it is popped). The remaining case is that the i-th rule begins with a nonterminal, say X'_j. But this means that we did not pop a letter to the left from the j-th rule when it was considered. Therefore, by (4), the $\mathrm{val}(X'_j)$ and $\mathrm{val}(X_j)$ begin with the same letter, which is the same as the first letter of $\mathrm{val}(X_i)$, so it is in Σ_r, which is a contradiction with (2) for X_j. The same analysis applies to (3) as well. Finally, for (4), note that the i-th string does not change when we consider $j \neq i$. Thus, the only changes to the i-th rule are done when the algorithm considers it and the claim is obvious.

Returning to the main claim, we want to show that it is impossible that after Pop there is a crossing pair (i.e., that one of (CP 1)–(CP 3) holds). Suppose for the sake of contradiction that (CP 1) holds—that is, for some $j < i$, the aX'_j occurs in the rule for X'_i, where $\mathrm{val}(X'_j)$ begins with $b \in \Sigma_r$. By (4), we know that $\beta_j \,\mathrm{val}(X'_j)\gamma_j = \mathrm{val}(X_j)$. Therefore, $\mathrm{val}(X_j)$ begins with a letter from Σ_r: if β_j is a letter, then it is from Σ_r by (1), and if not, then we know that $\mathrm{val}(X'_j)$ begins with a letter from Σ_r. Then, by (2), we popped a letter from X_j and so β_j is a letter. But then the letter to the left of X'_j is this β_j, and we assume that it is from Σ_ℓ, which is a contradiction, as $\Sigma_\ell \cap \Sigma_r = \emptyset$.

The other cases are shown in the same way.

Concerning the running time, note that we do not need to read the whole G: it is enough to read the first and last letter in each rule. To perform the replacement, for each nonterminal X_i we keep a list of pointers to its occurrences so that X_i can be replaced with aX_ib in $\mathcal{O}(1)$ time, and there are at most $2(n+m)$ occurrences of nonterminals in G.

Note that at most two letters are popped from each nonterminal, so at most $4(n+m)$ are introduced to G. □

After $\mathsf{Pop}(\Sigma_\ell, \Sigma_r)$, the pairs $ab \in \Sigma_\ell\Sigma_r$ are no longer crossing, so we can compress them. Since such pairs do not overlap, this can be done in parallel in linear time, similarly as in Lemma 4.6 and in time $\mathcal{O}(|G|)$.

The obvious way to compress all crossing pairs is to make a series of partitions $(\Sigma_\ell^{(1)}, \Sigma_r^{(1)})$, $(\Sigma_\ell^{(2)}, \Sigma_r^{(2)})$, ... such that each crossing pair is in at least one of those partitions. Since there are $4(n+m)$ crossing pairs (see Lemma 4.3), in the naive solution we would have $4(n+m)$ partitions. However, we can reduce this number to $\mathcal{O}(\log(n+m))$. Roughly, we make the partitions according to the binary expansion of notations of letters. For $i = 1, \ldots \lceil \log|\Sigma| \rceil$, define:

—$\Sigma_\ell^{(2i-1)} = \Sigma_r^{(2i)}$ consist of elements of Σ that have 0 at the i-th position in the binary notation (counting from the least significant digit); and
—$\Sigma_r^{(2i-1)} = \Sigma_\ell^{(2i)}$ consist of elements of Σ that have 1 at the i-th position in the binary notation.

For $a \neq b$, their binary notations differ at some position, so the pair ab is in some group $\Sigma_\ell^{(j)}\Sigma_r^{(j)}$. Note that ab may be in many $\Sigma_\ell^{(j)}\Sigma_r^{(j)}$, but it will be compressed only once for the smallest possible j. Thus, we define the lists P_j, where we include ab from P' (i.e., a crossing pair) in the group P_j when j is the smallest number such that $a \in \Sigma_\ell^{(j)}$ and $b \in \Sigma_r^{(j)}$. Observe that using standard bit operations, we can calculate the first position on which a and b differ (and so also j) for ab in constant time. Finally, since $|\Sigma| = \mathcal{O}((n+m)\log(n+m)\log M) = \mathcal{O}((n+m)^3)$ by Lemma 4.10, we create $\mathcal{O}(\log(n+m))$ partitions for P'.

ALGORITHM 8: PairComp compressing all crossing pairs

```
1: find partitions of Σ into {Σ_ℓ^(i), Σ_r^(i)}, i ∈ O(log(n+m))      ▷ see earlier discussion
2: partition the crossing pairs into groups P_1, P_2, ..., P_i according to partitions of Σ
3: for j ← 1..i do
4:     run Pop(Σ_ℓ^(j), Σ_r^(j))
5:     compress each of the pairs ab ∈ P_j      ▷ P_j is more or less P' ∩ Σ_ℓ^(j)Σ_r^(j)
```

Concerning the running time of an efficient implementation, we first compute the list of explicit occurrences of each crossing pair, which is done in linear time using the same methods as in the case of noncrossing pairs (see Lemma 4.6) and divide those pairs into groups in linear time as well. For each group P_j, which corresponds to a partition $\Sigma_\ell^{(j)}, \Sigma_r^{(j)}$, we first apply $\mathsf{Pop}(\Sigma_\ell^{(j)}, \Sigma_r^{(j)})$ and then compress all pairs from P_j; each of those operations takes linear time. However, Pop creates new explicit occurrences of pairs, which should also be compressed. Still, we can easily identify those new occurrences and assign them to appropriate groups. Re-sorting each group before the compression ensures that we replace the pairs appropriately.

LEMMA 4.8. *The* PairComp *properly compresses all crossing pairs. It runs in* $\mathcal{O}(|G| + (n+m)\log(n+m))$ *time. It introduces* $\mathcal{O}(\log(n+m))$ *letters to each rule.*

PROOF. The sorted list of all occurrences of each crossing pair is obtained as a by-product of creation of a similar list for noncrossing pairs (see Lemma 4.6). Each pair ab is assigned to the appropriate group P_j (according to the partition for $\Sigma_\ell^{(j)}, \Sigma_r^{(j)}$) in constant time: we just need to find the first bit on which a and b differ, which can be done in constant time using standard operations on machine words.

We analyse the processing of a single group P_j; by p_j, we denote its initial size and by p'_j its size when it is processed. By induction on the number of the group (j), we show the following claim:

(P 1) Compression of pairs from one group P_j—that is, lines 4 and 5—is done in time $\mathcal{O}(p'_j + n + m)$.

First, by Lemma 4.7, the application of $\mathsf{Pop}(\Sigma_\ell^{(j)}, \Sigma_r^{(j)})$ takes time $\mathcal{O}(n+m)$, and afterward the pairs from P_j are noncrossing. Note that $\mathsf{Pop}(\Sigma_\ell^{(j)}, \Sigma_r^{(j)})$ introduces new explicit pairs to G: when we replace X_i by bX_i and a is a letter to the left of X_j, a new explicit pair ab occurs. In constant time, we can decide to which $P_{j'}$ this pair should belong. We simply add it an appropriate tuple $(a, b, 1, p)$ to the list $P_{j'}$ (which makes the list $P_{j'}$ unsorted). There two remarks: first, by inductive assumption on (P 1), all occurrences of pairs from $P_{j''}$ for $j'' < j$ were already replaced. Thus, if $j' > j$, then this pair should not be compressed at all (it is not one of the crossing pairs). Hence, we can focus on $j' \geq j$, in which case the newly introduced pairs will be handled later. Second, we do not know in advance whether the pair ab is one of the crossing pairs added initially to P_j or whether it was added later on and should not be compressed. To remedy this, each element P_j stores information as well, whether it was a crossing pair or perhaps was added later on; those are used to decide whether ab should be compressed at all, as described later.

Now, since we cannot assume that the records in the list P_j are sorted or even that they should be compressed at all (as we might have added some pairs to P_j when considering $P_{j'}$ for $j' < j$), we sort them again using RadixSort, ignoring the coordinate for the pointers; furthermore, we add another coordinate, which is 0 for original crossing pairs and 1 for those introduced due to recompression. The sorting can be done in time $\mathcal{O}(p'_j + n + m)$: there are p'_j elements, and by Lemma 4.10 the size of Σ is at most $\mathcal{O}((n+m)^3)$.

Now, as the list of occurrences of pairs are sorted, we can cut it into several lists, each consisting of occurrences of a fixed pair. Going through one list, say for a pair ab, we first check whether the first occurrence is an original crossing pair; if not, then we do not compress occurrences of this pair at all (as this pair is not one of the original crossing pairs). If it is an original crossing pair, we replace the listed occurrences of ab (if they are still there) in $\mathcal{O}(p'_j)$ time: since we replace occurrences of one fixed pair, we always replace by the same (fresh) letter and do not need to use any dictionary operations to look up the appropriate letter. Clearly, this procedure properly implements the pair compression for a single pair ab and thus also for all pairs in P_j (note that pairs in P_j cannot overlap). This ends the inductive proof of (P 1).

The running time of the whole loop 3 is at most (for some constant c)

$$\sum_{j=1}^{i} c(p'_j + n + m) = c(n+m)i + c\sum_{j=1}^{i} p'_j$$
$$= \mathcal{O}((n+m)\log(n+m)) + c\sum_{j=1}^{i} p'_j.$$

Initially, each element of $\bigcup_{j=1}^{i} p_j$ corresponds to some occurrence of a (crossing) pair in G, and there are only $|G| + 4(n+m)$ such occurrences by Lemma 4.3. Hence, $\sum_{j=1}^{i} p_j = |G| + 4(n+m)$. The difference $p'_j - p_j$ is the number of pairs introduced to P_j by Pop. Still, there are at most $4(n+m)$ pairs added by one run of Pop(see Lemma 4.7), and

hence in total there are only $4i(n+m)$ pairs introduced in this way (as in total there are i groups). Hence,

$$\sum_{j=1}^{i} p'_j \le \underbrace{\sum_{j=1}^{i} p_j}_{\text{original occurrences of crossing pairs}} + \underbrace{\sum_{j=1}^{i} (p'_j - p_j)}_{\text{occurrences introduced by Pop}}$$
$$\le \big(|G| + 4(n+m)\big) + 4i(n+m)$$
$$= \mathcal{O}(|G| + (n+m)\log(n+m)).$$

Thus, the total running time is $\mathcal{O}((n+m)\log(n+m) + |G|)$, and at most $\mathcal{O}(\log(n+m))$ symbols are introduced into a rule. □

4.3. Blocks Compression

We now turn our attention to the block compression. Suppose first that G has no letters with a crossing block. Then a procedure similar to the one compressing noncrossing pairs can be performed: when reading G, we establish all maximal blocks of letters. We group these occurrences according to the letter—that is, for each letter a, we create a list of a's maximal blocks in G and sort this list according to the lengths of the blocks. We go through the list and replace each occurrence of a^ℓ by a fresh letter a_ℓ.

However, usually there are letters with crossing blocks. We deal with this similarly as in the case of crossing pairs: recall that a letter a has a crossing block if and only if aa is a crossing pair. Thus, suppose that aa is a crossing pair because of (CP 1)—that is, aX_i occurs in a rule and the first letter of $\mathrm{val}(X_i)$ is a. In such a case, we left-pop a letter from X_i. In general, this does not solve the problem, as it may happen that still a is the first letter of $\mathrm{val}(X_i)$ (and clearly aX_i still occurs in the rule). So we keep on left-popping until the first letter in $\mathrm{val}(X_i)$ is not a (this includes $\mathrm{val}(X_i) = \epsilon$). In other words, we remove the a-prefix of $\mathrm{val}(X_i)$. A symmetrical procedure is applied to X_j such that a is the last letter of $\mathrm{val}(X_j)$ and $X_j a$ occurs in a rule.

As in the case of pairs, it turns out that even a simplified approach works: for each nonterminal X_i, where the first letter of $\mathrm{val}(X_i)$ is a and the last letter of $\mathrm{val}(X_i)$ is b, it is enough to pop its a-prefix and b-suffix (see RemCrBlocks).

Observe that during the procedure, long blocks of a (up to 2^{n+m}) may be explicitly written in the rules. This is conveniently represented: a^ℓ is simply denoted as (a, ℓ), with ℓ encoded in binary. When ℓ fits in one code word, a^ℓ representation is still of constant size and everything works smoothly. For simplicity, for now we consider only this case; the general case (in which we assume only that $n+m$ fits in $\mathcal{O}(1)$ machine words) is treated in Section 6.

After RemCrBlocks, every letter a has no crossing blocks, and we may compress maximal blocks using the already described method.

LEMMA 4.9. *After* RemCrBlocks*, there are no crossing blocks. This algorithm and following block compression can be performed in time* $\mathcal{O}(|G| + (m+n)\log(m+n))$. *Together, they introduce at most four new letters to each rule.*

PROOF. We first show the first claim of the lemma—that is, that after RemCrBlocks there are no letters with crossing blocks. This follows from the following observations:

(1) When X_i is processed by RemCrBlocks, $\mathrm{val}(X_j)$ for $j \neq i$ is not changed.
(2) When RemCrBlocks considers X_i with a rule $X_i \to \alpha_i$ such that $\mathrm{val}(X_i) = a^\ell w b^r$, where w does not start with a and does not end with b and $\ell, r > 0$, then α_i has an explicit a^ℓ prefix and explicit b^r suffix. If $\mathrm{val}(X_i) = a^\ell$, then $\alpha_i = a^\ell$.

ALGORITHM 9: RemCrBlocks: removing crossing blocks

```
1: for i ← 1..m+n, except m and n+m do
2:     let X_i → α_i be the production for X_i and a its first letter
3:     calculate and remove the a-prefix a^{ℓ_i} of α_i
4:     replace each X_i in rule's bodies by a^{ℓ_i} X_i
5:     if val(X_i) = ϵ then
6:         remove X_i from the rules' bodies
7:     else
8:         let b be the last letter of α_i
9:         calculate and remove the b-suffix b^{r_i} of α_i
10:        replace each X_i in rule's bodies by X_i b^{r_i}
11:        if val(X_i) = ϵ then
12:            remove X_i from the rules' bodies
```

(3) When RemCrBlocks replaces X_i with $a^{\ell_i}X_ib^{r_i}$, then afterward the only letter to the right of X_i in the rules is b; similarly, the only letter to the left is a.
(4) After RemCrBlocks considered X_i, and X_i is to the right (left) of a, then a is not the first (last, respectively) letter of $\mathrm{val}(X_i)$.

As in Lemma 4.7, all properties follow by a simple induction on the number i of the considered nonterminal.

We infer from these observations that after RemCrBlocks, there are no crossing blocks in G. Suppose for the sake of contradiction that there are; let a be the letter that has a crossing block. We consider only the case of (CP 1)—that is, when there is X_j such that aX_j occurs in some rule and $\mathrm{val}(X_j)$ begins with a. Note that by observation (2) when RemCrBlocks considered X_j, it replaced it with $b^\ell X_j c^r$ for some letters b and c. By observation (3), the letter to the left of X_j in the rule for X_i is not changed by RemCrBlocks afterward (except that it can be popped when considering some other nonterminal); hence, $b = a$. Finally, by observation (4), the first letter of $\mathrm{val}(X_j)$ is not $b = a$, which is a contradiction.

RemCrBlocks is performed in $\mathcal{O}(|G|)$ time: we represent block a^ℓ as a pair (a, ℓ), then the length of the a-prefix (b-suffix) is calculated simply by reading the rule until a different letter is read (note that the lengths of the blocks fit in one machine word). Since there are at most four symbols introduced by RemCrBlocks to the rule, this takes $\mathcal{O}(|G| + n + m)$ time. The replacement of X_i by $a^{\ell_i}X_ib^{r_i}$ is done at most twice inside one rule and therefore takes in total $\mathcal{O}(n + m)$ time.

Note that right after RemCrBlocks, it might be that there are neighbouring blocks of the same letter in the rules of G. However, we can easily replace such neighbouring blocks by one block of appropriate length in one reading of G in time $\mathcal{O}(|G|)$.

Concerning the compression of the blocks of letters, we adapt the block compression from SimpleEqualityTesting(see Lemma 2.4). This is done similarly to the way in which we adapted the compression of noncrossing pairs from SimpleEqualityTesting(see Lemma 4.6). For the sake of completeness, we present a sketch. We read the description of G. Whenever we spot a maximal block a^ℓ for some letter a, we add a triple (a, ℓ, p) to the list, where p is the pointer to this occurrence of the block in G. Notice that as there are no crossing blocks, the nonterminals (and end or rules) count for termination of maximal blocks.

After reading the whole G, we sort these pairs lexicographically. However, separately, we sort the blocks that include the a-prefixes (or b-suffixes) popped from nonterminals and the other blocks. In total, we popped at most $4(n + m)$ prefixes and suffixes, so there are at most $4(n + m)$ blocks of the former form, and thus we can sort their tuples in $\mathcal{O}((n + m)\log(n + m))$ time using any usual sorting algorithm of guaranteed

$\mathcal{O}(n \log n)$ running time. The remaining blocks are sorted using RadixSort in linear time: note that each of those blocks cannot have length greater than $|G|$, as they consist only of explicit letters that were present in G before RemCrBlocks. Furthermore, as $\Sigma = \mathcal{O}((n+m) \log M \log(n+m)) = \mathcal{O}((n+m)^3)$, those tuples can be sorted in $\mathcal{O}(|G|+n+m)$ time. Finally, we can merge those two lists in $\mathcal{O}(|G| + n + m)$ time.

Now, for a fixed letter a, we use the pointers to localise a's blocks in the rules and replace each of its maximal block of length $\ell > 1$ by a fresh letter. Since the blocks of a are sorted according to their length, all blocks of the same length are consecutive on the list; therefore, replacing them by the same letter is done easily.

Since we already know that there are no letters with a crossing block, we can show, as in Lemma 4.6, that this procedure implements the block compression. The simple proof, which is essentially the same as the proof in Lemma 4.6, is omitted. □

4.4. Grammar and Alphabet Sizes

The subroutines of FCPM run in time that depends on $|G|$ and $|\Sigma|$; we bound these sizes.

LEMMA 4.10. *During* FCPM, $|G| = \mathcal{O}((n + m) \log(n + m))$ *and* $|\Sigma| = \mathcal{O}((n + m) \log(n + m) \log |M|)$.

The proof is straightforward. Using an argument similar to Lemma 2.3, we show that the size of the words that were in a rule at the beginning of the phase shorten by a constant factor (in this phase). On the other hand, only Pop and RemCrBlocks introduce new letters to the rules, and it can be estimated that in total they introduced $\mathcal{O}(\log(n + m))$ letters to a rule in each phase. Thus, bound $\mathcal{O}(\log(n + m))$ on each rules' length holds. Concerning $|\Sigma|$, new letters occur as a result of a compression. Since each compression decreases the size of $|G|$ by at least 1, there are no more than $|G|$ of them in a phase, which yields the bound.

PROOF. We begin by showing the bound on $|G|$. Consider a rule of G. On one hand, its size drops as we compress letters in it. On the other, some new letters are introduced to the rule by popping them from nonterminals. We estimate both influences.

Observe that the main claim of Lemma 2.3 (about two consecutive letters) applies to the bodies of the rules, so an argument similar to the one in the proof of Lemma 2.3 can be used to show that the length of the explicit strings that were in the rules at the beginning of the phase decreases by a constant factor in each phase. To be more precise, if the rule had k letters, it is shortened by at least $\frac{k-3}{3}$ letters (the "-3" represents that there are at most three substrings separated by nonterminals and the last letter in the rule, and the last letter in those substring may be uncompressed). Since there are at most $n+m$ rules and the sum of lengths of strings in them is at least $G-2(n+m)$, in one phase the length of strings in G is reduced by at least $\frac{|G|-5(n+m)}{3}$. Of course, the newly introduced letters may be unaffected by this compression. By routine calculations, if $\mathcal{O}((n+m) \log(n+m))$ letters are introduced to G, the $|G|$ is also $\mathcal{O}((n+m) \log(n+m))$ (with a larger constant, however). Hence, it is left to show that $\mathcal{O}((n + m) \log(n + m))$ letters are introduced to G in one phase. We do not count the letters that merely replaced some other letters (as a compression of maximal block or a pair compression), but only the letters that were popped into the rules.

In noncrossing pair compression, there are no new letters introduced. Concerning the crossing pairs compression, by Lemma 4.8 this introduces at most $\mathcal{O}(\log(n + m))$ letters to a rule, which is fine. When RemCrBlocks is applied, it introduces at most four new symbols into a rule (see Lemma 4.9). In total, this gives $\mathcal{O}(\log(n + m))$ letters per rule, so $\mathcal{O}((n + m) \log(n + m))$ letters in total.

Concerning the alphabet, the time used in one phase is $\mathcal{O}((n+m)\log(n+m)+|G|)$, which is $\mathcal{O}((n+m)\log(n+m))$. Thus, no more than this amount of letters is introduced in one phase. Lemma 2.3 guarantees that there are $\mathcal{O}(\log(\min(M,N)))$ phases, so a bound $\mathcal{O}((n+m)\log(\min(M,N))\log(n+m))$ on $|\Sigma|$ follows. □

Main result.

THEOREM 4.11. EqualityTesting *correctly verifies the equality of two strings defined by SLPs. It runs in time* $\mathcal{O}((n+m)\log(\min(M,N))\log(n+m))$*, where* $n+m$ *is the sum of sizes of the SLPs and* M*,* N *are lengths of those strings. It uses* $\mathcal{O}((n+m)\log(n+m))$ *space (counted in machine words).*

The bounds in the theorem are shown under the assumption that M, N fit in the machine word; those assumption are relaxed in Section 6 (see proof of Theorem 1.1 there).

PROOF. The cost of one phase of EqualityTesting is $\mathcal{O}(|G|+(n+m)+(m+n)\log(n+m))$, by Lemmata 4.6, 4.8, and 4.9, whereas Lemma 4.10 shows that $|G|=\mathcal{O}((n+m)\log(n+m))$ and Lemma 2.3 shows that there are $\mathcal{O}(\log(\min(M,N)))$ phases. Thus, the total running time is $\mathcal{O}((n+m)\log(\min(M,N))\log(n+m))$.

EqualityTesting uses memory proportional to the size of grammar representation, so $\mathcal{O}((n+m)\log(n+m))$.

The correctness follows as in Lemma 2.1. □

5. PATTERN MATCHING

In Section 3, we showed how to perform the pattern matching using recompression on explicit strings. In this section, we extend this method to the case in which p and t are given as SLPs. Note that most of the tools are already known: on one hand, in Section 4 it was shown how to perform the equality testing when both p and t are given as SLPs; on the other hand, in Section 3 it was shown that extending the equality testing algorithm to pattern matching algorithm boils down to fixing the beginning and ends. We already know the appropriate procedures FixEndsDifferent and FixEndsSame. Therefore, we need to focus only on the efficient implementations of FixEndsDifferent and FixEndsSame in the compressed setting, as other operations are used already in EqualityTesting.

The outline of the algorithm is similar to the variant for explicit pattern (see SimplePatternMatching). In the rest of the section, we comment on the implementation details and running time.

ALGORITHM 10: FCPM: outline

```
1: while |p| > 1 do
2:     P ← list of pairs
3:     L ← list of letters
4:     fix the beginning and end                    ▷ See Section 3
5:     for each a ∈ L do
6:         compress blocks of a
7:     P' ← crossing pairs from P
8:     P ← noncrossing pairs from P
9:     for each ab ∈ P do
10:        compress pair ab
11:    for ab ∈ P' do
12:        compress pair ab
13: Output the answer.
```

Fixing of beginning and end. The first operation in the FCPM is the fixing of the beginning and end, which adapts FixEndsSame and FixEndsDifferent to the compressed setting.

LEMMA 5.1. *The fixing of beginning and end for an SLP represented as p and t can be performed in $\mathcal{O}(|G| + (n+m)\log(n+m))$ time. It introduces $\mathcal{O}(n+m)$ new letters to G.*

PROOF. To see this, we look at the operations performed by FixEndsSame (the ones for FixEndsDifferent are even simpler) and comment on how to perform them efficiently in the compressed setting. First, in linear time, we can find out the first and last letter of $p$to see whether FixEndsDifferent or FixEndsSame should be applied; suppose the latter. Now, FixEndsSame performs a (modified) block compression; the only difference is that we compress only blocks of a and replace them not by a single letter but by up to three letters. To this end, we apply a modified RemCrBlocks, which removes only a-prefixes and a-suffixes; using the same argument as in Lemma 4.9, it can be shown that after the modified removal of prefixes and suffixes, there are no crossing a blocks. Afterward, we compress blocks of a. The running time bounds $\mathcal{O}(|G|+(n+m)\log(n+m))$ (see Lemma 4.9) are preserved. Furthermore, by the same lemma $\mathcal{O}(n+m)$ new letters are introduced to G.

The next operation in FixEndsSame is the compression of pairs of the form $\{a_L b \mid b \in \Sigma \setminus a_L\}$, then $\{ba_R \mid b \in \Sigma \setminus a_R\}$ (and then perhaps also $\{a_1 b \mid b \in \Sigma \setminus a_1\}$). In each case, the pairs are obtained by partitioning the alphabet into Σ_ℓ and Σ_r (where one of the parts is a singleton); in the following discussion, we focus on the pairs of the form $\{a_L b \mid b \in \Sigma \setminus a_L\}$, in which case $\Sigma_\ell = \{a_L\}$ and $\Sigma_r = \Sigma \setminus \{a_L\}$. Thus, by Lemma 4.7, one such group can be uncrossed in $\mathcal{O}(n+m)$ time; the uncrossing introduces $\mathcal{O}(n+m)$ letters to G. Afterward, we can compress pairs from this partition in $\mathcal{O}(|G|+n+m)$ time, similarly as in PairComp(see Lemma 4.8): we read G, listing all explicit pairs in which the first letter is a_L. Then, we sort them using RadixSort and replace their occurrences. This clearly takes $\mathcal{O}(|G|+n+m)$ time. □

The rest of the operations on G (pair compression, block compression) are implemented as in Section 4 and have the same running time.

We now move to the analysis of FCPM. We show that FCPM preserves the crucial important property of EqualityTesting: that $|p|$ decreases by a constant factor in each phase and that $|G| = \mathcal{O}((n+m)\log(n+m))$ as well as $|\Sigma| = \mathcal{O}((n+m)\log(n+m)\log M)$.

LEMMA 5.2. *In each phase, the* FCPM *shortens p by a constant factor. The size of G is $\mathcal{O}((n+m)\log(n+m))$, whereas the size of Σ is $\mathcal{O}((n+m)\log(n+m)\log M)$.*

PROOF. Observe that FCPM performs the same operations on p as SimplePatternMatching, but it only does it on the compressed representation. Thus, it follows from Lemma 3.5 that both p and t are shortened by a constant factor in one phase of FCPM.

Concerning the size of the grammar, a similar argument as in Lemma 4.10 applies. Note that as in EqualityTesting, the FCPM introduces $\mathcal{O}((n+m)\log(n+m))$ letters per phase into G: it uses the same operations as SimpleEqualityTesting as well as the fixing of beginning and end, which introduce $\mathcal{O}(n+m)$ letters per phase by Lemma 5.1. On the other hand, the analysis performed in Lemma 3.5 (that SimplePatternMatching shortens p) applies to each substring of p and t, so each explicit string in the rules of G is shortened during the phase by a constant factor (i.e., the same as in Lemma 4.10). Hence, the size of G kept by FCPM can be also bounded by $\mathcal{O}((n+m)\log(n+m))$. Consequently, also $|\Sigma| = \mathcal{O}((n+m)\log(n+m)\log M)$. □

This is enough to show the running time and correctness of FCPM.

THEOREM 5.3. FCPM *runs in* $\mathcal{O}((n+m)\log(n+m)\log M)$ *time and returns a representation of all pattern occurrences in text.*

Observe that as the proof of Lemma 4.9 assumes that N and M fit in $\mathcal{O}(1)$ machine words, the same assumption applies here. This assumption is lifted in Section 6, so the proof there shows the same result without this assumption.

PROOF. Lemma 5.2 implies that FCPM runs in $\mathcal{O}((n+m)\log(n+m)\log M)$ time: each subprocedure runs in time $\mathcal{O}(|G| + (n+m)\log(n+m)) = \mathcal{O}((n+m)\log(n+m))$, so this is the running time of one phase as well. Since the pattern is shortened by a constant factor in each phase (see Lemma 5.2), there are $\mathcal{O}(\log M)$ many phases. The correctness (returning representation of all pattern occurrence) follows from the correctness of SimplePatternMatching (as the performed operations are the same, just the representation of p and t is different). □

Occurrences and their positions. When p was reduced into a single letter, say c, then there are two possibilities: if the last compression applied on the pattern was a pair compression, then each occurrence of c in t corresponds to one occurrence of original pattern in the original text. Note that t is given by a grammar G, which is of size $\mathcal{O}((n+m)\log(n+m))$ (in the next section, we show a bound $\mathcal{O}(n+m)$). The other possibility is that the last compression was a-block compression. If the pattern is a letter a_ℓ, then each occurrence of a letter a_m in t (for $m \geq \ell$) corresponds to $m-\ell+1$ occurrences of the original pattern in the original text.

Recall the notion of weight. For a letter, it is the length of the corresponding substring in the input string; every letter in that instance has a positive integer weight. When N fits in a constant amount of code words, the weight of each letter can be calculated in constant time, so we store the weights of the letters on the fly in a table. To calculate the position of the first pattern occurrence, it is enough to calculate the weight of the string preceding it. To this end, we keep up-to-date table of weights of $\mathrm{val}(X_i)$ for each X_i. To return the first position of a pattern occurrence, we determine the derivation path for this occurrence and sum up the weights of nonterminals and letters that are to the left of this derivation path; this is easy to perform in time linear in $\mathcal{O}(|G|)$.

Note that there is a small technical issue: in one special case, we remove the first letter from $t$when it is a_R. However, $\mathrm{w}(a_R) = 0$, so it does not influence anything.

This approach can be extended to return a position of an arbitrary occurrence of the pattern. To this end, for each nonterminal, we also store the number of occurrences of the pattern that it derives. This is easy to built in a bottom-up fashion as soon as the pattern is equal to a^k for some $k \geq 1$. Then, in $\mathcal{O}(|G|)$ time, we can also find this particular occurrence of the pattern in text.

6. IMPROVING THE RUNNING TIME

To reduce the running time to $\mathcal{O}((n+m)\log M)$, we need to make sure that the grammar size is $\mathcal{O}(n+m)$ and improve the running time of block compression (see Lemma 4.9) so that it is $\mathcal{O}(|G|)$, without the extra $(n+m)\log(n+m)$ summand. For the former, the argument in Lemma 4.10 (and its adaptation in Lemma 5.2) guarantees this size as long as there are only $\mathcal{O}(n+m)$ letters introduced to G in a phase. The crossing blocks compression, as well as fixing the beginning and end, already possesses this property (see Lemma 4.9 and Lemma 5.1), so it is enough to alter the crossing pairs compression.

We show that it is enough to consider $\mathcal{O}(1)$ partitions Σ_ℓ, Σ_r and pairs that fall into them. Roughly, we choose a partition such that a constant fraction of crossing pairs occurrences in p fall into this partition. In particular, for each crossing pair ab, we calculate the number of its occurrences in p. This requires that we can manipulate

numbers of size M in constant time—that is, this construction requires that M fits in $\mathcal{O}(1)$ machine words.

For the block compression, we improve the sorting time: we group block lengths into groups of similar lengths and sort one such group in linear time using RadixSort. The groups are also established using RadixSort performed on representatives of groups. The latter sorting treats numbers as a bit string and therefore may have high running time, but we show that overall it cannot exceed $\mathcal{O}((n+m)\log M)$ during the whole FCPM.

6.1. Faster Compression of Crossing Pairs

For a given partition Σ_ℓ, Σ_r, we say that it *covers* the occurrences of $ab \in \Sigma_\ell\Sigma_r$ in p. The main idea of improving the running time of the crossing pairs is quite simple: instead of considering $\mathcal{O}(\log(n+m))$ partitions such that each crossing pair from P' is covered by at least one of them, we consider only one partition Σ_ℓ, Σ_r such that at least one fourth of occurrences of crossing pairs in p are covered by it. Then, estimations about shortening of the pattern in one phase hold as before, although with a larger constant.

Existence of such a partition can be shown by a simple probabilistic argument: if we assign each letter to Σ_ℓ with probability $1/2$, then a fixed occurrences of ab in p is covered with probability $1/4$. The standard expected value derandomisation technique gives a deterministic algorithm finding such a partition, and it can easily be implemented in $\mathcal{O}(|G|)$ time (see Lemma 6.2).

It is not guaranteed that this partition shortens $|G|$ as well; however, we can use exactly the same approach to shorten G: we find another partition $\Sigma_\ell\Sigma_r$ such that at least one fourth of crossing pairs expliciting occurrences in G are from this partition.

Our to-be-presented algorithm constructing a partition requires a list of all crossing pairs together with the number of their occurrences in p. This can be supplied using a simple linear-time algorithm: for each nonterminal X_i, we calculate the amount k_i of substrings $\mathrm{val}(X_i)$ that it generates in p. We associate an occurrence of ab with the least nonterminal that generates it. Then the number of all occurrences of ab can be calculated by summing the appropriate k_i's.

LEMMA 6.1. *Assuming that M fits in $\mathcal{O}(1)$ code words, in $\mathcal{O}(|G|+n+m)$ we can return a sorted list of crossing pairs together with number of their occurrences in p as well as links to these occurrences.*

PROOF. Clearly, $k_m = 1$ (as X_m simply generates the whole p), and other numbers satisfy a simple recursive formula:

$$k_j = \sum_{i>j} k_i \cdot \#\{\text{number of times } X_j \text{ occurs in the rule for } X_i\}. \tag{3}$$

Then, (3) can be used in a simple linear-time procedure for calculation of k's: for $i = m..1$, we add k_i to k_j when X_j occurs in the rule for X_i (we add twice if there are two such occurrences). Clearly, this can be implemented in $\mathcal{O}(m)$ time.

Concerning the number of occurrences of crossing pairs, observe that each occurrence of ab in p can be assigned to a unique rule $X_i \to \alpha_i$: this is the rule that generates this particular occurrence of ab; moreover, this occurrence of ab comes from an explicit occurrence of ab in α_i or a crossing occurrence of ab in this rule. To see this, imagine that we try to retrace the generation of this particular ab. Given X_i generating this occurrence of ab (we start with X_m, as we know that it generates this ab), we check if it is generated by nonterminal X_j in the rule. If so, we replace X_i with X_j and iterate the process. If not, then this ab is comes from either an explicit or crossing pair in this X_i.

Given a rule for X_i, listing all pairs that occur explicitly or have a crossing occurrence in the rule for X_i is easy; we only need to know the first and last letter of each nonterminal. Then, for each such pair ab, we create a tuple (a, b, k_i, p) (where k_i is the number of substrings that X_i generates and p the pointer to this occurrence in the rule). We sort the tuples using RadixSort, ignoring the two last coordinates, which takes $\mathcal{O}(|G| + n + m)$ time: there are $|G| + n + m$ such tuples by Lemma 4.3, and letters have value at most $\mathcal{O}((n + m) \log(n + m) \log M) = \mathcal{O}((n + m)^3)$ by Lemma 4.10.

Now, for a given pair ab, the tuples with the number of its occurrences are listed consecutively on the list, so for each pair we can add those numbers and obtain the desired (sorted) list of pairs with numbers of their occurrences in G; this also takes linear time, since the list is of this length.

This list includes both crossing and noncrossing pairs. We use the same procedure as in Lemma 4.6 to establish the crossing and noncrossing pairs. Note that it generated a sorted list of crossing (and noncrossing) pairs; this takes $\mathcal{O}(|G| + n + m)$ time. Without loss of generality, the order on those lists is the same as on our list, so we can filter from it only crossing pairs in time linear in the size of the lists—that is, $\mathcal{O}(|G| + n + m)$. □

In the following discussion, for a crossing pair ab, we shall denote by k_{ab} the number of its occurrences in p, calculated in Lemma 6.1.

Now we are ready to find the partition covering at least one fourth of the occurrences of crossing pairs. This is done by a derandomisation of a probabilistic argument mentioned at the beginning of this section.

LEMMA 6.2. *For p in $\mathcal{O}(|G| + n + m)$ time, we can find a partition of Σ into Σ_ℓ, Σ_r such that the number of occurrences of crossing pairs in p covered by this partition is at least one fourth of all such occurrences in p. In the same running time, we can provide for each covered crossing ab a lists of pointers to its explicit occurrences in G.*

PROOF. The partition shall be represented by two bit vectors of size $|\Sigma|$: the first keeps the letters in Σ_ℓ and the second in Σ_r. When such bit vectors are known, obtaining the list of pointers to occurrences of covered ab is easy: using Lemma 6.1, we create a sorted list of occurrences of pairs in G. We read the list for each ab on this list that we check whether $ab \in \Sigma_\ell \Sigma_r$ (which can be done in $\mathcal{O}(1)$ using the bit vectors). If so, nothing happens; if not, we remove the pair from the list. In the following discussion, we shall consider only the problem of finding the promised partition.

Observe that the problem of finding such a partition reduces to the problem of finding a Max-Cut in a directed weighted graph: for the reduction, we create a node for each letter and put an edge from a to b with weight k if there are k occurrences of pair ab. It is easy to see that a (directed) cut of weight w corresponds to a partition of letters covering exactly k occurrences of pairs (and vice versa). The rest of the the proof recalls the standard construction [Mitzenmacher and Upfal 2005, Section 6.3] in terminology used in the article.

First, observe that the probabilistic argument given before the lemma can be altered: if we were to count the number of pairs that are covered *either* by $\Sigma_\ell \Sigma_r$ or $\Sigma_r \Sigma_\ell$, then the expected number of crossing pairs occurrences covered by $\Sigma_\ell \Sigma_r \cup \Sigma_r \Sigma_\ell$ is one half.

The deterministic construction of such a partition follows by a simple derandomisation, using an expected value approach. It is easier to first find a partition such that at least half of crossing pairs occurrences in p are covered by $\Sigma_\ell \Sigma_r \cup \Sigma_r \Sigma_\ell$ and then choose $\Sigma_\ell \Sigma_r$ or $\Sigma_r \Sigma_\ell$, depending on which of them covers more occurrences.

According to Lemma 6.1, we assume that we are given a sorted list P', on which we have all crossing pairs together with the number k_{ab} of their occurrences in p.

Suppose that we have already assigned some letters to Σ_ℓ and Σ_r and that we are to decide where the next letter a is assigned. If it is assigned to Σ_ℓ, then all occurrences

of pairs from $a\Sigma_\ell \cup \Sigma_\ell a$ are not going to be covered, whereas occurrences of pairs from $a\Sigma_r \cup \Sigma_r a$ are; similarly, observation holds for a being assigned to Σ_r. The algorithm makes a greedy choice, maximising the number of covered pairs in each step. As there are only two options, the choice brings in at least half of the occurrences considered. Finally, as each occurrence of a pair ab from p is considered exactly once (i.e., when the second of a, b is considered in the main loop), this procedure guarantees that at least half of the occurrences of crossing pairs in p are covered.

To make the selection effective, the algorithm GreedyPairs keeps counters $count_\ell[a]$ and $count_r[a]$, denoting, respectively, the number of occurrences of pairs from $a\Sigma_\ell \cup \Sigma_\ell a$ and $a\Sigma_r \cup \Sigma_r a$ in p. Those counters are updated as soon as a letter is assigned to Σ_ℓ or Σ_r—that is, when one of Σ_ℓ, Σ_r is increased. Note that as by Lemma 2.2 we can assume that letters in p are from an interval of consecutive $|G|$ letters, the counters can be organised as a table with constant access time to $count_\ell[a]$ and $count_r[a]$.

ALGORITHM 11: GreedyPairs

```
1:  L ← set of letters used in P'
2:  Σ_ℓ ← Σ_r ← ∅                                          ▷ Organised as bit vectors
3:  for a ∈ L do
4:      count_ℓ[a] ← count_r[a] ← 0                        ▷ Initialisation
5:  for a ∈ L do
6:      if count_r[a] ≥ count_ℓ[a] then                    ▷ Choose the one that guarantees larger cover
7:          choice ← ℓ
8:      else
9:          choice ← r
10:     Σ_choice ← Σ_choice ∪ {a}
11:     for each b ∈ L do
12:         count_choice[b] ← count_choice[b] + k_ab + k_ba   ▷ Update counters
13: if # occurrences of pairs from Σ_r Σ_ℓ in p> # occurrences of pairs from Σ_ℓ Σ_r in p then
14:     switch Σ_r and Σ_ℓ
15: return (Σ_ℓ, Σ_r)
```

By the preceding argument, when Σ is partitioned into Σ_ℓ and Σ_r, at least half of the occurrences of pairs from p are covered by $\Sigma_\ell\Sigma_r \cup \Sigma_r\Sigma_\ell$. Then one of the choices $\Sigma_\ell\Sigma_r$ or $\Sigma_r\Sigma_\ell$ covers at least one fourth of the occurrences.

It is left to give an efficient variant of GreedyPairs; the nonobvious operations are the choice of the actual partition in line 14 and the updating of $count_\ell[b]$ or $count_r[b]$ in line 12. All other operations clearly take $\mathcal{O}(|G| + n + m)$ time. The latter is simple: since Σ_ℓ and Σ_r are organised as a bit vector, we can read P'; for each pair ab in it, check if it is covered by $\Sigma_\ell\Sigma_r$ or $\Sigma_r\Sigma_\ell$ (or by none of them) by verifying whether $a \in \Sigma_\ell$ or $a \in \Sigma_r$ and similarly $b \in \Sigma_\ell$ or $b \in \Sigma_r$. In this way, we can calculate the total number of pairs occurrences covered by each of those two partitions and choose the larger one.

To implement the *count*, for each letter a in p we have a table *right* of *right lists*: $right[a] = \{(b, k_{ab}) \mid ab \text{ occurs in } P'\}$, represented as a list. There is a similar *left list*: $left[a] = \{(b, k_{ba}) \mid ba \text{ occurs in } P'\}$. Since at the input we get a sorted list of all pairs ab together with k_{ab}, creation of $right[a]$ can be easily done in in linear time (and similarly $left[a]$).

Given *right* and *left*, performing the update in line 12 is easy (suppose that we are to update $count_\ell$): we go through $right[a]$ ($left[a]$). When we spot (b, k_{ab}) (or (b, k_{ba}), respectively), we increase the $count_\ell[b]$ by k_{ab} (k_{ba}, respectively). As *right*, *left* and *count* are organised as tables; this takes $\mathcal{O}(1)$ time per read element of $right[a]$ ($left[a]$). We can then discard $right[a]$ ($left[a]$), as they are not going to be used again. In this way, each

of the list *right*[*a*] (*left*[*a*]) is read $\mathcal{O}(1)$ times during GreedyPairs, so this time is at most as much as the time of their creation—that is, $\mathcal{O}(|G| + n + m)$. □

A similar construction works when we want to calculate the partition that covers one fourth of occurrences of crossing pairs in G: when calculating the number of occurrences of pair ab, it is enough to drop the coefficient k_i for occurring in the rule X_i and take one for every rule. The rest of the construction and proofs are the same as in Lemma 6.2.

LEMMA 6.3. *In $\mathcal{O}(|G| + n + m)$ time, we can find a partition of Σ into Σ_ℓ, Σ_r such that the number of occurrences of crossing pairs in G covered by this partition is at least one fourth of all such occurrences in G. In the same running time, we can provide for each covered crossing ab a lists of pointers to its explicit occurrences in G.*

Thus, the modification of FCPM is as follows. After establishing the list of all crossing pairs, we find two partitions Σ_ℓ, Σ_r, Σ'_ℓ, Σ'_r, one of which covers one fourth of occurrences of crossing pairs in the pattern the other in G. Then, instead of compressing all crossing pairs, we compress only those covered by the first and then the second partition. Each of those compressions requires only one call to Pop, so there are only $\mathcal{O}(1)$ letters introduced to a rule during the crossing pairs compression.

LEMMA 6.4. FCPM *using the modified crossing pair compression subprocedure introduces $\mathcal{O}(1)$ letters to a rule in one phase.*

It is left to estimate that indeed this modified compression schema shortens $|p|$ and $|G|$ by a constant factor in a phase. The former will guarantee the $\mathcal{O}(\log M)$ number of phases and the latter an upper bound $\mathcal{O}(n + m)$ on the size of G.

LEMMA 6.5. FCPM *using the modified crossing pair subprocedure keeps the size of the grammar $\mathcal{O}(n + m)$ and has $\mathcal{O}(\log M)$ phases.*

PROOF. Let us consider the simpler case of EqualityTesting. First, consider the length of p; we show that it is reduced by a constant factor in a phase. Consider two consecutive letters ab in p. Observe that if EqualityTesting tried to compress ab (when $a = b$, this means that a blocks were compressed), then at least one of those letters is compressed in the phase: we tried to compress this ab, and the only reason we could fail is because one of those letters was already compressed by some earlier compression. We want to show that for at least $1/4$ of occurrences of pairs, we tried to compress this occurrence. Thus, at least $1/8$ of all letters were compressed, so the length of p dropped by at least $1/16$ in a phase.

If $a = b$, then we compressed them during the blocks compression. If $a \neq b$ and ab was noncrossing, then we tried to compress them. Finally, when $a \neq b$ and ab was a crossing pair, then we chose a partition $\Sigma_\ell \Sigma_r$ such that at least one fourth of all occurrences of crossing pairs were covered by this partition. So for one in four of such occurrences, we tried to compress it. In total, for at least one fourth of occurrences of ab's, we tried to compress them, as claimed.

A similar analysis yields that we reduced the length of $|G|$ (excluding the new introduced letters) by $15/16$. Since we introduce only $\mathcal{O}(n + m)$ new letters to G per phase, the size of G remains $\mathcal{O}(n + m)$, which is shown in Lemma 4.10.

Now, in the general case of FCPM, we combine this analysis with the one in Lemma 3.5. We define the fragments of more than one letter as in Lemma 3.5—that is, the letters that were replaced during the compression are grouped so that one group (fragment) is replaced with a shorter string. The letters that were not altered are not assigned to any fragments.

Similarly as in the earlier argument for EqualityTesting, we want to show that for at least one fourth of all pairs of consecutive letters, one of those letters was assigned to a fragment. Since fragments are replaced with strings of at most three fourths their length, as earlier, this shows that p is shortened by a constant factor. To show that at least one of ab is in a fragment, it is again enough to show that we tried to compress ab (either as a pair of different letters or as a part of a block of letters): if we succeed, then a, b are in the same fragment; if we fail, then this means that at least one of them is in some other fragment.

Thus, consider any two consecutive letters a and b. If any of them was compressed during the fixing of beginning or end, then we are done, as it was assigned to a fragment. Otherwise, if $a = b$, then they are compressed during the blocks compression, so both of them are assigned to the same fragment. If $a \neq b$ and ab is a noncrossing pair, then we tried to compress it during the noncrossing pairs compression. Finally, if $a \neq b$ and ab is a crossing pair, then due to our construction of Σ_ℓ and Σ_r from Lemma 6.3, at least one fourth of occurrences of crossing pairs is chosen for the compression.

The rest of the argument follows as in the case of the one for EqualityTesting, with a slightly larger constant. Hence, in each round, p is shortened by a constant factor and so there are $\mathcal{O}(\log M)$ phases.

Observe that a similar argument holds for G. Consider two consecutive letters in the rule of G (note that as G has up to $m+n$ rules, at most $3(n+m)$ do not have a following letter). Then, there is a second round of compression of crossing pairs that tries to compress at least one fourth of crossing pairs occurrences in G. Hence, the explicit strings in G can be grouped into fragments as well, as described earlier. On the other hand, by Lemmata 4.9, 5.1, and 6.4, there are $\mathcal{O}(n+m)$ letters introduced to G in one phase (and those are not necessarily compressed). Therefore, the size of the new grammar (at the end of the phase) G' can be given using the recursive equation

$$|G'| \leq 3(n+m) + \alpha|G| + \beta(n+m)$$

for some $\alpha < 1$ and β (the $3(n+m)$ is for the letters that do not have a letter to its right and so are unaffected by our analysis). Since in the first phase $|G| = n+m$ by simple calculations, it follows that $|G'| \leq \frac{2\beta+6}{1-\alpha}(n+m)$. □

6.2. Block Compression

As already noted, we should improve the $\mathcal{O}(|G| + (n+m)\log(n+m))$ running time (see Lemma 4.9) used for sorting of blocks' lengths to $\mathcal{O}(|G|)$; recall that the cost $\mathcal{O}((n+m)\log(n+m))$ came from sorting. Moreover, in Section 4, we assumed that M and N fit in $\mathcal{O}(1)$ machine words for the purpose of blocks compression and promised to relax this assumption. We now both improve the running time and relax this assumption, requiring only that $n+m$ fits in $\mathcal{O}(1)$ machine words. We do this by introducing a special representation of the lengths of a blocks, which is represented as a sum of two numbers: a big one (represented as a bit vector) and a small one (of size $\mathcal{O}(|G|)$), which is stored in $\mathcal{O}(1)$ machine words. The crucial observation will be that we can restrict ourselves to $\mathcal{O}(n+m)$ such large numbers. The big numbers are so much larger that in order to sort, we can sort the numbers with the same large component separately—that is, it is enough to sort the small numbers. Since they are small (at most $|G|$), this can be done by RadixSort in $\mathcal{O}(|G|)$ time. On the other hand, the large numbers shall also be sorted using RadixSort (in which case those numbers are treated as bit strings), and more involved analysis shows that the total time spent on such sorting is $\mathcal{O}((n+m)\log M)$.

However, representation as bit strings introduces new problems. Observe that when t is a block of letters, calculating the bit vector representation of its length takes $\Omega(\log N)$ time. It is difficult to account for such large costs. We avoid the problem by

not calculating exactly the lengths of blocks that are longer than M (i.e., the original pattern).

Too long blocks. Consider the blocks of letter a that do not occur in p. Then, there is no difference whether we replace two occurrences of a^ℓ with the same letter or with different letters, as they cannot be part of a pattern occurrence. Thus, for a that does not occur in p, we perform a " sloppy" blocks compression: we treat each maximal block as if it had a unique length. To be precise, we perform RemCrBlocksbut represent a^ℓ blocks as $(a, ?)$ for $\ell > 1$. Then, when replacing blocks of a (we exclude the blocks of length 1), we replace each of them with a fresh letter. In this way, the whole blocks compression does not include any cost of sorting the lengths of blocks of a. Still, the occurrences of the pattern are preserved.

A similar situation occurs for a that occurs in p, but t has a blocks of length greater than M. We treat them similarly: as soon as we realise that a^ℓ has $\ell > M$, we represent such blocks as $(a, > M)$ and do not calculate the exact length and do not insist that two such blocks of the same length are replaced with the same symbol. In this way, we avoid the cost associated with sorting this length. Of course, when a is the first or last letter of the pattern, we need to replace them with $a_R a_? a_L$, to allow the pattern beginning/ending at this block.

Faster sorting. Length representation. The intuition is as follows: although the a blocks can have exponential length, most of them do not differ much, as they are obtained by concatenating letters a that occur explicitly in the grammar. Such concatenations can, in total, increase the lengths of a blocks by $|G|$. Still, there are blocks of exponential length: these "long" blocks are created only when two blocks coming from two different nonterminals are concatenated. However, there are only $n + m$ concatenations of nonterminals (one per production), so the total number of long blocks "should be" at most $n + m$. Of course, the two mentioned ways of obtaining blocks can mix, and our representation takes this into the account: we represent each block as a concatenation of two blocks—a long one and short one:

—The long one corresponds to a block obtained as a concatenation of two nonterminals; such a long block is common for many blocks of letters.
—The short one corresponds to concatenations of letters occurring explicitly in G; this length is associated with the given block alone.

More formally, we store a list of *common lengths*—that is, the lengths of common long blocks of letters. Such a length is stored as a bit vector. Each block length ℓ is represented as a sum $c + o$, where c is one of the common lengths and o (*offset*) is a number associated with ℓ. Internally, ℓ is represented as a number o and a pointer to c. One of the common lengths is 0. In the intermediate construction, some blocks can be represented without the common length, which is different from having a common length 0. The construction will guarantee that each offset is at most $|G|$.

We treat common lengths that are larger than M with a special disregard; in particular, we do not calculate them exactly but just store the information that they are larger than M. To this end, for each common length, we also store the length of the associated bit vector. If it is longer than $\log(M + 1)$, then the common length is larger than M. Note that the length of the bit vector is $\log(M + 1) = \mathcal{O}(m)$, so it can be manipulated in constant time.

Creating a common length. During RemCrBlocks, each prefix and suffix popped from nonterminals inside the rule is represented as a common length and an offset in the following way (we add to them the common length 0 if they are represented without a common length). Then we calculate the lengths of blocks inside the rule for X_i: the

explicit letter inside the rule simply increases the offset; the blocks that are formed solely from explicit letters do not have a common length. If any length of the block is represented as a sum of two common lengths (and perhaps some offset), we create a new common length for it.

Before proceeding, let us note how large the offsets may be and how many of them there are.

LEMMA 6.6. *There are at most $n+m+1$ common lengths. There are at most $|G|+n+m$ offsets in total, and the largest offset is at most $|G|$.*

PROOF. One common length is 0. Each other common length is created inside a rule for a nonterminal. Furthermore, at most one common length can be created inside a single rule: for a new common length c to be created, there have to be two nonterminals X_j and X_k inside a rule before popping, and the block a^c is then created between those two nonterminals.

Creation of an offset corresponds to an explicit letter in G, so there are at most $|G|$ offsets created.

An offset is created or increased when an explicit letter a (not in a compressed form) is concatenated to the block of a's. One letter is used once for this purpose, and there is no other way to increase an offset, so the maximal of them is at most $|G|$. □

Comparing lengths. Since we intend to sort the lengths, we need to compare the lengths of two numbers represented as common lengths with offsets, say $o+c$ and $o'+c'$. Considering that the common lengths are so large, we expect that we can compare them lexicographically, such as

$$o+c \geq o'+c' \iff \begin{cases} c > c', & \text{or} \\ c = c' \wedge o \geq o'. \end{cases} \tag{4}$$

Furthermore, (4) allows a simple way of sorting the lengths of maximal blocks:

—We first sort the common lengths (by their values).
—Then, for each common length, we (separately) sort the offsets assigned to this common length.

Although (4) may not initially be true, we can apply a couple of patches that make it true. Before that, however, the common lengths need to be sorted. We sort them using RadixSort and treat each common length as a series of bits. Although this looks more expensive, it allows a nice amortised analysis, as demonstrated later (see Lemma 6.10). Recall that we do not sort lengths of blocks a^ℓ for $\ell > M$: we represent them as $(a, > M)$ as soon as we find out that they are longer than M.

LEMMA 6.7. *Let $c_1, c_2, \ldots, c_k$ be the common lengths. The time needed to find those whose bit vectors are longer than $\log(M+1)$ and sort the others is $\mathcal{O}(k + \sum_{i=1}^{k} \log(\min(c_i, M+1)))$.*

The large common lengths are found by simply counting the lengths of their bit vectors until we reach a position larger than $\log(M+1)$. The sorting is done by a standard implementation of RadixSort that sorts the numbers of different lengths.

In the following discussion, we consider only the common lengths whose bit vectors have length at most $\log(M+1)$.

The problem with (4) is that even though c_i and c_j are so large, it can still happen that $|c_i - c_j|$ is small. We fix this naively: first, we remove some common lengths so that $c_{i+1} - c_i > |G|$. A simple greedy algorithm does the job in linear time. Since common lengths are removed, we need to change the representations of lengths: when o was assigned to remove c, consider the c_i and c_{i+1} that remained in the sequence and

$c_i < c < c_{i+1}$. We reassign $\ell = c + o$ to either c_i or c_{i+1}: if $o + c \geq c_{i+1}$, then we reassign it to c_{i+1} and otherwise to c_i. It can be shown that in this way, all offsets are at most $2|G|$ and that (4) holds afterward.

LEMMA 6.8. *Given a sorted list of common lengths $c_1 \leq c_2 \leq \cdots \leq c_k$ (whose bit vectors have length at most $\log(M+1)$), we can in $\mathcal{O}(\sum_{i=1}^{k} \log(c_i) + k)$ time choose its sublist and reassign offsets (preserving the represented lengths) such that all offsets are at most $2|G|$ and (4) holds.*

PROOF. We include $c_0 = 0$ for simplicity of presentation.

First, we calculate the differences Δ between consecutive c's and store those that are at most $|G|$. This can be done in $\mathcal{O}(\sum_{i=1}^{k} \log(c_i) + k)$ time: for two consecutive c_{i+1} and c_i, we calculate their difference using the bivectors in time $\mathcal{O}(\log c_{i+1} + 1)$. When the difference is known, we begin to evaluate it (as a number) until it is calculated or shown to be greater than $|G|$. In the former case, we store Δ_i. The running time thus far is $\mathcal{O}(\sum_{i=1}^{k} \log(c_i) + k)$.

Given a sorted list $c_0 \leq c_1 \leq c_2 \leq \cdots \leq c_k$ of common lengths and their differences $\Delta_1, \Delta_i, \ldots$, we choose its subsequence $0 = c'_0 \leq c'_1 \leq c'_2 \leq \cdots \leq c'_{k'}$ such that the distance between any two consecutive common lengths in it is at least $|G|$—that is, $c'_{i+1} - c'_i \leq |G|$. This is done naively: we choose $c_0 = 0$ and then go through the list. Having last chosen c, we look for the smallest common length c' such that $c' - c > |G|$ and choose this c' as the next element. The condition $c' - c > |G|$ is verified by summing up appropriate Δs. Since there are k common lengths in the beginning and adding defined Δs can be done in $\mathcal{O}(1)$ time, this can be done in $\mathcal{O}(k)$ time. As the last step, we also update the Δs so that they still show the difference between consecutive common lengths (again, we do not store those that are larger than $|G|$).

For any removed c such that $c_i < c < c_{i+1}$, we reassign offsets assigned to c as described earlier: for o assigned to c, if $c + o \geq c_{i+1}$ (which can be equivalently stated as $c_{i+1} - c \leq o$), then we reassign o to c_{i+1} and otherwise to c_i. As $o \leq |G|$, this condition can be verified using Δs alone.

When o is reassigned to c_i, then $c - c_i \leq |G|$ (as c was removed), and if reassigned to c_{i+1}, then $c_{i+1} - c \leq o \leq |G|$. Thus, in any case, the new offset o' can be calculated in constant time using Δs and o. As there are $\mathcal{O}(|G|)$ offsets (see Lemma 6.6), all reassigning takes $\mathcal{O}(|G|)$ time in total.

Let o' be the offset after the reassignment. Then,

—$o' \leq 2|G|$, since $o \leq |G|$ and the only way to increase it is to reassign it to c_i. Since c is removed, it holds that $c - c_i \leq |G|$. Hence, $o' = o + (c - c_i) \leq |G| + |G|$.
—When o_i is assigned to c_i, then $o_i + c_i < c_{i+1}$: indeed, if o_i was reassigned from $c > c_i$, then by definition $c_i + o_i = c + o < c_{i+1}$; if o_i was originally assigned to c_i or it was reassigned from c_{i-1}, then $o_i < |G|$ and so $c_i + o_i \leq c_i + |G| < c_{i+1}$.

Note that the second item in the preceding list implies (4). Hence, the claim of the lemma holds. □

Now, since (4) holds, to sort all lengths it is enough to sort the offsets within groups. We do it simultaneously for all groups: offset o_j assigned to common length c_i is represented as (i, o_j), and we sort these pairs lexicographically using RadixSort. Since the offsets are at most $2|G|$ and there are at most $|G|$ of them and at most $\mathcal{O}(n+m)$ common lengths (see Lemma 6.6 for all those three estimations), RadixSort sorts them in $\mathcal{O}(|G|)$ time.

LEMMA 6.9. *When common lengths whose bit vectors are of length at most* $\log(M+1)$ *are sorted and satisfy (4), sorting all lengths of blocks that are at most* M *takes* $\mathcal{O}(|G|)$ *time.*

Finally, we bound the time needed to identify and sort the common lengths whose bit vectors are of length at most $\log(M+1)$. Due to Lemmas 6.7 and 6.8, this cost is $\mathcal{O}(k+\sum_{i=1}^{k}\log(\min(c_i, M+1)))$, where $c_1, \ldots, c_k$ are all common lengths. Hence, we can assign $\mathcal{O}(1+\log(\min(c, M+1)))$ cost to a common length c and redirect this cost toward the rule in which c was created. The $\mathcal{O}(1)$ cost for maintaining the common length 0 is redirected toward the rule for X_1. We estimate the total such cost over the whole run of FCPM.

LEMMA 6.10. *For a single rule, the cost redirected from common lengths toward this rule during the whole run of* FCPM *is* $\mathcal{O}(\log M)$.

PROOF. First of all, we can consider $\mathcal{O}(\log(\min(c, M+1)))$ instead of $\mathcal{O}(1+\log(\min(c, M+1)))$: the 1 is added $\mathcal{O}(1)$ times per phase, and there are $\mathcal{O}(\log M)$ many phases. Second, the cost associated with the common length 0 is $\mathcal{O}(1)$ per phase, so $\mathcal{O}(\log M)$ in total. As said, we associate it with the rule for X_1.

Each other common length $c > 0$ (of some a block) is created inside a rule for some X_i; let this rule be $X_i \to uX_jvX_kw$ (by definition, two nonterminals in a rule are needed for a creation of a new common length). Then, $\mathrm{val}(X_j)$ right-popped an a-suffix and $\mathrm{val}(X_k)$ left-popped an a-prefix; moreover, $v \in a^*$.

First, consider the easier case of common lengths c, whose bit vector is of length greater than $\log(M+1)$. We spend $\mathcal{O}(\log M)$ time to find out that $c > M$. Let this common length be created in the rule for X_i. If during the creation of c a nonterminal is removed from the rule, then this is fine because this happens only once per nonterminal/rule and so the cost charged toward the rule in such a case is $\mathcal{O}(\log M)$. In the other case, this common length occurs between nonterminals in a rule for X_i—that is, between X_j and X_k. Afterward, between those nonterminals, there is a letter not occurring in p. Furthermore, compression cannot change it: in each consecutive phase, there will be such a letter between those nonterminals. So there can be no more creation of common lengths of letters occurring in strings between those two nonterminals. Therefore, the $\mathcal{O}(\log M)$ cost is charged to this rule only once in this way.

Finally, we consider the main case of creation of common lengths whose bit vector is of length at most $\log(M+1)$. If the creation of the common length c removes a nonterminal from the rule for X_i, then this happens at most twice for this rule and the associated cost is $\mathcal{O}(\log M)$.

Consider the case in which no nonterminal is removed from the rule during the creation of a new common length c of an *ablock. Consider all such creations of powers in a rule for* X_i. Let the consecutive letters, whose blocks were compressed, be $a^{(1)}$, $a^{(2)}, \ldots, a^{(\ell)}$ and the corresponding blocks' lengths $c_1, c_2, \ldots, c_\ell$ (the c_ℓ repetitions of $a^{(\ell)}$ are replaced by $a^{(\ell+1)}$). Observe that $a^{(i+1)}$ does not need to be $a_{c_i}^{(i)}$, as there might have been some other compression in between.

Recall the definition of *weight*: for a letter, it is the length of the represented string in the original instance. Consider the weight of the strings between X_j and X_k. Clearly, after the i-th blocks compression, it is exactly $c_i \cdot \mathrm{w}(a^{(i)})$. We show that

$$\mathrm{w}(a^{(i+1)}) \geq c_i\,\mathrm{w}(a^{(i)}). \tag{5}$$

Right after the i-th blocks compression, the string between X_j and X_k is simply $a_{c_i}^{(i)}$. After some operations, this string consists of c_{i+1} letters $a^{(i+1)}$. All operations in FCPM do not remove the symbols from the string between two nonterminals in a rule

(removing of leading a_R or ending a_L from t cannot affect letters between nonterminals). Recall that we can think of the recompression as building an SLP for p and t. In particular, one of the letters $a^{(i+1)}$ derives $a^{(i)}_{c_i}$ (which is the letter that replaced the block of c_i letters $a^{(i)}$). Since in the derivation the weight of the right- and left-hand sides are equal, it holds that

$$\mathrm{w}(a^{(i+1)}) \geq \mathrm{w}(a^{(i)}_{c_i}) = c_i \cdot \mathrm{w}(a^{(i)}).$$

Thus, $\mathrm{w}(a^{(\ell)}) \geq \prod_{i=1}^{\ell-1} c_i$. Recall that we consider only the cost of letters that occur in the pattern. Hence, $a^{(\ell)}$ (or some heavier letter) occurs in the pattern, so $M \geq \mathrm{w}(a^{(\ell)})$ (note that this argument does not apply to $a^{(\ell+1)}$, as it does not necessarily occur in p). Hence,

$$\log(M) \geq \log\left(\prod_{i=1}^{\ell-1} c_i\right) = \sum_{i=1}^{\ell-1} \log c_i.$$

What is missing is the cost of creation c_ℓ, which is at most $\log(M+1)$; the whole charge of $\sum_{i=1}^{\ell} \log c_i$ to the single rule is $\mathcal{O}(\log M)$. □

Summing over the rules gives the total cost of $\mathcal{O}((n+m)\log M)$, as claimed. This ends the proof of Theorem 1.1.

ACKNOWLEDGMENTS

I would like to thank Paweł Gawrychowski for introducing me to the topic, for pointing out the relevant literature [Alstrup et al. 2000; Lifshits 2007; Lohrey and Mathissen 2011; Mehlhorn et al. 1997], and discussions [Gawrychowski 2011a], and anonymous referees whose comments helped to improve the article.

REFERENCES

Stephen Alstrup, Gerth Stolting Brodal, and Theis Rauhe. 2000. Pattern matching in dynamic texts. In *Proceedings of SODA*. 819–828. DOI:http://dx.doi.org/10.1145/338219.338645

Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. 2005. The smallest grammar problem. *IEEE Transactions on Information Theory* 51, 7, 2554–2576. DOI:http://dx.doi.org/10.1109/TIT.2005.850116

Leszek Gąsieniec, Marek Karpiński, Wojciech Plandowski, and Wojciech Rytter. 1996a. Efficient algorithms for Lempel-Ziv encoding. In *Algorithm Theory—SWAT'96*. Lecture Notes in Computer Science, Vol. 1097. Springer, 392–403. DOI:http://dx.doi.org/10.1007/3-540-61422-2_148

Leszek Gąsieniec, Marek Karpiński, Wojciech Plandowski, and Wojciech Rytter. 1996b. Randomized efficient algorithms for compressed strings: The finger-print approach. In *Combinatorial Pattern Matching*. Lecture Notes in Computer Science, Vol. 1075. Springer, 39–49. DOI:http://dx.doi.org/10.1007/3-540-61258-0_3

Leszek Gąsieniec and Wojciech Rytter. 1999. Almost optimal fully LZW-compressed pattern matching. In *Proceedings of DCC*. IEEE, Los Alamitos, CA, 316–325.

Paweł Gawrychowski. 2011a. Personal communication.

Paweł Gawrychowski. 2011b. Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. In *Algorithms—ESA 2011*. Lecture Notes in Computer Science, Vol. 6942. Springer, 421–432. DOI:http://dx.doi.org/10.1007/978-3-642-23719-5_36

Paweł Gawrychowski. 2012a. Simple and efficient LZW-compressed multiple pattern matching. In *Combinatorial Pattern Matching*Lecture Notes in Computer Science, Vol. 7354. Springer, 232–242. DOI:http://dx.doi.org/10.1007/978-3-642-31265-6_19

Paweł Gawrychowski. 2012b. Tying up the loose ends in fully LZW-compressed pattern matching. In *Proceedings of STACS*. Leibniz International Proceedings in Informatics, Vol. 14. Schloss Dagstuhl, 624–635. DOI:http://dx.doi.org/10.4230/LIPIcs.STACS.2012.624

Paweł Gawrychowski. 2013. Optimal pattern matching in LZW compressed strings. *ACM Transactions on Algorithms* 9, 3, 25. DOI:http://dx.doi.org/10.1145/2483699.2483705

Masahiro Hirao, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. 2000. Fully compressed pattern matching algorithm for balanced straight-line programs. In *Proceedings of SPIRE*. 132–138.

Artur Jeż. 2014. Compressed membership for NFA (DFA) with compressed labels is in NP (P). *Theory of Computing Systems* 55, 4, 685–718.

Artur Jeż. 2013a. Approximation of grammar-based compression via recompression. In *Combinatorial Pattern Matching*. Lecture Notes in Computer Science, Vol. 7922. Springer, 165–176. DOI:http://dx.doi.org/10.1007/978-3-642-38905-4_17 full version at http://arxiv.org/abs/1301.5842.

Artur Jeż. 2013b. One-variable word equations in linear time. In *Automata, Languages, and Programming*. Lecture Notes in Computer Science, Vol. 7966. Springer, 324–335. DOI:http://dx.doi.org/10.1007/978-3-642-39212-2_30 full version accepted to *Algorithmica* DOI:http://dx.doi.org/10.1007/s00453-014-9931-3.

Artur Jeż. 2013c. Recompression: A simple and powerful technique for word equations. In *Proceedings of STACS*. Leibniz International Proceedings in Informatics, Vol. 20. Schloss Dagstuhl, 233–244. DOI:http://dx.doi.org/10.4230/LIPIcs.STACS.2013.233

Artur Jeż. 2014. Context unification is in PSPACE. In *Automata, Languages, and Programming*. Lecture Notes in Computer Science, Vol. 8573. Springer, 244–255. Available at http://arxiv.org/abs/1310.4367.

Artur Jeż and Markus Lohrey. 2014. Approximation of smallest linear tree grammar. In *Proceedings of STACS*. Leibniz International Proceedings in Informatics, Vol. 25. Schloss Dagstuhl, 445–457.

Juha Kärkkäinen, Pekka Mikkola, and Dominik Kempa. 2012. Grammar precompression speeds up Burrows-Wheeler compression. In *String Processing and Information Retrieval*. Lecture Notes in Computer Science, Vol. 7608. Springer, 330–335. DOI:http://dx.doi.org/10.1007/978-3-642-34109-0_34

Marek Karpiński, Wojciech Rytter, and Ayumi Shinohara. 1995. Pattern-matching for strings with short descriptions. In *Combinatorial Pattern Matching*. Lecture Notes in Computer Science, Vol. 937. Springer, 205–214. DOI:http://dx.doi.org/10.1007/3-540-60044-2_44

N. Jesper Larsson and Alistair Moffat. 1999. Offline dictionary-based compression. In *Proceedings of the Data Compression Conference*. IEEE, Los Alamitos, CA, 296–305. DOI:http://dx.doi.org/10.1109/DCC.1999.755679

Yury Lifshits. 2007. Processing compressed texts: A tractability border. In *Combinatorial Pattern Matching*. Lecture Notes in Computer Science, Vol. 4580. Springer, 228–240. DOI:http://dx.doi.org/10.1007/978-3-540-73437-6_24

Markus Lohrey and Christian Mathissen. 2011. Compressed membership in automata with compressed labels. In *Computer Science—Theory and Applications*. Lecture Notes in Computer Science, Vol. 6651. Springer, 275–288. DOI:http://dx.doi.org/10.1007/978-3-642-20712-9_21

Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. 1997. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica* 17, 2, 183–198. DOI:http://dx.doi.org/10.1007/BF02522825

Michael Mitzenmacher and Eli Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press.

Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda. 1997. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Combinatorial Pattern Matching*. Lecture Notes in Computer Science, Vol. 1264. Springer, 1–11. DOI:http://dx.doi.org/10.1007/3-540-63220-4_45

Craig G. Nevill-Manning and Ian H. Witten. 1997. Identifying hierarchical strcture in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research* 7, 67–82. DOI:http://dx.doi.org/10.1613/jair.374

Wojciech Plandowski. 1994. Testing equivalence of morphisms on context-free languages. In *Algorithms—ESA '94*. Lecture Notes in Computer Science, Vol. 855. Springer, 460–470. DOI:http://dx.doi.org/10.1007/BFb0049431

Wojciech Plandowski and Wojciech Rytter. 1999. Complexity of language recognition problems for compressed words. In *Jewels Are Forever*, Juhani Karhumäki, Hermann A. Maurer, Gheorghe Paun, and Grzegorz Rozenberg (Eds.). Springer, 262–272.

Wojciech Rytter. 2003. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science* 302, 1-3, 211–222. DOI:http://dx.doi.org/10.1016/S0304-3975(02)00777-6

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

Received March 2013; revised March 2014; accepted June 2014